Design and evaluation of multiple level data staging for Blue Gene systems

Florin Isaila, Javier Garcia Blas, Jesus Carretero - *University Carlos III of Madrid*
Robert Latham, Robert Ross - *Argonne National Laboratory*

***Abstract*—Parallel applications currently suffer from a significant imbalance between computational power and available I/O bandwidth. Additionally, the hierarchical organization of current Petascale systems contributes to an increase of the I/O subsystem latency. In these hierarchies, file access involves pipelining data through several networks with incremental latencies and higher probability of congestion. Future Exascale systems are likely to share this trait.**

This paper presents a scalable parallel I/O software system designed to transparently hide the latency of file system accesses to applications on these platforms. Our solution takes advantage of the hierarchy of networks involved in file accesses, to maximize the degree of overlap between computation, file I/O-related communication and file system access. We describe and evaluate a two-level hierarchy for Blue Gene systems consisting of client-side and I/O node-side caching. Our file cache management modules coordinate the data staging between application and storage through the Blue Gene networks. The experimental results demonstrate that our architecture achieves significant performance improvements through a high degree of overlap between computation, communication, and file I/O.

***Index Terms*—MPI-IO, Parallel I/O, Parallel File Systems, Supercomputers.**

I. Introduction

The needs of scientific applications have driven a continuous increase in scale and capability of leading parallel systems [13]. However, the improvement in rates of computation has not been matched by an increase in I/O capabilities. For example, earlier supercomputers maintained a ratio of 1GBps of parallel I/O bandwidth 1GBps for every TFLOP, whereas in current systems 1GBps for every 10TFLOPS [3] is the norm. This increased disparity makes it even more critical that the I/O subsystem be used in the most efficient manner.

Scalable I/O has been already identified as a critical issue for PFLOP systems. Future exascale systems forecasted for 2018-2020 will presumably have O(1B) cores and will be hierarchical in both platform and algorithms [1]. This hierarchy will imply a longer path in moving data from cores to storage and vice-versa, resulting in even higher I/O latencies, relative to the rates of computation and communication in the system.

IBM's Blue Gene supercomputers have a significant share in the Top 500 lists and additionally bring the advantage of a highly energy-efficient solution. Blue Gene systems scale up to hundreds of thousands of cores, tightly inter-connected through a high-performance scalable network. The architectural separation of processing from the I/O system (Figure 1) allow for tight packaging of low-power compute resources, but it implies a higher number of hops from application processes to storage, translating into a higher latency for file access. For instance, file writes are pipelined through three different networks (tree, Myrinet/Ethernet, and storage network) from compute nodes through I/O nodes and storage servers to finally reach storage. Every component of this heterogeneous network contributes latency, reducing the file access performance seen by the applications. Of particular concern is the tree network, shared by all cores for I/O forwarding, which may be the cause of significant bottlenecks if inefficiently used. At the same time, this hierarchical system offers numerous opportunities for performance optimizations through overlapping of computation, communication, and disk I/O, if these activities can be properly orchestrated.

Large computational resources are employed both for highly coupled parallel applications as well as for loosely coupled "many task" computations that communicate via file system operations [28]. Both these categories of applications involve data transfer between compute nodes and storage.

The main goals of this work are to analyze the potential for high-performance I/O in supercomputer platforms and to offer a user-transparent approach for optimization of data transfer between applications and storage. More precisely, we aim to increase the file access performance as seen by parallel applications by hiding the latency of data transfers over the different networks, without requiring any changes to the applications themselves. Given the increasing hierarchy of networks involved in file accesses, our optimizations are focused on maximizing the degree of overlap between computation, file I/O related communication, and file system access.

