JOHANNES GUTENBERG UNIVERSITÄT MAINZ
Department of Computer Science

Exploiting File Caching Infrastructures in HPC Using Guided I/O Interfaces

Candidate:
Giuseppe Congiu
Version 1.0, November 17, 2016

Advisors:
Prof. Dr. André Brinkmann
Dr. Sai Narasimhamurthy

Contents

Chapter 1

Introduction

The gap between hard disk drives' (HDDs) performance and processors' computing power, better known as the I/O performance gap problem, represents a serious scalability limitation for scientific applications running on High End Computing (HEC) clusters. To address this problem, HEC clusters system software is built with multiple layers stacked on top of each other. These layers were added incrementally over the years to provide new features/optimizations, and to solve existing problems. Unfortunately, since the corresponding software components were not designed and built in an integrated manner, they often duplicate existing functionalities and introduce additional parameters that make the optimal configuration of the system a very difficult task. Figure 1.1 shows a typical layering for the High Performance Computing (HPC) I/O software stack.

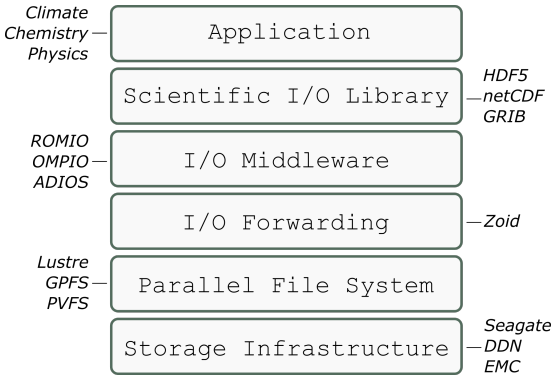


Figure 1.1: HPC I/O Stack

At the bottom of the stack in Figure 1.1 is the *Storage Infrastructure* represented by vendor specific hardware/software solutions that aggregate and export data blocks from remote HDDs to the clients using a low latency Storage Area Network (SAN). The *Parallel File System* software, on top of the Storage Infrastructure, consolidates these blocks into a more usable and familiar file interface. Parallel File Systems (PFSs) such as Lustre [22], PVFS [25] and GPFS [45], just to mention a few, exploit the storage devices available on the

network by striping files across them and providing clients with parallel paths to reach data, thus increasing the aggregated I/O bandwidth. Frequently storage vendors also provide the parallel file system software optimized for their hardware configuration (e.g. GPFS for IBM and custom Lustre for Seagate). The *I/O Forwarding* layer is optionally used in some supercomputer designs, such as the IBM Blue Gene systems, to delegate file system access to a limited number of intermediate I/O clients, with the benefit of reducing the Operating System (OS) noise in the compute nodes [21]. The *I/O Middleware* provides parallel I/O towards the underlying file system, typically using the MPI-IO interface to achieve high performance data access as well as consistency. ROMIO[1] is an example of popular I/O middleware used in HPC. It implements the MPI-IO extensions to the POSIX I/O interface, and through the Abstract Device I/O (ADIO) drivers [48] enables transparent file access optimizations based on two-phase I/O and data sieving, adapting I/O patterns to the characteristics of the underlying file system [49]. However, MPI-IO does not provide applications with the infrastructure they need to organize and manage their data in a more abstract and efficient manner. Thus, *Scientific I/O Libraries* have been introduced at the top of the stack to bridge the gap between how data is represented and organised at the application level (data model) and how it is mapped onto the underlying file system (storage model). Popular I/O libraries include the Hierarchical Data Format library (HDF5) [10] and the network Common Data Format (netCDF) [15].

In this thesis we focus on the I/O middleware layer of the I/O software stack presented in Figure 1.1. The I/O middleware represents a key component for achieving high performance in HPC since it connects the parallel file system to the high-level I/O libraries (most commonly HDF5 or netCDF). For this reason the I/O middleware is the best place to implement automatic and semi-automatic optimisations that use guided I/O interfaces as mechanism to deliver the requested optimisation messages.

1.1 Guided I/O Interfaces

Generally guided I/O interfaces come in the form of extensions to the standard API of the main software component. Users, as well as other software packages, can exploit these extensions to control the internal behaviour of the component, altering the way the standard API works. Guided I/O APIs are present at different levels in the I/O software stack but are most commonly found in the I/O middleware and the file system components.

1.1.1 Guided I/O in MPI-IO

The ROMIO middleware provides guided I/O support through the MPI-IO hints API, specifically designed to control internal ROMIO parameters and trigger different I/O optimisations. MPI-IO hints are packed into a special info object and passed down to the middleware through the MPI file open API. The MPI-IO hints mechanism allows the middleware to convert I/O patterns

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

generated by applications into I/O patterns that are compatible with the characteristics of the underlying file system, thus improving I/O performance. In this way the same MPI file read and write operations can behave differently depending on the hints passed to the middleware when the file was opened. Specifically, if the collective I/O hint is enabled ROMIO will convert every non-contiguous independent write to the file into a smaller number of sequential coordinated writes.

Besides I/O pattern selection, MPI-IO hints can be also used to control the size of the buffers used by ROMIO when performing I/O, to select the stripe size used by the file system to store data across the available I/O servers and even to select the number of I/O servers used to store the file.

1.1.2 Guided I/O in Linux File Systems

At the file system level the Linux kernel provides users with the capability to communicate access pattern information to the local file system through the `posix_fadvise()` [17] 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.

The two most used PFSs in HEC clusters nowadays, GPFS and Lustre, are both POSIX compliant. However, neither of them supports 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()` [9] function and can be used to guide prefetching (or releasing) of file blocks in the page pool[2]. Lustre, on the other hand, does not provide any client side mechanism similar to GPFS hints or POSIX advice.

1.2 Opportunities with Non-Volatile Memories

The availability of Solid State Drives (SSDs) and, more generally, Non-Volatile Memory (NVM) devices in the storage hierarchy provides new opportunities for optimisations based on data placement and migration across the different memory tiers of the system. Although HPC compute nodes have access to locally attached SSDs, these are rarely integrated in the I/O software stack. A possible way of integrating the new memory devices in the stack is given by guided I/O interfaces in the I/O middleware component. By using the guided I/O API in the middleware users can control data placement and migration to improve the efficiency of specific I/O patterns.

One practical use case for NVM devices is file caching. File caching can be useful in many different scenarios. Generally, caching can be used to reduce the I/O latency as well as the number of network requests generated by file system clients. More specifically, since the number of NVM devices can scale linearly with the number of compute nodes in the cluster, it can also improve the aggregated I/O bandwidth. Additionally, caching can be useful to improve collective I/O performance. Collective I/O is a parallel I/O technique in

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

which write performance is highly dependent upon the storage system response time and limited by the slowest writer. The storage system response time in conjunction with the need for global synchronisation, required during every round of data exchange and write, severely impacts collective I/O performance. Future Exascale systems will have an increasing number of processor cores, while the number of storage servers will remain relatively small. Therefore, the storage system concurrency level will further increase, worsening the global synchronisation problem.

NVM file caching can alleviate the global synchronisation problem in collective I/O by minimising the I/O time variation across compute nodes and can increase the I/O bandwidth by aggregating together multiple NVM devices on different compute nodes.

1.3 Contribution

In the described context our work provides two main contributions. First, we bring the POSIX Advice and GPFS hints APIs together in a single middleware component called Mercury. Mercury exports a unique API and can transparently select the appropriate underlying hints interface depending on the file system backend used to store the data. Additionally, we modify the Virtual File System (VFS) layer in the Linux kernel to enable the `posix_fadvise()` system call for Lustre and more generally any network file system. Second, we integrate new NVM devices inside the ROMIO middleware by extending the MPI-IO hints API. Locally attached SSDs are thus used to boost collective I/O performance in HPC workloads.

1.4 Thesis Reminder

The reminder of this thesis is organised as follow:

Chapter 2

Background on Guided I/O Intefaces in HPC

Chapter 3

The Mercury Middleware

3.1 Motivation

The gap between hard disk drives' (HDDs) performance and processors' computing power, better known as the 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 [22], PVFS [25] and GPFS [45], 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 I/O Operations per Second (IOPS). The ROMIO middleware[1] implements extensions to the POSIX I/O interface typically provided by PFSs that result in a richer parallel I/O interface, and through the Abstract Device I/O (ADIO) driver [48] 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 [49] [52] [44].

Nevertheless, as Carns et al. have pointed out in their study [24] most of the scientific applications running on big clusters today still use the POSIX I/O interface to access their data. Furthermore, it has also been ascertained that using POSIX I/O to access non-contiguous regions of the file causes extremely poor performance in the case of PFSs [30]. Indeed, PFSs provide best I/O bandwidth performance for large contiguous requests while they typically provide only a fraction of the maximum bandwidth in the opposite case. This is primarily due to the high number of remote procedure calls generated by the file system clients that overwhelms I/O servers, the resulting high number of HDDs' head movements in every I/O target (seek overhead) and ultimately by the file system block locking contention.

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()` [17] system call. The file system can use this information to

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

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. However, 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()` [9] 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 [12]. The DDN approach provides control over the storage servers (OSSs) cache instead of the file system client cache.

In this thesis we propose and evaluate a novel guided I/O framework called *Mercury* [13] able to optimize file access patterns at run-time through data prefetching using available hints 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 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 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] [18] 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 chapter is organised as follows. Section 3.2 covers the background on file systems support to guided I/O interfaces (POSIX advice and GPFS hints), Section 3.3 presents concept, design and implementation of the

[2]GPFS pinned memory used for file system caching.
[3]Data analysis framework developed at CERN.

MERCURY prototype highlighting the main contributions of the work. This section also describes the kernel modifications that enable POSIX advice on Lustre, Section 3.4 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 3.5 presents related works on data prefetching, and finally Section 3.6 presents conclusion and future work.

3.2 Background on File System I/O Hints

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 4.3.

3.2.1 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.

Table 3.1: Values for *advice* in the *posix_fadvise()* system call

Advice	Description
`POSIX_FADV_SEQUENTIAL`	file I/O pattern is sequential
`POSIX_FADV_RANDOM`	file I/O pattern is random
`POSIX_FADV_NORMAL`	reset file I/O pattern to normal
`POSIX_FADV_WILLNEED`	file range will be needed
`POSIX_FADV_DONTNEED`	file range won't be needed
`POSIX_FADV_NOREUSE`	file is read once (not implemented)

The first two advice in Table 3.1 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'.

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.

3.2.2 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 4.2 summarizes all the available hints data structures and reports the corresponding description for each of them.

Table 3.2: GPFS hint data structures

Hint data structure	Description
`gpfsAccessRange_t`	defines a file range to be accessed
`gpfsFreeRange_t`	defines a file range to be released
`gpfsMultipleAccessRange_t`	defines multiple file ranges to be accessed
`gpfsClearFileCache_t`	releases all the page pool buffers for a certain file

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 4.2). The data structure corresponding to this hint is reported in Listing **??**.

```
1 #define GPFS_MAX_RANGE_COUNT 8
```

```
2
3  typedef struct
4  {
5      int structLen;
6      int structType;
7      int accRangeCnt;
8      int relRangeCnt;
9      gpfsRangeArray_t accRangeArray[GPFS_MAX_RANGE_COUNT];
10     gpfsRangeArray_t relRangeArray[GPFS_MAX_RANGE_COUNT];
11
12 } gpfsMultipleAccessRange_t;
```

Listing 3.1: 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.

3.3 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 3.1. 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 [4] during a set of preliminary runs and then written to a configuration file to be used in following runs.

[4] How this can be done effectivelly and in a generalized way is itself a research topic and is therefore left as part of future works.

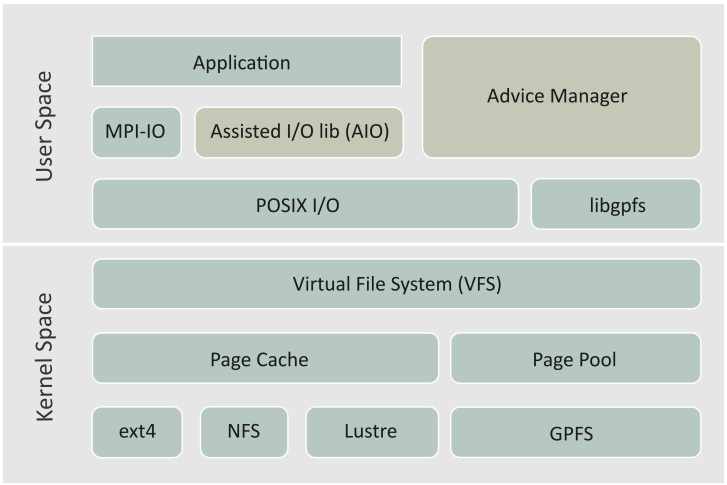


Figure 3.1: MERCURY I/O software stack. *Assisted I/O library* and *Advice Manager* communicate through UNIX domain sockets. The AM binds its socket to the local file system pathname `/tmp/channel`, while the AIO connects its socket to the same pathname; exactly in the same way they would bind and connect to an IP address if they were located on different nodes in the network. Unix domain sockets are used to pass ancillary data as well as custom messages between the two software entities. Data can reside in a local Linux file system, in Lustre or in GPFS.

