Improving I/O Performance in HPC Through Guided Prefetching and Non-Volatile Memory Devices

Dissertation submitted
for the award of the title
"Doctor
of Natural Sciences"
to the Faculty of Physics, Mathematics, and Computer Science
of Johannes Gutenberg University Mainz
in Mainz

Giuseppe Congiu

Abstract

High performance computing has become one of the fundamental contributors to the progress of science and technology. However, one challenge of high performance computing remains the gap between compute components and hard disk drives performance, used to persistently store data, that has made I/O the main limitation to the scalability of applications. Over the years many research efforts have been dedicated to the alleviation of the I/O gap; a consistent portion of these has focused on caching techniques to hide disk accesses and reduce the time applications spend stalled on I/O transfers. Two popular caching techniques used to improve read and write performance are, respectively, prefetching and write-behind. Prefetching can mask disk accesses by preemptively loading data into main memory and serving it to applications from there, while write-behind buffers data updates into main memory and allows applications to return to compute faster, taking care of moving data to the disk at a later time.

More recently the emergence of larger dynamic random access memories and new storage technologies has opened up a new range of possibilities to implement caching. In this thesis we focus on guided prefetching in Linux and collective write optimizations based on write-behind that exploit solid state drives on compute nodes of high performance computing clusters. Our prefetching strategy is directed to improve the I/O throughput of the parallel file system and hide its access time to scientific analysis codes; while our collective I/O solution is directed to improve the I/O throughput of applications writing their datasets to the parallel file system for defensive checkpoint restart.

We have implemented our prefetching strategy into a new middleware prototype called Mercury and extended the ROMIO MPI-IO implementation with additional support for solid state drives; these can be exploited during collective write operations to locally buffer bursts of I/O activity in compute nodes, postponing the transfer of the data to the global file system at a later time.

Experimental results in real environments demonstrate the effectiveness of our ideas and provide a base for the development of future production ready solutions based on these.

Acknowledgments

[Acknowledgements have been removed from the electronic version of the thesis to comply to University of Mainz regulations.]

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Today *high performance computing* (HPC) has penetrated both industry and science domains and is effectively employed in the design and development of new products [Isa13] as well as the study of complex natural phenomena ranging from high-energy physics [Cha11a] to space weather [MLRu10] [DLMM13] and earth science [SHK11]. Codes running on HPC clusters need to process large amounts of data that frequently do not fit into the available system memory and thus need to be stored out-of-core into an external storage system. These codes have large demand for storage capacity that, due to their low cost, is frequently satisfied by means of *hard disk drives* (HDDs).

Although HDDs can offer high capacity at low cost, their access performance is limited by the mechanical parts used to store and retrieve information on the magnetic media. The gap between hard disk access time and CPU compute capabilities imposes a performance gap on applications that have to spend a large portion of their run-time waiting on data transfers between out-of-core storage (HDDs) to in-core *dynamic random access memory* (DRAM) [CHA$^+$11b] [CR10].

In order to alleviate this performance gap, designers have built distributed storage systems in which many hard disks can be accessed concurrently through a high-performance network and corresponding parallel file system softwares [Bra02] [SH02] [CLI$^+$00] [Hei02] to efficiently manage them; such systems are optimized for large sequential I/O transfers that exploit the characteristics of the underlying storage media. To further reduce the I/O latency file systems use a DRAM cache to buffer frequently accessed data that, in this way, can be served faster from memory instead of fetching it from remote disks.

Prefetching is a well known technique that allows to anticipate I/O needs by preemptively fetching data from disks into the cache before it is referenced by the application. In order to effectively hide disk access time to applications, prefetching has to be performed at the right moment and for the right amount of data. Because the amount of memory dedicated to caching is limited, prefetching too much data or prefetching it too early might cause more urgently needed data to be removed from the cache, forcing the application to fetch it again from disk. Similarly, prefetching too little data or prefetching it too late might only partially hide disk latency or add no benefit if requested data is already in the cache; even worse, delayed prefetching might cause data to be re-fetched from disk although no longer needed. If performed appropriately prefetching can boost application performance completely hiding I/O latency.

Jobs running on HPC cluster also have to be protected from the failure of hardware components and soft-errors by periodically writing their computational context to stable storage (process commonly known as checkpointing) [SG06] [SG07]; while reads are limited to the loading of initial configuration parameters at the beginning of the simulation or to restore the computational context after a system crash and restart. The computational context is represented by large multi-dimensional variables which value is determined by the collaboration of the application's processes running concurrently on different nodes of the cluster. When transferring program variables from memory to disk the layout of data is changed to adapt the multi-dimensional domain to the single-dimensional representation on the device (i.e., sequence of blocks). This conversion causes a mismatch between memory and storage representations that results into a large number of small non-contiguous disk requests which degrades the overall I/O performance [NKP+96] [SR98].

To address the mismatch between memory and storage layout additional software components, called I/O middlewares, have been added to the I/O stack. I/O middlewares can transparently adapt the I/O behaviour of the application to the storage system by converting the original access pattern into an intermediate representation that is presented to the parallel file system. The intermediate representation is built keeping into consideration the characteristics of the storage hardware and extract maximum performance from it [TC96] [BGG+09] [MBMS10] [FWP09] [LKS+08].

Storage devices like *solid state drives* (SSDs) offer another opportunity for reducing the I/O gap. SSDs based on *flash* technology are block based and provide better I/O throughput at higher cost tag compared to HDDs. Solutions using SSDs in combination with I/O middlewares can be adopted to implement an additional, faster, storage tier between DRAM and disks that buffer bursts of writes generated by checkpointing activity [LCC+12]. These block based buffers can effectively absorb the intense I/O activity of applications, hiding to them the latency of slower disk based tiers. Buffered data is kept in the SSD until it is full or when the user explicitly requests to flush it to its final location on disk; however, control is returned to the application as soon as data is persisted into the SSD buffer, allowing it to proceed with its tasks.

More recently, the emergence of new storage and memory devices like *storage class memories* (SCMs) [WAA13] [ZGF+15], providing access times comparable to DRAM, opens up a new range of possibilities to implement storage systems in memory. *Non volatile main memory* (NVMM) implemented using SCM devices can be exploited as persistent storage media to implement a new class of file systems. These file systems will be based on totally different assumptions compared to current disk based implementations. For example, classical file systems use a buffer cache to consolidate writes in memory before transferring data to the storage device; this design choice is forced by the geometry of hard disks in which accesses are more efficient for large contiguous blocks of data instead of small non-contiguous requests. With the new devices the buffer cache only adds an extra memory copy that does not bring any benefit to access performance. In this case, file systems can bypass the buffer cache and write directly to the NVMM.

1.1 Contributions

In the depicted scenario the contribution of this work is two fold. First, we present Mercury [CGP+17], a transparent guided I/O framework able to optimize file read patterns in scientific applications, allowing users and administrators to control the I/O behavior of their applications without modifying them. Mercury is especially helpful for converting numerous small read requests into a few larger requests using a technique called data sieving. The immediate effect of this optimization is the increase of the application perceived I/O bandwidth, the reduction of the number of I/O requests reaching the remote back-end storage devices and, ultimately, the reduction of the running time of the application. Additionally, we also present a *virtual file system* (VFS) modification of the Linux kernel that allows Mercury to forward prefetching hints to the Lustre file system. Second, we present an optimization for parallel write operations that exploits SSDs in HPC compute nodes [CNSB16]; we demonstrate that the use of SSDs as additional persistent cache layer on file system clients can speed up parallel write performance to a shared file in MPI-IO. We have integrated SSDs support into the ROMIO middleware using additional

MPI-IO hints and service routines, and implemented a ROMIO driver for the BeeGFS file system, which can autonomously handle the local SSDs.

1.2 Remainder

The remainder of this thesis is organized as follows: Chapter 2 reviews the technical background on high performance storage systems, current and emerging storage technologies, caching and prefetching, and I/O middleware solutions employed to improve checkpointing patterns; Chapter 3 presents the Mercury middleware design and implementation, outlining the modifications required to the Linux kernel to enable forwarding of prefetch calls to the Lustre parallel file system; Chapter 4 presents our solution to integrate SSDs into MPI-IO using ROMIO; Chapter 5 presents experimental results for read and write intensive I/O patterns using, respectively, Mercury and our extended ROMIO implementation; and finally Chapter 6 presents conclusions.

Chapter 2

Background

In this chapter we review the background on parallel I/O for high performance computing, including optimization and caching techniques that are at the base of the work presented in the following chapters. We start by giving a high-level overview of HPC I/O systems in Section 2.1, including an introduction to basic parallel file system architectures, along with the limitations imposed by most common designs and corresponding impacts on scientific I/O workloads performance; we then review different memory technologies in Section 2.2, including emerging storage class memories and non-volatile memory devices, also giving some example of how these technologies can be applied to HPC I/O systems; in Section 2.3 we look at how the gap between compute and I/O can be alleviated by using prefetching; finally, in Section 2.4, we present some of the most relevant I/O library solutions used to adapt applications' I/O behaviour to the characteristic of the underlying file systems, thus improving performance of scientific codes at large scales.

2.1 Introduction to HPC I/O

Nowadays high performance computing clusters have penetrated industry and science domains and are used in the development of products as well as in the study of complex natural phenomena that would be otherwise difficult to analyze in a real experimental setup. For example, *computational fluid dynamic* (CFD) codes [Isa13] are widely used by aircraft manufacturer in the design and development of airliners, limiting the number of required wind tunnel tests and physical prototypes, cutting costs and reducing the time to market. Similarly, high performance computing is also applied to the study of climate change [VGL$^+$13], medium range weather forecasting [And15], and more generally to many other branches of science [Cha11a] [MLRu10] [DLMM13] [SHK11].

Scientific codes need to process large amounts of data, that most often do not fit in the system memory and thus have to be stored permanently elsewhere (out-of-core). Applications store data for mainly two reasons, as a defensive mechanism to protect from the failure of hardware components, also called checkpointing, or as an input to be fed into other computational tasks of a larger workflow. Workflows are intended as ensemble of several applications that overall collaborate to the solution of a larger and more complex problem; this can include the pre-processing of inputs coming from external sensors, the simulation of some physics phenomena that ingests the inputs and generates an output as a result, the post-processing of the output and, possibly, its visualization for user inspection [FOR$^+$00] [MLRu10]. Independently from the specific field, applications save their data as files in a file system, which mostly relies on hard disk drives as permanent storage devices.

Modern CPUs can process data hundreds of thousands times faster than the time needed to fetch it from hard disks. This creates a performance gap between compute and I/O that imposes serious limitations to the scalability of HPC applications. To alleviate this I/O gap designers have built distributed storage systems in which data from one file is partitioned and spread across many hard disks. Such disks are assembled into enclosures, or disk arrays, and are connected to servers through a Storage Area Network (SAN). The storage servers are then placed in the same network of compute nodes, that reach them with data requests. Figure 2.1 shows how the components just described are organized in the system. *Redundant array of inexpensive disks* (RAID) [PGK88] volumes in disk enclosures protect data against single or multiple disk failures, depending on the RAID scheme. Similarly, disk enclosures are connected to two servers to maintain data availability in the event of a server failure.

In the described system, a compute node writing data to a file would need to keep track of what servers store which part of the file and contact directly each of them. Moreover, this tracking information should be kept in a publicly accessible place so that other clients can learn about the file and retrieve it if needed. Parallel file systems perform these operations on behalf of the clients, relieving them from the burden of directly managing the storage hardware. The parallel file system exports to the clients a simple file model and corresponding interface that can be used to access and manipulate file attributes and data. The model is normally, but not always, compliant to the POSIX-IO standard [POS], supported by

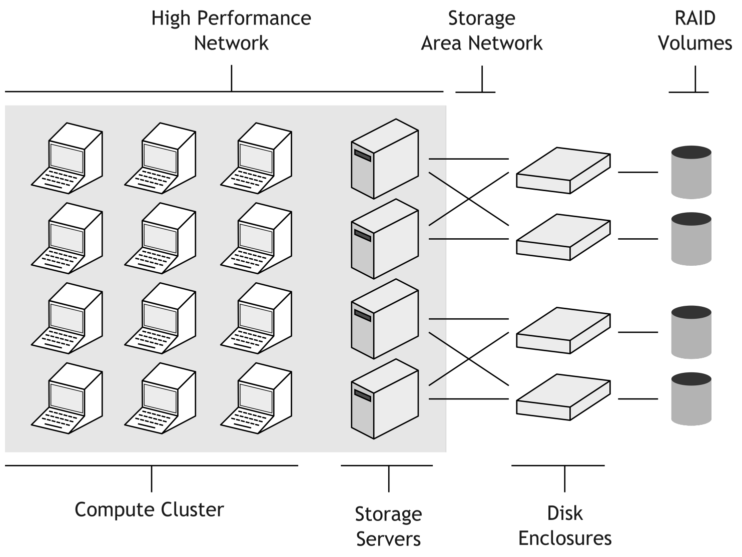


Figure 2.1: Example of HPC storage system high-level architecture. Compute nodes are connected to servers through a low-latency network. Each RAID volume protects data against single, or multiple, disk failures. RAID volumes are contained into disk enclosures, each of which is connected to two servers using a storage area network; in this way if one of the servers fails data is still reacheable through the failover node.

UNIX-like operating systems.

Accessing data in a parallel file system involves additional overhead because of the extra hardware components that are present in the distributed I/O architecture. A file read operation, for example, is converted by the file system clients into a number of remote requests, or *remote procedure calls* (RPCs). These RPCs are sent to the appropriate servers through the network. The servers satisfy the requests by generating a set of hardware commands for the disks, which finally transfer the data into the servers' memory. When the data is available in memory, the servers send it back to the clients through the network. Figure 2.2 shows an example of the described process. For simplicity we do not consider the contribution of RAID controllers or the SAN and assume all the data is located in one disk.

The time required to perform an I/O operation is given by Equation 2.1.

$$T_{IO} = T_{client} + T_{network} + T_{server} + T_{disk} \tag{2.1}$$

In this equation there are four contributions, one for each hardware component involved in I/O. T_{client} indicates the time needed by the file system client to initiate and finalize the operation; $T_{network}$ indicates

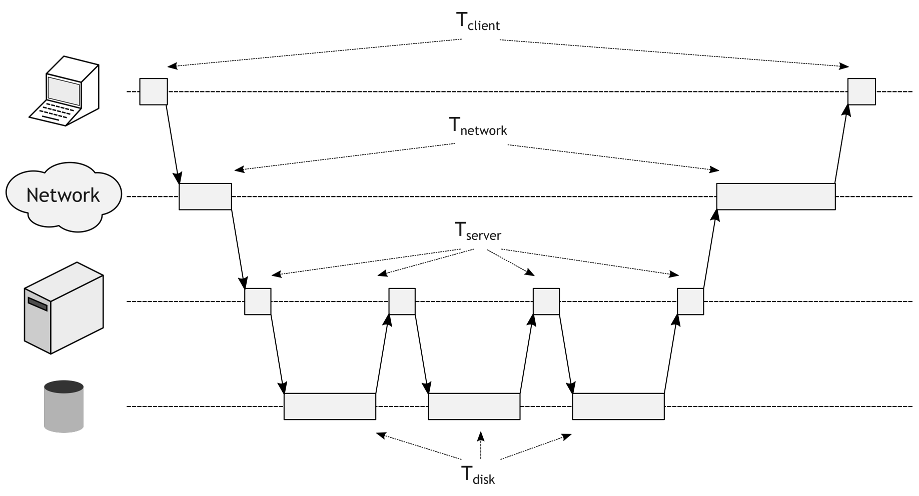


Figure 2.2: Contribution to I/O time in a distributed storage system.

the time required to transfer the RPC from the client to the server, and the data from the server back to the client; T_{server} indicates the time between the receiving of the RPC by the server and its serving; finally, T_{disk} indicates the time needed to transfer the data from the disk to the server's memory.

In the considered example, a single read operation is converted into multiple server operations because we have assumed data to be laid out in the disk non-contiguously. Normally, file systems try to avoid this eventuality but this is not always possible due to file fragmentation. I/O performance is measured in terms of bandwidth by dividing the size of accessed data by the transfer time, as reported in Equation 2.2.

$$BW = \frac{S}{T_{IO}} \tag{2.2}$$

It is important to notice that I/O performance is highly dependent from the way data in the file is accessed. Ideally, we would like to access large chunks of contiguous data through a few requests, which translate into a minimum number of network and disk operations. Contrarily, a large number of small non-contiguous requests produces a higher number of network and disk operations. Because scientific codes often access data using many small non-contiguous requests, additional software layers are used to adapt the application I/O behaviour to the characteristics of the underlying storage system, allowing for better performance. These components are called middlewares and are positioned between the application and the parallel file system. Middlewares are reviewed in Section 2.4.

2.1.1 Parallel File Systems

In Linux systems, and more generally in any UNIX-like system, data is organized into files arranged in a directory tree, or namespace. In Linux everything is represented as a file, including physical I/O devices (keyboards, disks, printers, etc), sockets (active network connections), and even operating system statistics (through the */proc* file system [1]). Files in Linux are accessed through an interface based on the POSIX-IO standard. The standard defines operations to write and retrieve file data as well as metadata. Metadata operations are used to inspect and modify file attributes like access permission, creation time, size, and so on. These operations access the file's *inode* (i.e., the file system data structure representing the physical file) along with its attributes. Data operations allow users to store and retrieve raw data to and from the file; such data is contained into *data blocks*. Both inodes and data blocks are represented as contiguous byte ranges of size multiple of the disk sector, simply called blocks. The block is also the elementary transfer unit in the file system[2]. Data blocks belonging to the same file are referenced by its inode. Directories are special type of files that do not contain raw data but only key-value pairs. In each of these pairs the key is the name of a file inside the directory, while the value is the inode number of the file in the file system. Figure 2.3 shows the relationship between inodes and data blocks in a Linux file system.

As said, inodes have two sections, one containing file metadata or attributes, and one containing references to data blocks storing raw data. Because inodes and data blocks are of the same fixed size, commonly 4 KB, there is a limitation to the number of data blocks that can be directly referenced by an inode, and thus a limit to the file size. If we disregard the attributes and consider 4 bytes addresses, we have that a file cannot be bigger than 4 MB (1024×4 KB). For this reason inodes also reference data blocks indirectly through other blocks, the latter storing addresses instead of data. With a single level of indirection, for example, we can have an additional 4 GB ($1024 \times 1024 \times 4$ KB) of space, 4 TB with a double level of indirection.

Parallel file systems [Bra02] [SH02] [WUA+08] [CLI+00] [Hei02] differ from local file systems since instead of storing data into a single hard disk, they use several disks connected to remote servers. In most parallel file system designs data is partitioned and spread across many *I/O servers* (IOS), or *data servers* (DS), while metadata, including file mapping

[1] http://man7.org/linux/man-pages/man5/proc.5.html

[2] The typical sector size in a hard disk is 512 Bytes, while the most common block size is 4 KB (or 8 sectors) for local file systems.

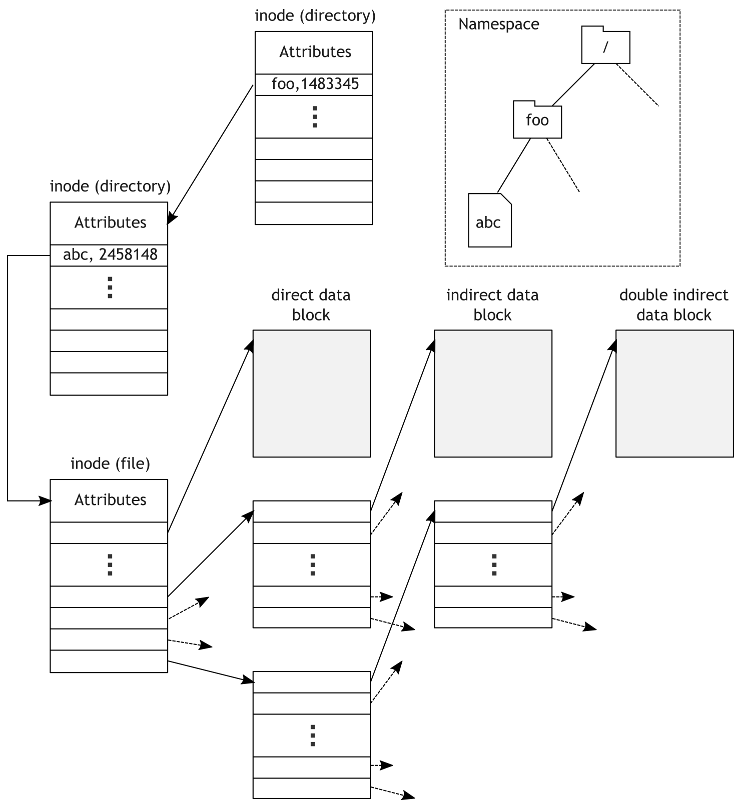


Figure 2.3: Simple namespace example and corresponding representation using inodes and data blocks in the file system.

information, is stored in one or more *metadata servers* (MDSs). The division between data and metadata allows for a specialization of the two services and a better efficiency of the system. The reason is that data and metadata access patterns have different characteristics, which can be better addressed by dedicated hardware and software solutions. For example, MDSs are designed to perform a large number of small metadata queries and updates. Therefore, besides hard disks, some times they also contain solid state drives that provide better performance for small random accesses; additionally, they are also equipped with larger DRAM memories and more CPU cores. I/O servers, on the other hand, are designed for sustaining high aggregated bandwidth for bulk I/O operations and thus have less demanding CPU requirements. Figure 2.4 shows the high-level architecture of a parallel file system.

Many parallel file systems ultimately rely on local file system installations in the different servers to work. For this reason a distributed file comprises several local files in metadata and data servers. A file, or metadata object,

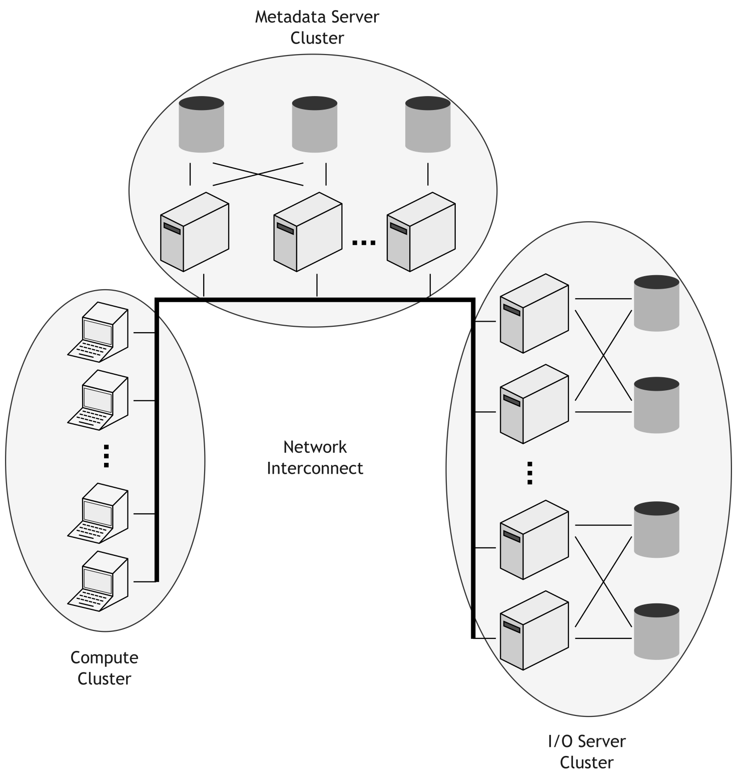


Figure 2.4: High-level architecture of a parallel file system.

in the metadata server is equivalent to an inode in a local file system. It stores all the classical file attributes, plus additional mapping information. Such information allows to determine the location of data blocks into IOSs. Mapping attributes are normally represented by three parameters; the first identifies the data server from which the file starts; the second the number of data servers used to store the file, also called *striping factor* or *stripe count*; the third the file system block size or *stripe size*.

In order to uniquely locate blocks in the storage system one additional parameter is needed. This identifies the strategy used to distribute data blocks among the available servers. The simplest and most common strategy used is the round robin distribution. I/O servers store data blocks using files, or data objects; for each distributed file, every I/O server has one data object containing a certain number of data blocks, either zero or greater than zero.

As said, parallel file systems can use several metadata servers. This allows for better scalability of the metadata service, since the workload can be distributed among several machines. Nevertheless, the file system can still suffer poor metadata performance if many processes access the same directory or file at the same time. This is a very common problem in HPC applications that sometime need to create a large number of files in the same directory, as we will see later. Although metadata distribution can be useful to sustain performance at scale, it also complicates the file system design and creates new problems. One of such problems comes from the fact that distributed metadata updates, involving several servers, become possible. Consider Figure 2.5 as an example. The `RENAME` of $dir_1/dir_2/dir_4$ to $dir_1/dir_3/dir_4$ involves four servers: MDS_1, MDS_2, MDS_3 and MDS_4. Indeed, we need to remove the metadata object for dir_4 from MDS_2 and add it to MDS_4. Similarly, we need to remove dir_4's reference from dir_2 and add it to dir_3.

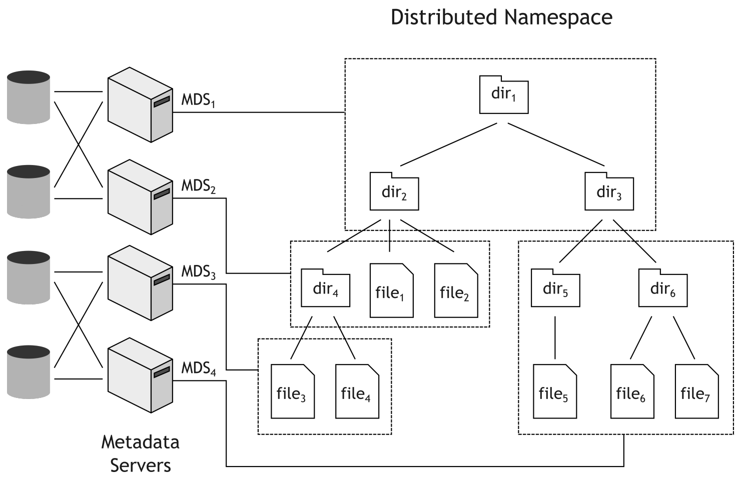


Figure 2.5: Example of namespace distributed across four metadata servers.

If distributed metadata updates are not handled properly, a server crash may lead to inconsistencies. To avoid this eventuality file systems employ appropriate distributed commitment protocols. The drawback this time is that these protocols involve the exchange of a large number of network messages and synchronous writes to disk in order to safely commit a metadata update [SC90] [GL06], thus impacting performance. Distributed metadata services in parallel file systems have been widely studied [ZK01] [FXM04] [MHH+05] [SSH+10] [CGNB12], but there is no best solution that can fit every use case.

Consistency Semantics

All information processing systems are composed by a set of hardware and software components that communicate with each other through electric signals and messages. The simplest computer, for example, is composed by at least one CPU, some amount of DRAM memory, one hard disk and the operating system. The CPU executes programs' instructions, loads data from the disk into the DRAM, modifies data in the DRAM and possibly writes it back to the disk. Assuming that the executed program is coded properly and does not contain any bug, we want to make sure that its execution in the computer always generates the same result and that this result is the correct one.

We focus our attention on the disk subsystem and more specifically on the file system software responsible for storing and retrieving data. In this case we want to make sure that the subset of operations involving I/O does not alter the correct program execution; that is, if the program completes it generates the right result. In the case of the file system, and generally of every other software component, this boils down to the identification of a set of invariants and the enforcement of the condition that these are respected before and after every I/O operation. These conditions translate into a list of semantic requirements for the file system's operations.

A very common semantic model, especially in relational databases, is the ACID model [Gra81] [WSSZ07]. ACID stands for *atomicity*, *consistency*, *isolation* and *durability*. Atomicity means that if a file system operation is composed by multiple steps, either all of them complete or neither of them completes; consistency means that every operation takes the file system from one valid state to another; isolation means that the concurrent execution of multiple file system operations is equivalent to their sequential execution and; finally, durability means that once an operation has completed its effects cannot be lost, even in the presence of a system crash.

The POSIX-IO standard only enforces atomicity and consistency. Durability is not considered because it would result in serious performance penalties [WSSZ07]. For example, in a durable file system every write operation should be carried out synchronously. Because disks are orders of magnitude slower than CPUs, the program would spend most of its time waiting on I/O, thus making little progresses. To avoid this, data and metadata updates are applied to DRAM and are committed to disk only explicitly by the user through the invocation of either `sync()` or `close()` of the file. Isolation is also not considered because it would require the file system to lock every data structure that can be accessed concurrently, even if this does not violate any invariant. However, the lack of isolation leaves programs open to vulnerabilities due to race conditions. For example, if the execution of an operation in one program depends on the value of a certain variable that can be modified by another process after it was read, the program might perform an operation that relies on a condition that is no longer true. This is known as *Time-of-check-time-of-use* (TOCTOU) [WSSZ07] problem.