This paper builds on our previous work [10], in which we presented an asynchronous file access strategy based on a file cache on the I/O nodes in a Blue Gene/L system. On Blue Gene/L, the asynchronous file access approach is limited by the lack of support for threads, preventing further local overlapping of computation and I/O on compute nodes. The limitations are imposed by the non-coherent L1 caches of the two cores, forcing multi-threaded processes to run on a single core. Consequently, on the I/O nodes, an asynchronous data transfer approach could merely hide the latency of transferring the data from I/O nodes to the storage servers: data movement over the tree network could not be hidden. The limitations discussed above have been removed from the Blue Gene/P architecture. The Blue Gene/P has limited multithreading support, but it is adequate for our caching strategies, and the L1 caches of the four cores are coherent. In a further work [4] we extended the solution presented in [10] with a prefetching module on the I/O node and with a write-back module on the

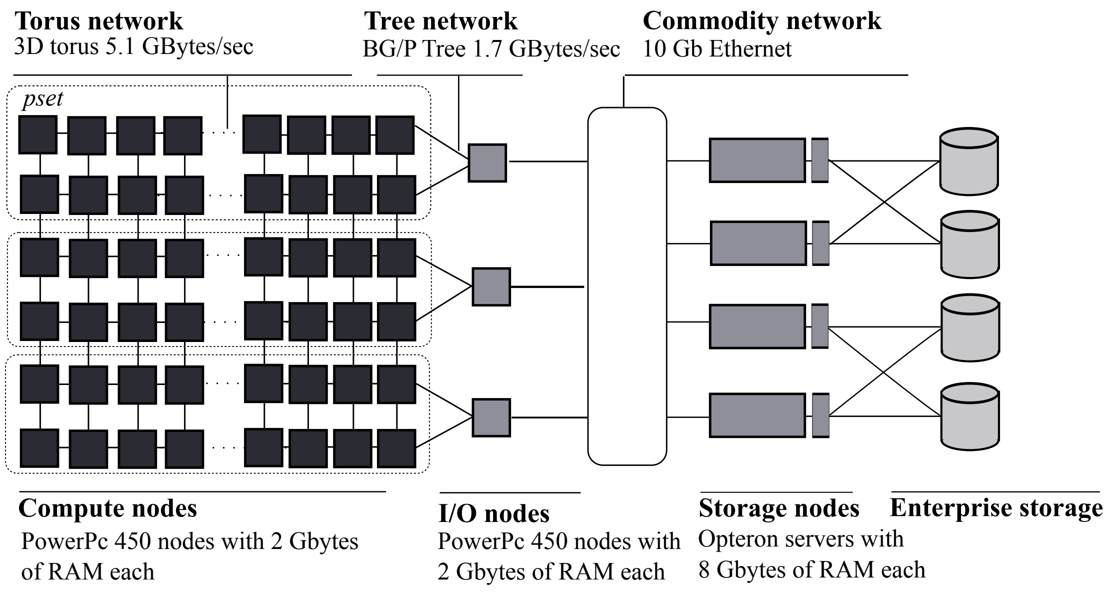


Fig. 1. Blue Gene/P architecture overview. Processing is separated from the I/O nodes and storage nodes. Four-core compute nodes are interconnected through a 3D torus network and are grouped in processing sets (psets). The I/O system calls of all cores in a pset are forwarded to exactly one master I/O node through a tree network. Each I/O node mounts all file systems. The file system servers run on storage nodes connected to the I/O nodes through a commodity network. Disks are attached to the storage nodes through a separate storage network.

compute node, taking advantage of the opportunities afforded by this new hardware capability.

In this paper we further extend this previous work by presenting a fully integrated two-level file cache solution. The design includes a new prefetching module on the compute node and its integration with the prefetching module on the I/O node. Our extensive evaluation answers the following questions: *What is the benefit of employing multiple-level file caching on compute nodes and I/O nodes? Does the use of the torus network for file access optimization pay off? Which asynchronous policies are suitable for data staging (pipelining)? How do policies at different hierarchy levels interact between each other? What coordination is needed? What are good ratios/sizes of file caches on different levels of the hierarchy? What access semantics are appropriate? How to we ensure data consistency?*

The remainder of the paper is structured as follows. Section II reviews related work. The hardware and operating system architectures of the Blue Gene/P are presented in Section III. We discuss our novel solution for Blue Gene/P system in Section V. The experimental results are presented in Section VI. We summarize and discuss future work in Section VIII.