In the rest of this section we describe 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.

3.3.1 Interprocess Communication

We now describe how interprocess communication is implemented and how messages sent from the *Assisted I/O library* are handled by the *Advice Manager*. Figure 3.2 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, performs the open and then 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.

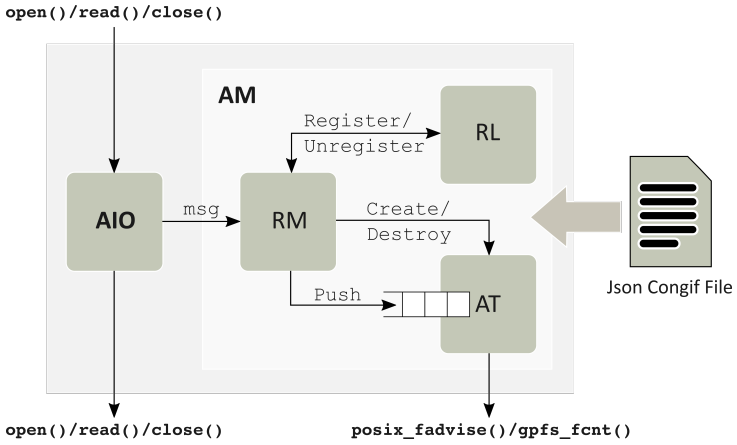


Figure 3.2: Detailed architecture for the *Advice Manager* (AM) component. This can be further divided into three blocks: *Request Manager* (RM), *Register Log* (RL), and *Advisor Thread* (AT).

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 (or control) information. One use of such information is to send a remote process a 'file descriptor' [46] via a UNIX domain socket [20]. 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.

3.3.2 File Data Prefetching

POSIX advice and GPFS hints are issued using the *Advisor Thread* created by the *Request Manager* during the register operation (Figure 3.2). 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 3.2). For files residing in local file systems and Lustre, the `POSIX_FADV_WILLNEED` advice from Table 3.1 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 **??** 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[5] 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 3.2: Example of Json Configuration File

As it can be seen in Listing **??** 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

[5]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).

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 **??** is not exhaustive. More complex configuration files can be generated by administrators (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 an 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 3.1 and the hints listed in Table 4.2. 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 3.2.1, 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.

3.3.3 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 3.3. 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.

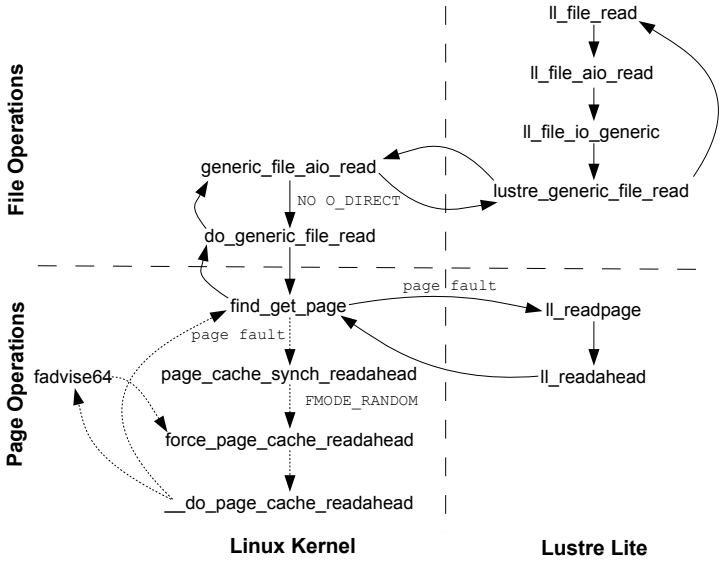


Figure 3.3: Simplified function call graph for the read operation in Lustre. For page operations in the Linux kernel the picture also shows the call graph typically followed by local reads as well as the call graph for the `POSIX_FADV_WILLNEED` advice in the `posix_fadvise()` implementation (dashed line).

In order to enable `POSIX_FADV_WILLNEED` in Lustre we modified the call graph of `fadvise64()` presented in Figure 3.3 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.

3.4 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.

3.4.1 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.

3.4.2 Real World Application

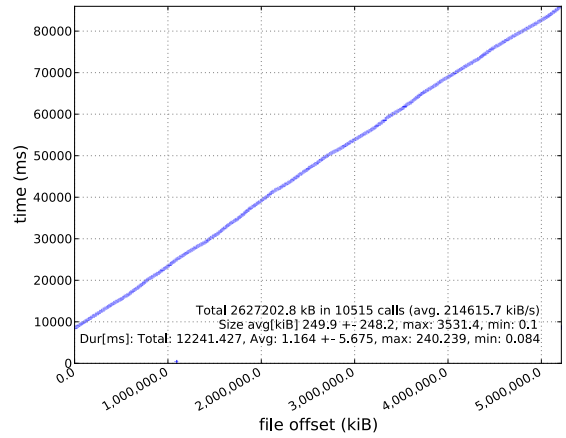
Our target real world application is written using 'ROOT', an object-oriented framework widely adopted in the experimental high energy physics community to build software for data analysis. The application analyzes data read from an input file in the 'ROOT' format (structured file format).

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* [11] and GPFS's *mmpmon* [8] monitoring tool. Figure 3.4 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.

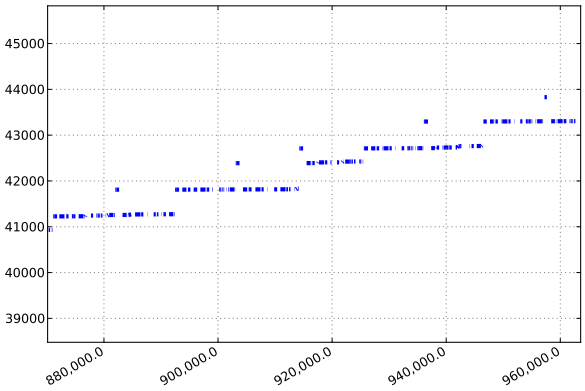
At a first glance the general I/O behaviour of the application looks linear, most of the accesses to the file follow an increasing offset. Nevertheless, adjacent reads are separated by gaps (a strided read pattern). In a few cases this gap becomes negative, meaning that the application is moving backwards in the file to read some data previously left behind (as reported in Figure 3.4b).

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[6], 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

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



(a)



(b)

Figure 3.4: I/O read profile of the target application under analysis (3.4a), extracted from the the GPFS file system in the test cluster, and zoomed window (3.4b) showing the actual pattern details.

by Figure 3.5. 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.

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

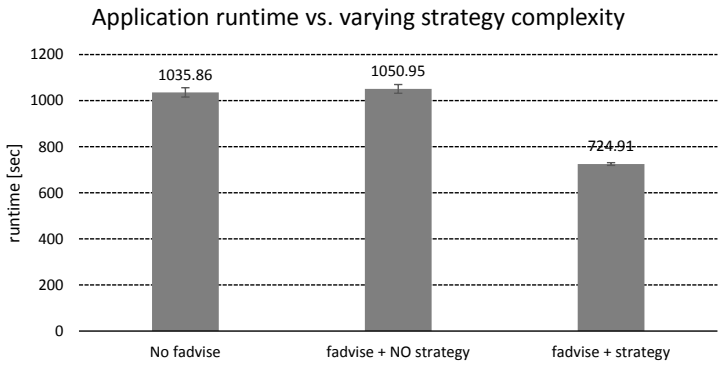


Figure 3.5: Comparison between different usage stategies of posix_fadvise for an input file of 55GB residing in an ext4 file system. The first bar represents the case in which no advice is used, the second bar represents the case in which a POSIX_FADV_WILLNEED is issued for the whole file at the beginning of the application and the third bar represents the case in which POSIX_FADV_-WILLNEED is issued using MERCURY.

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 4.2, the NSD servers do not cache any data.

3.4.3 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 3.6 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 512MB 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 3.6a and 3.6c. 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 3.6b). 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 observed reductions in the execution time comparable to the other file systems (not reported here).

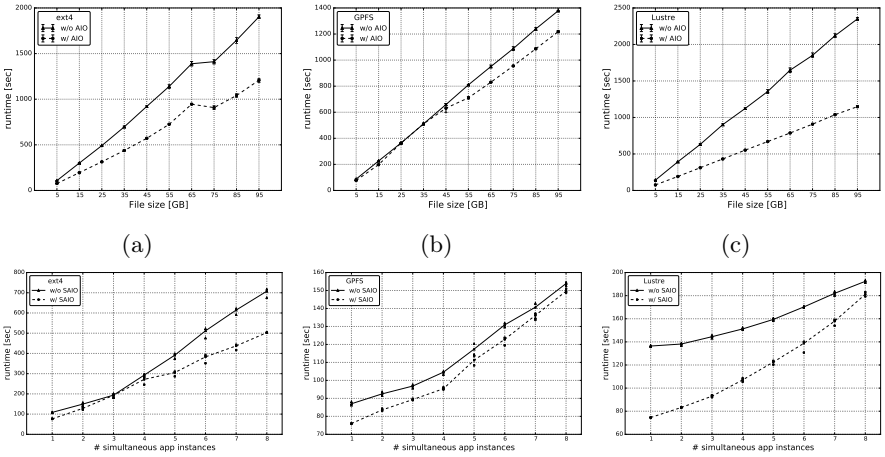


Figure 3.6: Running time of the ROOT application for the three file system under study using different input file sizes (3.6a, 3.6b and 3.6c) and different number of instances accessing a file of 5GB (3.6d, 3.6e and 3.6f).

As far as Figures 3.6d, 3.6e and 3.6f 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 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

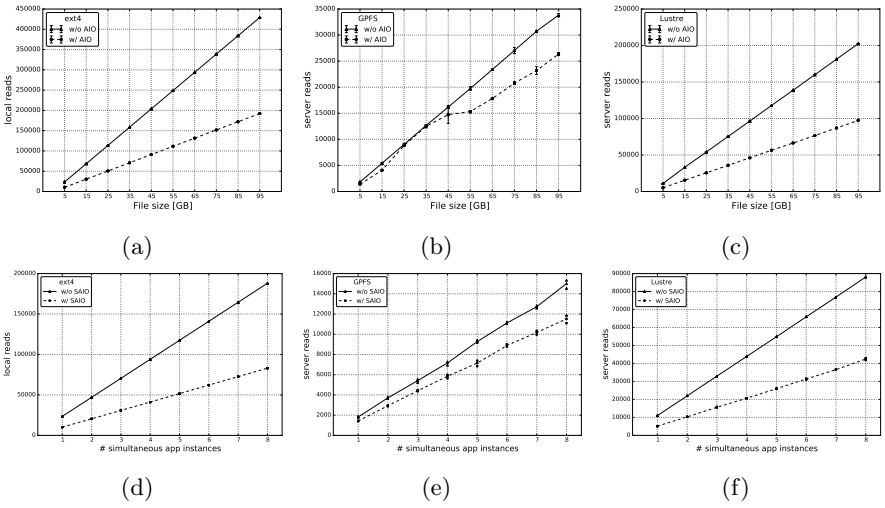


Figure 3.7: Reads processed by local ext4, GPFS and Lustre I/O servers for various input file sizes (3.7a, 3.7b and 3.7c) and multiple instances of ROOT accessing a file of 5GB (3.7d, 3.7e and 3.7f).

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.

3.4.4 Read Request Rate

Figure 3.7a, 3.7b and 3.7c 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 3.7d, 3.7e and 3.7f).

3.5 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 [50] 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. [33] 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 [26], 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 [27] 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 [23] 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 [28] 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 [35] 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 [51] for programmers to use. Prost et al. integrated the GPFS hint functionalities in the ROMIO ADIO driver for GPFS [43]. 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.

3.6 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.