In a distributed system, like a parallel file system, atomicity and consistency requirements are particularly hard to enforce and require the implementation of special distributed locking mechanisms. Consistency, for example, requires that every write operation becomes immediately visible to every node accessing the file. This is equivalent to saying that all the caches in the different nodes have to be coherent with the data in the file. One problem in this sense comes from the so called *false sharing* of file system blocks. Because file systems write data at the block granularity, if two nodes write to the same block, even though the writes do not overlap, the last writer will end up overwriting the other node's data, causing data inconsistencies and wrong results. This condition is shown in Figure 2.6 and is normally solved by implementing locks at the block granularity. False sharing also causes performance degradation because it forces the file system to perform read-modify-write to partially update a block. Lock contention and extra read overhead are important aspects that users have to keep in mind when reaching out for high performance data access in their codes.

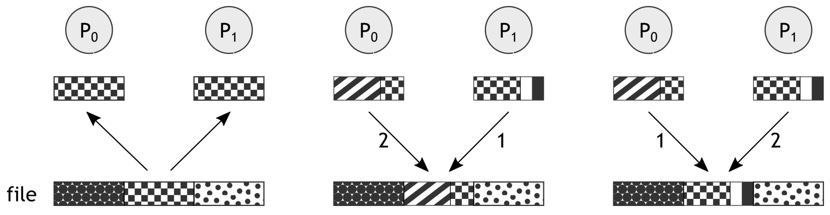


Figure 2.6: In this case false sharing of block two in the file can lead to two different outcomes. In the first case process P_1 writes before P_0, in the second case the order is inverted.

The locking strategy is implemented differently by different file systems. Nevertheless, in order to reduce overhead and improve performance, most solutions use an extent based locking [Bra02] [SH02]. In the extent based locking the file system client requesting exclusive access to a file region, is granted a lock for a larger extent, multiple of the block size. Because programs do not access only one part of the file, the extent based approach allows them to avoid acquiring further locks from the lock manager for future accesses.

Atomicity is frequently enforced through the *write ahead logging* (WAL) of file system metadata, also called journaling. In WAL metadata updates are not committed to the disk immediately but are instead logged into a journal. Every once in a while, or when the user explicitly requests it (e.g., through `sync()`), data is written to the file and afterwards the journal is forced to disk as well. For example, if the user creates a new file, the journal will contain two entries, one for the allocation of the new inode and one to add a reference to the inode in the parent directory (link). Upon a system crash and restart the file system inspects the log to check whether the create operation has completed successfully (all the relevant data structures have been updated) or has completed only partially. If the operation has completed the corresponding entries can be removed from the journal, while if the operation has completed partially the file system can decide whether to complete it or undo the partial changes. (In any case the file system is always recovered to a consistent state. While WAL is used in many file systems, other solutions are also possible like, for example, *copy-on-write*.) The advantage of using journaling is that in the eventuality of a crash, upon restart, the file system does not need to scan all the disk to verify if any of its data structures has been left into an inconsistent state, which is a time consuming operation that in large disks can require up to several hours.

2.1.2 I/O Gap

Moore's law has been driving technological advancements in integrated circuits for the past five decades, almost doubling density and speed of microprocessor every two years [Mac11]. Comparable technological advancements in hard disk drives (HDDs) have been achieved for capacity increase [GH96] [NIS99] but not for latency reduction. As shown in Figure 2.7, areal density has been increasing steadily at a rate of 100% every year since 1997. This trend slowed down to 30% in 2001 to start growing again at 50% in 2005. Today, the areal density increase is 40% per year [MHS15]. Unlike density, latency improvements are limited by the speed at which mechanical parts in the disk can move. Several research works have tried to alleviate the impact of mechanical latency on performance by proposing more efficient strategies to schedule disk requests [JW91] [WGP94] [ID01].

The technological gap between CPUs and hard disks, also known as *I/O gap*, has caused performance of HDDs to fall far behind compute components, imposing time penalties of millions of CPU cycles on applications performing I/O operations. This problem is particularly relevant in HPC environments because the performance of compute components increases much faster than storage, as shown in Table 2.1 [ASC10]. The table compares the characteristics of a typical 2010 supercomputer with the characteristics of future Exascale machines, which were predicted to be

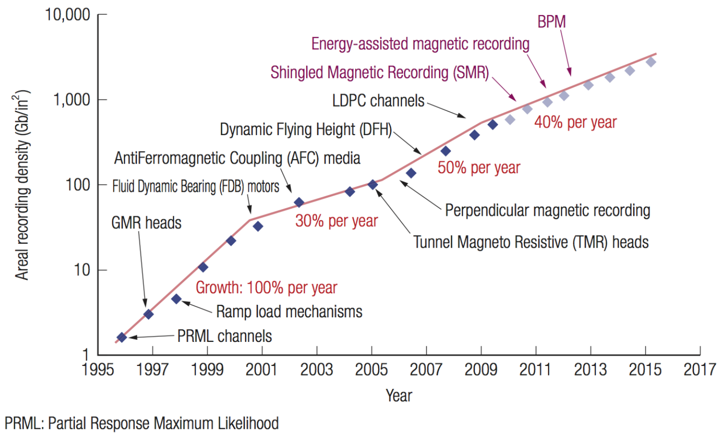


Figure 2.7: Recording density has been improved thanks to technological advancements in different fields including magnetic sensors and recording media [SFT$^+$09].

available in 2018. Exascale machines are not yet available and will not be available by the 2018 time frame. More recently, the U.S. *departmment of energy* (DOE) *advanced scientific computing research* (ASCR) initiative has worked on updating the requirements of HPC systems at Exascale for different areas of interest including *high energy physics* (HEP) [Phy15], *basic energy sciences* (BES) [Ene15], *fusion energy sciences* (FES) [Sci16a], *biological and environmental research* (BER) [ERA16] and *nuclear physics* (NP) [Sci16b] through the 2025 time frame. What has emerged from the updated requirement reviews is that initial estimates of hardware resources, as presented in the first ASCAC report [ASC10], are a coarse grain approximation coming from a static projection of 2010 systems to 2018. As an example, the pre-exascale 180 PFlops Aurora system that will be installed at the *Argonne leadership computing facility* (ALCF) in 2018 is expected to have an I/O bandwidth of only 1 TB/s. This bandwidth is aligned with currently installed systems, like the Cray Blue Water supercomputer. For this reason, although the projections in Table 2.1 are seven years old, here we assume they provide an upper bound coarse grain approximation that is valid for the development of this dissertation.

Table 2.1 estimates that peak performance for Exascale systems will increase by a factor of 500, while I/O bandwidth will only increase by a factor of 100. This will cause a farther widening of the I/O gap and lead to serious scalability limitations for large scale simulations. To address this problem scientists are investigating deep memory hierarchies as well as non-volatile byte addressable memory technologies

Table 2.1: Potential Exascale Computer Design for 2018 and its relationship to current HPC designs.

	2010	**2018**	**Factor Change**
System Peak	2 Pf/s	1 Ef/s	500
Power	6 MW	20 MW	3
System Memory	0.3 PB	10 PB	33
Node Performance	0.125 Gf/s	10 Tf/s	80
Node Memory BW	25 GB/s	400 GB/s	16
Node Concurrency	12 cpus	1,000 cpus	83
Interconnect BW	1.5 GB/s	50 GB/s	33
System Size (nodes)	20 K nodes	1 M nodes	50
Total Concurrency	225 K	1 B	4,444
Storage	15 PB	300 PB	20
I/O Bandwidth	0.2 TB/s	20 TB/s	100

[WAA13] [ZGF+15] [Bre08]. Some of these technologies, like flash based *non-volatile memory* (NVM) devices, are already used as storage accelerators in so called *burst buffers*. We will discuss memory technologies in Section 2.2 and bust buffers in Section 2.3.

2.1.3 Small I/O

HPC applications often exhibit irregular I/O patterns that are not handled efficiently by the parallel file systems. Indeed, most scientific codes comply to the *single program multiple data* (SPMD) model in the Flynn taxonomy. This means that a single program is executed concurrently by an ensemble of nodes in the cluster; each instance performs the same set of operations but on a different portion of a larger multi-dimensional domain. Because data is organized as sequence of blocks on disk, parallel I/O from the application to the different portions of the dataset might translate into a large number of, possibly small, non-contiguous requests that are notoriously serviced inefficiently by hard disk drives [NKP+96] [SR98].

This problem is also known as the *small I/O problem*, and is typically addressed by specific software components called I/O middlewares.

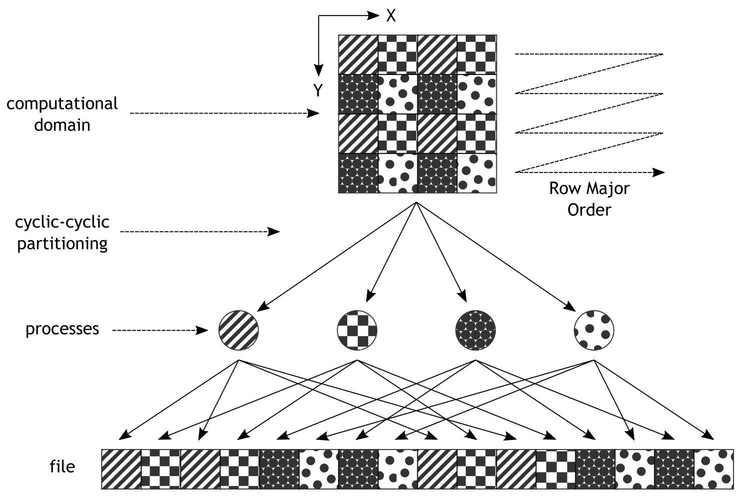


Figure 2.8: Example of two-dimensional domain partitioning in parallel applications. The original domain is divided among four processes using a cyclic-cyclic partitioning strategy. Each process in the application performs its tasks on the data and then writes the results to a shared file concurrently.

To better understand why small I/O is a problem consider a simple two-dimensional dataset as show in Figure 2.8. In the example, the original domain is divided into smaller sub-domains which are afterwards assigned to the available processes for computation using a cyclic-cyclic partitioning strategy [dRBC93]; that is, the domain is first divided into four blocks and inside each of these data is assigned to the available processes cyclically[3]. Every process performs its tasks and then concurrently writes the results into a shared file using the row-major order[4].

The row-major mapping of the original domain onto the file destroys the original locality of data and because I/O is performed independently, without any coordination among the processes, the file system cannot guarantee that accesses will be served in an ordered way by increasing offset. On the contrary, requests will most likely arrive out of order, generating an increased seek activity in the disks, stalling the application on I/O for a long time. Moreover, because the size of the sub-domains can be very small, disks might spend most of the time seeking from one point

[3]Many other domain partitioning strategies are possible and each of these reflects differently on the access pattern characteristics of the application.

[4]The input domain is scanned row by row and mapped onto a single dimensional representation in the file.

of the file to the other, underutilizing the available throughput. Another deleterious effect of uncoordinated I/O is the false sharing of file system blocks, that results in expensive communication between the application and the file system's lock manager to acquire and release exclusive access rights to the file regions of interest. This produces the serialization of I/O operations and further increases the I/O stall time.

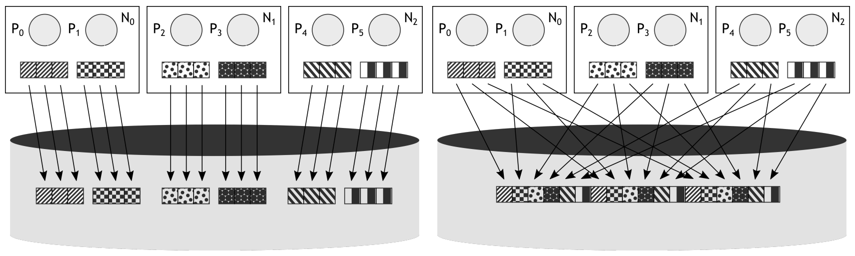
As we will see in the rest of the chapter this problem is commonly addressed by reorganizing and consolidating I/O requests generated by file system clients (i.e., application's processes) to match the logical distribution of data blocks in the file. In this way a smaller number of large sequential requests is presented to the disks, also minimizing file system block contention and ultimately I/O stalls. Some of the middlewares that we will review address this problem by converting parallel I/O to the shared file into parallel I/O to independent files (PLFS [BGG+09]). Others leverage global access pattern knowledge exposed by processes to make sure that I/O operations are intelligently coordinated (e.g., ROMIO [TC96], ADIOS [LKS+08]).

2.1.4 Checkpointing

Large scale HPC systems are subject to the failure of hardware components [SG06] [SG07] and to soft errors [MHH+05]. Such failures represent a threat for long time running applications that in such cases would need to restart their execution from scratch, loosing all the progresses made and wasting precious time and system resources. For this reason, parallel applications typically divide their execution into multiple phases of computation, interleaved by periods of intense write activity during which the execution context is pushed to stable storage, also called checkpoints. In the eventuality of a system failure, the application can pick up computation from the most recent checkpoint, thus saving time.

Clearly checkpointing is a viable technique but since I/O is expensive, applications should make sure they do not abuse it; that is, checkpoints should not be written more often than strictly necessary. Therefore, the checkpointing interval should be chosen to minimize the wall-clock execution time of the application. An estimate of the execution time as a function of the checkpoint interval is given by Daly [Dal06]. More recent works look at the severity of the failures and propose a multi-level checkpointing approach in which parallel file system checkpoints are extended with node local storage checkpoints [MBMdS10] [MBMS10]. This approach exploits local storage for less critical failures and relies on the parallel file system only for the most critical failures, that cannot reconstruct the requested checkpoint data from the local storage devices.

Due to their sensitivity to system failures, checkpointing has become one of the most predominant I/O patterns in HPC applications. Applications perform checkpointing using two different write strategies. In the first case, every process writes its state to a separate file, also called N to N pattern, or N-N for short. In the second case, every process writes its state to a single shared file, also called N to 1 pattern, or N-1 for short. Because applications need to guarantee that processes running on different physical nodes in the system progress together in their compute tasks, they use barriers (i.e., global synchronization points in the code) to avoid the implementation of expensive distributed synchronization algorithms [Lam78]. Unfortunately, when transferring application state to stable storage, the use of barriers just pushes the synchronization burden from the application to the parallel file system, that has to guarantee data consistency implementing mutual exclusion through locking.



(a) N-N pattern. (b) N-1 strided pattern.

Figure 2.9: Common checkpoint patterns in HPC.

The N-N and N-1 checkpointing patterns just described are shown in Figure 2.9. These try to address different performance bottlenecks in parallel file systems, but frequently they just trade one for another. For example, in large scale applications composed by thousands of concurrent processes, the adoption of a N-N pattern can alleviate the single shared file locking contention, but requires the creation in the same directory of a corresponding number of files to dump each process state. Parallel file systems typically partition the namespace across multiple metadata servers for better performance [PG11]. However, the namespace partitioning is often done at the mount point granularity. This causes N-N patterns to overwhelm metadata servers with a storm of file creates, degrading performance. The namespace partitioning also leads to distributed metadata operations that have to be handled carefully by the metadata protocol [SSH$^+$10] [CGNB12]. N-1 patterns overcome this problem by using a single shared file for checkpointing, but because scientific codes often write small amounts of non-aligned data, this results into an increased contention on the file and, correspondingly, in the serialization of writes.

N-1 patterns are often preferred to N-N patterns because of their convenience. Indeed, data in a shared file is organized to replicate the original input domain layout and it does not depend on the number of checkpointing processes. As a result computation can be restarted from the checkpoint with an arbitrary number of processes; visualization tools that need to display partial results to users are also advantaged because they only need to access one file instead of reconstructing the original domain from many small segments.

So far we have implicitly talked about user-level checkpointing in which writes are directly performed by the application. Different user-level checkpoint libraries have been developed over the years [BGG$^+$09] [FWP09] [MBMS10] [AAC09] to improve performance. A system-level approach, that does not require the direct involvement of the application, is also possible [WMES07]. In this case, the entire memory of every node in the system is dumped to the file system at predetermined intervals. This is, of course, much more expensive because a larger amount of data and system information needs to be written as further studies have demonstrated [KGS$^+$16]. One additional challenge is the capturing of system information that includes descriptors of open sockets and files, as well as the state of transient network messages. A solution to the problem, that makes use of resource virtualization, has been proposed by Arya et al. [AGPC16]. However, user-level checkpointing still remains the preferred choice, especially when moving to extreme large scale systems.

2.2 Memory Technologies

Current compute systems are based on the Von Neumann architecture in which CPU and memory are integral components of the system while storage is relegated to peripheral and accessed through the I/O subsystem using many different layers of software (e.g., file system interface, block layer, device drivers). Because external storage typically relies on mechanical devices, such as hard disk drives, that are notoriously much slower than DRAM memories, I/O has become the main bottleneck for many applications. As a results there has been a huge effort in terms of research and development of dedicated parallel file systems and I/O middleware solutions that try to bridge the performance gap between compute and storage. Historically these solutions have worked out well for specific workloads but they are not resolutive for the problem. Furthermore, the I/O gap is widening and software components alone cannot address the requirements of future leadership class systems. Figure 2.10 shows the gap between the different memory components in today's systems.

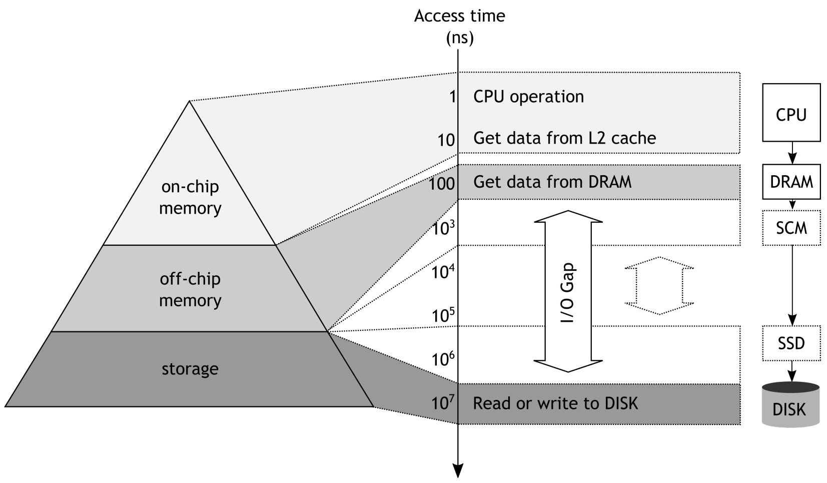


Figure 2.10: Classical memory hierarchy in a standard computer. In the figure the dashed boxes represent existing and emerging solid state devices like SSDs and SCMs.

As we can see from the figure, DRAM access times are in the order of hundreds of nanoseconds while disk access times are in the order of tens of milliseconds, a hundred thousand times higher than DRAM. In this section we look at existing and emerging mass memory technologies. Some of these technologies, like *flash memories*, have better performance than spinning disks but are still accessed using a block based interface and for this reason are a good candidate for implementing a faster storage tier between DRAM and HDDs. Other technologies, like *storage class memories* (SCMs), have performance that approach DRAM and are also byte addressable; therefore, they can bring the density and non-volatility of storage media into memory land, providing a good candidate to either replace or sit next to DRAMs.

2.2.1 Magnetic Disks

Hard disks are the most used mechanical storage components for persistently storing data in a computer. Data in hard disks is organized into sectors (elementary transfer units), dislocated on the surface of a magnetic disk (or platter). Hard disks can have many platters arranged into a stack, connected through a spindle powered by an electrical motor. Data in the disk is accessed through magnetic sensors, or heads, that are moved over the disk surface to read and write data using a set of mechanically actuated arms. Sectors are further organized into tracks (i.e., concentric strips that can be accessed with a single head positioning); the disk spins at high speed allowing the head to visit all the sectors in a track. Finally the set of tracks, from the different platters, that are located at the same distance from the spindle is called cylinder [RW94]. Figure 2.11 shows the described device geometry. Although hard disks could transfer data at the byte granularity, in order to achieve acceptable performance disks use sectors including multiple bytes (512 bytes most commonly); for this reason hard disks are also called block devices. Another feature of hard disks is that they allow data to be accessed in any order, making them another type of random access memory like DRAM.

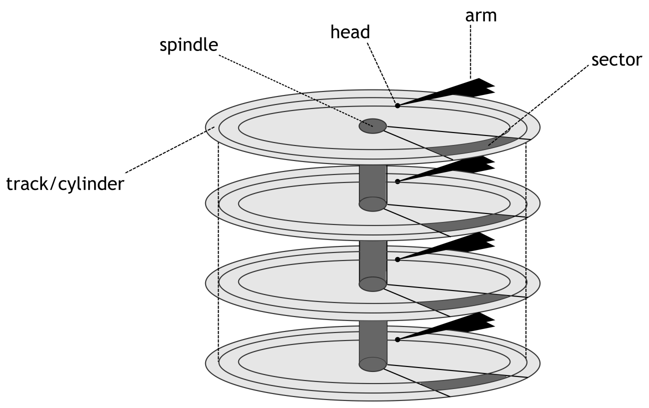


Figure 2.11: Schematic representation of the components of a hard disk drive.

Data access time in HDDs is governed by the mechanical parts that compose the device. The two main contribution in this sense are: **(a)** the *seek time*, accounting for the time needed to position the heads above the track where data will be read or written; and **(b)** the *rotational delay*, accounting for the rotation time needed to bring the desired sector under the disk head. Additionally there is also the *transfer time* accounting for the speed at which data can be read or written and the *queuing time*, accounting for how long a request waits before being served. The *controller time* is typically much smaller than the other contributions and is thus ignored here.

As an example, a typical hard disk for desktop applications can have an average seek time of 9 milliseconds, a rotational speed of 7200 RPMs (120 revolutions per second), and a sustained transfer rate of up to 300 MB/s

[5]. We can determine the average rotational delay from the inverse of the rotational speed as $T_{rot} = \frac{1}{120} \times \frac{1}{2} = 4.16ms$. The transfer time depends on how long it takes for every sector to transit under the disk head. This figure can be determined as $T_{trans} = \frac{512B/sect}{300MB/s} = 1.62\mu s/sect$. If we ignore the queuing time and assume every request is served immediately, the average access time for transferring 2048 sectors ($N_{sect} = 2048$), or 1 MB, in the considered disk is 16.4 ms, as given by plugging the previous values into Equations 2.3.

$$T_{disk} = T_{seek} + T_{rot} + N_{sect} \times T_{trans} \tag{2.3}$$

The result of 16.4 ms corresponds to a throughput of about 60 MB/s, that is much lower than the 300 MB/s reported by the manufacturer. This is because 300 MB/s is the maximum throughput that can be achieved by the drive when reading sequentially. By transferring a large data segment from the drive the impact of head re-positioning can be minimized, on the contrary transferring many small data segments involves several head movements that end up dominating the transfer time.

Equation 2.3 can be replaced into Equation 2.1 to determine the total average data access time in a distributed storage system.

2.2.2 Solid State Drives

Hard disk drives have dominated the mass storage device market for decades. In 1984 Fujio Masuoka invented the *flash* technology [MAI+84] that today is at the base of modern *solid state drives* (SSDs). Unlike HDDs, solid state drives have no moving mechanical parts. Because of this characteristic SSDs are faster and more reliable than HDDs. However, as we will see, the flash technology makes the design of SSDs a challenge.

Data in SSDs is stored inside flash cells. A flash cell can be built using a floating gate CMOS transistor, which represents information as electrical charge trapped inside the dielectric between the channel and the gate. There are different types of cells based on how many bits are stored. *single level cells* (SLCs) store only one bit, *multi level cells* (MLCs) store two bits, while *triple level cells* (TLCs) store three bits. SLCs are more reliable and have better performance among all and thus are also more expensive. Flash cells are organized into larger units called *pages*,

[5]Values reported for the Seagate Barracuda SATA drive, available at http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369h.pdf.

typically 8 or 16 KB in size, and pages are further grouped into *erasure blocks*, using a NAND gate configuration. Data can be read and written from and to the flash chip at the page granularity. Nevertheless, a page cannot be re-written unless it is erased first. Unfortunately, the flash chip cannot erase single pages but has instead to erase all the pages inside a block.

Normally read operations are quite fast because the value of the electrical charge in the cell can be retrieved easily (in the order of tens of microseconds). Write operations, on the other hand, are typically more expensive because the programming of the cell takes more time (in the order of hundreds of microseconds). Additionally, at some point, block need to be erased and this causes a wear out of the device that has a limited number of erase/programming cycles. These two elements are important aspects that have to be considered when designing a solid state drive based on the flash technology. Figure 2.12 shows the high-level architecture of a flash based storage device as we have just described it.

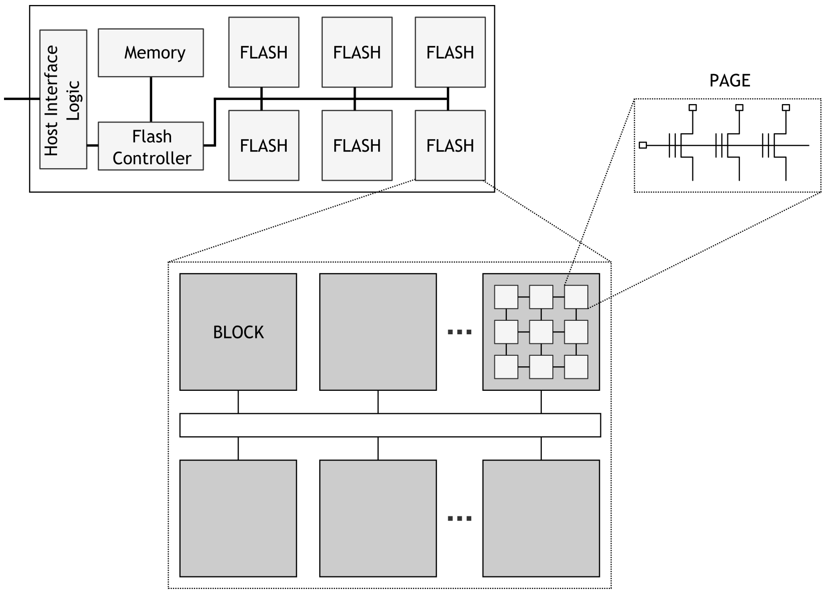


Figure 2.12: High-level architecture of a flash based solid state drive.

A solid state drive is composed of many flash chips, some memory for temporarily storing data read and written to the device, a flash controller and a host interface (SATA for commodity SSDs, PCIe for enterprise SSDs). Most of the design effort in this class of devices goes into the flash controller, also called *flash translation layer*, or FTL. The FTL makes the flash chips in the SSD appear externally like a block device, exactly like a standard HDD. This means that users can still reference data in the device using logical block numbers and the FTL will convert, or map, these onto addresses in the flash chips.

When a page is updated, the FTL reads it from the target block, applies the changes in memory and writes the updated data to a new page. When all the pages in the block are full the FTL moves to a fresh block; this time, along with the updated page it also copies all the pages in the old block (while this is selected to be erased later on). Since the erasure of the block is the most time consuming operation (in the order of milliseconds), blocks are not erased immediately but are instead marked for erasure. Subsequently, a garbage collection algorithm gathers all marked blocks and erases them. During the update of a page it is also important to decide where data from one block is moved. Because blocks have a finite number of erase/program cycles, it is crucial that all of them receive a similar number of erase/program operations. This is called wear leveling and is also responsibility of the FTL. In this sense it is worth to mention the work done by Margaglia and Brinkmann to improve SSDs endurance (as well as performance) by doing write in place updates to pages [MB15].

We have seen as for HDDs read and write performance are quite similar and we have also seen that for flash chips this is not the case, with write operations being ten times slower than reads (tens of microseconds for reads against hundreds of microseconds for programs). This performance issue is commonly addressed by the SSD writing in parallel to multiple flash chips. However, the FTL logic and the disk block interface limit the device performance. As an example, a modern Samsung 6 Gb/s SATA SSD can have sequential read and write performance of 540 MB/s and 520 MB/s, respectively[6], while a PCIe Samsung SSD can have sequential read and write performance of 3,500 MB/s and 2,100 MB/s respectively[7]. The performance difference is due to the higher transfer rates of the PCIe interface.

Because PCIe SSDs can perform an order of magnitude better than HDDs but are still inferior to DRAM, they can be used as additional block based cache layer placed between DRAM and hard disks. This is exactly the type of application these devices are finding in high performance storage systems.

[6]Values reported for the Samsung 805 Evo SSD, available at http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_850_EVO_Data_Sheet_Rev_3_1.pdf.

[7]Values reported for the Samsung 960 SSD, available at http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_960_PRO_Data_Sheet_Rev_1_1.pdf.