II. Related Work

Latency hiding in file access. Several researchers have contributed techniques for hiding the latency of file system accesses. Zhang et al [33] propose a collective I/O model for loosely coupled applications based on an in-memory file system, located on the compute nodes. While our work also targets the locality of accesses, it differs in its focus on coordinated, multiple-level data staging, including an I/O node caching level. Active buffering is an optimization for MPI-IO collective write operations [22] based on using an I/O thread to manage write-back. Active buffering helps out MPI-IO collective writes, while our approach benefits not only MPI-IO collective writes but independent writes and even POSIX writes.

Write-behind strategies [20] accumulate multiple, small writes into large, contiguous I/O requests in order to better utilize the network bandwidth. In this paper we present a coordinated, multi-level write back strategy: small requests are merged at compute nodes into file blocks asynchronously written to I/O nodes, while I/O nodes simultaneously cache file blocks and write them asynchronously to the storage nodes.

A number of works have proposed I/O prefetching techniques based on application disclosed access patterns. Informed prefetching and caching [26] leverages application-disclosed access patterns in order to make cost-efficient trade offs between prefetching and caching policies. Chang and Gibson [7] propose an automatic prefetching technique based on speculative execution. A similar idea is used in [8] for hiding the latency of read accesses for MPI-IO based accesses. PC-OPT [17] is an off-line prefetching and caching algorithm for parallel I/O systems. When PC-OPT has a priori knowledge of the entire reference sequence, it generates a schedule of minimal length.

In [6] the authors propose an I/O prefetching method based on adaptive I/O signatures derived from file access pattern classifications. In contrast, our prefetching approach is multi-level and is based on views and collective I/O aggregation patterns. DataStager [2] is a one-level data staging framework for CrayXT machines based on a server-pull model. In turn our focus is on data staging in a two-level cache hierarchy of Blue Gene architecture. A performance model for overlapping computation, communication and I/O is presented in [25].

Collective I/O. Collective I/O techniques merge small, individual requests from compute nodes into larger, global requests in order to optimize the network and disk performance. Depending on where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at I/O nodes, the method follows the *disk-directed I/O* [18] approach. If the merging occurs at intermediary nodes or at compute nodes, the method is called *two-phase I/O* [9]. Data shipping [27] is a GPFS I/O optimization that uniquely binds each file block in a round-robin manner to a unique I/O agent. All subsequent read and write operation on the file go through the I/O agents, which ship the requested data between the file system and the appropriate processes. Other works have focused on improving the access locality of collective I/O [19], [15], an approach we are extending to a multiple level hierarchy. While using collective I/O techniques in order to gather small requests on a client-side cache, our approach is suitable to optimize both collective and independent I/O.

Parallel I/O on supercomputers. A limited number of recent studies have proposed and evaluated parallel I/O solutions for supercomputers. An implementation of MPI-IO for Cray architecture and the Lustre file system is described in [36]. In [35] the authors propose a collective I/O technique, in which processes are grouped together for collective I/O according to the Cray XT architecture. Yu et al. [30] present a GPFS-based three-tiered architecture for Blue Gene/L. The tiers are represented by I/O nodes (GPFS clients), network-shared disks, and a storage area network. Our solution focuses on the Blue Gene/P, extends this hierarchy to include the memory of the compute nodes.

III. Blue Gene/P

This section presents the hardware and operating system architectures of Blue Gene/P.

A. Blue Gene/P Architecture

Figure 1 shows a high-level view of a Blue Gene/P system. Compute nodes are grouped into processing sets, or “psets”. Applications run in exclusivity on partitions, consisting of multiples of psets. Each pset has an associated I/O node that performs I/O operations on behalf of the compute nodes from the pset. The file system components run on dedicated file servers connected to storage nodes through a 10Gbit Ethernet switch. Compute and I/O nodes use the same ASIC with four PowerPC 450 cores, with core-private hardware-coherent L1 caches, core-private stream prefetching L2 caches, and a 8 MBbytes of shared DRAM L3 cache.

Blue Gene/P compute nodes are interconnected by a 3D torus (5.1 GBytes/s). A collective network (1700 MBytes/s) with a tree topology provides support for a set of collective communication operations and manages I/O traffic. A commodity 10 Gbit/sec Ethernet network interconnects I/O nodes and file servers.