In the future we plan to further explore the problem of data science analytics applications in HPC, studying more profusely the different types of existing I/O patterns and how hints can be used to improve the cache utilization efficiency and thus applications' performance. Moreover, although not explicitly treated here, we recognise how important is to automatically extract information concerning the application's I/O pattern profile. Almost all the current analysis tools, including the ones we have used, approach the I/O pattern analysis problem from an ultra fine-grain point of view (i.e. considering the offset and length requests) with the result of having a very specific file per application characterization. On the other hand, in this work we have observed that a course-grain approach (i.e. considering blocks instead of single requests) can be more beneficial when trying to extract the general application behaviour. The study of an automatic I/O pattern analysis tool with the described features will be thus part of future work.

Chapter 4

NVM Caching in MPI-IO

4.1 Motivation

High Performance Computing (HPC) applications process large amounts of data that have to be written (read) to (from) large shared files residing in the global parallel file system. In order to make large datasets manageable, these are usually partitioned into smaller sub-sets and assigned to available cores for parallel processing. Complex datasets such as N-dimensional arrays are logically flattened into a linear sequence of bytes and striped across several I/O targets for best performance. This results in the loss of the original spatial locality, as shown in Figure 4.1. Due to this characteristic, accesses to spatially contiguous regions translate into non-contiguous accesses to the file. Therefore, applications generating a large number of small, non-contiguous I/O requests to the parallel file system usually experience degradation of I/O performance. Such performance degradation is known as the 'small I/O problem' and is related to the fact that parallel file systems provide best I/O bandwidth performance for large contiguous requests, while they typically provide only a fraction of the maximum available bandwidth in the opposite case [30] [34]. This is due to the large number of Remote Procedure Calls (RPCs) generated by the file system clients overwhelming I/O servers, the resulting high number of hard disk head movements in every I/O target (seek overhead) and, ultimately, to the restrictions imposed by the POSIX I/O write semantic that generates lock contention in the file systems' blocks.

Having recognised the small I/O problem, collective I/O was proposed by the MPI-IO community [49]. Collective I/O exploits global I/O knowledge in parallel I/O to a shared file. This knowledge is used to build an aggregated view of the accessed region in the file and coalesce all the corresponding small non-contiguous requests into a smaller number of large contiguous accesses, later issued to the parallel file system. File system accesses are orchestrated in such a way that only a subset of the available processes actually performs I/O. These processes are called 'aggregators', because they gather and aggregate all the requests on behalf of the other processes, whose only role in this case is to send (receive) data to (from) aggregators.

In conclusion, collective I/O can convert many small random accesses into large sequential accesses to the I/O subsystem, reducing the actual number

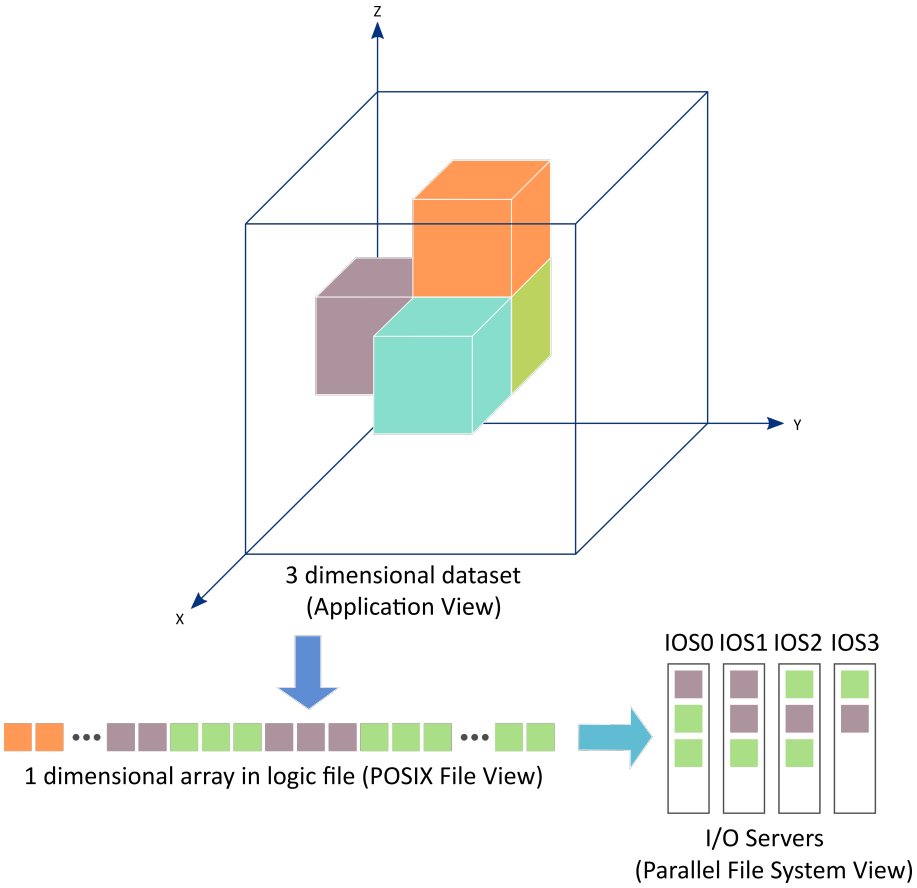


Figure 4.1: Example of how a 3-Dimensional dataset is represented logically in the file as a sequence of bytes and how it is finally translated in the parallel file system.

of I/O requests that need to be accounted for by the I/O stack. Even when processes generate large I/O requests, it might be still beneficial to coordinate them to reduce parallel file system block locking contention as well as concurrency level on data servers. This mechanism effectively adapts the I/O pattern to the characteristics of the file system, extracting maximum performance from it. Figure 4.2 exemplifies the basic collective I/O mechanism just described. In the figure there are six processes, four of which play the role of aggregators. Collective I/O proceeds in two phases: 'data shuffling' and 'data I/O'. Data shuffling takes place between all the processes and aggregators and is aimed to build the logically contiguous file regions (or file domains) that will be later accessed during the data I/O phase.

Two phase I/O has a number of problems that limit its scalability. The main issues are: (**a**) global synchronisation overhead between processes exchanging data, where I/O is bottlenecked by the slowest aggregator process [53], (**b**) aggregators' contention on file system stripes (primarily due to stripe boundary misalignment of the file domains and the underlying file system locking mecha-

nism[1]) [38], (**c**) aggregators' contention on I/O servers (related to inefficient file domain partitioning strategy), (**d**) pressure that large collective buffers can have on the memory system (due to scarce amount of available memory per single core in current and future large scale HPC clusters), and (**e**) high memory bandwidth requirements due to the large number of data exchanges between processes and aggregators [41]. There is also a mismatch between logical file layouts and the collective I/O mechanism, which does not seem to take the physical data layouts into consideration [29] [54].

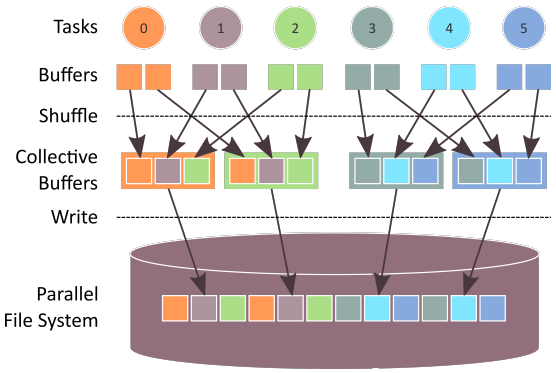


Figure 4.2: Collective I/O write example. A data set is partitioned and assigned to six processes. Four of them work as aggregators writing data to the global file system.

Nowadays solid state drives (SSDs) are becoming more affordable and many HPC systems provide SSDs attached to compute nodes. Despite the fact that these devices formally provide an additional tier in the storage hierarchy, they are mostly used to store local data and rarely exploited for parallel I/O. As part of the Exascale10 (E10) [5] initiative in the DEEP-ER [3] project, we explore the benefits that the usage of such fast devices can bring when writing collectively to a shared file.

We integrate local storage resources attached to compute nodes (non-volatile memory devices) as persistent cache layer in ROMIO, providing all the additional infrastructure required to asynchronously flush the content of the cached data to the global file system, while keeping the MPI-IO consistency semantics. The new persistent cache layer is fully integrated inside ROMIO and can be easily accessed through a set of newly defined MPI-IO hints.

We show how when using local SSDs, if the number of aggregators is selected properly, significantly better I/O performance can be achieved compared to the case in which only the global file system is used. Contrarily, if the number of aggregators is not properly selected, I/O performance can even degrade. The additional cache layer also delivers more stable response times, reducing the impact of global synchronization on collective I/O, thus addressing point (**a**) previously discussed, and can reduce the memory pressure in systems with scarce amounts of main memory per core, thus addressing point (**d**).

[1]In current ROMIO versions the ADIO driver for Lustre can detect and align file domains to stripe boundaries thus avoiding stripe collisions. A new ADIO driver for BeeGFS supporting the same functionality has been developed in the course of this work.

The remainder of this chapter is organised as follows. In Section 4.2 we describe in detail the collective I/O implementation in ROMIO highlighting all the possible bottlenecks, in Section 4.3 we present the concept and design of the caching layer and the new hints extensions supporting it, along with all the corresponding semantics implications, in Section **??** we show the benefits of collective I/O using different benchmarks, in Section 4.5 we present collective I/O related works and finally in Section **??** we draw conclusions and discuss future developments.

4.2 Collective I/O in ROMIO

ROMIO is a popular implementation of the MPI-IO specification developed at the Argonne National Laboratory and currently supported by MPICH as well as OpenMPI and other packages. ROMIO provides parallel I/O functionalities for different file systems through the Abstract Device I/O interface [48] (ADIO). Latest versions of ROMIO include support for Lustre [52], GPFS [44], PVFS [25] and others through a dedicated ADIO driver.

4.2.1 Two Phase I/O

In ROMIO, the core component of collective I/O is the 'two phase I/O', also known as 'extended two phase algorithm' (ext2ph) [47]. The ROMIO implementation for collective I/O consists of several steps as follows:

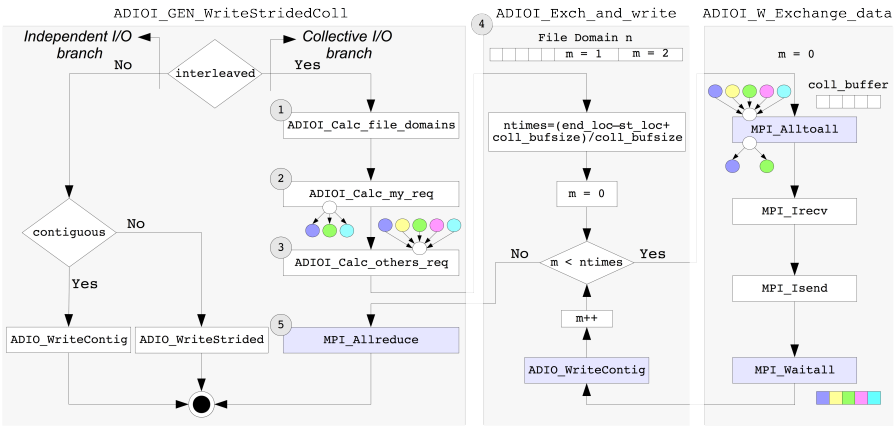


Figure 4.3: Collective I/O flow diagram for the write path in aggregators (non-aggregators neither receive nor write any data, just send it to aggregators). `MPI_File_write_all()` invokes `ADIOI_GEN_WriteStridedColl()`. `ADIO_WriteContig` is a macro that is replaced by `ADIOI_GEN_WriteContig()`. Performance critical functions for the collective I/O branch are highlighted in grey.

1. All processes taking part in the I/O operation exchange access pattern information with each other. The access pattern information is represented by start and end offsets for the accessed region (disregarding holes that

may be present). Once file offsets are available, every process works out how big the global accessed region in the file is by taking maximum and minimum among all. The resulting byte range is divided by the number of available aggregators to build the so called 'file domains' (contiguous byte ranges accessed independently by every aggregator).

2. Every process works out which file domains (and thus aggregators) its local data belongs to. In doing so, every process knows which aggregators it has to send (receive) data to (from), if any.

3. Every aggregator works out which other processes' requests map to its file domain. Doing so every aggregator knows what processes need to receive (in case of reads) or send (in case of writes) data for that particular file domain.

4. Actual two phase I/O starts. In the case of writes, that we exclusively consider here (the read case is similar), every process sends its data to the right aggregators (data shuffle phase) while these write the data to the parallel file system (data I/O phase). Data is written in blocks of predefined size (collective buffer size). If the size of the collective buffer is smaller than the file domain, the file domain is broken down into multiple sub-domains which are written in different rounds of the ext2ph algorithm. In order to handle multiple rounds of data shuffle and I/O, additional access information is required. This is disseminated by every process (collectively) to aggregators at the beginning of the data shuffle phase.