2.2.3 Storage Class Memories

Storage class memories promise to bring together the characteristics of permanent storage (capacity and non-volatility) and memory (low latency and byte addressability) in the same device, providing at the same time better power efficiency than SRAM and DRAM technologies. Indeed, conventional main memories are volatile and require a permanent source of energy to maintain, or periodically restore, the integrity of the data. This last characteristic makes SCMs particularly attractive for the deployment in large scale leadership class machines, that can have several Terabytes of main memory.

Different technological solutions have been proposed for storage class memory devices, including *spin-transfer torque magnetic RAM* (STT-MRAM) [WAA13], which provide access times comparable to DRAMs but are less dense; *resistive RAM* (RRAM) [ZGF$^+$15], which are a little slower than STT-MRAM but more dense; *3D XPoint*, which resulted from the collaboration between Micron and Intel; and *phase change memories* (PCMs) [Bre08]. SCM devices based on these technologies can be addressed at the cache line level like DRAMs and can thus be employed as *non-volatile main memories* (NVMMs) placed directly on the memory bus using a *non-volatile DIMM* (NVDIMM) interface [LIMB09].

The availability of NVMM opens up a new range of alternatives for implementing persistent data storage solutions. For example, so far file system developers had to rely on magnetic disks to store files. Because hard disks are slow block devices, disk based file systems interfaces and data consistency semantics have been defined accordingly. As we have seen in the introduction of this chapter, POSIX enforces atomicity and consistency but not durability. Durability is deliberately avoided because of the cost of committing updates to disk. Similarly, data accesses are typically block aligned in both offset and size to improve transfer rates. These design choices no longer hold for NVMM based file systems [XS16] which can access data fast and at the cache line level. Because data can be stored permanently in the NVMM, file systems no longer need to use the page cache to coalesce updates into large batches, thus avoiding the extra copy operation. Direct access to the NVMM is provided by the operating system using *direct access extensions* (DAX) or *execute in place* (XIP).

A problem with implementing file systems in main memory comes from the consistency requirements that are normally imposed upon update operations and that guarantee correctness of applications performing I/O. In this case, CPUs can rearrange the order of operations to improve performance, potentially breaking consistency in the case of a system failure. To avoid this, file systems in main memory have to explicitly enforce the flushing of CPU caches to respect ordering but, at the same time, adding considerable overhead. In order to ensure atomicity, NVMM based file systems have to employ journaling mechanisms that perform efficiently on the new hardware. In fact, although SCMs can effectively bridge the gap between storage and compute, they will just move the bottleneck from the device to the system software and applications, which performance will be no longer dominated by storage latency [KGW16].

2.3 Caching

In the previous section we have introduced hard disk drives and discussed the performance gap between storage and compute. We have also seen that high performance storage systems improve performance of disk based file systems by providing independent parallel data paths from clients to storage devices. Another effective way to improve performance is to exploit common access pattern characteristics to intelligently move data from storage to main memory. Many computer programs exhibit data locality characteristics both in space and time. For example, during normal execution the CPU will load instructions from a binary file stored on disk and execute them in sequence, from the first to the last. Because file systems try to minimize disk access time by placing data contiguously on the device, the corresponding instructions are also spatially contiguous. Moreover, most programs repeatedly perform the same set of instructions inside a loop. These characteristics can be exploited by the operating system to intelligently move data from disk into a region of DRAM to serve it more rapidly to applications, thus reducing I/O stall time. To this end the Linux kernel uses a dedicated area of main memory called *page cache* [BC08]. The page cache buffers data from storage devices using the file system block granularity and normally adopts the same block size also for pages.

In order to understand how the page cache can improve I/O performance let us consider a simple read example. When the kernel receives a read call, it first tries to locate the requested data in the page cache. If data is not in the page cache, the read call causes a *page miss*, which triggers a disk operation to retrieve it from the external storage device. To handle the disk operation, assuming there is enough space available in the cache, the kernel allocates a new page and instructs the DMA controller to fill it with data from disk. When the DMA controller has completed the transfer, it notifies the kernel using an interrupt. At this point the kernel copies the freshly read data from the page cache into a user buffer in the program address space and returns control to it. The additional copy from page cache to user buffers adds latency but because disk I/O is much more expensive than a memory copy, the cost is hidden in the total data transfer. Once data is in the page cache, future references to it can be served faster without involving the external storage device. Similarly, the cache can be also used to hide disk access to write operations. In this case data updates are applied to the page cache and are committed to disk later on, while control is immediately returned to the application. Cached data can be committed to disk either explicitly, using an appropriate system call (e.g., `flush()`) or by the kernel after some time has elapsed. The length of this interval depends on several parameters, like the amount of available memory, the number of *dirty pages* (i.e., the number of uncommitted updates in the cache), and so on.

File caching is effective in improving performance of local and distributed applications by hiding to them disk latency. However, caching is typically performed at the kernel level and thus does not account for global context in distributed codes. Parallel applications have multiple processes that are executed simultaneously in different compute nodes and access the same file concurrently. In this case if caching is done independently by every process, the burden of enforcing cache coherency is pushed to the parallel file system which has to grant exclusive locks to processes, ultimately serializing data access. These processes could instead collaborate to cache data as they were a single program running on the same physical machine. The advantage in this case is that file system clients can manage data consistency more efficiently without involving the parallel file system. Moreover, by using global context, overlapping accesses can be filtered to eliminate duplicated data in the cache. Liao et al. [kLCC$^+$05] have implemented a distributed caching scheme, called collective caching, in the MPI-IO layer [mpi12]. The proposed scheme takes advantage of the global application knowledge to efficiently buffer data at the user level. Because each file block can be cached only by one client this approach avoids false sharing and thus reduces the interaction between the application and the file system lock manager.

2.3.1 Readahead and LRU

In read patterns the locality of reference in the data can be exploited in two ways. First, the kernel can extend the size of a read request to transfer a little bit more data than originally requested, in a process called *readahead*; because the probability of this data being accessed next is high, a future reference to it can be quickly served from the page cache, completely hiding disk latency to the program. Second, because of the cyclic nature of programs, when cache space is scarce pages to be removed, or *evicted*, from the cache are selected among the *last recently used* (LRU).

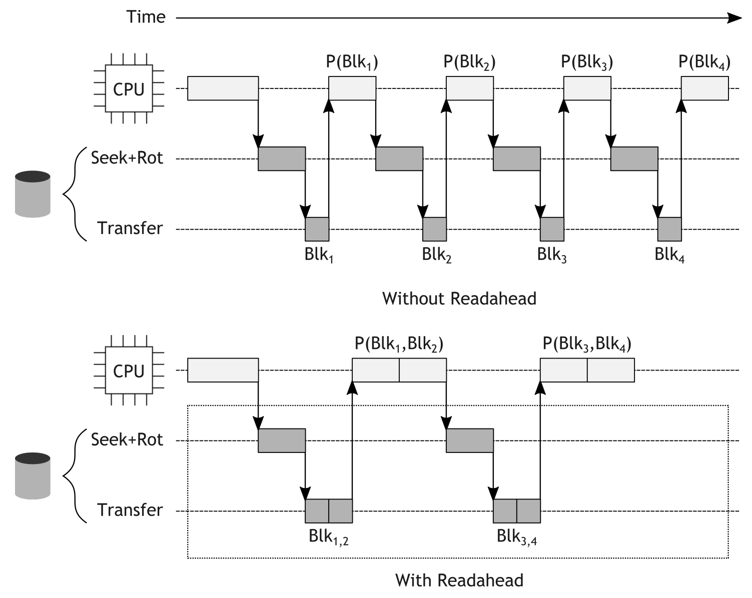


Figure 2.13: Effect of readahead on a program that reads sequentially blocks of data from a file in the disk. In the example, every data block Blk_i is read and afterwards processed $P(Blk_i)$ by the application. Readahead can merge the reading of two adjacent blocks amortizing the disk latency.

To further clarify the concept of readahead let us consider the example shown in Figure 2.13. In the example, the disk access latency is broken down into two components: the head positioning, given by the sum of seek time and rotational delay, and the data transfer time. Because readahead can enlarge the request size, two blocks of data can be read with a single head positioning, saving two seek operations to the application.

Although locality based heuristics can be useful to improve the performance of many programs, they cannot be applied to all the use cases. For example, codes might exhibit non-sequential file access patterns. A readhead strategy in this case would bring into the page cache data that has little chances to be referenced. Not only this does not improve performance but can have harmful effects on the overall system efficiency. In fact, the extra requested data is likely to steal disk bandwidth from other requests, and since the physical amount of memory in the system is limited, it will pollute the cache, possibly causing the eviction of more valuable data that will have to be fetched again from the disk later on. However, a locality based strategy that takes into account the global application access pattern could be beneficial for some parallel codes. As we have introduced in Section 2.1, when talking about the small I/O problem, scientific codes often access data using a regular pattern that can be easily exploited to effectively bring data into the cache. This observation is explored by Chen and Roth [CR10] whom show that for strided patterns in the pio-benchmark [Sho03] locality based heuristics can be still useful when applied to the application's processes as a whole instead of considering them independently.

2.3.2 Prefetching

In normal working conditions, data is transferred from disk to page cache in response to a page miss, triggered by a read call. Therefore, disk latency can only be hidden to following page references while the first access will always suffer the full I/O latency. Readahead can hide disk latency for data that is spatially local to the current request but is harmful to the performance of applications that do not exhibit such locality. For these applications readahead is typically disabled. To overcome such limitations more sophisticated caching strategies have been proposed.

Prefetching is a proactive caching mechanism used to completely hide disk latency to read operations. In this case, data from disk is brought into the page cache before it is actually referenced by a read call and, when needed, is served to the application from the cache instead of the disk. The readahead strategy previously described is in fact a special case of prefetching that relies on data locality. In readahead the prefetching request is piggy backed onto the original read operation, triggered by a page miss, and targets data that is contiguous to the originally request in disk [BC08].

In this thesis we focus on *guided prefetching* strategies because, unlike readahead, data can be fetched at any time and from anywhere in the file. However, when and how much data to prefetch become important parameters that need to be carefully evaluated. Indeed, prefetching has to be performed at the right moment and only for the right amount of data in order to bring improvements to the run-time, especially in systems with limited available memory. Prefetching too much data, or prefetching it too early, may cause more urgently needed data to be evicted from the cache, forcing the application to fetch it from disk again later on. Contrarily, prefetching data too late adds unnecessary overhead associated to the processing of the corresponding system call by the operating system and might even fetch data that was previously evicted because no longer needed.

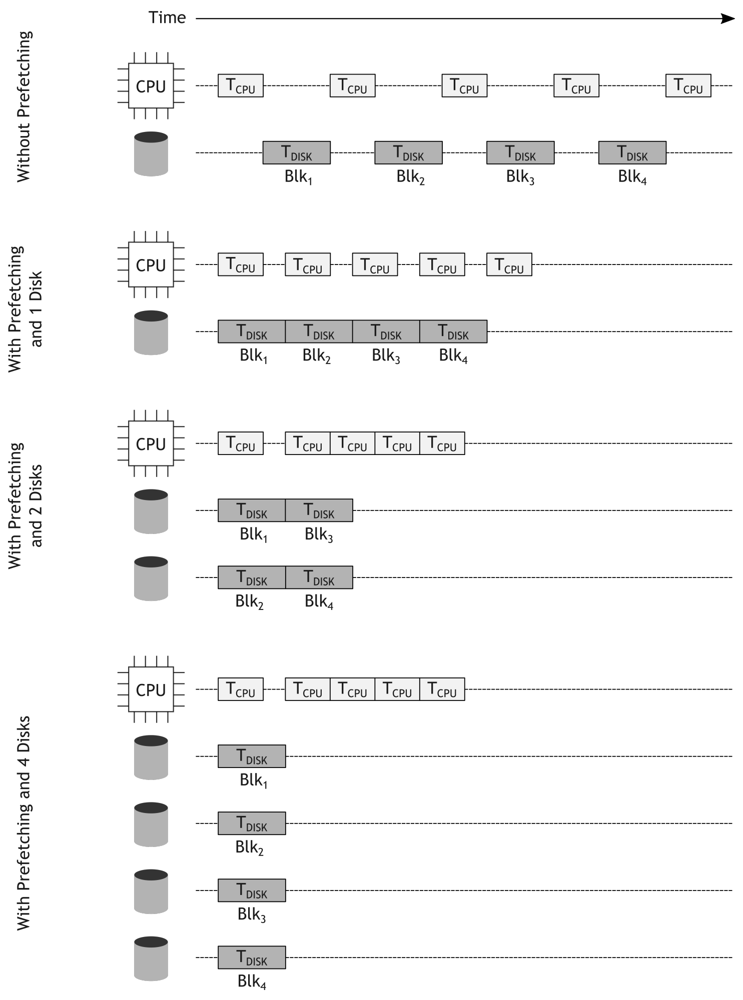

Time
Without Prefetching
With Prefetching and 1 Disk
With Prefetching and 2 Disks
With Prefetching and 4 Disks
CPU
T_CPU
T_DISK
Blk1
Blk2
Blk3
Blk4

Figure 2.14: Effect of increased storage bandwidth on prefetching performance.

Disk bandwidth and memory capacity directly affect prefetching performance. To understand how consider the examples reported in Figure 2.14. We have an application that alternates phases of compute, at the end of which some data is read from disk, to phases of data transfer from disk to memory. The performance baseline for the run-time is given by the first case, in which prefetching is disabled. The remaining three cases show how run-time behaves when prefetching data from one or more disks. In the first of the three only one disk is used and we can observe that, because the application is I/O bound, the minimum run-time is limited by the total disk latency. Increasing the number of disks from one to two doubles the bandwidth and thus the number of blocks that can be transferred at the same time. In this case disk latency can be completely hidden to the application, which progresses with no I/O stalls. Further increasing the number of disks from two to four does not add any further benefit because there is no latency left to hide. In general, for disk bound applications like the one considered, disk accesses can be completely hidden through prefetching if data is distributed appropriately across, at least, $\left\lceil \frac{T_{disk}}{T_{CPU}} \right\rceil$ disks [CGML01].

Although higher disk bandwidth can improve prefetching performance by allowing more data to be retrieved in parallel from the available devices, final performance also depends on the amount of cache space; it does not matter if two blocks can be prefetched at the same time, unless there is enough space in the cache to host both of them. If cache space is not enough to hold the two blocks, prefetching can degrade I/O performance. This eventuality is shown in Figure 2.15. When the application begins execution, it immediately issues to the kernel a prefetching request that covers Blk_1 and Blk_2. The kernel receives the request and initiates two disk transfers, one per block. Assuming Blk_1 is requested first, the kernel allocates a new page and initiates a disk operation to fill it, then it moves to Blk_2 and tries to do the same. However, because the only available page is already in use for Blk_1, it has to wait. Although the available disk bandwidth is enough to fetch two blocks at the same time, the lack of memory space results in the serialization of I/O at the page cache level. Moreover, when Blk_1 is finally in the cache, its page is evicted to transfer Blk_2. While Blk_2 is being fetched, the application makes the actual reference to Blk_1, which was previously evicted and has to be fetched again after Blk_2. The same happens for Blk_3 and Blk_4 later on. This causes a continuous transfer of blocks between memory and disk that increases I/O activity and correspondingly the application execution time.

Guided prefetching strategies rely on applications access pattern knowledge to timely transfer the necessary file data from disk into the cache. This knowledge can be directly exploited by manually inserting prefetch calls in the right place inside the code. However, this requires manual intervention and increases both complexity and programming efforts. Moreover, programmers are often reluctant to make changes to codes that have taken years of development to mature to a satisfactory level. Sometimes, the source code is not even available and only the program binaries are at hand. This makes it impossible to manually insert any prefetch call into the code. To overcome these problems, automatic and dynamic history-based approaches to prefetching have been proposed. Automatic prefetching can be further divided into *static* and *speculative* strategies,

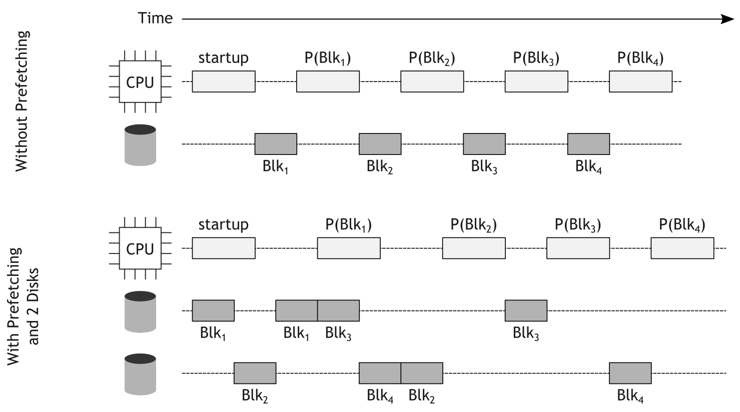


Figure 2.15: Effect of limited cache space on prefetching performance.

depending on how the prefetch requests are generated and added to the application. Dynamic history-based approaches, on the other hand, rely on the analysis of the I/O history to infer when prefetching should be performed; such solutions do not alter the application code or binary and delegate prefetch call issuing to an external software component, typically a middleware or the file system itself.

Automatic Prefetching

For automatic strategies it is important that prefetch requests are not issued using asynchronous read calls. This is necessary for two reasons; first, read calls are immediately converted by the operating system into data transfers from the storage device. However, automatically generated requests are not accurate because they do not consider the dynamically changing system parameters, like memory usage. Therefore, the operating system needs to have some flexibility to decide whether or not a request should be actually converted into a disk access; second, read operations are binding, which means that data fetched using a read operation is supposed to be consumed immediately and can be changed afterwards. Nevertheless, the basic assumption about prefetching is that data fetching and consumption can be decoupled so that data is brought into the cache ahead of time and consumed only later on, when it is referenced. For this reason prefetched data should not be mistakenly changed by the program if the prefetch call is moved ahead of a store to the same memory location by the compiler. To address these issues prefetch hint interfaces are employed. Hints are not immediately converted into data transfers and are non-binding, ensuring the correct behaviour of the application, no matter how far ahead in time they are submitted.

Static prefetching approaches exploit a code analyzer, typically a compiler, to extract access pattern information from the application and identify in which points of the code prefetch calls have to be inserted in order to hide the latency of a page fault triggered I/O operation[8]. Because static analysis does not consider dynamic system parameters, the compiler has to take an aggressive approach and inserts a prefetch call every time a page is reused. However, during execution the system might have sufficient memory to keep the target page in the cache between two consecutive accesses. In this situation, a prefetch call that is automatically converted into a system call for the OS would add no benefit to the execution time; on the contrary, it would add the overhead for the corresponding system call processing. For this reason static approaches combine compiler inserted hints with run-time and operating system support. The run-time and operating systems collaborate to exchange virtual memory information and, based on this, the run-time decides whether a prefetch call should be converted into a system call or not.

Mowry et al. [MDK96] and Brown et al. [BMK01] have used compiler inserted hints to improve performance of paged virtual memory for the applications contained in the NAS parallel benchmark suite [BBLS91]. The run-time and the operating systems share virtual memory information using a memory page, which is allocated at application start up when this registers with the OS. The shared memory page contains a bit-map describing used and unused pages in the address space. When the run-time system receives a prefetch call for a page that, according to the bit-map, is already in the cache it drops the request, avoiding to incur into unnecessary system call overhead. The bit-map is updated by both the run-time and the operating system as the application progresses.

In a multiprogramming environment, main memory is shared among several applications. If on one side, prefetching has the potential to improve the performance of out-of-core applications, on the other side it might negatively impact the performance of others, especially interactive codes that can be idle for a long time. For such codes inactive pages would be sacrificed to prefetch requests of more memory intensive codes, degrading their responsiveness. To improve memory management in multiprogramming environment, Brown and Mowry [BM00] have exploited *release* hints to explicitly sacrifice pages that are not expected to be reused again, or are expected to be reused but, due to insufficient memory, are going to be selected for eviction by the OS. Once again, because static analysis does not consider dynamic conditions, release hints are aggressively inserted for every page that is referenced multiple times. The run-time system

[8]Here we are implicitly considering automatic static prefetching for applications that access the disk using paged virtual memory and not explicit I/O operations.

compensates the static behaviour by dynamically adjusting the hints to the changing memory conditions. For example, if a release hint is issued for a reused page, the run-time checks whether the available memory is sufficient to keep the page in the cache. If it is, the release hint is discarded, otherwise it is forwarded to the operating system.

Although compiler inserted hints can automatically generate prefetch calls and insert them in the code, the required code analysis in the compiler has to make simplified assumptions that affect the accuracy of the technique.

Speculative prefetching issues prefetch calls to the underlying run-time or operating system by pre-executing application code. In order to move ahead in the code and find out about future reads, the program makes hypothesis about the value of dynamic variables. If the speculation is correct the technique can effectively predict what data will be accessed next and timely issue a hint for it. Speculative prefetching can be implemented using a compiler or, if the source code is not available, a binary modification tool. In both cases the original code is extended with a speculative thread that executes a modified copy of the original code, also called *shadow* code. In the shadow code read calls are replaced with prefetch hints and can be executed either using the spare CPU cycles resulting from an I/O stall of the main thread or using a further physical thread to run in parallel with the main thread.

One problem with speculative execution is that main and speculative threads share the same address space. Speculative execution might thus alter the original program behaviour by changing code or data used by the original program. This can cause side-effects that are visible outside the process, impacting the whole system behaviour, or can inadvertently produce data values that cause the program to deviate from normal execution.

Chang and Gibson [CG99] proposed *SpecHint* as binary modification tool to implement speculative execution. In SpecHint the shadow code is generated appropriately from the original binary to address the previously described problems and is executed during I/O stalls of the main thread using the spare CPU cycles. In SpecHint, load and store operations that involve shared variables are preceded by additional checks to make sure that memory accesses are redirected to a private copy of the original variable (copy-on-write). External side-effects are avoided by cleaning out the shadow code of every system call that is not a hint, a `fstat()`[9]

[9]Returns information about an open file.

or a `sbrk()`[10]. Finally, automatically inserted exception handlers make sure that inappropriate data values do not disturb normal execution. Whenever an exception is encountered speculative execution is halted and is resumed only when the original thread blocks on a read call. In order to ensure that speculative execution does not fall behind the main thread, references to issued hints are kept and compared with following read calls. If a read call is not paired by a previously issued prefetch, speculation is not working as expected and should be realigned with the original execution. SpecHint does not use any special run-time to handle hints but fully relies on operating system support, implemented by replacing the default buffer manager with the TIP manager [PGG$^+$95].

While SpecHint modifies the application's binary to pre-execute the shadow code and issue prefetch calls, Chen et al. [CBS$^+$08] have applied the same idea to MPI programs using a source-to-source compiler. Additionally, instead of exploiting unused spare CPU cycles resulting from I/O stalls, the parallel approach takes advantage of the additional physical threads to access data concurrently with the main thread. To avoid the pre-execution thread to change memory regions shared with the main thread a variable renaming technique is employed. This is more expensive in terms of memory requirements compared to the copy-on-write technique used in previous work but is not that critical in parallel machines with large memories. The problem of non-binding hints, previously described in the context of automatic prefetching, is solved by marking write operations as synchronization points.

Every time the pre-execution thread encounters a write call it waits for the main thread to arrive before continuing with execution. The drawback is that the amount of progress that can be made is limited. To alleviate this effect the authors observe that dependencies are critical only for RAW (Read After Write) operations. Therefore, when the speculating thread encounters a write call it registers its range (dirty range) and continues with execution. When the following read call is encountered its range is checked against the dirty range and if the two do not overlap speculation can continue; otherwise, delayed synchronization is performed. Hints are issued by the modified application to the underlying run-time system, that is represented by the ROMIO MPI-IO implementation. The used ROMIO version includes a collective caching infrastructure [kLCC$^+$05] and a prefetching library that manages prefetch calls. Because the cache is implemented in the run-time library, there is no required support from the operating system.

[10]Moves the top address of the heap up and down. This system call is used by libc functions such as malloc and free when allocating and deallocating dynamic memory.

Dynamic History-based Prefetching

While automatic prefetching mechanisms are suited for random access patterns that cannot be predicted by analyzing the history of the application, dynamic history-based prefetching can be applied to applications that exhibit some correlation in their access pattern. In order to answer the questions of when and how much data to prefetch, history based solutions accumulate I/O pattern knowledge and use it to take decisions. Such knowledge includes temporal behaviour, some times captured by looking at the inter-arrival time of read requests, as well as spatial behaviour, frequently captured by looking at what blocks in the file are referenced and in which order. The accumulated knowledge can then be used to train a neural network or to build a mathematical model to guide prefetching in the underlying run-time or operating system.

Tran and Reed proposed a time series modeling system called TsModeler [TR04]. TsModeler receives inter-arrival access pattern information from the underlying PPFS2 file system framework [SRF$^+$99], computes the model parameters and feeds them back into PPFS2. The file system is responsible for building the spatial model, interpreting the time series coefficients and combine the two models to guide data prefetching. In TsModeler time series analysis is performed to build a mathematical model of the inter-arrival time of read requests using an *autoregressive integrated moving average* (ARIMA) [New83] for the model's coefficients. The built model predicts when prefetching should be performed, typically before the actual request for the data arrives, but not too early to avoid other needed data to be evicted from the cache before it is referenced. PPFS2 employes Markov models to predict the spatial distribution of future accessed file blocks. Markov models can represent blocks as rows and columns of a matrix and the probability of accessing one block after the other as the matrix values. The Markov model can be either built online, for patterns that have data dependencies, or offline, for patterns that repeat.

He et al. [HBT$^+$13] have observed that many data intensive codes exhibit regular access patterns over multiple runs. To improve the performance of such patterns they have proposed a caching and prefetching framework built in the PLFS file system. The resulting PLFS modification can detect recurrent access patterns and use them to guide data prefetching through a dedicated thread. Byna et al. [BCS$^+$08] also targeted regular access patterns to build a mathematical representation of the application I/O behaviour, called I/O signature, and use this representation to guide prefetching during following runs. The prefetching and caching functionalities in this case are implemented at the MPI-IO level inside the ROMIO middleware.

The solutions discussed so far rely on low level access patterns information (i.e., offset and length pairs of I/O requests) to build the knowledge required to guide I/O prefetching; however, this type of low level information carries little or no information about the use of data made by the application. High-level I/O libraries, such as HDF5 [FCY99] and netCDF, provide better insights on the data usage pattern because they naturally carry semantic information including, temporal sequence of data accesses, combination of data (e.g., two variables that need to be fetched from disk to be combined and produce some results), and relations between data and computational phases (e.g., read of variables that are used in computation and produce a result variable that is written to disk). Basing on these observations, He et al. [HST12] have exploited high-level I/O behaviour in netCDF to build data dependency graphs that are afterwards used by their PnetCDF [LkLC$^+$03] framework to match the run-time I/O behaviour with the pre-stored patterns and perform data prefetching.

Manual Prefetching

Manual prefetching refers to the insertion of prefetch calls inside the application's source code by the programmer; additional support is normally provided by the run-time and operating system for managing the issued hints. Patterson et al. [PGG$^+$95] have proposed a hint interface and a customized buffer manager. The buffer manager has to balance the benefit of prefetching data blocks from disks with the cost of evicting existing blocks. Because the amount of memory at disposal is limited, the buffer manager delays block fetching just before the corresponding data is referenced. VanDeBogart et al. [VFK09] take advantage of larger memories to implement a greedier prefetching strategy that fetches as many blocks as possible in a single batch. The proposed *libprefetch* and kernel extensions advocate to maximize I/O throughput reducing seeks rather than overlapping disk and CPU activity to hide disk latency. Whenever possible, seek cost is reduced by filling the gap between two requests with the data in between, using a technique similar to the *data sieving* mechanism supported by some parallel I/O middleware.

In this thesis we pursue the same approach and take advantage of large primary memories in HPC compute nodes. We use programmer I/O knowledge to drive prefetching of data into the cache and use data sieving to minimize the effect of disk seeks on performance. We do this by relying on the file system prefetch interfaces provided by the Linux *virtual file* *system* and the GPFS file system. Unlike libprefetch that synchronously transfers a large number of blocks in a single batch from the disk we try, as much as possible, to issue prefetch calls to the kernel asynchronously using an additional thread.

2.3.3 Write Behind

In previous sections we have seen how caching can be useful to reduce I/O stalls when accessing frequently requested data. We have also seen how caching can be used to preemptively fetch disk data into memory, hiding partially or even completely I/O latency to applications. Although so far we have focused on read access patterns, caching can also be useful in write operations using a *write-behind* strategy. In write-behind data updates are not immediately committed to disk. In fact, as we have mentioned earlier when discussing file systems consistency semantics, synchronous writing of data to disk would impose a huge burden on applications that would end up spending most of their execution time waiting on I/O. Instead, updates are staged in the cache and marked to be transferred to disk at a later time. The flushing of cached data to disk can be enforced manually by users using an apposite file system function or triggered automatically by the operating system when a certain condition, or set of conditions, is met; for example, when amount of available memory falls under a predetermined threshold.