B. Operating System Architecture

The operating system provided by IBM for the BG/L and BG/P systems [23], the compute node kernel (CNK), is a light-weight operating system offering basic services such as setting an alarm or getting the time. As shown in Figure 2, I/O system calls (e.g. file system calls, socket calls, etc.) are forwarded through the tree collective network to the I/O node by a Remote Procedure Call (RPC)-like mechanism. The forwarded calls are replayed on the I/O node by the control and I/O daemon (CIOD). CIOD executes the requested system calls on locally mounted file systems and returns the results to the compute nodes.

The ZeptoOS project [12] provides an open-source Linux alternative to the IBM CNK. Under ZeptoOS, I/O forwarding is implemented in a component called ZOID, as shown in Figure 3 (a). The I/O forwarding process in ZeptoOS and ZOID is similar to the one based on CIOD, in the sense that I/O related calls are forwarded to the I/O nodes, where a multi-threaded daemon serves them. However, there are two notable differences in design and implementation between CIOD-based and ZOID-based solutions. First, ZOID comes with its own network protocol, which can be conveniently extended with the help of a plug-in tool that automatically generates the communication code for new forwarded calls. Second, the file system calls are forwarded through ZOIDFS [16], an abstract interface for forwarding file system calls. ZOIDFS abstracts away the details of a file system API under a stateless interface consisting of generic functions for file create, open, write, read, close, and so forth. This facilitates experimentation with alternative I/O strategies.

In our solution the I/O node-side cache as well as the data staging modules on the I/O node are implemented under ZOIDFS. A comparison between ZOIDFS based I/O forwarding pipeline and our solution is shown in Figure 3. Further details about our solutions are given in the following sections.

IV. Architecture overview

Our proposed solution is based on the multi-tiered architecture depicted in Figure 4. The five tiers of the architecture are: the application tier, the client-side I/O forwarding tier, the *client-side* file cache management tier, the *I/O-side* file cache management tier, and the storage system tier.

Application tier. Applications run on a set of compute nodes and can be parallel MPI programs or a collection of sequential applications. Access to the file system is performed via MPI-IO or POSIX calls that are translated to forwarded I/O calls.

Client-side I/O forwarding tier. Client-side I/O forwarding pushes file accesses to the next tier through the scalable torus network. This forwarding is performed on-demand, when the application issues a file access (POSIX or MPI-IO). The POSIX interface is implemented using FUSE [11], while MPI-IO support is provided via the ROMIO [32] MPI-IO implementation.

Client-side file cache management tier. The client-side file cache module manages a file cache close to application processes and efficiently transfers data between applications and the I/O subsystem. Because the tree network is shared among all the processes inside a pset, it may represent a bottleneck if used in an uncoordinated manner. The proximity of the cache to application processes enables low-latency

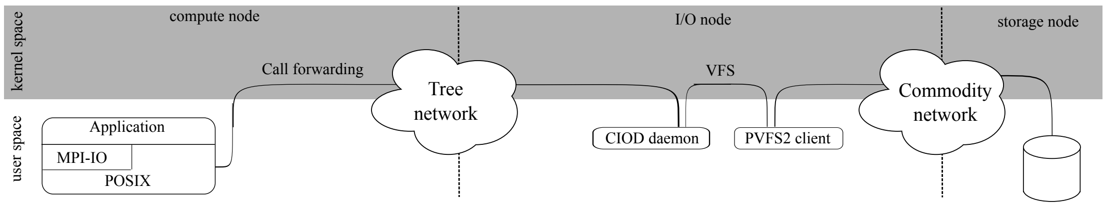


Fig. 2. File I/O forwarding for IBM solution. Applications access the file system through the MPI-IO or POSIX interface. MPI-IO is implemented on top of POSIX file system calls. POSIX calls are forwarded in an RPC-like manner to the I/O nodes. The forwarded calls are served on the I/O node by a user-level daemon called the CIOD. The CIOD executes the file system call on behalf of the compute node through the VFS interface, which communicates with a local PVFS2 client. The PVFS2 client sends the request on to the PVFS2 servers running on the storage nodes. The call return value and data are sent back to the compute node using the tree network as well.