5. Once all the data has been written, all the processes must synchronise and exchange error codes. This is necessary to guarantee that it is safe to free the memory buffers containing the data.

Figure 4.3 shows how the previous steps map to the collective I/O implementation for the write operation. The collective write function (`MPI_File_write_all()`) in ADIO is implemented through `ADIOI_GEN_WriteStridedColl()`. This is responsible for selecting the most suitable I/O method between those available. For example, independent I/O is selected if the access requests are not interleaved. Nevertheless, users can always enforce collective I/O by setting the appropriate MPI-IO hint. The `ADIOI_Exch_and_write()` function contains the ext2ph algorithm implementation, including data shuffle and write methods. At the beginning of the data shuffle (`ADIOI_W_Exchange_data()`) we have the dissemination function (`MPI_Alltoall()`) used to exchange information concerning which part of the data has to be sent during a particular round of two phase I/O.

There are three main contributors to collective I/O performance: (**a**) global synchronisation cost; (**b**) communication cost; and (**c**) write cost. `MPI_Allreduce()` and `MPI_Alltoall()` account for the global synchronisation cost. When a process reaches them it has to wait for all the other processes to arrive before continuing. `MPI_Waitall()` accounts for communication cost since every process first issues all the non-blocking receives (if any) and sends, and afterwards waits for them to complete (refer to the right part of the diagram in Figure 4.3). Finally, `ADIO_WriteContig()` accounts for write cost.

4.2.2 Collective I/O Hints

Collective I/O behaviour can be controlled by users through a set of MPI-IO hints. Users can control whether collective I/O should be enabled or disabled with `romio_cb_write` and `romio_cb_read`, for write and read operations respectively, how many aggregators should be used during a collective I/O operation with `cb_nodes` and how big the collective buffer should be with `cb_buffer_size`. Table 4.1 summarises the hints just described.

Table 4.1: Collective I/O hints in ROMIO.

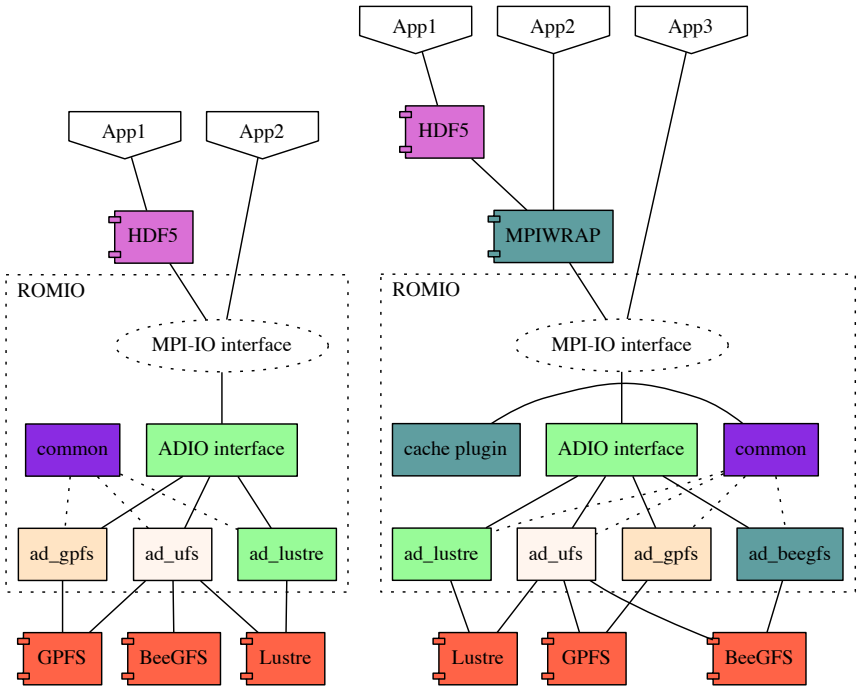
Hint	Description
`romio_cb_write`	`enable` or `disable` collective writes
`romio_cb_read`	`enable` or `disable` collective reads
`cb_buffer_size`	set the collective buffer size [bytes]
`cb_nodes`	set the number of aggregator processes

Each of these hints has an effect on collective I/O performance. For example, by increasing the number of aggregators there will be a higher number of nodes writing to the parallel file system and thus a higher chance that one of these will experience variable performance due to load imbalance among available I/O servers, with increasing write time variation and associated global synchronisation cost. Furthermore, by increasing the collective buffer size users can reduce the number of two phase I/O rounds and, consequently, the number of global synchronisation events. Bigger collective buffers will also affect the write cost since more I/O servers will be accessed in parallel potentially increasing the aggregated I/O bandwidth.

Besides the hints described in Table 4.1, there are other hints that do not directly concern collective I/O but affects its performance. The first is the `striping_factor` hint, which defines how many I/O targets will be used to store the file. The second is the `striping_unit` hint, which defines how big the data chunks written to each I/O target will be (in bytes). These two hints change the file characteristics in the parallel file system and typically the 'striping_unit' also defines the locking granularity for the file (e.g. Lustre).

4.3 Concept & Design

In this section we present the high-level architecture of the current ROMIO implementation as well as our proposed architecture. The two are shown in Figure 4.4a and 4.4b respectively. The ROMIO middleware is designed to be modular and easily extensible. Support for different parallel file systems is provided through additional software modules called drivers. The appropriate driver is selected at file open time through an interface called Abstract Device I/O (ADIO), following an approach similar to the Virtual File System layer of the Linux Kernel.



(a) ROMIO architecture (b) Proposed ROMIO architecture

In Figure 4.4a there are three different file system drivers: *ad_gpfs* for GPFS support, *ad_ufs* for Universal File System (UFS) support, and *ad_lustre* for Lustre support. These drivers share features implemented in the *common* module. The common module contains the implementation for most of the I/O operations used by the UFS driver (ad_ufs) and other drivers. File system drivers can implement their own I/O operations or use the ones made available by the common module. Lustre, for example, uses the common collective open operation (`ADIOI_GEN_Opencoll()`) but implements its own collective write operation (`ADIOI_LUSTRE_WriteStridedColl()`) described in Figure 4.3. Specific implementations are selected using a file operation table that has to be defined by every file system driver. Listing 4.1 shows the operation table for the ad_lustre driver.

```
1  struct ADIOI_Fns_struct ADIO_LUSTRE_operations = {
2      ADIOI_LUSTRE_Open, /* Open */
3      ADIOI_GEN_OpenColl, /* OpenColl */
4      ADIOI_LUSTRE_ReadContig, /* ReadContig */
5      ADIOI_LUSTRE_WriteContig, /* WriteContig */
6      ADIOI_GEN_ReadStridedColl, /* ReadStridedColl */
7      ADIOI_LUSTRE_WriteStridedColl, /* WriteStridedColl */
8      ADIOI_GEN_SeekIndividual, /* SeekIndividual */
9      ADIOI_GEN_Fcntl, /* Fcntl */
10     ADIOI_LUSTRE_SetInfo, /* SetInfo */
11     ADIOI_GEN_ReadStrided, /* ReadStrided */
12     ADIOI_LUSTRE_WriteStrided, /* WriteStrided */
13     ADIOI_GEN_Close, /* Close */
```

```
#if defined(ROMIO_HAVE_WORKING_AIO) && !defined(
    CRAY_XT_LUSTRE)
    ADIOI_GEN_IreadContig, /* IreadContig */
    ADIOI_GEN_IwriteContig, /* IwriteContig */
#else
    ADIOI_FAKE_IreadContig, /* IreadContig */
    ADIOI_FAKE_IwriteContig, /* IwriteContig */
#endif
    ADIOI_GEN_IODone, /* ReadDone */
    ADIOI_GEN_IODone, /* WriteDone */
    ADIOI_GEN_IOComplete, /* ReadComplete */
    ADIOI_GEN_IOComplete, /* WriteComplete */
    ADIOI_GEN_IreadStrided, /* IreadStrided */
    ADIOI_GEN_IwriteStrided, /* IwriteStrided */
    ADIOI_GEN_Flush, /* Flush */
    ADIOI_GEN_Resize, /* Resize */
    ADIOI_GEN_Delete, /* Delete */
    ADIOI_GEN_Feature, /* Features */
    "LUSTRE:",
};
```

Listing 4.1: Operation Table for Lustre Driver

In Figure 4.4b we extend the presented ROMIO architecture with two additional modules: a dedicated driver supporting the BeeGFS file system (*ad_ beegfs*) and a *cache plugin* that links directly to the common module, thus providing NVM caching features to all the underlying file system drivers. We also developed an external library called *MPIWRAP* that is used to allow transparent integration of the new caching functionalities into existing applications without any need of modifying them.

In the rest of the chapter we describe in details the new MPI-IO hints supporting local NVM caching and the corresponding cache plugin module implementation.

4.3.1 MPI-IO Hints Extensions

To take advantage of attached non-volatile memories in computing nodes we introduced a new set of MPI-IO hints, reported in Table 4.2, and a corresponding set of modifications in the ROMIO implementation of the UFS driver supporting them.

The new hints are used to control the data path in the storage system as well as to define a basic set of cache policies for synchronisation and space management. In particular, the `e10_cache` hint is used to `enable` or `disable` the cache, directing applications' data to the local file system instead of the global file system. When the hint is set to `coherent` all the written data extents will be locked until cache synchronisation is completed. The `e10_cache_path` hint is used to control where, in the local file system tree, the cache file will reside. The `e10_cache_flush_flag` hint is used to control the synchronisation policy of cached data to the global file. If the hint is set to `flush_immediate` data will be immediately flushed to the global file. Alternatively, if the hint is set to `flush_onclose` data will be flushed to the global file when it is closed. A `flush_none` option is also provided to keep data local to the node and never

Table 4.2: Proposed MPI-IO hints extensions.

Hint	Value
`e10_cache`	`enable`, `disable`, `coherent`
`e10_cache_path`	cache directory pathname
`e10_cache_flush_flag`	`flush_immediate`, `flush_onclose`, `flush_none`
`e10_cache_discard_flag`	`enable`, `disable`
`e10_cache_threads`	number of synchronisation thread in pool
`ind_wr_buffer_size`	synchronisation buffer size [bytes]

flush it to the global file system. The `e10_cache_discard_flag` hint is used to perform basic cache space management. In particular, if the hint is set to `enable` the cache file will be removed after the file is closed, otherwise (`disable`) it will be retained until the user manually removes it. The `e10_cache_threads` hint is used to communicate the implementation the number of threads that need to be created in the synchronisation thread pool when the file is opened (default is 1). Finally, the `ind_wr_buffer_size` hint controls the size of the buffer used to synchronise cached data to the global file. This hint already existed in ROMIO but was only used during independent I/O to determine the write granularity. The hints in Table 4.2 can be used in conjunction with the collective I/O hints described in section 4.2.2.

Besides the proposed cache policies, more complex ones are possible. For example, the cache synchronisation could take into account the level of congestion of the I/O servers. The cache replacement policy could also use a more complex strategy to evict cached files (or extents of data inside the file). Although these can be implemented in ROMIO, they introduce more sophisticated functionalities that go beyond the scope of this work.

4.3.2 Cache Hints Integration in ROMIO

As already mentioned, the introduced MPI-IO hints are supported by a corresponding set of modifications in the ROMIO implementation [4] in the form of cache plugin. These modifications, following described, provide the functionalities necessary to handle the additional cache layer.

Cache Synchronisation Request

The `ADIOI_Sync_req_t` object provides the infrastructure required to initialise/finalise, set/get attributes to/from synchronisation requests. Synchronisation requests are initialised by the main thread in the write function and submitted to a separate thread (synchronisation thread) which will satisfy them while the main thread can progress with useful work. Listing 4.2 shows the

object API.

```
/* for ADIOI_Sync_req_{get,set}_key() */
enum {
  ADIOI_SYNC_TYPE = 0, /* sync req type        */
  ADIOI_SYNC_OFFSET,   /* sync req write off   */
  ADIOI_SYNC_DATATYPE, /* sync req datatype    */
  ADIOI_SYNC_COUNT,    /* sync req count       */
  ADIOI_SYNC_REQ,      /* sync req MPI_Request */
  ADIOI_SYNC_ERR_CODE, /* sync req error_code  */
  ADIOI_SYNC_FFLAGS,   /* sync req flush flag  */
  ADIOI_SYNC_ALL,      /* sync req all fields  */
  ADIOI_SYNC_REQ_ERR   /* sync req err code    */
};

struct ADIOI_Sync_req {
  int type_;
  ADIO_Offset off_;
  MPI_Datatype datatype_;
  int count_;
  ADIO_Request *req_;
  int error_code_;
  int fflags_;
};

/* Synchronisation Request Interfaces */
typedef struct ADIOI_Sync_req *ADIOI_Sync_req_t;

int ADIOI_Sync_req_init(ADIOI_Sync_req_t *, ...);
int ADIOI_Sync_req_init_from(ADIOI_Sync_req_t *,
  ADIOI_Sync_req_t);
int ADIOI_Sync_req_fini(ADIOI_Sync_req_t *);
int ADIOI_Sync_req_get_key(ADIOI_Sync_req_t, ...);
int ADIOI_Sync_req_set_key(ADIOI_Sync_req_t, ...);
```

Listing 4.2: Synchronisation Request API

There are two types of synchronisation requests, `ADIOI_THREAD_SYNC` and `ADIOI_THREAD_SHUTDOWN`. The first is used to describe a written file extent that needs to be copied from the cache to the global file system, the second is used to shut down the synchronisation thread when the file is closed.