In the Linux kernel writes are applied to the corresponding page in the page cache. If the write targets an existing page, the page is updated and is marked as dirty; otherwise, a new page is allocated, data is written to it and finally it is marked as dirty. All the pages in the page cache are added to a LRU list. When memory usage grows, the kernel needs to reclaim old pages to recycle them. The victim pages are selected from the tail of the LRU list. If a victim page is dirty, the kernel triggers a disk transfer to flush its content to stable storage before recycling it. Alternatively, a `flush()` or `close()` system call issued for the corresponding file will cause all the dirty pages of the corresponding file to be written to disk immediately.

Write-behind can be extremely helpful in checkpointing workloads when used in combination with high performance storage devices like SSDs. The burst buffer concept, presented next, heavily relies on SSDs and write-behind to hide to applications the parallel file system performance issues with this type of access patterns.

2.3.4 Burst Buffers

The ever increasing need for high performance storage systems, able to sustain high data throughput for large scale simulations, has led to the design and implementation of complex storage clusters and parallel file systems. However, as previously discussed, many scientific applications interleave phases of computation with phases of intense write activity during which they dump their internal state to stable storage as a defensive mechanism against failures [WXH+04] [KGS+10]. The resulting workloads exhibit bursty characteristics that cause the underutilization of the storage system resources. In many cases, the storage system works at a fraction of its potential for most of the time [CHA+11b]. For this reason, the mere increase in I/O parallelism, achieved by adding more disks and storage servers, does not work.

I/O efficiency for checkpointing patterns can be improved by using appropriate software components called I/O middlewares. These components are placed between the application and the parallel file system and adapt the original I/O pattern, generated by the application, to the characteristics of the underlying storage system. I/O middlewares can also be combined with additional hardware resources to build storage systems that have reduced peak performance but better utilization profile. An interesting solution in this direction is represented by *burst buffers*. Burst buffers exploit high-performance storage devices, like SSDs, to absorb bursts of write activity at a specific level of the I/O architecture.

In Section 2.1 we have presented a simple HPC system architecture in which compute clients access the parallel file system by contacting directly I/O servers. Other supercomputer designs, like the Blue Gene systems, separate the compute cluster from the parallel file system using an intermediate set of I/O nodes. In this case clients perform I/O by sending their requests to the I/O nodes using a forwarding software [IRYB08]. The advantage in pushing I/O away from compute is that applications are less subject to operating system noise related to the I/O activity. Figure 2.16 shows the high-level architecture of the system just described.

Several proposed solutions [LCC+12] [WOW+14] deploy burst buffers on I/O nodes. These nodes stage user data locally into the dedicated SSD devices and allow applications to return to their compute faster. The burst buffer software (or middleware) takes responsibility for transferring data from the SSDs to the parallel file system in the background, using an approach similar to the write-behind caching strategy previously introduced. If implemented properly, this mechanism allows applications to overlap compute phases with I/O, drastically improving performance.

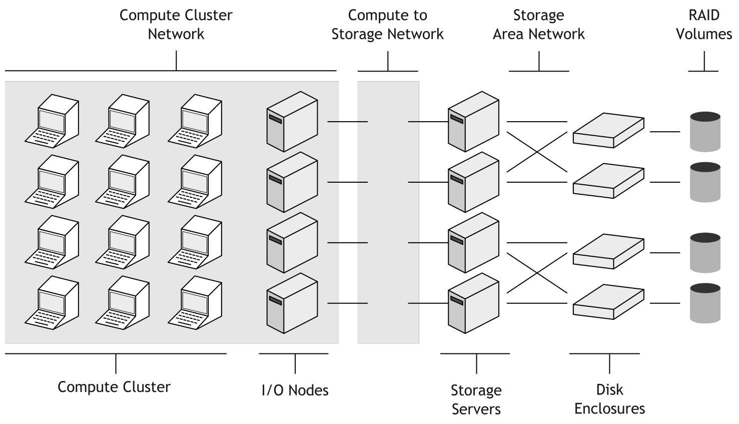


Figure 2.16: High-level architecture of an alternative HPC storage system design that uses I/O nodes to decouple compute from parallel file system access.

2.4 I/O Middlewares

I/O Middlewares are special software components used to adapt the access pattern of the application to the characteristic of the underlying file system, thus improving its efficiency and performance. Many libraries have been developed over the years to bridge the semantical gap between how data is represented at the application level and how it is finally stored in the file system. We have already mentioned in previous sections the importance of defensive checkpoint/restart patterns and how these represent a large portion of the workloads of today's HPC applications. In this section we present a more detailed view on the available software solutions that aim to, but are not limited to, improving checkpointing performance, how they work and what benefit they provide.

2.4.1 MPI-IO

The *message passing interface* (MPI) [mpi12] is the most used programming interface for parallel machines. In MPI processes exchange state and data using a set of message passing primitives that include point-to-point, one-sided and collective communication constructs. These are paired with a range of datatype primitives, called *derived datatypes*, that allow for compact representation of memory and file data layouts. MPI also defines a set of interfaces for parallel I/O, known as MPI-IO [mpi12], that rely on the MPI communication and derived datatype primitives to support efficient data access to networked file systems. With MPI-IO parallel codes can open and close files and write (or read) data to (or from) them in a scalable and efficient way.

The MPI-IO standard extends the classical POSIX-IO interface, historically used in multi-tenant machines, by relaxing its consistency semantics requirements that impose a synchronization burden on the parallel file system software. In fact, the semantics requirements defined in POSIX were specified for computational environments in which the file system had to serve I/O requests from users sharing the same physical hardware. Most parallel file systems used in HPC today adopt the same consistency semantics and are therefore said to be POSIX compliant.

The problem with applying the POSIX-IO semantics to parallel environments is that the file system has to implement additional mechanisms to enforce data consistency across the whole cluster. Such mechanisms are based on file locking and thus, if I/O is not properly handled by users, it results in the serialization of concurrent operations to the same file, voiding the benefit of having a parallel file system. MPI-IO focuses exactly on this aspect to make sure that I/O is properly orchestrated at the user level (or middleware level) and thus avoid conflicting operations to the same file regions in the first place.

MPI-IO defines different types of read and write interfaces, some of which are reported below:

- `MPI_File_read`, `MPI_File_read_at`
- `MPI_File_read_all`, `MPI_File_read_at_all`
- `MPI_File_write`, `MPI_File_write_at`
- `MPI_File_write_all`, `MPI_File_write_at_all`

There are two types of listed operations: *independent* and *collective*, which are further divided into *explicit* an *implicit* offset operations. Independent operations to the same file are trivially uncoordinated by the MPI-IO implementation. Collective operations, on the other hand, are coordinated and can benefit from this coordination in terms of improved I/O performance. Explicit offset operations require the user to pass the starting offset as a parameter, while implicit offset operations extract this information from the derived datatype, which in this case is also called *file view*.

The ROMIO Middleware

ROMIO is a popular implementation of the MPI-IO specifications developed at the Argonne National Laboratory and currently included in MPICH as well as OpenMPI and other MPI implementations. ROMIO provides MPI-IO functionalities for different file systems through the *abstract device I/O* interface [TGL96] (ADIO). Latest versions of ROMIO include support for Lustre [Yin08], GPFS [PTH$^+$00], PVFS and other parallel file systems through a dedicated ADIO driver.

Collective I/O is a parallel I/O strategy used to alleviate the small I/O problem, thus improving disk access performance in distributed environments. The idea at the base of collective I/O is to rearrange file access requests, at a certain level of the I/O stack, converting the original I/O pattern into an intermediate pattern that is served more efficiently by the underlying storage system. Depending on where the strategy is deployed in the I/O stack, either disks or file system clients, we talk about *disk directed* [Kot94] [SCJ$^+$95], or *two phase I/O* [dRBC93] [BdRC93]. In this thesis we focus on the two phase I/O implementation of collective I/O.

The *extended two phase algorithm* (ext2ph) [TC96] is an improved version of the original two phase I/O strategy. It exploits global application knowledge in parallel I/O to a shared file. This knowledge is used to build an aggregated view of the accessed region 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 a way such that only a subset of the available processes actually performs I/O. These I/O proxies, also called *aggregators*, 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) them.

This mechanism effectively adapts the I/O pattern to the characteristics of the file system, extracting maximum performance from it. Figure 2.17 exemplifies the basic two phase I/O mechanism just described. In the figure there are four processes, two of which play the role of aggregators. Two phase I/O proceeds in two stages: *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, also called *file domains*, that will be later accessed during the data I/O phase. In the example shown in Figure 2.17, the ext2ph algorithm converts the original column-block partitioning generated by the application into a

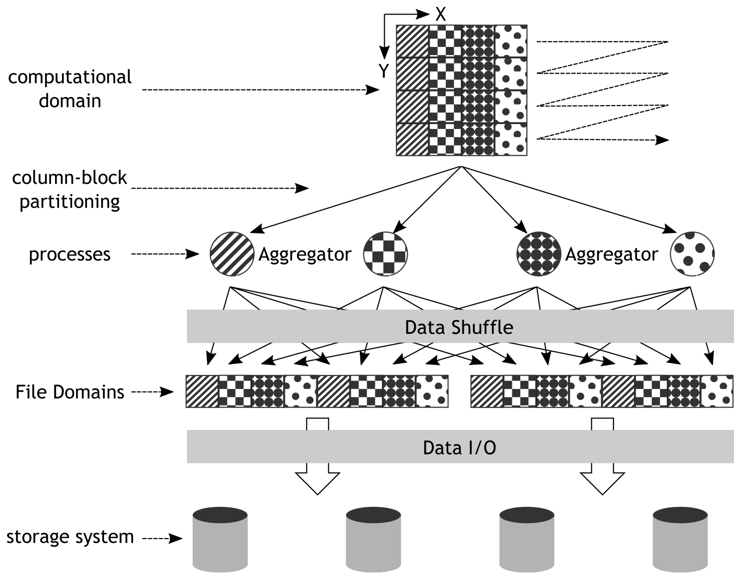


Figure 2.17: Simplified two phase I/O scheme. A two-dimensional domain is partitioned among four processes in the parallel application using a column-block strategy. Data is then rearranged to form an intermediate pattern that matches the logical data organization in the file. This pattern is afterwards used by the two aggregators to access the storage system.

row-block partitioning that matches the row-major mapping of data in the file.

Summarizing, the ext2ph has the following benefits: **(a)** improves disk utilization by converting small non-contiguous requests into large sequential accesses; **(b)** reduces block contention, and thus lock manager overhead, in POSIX compliant file systems. (In fact even when processes generate large I/O requests, it might still be beneficial to coordinate them to reduce file system block locking contention as well as concurrency level on I/O servers.) Additionally, ext2ph also: **(c)** improves network utilization by reducing the number of remote procedure calls sent to I/O servers by file system clients; and **(d)** reduces load imbalance on I/O servers.

2.4.2 Parallel Log structured File System

PLFS [BGG$^+$09] is an I/O middleware developed at the Los Alamos National Laboratory. It converts N-1 checkpoint patterns into N-N checkpoint patterns, thus moving the burden from file system lock contention to metadata creation operations. As shown in Figure 2.18, PLFS works as

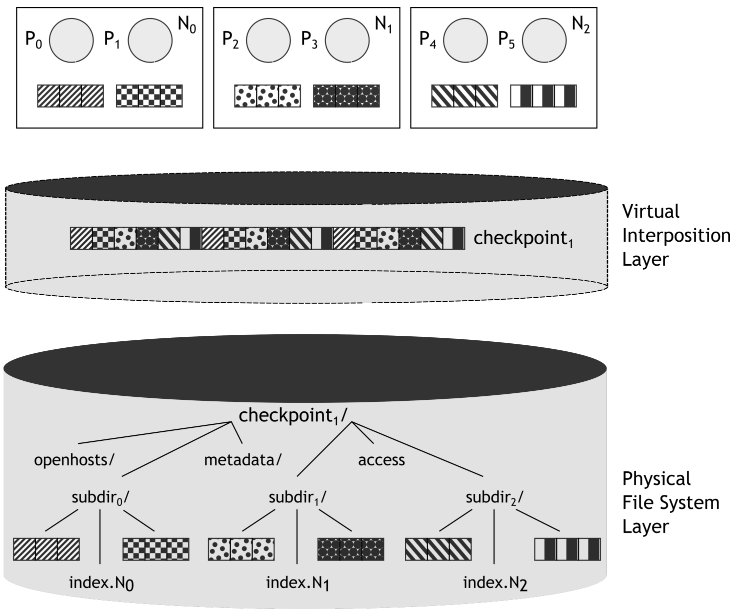


Figure 2.18: Pictorial representation of the PLFS translation mechanism that converts original application N-1 patterns into N-N patterns.

translation layer between the application and the parallel file system, and can perform its manipulations transparently to the application through a FUSE module that runs on every node. This allows PLFS to exploit the underlying parallel file system infrastructure for data distribution and protection across I/O servers and only focus on the layout manipulation of checkpoint data.

As the figure shows, a new container, represented by a directory in the physical file system, is created for every new checkpoint. The checkpoint container includes, among other things, logical file metadata and access permissions. For every node, PLFS also creates a new subdirectory in the main container to accommodate one file per process. This allows the N-N transformation giving every process an independent physical file to work with. For every node an index file is also stored, which contains mapping information to reconstruct the original file layout during read operations. Mapping data consists of offsets and lengths of each write call.

As already said, the N-N pattern strategy moves the performance bottleneck from the file system lock manager to the metadata servers, that in this case have to handle a possibly large number of concurrent file create requests. Another problem with the N-N strategy used in PLFS is that reading back checkpoiting data involves the open of many index files (one per node), from each of the N processes. Indeed, the mapping data in these files needs to be aggregated in order to be able to reconstruct the original layout in the logical file. For these reasons, in PLFS, I/O time is dominated by metadata operations. This has led to improvements of the original PLFS version that address metadata performance issues in the underlying parallel file system. In particular, Manzanares et al. [AJM+11] have proposed different alternatives for reading the index data. All the solutions move the aggregation of information from the file system level to the network level using MPI. For example, instead of allowing every process to read every index file independently, only one process can read all index files and afterwards distribute the mapping across the network, thus bypassing the file system. The same work addresses the problem of metadata server congestion during file creation by federating multiple metadata servers, handling different mount points in the file system (Panasas PanFS [WG04] was used in this case), and distributing nodes' subdirectories among them.

2.4.3 Scalable Checkpoint Restart Library

SCR [MBMS10] [MBMdS10] is a library for multi-level checkpoint restart developed at Lawrence Livermore National Laboratory and targeting N-N checkpointing patterns. It derives from two key observations on applications running on HPC systems. The first observation is that jobs only need the most recent checkpoint in order to restart. The second observation is that even in the case of failures, the number of affected nodes is typically limited to one or two. Therefore, HPC jobs can overbook the system to have an additional small number of spare nodes that can takeover the crashed ones when needed. Because only a limited number of nodes fail, and because writing data to the parallel file system can be a very slow operation, SCR uses local storage resources in compute nodes to store data. These can be RAM disks, hard disks or solid state drives. Local checkpoints are flushed to the parallel file system infrequently, thus reducing performance degradation and overload of storage system components that are more prone to failure when heavily stressed.

SCR can replicate information on local nodes across the network using MPI. There are three different replication strategies supported: *LOCAL*, *PARTNER* and *XOR*. The LOCAL strategy does not replicate data at all, only a local copy is kept; the PARTNER strategy replicates each file for every process into another partner node; finally, the XOR strategy uses XOR to write parity data across multiple nodes. The PARTNER replication can withstand the failure of only one node, while the XOR replication uses RAID-5 and can thus withstand the failure of multiple nodes as long as they do not belong to the same parity group. When an application fails because of a node crash, SCR tries to recover the data from the local storage. If it is successful it can either restart the application, if there are enough spare resources, or can flush all the local checkpoints to the parallel file system. In the latter case the job will be restarted at a later time by reading the checkpoints from the parallel file system.

2.4.4 SIONlib

SIONlib [FWP09] [FFS09] is an I/O middleware developed at the Juelich Supercomputing Center in Germany. It addresses the problem of meta-data performance in N-N checkpointing patterns by converting these into N-M patterns, where M is a smaller number of larger files (also called multifiles), most commonly equal to 1. Like the previously described PLFS, SIONlib is positioned between the application and the parallel file system, but unlike PLFS it requires modification of the target application in order to perform its transformations. For this purpose it provides an I/O API similar to ANSI C that processes (or tasks) use to access their independent files (or task-local files).

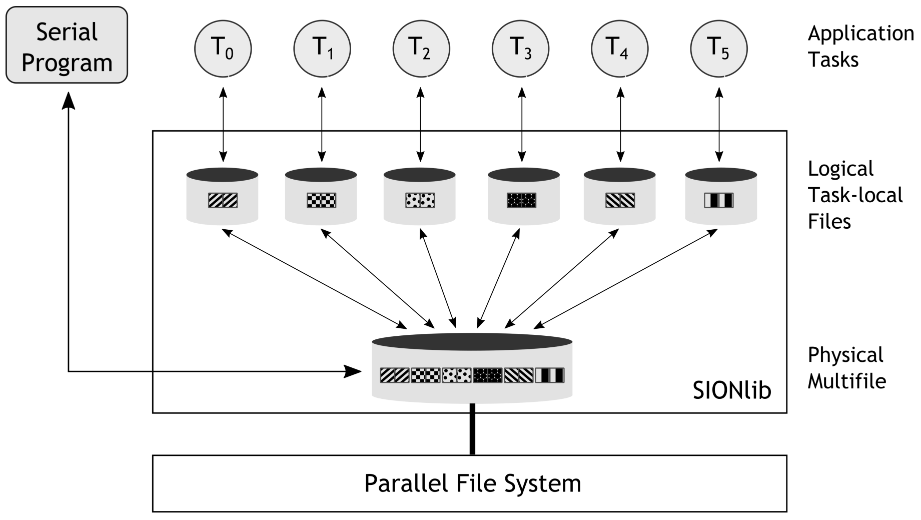


Figure 2.19: Simplified representation of the SIONlib translation mechanism. A large number of task-local files is mapped into a smaller number of multifiles.

In SIONlib data from independent task-local files is not rearranged to reconstruct the layout that it would have had if it was written to a shared file directly; that is, SIONlib merely unifies the address space of independent files into a larger address space without recreating a strided I/O pattern. Each task writes its data segments to the file independently, thus avoiding additional network communication overhead like in two phase I/O. To guarantee that data segments do not collide to the same stripe, these are naturally aligned to the file system boundaries.

Since the amount of data written by every task may vary, SIONlib needs to allocate chunks of data in the multifile for each of them upfront; the size of the chunk is estimated by taking the largest write among all. Chunks are organized into adjacent blocks in the multifile, in this way if tasks exceed the chunk size they can write the additional data into a new chunk in the next block. Additional metadata is written at the beginning and at the end of the multifile to keep track of the data distribution into chunks and blocks. Because SIONlib does not rearrange file data in the multifiles but just groups independent files under a unique container, multifiles cannot be used to restart an application using a different task configuration. Indeed, in this case the amount of data would change with the number of processes, breaking the original layout.

Most recently, in the context of the DEEP-ER project[11], SIONlib has been extended with multi-level checkpointing capabilities that allow applications to replicate checkpoints using a PARTNER configuration, like in the SCR library, also called *buddy checkpoiting*. Additionally, the library supports a XOR checkpointing configuration through an additional memory component developed during the course of the project and called *network attached memory* (NAM). When using the NAM, every task writes data to local storage and also sends a copy to the NAM, which computes and stores the parity information. If a node fails the NAM retrieves the survived nodes' data and reconstructs the failed node information through XOR with the parity previously stored.

2.4.5 Adaptable I/O System

ADIOS [LKS+08] [LLT+14] is an adaptable I/O middleware developed at the Oak Ridge National Laboratory. The key points on which ADIOS is built are: **(a)** the need for porting scientific codes from one system to another without changing I/O APIs. This also includes the possibility to add richer metadata annotations to the data, currently supported by other libraries like HDF5 and netCDF; **(b)** the need for supporting advanced technique for asynchronous I/O, thus allowing the application to be free to progress with computation while the middleware takes care of moving data to the file system; and finally, **(c)** the need for integrating scientific

[11]www.deep-er.eu

codes with additional tools for post processing and data visualization. In this sense, while previously reviewed middlewares focus on raw I/O performance, ADIOS provides an holistic approach to I/O that embraces all the aspects of data generation and exploitation.

ADIOS achieves the first goal by providing a simple flexible interface that applications can use to perform their I/O. Nevertheless, the actual I/O transport method is not determined at the application level but instead configured using an external XML file. This effectively makes applications portable from one system to another and allows flexible data representations that can include richer metadata annotations. Moreover, public I/O methods can be overloaded to send trigger messages to other tools (e.g., visualization tools) that can thus be notified when new data becomes available for processing. Like previously reviewed middlewares, ADIOS supports N-N, N-M and N-1 patterns. When using N-1 and N-M patterns, ADIOS avoids data reorganization, and thus global synchronization, by allowing every process to write independently to the file; like in SIONlib, this is possible through a custom binary format (BP-ADIOS).

Chapter 3

Guided Prefetching in Linux

The I/O gap represents a serious scalability limitation for scientific applications running on HPC clusters. Parallel file systems such as Lustre [Bra02] and GPFS [SH02] try to bridge this gap by striping files across multiple storage devices and providing parallel data paths to increase the aggregate I/O bandwidth and the number of I/O operations per second (IOPS). The ROMIO middleware implements extensions to the POSIX-IO interface, typically provided by parallel file systems, that result in a richer parallel I/O interface, and through the ADIO drivers 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 [TGL99] [Yin08] [PTH+00].

Nevertheless, as Carns et al. [CHA+11b] have pointed out, most of the scientific applications running on big clusters still use the POSIX-IO interface to access their data. Furthermore, it has also been ascertained that using POSIX-IO to access non-contiguous regions of the file causes extremely poor performance in the case of parallel file systems [CCL+06]. Indeed, parallel file systems 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.

In Section 2.3 we have seen that applications' I/O behaviour can be altered by using compiler inserted hints inside the source code. However, the resulting hints are not always accurate; sometimes the source code might not be even available. In this case hints can be still used through a binary modification tool that exploits speculative execution of the original code. Unfortunately, speculative execution requires special operating system support that is not provided in Linux systems. Therefore, currently there is no available solution to overcome limitations caused by non-optimal file I/O patterns generated by applications in Linux, 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()`[1] 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 readahead in the case of random read patterns. However, the `posix_fadvise()` system call automatically triggers a disk transfer and thus lacks the characteristics of a proper hint interface. Moreover, it is barely used in practice and has intrinsic limitations that discourage its employment in real applications.

The two most used parallel file systems in HPC 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()`[2] function and can be used to guide prefetching (or releasing) of file blocks in the page pool[3]. However, unlike POSIX advice, GPFS hints can be discarded by the file system if certain requirements are not met. Lustre, on the other hand, does not provide any client side mechanism similar to GPFS hints or POSIX advice. Recently a new Lustre advice mechanism has been proposed by DDN during the Lustre User Group 2014 (LUG14) in Miami[4]. The DDN approach provides control over the *object 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* [CGP$^+$14] [CGP$^+$17] 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

[1]http://man7.org/linux/man-pages/man2/posix_fadvise.2.html.

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

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

[4]http://opensfs.org/wp-content/uploads/2014/04/D2_S27_LustreFileSystemAccelerationUsingServerorStorageSideCaching.pdf.

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 into 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 the execution time of a *ROOT* [5] 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 3.1 reviews the POSIX advice and GPFS hints interface; Section 3.2 presents concept, design and implementation of the Mercury prototype, highlighting the main contributions of the work. This section also describes the kernel modifications that enable POSIX advice on Lustre; Finally, Section 3.3 presents related work on data prefetching.

3.1 File System Prefetching Intefaces

This section reviews the file system interfaces used by the Mercury middleware to drive prefetching in Linux environments.

[5]Data analysis framework developed at CERN, http://root.cern.ch/drupal.

3.1.1 POSIX Advice

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 in 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 readahead window size in order to have a greedier readahead algorithm. `POSIX_FADV_RANDOM`, on the other hand, can be used when a file is accessed randomly and has the effect of completely disabling readahead, therefore only ever reading the requested data. Finally, `POSIX_FADV_NORMAL` can be used to cancel the previous two advice-messages and reset the readahead algorithm to its default. 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 readahead 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.