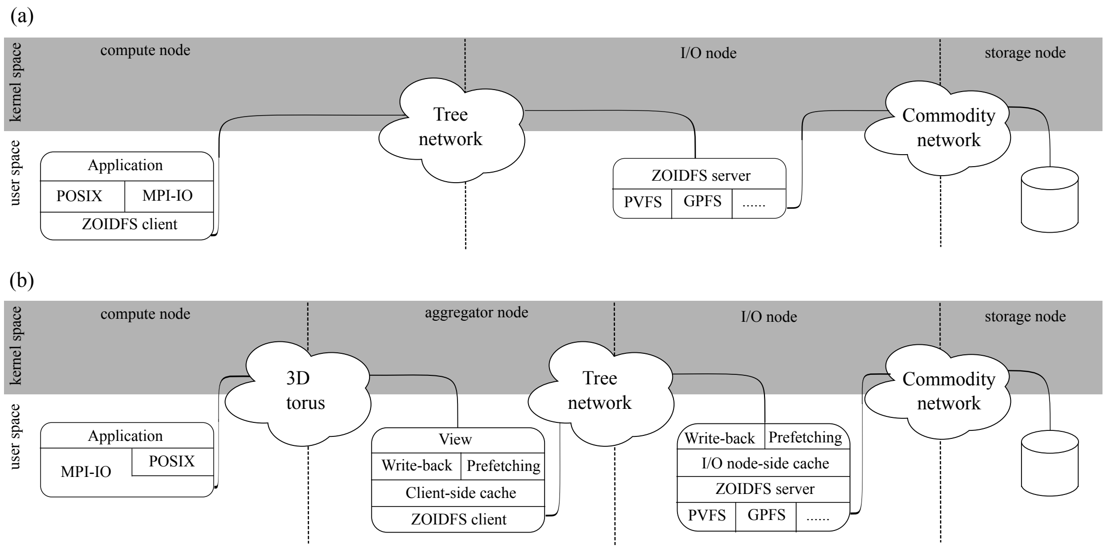


Fig. 3. File I/O forwarding in ZeptoOS. **(a)** ZOIDFS based solution without caching. MPI-IO and POSIX calls are mapped to the abstract file system interface, ZOIDFS, and forwarded to the I/O nodes. The ZOID daemon acts as a ZOIDFS server and maps ZOIDFS calls onto specific file systems. **(b)** ZOIDFS with client-side and I/O-side caching. Applications access the file system through MPI-IO or POSIX interface. POSIX may be implemented on top of MPI-IO. The MPI-IO calls are implemented based on MPI communication and are performed with cooperation by aggregator nodes. A client-side cache, write-back, and prefetching modules manage data on each aggregator. The aggregator nodes forward ZOIDFS calls through ZOID to the I/O node. I/O node services the ZOIDFS calls either from cache or by contacting the appropriate file system.

access to recently accessed data [28], and asynchronous data transfer from cache to the I/O subsystem over the tree network, hiding tree network access latency.

The client-side file cache is organized as a distributed file cache, stored on the local memories of compute nodes. A client-side cache management module runs on each node managing a local cache. Data is forwarded from/to other compute nodes through the upper client-side I/O forwarding tier, and from/to associated I/O nodes through the I/O forwarding layer.

I/O-side file cache management tier. The I/O-side file cache management tier provides file caching close to the storage system (i.e. file system) and offers efficient transfer methods between I/O nodes and the storage system. As in the case of the client-side file cache management tier, data staging is managed in two modules operating on a file cache: write-back and prefetching modules. Each of these modules acts in coordination with the corresponding module on the compute node side.

Storage system. The storage system consists of file system servers running on storage nodes and accessing disks over a storage area network. File systems are mounted on I/O nodes and accessed via kernel interfaces.

V. Data staging

In Blue Gene systems, file system access implies pipelining data through three different networks (tree, commodity and storage network) from compute nodes through I/O nodes and storage servers to finally reach the storage. A bottleneck in any of these networks may be propagated up to applications. The data staging is designed to hide the latency of the transfers through these networks and, therefore, reduce the probability of applications perceiving I/O congestion. Our solution addresses two potential hot spots in the Blue Gene architecture: the tree network and the file system. The tree network is especially problematic, given its shared use by all the processors in a pset. The file systems are also shared by the whole system, and they may provide unexpectedly slow service to a particular partition when data intensive applications are run in other partitions.