Every synchronisation request has a `MPI_Request` handle used to start asynchronous copy of data from the cache to the global file system and to check upon request completion by the main thread using `MPI_Wait()`.

Cache Synchronisation Thread

The `ADIOI_Sync_thread_t` object provides the infrastructure to initialise/-finalise threads and to enqueue, flush and wait for synchronisation requests. A pool of synchronisation thread is created when the file is opened and destroyed when the file is closed. Listing 4.3 shows the object API.

```
enum {
  ADIOI_THREAD_SYNC = 0,
```

```
  ADIOI_THREAD_SHUTDOWN
};

struct ADIOI_Sync_thread {
  ADIO_File fd_;
  pthread_t tid_;
  pthread_attr_t attr_;
  ADIOI_Atomic_queue_t sub_;
  ADIOI_Atomic_queue_t pen_;
  ADIOI_Atomic_queue_t wait_;
};

typedef struct ADIOI_Sync_thread *ADIOI_Sync_thread_t;

int ADIOI_Sync_thread_init(ADIOI_Sync_thread_t *, ...);
int ADIOI_Sync_thread_fini(ADIOI_Sync_thread_t *);
void ADIOI_Sync_thread_enqueue(ADIOI_Sync_thread_t,
  ADIOI_Sync_req_t);
void ADIOI_Sync_thread_flush(ADIOI_Sync_thread_t);
void ADIOI_Sync_thread_wait(ADIOI_Sync_thread_t);
```

Listing 4.3: Synchronisation Thread API

The synchronisation thread object has three queues, a pending queue (*pen_*), a submission queue (*sub_*) and a wait queue (*wait_*). New synchronisation requests are first added to the pending queue using the `ADIOI_Sync_thread_enqueue()` function. Pending requests are not satisfied until they are moved to the submission queue using the `ADIOI_Sync_thread_flush()` function. This function also pushes a copy of the request pointer to the wait queue, which is later used by the `ADIOI_Sync_thread_wait()` function to wait for submitted requests to complete.

Synchronisation Thread Pool Initialisation

The thread pool is initialised in `ADIOI_GEN_Opencoll()`. This function is invoked by `MPI_File_open()` every time a new file is opened. Listing 4.4 shows the code responsible for opening the cache file in the local file system and to start the synchronisation threads.

```
void ADIOI_GEN_Opencoll(...) {
  ...
  /* check cache mode */
  if (fd->hints->e10_cache == ADIOI_HINT_ENABLE) {
    /* get MPI_File handle for cache and init it */
    MPI_File mpi_fh = MPIO_File_create(sizeof(
      struct ADIOI_FileD));
    fd->cache_fd = MPIO_File_resolve(mpi_fh);
    fd->cache_fd->filename = ADIOI_GetCacheFilePath(
      fd->filename, fd->hints->e10_cache_path);
    fd->cache_fd->file_system = fd->file_system;
    fd->cache_fd->fns = fd->fns;
    fd->cache_fd->cache_fd = ADIO_FILE_NULL;

    ...
```

```
16
17      /* every process tries opening the cache file */
18      (*(fd->fns->ADIOI_xxx_Open))(fd->cache_fd,
19        error_code);
20
21      ...
22
23      /* init synchronisation thread pool */
24      if (*error_code == MPI_SUCCESS &&
25          fd->hints->e10_cache_flush_flag !=
26            ADIOI_HINT_FLUSHNONE) {
27        if (fd->hints->cb_write == ADIOI_HINT_ENABLE)
28          if (fd->is_agg)
29            *error_code =
30              ADIOI_Sync_thread_pool_init(fd, NULL);
31          else
32            *error_code = MPI_SUCCESS;
33        else
34          *error_code = ADIOI_Sync_thread_pool_init(
35            fd, NULL);
36      }
37
38      /* check every process has returned success */
39      MPI_Allreduce(error_code, &max_error_code, 1,
40        MPI_INT, MPI_MAX, fd->comm);
41
42      if (max_error_code != MPI_SUCCESS) {
43        if (*error_code == MPI_SUCCESS ||
44            *error_code == ADIOI_ERR_THREAD_CREATE) {
45          (*(fd->fns->ADIOI_xxx_Close))(fd->cache_fd,
46            error_code);
47          if ((fd->cache_fd->access_mode & ADIO_CREATE) &&
48              (fd->cache_fd->access_mode &
49                ADIO_DELETE_ON_CLOSE)) {
50            ADIO_Delete(fd->cache_fd->filename, error_code);
51          }
52          /* fini synchronisation thread pool */
53          if (fd->hints->e10_cache_flush_flag !=
54              ADIOI_HINT_FLUSHNONE && fd->thread_pool) {
55            ADIOI_Sync_thread_pool_fini(fd);
56          }
57        }
58
59        /* Revert to standard file access: no cache */
60        ADIOI_Info_set(fd->info, "e10_cache", "disable");
61        fd->hints->e10_cache = ADIOI_HINT_DISABLE;
62      } else {
63        fd->cache_fd->is_open = 1;
64      }
65    }
66    ...
67  }
```

Listing 4.4: Synchronisation Thread Initialisation

In order for the implementation to exploit local NVM caching both local file creation and thread initialisation have to succeed. The MPI file handle for the cache file is stored in *cache_fd* inside the file handle of the global file. When collective I/O is enabled only the aggregators will initialise the synchronisation thread pool. In any other case every process will do independent I/O and will thus initialise its own pool.

Synchronisation Request Submission

`ADIOI_GEN_WriteContig()` is the function responsible for writing data to the file. This function is used by both collective and independent I/O. We have extended the write function to support creation and submission of new synchronisation requests to the thread pool. Listing 4.5 shows the corresponding implementation.

```
1  void ADIOI_GEN_WriteContig(...) {
2    ...
3
4    /* if using the cache select the cache file handle */
5    if (fd->cache_fd && fd->cache_fd->is_open) {
6      fh = fd->cache_fd;
7
8      /* if cache is coherent lock global file extent */
9      if (fd->hints->e10_cache_coherent == ADIOI_HINT_ENABLE)
10       ADIOI_WRITE_LOCK(fd, offset, SEEK_SET, len);
11   }
12
13   /* write to the cache file handle */
14
15   /* init and submit sync request */
16   if (fd->cache_fd && fd->cache_fd->is_open &&
17       fd->hints->e10_cache_flush_flag !=
18         ADIOI_HINT_FLUSHNONE) {
19     ADIOI_Sync_req_t sub;
20     int threads, curr_thread, idx;
21     ADIO_Request *r = (ADIO_Request *)ADIOI_Malloc(
22       sizeof(ADIO_Request));
23
24     *r = MPI_REQUEST_NULL;
25     threads = fd->hints->e10_cache_threads;
26     curr_thread = fd->thread_curr;
27     idx = curr_thread % threads;
28
29     /* init sync req */
30     ADIOI_Sync_req_init(&sub, ADIOI_THREAD_SYNC, offset,
31       datatype, count, r, 0);
32
33     /* enqueue sync request to thread and flush */
34     ADIOI_Sync_thread_enqueue(fd->thread_pool[idx], sub);
35     if (fd->hints->e10_cache_flush_flag ==
36         ADIOI_HINT_FLUSHIMMEDIATE)
37       ADIOI_Sync_thread_flush(fd->thread_pool[idx]);
38
```

```
39      /* select next thread in the pool */
40      fd->thread_curr = (curr_thread + 1) % threads;
41    }
42  }
```

Listing 4.5: Synchronisation Request Submission

For each write a new synchronisation request is initialised using the same set of I/O parameters (i.e. offset, datatype, count, etc) as the original write operation. The request is then enqueued to the appropriate thread in the pool using `ADIOI_Sync_thread_enqueue()`. If the `e10_cache_flush_flag` is set to `flush_immediate` the request is immediately scheduled to be served by invoking `ADIOI_Sync_thread_flush()`, otherwise it will be flushed when the file is closed.

Cache Flushing

`ADIOI_GEN_Flush()` is the function responsible for flushing the page cache. This function is used by `MPI_File_sync()`. We have extended the file flush function to support flushing and waiting of outstanding synchronisation requests. Listing 4.6 shows the corresponding implementation.

```
1  void ADIOI_GEN_Flush(...) {
2
3    ...
4
5    if (fd->cache_fd == NULL || fd->thread_pool == NULL ||
6        (fd->cache_fd && !fd->cache_fd->is_open) ||
7        (fd->cache_fd && fd->cache_fd->is_open &&
8        fd->hints->e10_cache_flush_flag ==
       ADIOI_HINT_FLUSHNONE))
9      goto fn_flush;
10
11   threads = fd->hints->e10_cache_threads;
12
13   /* Flush all the requests in each thread */
14   for (idx = 0; idx < threads; idx++) {
15     ADIOI_Sync_thread_flush(fd->thread_pool[idx]);
16   }
17
18   /* Wait for submitted requests to complete */
19   for (idx = 0; idx < threads; idx++) {
20     ADIOI_Sync_thread_wait(fd->thread_pool[idx]);
21   }
22   return;
23
24 fn_flush:
25   /* fsync global file handle */
26 }
```

Listing 4.6: Cache Flushing

When invoked, the flush routine will trigger cache synchronisation for every thread in the pool by invoking `ADIOI_Sync_thread_flush()`. As previously

described this will move all the requests in the pending queue to the submission queue to be served. If the pending queue is empty, the thread flush routine will return immediately. Once all the threads in the pool have been flushed, the routine waits on each of them to complete by invoking `ADIOI_Sync_thread_wait()` and finally returns.

Synchronisation Thread Pool Finalisation

`ADIO_Close()` is the function responsible for closing a file. This function is used by `MPI_File_close()`. We have extended the close function to support closing of cache file and finalisation of the synchronisation thread pool. Listing 4.7 shows the corresponding implementation.

```
1  void ADIO_Close(...) {
2
3    ...
4
5    if (fd->cache_fd) {
6      if (fd->cache_fd->is_open) {
7        /* First check if there is any outstanding sync */
8        (*(fd->fns->ADIOI_xxx_Flush))(fd, error_code);
9
10       /* Afterwards we can close the cache file */
11       (*(fd->fns->ADIOI_xxx_Close))(fd->cache_fd,
12         error_code);
13
14       /* Finilise thread pool */
15       ADIOI_Sync_thread_pool_fini(fd);
16     }
17
18     /* Only the process that created the file deletes it */
19     if ((fd->cache_fd->access_mode & ADIO_CREATE) &&
20         (fd->cache_fd->access_mode & ADIO_DELETE_ON_CLOSE))
21       ADIO_Delete(fd->cache_fd->filename, &err);
22
23     /* Free cache_fd */
24     ADIOI_Free(fd->cache_fd->filename);
25     MPIO_File_free(&(fd->cache_fd));
26     fd->cache_fd = ADIO_FILE_NULL;
27   }
28
29   ...
30
31 }
```

Listing 4.7: Synchronisation Thread Pool Finalisation

When invoked, the close routine will trigger the flushing of the cache (`(*(fd->fns->ADIOI_xxx_Flush))()`), close the cache file (`(*(fd->fns->ADIOI_xxx_Close))()`) and finalise the thread pool (`ADIOI_Sync_thread_pool_fini()`).