3.1.2 GPFS Hints

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 3.2 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 3.1.

```
#define GPFS_MAX_RANGE_COUNT 8

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

} 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.

3.2 The Mercury Middleware

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.

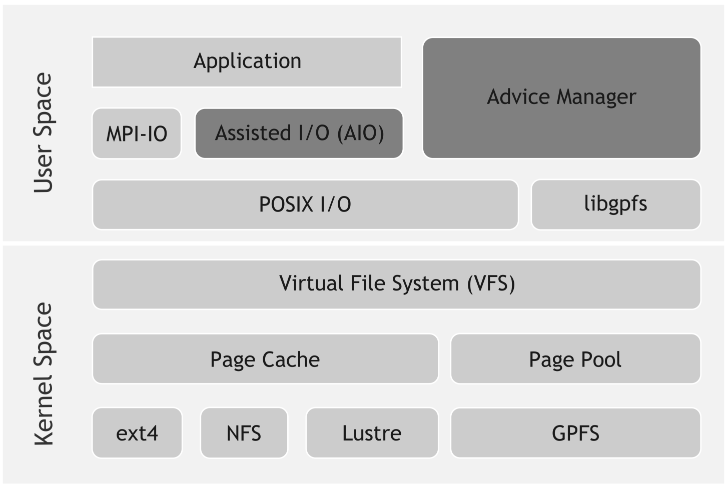


Figure 3.1: Mercury I/O software stack. *assisted I/O library* and *advice manager* communicate through UNIX domain sockets. The AM binds its socket to the local file system pathname `/tmp/channel`, while the AIO connects its socket to the same pathname; exactly in the same way they would bind and connect 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.

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

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.

[6]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.

3.2.1 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.

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’ [SR13] via a UNIX domain socket [Uni]. 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.

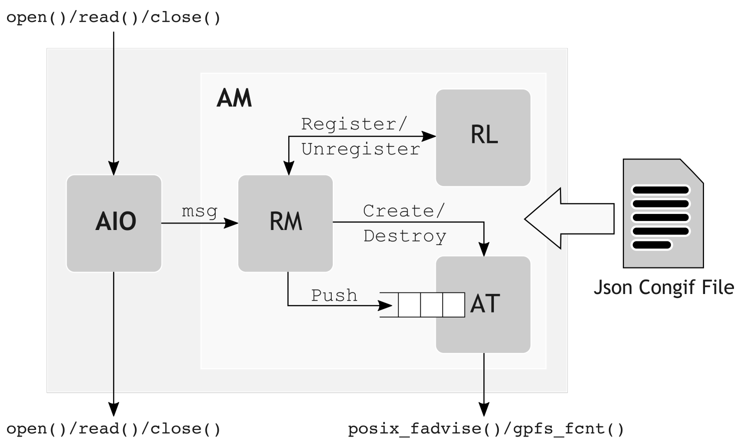


Figure 3.2: Detailed architecture for the *Advice Manager* (AM) component. This can be further divided into three blocks: *Request Manager* (RM), *Register Log* (RL), and *Advisor Thread* (AT).

3.2.2 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 3.1 is used to define which blocks of the file should be brought into the GPFS internal cache (page pool). The size of the file regions to prefetch is defined inside a Json[7] configuration file,

[7]Open standard format that uses human-readable text to transmit data objects

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 3.2 the structure of the configuration file is very simple. It allows users to define which files POSIX advice or GPFS hints should be applied to by setting the `Path` field to the full file path and the regions of the file that are likely to be accessed in terms of offset and length. In the case of POSIX advice users can also define directories to which a global advice should be applied (e.g., randomly accessed files in the directory). Additionally, when indicating a `WillNeed` advice users can directly control the caching behaviour of the *advisor thread* block cache. In particular, they can define the granularity of the prefetch request (`BlockSize`), how many blocks can be fitted into the *advisor thread* cache (`CacheSize`) and how many blocks of data should be read ahead starting from the current accessed block (`ReadAheadSize`). Clearly the example in Listing 3.2 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

consisting of attribute-value pairs (http://www.rfc-editor.org/rfc/rfc7159.txt).

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 3.1, 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.

3.2.3 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 readahead window size to zero. This is done so that Lustre will not speculatively try to gain a highly-contended lock to fulfil an optimistic readahead request.

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 readahead 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 readahead for random accessed files). This mechanism works perfectly with local file systems but has no effect on Lustre's readahead algorithm which is independent from the Linux kernel readahead. 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

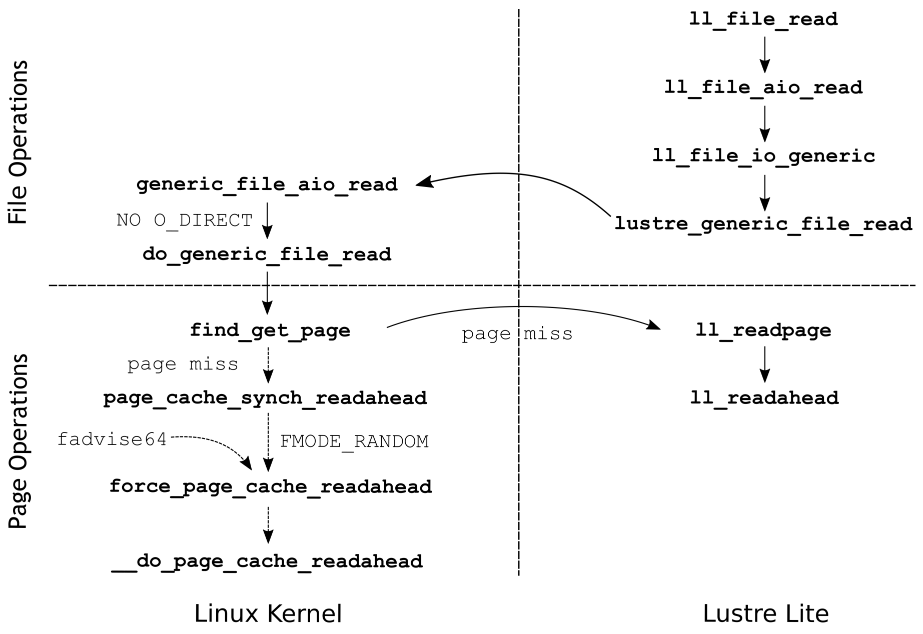


Figure 3.3: Simplified function call graph for the read operation in Lustre. For page operations in the Linux kernel the picture also shows the call graph typically followed by local reads as well as the call graph for the `POSIX_FADV_WILLNEED` advice in the `posix_fadvise()` implementation (dashed line).

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.

3.3 Contributions

In the past researchers have tried to alleviate the I/O gap by analyzing I/O patterns and exploiting their knowledge to guide I/O using, for example, data prefetching. Tran and Reed [TR04] 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 to predict future accesses and tested against a selected physics code. Several characteristics, such as execution time improvements and cache miss reduction over different hardware configurations, are considered in the experiments. The results show that execution time can be reduced by the 30% in some cases and cache misses can be reduced up to three order of magnitude.

He et al. [HBT+13] 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 [CG99], unlike previous works, did not build mathematical models but instead used speculative execution of the application code to guide data prefetching. Some authors have also used code analysis during source code compilation to automatically insert prefetch hints and hide disk latency to applications [MDK96] [BM00] [BMK01].

Other works tried to bring the same idea to higher level I/O libraries such as MPI-IO, HDF5 or PnetCDF to take advantage of the richer semantic, data dependencies and layout information. Chen et al. [CBS+08] 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. Results, with PBench running on top of NFS and PVFS as file systems backend, show execution time reduction and sustained bandwidth improvements. The same authors in [BCS+08] proposed to exploit parallel prefetching using a client-side, thread based, collective prefetching cache layer for MPI-IO. 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. Experimental results show sustained bandwidth improvements even in this case.

Chen and Roth [CR10] 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. They compare the sustained bandwidth speed-up of individual prefetching with collective prefetching for a parallel benchmarking tool using PVFS2, and demonstrate that the latter performs better than the former by over two fold on average. He et al. [HST12] 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 [VFK09] for programmers to use. Prost et al. integrated the GPFS hint functionalities in the ROMIO ADIO driver for GPFS [PTH$^+$01]. 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 Linux users to access file system specific interfaces for guided I/O without modifying applications and hiding the intrinsic complexity that such interfaces introduce.

Chapter 4

NVM Based Write-behind Caching

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 the dataset manageable, this is usually partitioned into smaller subsets and assigned to available cores for parallel processing. Complex datasets such as multi-dimensional arrays are logically flattened into a linear sequence of bytes and striped across several I/O targets for best performance. This results into the loss of the original spatial locality. 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 [CCL$^+$06] [HSS$^+$11]. This is due to the large number of 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-IO write semantics that generates lock contention on file systems' blocks.

Having recognised the small I/O problem, collective I/O was proposed by the MPI-IO community [TGL99]. 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.

In this chapter we present our approach to improve collective I/O performance using non-volatile memory devices in HPC clusters' compute nodes. The fundamental observation that motivates our approach is that collective I/O performance is limited by the slowest aggregator. Aggregators experience different run-times mainly for two reasons: the communication pattern required to build file domains might differ among aggregators and the scheduling of requests in I/O servers might not happen simultaneously for all aggregators. These two factors, in conjunction with the need for global synchronization between following rounds of two phase I/O, represent the main cause of suboptimal performance.

The motivation for using a file system based write-behind approach in collective I/O is that at Exascale the amount of memory per core will shrink [ASC10]. For this reason we need to make sure that main memory is dedicated to progressing in the computational tasks instead of the caching of large out-of-core arrays. Non-volatile memory devices provide an additional memory tier right between the DRAM and HDDs and can thus be used effectively as fast persistent buffers to mask remote file access time.

The remainder of this chapter is organized as follow: in Section 4.1 we describe in detail the extended two phase algorithm at the base of the collective I/O implementation in ROMIO; in Section 4.2 we review the main performance bottlenecks in the extended two phase algorithm providing, whenever available, possible solutions to overcome each of them; in Section 4.3 we discuss in detail our proposed solution to address global synchronization limitations; finally, in Section 4.4 we review the related works and outline the main differences with our approach.

4.1 Extended Two Phase Algorithm

We now describe in detail the ext2ph algorithm implementation in the ROMIO middleware. We pin-point where the major contributions to performance are located and on what aspect of collective I/O they reflect. Figure 4.1 shows the ext2ph algorithm flow diagram for collective write operations in aggregators; the diagram for non-aggregators is similar but does not contain write functions. (Collective read operations are not described since the behaviour is specular to the write case.) The diagram divides the algorithm into three main phases: an initialization phase (on the extreme left end side), the shuffle phase (on the extreme right end side), and the write phase (in the center).

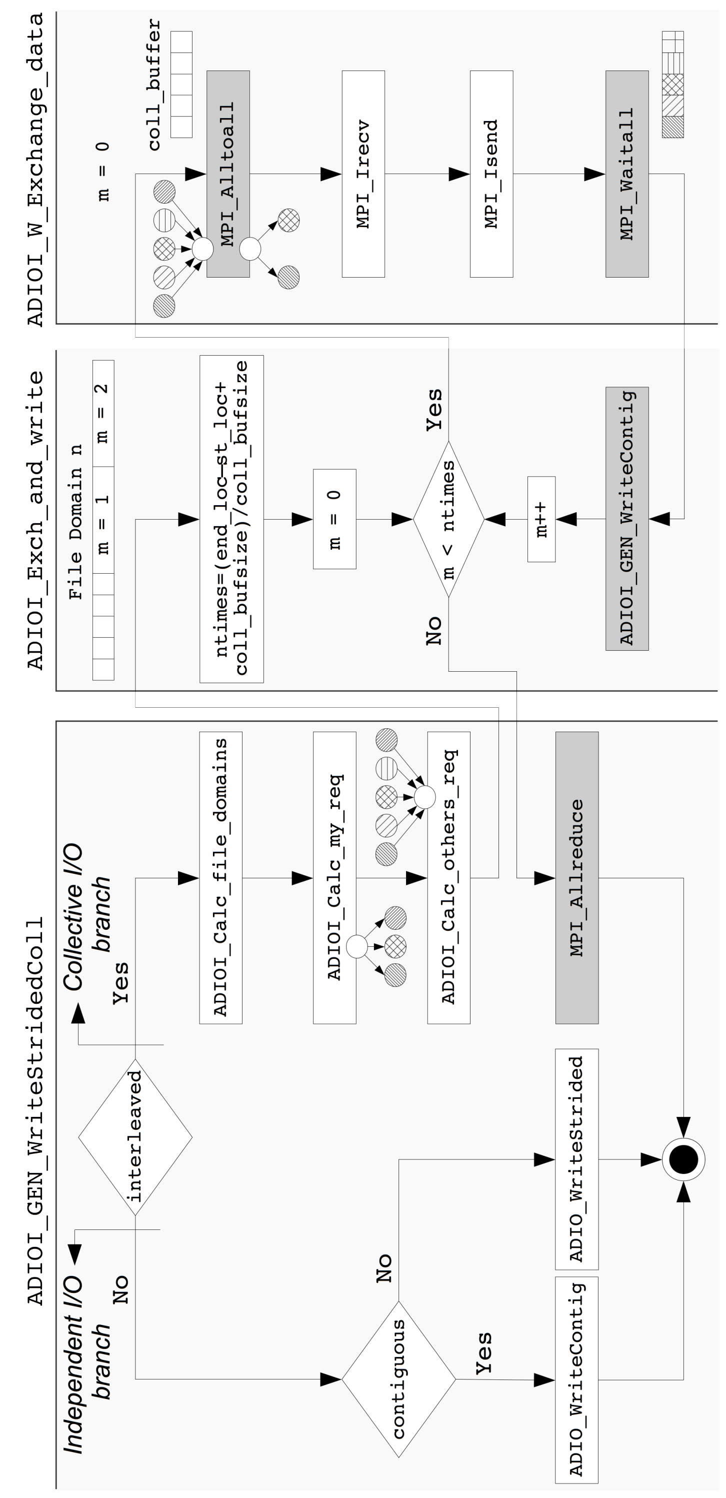


Figure 4.1: Collective I/O flow diagram for the write path in aggregators (non-aggregators neither receive nor write any data, just send it to aggregators). `MPI_File_write_all()` invokes `ADIOI_GEN_WriteStridedColl()`. Performance critical functions for the collective I/O branch are highlighted in grey.

The `MPI_File_write_all()` and `MPI_File_write_at_all()` functions are translated into the `ADIOI_GEN_WriteStridedColl()`[1] function in the presented diagram. This function is responsible for the initialization of the ext2ph algorithm. First, it computes the file domains by dividing the aggregated access region by the number of available aggregators (`ADIOI_Calc_file_domains`) as shown in Equation 4.1.

$$\left\lceil \frac{MAX(end\_offset) - MIN(st\_offset)}{cb\_nodes} \right\rceil \tag{4.1}$$

Second, for every process it computes to what file domain requests belong (`ADIOI_calc_my_req`) and third, for every aggregator it keeps track of what other requests fit into the file domain handled by the aggregator. The tracking information is stored into a *file domain access table* (FDAT) and each aggregator has one to remember what parts of the file domain belong to which process in the application. FDATs are implemented using two arrays, one for the starting offsets and one for the lengths[2]. The FDATs arrays have an entry for every process in the application, even if the process has no data in that file domain.

Once the initialization phase is complete, two phase I/O can start. Because file domains might not fit entirely in main memory, they can be broken down into smaller units using a collective buffer (by default this is only a few MB in size) and two phase I/O is carried out in multiple rounds of data shuffling and I/O. The central part of the diagram contains a *for* loop that is iterated once for every round of two phase I/O. The number of rounds is computed dividing the file domain by the collective buffer size (`coll_bufsize`).

At the beginning of the data shuffling phase (on the right end side of the diagram) every process has to communicate to aggregators what part of their file domain will be exchanged in that round. This is done using the `MPI_Alltoall()` collective communication function. Afterwards, every aggregator invokes a non-blocking receive operation (`MPI_Irecv()`) and can optionally send its own data to other aggregators by invoking a non-blocking send operation (`MPI_Isend()`). Finally, every aggregator waits for all the send and receive to complete by invoking `MPI_Waitall()`.

When the shuffling phase has completed, the collective buffer contains all the required data and it can be written to the file using `ADIOI_GEN_-WriteContig()`. This function internally calls the standard POSIX-IO write operation. When all the file domains have been written to the

[1] ADIOI_LUSTRE_WriteStridedColl() for Lustre.

[2] A process might have more than one request for the same file domain.

file, aggregators need to exchange error codes to make sure that each of them has completed correctly and user buffers can be reused for another collective write operation. This task is performed by invoking `MPI_Allreduce()`.

There are three main performance contributions to the ext2ph implementation just described: (**a**) global synchronisation cost; (**b**) communication cost; and (**c**) write cost. `MPI_Allreduce()` and `MPI_Alltoall()` account for global synchronisation cost; when a process reaches them it has to wait for all the other processes to arrive before continuing. Because aggregators are the only processes writing data to the parallel file system, they experience the highest run-time and the slowest aggregator among all governs the overall collective I/O performance. `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. Finally, `ADIOI_GEN_WriteContig()` accounts for write cost.

The collective I/O behaviour in ROMIO can be controlled by users through a set of MPI-IO hints. Users can decide 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 different scheduling time at 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 also affect the write cost since more I/O servers will be accessed in parallel potentially increasing the aggregated I/O bandwidth.

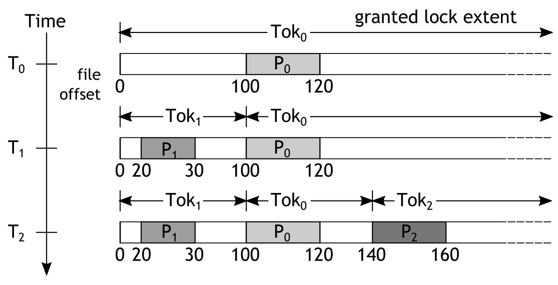
Besides the hints described in Table 4.1, there are other hints that do not directly concern collective I/O but affect its performance. The first is the `striping_factor` hint, which defines how many I/O targets (servers) will be used to store the file. The second is the `striping_unit` hint, which defines 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 block size and thus the locking granularity for the file (e.g., Lustre).

4.2 Collective I/O Limitations

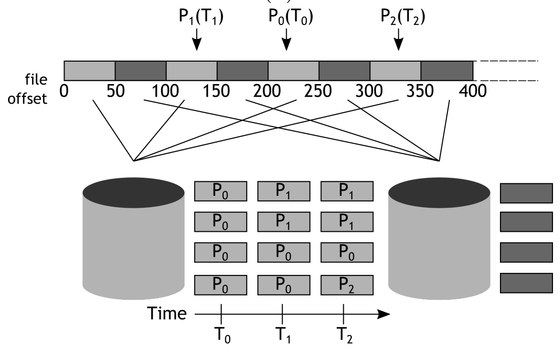
The goal of the extended two phase algorithm is to produce an intermediate I/O pattern in which the aggregated access region, in the logical file representation, is divided into multiple contiguous ranges, named *file domains*, that are transfered concurrently and in liaison by aggregators to the file system. Additional interprocess communication is traded against reduced I/O cost, since this typically dominates performance. In this section we discuss the limitations in the ext2ph algorithm and provide for each of them possible solutions that have been explored in previous research works.

4.2.1 File System Stripe Contention

POSIX compliant file systems use locking to enforce data consistency in the file. Parallel file systems typically adopt an extent based locking strategy in which the client accessing the file is granted a lock covering a larger portion than requested. Because the process might access other parts of the file in subsequent I/O operations, this strategy reduces the communication between the client and the lock manager. The way the extent based locking strategy is implemented differers between file systems. GPFS for example, uses a distributed token based approach in which the client requesting access to a region of the file is granted a token covering the whole file. This client becomes the owner of the lock and other clients wishing to access the file have to contact it in order to get tokens for other regions of the file. Lustre, on the other hand, uses a centralized server based approach in which a lock for all the file stripes, managed by a specific server, is granted to the client. The locking mechanisms just described are presented in Figure 4.2a and Figure 4.2b for GPFS and Lustre respectively.



(a)

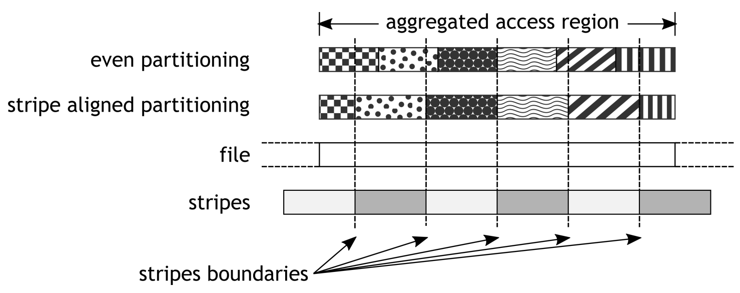


(b)

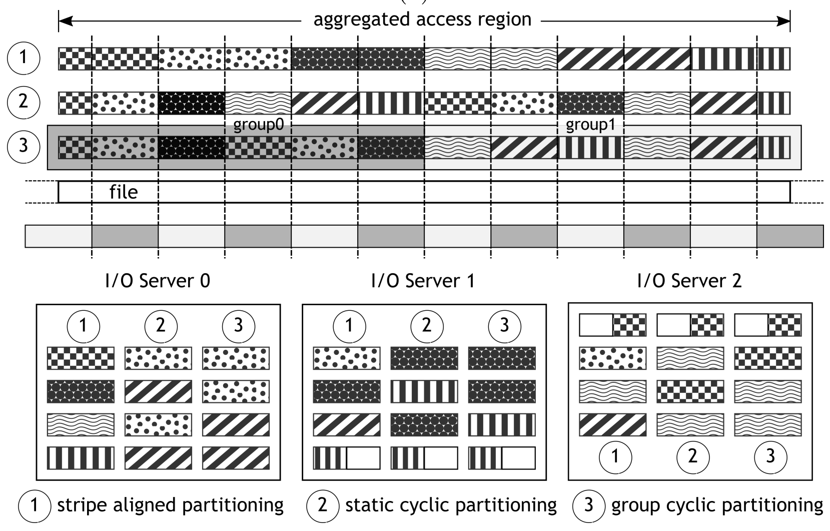
Figure 4.2: Extent based locking for GPFS 4.2a and Lustre 4.2b. In both figures there are three process requesting access to different parts of the file at different times.

In the ext2ph algorithm file domains are built using an even partitioning by default. The even partitioning guarantees an optimal load balancing because every aggregator gets exactly the same amount of data. Nevertheless, it also allows file systems stripes to be shared among multiple clients, causing false sharing of file system blocks and serializing requests at the file system level. For this reason, it is important to make sure that file domains are built keeping the file system's locking protocol in mind. A simple solution to the problem of file system stripe contention is to align file domains to stripe boundaries as shown in Figure 4.3a.

The stripe aligned partitioning works best for GPFS because the token based extent covers the file using the logical offset. For Lustre, on the other hand, the locking protocol depends on how stripes are arranged in the I/O servers. Therefore, the stripe aligned strategy causes aggregators to communicate with multiple servers, generating an increased locking protocol overhead. A better partitioning strategy would distribute stripes among aggregators in a fixed order, minimizing the communication with



(a)


aggregated access region
1
2
3
group0
group1
file
I/O Server 0
I/O Server 1
I/O Server 2
1 stripe aligned partitioning
2 static cyclic partitioning
3 group cyclic partitioning

(b)

Figure 4.3: Possible partitioning strategies for GPFS 4.3a and Lustre 4.3b. File domains, and thus aggregators, are marked with different filling patterns.

multiple servers. This approach is shown in Figure 4.3b and is called static cyclic partitioning. Static cyclic partitioning always assigns stripes to the same aggregator even across multiple I/O operations. This is done by taking the stripe number, dividing it by the number of available aggregators and taking the modulo. In this way static cyclic partitioning can also minimize the lock protocol overhead across multiple collective I/O operations. Although static cyclic partitioning can reduce locking protocol overhead, it is not yet optimal. To understand why consider case 2 in the figure. In the example, stripes from different aggregators are interleaved in I/O servers. To assign locks to each of them the lock manager will need four messages, two for each time the stripes are interleaved.

A group cyclic partition, as shown in case 3, works better because it arranges stripes belonging to the same aggregator contiguously in the server. In the group cyclic strategy the aggregated access region is divided into groups, each of these having a number of stripes multiple of the number of I/O servers. Inside each group static cyclic partitioning is then performed to assign stripes to a subset of aggregators equal to the number of servers. In the figure, for example, there are three I/O servers and six aggregators. The aggregated access region is therefore divided into two groups. Each group has six stripes and three aggregators. Using group cyclic partitioning the lock manager only needs two messages to assign the requested locks to the clients.

The interaction between the different file domain partitioning strategies and the locking protocol used by the file system has been studied by Liao and Choudhary [LC08] [kL11].

4.2.2 Logical to Physical Layout Mismatch

As we have seen in Figure 4.3b a file in Lustre, as well as in other parallel file systems, is stripped across multiple I/O servers. For this reason the logical file representation as sequence of blocks does not match the physical layout of data in the parallel file system. We have also shown how this mismatch can affect negatively performance when the file domain partitioning strategy does not take into account the characteristics of the locking protocol. More trivially, the logical to physical layout mismatch affects performance because aggregators might request data from more than one I/O server concurrently. Requests at I/O servers will therefore arrive from multiple clients and because they are served by arrival order the file system cannot guarantee that disk access will happen sequentially.

In order to have best performance we would therefore need requests at I/O servers to be served by increasing offset. This happens naturally for some specific configuration of I/O pattern when independent I/O is used over collective I/O; consider the example show in Figure 4.4 as reference. When independent I/O matches the physical layout of data in the file system we have the best case scenario and correspondingly maximum data transfer performance.

The configuration in Figure 4.4 can be easily replicated using collective I/O by building file domains appropriately. This approach has been proposed by several research works with slight changes and improvements for different type of I/O operations. Zhang et al. [ZJD09] have proposed a file domain partitioning scheme to make collective I/O resonant by matching the logical and physical layout of data in the file system. They

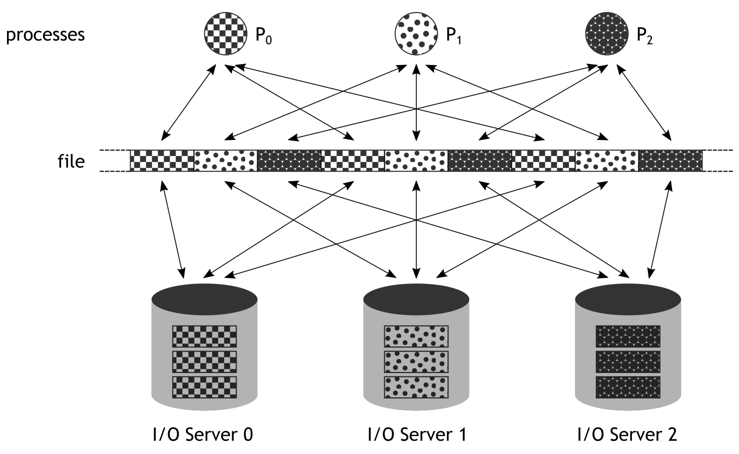


Figure 4.4: Ideal configuration of processes, I/O servers and data distribution in the system.

also proposed a solution to improve collective read performance by issuing asynchronous read operations and then synchronizing the processes in the application. The asynchronous reads work as prefetching hints for the I/O servers. Afterwards, data in the I/O servers is accessed directly by processes, thus avoiding the data shuffling phase in the ext2ph algorithm. Chen et al. [CST$^+$11] proposed a layout aware collective I/O strategy called LACIO to achieve the same results.

Layout aware strategies require additional information from the file system to discover stripe size and count (how many I/O servers are used to store the file) to build file domains accordingly. Because file domains are no longer contiguous in the logical file representation, but are instead contiguous in the physical representation, file system clients cannot access one file domain with a single I/O operation; non-contiguous I/O interfaces are therefore required. PVFS supports non-contiguous I/O constructs in the form of list I/O [CCkL$^+$02] [CCC$^+$03]. The ADIO Lustre driver writes one stripe at a time thus avoiding the need for non-contiguous I/O interfaces.

Finally, we mention that a layout aware approach can be also exploited to optimize network communication as done by Filgueira et al. [FSP$^+$08] in their *locality aware two phase I/O* (LATP I/O) solution.

4.2.3 Memory Pressure

In ROMIO the ext2ph algorithm assigns one aggregator per node by default. This does not take into account the available I/O bandwidth in the node and therefore does not make full utilization of it. Prediction of different system characteristics at Exascale estimate that node memory will not scale by the same factor as concurrency (i.e., number of cores per node) [ASC10], which translates into reduced available memory per core. Because collective I/O requires additional buffering resources to shuffle and write data, placing them in nodes without considering memory utilization may translate into reduced performance due to page reclaiming and swapping to the disk. For this reason, in order to guarantee high performance at large scales we need to consider memory utilization as well. In particular, we would like to place aggregators in nodes that have enough memory to accommodate collective I/O buffering requirements and place in every node a sufficient number of aggregators, so that they can saturate the available I/O bandwidth. This approach has been explored by Lu et al. [LCTZ12] [LCZT13].

The proposed memory-conscious collective I/O has four components. The first one is an aggregation group division responsible for identifying groups of processes that are isolated and do their aggregation independently. This maximises the data movement speed during data shuffle, reducing variance for each aggregator. The second component is an I/O workload partitioning responsible for partitioning the file region inside every aggregation group until the ideal file domain size is met. At this point all the hosts have an ideal number of aggregators each with an ideal file domain size that can saturate the host I/O bandwidth. However, aggregators' hosts might be short of memory; therefore, a third component, namely the workload portion re-merging, merges the file domains of the hosts that do not have enough memory with the neighbour until the memory requirements are met. Finally, inside every host an aggregator allocator assigns aggregators by picking processes that have more data than others in that file domain, thus minimizing the amount of data shuffled across the network.

4.2.4 Network Concurrency

Parallel file systems are shared resources accessed concurrently by many processes and possibly different applications. Requests reach I/O servers and are typically served in the order of arrival. This means that I/O operations arriving from one application might be interleaved with operations coming from another application. In order to achieve optimal performance all the requests from the same application should be served at the same time and every aggregator should spend the same amount of time in the data communication phase. If these conditions are verified, all the aggregators take approximately the same time to complete one round of two phase I/O and the total I/O time can be minimized.

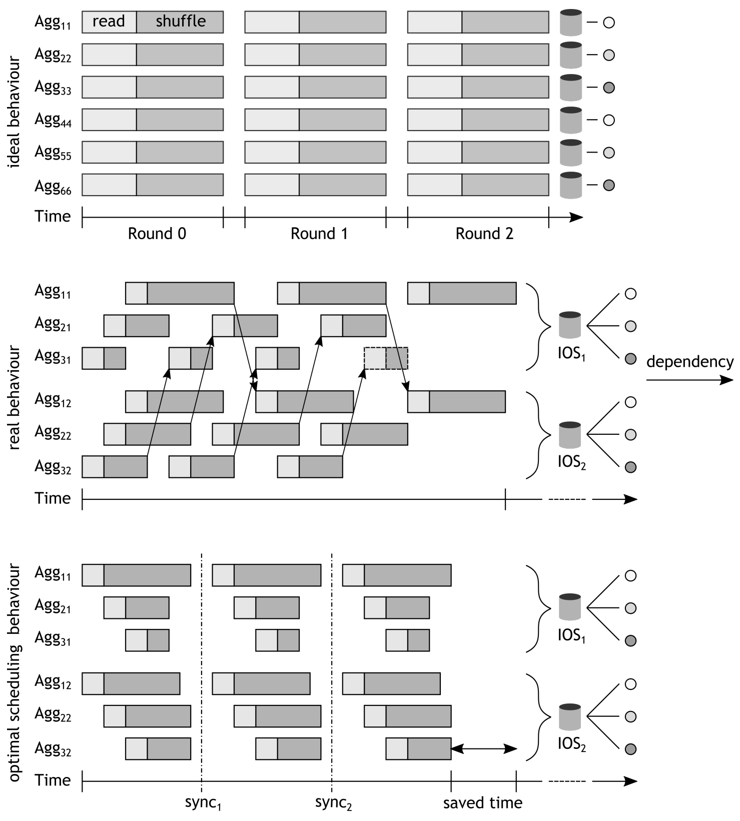


Figure 4.5: Effect of I/O server scheduling strategies on collective I/O performance. Three examples are shown, in the first every aggregator reads from a different I/O server; in the second three aggregators read from the same I/O server, which does not perform any scheduling optimization; and in the third three aggregators read from the same I/O server, which this time does perform a scheduling optimization.

Consider Figure 4.5, showing three applications with two aggregators each and three cases. Agg_{11} identifies the first aggregator of the first application reading data from IOS_1, Agg_{12} identifies the second aggregator of the first application reading data from IOS_2, and so on. Every application performs three rounds of two phase I/O. In the 'ideal behaviour' every aggregator reads data from a different I/O server and takes the same time to shuffle data to other processes across the network; in this case all the requests can be ideally scheduled at the same time, giving the shortest running time for every application. In the 'real behaviour' aggregators read data only from two I/O servers and take different time to shuffle data; in this case the shuffle time varies for every aggregator and is higher for those aggregators that need to exchange data with processes that are placed in a different node[3]. In the 'optimal scheduling behaviour' the slowest aggregator is always served first; in this case while aggregators are busy shuffling data to other processes, I/O servers can satisfy following read requests, thus overlapping network communication and I/O time, minimizing the runtime for all applications.

Liu et al. [LCZ13] have proposed a new scheduling algorithm for PVFS servers that takes into account the shuffle cost of aggregators across multiple applications and always schedules the slowest aggregators first, thus achieving the slowest average running time for all of them. This optimization only works for collective read operations. Indeed, for writes the shuffle phase happens before the I/O phase. In order to implement the same strategy for write operations, one could use a double buffering approach to pipeline data shuffling and writes. This solution has been proposed by Sehrish et al. [SSL+13].

4.2.5 Global Synchronization Overhead

As we have previously discussed in the ext2ph algorithm collective I/O performance is negatively impacted by global synchronization. Collective MPI constructs are used to coordinate processes in the application and to exchange state during multiple rounds of data shuffling and I/O. Nevertheless, there are cases in which the input domain decomposition, and thus the distribution of requests in the file, does not require to globally synchronize all processes but only smaller groups of them. These I/O patterns can be exploited to reduce the global synchronization overhead. Figure 4.6 shows an example of six processes participating in a collective write operation. In the example there are four aggregators and, as we can see, two groups of three processes exchange data with only two aggregators.

This observation has been exploited by Yu and Vetter [YV08] to partition collective I/O into smaller groups of processes that coordinate independently from each other, that is, in the ext2ph implementation these processes can use different communicators when exchanging data shuffling information with `MPI_Alltoall()` (refer to Figure 4.1). Because some I/O patterns do not allow the partitioning of processes, they convert the original I/O pattern using an intermediate file view and rearrange data in the file to match the intermediate pattern. This approach works but has considerable limitations. In particular, if the file is written using a certain number of processes and aggregators, the original input can be

[3]When aggregators are served by arrival order and the slowest aggregators always arrive last, the overall collective I/O time dilates.

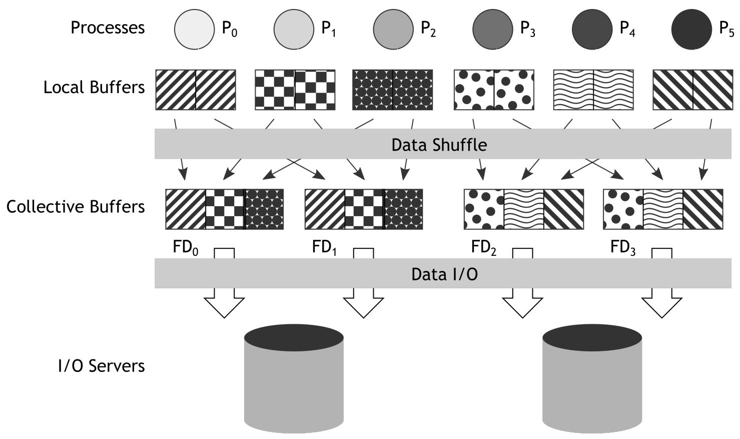


Figure 4.6: Six processes are collaborating in collective I/O. Because P_0, P_1 and P_2 do not exchange data with other processes there is no need for them to communicate data shuffling information to P_3, P_4 and P_5 during two phase I/O rounds.

reconstructed only if data in the file is read using the same number of processes and aggregators. The reason is that MPI-IO does not define a binary format. Data is written using byte information and in order to reconstruct the intermediate file view the collective I/O configuration must be the same. The immediate limitation of this approach is that if data is written for checkpoint/restart purposes, the application cannot be restarted using a different number of processes because read and write layout would not match.

4.3 A Non-Volatile Memory Based Approach

As we have discussed in Section 4.2, collective I/O performance is limited by the slowest aggregators run-time. This is mainly due to the fact that the parallel file system is shared among many applications in the cluster and requests coming from the same application are not served simultaneously by all I/O servers. This, in conjunction with the need for global synchronization at the beginning of every phase of data shuffle and I/O, contributes to the suboptimal performance in large scale clusters. In this work we focus on write performance improvements since HPC simulation codes are write intensive; while read operations are typically limited to loading of initialization parameters that are used during the simulation. Our approach focuses on global synchronization overhead reduction in the extended two phase algorithm. We achieve our goal by taking advantage of non-volatile memory devices, more specifically SSDs, in compute nodes.

Instead of performing collective I/O to the global parallel file system directly, we perform collective I/O to the local NVM storage and then move data to the global file system asynchronously (i.e., in the background), allowing the application to continue with useful work. Data synchronization is not performed collectively but instead independently. This effectively converts collective I/O to the parallel file system into independent I/O, taking the parallel file system out of the collective I/O critical path. Since local NVM storage devices are not shared with other nodes, I/O requests can be served almost simultaneously, thus reducing the I/O response time in aggregators and limiting the amount of time spent in global synchronization operations. Our approach also benefits the memory pressure on compute nodes because we can achieve high performance using smaller collective buffers.

In this section we present the high-level architecture of the ROMIO implementation of MPI-IO as well as our proposed design. The two are shown in Figure 4.7a and 4.7b 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 the ADIO interface, following an approach similar to the Virtual File System layer of the Linux Kernel.

In Figure 4.7a 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 version of 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()`). 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.