In this section we discuss how our system manages data staging between client-side and ION-side caches and how write-back and prefetching are integrated into this multi-tier system.

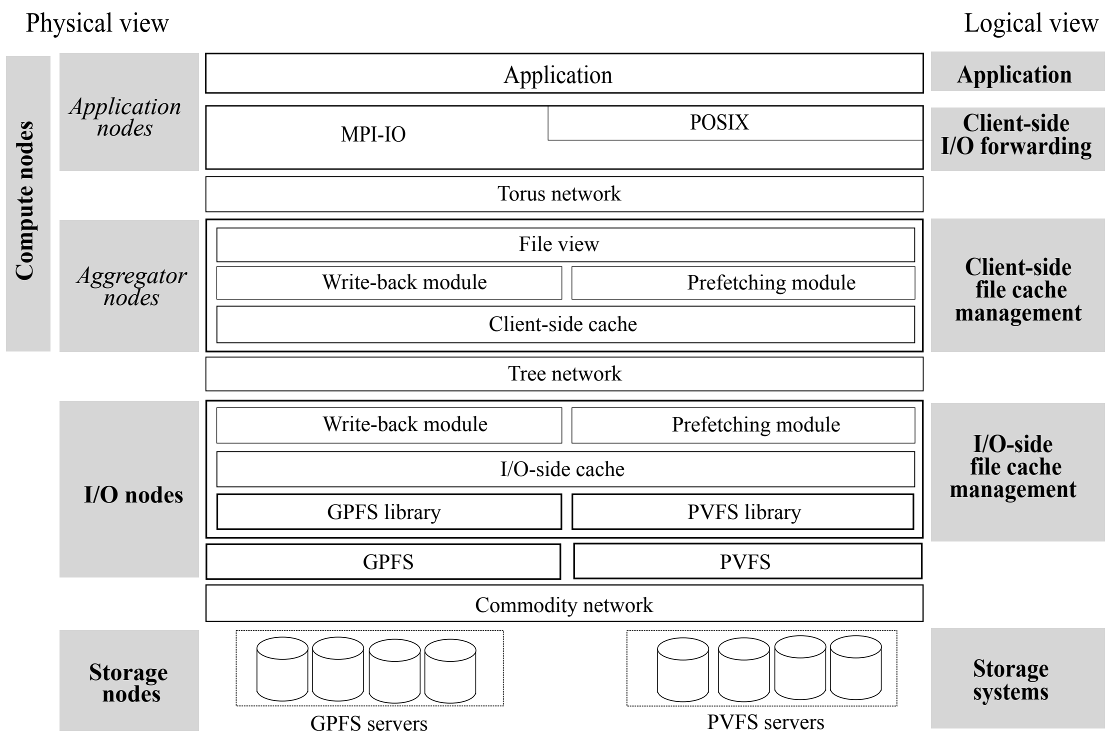


Fig. 4. Blue Gene cache architecture organized on five tiers: application, client-side I/O forwarding, client-side cache, I/O node-side cache and storage. Applications issue file access calls through POSIX or MPI-IO interfaces. The client-side I/O forwarding layer transfers data between applications and the client-side cache module through the torus network. The client-side file cache management tier orchestrates caching on the compute nodes, offers access to the applications (optionally through views), and transfers data between compute nodes and I/O nodes over the tree network. The I/O node -side file cache management tier handles caching on I/O node, serves requests from the client-side file cache management tier, and accesses file systems over the commodity network. The storage system includes GPFS and PVFS file systems.