4.3.3 Cache Consistency Semantics

As far as write consistency is concerned, the MPI-IO interface does not make any assumption regarding the underlying storage system or its semantics. ROMIO specifically supports file systems that are both POSIX compliant, like Lustre, and non-POSIX compliant, like NFS or PVFS. In MPI-IO, written data becomes globally visible only after either `MPI_File_sync()` or `MPI_File_close()` are invoked on the MPI file handle and by default there is no write atomicity. The motivation is that data can be cached at some level locally in the compute nodes. The ROMIO implementation can overcome the risk of data inconsistency, e.g. related to false sharing of file system blocks, using persistent file realms [31], and can even enforce atomicity using `MPI_File_set_atomicity()`.

In our implementation we comply to the MPI-IO semantics just described. This means that data written to the local file system cache using the newly introduced MPI-IO hints will be globally visible to the rest of the nodes only under the following circumstances:

- The `e10_cache_flush_flag` has been set to `flush_immediate` by the user and synchronisation, started automatically by the implementation right after the write operation, has completed;
- The `e10_cache_flush_flag` has been set to `flush_onclose` by the user and the invoked `MPI_File_close()` has returned;
- The `MPI_File_sync()` function has been invoked by the user and it has returned.

Consistency for reading data from the cache is not clearly defined by the ext2ph algorithm. In general, data written to the local file system cache can be read back from the user without accessing the global file system. Nevertheless, the algorithm calculates the location of a data block based on the number of aggregators, their logical position within the set of aggregators, and the size of the complete data set. This means that a collective read that matches the previous write could safely read the data from the aggregators' cache without incurring any problem. In spite of that, in general reading from the cache requires additional metadata describing the file layout across the caches. For this reason, we currently do not support reads from the local file system cache.

Furthermore, whenever required, we can enforce cache coherency ensuring that read operations cannot access data that is currently in transit, i.e., not or only partially moved from the cache to the global file. This can be done by locking the file domain extent being cached until all the data has been made persistent in the global file. For this purpose ROMIO provides a set of internal locking macros, namely `ADIOI_WRITE_LOCK`, `ADIOI_READ_LOCK` and `ADIOI_UNLOCK` that we used in our implementation. The lock of cached data can be selected by setting the `e10_cache` hint in Table 4.2 to `coherent`. This will `enable` the cache and set locks appropriately, assuming underlying file system support.

4.3.4 Changes to the Application's Workflow

Simplifying, most HPC applications can be divided into multiple phases of computation, in which data is produced, and I/O, in which data is written to persistent storage for post-processing purposes as well as defensive checkpoint-restart. Focusing on the I/O phase and considering the case of applications writing to a shared file, the I/O phase can be divided into the following steps:

1. The file is opened using `MPI_File_open()`: at this point the info object containing the user defined MPI-IO hints is passed to the underlying ROMIO layers.

2. Data is written to the file using `MPI_File_write_all()`: these functions invoke the underlying `ADIOI_GEN_WriteStridedColl()` previously described in Figure 4.3.

3. The file is closed using `MPI_File_close()`: after the file is closed data must be visible to every process in the cluster.

To take advantage of the proposed MPI-IO hint extensions, the application's workflow has to be modified. Figure 4.5 shows the classical application's workflow (cache disabled) as well as the modified version using the new hints (cache enabled). The difference is that, in order to take advantage of the proposed hints and hide the cache synchronisation to the computation phase, the `MPI_File_close()` for the I/O phase 'k' has been moved at the beginning of the I/O phase 'k+1', just before the new file is opened.

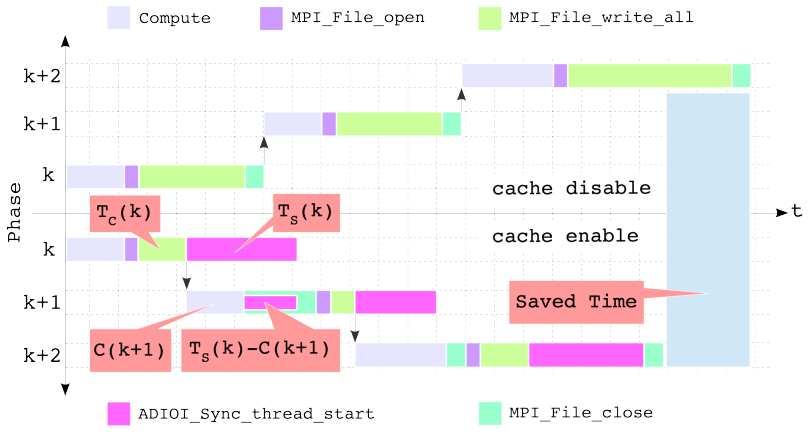


Figure 4.5: Standard and modified workflows. When cache is disabled compute phase 'k+1' starts after file 'k' has been closed. When the cache is enabled compute 'k+1' can start immediately after data has been written. At the same time, background synchronisation of cached data starts. File 'k' is closed before the file 'k+1' is opened, forcing the implementation to wait for cache synchronisation to complete.

MPIWRAP Library

Since the workflow modification just presented might not be feasible for legacy applications, we developed a MPI-IO wrapper library (called MPIWRAP), written in C++, that can reproduce this change behind the scenes. The library can be linked to the application or preloaded with `LD_PRELOAD` and has been used for all the experiments contained in this thesis. MPI-IO hints are defined in a configuration file and passed by the library to `MPI_File_open()`. We can define multiple hints targeting different files or groups of files. The library overloads `MPI_{Init,Finalize}()` and `MPI_File_{open,close}()` using the PMPI profiling interface. The workflow modification can be triggered for a specific set of files (identified by the same base name) in the configuration file. Whenever one of such files is closed, our `MPI_File_close()` implementation will return success. Nevertheless, the file will not be really closed. Instead, its handle will be kept internally for future references. When the next shared file with the same base name is opened, our `MPI_File_open()` implementation will search for outstanding opened file handles and will invoke `PMPI_File_close()` on them before opening the new file, thus triggering the cache synchronisation completion check for each of them.

4.3.5 I/O Bandwidth

According to the new I/O workflow, described in this section, we have that being $S(k)$ the amount of data written during phase 'k', $T_c(k)$ the time needed to write $S(k)$ collectively to the cache using `MPI_File_write_all()`, $T_s(k)$ the time needed to synchronise the cached data in every aggregator to the global file system (through `ADIOI_Sync_thread_start()`), and $C(k+1)$ the computation time of phase 'k+1', the resulting I/O bandwidth for 'k' is expressed by Equation 4.1:

$$bw(k) = \frac{S(k)}{T_c(k) + max(0,\ T_s(k) - C(k+1))} \tag{4.1}$$

Therefore, the total average bandwidth perceived by the application is:

$$BW = \frac{\sum_{k=0}^{N-1} S(k)}{\sum_{k=0}^{N-1} T_c(k) + max(0,\ T_s(k) - C(k+1))} \tag{4.2}$$

From Equation 4.1 (in which we have considered the open time neglectable) it is clear that the maximum performance can be obtained when $C(k+1) \geq T_s(k)$, that is, when we can completely hide cache synchronisation by the computation phase. On the other hand when $C(k+1) < T_s(k)$ we might have a minima in the bandwidth since `MPI_File_close()` needs to wait for cache synchronisation completion (Figure 4.5).

4.4 Evaluation

To evaluate the proposed MPI-IO hints we use three popular I/O benchmarks frequently adopted to profile collective I/O performance in other research works:

coll_perf[2], Flash-IO and IOR. Minor changes to the source code of the three benchmarks have been made to adapt them to our needs. For example, coll_perf and Flash-IO do not support writing to multiple files and the emulation of computing delays. Thus, we modified them to reproduce the workflow shown in Figure 4.5. The number of written files and a compute delay are now parameters that can be passed from the command line. In all our tests we used 512 MPI processes distributed over 64 nodes (8 procs/node), fixed the file stripe size to 4 MB and the stripe count to 4. Moreover, for simplicity, we also fixed the size of the cache synchronisation buffer (i.e. `ind_wr_buffer_size`) to 512 KB, which corresponds to the standard independent I/O buffer size. On the other hand, we varied the collective I/O parameters, i.e., the number of aggregators (from 8 to 64) and the collective buffer size (from 4 MB to 64 MB). For every combination of these parameters (<aggregators>_<coll_bufsize>) each benchmark writes four files of the same size (32 GB) with a compute delay of 30 seconds, which is in most cases enough to hide the synchronisation time. We compute the bandwidth as the average bandwidth over the four collective write operations (Equation 4.2). The different contributions within the collective I/O write path shown in Figure 4.3 are extracted from the ROMIO layer using MPE profiling [14]. Whenever the compute delay is not enough to hide synchronisation (e.g. when a small number of aggregators is used), the remaining synchronisation time is added to the total write time, thus reducing the bandwidth.

4.4.1 Testbed

Our testbed is a research cluster designed and developed in the context of the DEEP/-ER [2][3] projects (Dynamic Exascale Entry Platform/-Extended Reach). The DEEP/-ER cluster has 2048 cores distributed over 128 computing nodes (dual socket Sandy Bridge architecture). The storage system is composed of 6 Dell PowerEdge R520 servers equipped with 2 Intel Xeon Quad-core CPUs and 32 GB of memory and run the BeeGFS file system from Fraunhofer ITWM [7] (formerly known as FhGFS). The servers are connected to a SGI JBOD with 45 2TB SAS drives through a SAS switch using two 4x ports at 6 GB/s, for a total of four 8+2 RAID6 storage targets and 2 RAID1 targets for metadata and management data (1 drive is left as spare). One of the six I/O servers is dedicated as metadata server, one as management server and the remaining four as data I/O servers. Additionally, every compute node is equipped with 32 GB of RAM memory and a 80 GB SATA SSD containing the operating system plus an additional 30 GB ext4 partition (mounted under '/scratch') for general purpose storage. This partition, in our case, is used to locally cache collective writes. Finally, all the computing nodes are connected through an Infiniband QDR network and use ParaStation MPI [16] (PSMPI) version 5.1.0-1 as message passing library.

4.4.2 Coll_perf

In our coll_perf configuration every process writes one 64 MB block being part of a tridimensional distributed array to a shared file, thus generating a

[2]Collective I/O benchmark distributed with the MPICH package.

strided pattern. For every experiment, in which we vary the number of aggregators and the collective buffer size, we measure the coll_perf perceived write bandwidth in three different cases: *1*) when writing directly to the global file system (BW Cache Disabled), *2*) when writing to the cache (BW Cache Enabled) and afterwards flushing its content to the global file system asynchronously, and *3*) when writing to the cache without flushing its content to the global file system (TBW Cache Enable). The last case provides the theoretical bandwidth achievable when the cache synchronisation cost is completely hidden. Additionally, our coll_perf experiments do not include the last write phase contribution (Figure 4.5). In fact, for the last write the cache synchronisation cost cannot be hidden since there is no following compute phase. We will show the effect that the last write has on the average bandwidth of the IOR benchmark at the end of this evaluation.

Figure 4.6 shows the write bandwidth for the three cases previously discussed. First of all, we can observe that for most of the experiments the aggregate bandwidth when using the cache is higher than the bandwidth measured when using only the global file system. In particular, we can reach a peak performance of about 20 GB/s, compared to the 2 GB/s of the standard case (BW Cache Disabled), which gives a ten fold improvement. Second, when the number of aggregators is equal to 8, we notice a reduced performance, as the synchronisation cost cannot be completely hidden.

The effect of the non-hidden cache synchronisation (not_hidden_sync) is shown in Figure 4.7. This figure presents the collective I/O performance breakdown for all the components shown in Figure 4.3. As we can see, although

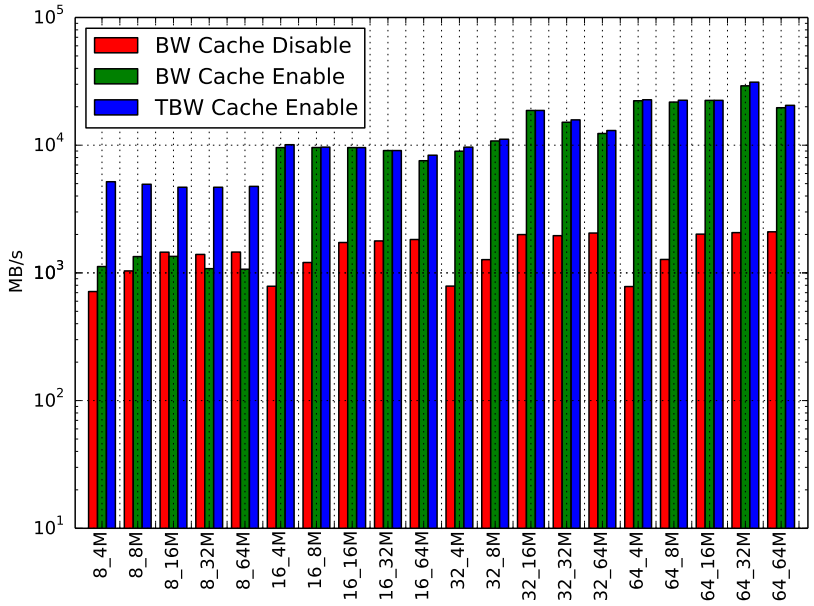


Figure 4.6: coll_perf perceived bandwidth for all combinations of <aggregators>_<coll_bufsize>.

the theoretical bandwidth (TBW Cache Enable) peaks at 4 GB/s (Figure 4.6), the measured bandwidth (BW Cache Enable) can be even worse than the standard case (BW Cache Disable). This happens because the cache data cannot be flushed to the global file system quickly enough. In all the other experiments, the perceived and theoretical bandwidth are well aligned.