```
struct ADIOI_Fns_struct ADIO_LUSTRE_operations = {
    ADIOI_LUSTRE_Open,             /* Open */
    ADIOI_GEN_OpenColl,            /* OpenColl */
    ADIOI_LUSTRE_ReadContig,       /* ReadContig */
    ADIOI_LUSTRE_WriteContig,      /* WriteContig */
    ADIOI_GEN_ReadStridedColl,     /* ReadStridedColl */
    ADIOI_LUSTRE_WriteStridedColl, /* WriteStridedColl */
    ADIOI_GEN_SeekIndividual,      /* SeekIndividual */
```

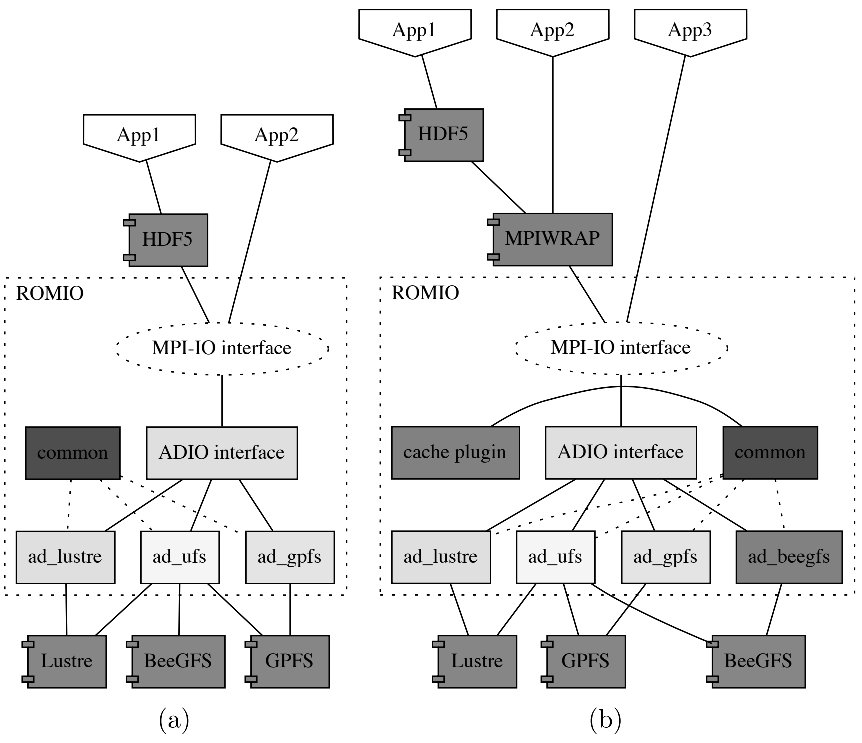


Figure 4.7: Original ROMIO architecture (4.7a) and proposed ROMIO architecture (4.7b).

```
    ADIOI_GEN_Fcntl,                 /* Fcntl */
    ADIOI_LUSTRE_SetInfo,            /* SetInfo */
    ADIOI_GEN_ReadStrided,           /* ReadStrided */
    ADIOI_LUSTRE_WriteStrided,       /* WriteStrided */
    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 4.7b 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. Indeed, most of the file system drivers supported in ROMIO use the common implementation of the basic I/O functionalities like, for example, `ADIOI_GEN_OpenColl()` to collectively open a file, `ADIO_-Close()` to collectively close a file, and `ADIOI_GEN_WriteContig()` to write a contiguous extent of data to the file using the POSIX write operation. Furthermore, we have 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.

4.3.1 MPI-IO Hints Extensions

In order to take advantage of attached non-volatile memories in compute nodes we have 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 common layer supporting them.

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 synchronization thread in pool
`ind_wr_buffer_size`	synchronization buffer size [bytes]

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 synchronization 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 are locked until cache synchronization is completed. This prevents other processes from modifying the same data before this is persisted in the global file system. 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 synchronization policy of cached data to the global file. If the hint is set to `flush_immediate` data is immediately flushed to the global file. Alternatively, if the hint is set to `flush_onclose` data is flushed to the global file when it is closed. A `flush_none` option is also available to keep data local to the node and never flush it to the global file system. This might be used in the case the user does not wish to flush every checkpoint 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 is removed after closed, otherwise (`disable`) it is retained until the user manually removes it. The `e10_cache_threads` hint is used to communicate to the implementation the number of threads to be created in the synchronization 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 synchronize 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 Table 4.1 of Chapter 2.

Besides the proposed cache policies, more complex ones are possible. For example, the cache synchronization 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 are beyond the scope of this work. Here, our goal is to demonstrate the benefit that the employment of NVM devices in compute nodes can have on parallel I/O performance.

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], which come in the form of cache plugin. The proposed cache plugin

[4] http://www.github.com/gcongiu/E10.git

class diagram is shown in Figure 4.8. The cache plugin provides all the functionalities necessary to handle the additional cache layer. In the figure there are three main software components:

- a synchronization thread object that takes care of moving data from the local to the global file system;
- a synchronization request object used to describe what data should be moved and finally;
- an atomic queue object that makes up the communication channel between main and synchronization threads.

In order to handle the cache file properly we have also extended the `MPI_File` opaque object with two additional attributes, a cache file handle named `cache_fd` and an array of pointers to available synchronization thread instances named `thread_pool`.

Moreover, we have added two functions, `ADIOI_Sync_thread_pool_init()` and `ADIOI_Sync_thread_pool_fini()`, to initialize and finalize the thread pool. In the following we describe in detail the three software components just introduced and explain how they work together.

Cache Synchronization Thread

The `ADIOI_Sync_thread_t` object provides the infrastructure to initialize/finalize threads and to enqueue, flush and wait for synchronization requests. A pool of synchronization threads is created when the file is opened with `MPI_File_open()` and destroyed when the file is closed with `MPI_File_close()`. The synchronization thread object has six attributes: the `fd_` attribute is a pointer to the internal ROMIO representation of the MPI file object and is used by the thread to retrieve all the information it requires to perform its tasks (e.g., POSIX file descriptors) without having to pass such information through the synchronization request object; the `tid_` attribute is the POSIX thread identifier and is used by the main thread in the `pthread_join()` operation when the file is closed; the `attr_` are the POSIX thread attributes; finally there are three queues, a submission queue named `sub_` that contains requests that are ready to be served, a pending queue named `pen_` that contains requests that are not yet ready to be served, and a wait queue named `wait_` that contains a copy of every request that is in the submission queue and is used by the main thread to check status of submitted requests.

The public interface of the synchronization thread provides two methods to initialize and destroy the thread object and three additional methods:

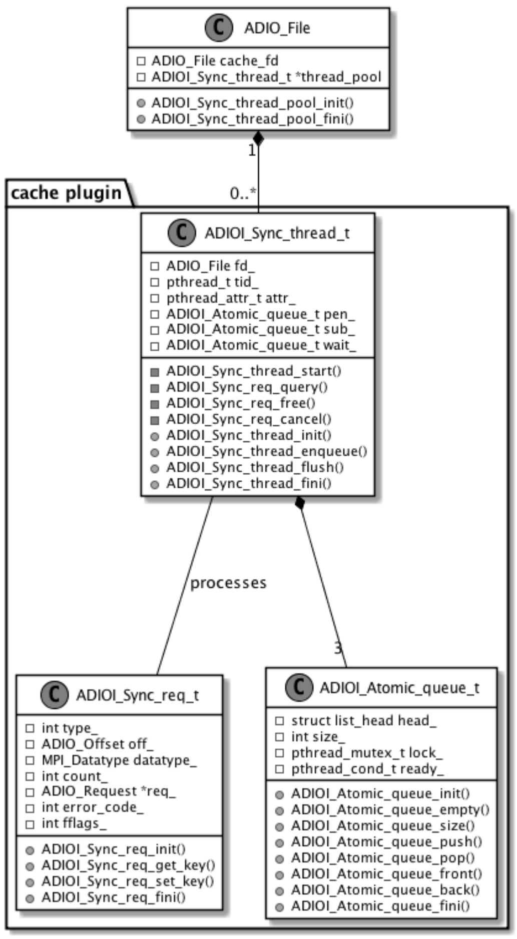


Figure 4.8: Cache plugin class diagram. The synchronization thread `ADIOI_Sync_thread_t` serves synchronization requests of type `ADIOI_-Sync_req_t`.

`ADIOI_Sync_thread_enqueue()` used by the main thread to enqueue new requests in the pending queue, `ADIOI_Sync_thread_flush()` used by the main thread to move requests from the pending queue to the submission queue, thus allowing the thread to serve them, and finally `ADIOI_Sync_-thread_wait()` used by the main thread to wait for all the submitted requests to complete. The thread object also contains four additional internal methods that are not directly visible to the main thread and are used to implement the supported services through the MPI generalized request interface [mpi12].

Cache Synchronization Request

The `ADIOI_Sync_req_t` object provides the infrastructure required to initialize/finalize, set/get attributes to/from synchronization requests. Synchronization requests are initialized by the main thread in the `ADIO_-GEN_WriteContig()` function and submitted to synchronization threads which will satisfy them while the main thread can progress with its work. The synchronization request object has seven attributes: the `type_-` attribute specifies the type of the request, either `ADIOI_THREAD_SYNC` or `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, while the second is used to shut down the synchronization thread when the thread is no longer needed; the `count_` attribute represents the number of elements of type `datatype_` to be transfered, while the initial position of these in the file is defined by the `offset_` attribute; the `fflags_` attribute tells the synchronization thread when data should be transfered; the `req_-` attribute is a MPI request handle and is used by the main thread to check the synchronization status of the request by invoking `MPI_Wait()`; finally, the `error_code_` attribute is used by the synchronization thread to set the return code that is afterwards interpreted by the main thread.

The public interface of the synchronization request object provides two methods to initialize and destroy the synchronization request object, a get method named `ADIOI_Sync_req_get_key()` and a set method named `ADIOI_Sync_req_set_key()`.

Atomic Queue

The `ADIOI_Atomic_queue_t` object provides the communication channel between the main thread and the synchronization thread enforcing atomicity through POSIX mutual exclusion constructs. The atomic queue object has four attributes: the `head_` attribute is a pointer to the head of a double linked list used to implement the queue[5]; the `size_` stores the number of elements currently present in the queue; the `lock_` attribute is a POSIX mutex used to ensure internal data structure consistency during queue manipulation; finally, the `ready_` attribute is a condition variable used by the main thread to signal the synchronization thread that the queue is no longer empty.

The atomic queue object supports the standard queue APIs[6] and thus we do not give a description for them here.

[5]We used the Linux kernel implementation of the double linked list.

[6]http://www.cplusplus.com/reference/queue/queue/?kw=queue

Collective Write Caching

Now that we have described all the cache plugin components, we can explain how the cache plugin works in collective write operations. Figure 4.9 shows the flow diagram obtained by extending the diagram in Figure 4.1 with the cache plugin.

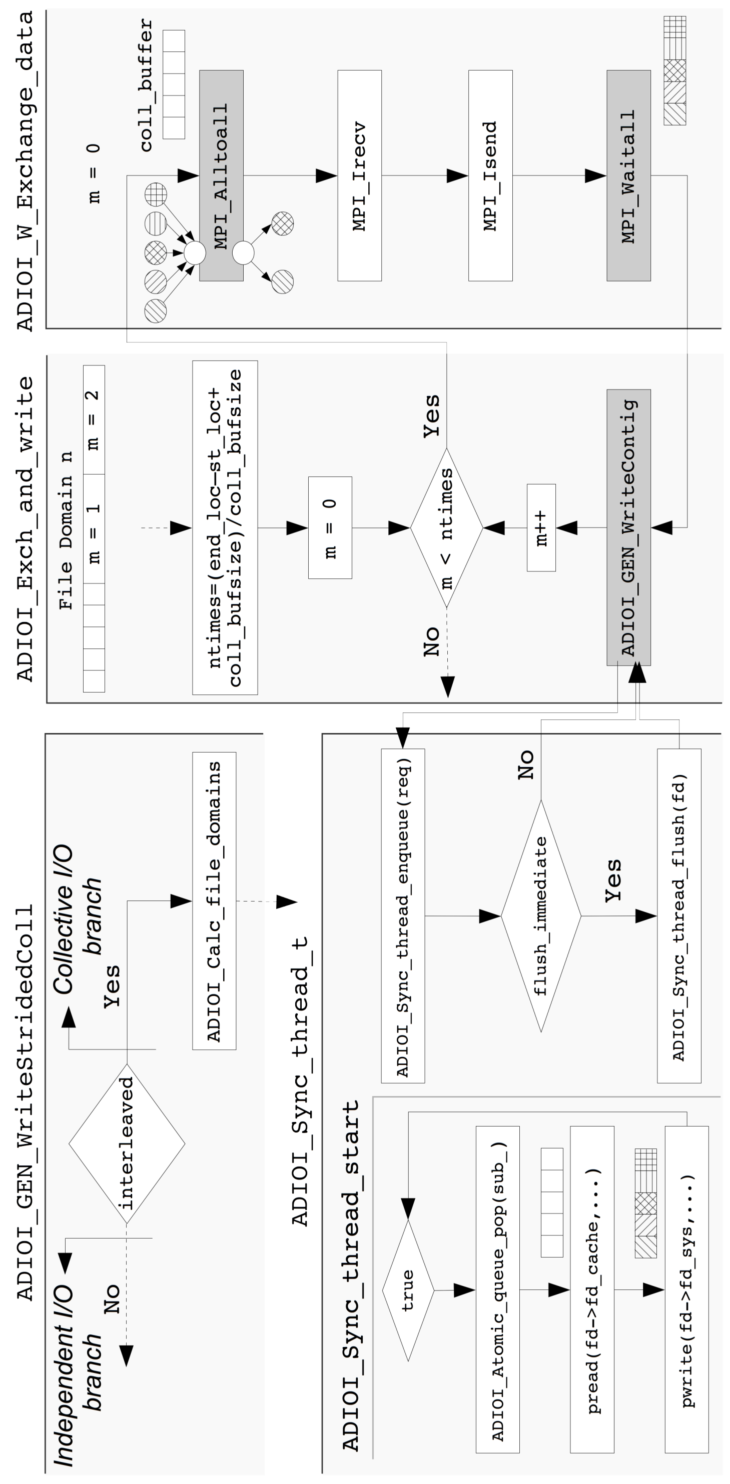


Figure 4.9: Extended collective I/O flow diagram including cache plugin support.

The first part of the diagram in the upper left part is left unchanged. The changes are introduced in the `ADIO_GEN_WriteContig()`[7] operation. This has been modified to redirect writes to the local file system cache whenever the cache is enabled, that is, when the `e10_cache` hint is set to enable. After data is written to the cache, the function creates a new synchronization request of type `ADIOI_THREAD_SYNC` by invoking `ADIOI_Sync_req_init()` and enqueues it into the pending queue of the synchronization thread through `ADIOI_Sync_thread_enqueue()`. At this point the main thread also checks the desired flushing policy defined by the `e10_cache_flush_flag` hint. If the hint is set to `flush_immediate` the main thread immediately flushes the enqueued requests to the submission queue by invoking `ADIOI_Sync_thread_flush()` and then returns. Otherwise requests are left in the pending queue and will be served when the file is closed with `MPI_File_close()` or when the main thread invokes `MPI_File_sync()`.

4.3.3 BeeGFS Cache Integration in ROMIO

The BeeGFS file system provides its own caching infrastructure, which includes a set of APIs to control the caching functionalities and a daemon process, running on the host machine, that takes care of moving data between the cache and the global file system. For this reason, some of the cache plugin features presented before for the universal file system module in the common layer are redundant. In particular, we do not any longer need to manually open a cache file in the host and start a pool of synchronization threads. When using the BeeGFS cache API the file system takes care of all these aspects for us. Nevertheless, we still reused some parts of the cache plugin to integrate the BeeGFS cache support into ROMIO. For more details about the cache implementation in the BeeGFS driver refer to the source code at http://www.github.com/gcongiu/E10.git.

4.3.4 Consistency Semantics

As far as write consistency semantic is concerned, the MPI-IO interface does not make any assumption regarding the underlying parallel file 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

[7] In the BeeGFS driver this is replaced by `ADIOI_BEEGFS_WriteContig()`.

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 [CCL+04], 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 synchronization, 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. However, 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 retrieve the data from the aggregators' cache without incurring into 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.

4.3.5 Changes to the Application's Workflow

Simplifying, most HPC simulation codes 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()`: this function invokes the underlying `ADIOI_GEN_WriteStridedColl()` previously described in Figure 4.1;
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 4.10 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 synchronization 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.

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. However, the file will not be really closed. Instead, its handle will be

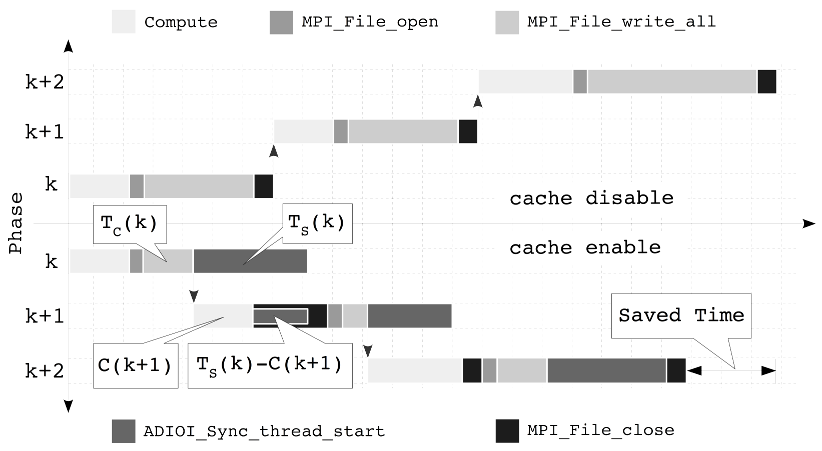


Figure 4.10: Standard and modified workflows. When cache is disabled compute phase 'k+1' starts after file 'k' has been closed. When the cache is enabled compute 'k+1' can start immediately after data has been written. At the same time, background synchronization of cached data starts. File 'k' is closed before the file 'k+1' is opened, forcing the implementation to wait for cache synchronization to complete.

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 synchronization completion check for each of them.

4.3.6 Write 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 write bandwidth for k is expressed by Equation 4.2:

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

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

$$BW = \frac{\sum_{k=0}^{N-1} S(k)}{\sum_{k=0}^{N-1} T_c(k) + max(0,\ T_s(k) - C(k+1))} \tag{4.3}$$

From Equation 4.2 (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 synchronization to 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 synchronization completion (Figure 4.10).

4.4 Contributions

Many research works have tried to optimize collective I/O focusing on different aspects. Yu and Vetter [YV08] before us have identified the global synchronization 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 [LKS+08] 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 et al. [LCTZ12] [LCZT13] further explored collective I/O performance beyond global synchronization 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 [kL11] 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.

Chen et al. [CST+11] addressed the problem of I/O server contention using a layout aware strategy to reorganize data in aggregators. On the same lines, Xuechen et al. [ZJD09] proposed a strategy to make collective I/O 'resonant' by matching memory layout and physical placement of data in I/O servers and exploiting non-contiguous access primitives of PVFS. 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 et al. [LCZ13] exploited the scheduling capabilities of PVFS I/O servers to rearrange I/O requests' order and better overlap read and shuffle phases among different processes. Yu et al. [YVCJ07] used the file joining functionalities of Lustre to convert collective I/O into independent I/O to multiple files, thus avoiding file system stripe contention. All resulting files are afterwards rejoined into a single shared file.

Lee et al. [LRT$^+$04] 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) [MWLY03], 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 [MBMdS10] [MBMS10] 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 [DAC$^+$12], but we do an evaluation of collective I/O and show how the effect of I/O jitter can become even more prominent when using fast NVM devices. More recently the Fast Forward I/O project [LJM$^+$16], from U.S. Department of Energy (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[8]. 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 synchronization 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 synchronization is implemented in

[8]http://www.ddn.com/products/infinite-memory-engine-ime14k

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.

Chapter 5

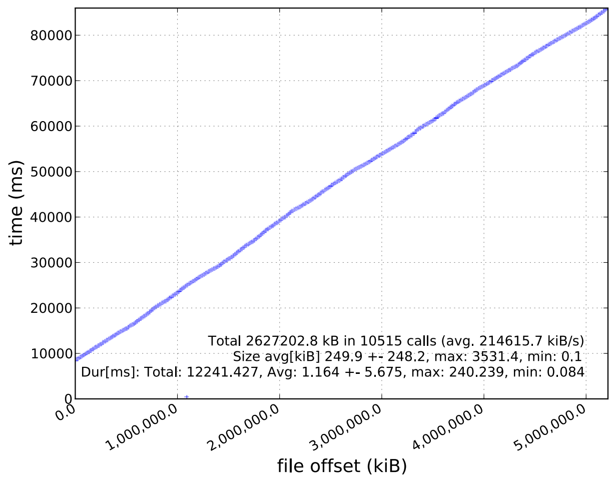
Evaluation

In this chapter we present results for the Mercury middleware and the MPI-IO hints extensions for collective I/O proposed in previous chapters. For Mercury we consider a high energy physics code. This application is representative of a class of HPC analysis applications that are becoming more and more common in workloads of leadership class systems. Our goal is to demonstrate that through user guided I/O prefetching even a non-optimized application can improve its I/O utilization (and thus performance) of the file system. For our ROMIO improvements we consider a range of collective I/O benchmarks that include synthetic and real application kernels. We run every benchmark with the baseline collective I/O implementation in ROMIO and discuss in detail all the ext2ph performance implications; then we enable our NVM based optimization and show how the same performance implications can be mitigated or completely cancelled when taking the global parallel file system out of the ext2ph critical I/O path.

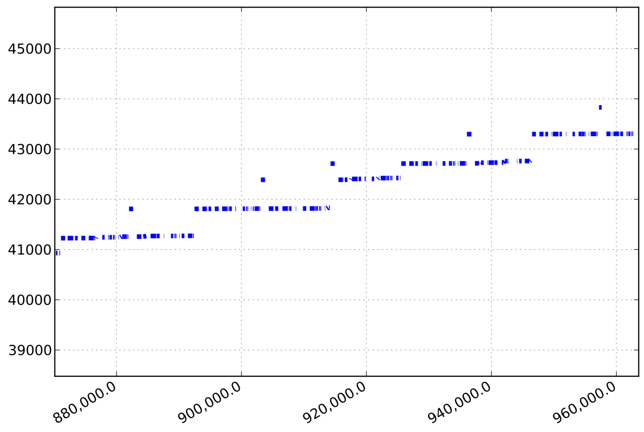
5.1 I/O Prefetching in ROOT

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 runs on the Mogon cluster at the data processing center of the University of Mainz (ZDV), in Germany. It analyzes particle data read from an input file structure using 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*,



(a)



(b)

Figure 5.1: I/O read profile of the target application under analysis (5.1a), extracted from the the GPFS file system in the test cluster, and zoomed window (5.1b) showing the actual pattern details.

ioapps[1] and GPFS's *mmpmon* monitoring tool. Figure 5.1 shows the I/O pattern along with some additional statistics. As it can be seen, in this specific case (5 GB file), the application issues a total of 10515 `read()` system calls to read about 2.6 GB of total data. The average request size is 250 KB 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 sequential, most of the accesses to the file follow an increasing offset.

[1] https://code.google.com/p/ioapps.

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 5.1b).