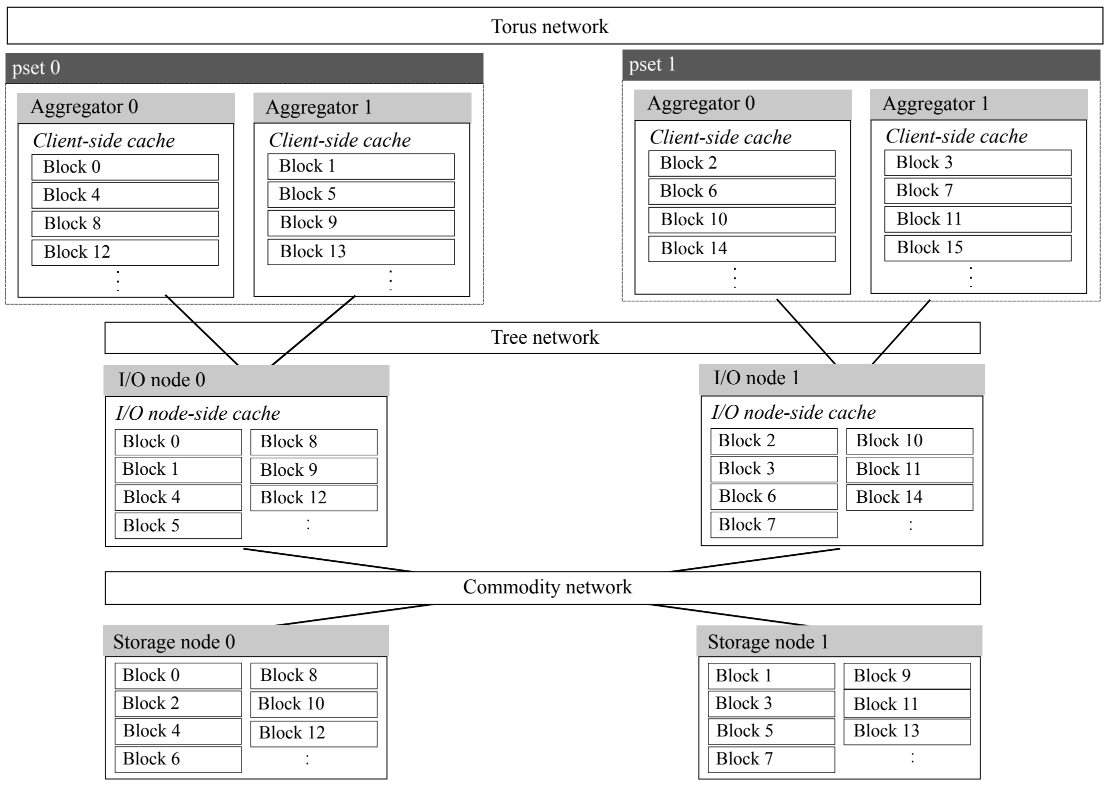


Fig. 5. An example offile mapping in our system. A file is mapped over four aggregators in two psets, with two aggregators per pset. The file blocks held by aggregators are mapped onto the I/O nodes in charge of the corresponding pset. Each I/O node caches and manages access to the file system blocks mapped to aggregators in their pset. Note that there may be exactly one copy of a file block at each level. For instance, the file block 2 may be cached only in pset 1 at aggregator 2, I/O node 1, and storage node 0.

A. Client-side file cache

The client-side cache absorbs writes from applications and hides the latency of data movement to and from I/O nodes over the tree network. An *aggregator* on compute nodes combines small accesses into larger file system blocks. Aggregators were initially used in ROMIO [32] for collective I/O implementations such as two-phase I/O [32] and view-based I/O [5]. In our solution the I/O aggregators participate not only in collective I/O operations but also in independent I/O. The I/O operations of each aggregator are performed in a dedicated I/O thread that manages both on-demand and asynchronous file-related communication with the I/O node. All file accesses from application processes are sent through client-side forwarding layer to the aggregators, which serve them either from the local cache, from the cache of another aggregator, or via the I/O thread. In the current implementation processes perform synchronous writes to the aggregator, while the aggregator asynchronously performs resulting file writes to the associated I/O node in the dedicated I/O thread. An additional optimization would allow application processes to forward file accesses asynchronously to the aggregators, making a further increase in the overlap between computation and I/O possible. This optimization is a subject of future work.

Files blocks are mapped in a round-robin manner over all the aggregators in the application's partition. Each file block is mapped to exactly one aggregator. This aggregator interacts directly with the I/O node connected to the pset. Figure 5 shows an example of a file mapped on a partition consisting of two psets, with two aggregators per pset. File block 2 can be cached only once at aggregator 2 and at the I/O node 1 and on the storage node 0. This also represents the transfer pipeline for file access.