As already said in the previous sections, our SSDs based approach can also help to reduce the global synchronisation cost in the extended two phase I/O algorithm at the base of collective I/O. In fact, by comparing chapters/chapter3/figures 4.7 and 4.8, we see that the global synchronisation costs in collective I/O, represented by `MPI_Alltoall()` (shuffle_all2all) and `MPI_Allreduce()` (post_write) are consistently reduced when using the cache.

Finally, we observe that most of the times, when using the cache, larger collective buffers do not produce large performance improvements. This means that we can achieve good performance with small buffers and thus reduce the memory pressure of collective write operations on compute nodes.

4.4.3 Flash-IO

Flash-IO is the I/O kernel of the Flash application. Flash is a block-structured adaptive mesh hydrodynamics code. The computational domain is divided into blocks which are distributed across the processors. Typically a block contains 16 zones in each coordinate direction (x,y,z) and a perimeter of guard cells (presently 4 zones deep) to hold information from the neighbors. The application writes three files using the parallel HDF5 library: a checkpoint file,

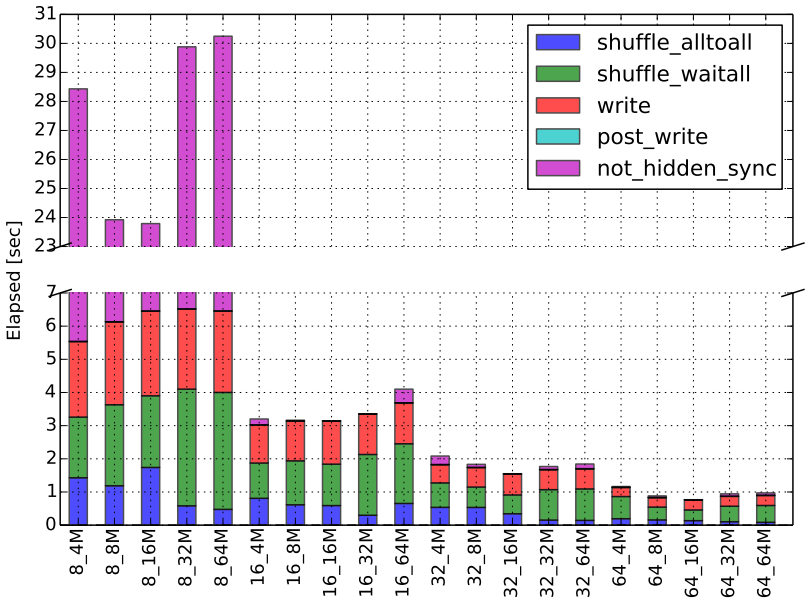


Figure 4.7: coll_perf collective I/O contribution breakdown when cache is enabled.

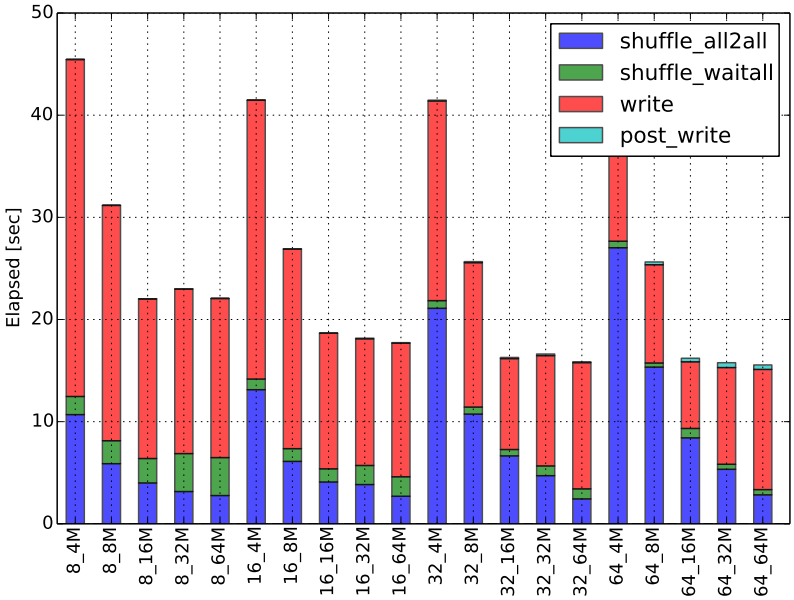


Figure 4.8: coll_perf collective I/O contribution breakdown when cache is disabled.

a plot file without corner data and a plot file with corner data. The checkpoint file is the biggest of the three and consumes the majority of the I/O time. In our configuration the checkpoint file contains 80 blocks/process and each of the 16 zones/block contains 24 variables encoded with 8bytes (768 KB/proc/block). Therefore, the total size is slightly bigger than 30 GB (including metadata).

Figure 4.9 shows the write bandwidth perceived by Flash-IO for all the experiments performed in the different cases under study. Similarly to coll_perf, when the cache is enabled we can hide the cache synchronisation cost for most of the experiments. Once again, like in the previous case, when the number of aggregators is equal to 8 we have a mismatch between the perceived and the theoretical bandwidth. For Flash-IO the peak bandwidth is about 40 GB/s when using 64 aggregators and 4 MB collective buffer size, against the 2 GB/s measured when writing directly to the parallel file system.

Figure 4.10 shows the effect of cache usage on the different collective I/O performance contributions. We can clearly see that when the number of aggregators is equal to 8 the cache synchronisation cannot be completely hidden, causing the bandwidth mismatch previously observed in Figure 4.9. Furthermore, like in the coll_perf case the global synchronisation contributions can be reduced when using the cache, and so can the memory pressure on the compute nodes. Nevertheless, for the 64 aggregators and 16 MB collective buffer size configuration the global synchronisation overhead associated to `MPI_Allreduce()` (post_write) has a larger value. The outlier causes a strong reduction in the write bandwidth, although we can still achieve more than 10 GB/s. This indicates that the effect of global synchronisation when using the cache can be even more severe, due

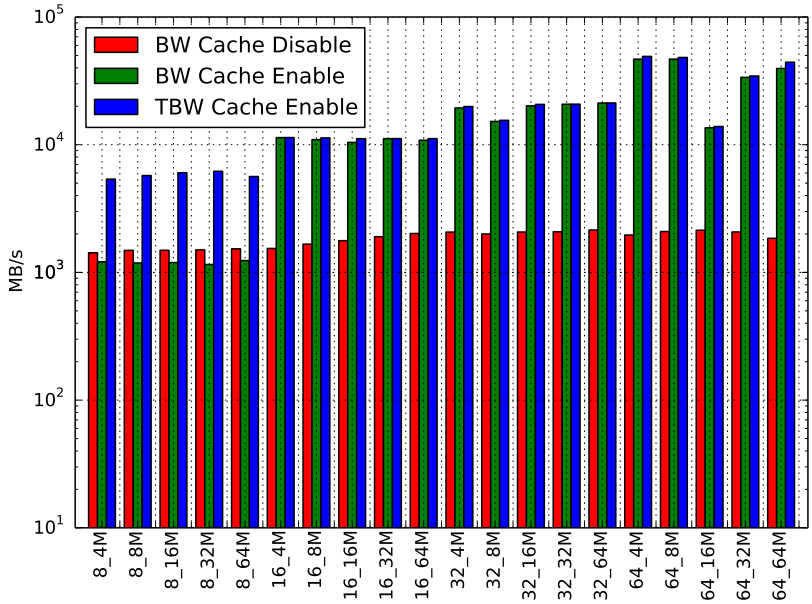


Figure 4.9: Flash-IO perceived bandwidth for all combinations of <aggregators>_<coll_bufsize>.

to the much higher bandwidth achievable compared to the standard global file system approach.

4.4.4 IOR

We tested IOR when writing collectively to a shared file. Each of the 512 MPI processes writes one 8 MB block of data for each of the 8 segments, that is, a 32 GB file during every test. As for the previous two benchmarks, Figure 4.11 shows the write bandwidth perceived by IOR. Nevertheless, unlike the previous benchmarks, in IOR we also take into account the non-hidden synchronisation cost coming from the last write phase. In this case, the peak bandwidth when using the cache can reach about 6 GB/s versus the 2 GB/s of the standard collective write to the global file system. We can also see that the theoretical bandwidth is aligned with the chapters/chapter3/figures presented for coll_perf and Flash-IO. In fact, although we can hide cache synchronisation costs for the three intermediate write phases in IOR, the fourth write phase will limit the peak performance.

chapters/chapter3/figures 4.12 shows the collective I/O cost breakdown for all the time contributions. In this figure we can clearly observe the not_hidden_sync term preventing IOR from achieving higher performance. This is added to the other time contributions and corresponds to the term $T_s(k) - C(k+1)$ reported in Equation 4.1. In our specific case, $k = 4$ and $C(4+1) = 0$, meaning that the total write time is accounting for the whole $T_s(4)$ term.

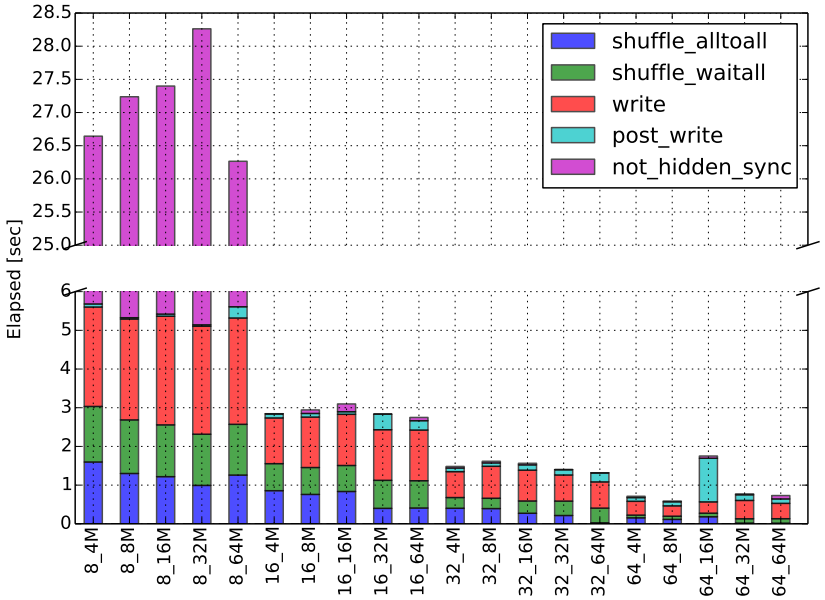


Figure 4.10: Flash-IO collective I/O contribution breakdown when cache is enabled.