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[2], the size of the non-overlapping ranges may change significantly. This general behaviour can be modelled using Mercury through a configuration file in which a 'WillNeed' hint covers the whole file from beginning to end (i.e., 'Offset' and 'Length' equal to 0). The backwards seeks can be accounted for using the 'CacheSize' parameter to keep previously accessed blocks in cache. In this way we effectively emulate a sliding window that tracks the application's I/O behaviour. This would not be possible by just using a, e.g., `POSIX_FADV_WILLNEED` advice on the whole file before starting the application like shown by Figure 5.2. The reason is that even if the file fits entirely into the cache, we would have a large number of valuable pages, possibly from other applications, 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 Mercury prototype on the application and file systems performance we considered the application execution time and the number of reads accounted for by the respective file systems. We conducted our experiments without file system hints and then with file system hints issued transparently to the application by the *Advice Manager*. Furthermore, we ran each experiment three times and calculated average, minima and maxima for each metric. In order to avoid caching affecting our measurements, extra care was taken to clean all the relevant caches for the different file systems. For ext4 and Lustre this was accomplished by using the command line:

echo 3 > */proc/sys/vm/drop_caches*

on the file system clients. Additionally, for Lustre this command was also executed on the *object storage servers* (OSSs) to avoid the server side

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

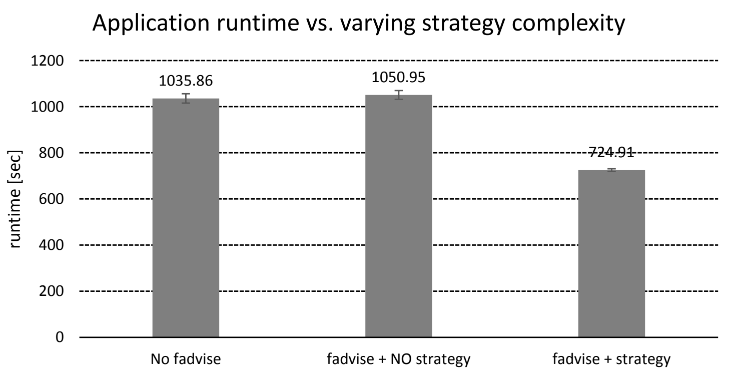


Figure 5.2: Comparison between different usage stategies of posix_fadvise for an input file of 55 GB residing in an ext4 file system. The first bar represents the case in which no advice is used, the second bar represents the case in which a POSIX_FADV_WILLNEED is issued for the whole file at the beginning of the application and the third bar represents the case in which POSIX_FADV_WILLNEED is issued using Mercury.

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 GPFS *network shared disks* (NSDs)[3] servers do not cache any data.

5.1.1 Testbed

Our testbed is composed by a test cluster of seven nodes, mainly intended to evaluate the proposed Linux kernel modifications with the Lustre file system. The reason for using a smaller cluster instead of the Mogon system 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 file systems have a single disk server each, one Dell R710 acts as GPFS network shared disk (NSD) server and another as Lustre object storage server. The R710 are equipped with two quadcore E5620 @2.4 GHz and 24 GB main memory. For storage, both disk servers share a MD3200 array with 2 controllers and 4 MD1200 expansion shelves for a total of 60 2 TB 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 128 KB, which results

[3]GPFS name for I/O servers.

in a RAID stripe size of 1 MB. 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 file systems and tests do not interfere with each other, as long as only one file system 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 @3.3 GHz and 16 GB of main memory, as metadata target (MDT) it uses a 120 GB SSD Intel 520. Four other machines of the same type, equipped with an eight core E3-1230 @3.3 GHz processor and 16 GB of main memory, work as compute nodes and file system clients. All machines, servers and clients, are equipped with Intel X520DA 10 Gigabit 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 4 MB.

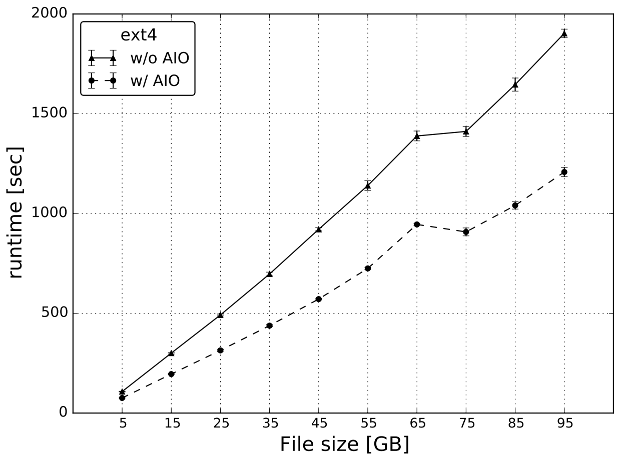
5.1.2 Performance Results

To measure the performance improvements that our Mercury prototype can deliver to the application's run-time we conducted two set of tests. In the first test we varied the size of the input file from 5 to 95 GB. 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. Figures 5.3 and 5.4 report the results for the described experiments. All the tests where performed using a 'BlockSize' of 4 MB, 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 32 MB of cache space and 512 MB in total for 8 application instances.

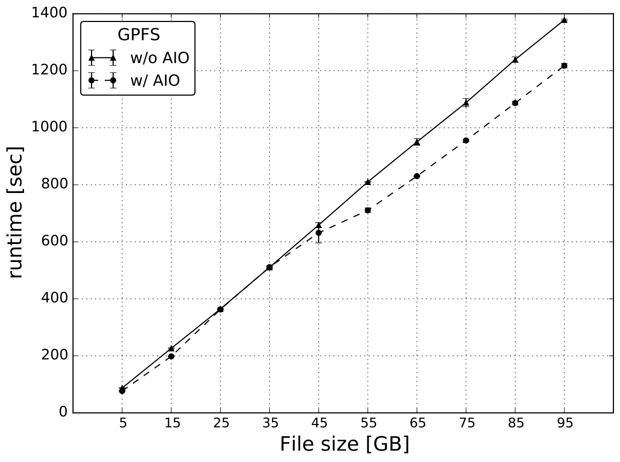
The 'WillNeed' on the whole file causes the *Advisor Thread* to issue up to 4 ('ReadAheadSize') prefetching requests for blocks of 4 MB sequentially, starting from the current accessed block. This has the same effect of data sieving in ROMIO, optimizing the access size and allowing the application to 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 5.3a and 5.3c. In these cases we measure reductions in the execution time of up to 50%, 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 5.3b). 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).

As far as Figures 5.4a, 5.4b and 5.4c 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.

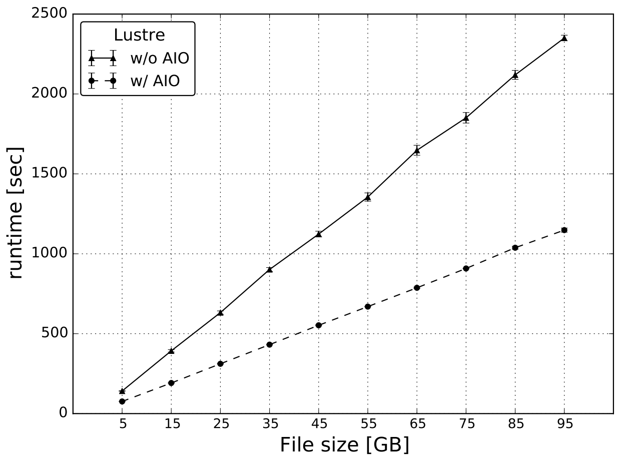
Looking at the figures generated with the new SAIO library we can assess the effectiveness of the prefetching hints for the three file systems considered. In particular, Lustre provides the best run-time improvements compared to the case in which no hints were used. GPFS shows a more contained improvement 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 run-time improvements shrink. Because ext4 transfers data directly from the local SATA HDD, in which I/O time is dominated by disk latency, we can still improve performance as the number of application instances increases by making efficient use of the cache to hide such latency. GPFS and Lustre servers, on the other hand, have much higher transfer performance compared to single disk ext4 and both use a network link to move data across the network; GPFS uses the NSD protocol while Lustre uses the LNET protocol. For GPFS and Lustre network bandwidth becomes critical and, in fact, as the number of instances increases the run-time decreases because of the saturation of the network link in the node.



(a)

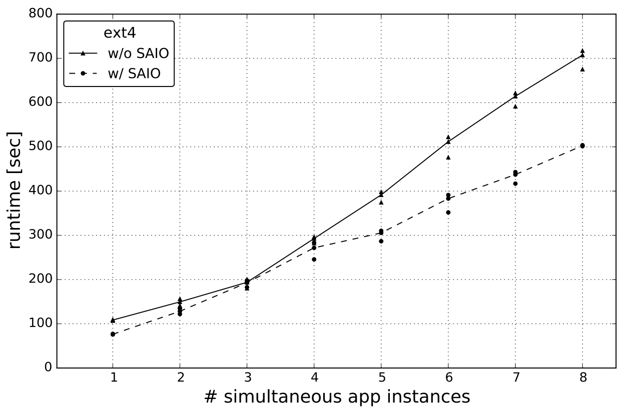


(b)

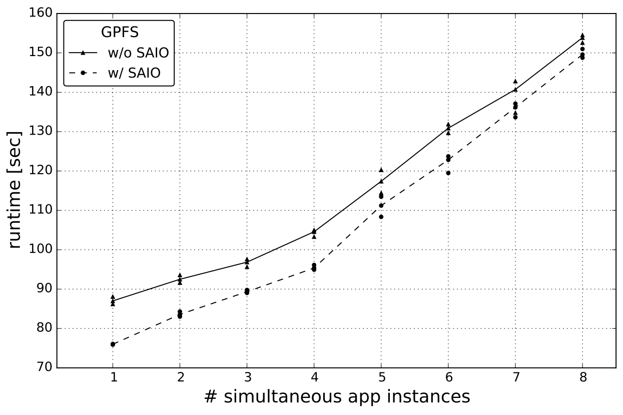


(c)

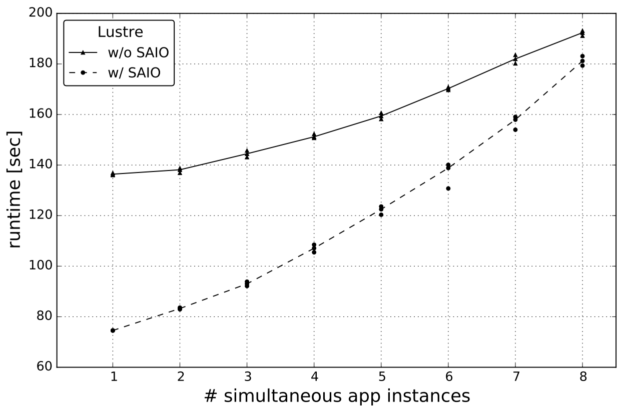
Figure 5.3: Running time of the ROOT application for the three file systems under study using different input file sized (5.3a, 5.3b and 5.3c).



(a)

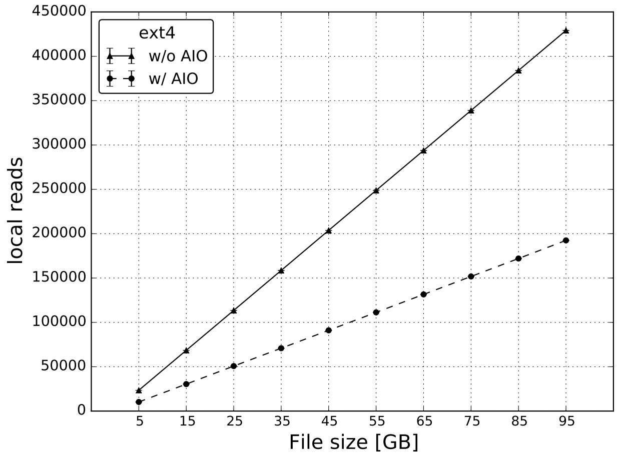


(b)

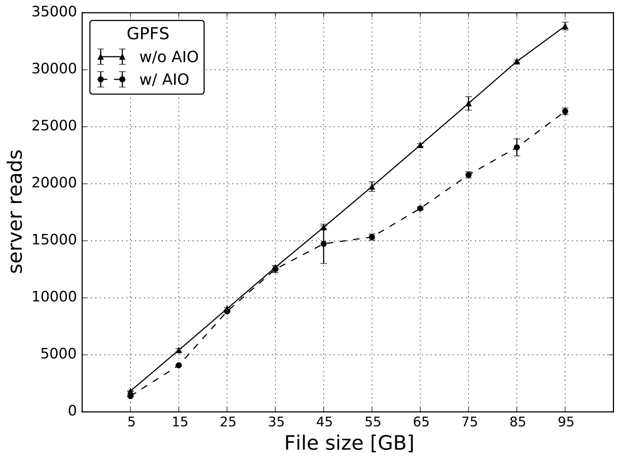


(c)

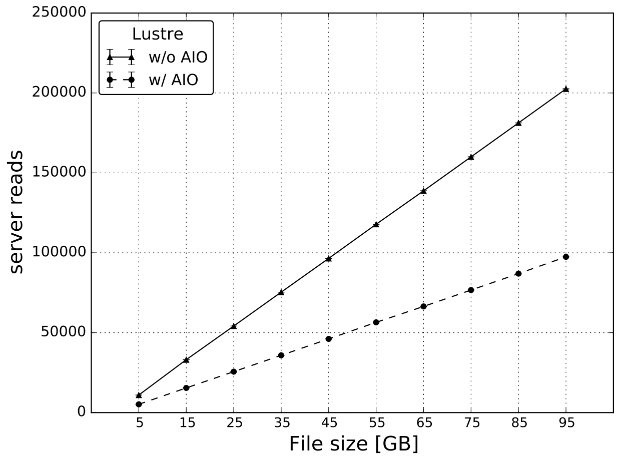
Figure 5.4: Running time of the ROOT application for the three file system under study using different of application instances accessing a file of 5 GB (5.4a, 5.4b and 5.4c).



(a)

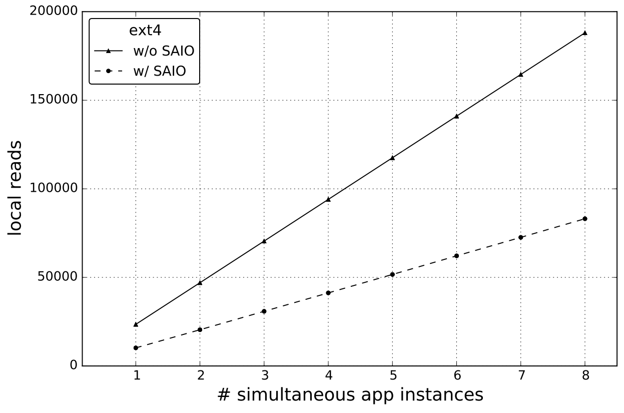


(b)

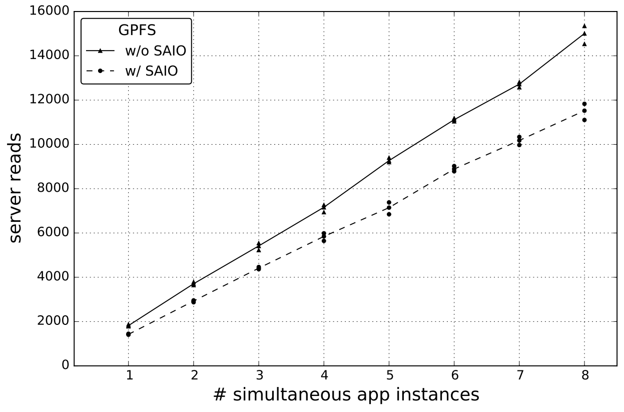


(c)

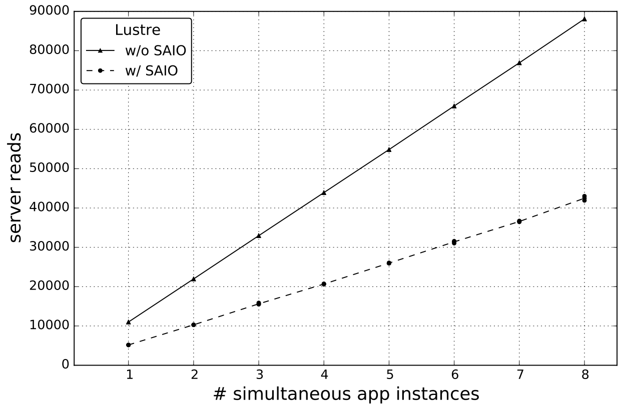
Figure 5.5: Reads processed by local ext4, GPFS and Lustre I/O servers for various input file sizes (5.5a, 5.5b and 5.5c).



(a)



(b)



(c)

Figure 5.6: Reads processed by local ext4, GPFS and Lustre I/O servers for multiple instances of ROOT accessing a file of 5 GB (5.6a, 5.6b and 5.6c).

Figure 5.5a, 5.5b and 5.5c report the number of read requests accounted for by the different file systems under study. More specifically, 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 5.6a, 5.6b and 5.6c).

5.1.3 Conclusions

Our experiments have focused on prefetching performance in a real scenario setup. In particular we have considered the cache utilization by a single application's instance as well as the combined cache utilization by multiple application's instances. We have shown how in both cases we can reduce the application runtime by aggressively prefetching data into main memory using our transparent approach. However, since our strategy is bandwidth bound we have seen that in the case of multiple application's instances such approach leads to the saturation of the network link in the case of networked file systems and ultimately to the shrinking of the performance gain.

5.2 Write Behind in Collective I/O

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[4], Flash-IO and IOR. The main difference between these three benchmarks is the amount of data written per I/O and access pattern. In fact, coll_perf writes all the strided data (32 GB) in a single collective I/O operation using `MPI_File_write_all()`, Flash-IO writes small amounts of strided data (in the order of few MB) over multiple collective I/O operations using `MPI_File_write_at_all()`, and finally IOR writes larger amounts of contiguous data than Flash-IO (4 GB) over multiple collective I/O operations.

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 did not support writing to multiple files or the emulation of computing time between writes. Thus, we modified them to reproduce a workflow

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

similar to the one shown in Figure 4.10. The number of written files and a compute time 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 the described 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 4.3).

The different contributions within the collective I/O write path shown in Figure 4.1 are extracted from the ROMIO layer using MPE profiling [GLS14]. Whenever the compute delay is not enough to hide synchronisation cost (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.

5.2.1 Testbed

Our testbed is a research cluster designed and developed in the context of the DEEP/-ER [ELMS13] 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 [Hei02] (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[5] (PSMPI) version 5.1.0-1 as message passing library.

5.2.2 Performance Results

We measured collective I/O performance using Formula 4.2 and 4.3 for the application perceived bandwidth. Additionally, we also measured the effect of the single ext2ph contributions to collective I/O, as reported in Figure 4.1. In the rest of this section we present our findings for the three target benchmarks.

Coll_perf

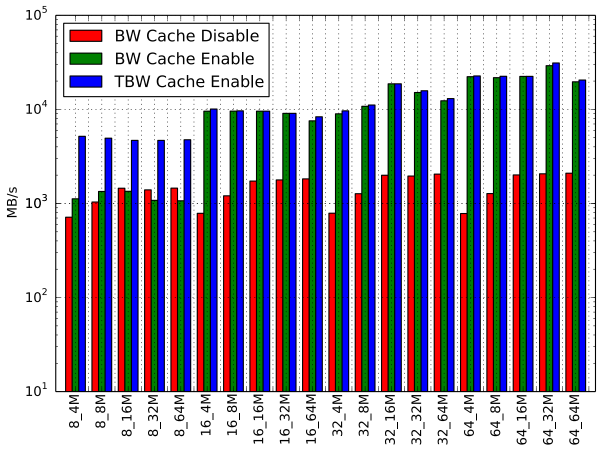
Coll_perf is a synthetic benchmark that performs collective I/O to a shared file using a single collective write operation. The application uses a three-dimensional dataset partitioned and assigned to available processes using a block-block strategy [BdRC93], that is, the original input domain is divided into a number of blocks equal to the number of processes, and each is assigned to a process. Each block is afterwards written to the file using a row-major order. In our configuration we use 512 processes distributed over 64 nodes of the 128 available in the DEEP cluster. Every process handles a block with $256 \times 256 \times 256$ integer elements, for a total 64 MB of data per process and 32 GB for the whole file.

In our experiments we measured three values of write bandwidth:

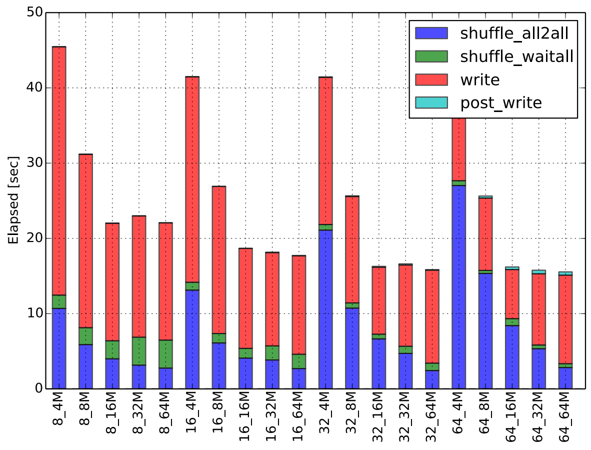
- the bandwidth when the cache is disabled (this is the default case) and collective I/O writes data to the global file system directly (*BW Cache Disable*);
- the bandwidth when the cache is enabled and data is flushed immediately to the global file system (*BW Cache Enable*); and
- the bandwidth when the cache is enabled but data is not flushed to the global file system (*TBW Cache Enable*).

The latter gives us a theoretical bandwidth figure (TBW) that we can use to estimate how well the cached collective I/O implementation is doing. In order to measure the potential benefits of our cached approach we modify the workflow in Figure 4.10 moving the write phase before the compute phase. In this way we can always overlap cache synchronization with compute.

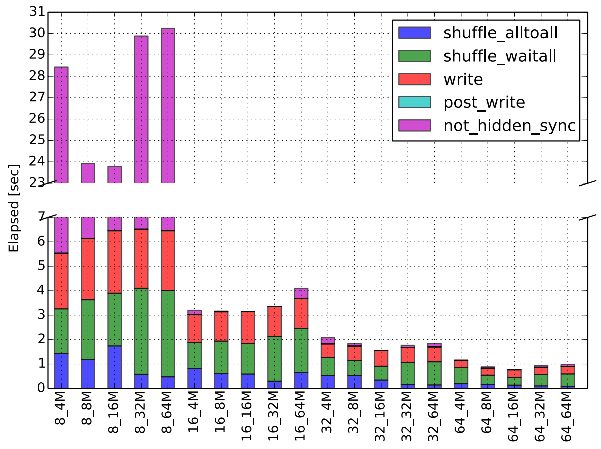
[5]http://www.par-tec.com/products/parastation-mpi.html.



(a)



(b)



(c)

Figure 5.7: Perceived and theoretical write bandwidth for all combinations of aggregators and collective buffer sizes (5.7a); collective I/O contribution breakdown when cache is disabled (5.7b); collective I/O contribution breakdown when cache is enabled (5.7c).

Figure 5.7 shows the results for the perceived and theoretical write bandwidth (5.7a), as well as the single performance contributions for standard collective I/O (5.7b) and cached collective I/O (5.7c). We start by analyzing the behaviour of standard collective I/O in Figure 5.7b.

Let us start by considering the effect of different collective buffer sizes on the collective write time. To this purpose we fix the number of aggregators and look at the time contributions when increasing the buffer size from 4 MB to 64 MB. We first observe that the global synchronization cost (*shuffle_ all2all*) decreases. This is consistent with the ext2ph algorithm behaviour because, as we have already said, increasing the collective buffer size decreases the number of two phase I/O rounds and therefore the number of global synchronization events (`MPI_Alltoall()`). The write cost associated to POSIX write operations also decreases because we are writing to more I/O servers at the same time (recall that we are using four I/O servers and a stripe size of 4 MB). When the buffer size is 4 MB we only write to one I/O server, when the buffer size is 16 MB we write to all of them. This also explains why further increasing the buffer size does not reduce the write cost. The communication cost (*shuffle_ waitall*), on the other hand, increases with the buffer size. This happens because, although the total amount of data shuffled remains constant, the amount of data shuffled during every round of two phase I/O increases, saturating the aggregators network bandwidth. When we use 8 aggregators, for example, up to 32 MB of data are shuffled across the network when using 4 MB buffers, up to 512 MB with 64 MB buffers. Finally, we can look at what happens when we keep the buffer size constant and increase the number of aggregators. In this case we notice that the global synchronization cost increases. This is due to the increased network concurrency at the I/O servers. Finally, *post_ write* time, represented by `MPI_Allreduce()`, does not contribute considerably to overall performance because the corresponding global synchronization event is only encountered once.

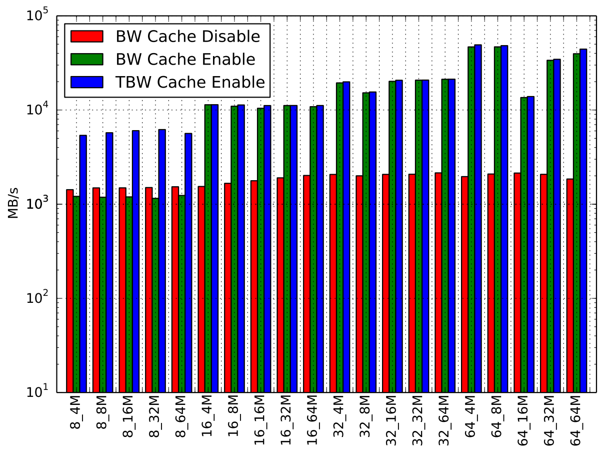
We now look at what happens to the ext2ph contributions when we use the cache (Figure 5.7c). The global synchronization cost can be still reduced by increasing the collective buffer size, but because local SSDs are not shared with other nodes across the network, I/O requests can be served in a more predictable way. This allows aggregators to complete I/O faster and consequently reduces the global synchronization overhead related to collective communications. Write time is not affected visibly by larger buffers because every aggregator writes data locally. Communication cost is not affected because the communication pattern does not change with respect to the non-cached collective I/O case. Additionally, we now have a new contribution representing the cache synchronization time that cannot be overlapped with computation (*not_hidden_sync*). We can observe that this contribution is consistent only for 8 aggregators. In fact, when using 8 aggregators the aggregated bandwidth provided by the local SSD devices cannot match the parallel file system.

The described behaviours are reflected on the perceived write bandwidth shown in Figure 5.7a. In particular we observe that only the 8 aggregators configuration achieves worse performance than the standard collective I/O case. All the remaining configurations always achieve better performance and can provide up to 30 GB/s when using 64 aggregators and 32 MB buffer size, an improvement of 15 times. We also observe that the aggregated bandwidth can be scaled by increasing the number of aggregators and thus the number of available SSD devices. Finally, when using the cache, the buffer size has limited impact on write bandwidth meaning that we can achieve good performance with small buffer sizes, reducing the pressure of collective I/O on system memory.

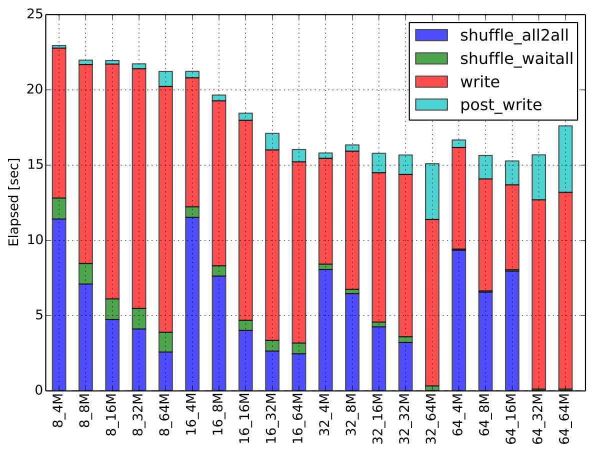
Flash-IO

Flash-IO is the I/O kernel of the Flash application [RCD+00]. The Flash-IO benchmark writes three different files, a checkpoint file a plot file with centered data and a plot file with corner data. The checkpoint file is written using either parallel HDF5 or PnetCDF. We modified the benchmark to only write the checkpoint file using parallel HDF5. In our configuration, every process in the Flash application writes 80 blocks. Each block contains 16 zones, and each zone has 24 variables encoded with 8 bytes, for a total of 768 KBs/block/proc, for a total file size of about 30 GB. Blocks are not written to the checkpoint file with a single collective write operation, like in coll_perf, but instead using multiple collective operations through `MPI_File_write_at_all()`. Similarly to coll_perf we have modified the Flash-IO workflow to completely hide the cache synchronization cost.

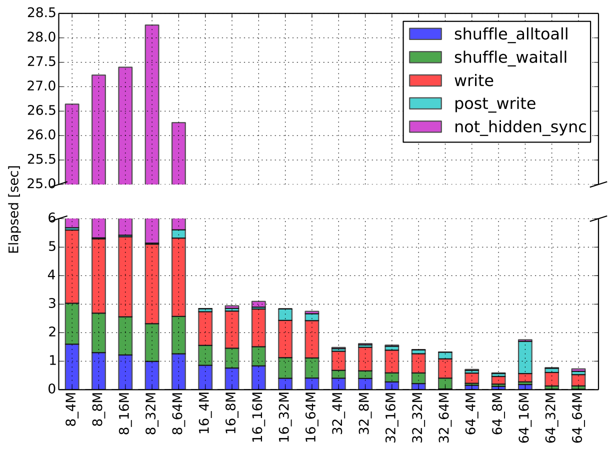
Like in coll_perf, for Flash-IO we measured the same performance parameters varying the number of aggregators and collective buffer size. Results are shown in Figure 5.8. When the cache is disabled, again we observe reduction of the global synchronization cost for increasing buffer sizes. Communication cost this time does not increase with the buffer size. In fact, the file domain size varies from about 49 MB to 6 MB, for 8 and 64 aggregators respectively. This is much less than the amount of data exchanged in coll_perf and thus can be better served by the network infrastructure. Write performance, on the other hand, get worse as the buffer size increases. This behaviour is probably due to the fact that,



(a)



(b)



(c)

Figure 5.8: Perceived I/O bandwidth for all combinations of aggregators and collective buffer sizes (5.8a); collective I/O contribution breakdown when cache is disabled (5.8b); collective I/O contribution breakdown when cache is enabled (5.8c).

unlike in coll_perf, writes are not multiple of the stripe size and therefore there might be additional locking overhead at the file system level. Finally, we observe that the *post_write* contribution is more consistent this time. The reason, as already anticipated, is that Flash-IO does not write data with a single collective operation but instead uses multiple operations, thus increasing the number of `MPI_Allreduce()` calls at the end of the ext2ph algorithm.

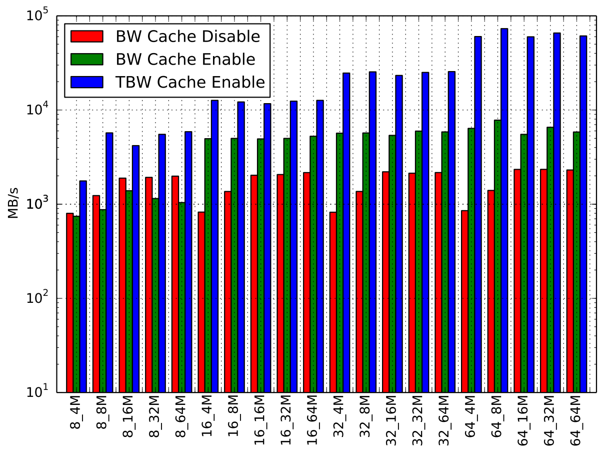
When the cache is enabled we can observe much better performance. Once again, 8 aggregators are not sufficient to completely hide cache synchronization cost to the application. As last note, we see that in the case of 64 aggregators and 16 MB buffer size there is a drop of write bandwidth in Figure 5.8a due to a peak in the *post_write* contribution in Figure 5.8c. This tells us that when using the cache, although we can minimize the scheduling effects on I/O requests, a small variation in the I/O time across aggregators can produce substantial effects on the perceived bandwidth. Indeed, we can see a drop of about 25 GB/s with respect to the 64 aggregators and 32 MB buffer size.

IOR

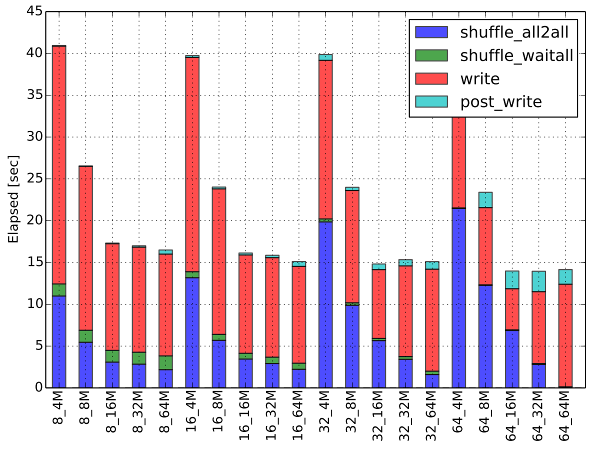
IOR[6] is a parallel I/O benchmark that supports both independent and collective I/O operations using a variety of interfaces including POSIX-IO, MPI-IO and HDF5. Although IOR supports collective I/O operations it does not allow users to define strided patterns (like the one in coll_-perf) using file views. Strided layouts can be built by reading or writing multiple data segments. A segment is a contiguous byte range in the file accessed by only one process. For example, if we have four processes and each of them writes three segments of 64 MB, the first process writes its first segment starting at 0 MB and ending at 64 MB, the second process writes its first segment starting at 64 MB and ending at 128 MB, and so on. When all the processes have written the first segment they initiate another write operation for the second segment starting at offset 256 MB. Every segment is written with an independent collective write operation.

In our experiments we used 8 segments and each of the 512 processes writes 8 MB for segment, for a total of 32 GB file. Unlike the previous two cases, in IOR we follow the workflow depicted in Figure 4.10; meaning that for the last write phase cache synchronization will not be hidden by

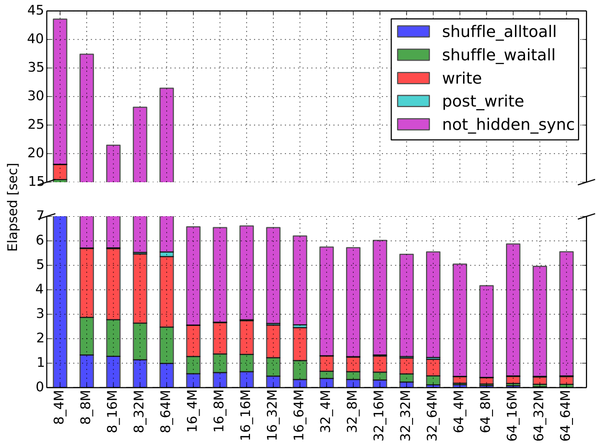
[6] `http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/`.



(a)



(b)



(c)

Figure 5.9: Perceived I/O bandwidth for all combinations of aggregators and collective buffer sizes (5.9a); collective I/O contribution breakdown when cache is disabled (5.9b); collective I/O contribution breakdown when cache is enabled (5.9c).

any following compute phase. This allows us to show what a real world use case would look like.

Figure 5.9 shows the obtained results. These results are similar to the previous except for the fact that now the *not_hidden_sync* contribution is visible for every collective I/O configuration. The cache synchronization cost represents a huge part of the total I/O time. Although, absolute bandwidth performance are lower than previous results we still have that writing data to local SSD devices can give advantages over the baseline ext2ph strategy. In fact, on average we can at least triple the perceived write bandwidth going from about 2 GB/s to 6 GB/s.

The reduction of the cache synchronization cost has not been explored in our work and therefore leaves opening for future optimizations that will be discussed in the conclusion chapter.

5.2.3 Conclusions

In this section we have analyzed collective write performance using a range of different benchmarks that use both synthetic and real I/O patterns to transfer large checkpoint buffers to the parallel file system. Our tests show how collective I/O is mainly limited by the ext2ph algorithm global synchronization overhead and how the use of fast local non-volatile memories is effective in reducing the impact that global file system latency has on such synchronization by taking it out of the critical I/O path. Our work practically proves how next generation I/O systems can benefits from multi tier storage and how first tier non-volatile storage devices can be leveraged by giving the user more control over them.

Chapter 6

Conclusions and Future Work

In this work we have proposed two I/O optimizations for improving performance of applications running on HPC clusters. The first optimization focuses on read patterns that are adopted by analytics codes but also simulation and visualization tools; the second optimization focuses on checkpointing patterns using the collective I/O strategy. For both the optimizations we have implemented a working prototype and demonstrated the effectiveness of the approach in a real setup. In the case of Mercury our solution is simple but effective at the same time and can transparently communicate prefetching information to the operating system without any intervention from the program.

In Mercury we do not focus on the modelling of the application's I/O behaviour but instead show how existing file systems' hints interfaces can be used to guide data prefetching. Our prefetching strategy relies on a configuration file that can adapt to different scenarios; however, it lacks a tighter integration with the operating system that would allow more accurate prefetching. In this direction one future optimization will include a page cache communication function like in [VFK09]. With a better integrated virtual memory management the choice of having a centralized *advice manager*, responsible for all the prefetches, can be even more effective since all the information is concentrated in one place and can be exploited for a more holistic memory management strategy that takes into account many applications. Moreover, because the *resource manager* receives access information from several applications, access pattern correlations among these can be better captured and analyzed.

In the case of checkpoint patterns, on the other hand, we have taken advantage of fast solid state drives installed on compute nodes to boost collective I/O performance. Our solution provides better scalability (by increasing linearly the number of drives with the number of nodes used for I/O), better I/O stability (because the parallel file system is no longer involved into global MPI synchronization operations), and better memory utilization (because solid state drives do not require extremely large writes to work efficiently). Moreover, our solution is simple and can be used by adding a few lines of extra code to the program to select the appropriate MPI-IO hints.

In our study we have focused on the improvement of collective writes from the application to local SSDs in compute nodes while we have ignored the independent writes from local storage to global file system. Although independent writes are not as sensitive as collective writes, they can be served better if we coordinate them in order to present to the file system requests that are arranged by increasing offset. In this case a solution similar to [ZJD09] in which aggregators writes are performed in sequence can be a good option. Moreover, we also plan to extend our implementation with collective read operations because, as said, during restart simulation codes also need to retrieve their checkpoint data. A possibility in this case is to provide a collective prefetching interface that can be used by the run-time system to preload data in the cache upon application restart.

One problem with our presented caching strategy is that it is designed to work in write behind mode, which means that data is committed to permanent global storage as soon as it is available in the cache. This implies that although data can be retained in the cache locally to be used in following application's restarts, this is not possible at the moment. The reason is that data placement information (which data is located in which cache) is not persisted to global storage and therefore, following application's runs have no means to locate it in the different caches. For this reason one important improvement to our design would be the addition of additional logging information to track data placement in the system, in a way similar to [FFS09].

The advent of new non-volatile memory technologies will provide an ever faster storage tier between DRAM and disks and will thus make the parallel file system a second level storage tier; in this case most of the intense I/O activity will be shifted to the file system clients and our prefetching and write-behind approaches go exactly in this direction, leveraging the speed of the first tier for performance and the parallel file system for persistence. Our prefetching approach can exploit high capacity memories to fetch large contiguous portions of the file, filtering non-contiguous requests from clients in NVMM. For write patterns, our MPI-IO collective I/O solution can absorb bursts of write activity in the first storage tier on the clients and move data to the file system only at a second time to make it globally accessible to other applications or workflows.

Bibliography

[AAC09] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[AGPC16] K. Arya, R. Garg, A. Y. Polyakov, and G. Cooperman. Design and implementation for checkpointing of distributed resources using process-level virtualization. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 402–412, Sept 2016.

[AJM$^+$11] C Adam, M John, Adam Manzanares, John Bent, Milo Polte, and Meghan Wingate. LA-UR-11-10426.pdf. 2011.

[And15] C. Andrews. The future of weather forecasting [communications met office supercomputer]. *Engineering Technology*, 10(2):65–67, March 2015.

[ASC10] The opportunities and challenges of exascale computing. Report, 2010.

[BBLS91] D Bailey, J Barton, T Lasinski, and H Simon. The NAS Parallel Benchmarks. Technical report, NASA, 1991.

[BC08] Daniel Bovet and Marco Cesati. *Understanding the linux kernel 3rd edition*. O'Reilly, 2008.

[BCS$^+$08] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel i/o prefetching using mpi file caching and i/o signatures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 44:1–44:12, Piscataway, NJ, USA, 2008. IEEE Press.

[BdRC93] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhayr. Design and evaluation of primitives for parallel I/O. *Proceedings of the Supercomputing Conference*, pages 452–461, 1993.

[BGG+09] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Nov 2009.

[BM00] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[BMK01] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, 19(2):111–170, May 2001.

[Bra02] Peter J. Braam. Lustre: a scalable high-performance file system. White Paper, November 2002.

[Bre08] M. J. Breitwisch. Phase change memory. In *2008 International Interconnect Technology Conference*, pages 219–221, June 2008.

[CBS+08] 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.

[CCC+03] A. Ching, A. Choudhary, K. Coloma, Wei keng Liao, R. Ross, and W. Gropp. Noncontiguous i/o accesses through mpi-io. In *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pages 104–111, May 2003.

[CCkL+02] Avery Ching, A. Choudhary, Wei keng Liao, R. Ross, and W. Gropp. Noncontiguous i/o through pvfs. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 405–414, 2002.

[CCL$^+$04] 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.

[CCL$^+$06] 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 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006.

[CG99] 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.

[CGML01] Fay W Chang, Garth A Gibson, Todd C Mowry, and James R Larus. *Using speculative execution to automatically hide I / O latency.* PhD thesis, 2001.

[CGNB12] G. Congiu, M. Grawinkel, S. Narasimhamurthy, and A. Brinkmann. One phase commit: A low overhead atomic commitment protocol for scalable metadata services. In *2012 IEEE International Conference on Cluster Computing Workshops*, pages 16–24, Sept 2012.

[CGP$^+$14] G. Congiu, M. Grawinkel, F. Padua, J. Morse, T. Süß, and A. Brinkmann. Poster: Optimizing scientific file i/o patterns using advice based knowledge. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 282–283, Sept 2014.

[CGP$^+$17] G. Congiu, M. Grawinkel, F. Padua, J. Morse, T. Süß, and A. Brinkmann. Mercury: A transparent guided i/o framework for high performance i/o stacks. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 46–53, March 2017.

[Cha11a] Measurement of w+w- production and search for the higgs boson in pp collisions at $\sqrt{s} = 7$ tev. *Physics Letters B*, 699(1-2):25 – 47, 2011.

[CHA$^+$11b] 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.

[CLI$^+$00] 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.

[CNSB16] G. Congiu, S. Narasimhamurthy, T. Süß, and A. Brinkmann. Improving collective i/o performance using non-volatile memory devices. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 120–129, Sept 2016.

[CR10] 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.

[CST$^+$11] 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* 25th *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 794–804, Anchorage, Alaska, USA, May 2011.

[DAC$^+$12] 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.

[Dal06] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, February 2006.

[DLMM13] Jan Deca, Giovanni Lapenta, Richard Marchand, and S Markidis. Spacecraft charging analysis with the implicit particle-in-cell code ipic3d. 20:2902–, 10 2013.

[dRBC93] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, December 1993.

[ELMS13] N. Eicker, T. Lippert, T. Moschny, and E. Suarez. The deep project - pursuing cluster-computing in the many-core era. In *2013 42nd International Conference on Parallel Processing*, pages 885–892, Oct 2013.

[Ene15] Basic Energy. Basic Energy Sciences Exascale Requirements Review. 2015.

[ERA16] D O E Exascale, Requirements Review, and B E R Ascr. Biological and Environmental Research Exascale requirements review. 2016.

[FCY99] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of SC'99*, Portland, OR, November 1999.

[FFS09] Jens Freche, Wolfgang Frings, and Godehard Sutmann. High-throughput parallel-i/o using sionlib for mesoscopic particle dynamics simulations on massively parallel computers. In *PARCO*, pages 371–378, 2009.

[FOR$^+$00] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal, Supplement*, 131:273–334, 2000.

[FSP$^+$08] Rosa Filgueira, David E. Singh, Juan C. Pichel, Florin Isaila, and Jesús Carretero. Data Locality Aware Strategy for Two-Phase Collective I/O. *High Performance Computing for Computational Science - VECPAR 2008*, 5336:137–149, 2008.

[FWP09] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel i/o to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[FXM04] Zhihua Fan, Jin Xiong, and Jie Ma. A failure recovery mechanism for distributed metadata servers in dcfs2. In *Proceedings. Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region, 2004.*, pages 2–8, July 2004.

[GH96] E. Grochowski and R. F. Hoyt. Future trends in hard disk drives. *IEEE Transactions on Magnetics*, 32(3):1850–1854, May 1996.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[GLS14] William Gropp, Ewing Lusk, and Anthony Skjellum. *The MPE Multiprocessing Environment*, pages 336–. MIT Press, 2014.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[HBT$^+$13] 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.

[Hei02] Jan Heichler. An introduction to BeeGFS. (November):1–11, 2002.

[HSS$^+$11] 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.

[HST12] 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.

[ID01] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 117–130, New York, NY, USA, 2001. ACM.

[IRYB08] Kamil Iskra, John W. Romein, Kazutomo Yoshii, and Pete Beckman. Zoid: I/o-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 153–162, New York, NY, USA, 2008. ACM.

[Isa13] John Isaac. CFD in the Aerospace and Aeronautics Industries. *Journal of Aeronautics & Aerospace Engineering*, 02(03):2–5, 2013.

[JW91] David M Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position.* Hewlett-Packard Laboratories Palo Alto, CA, 1991.

[KGS$^+$10] Y. Kim, R. Gunasekaran, G. M. Shipman, D. A. Dillow, Zhe Zhang, and B. W. Settlemyer. Workload characterization of a leadership class storage cluster. In *Petascale Data Storage Workshop (PDSW), 2010 5th*, pages 1–5, Nov 2010.

[KGS$^+$16] J. Kaiser, R. Gad, T. Süß, F. Padua, L. Nagel, and A. Brinkmann. Deduplication potential of hpc applications' checkpoints. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 413–422, Sept 2016.

[KGW16] N. Katzburg, A. Golander, and S. Weiss. Storage becomes first class memory. In *2016 IEEE International Conference on the Science of Electrical Engineering (ICSEE)*, pages 1–5, Nov 2016.

[kL11] W. k. 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, Feb 2011.

[kLCC$^+$05] Wei keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman. Collective caching: application-aware client-side file caching. In *HPDC-14. Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, 2005.*, pages 81–90, July 2005.

[Kot94] David Kotz. Disk-directed i/o for mimd multiprocessors. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[LC08] 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.

[LCC$^+$12] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, April 2012.

[LCTZ12] Y. Lu, Y. Chen, R. Thakur, and Y. Zhuang. Poster: Memory-conscious collective i/o for extreme-scale hpc systems. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1362–1362, Nov 2012.

[LCZ13] 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.

[LCZT13] Yin Lu, Yong Chen, Yu Zhuang, and Rajeev Thakur. Memory-conscious collective i/o for extreme scale hpc systems. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 5:1–5:8, New York, NY, USA, 2013. ACM.

[LIMB09] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[LJM$^+$16] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton. Daos and friends: A proposal for an exascale storage system. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 585–596, Nov 2016.

[LkLC$^+$03] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39, Nov 2003.

[LKS$^+$08] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[LLT$^+$14] 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.

[LRT$^+$04] 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.

[Mac11] C. A. Mack. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, May 2011.

[MAI$^+$84] F. Masuoka, M. Asano, H. Iwahashi, T. Komuro, and S. Tanaka. A new flash e2prom cell using triple polysilicon technology. In *1984 International Electron Devices Meeting*, volume 30, pages 464–467, 1984.

[MB15] F. Margaglia and A. Brinkmann. Improving mlc flash performance and endurance with extended p/e cycles. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2015.

[MBMdS10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. (November):1–11, 2010.

[MBMS10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[MDK96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 3–17, New York, NY, USA, 1996. ACM.

[MHH$^+$05] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, Sept 2005.

[MHS15] By Hattori Masakatsu, Suzuki Hiroshi, and Sugaya Seiichi. Trends in Technologies for HDDs , ODDs , and SSDs , and Toshiba ' s Approach. *TOSHIBA Storage Product s for ICT Society*, pages 2–9, 2015.

[MLRu10] Stefano Markidis, Giovanni Lapenta, and Rizwan-uddin. Multi-scale simulations of plasma with ipic3d. *Math. Comput. Simul.*, 80(7):1509 – 1519, March 2010.

[mpi12] MPI: a message-passing interface standard. Report, 2012.

[MWLY03] 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.

[New83] Paul Newbold. Arima model building and the time series analysis approach to forecasting. 2:23 – 35, 01 1983.

[NIS99] The future of data storage technologies. Report, 1999.

[NKP$^+$96] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Sclatter Ellis, and M. L. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, Oct 1996.

[PG11] Swapnil Patil and Garth Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[PGG$^+$95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 79–95, New York, NY, USA, 1995. ACM.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International*

Conference on Management of Data, SIGMOD ’88, pages 109–116, New York, NY, USA, 1988. ACM.

[Phy15] High Energy Physics. High Energy Physics Exascale requirements review. 2015.

[POS] Standard for information technology - portable operating system interface (posix).

[PTH$^+$00] 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 6th International Euro-Par Conference (Euro-Par)*, pages 1253–1262. Munich, Germany, August 2000.

[PTH$^+$01] 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.

[RCD$^+$00] R. Rosner, A. Calder, J. Dursi, B. Fryxell, D. Q. Lamb, J. C. Niemeyer, K. Olson, P. Ricker, F. X. Timmes, J. W. Truran, H. Tueo, Yuan-Nan Young, M. Zingale, E. Lusk, and R. Stevens. Flash code: studying astrophysical thermonuclear flashes. *Computing in Science Engineering*, 2(2):33–41, March 2000.

[RW94] Chris Ruemmler and John Wilkes. Introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[SC90] J. W. Stamos and F. Cristian. A low-cost atomic commit protocol. In *Proceedings Ninth Symposium on Reliable Distributed Systems*, pages 66–75, Oct 1990.

[Sci16a] Fusion Energy Sciences. Fusion energy Exascale requirements review. 2016.

[Sci16b] Advanced Scientific. Nuclear physics, Exascale requirements review. 2016.

[SCJ$^+$95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Proceedings of the IEEE/ACM SC95 Conference*, pages 57–57, 1995.

[SFT$^+$09] Y. Shiroishi, K. Fukuda, I. Tagawa, H. Iwasaki, S. Takenoiri, H. Tanaka, H. Mutoh, and N. Yoshikawa. Future options for hdd storage. *IEEE Transactions on Magnetics*, 45(10):3816–3822, Oct 2009.

[SG06] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[SG07] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78:012022, 2007.

[SH02] 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.

[SHK11] G. Sobhaninejad, M. Hori, and T. Kabeyasawa. Enhancing integrated earthquake simulation with high performance computing. *Adv. Eng. Softw.*, 42(5):286–292, May 2011.

[Sho03] Frank Shorter. *Design and analysis of a performance evaluation standard for parallel file systems.* PhD thesis, Clemson University, 2003.

[SR98] Huseyin Simitci and Daniel A. Reed. A comparison of logical and physical parallel i/o patterns. *International Journal of High Performance Computing Applications*, 12:364–380, 1998.

[SR13] 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.

[SRF$^+$99] Huseyin Simitci, Daniel A. Reed, Ryan Fox, Mario Medina, James Oly, Nancy Tran, and Guoyi Wang. A framework for adaptive storage input/output on computational grids. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 519–532, London, UK, UK, 1999. Springer-Verlag.

[SSH$^+$10] Shafeeq Sinnamohideen, Raja R. Sambasivan, James Hendricks, Likun Liu, and Gregory R. Ganger. A transparently-scalable metadata service for the ursa minor storage system. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

[SSL$^+$13] Saba Sehrish, Seung Woo Son, Wei-keng Liao, Alok Choudhary, and Karen Schuchardt. Improving collective i/o performance by pipelining request aggregation and file access. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 37–42, New York, NY, USA, 2013. ACM.

[TC96] 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.

[TGL96] 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.

[TGL99] 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.

[TR04] 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.

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

[VFK09] 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.

[VGL$^+$13] Jean-André Vital, Michael Gaurut, Romain Lardy, Nicolas Viovy, Jean-François Soussana, Gianni Bellocchi, and Raphael Martin. High-performance computing for climate change impact studies with the pasture simulation model. 98:131–135, 10 2013.

[WAA13] K L Wang, J G Alzate, and P Khalili Amiri. Low-power non-volatile spintronic memory: Stt-ram and beyond. *Journal of Physics D: Applied Physics*, 46(7):074003, 2013.

[WG04] Brent Welch and Garth Gibson. Managing scalability in object storage systems for hpc linux clusters. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 433–445, 2004.

[WGP94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 241–251, New York, NY, USA, 1994. ACM.

[WMES07] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. A job pause service under lam/mpi+blcr for transparent fault tolerance. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[WOW$^+$14] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 71–79, Oct 2014.

[WSSZ07] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *Trans. Storage*, 3(2), June 2007.

[WUA$^+$08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[WXH$^+$04] Feng Wang, Qin Xin, Bo Hong, Scott a Brandt, Ethan L Miller, Darrell D E Long, and Tyce T McLarty. File System Workload Analysis for Large Scale Scientific Computing Applications. *12th NASA Goddard Conference on Mass Storage Systems and Technologies*, (April 2004):139–152, 2004.

[XS16] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In

14th USENIX Conference on File and Storage Technologies (FAST 16), pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[Yin08] Liu Ying. Lustre ADIO collective write driver. White Paper, October 2008.

[YV08] 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.

[YVCJ07] W. Yu, J. Vetter, R. S. Canon, and S. Jiang. Exploiting lustre file joining for effective collective io. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 267–274, May 2007.

[ZGF^{+}15] Z. Zhang, B. Gao, Z. Fang, X. Wang, Y. Tang, J. Sohn, H. S. P. Wong, S. S. Wong, and G. Q. Lo. All-metal-nitride rram devices. *IEEE Electron Device Letters*, 36(1):29–31, Jan 2015.

[ZJD09] 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.

[ZK01] Zheng Zhang and C. Karamanolis. Designing a robust namespace for distributed file services. In *Proceedings 20th IEEE Symposium on Reliable Distributed Systems*, pages 162–171, 2001.

Curriculum Vitae

Work Experience

Sep 2017	Pre-Doctoral Appointee at Argonne National Laboratory, USA *Working on programming models and runtime systems in the Mathematics and Computer Science division* Study of heterogeneous memory systems in message passing (MPI) communications.
May 2017 Dec 2013	Research Engineer at Seagate System UK Ltd, Havant *Worked on EU funded research projects* Design and implementation of parallel I/O improvements for the ROMIO middleware (MPI-IO implementation from Argonne National Laboratory), during the course of the DEEP-ER project (http://www.github.com/gcongiu/E10.git). Basic working experience with scientific I/O libraries (HDF5, NetCDF) gained during the course of the SAGE and ESiWACE projects, with special focus on HDF5 plugin development for different storage backends.
Dec 2013 Feb 2011	Early Stage Researcher at Xyratex Ltd, Havant *Marie Curie ITN Fellow* Covered the role of research fellow in the SCALUS (SCALig by mean of Ubiquitous Storage, GA no. 238808) project (EU Funded Marie Curie ITN program). Covered research topics included Guided I/O mechanisms for efficient I/O (http://www.github.com/gcongiu/Mercury.git) and parallel I/O techniques.
Aug 2010 Feb 2009	Software Developer at Sardegna Ricerche & IBM Italy, Pula Worked at the MIACell Project (Medical Image Analysis on Cell broadband engine).

Education

December 2008	Master of Science in Electrical & Electronic Engineering, **University of Cagliari** Thesis: "Cell BE: Performance Analysis of the Element Interconnect Bus and Development of an Alternative Packed Switched Solution" | Advisor: Prof. Luigi Raffo
September 2005	Undergraduate Degree in Electrical & Electronic Engineering **University of Cagliari**

Scholarships and Certificates

February 2016, Marie Curie Initial Training Network Certificate

Professional Activities

Technical Reviewer for International Journals

- Elsevier Journal of Parallel and Distributed Computing (2017 – 2018)
- IEEE Transaction on Parallel and Distributed Systems (2017 – 2018)

Technical Reviewer for International Conferences and Workshops

- IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2015 – 2016)

Publications

- G. Congiu, M. Grawinkel, S. Narasimhamurthy, and A. Brinkmann, ”One Phase Commit: A Low Overhead Atomic Commitment Protocol for Scalable Metadata Services”, *2012 IEEE International Conference on Cluster Computing Workshops*, Beijing, 2012, pp. 16-24. doi: 10.1109/ClusterW.2012.16
- G. Congiu, M. Grawinkel, F. Padua, J. Morse, T. Süß, and A. Brinkmann, ”POSTER: Optimizing scientific file I/O patterns using advice based knowledge”, *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Madrid, 2014, pp. 282-283. doi: 10.1109/CLUSTER.2014.6968763
- G. Congiu, S. Narasimhamurthy, T. Süß, and A. Brinkmann, ”Improving Collective I/O Performance Using Non-volatile Memory Devices”, *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Taipei, 2016, pp. 120-129. doi: 10.1109/CLUSTER.2016.37

- G. Congiu, M. Grawinkel, F. Padua, J. Morse, T. Süß, and A. Brinkmann, "MERCURY: A Transparent Guided I/O Framework for High Performance I/O Stacks", *2017 IEEE Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, St. Petersburg, 2017. doi: 10.1109/PDP.2017.83