4.5 Related Work

Many research works have tried to optimise collective I/O focusing on different aspects. Yu and Vetter [53] before us have identified the global synchronisation problem as one of the most severe for collective I/O performance. They exploited access pattern characteristics, common in certain scientific workloads, to partition collective I/O into smaller communication groups and synchronise only within these. Block-tridiagonal patterns, not directly exploitable, are automatically reduced, through an intermediate file view, to a more manageable pattern and can thus take advantage of the proposed solution. The ADIOS library [40] addresses this problem similarly by dividing a single big file into multiple files to which collective I/O is carried out independently for separated smaller groups of processes. Lu, Chen, Thakur and Zhuang [41] further explored collective I/O performance beyond global synchronisation and considered memory pressure of collective I/O buffers. They proposed a memory conscious implementation that accounts for reduced memory per core in future large scale systems. Liao [37] focused on the file domain partitioning impact on parallel file systems' performance. He demonstrated that by choosing the right file domain partitioning strategy, matching the file system locking protocol, collective write performance can be greatly improved. Yong, Xian-He, Thakur, Roth and Gropp [29] addressed the problem of I/O server contention using a layout aware strategy to reorganize data in aggregators. On the same lines, Xuechen, Jiang and Davis [54] proposed a strategy to make collective I/O 'resonant' by matching memory layout and physical placement of data in I/O

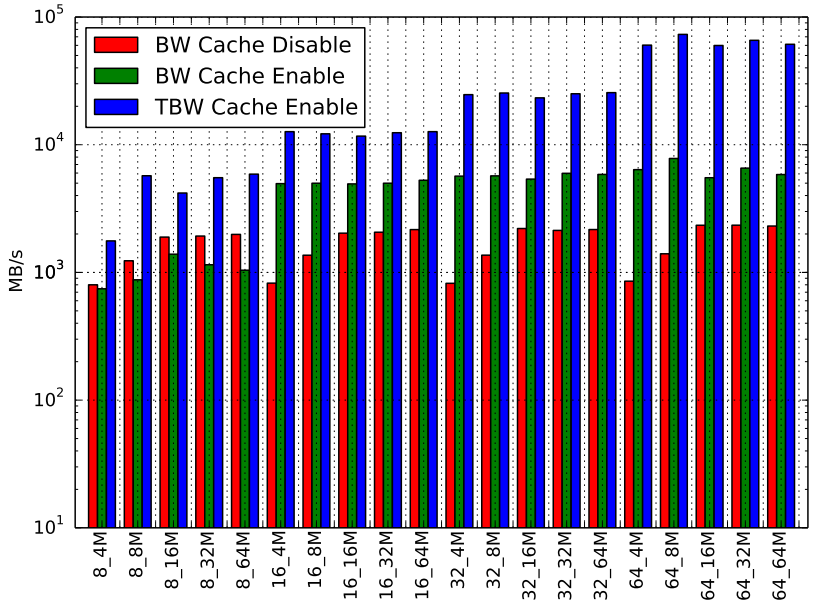


Figure 4.11: IOR perceived bandwidth for all combinations of <aggregators>_-<coll_bufsize>.

servers and exploiting non-contiguous access primitives of PVFS2. The strategy proposed is similar in concept to the Lustre implementation of collective I/O in which file contiguous patterns are converted to stripe contiguous patterns and the concurrency level on OSTs can be set using the MPI-IO hint `romio_lustre_co_ratio` (Client-OST ratio). Liu, Chen and Zhuang [39] exploited the scheduling capabilities of PVFS2 I/O servers to rearrange I/O requests' order and better overlap read and shuffle phases among different processes.

Lee, Ross, Thakur, Xiaosong and Winslett [36] proposed RTS as infrastructure for remote file access and staging using MPI-IO. Similarly to our approach, RTS uses additional threads, Active Buffering Threads (ABT) [42], to transfer data in background to the compute phase. Moreover, the authors also modified the ABT ROMIO driver implementation to stage data in the local file system whenever the amount of main memory runs low. Although they include collective I/O in their study, they lack a detailed evaluation of the impact that SSD caching can have on the different performance contributions of collective I/O and the additional reduction of memory pressure. Furthermore, remote staging of data requires additional nodes while we collocate storage with compute. The SCR library [19] also uses local storage resources to efficiently write checkpoint/restart data but this is targeted to a specific use case and requires the modification of the application's source code to be integrated. Other works, focus on I/O jitter reduction using multi-threading and local buffering resources [32], but we do an evaluation of collective I/O and show how the effect of I/O jitter can become even more prominent when using fast NVM devices. More recently the Fast Forward I/O project [6], from U.S. Department of Energy

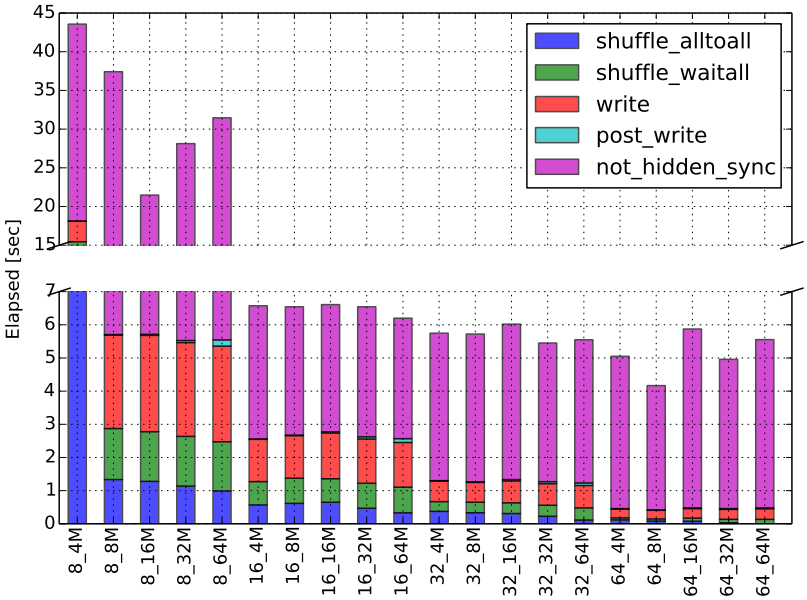


Figure 4.12: IOR collective I/O contribution breakdown when cache is enabled.

(DOE), proposed a burst buffer architecture to absorb I/O bursts from file system clients into a small number of high performance storage proxies equipped with high-end solid state drives. This technique has been, e.g., implemented in the DDN Infinite Memory Engine [1]. Even though the burst buffer solution is interesting, it may require very expensive dedicated servers as well as significant changes to the storage system architecture.

Unlike previous works, we proposed a fully integrated, prototype solution for new available memory technologies able to scale aggregate bandwidth in collective I/O with the number of available compute nodes. Additionally, our solution does not require any proprietary hardware or dedicated kit to work. We demonstrate that SSD based cache can reduce the synchronisation overhead intrinsic in the collective I/O implementation in ROMIO as well as the requirement for large collective buffers (memory pressure). Our implementation is compatible with legacy codes, since it does not require any change at the application level, and can work out of the box with any backend file system, although in DEEP-ER we focused on BeeGFS. At the moment the cache synchronisation is implemented in the ADIO UFS driver using pthreads. Future releases of BeeGFS will support native caching, including asynchronous flushing of local files to global file system. We have already integrated ROMIO with a BeeGFS driver that will take advantage of these functionalities.

Bibliography

[1] DDN Infinite Memory Engine. http://www.ddn.com/products/infinite-memory-engine-ime14k/.

[2] DEEP, Dynamic Exascale Entry Platform. http://www.deep-project.eu.

[3] DEEP-ER, Dynamic Exascale Entry Platform - Extended Reach. http://www.deep-er.eu.

[4] Exascale10 (E10) Code Base. http://www.github.com/gcongiu/E10.git.

[5] Exascale10 (E10) initiative. http://www.exascale10.com.

[6] Fast Forward I/O project. https://users.soe.ucsc.edu/~ivo/blog/2013/04/07/the-ff-stack.

[7] Fraunhofer File System. http://www.beegfs.com/cms.

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

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

[10] HDF5 Library. https://www.hdfgroup.org/HDF5.

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

[12] 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.

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

[14] MPI Parallel Environment. http://www.mcs.anl.gov/research/projects/perfvis/software/MPE.

[15] netCDF Library. http://www.unidata.ucar.edu/software/netcdf.

[16] Parastation MPI. http://www.par-tec.com/products/parastation-mpi.html.

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

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

[19] Scalable Checkpoint Restart library. http://computation.llnl.gov/projects/scalable-checkpoint-restart-for-mpi.

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

[21] Nawab Ali, Philip H. Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert B. Ross, Lee Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *CLUSTER*, pages 1–10. IEEE Computer Society, 2009.

[22] Peter J. Braam. Lustre: a scalable high-performance file system. White Paper, November 2002.

[23] Surendra Byna, Yong Chen, Xian he Sun, and Rajeev Thakur. Parallel i/o prefetching using mpi file caching and i/o signatures.

[24] 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 *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2011.

[25] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, Georgia, USA, October 2000. USENIX Association.

[26] Fay Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the 3rd Conference on Operating Systems Design and Implementation (OSDI)*, OSDI '99, pages 1–14, Berkeley, CA, USA, 1999. USENIX Association.

[27] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, SC '08, pages 40:1–40:10, Piscataway, NJ, USA, 2008. IEEE Press.

[28] Yong Chen and P.C. Roth. Collective prefetching for parallel i/o systems. In *Proceedings of the 5th Parallel Data Storage Workshop (PDSW)*, pages 1–5, 2010.

[29] Yong Chen, Xian-He Sun, R. Thakur, P.C. Roth, and W.D. Gropp. Lacio: A new collective i/o strategy for parallel i/o systems. In *Proceedings of the* 25^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 794–804, Anchorage, Alaska, USA, May 2011.

[30] Avery Ching, Alok N. Choudhary, Wei-keng Liao, Lee Ward, and Neil Pundit. Evaluating i/o characteristics and methods for storing structured scientific data. In *Proceedings of the* 20^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006.

[31] Kenin Coloma, A. Choudhary, Wei-Keng Liao, L. Ward, E. Russell, and N. Pundit. Scalable high-level caching for parallel i/o. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 96–, April 2004.

[32] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 155–163, Sept 2012.

[33] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. I/o acceleration with pattern detection. In *Proceedings of the 22nd*, HPDC '13, pages 25–36, New York, NY, USA, 2013. ACM.

[34] Jun He, Huaiming Song, Xian-He Sun, Yanlong Yin, and Rajeev Thakur. Pattern-aware file reorganization in mpi-io. In *Proceedings of the* 6^{th} *Parallel Data Storage Workshop (PDSW)*, pages 43–48, Seattle, Washington, USA, 2011.

[35] Jun He, Xian-He Sun, and R. Thakur. Knowac: I/o prefetch via accumulated knowledge. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, pages 429–437, 2012.

[36] J. Lee, R. Ross, R. Thakur, Xiaosong Ma, and M. Winslett. Rfs: efficient and flexible remote file access for mpi-io. In *Cluster Computing, 2004 IEEE International Conference on*, pages 71–81, Sept 2004.

[37] Wei-keng Liao. Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol. *IEEE Transactions on Parallel and Distributed Systems*, 22(2):260–272, 2011.

[38] Wei-keng Liao and Alok N. Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC)*, Austin, Texas, USA, November 2008.

[39] Jialin Liu, Yong Chen, and Yi Zhuang. Hierarchical I/O scheduling for collective I/O. In *Proceedings of the* 13^{th} *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, pages 211–218, Delft, Netherlands, May 2013.

[40] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: the challenges

and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[41] Yin Lu, Yong Chen, Rajeev Thakur, and Yu Zhuang. Abstract: Memory-conscious collective I/O for extreme-scale HPC systems. In *Proceedings of the 2012 High Performance Computing, Networking, Storage and Analytics (SCC)*, pages 1360–1361, Salt Lake City, UT, USA, November 2012.

[42] Xiaosong Ma, M. Winslett, J. Lee, and Shengke Yu. Improving mpi-io output performance with active buffering plus threads. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, April 2003.

[43] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia, and Alice Koniges. Mpi-io/gpfs, an optimized implementation of mpi-io on top of gpfs. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 17–17, New York, NY, USA, 2001. ACM.

[44] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice E. Koniges, and Alison White. Towards a high-performance implementation of MPI-IO on top of GPFS. In *Proceedings of the* 6^{th} *International Euro-Par Conference (Euro-Par)*, pages 1253–1262. Munich, Germany, August 2000.

[45] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 USENIX Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002.

[46] W. Richard Stevens and Stephen A. Rago. *Advanced programming in the UNIX environment*, chapter Advanced IPC, pages 642–652. Addison-Wesley professional computing series, May 2013.

[47] Rajeev Thakur and Alok N. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, 1996.

[48] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Proceedings of the* 6^{th} *IEEE International Symposium on Frontiers of Massively Parallel Computation (FRONTIERS)*, pages 180–187. IEEE Computer Society Press, 1996.

[49] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings of the 7th IEEE International Symposium on Frontiers of Massively Parallel Computation (FRONTIERS)*, page 182, 1999.

[50] Nancy Tran and Daniel A. Reed. Automatic arima time series modeling for adaptive i/o prefetching. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):362–377, April 2004.

[51] Steve VanDeBogart, Christopher Frost, and Eddie Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 24–24, Berkeley, CA, USA, 2009. USENIX Association.

[52] Liu Ying. Lustre ADIO collective write driver. White Paper, October 2008.

[53] Weikuan Yu and J. Vetter. Parcoll: Partitioned collective i/o on the cray xt. In *Proceedings of the* 37^{th} *International Conference on Parallel Processing (ICPP)*, pages 562–569, September 2008.

[54] Xuechen Zhang, S. Jiang, and K. Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, May 2009.

List of Figures

List of Tables