The number of aggregators is a configurable parameter; by default, all the compute nodes act as aggregators. The replacement policy of each aggregator cache is LRU. The application accesses the client-side cache of other processes through the torus network.

Non-contiguous accesses can be optimized through view-based I/O [5]. A view is an abstraction that allows application to see non-contiguous regions of a file as contiguous. View-based I/O leverages this abstraction for implementing an efficient non-contiguous strategy. When defined, the view-to-file mappings are sent to aggregators, where they are stored in memory for subsequent use. At access time, contiguous view data can be transferred between compute nodes and aggregators, avoiding many small network transfers. Using the view mapping, the aggregator can locally perform scatter/gather operations between view data and file blocks. Additional advantages of views are that they compactly represent access patterns and that they may be used as hints to future access patterns. This last feature is leveraged by the client-side prefetching module.

B. I/O-side file cache

The I/O-side cache absorbs blocks transfered from the client-side cache and hides transfers between I/O nodes and file systems. The replacement policy of the I/O node-side cache is LRU. The I/O-side file cache management layer is integrated into the ZOID daemon running on each I/O node. The daemon receives ZOIDFS requests from the compute nodes and serves them from the cache. The communication with the compute nodes is decoupled from the file system access, allowing for a full overlap of the two operations. An I/O thread is responsible for asynchronously accessing the file systems and incorporates write-back and prefetching functionality.

C. Two-level write-back

After a compute node issues a file write, data is pipelined from compute nodes through the I/O nodes to the appropriate file system. An application write request is transfered by client-side I/O forwarding tier to the client node responsible for that file block. The cached file blocks are marked dirty, the application is notified of a successful transfer, and computation resumes. A write-back module on clients is responsible for flushing the data from client-side cache to the I/O node attached to that pset. On the I/O node another write-back module is in charge of caching the file blocks received from the compute nodes and flushing them to the file system over the commodity network.

The write-back policy used in this work (for both caching levels) is based on a high/low water mark for dirty blocks. The high and and low water marks are percentages of dirty blocks. The flushing of dirty blocks is activated when a high water mark of blocks is reached. Once activated, the flushing continues until the number of dirty blocks falls below a low water mark. Blocks are chosen to be flushed in Least Recently Modified (LRM) order.

In order to efficiently hide latency, coordination along the pipeline is critical. We highlight two important aspects. First, the coordination has to take into account the application requirements. For instance, in parallel applications, processes frequently write shared files in non-overlapping manner, showing a good inter-process spatial locality. For these applications the high and low water marks should be sized in such a manner that makes it improbable that incompletely written blocks are transferred. On the other hand, blocks of files written by sequential applications may be immediately flushed. Second, the coordination must take into consideration the network characteristics and loads. We have implemented mechanisms to perform this coordination and will evaluate some potential policies in this work.

D. Two-level prefetching

Our prefetching solution is split across compute nodes and I/O nodes, with each instance enforcing its own prefetching policy and driving prefetching with an I/O thread. In this paper we present two simple policies implemented on the compute node and I/O node, respectively.

The client-side prefetching policy is based on two main parameters: mapping of files to aggregators and views. If no view is declared, the view is by default the whole file. The application process sends the view to all file aggregators after declaration, as described in Section V-A. Any time an on-demand read request misses the client-side cache, it is issued immediately. While it is being served, the subsequent file view offsets are used to calculate a new prefetching request, which is issued to the appropriate I/O node. A configurable number of prefetching requests can be generated. The views bring the advantage of generating any type of prefetching pattern, including the common sequential, simple and multiple-strided.

The I/O node prefetching is based on the mapping of aggregators to I/O nodes. Whenever an aggregator makes an on-demand request to an I/O node, it is first served and, subsequently, the next file blocks mapped to the same aggregator are computed and prefetch requests are issued. Prefetching requests of aggregators become on-demand file requests at the I/O node, driving prefetching at that layer as well.

E. File access semantics and consistency

Our solution provides a relaxed file system semantics, motivated by the well-known fact that POSIX-like semantics are not suitable for HPC workloads [14]. While a file is open, its data may reside any level of the cache hierarchy. Data are ensured to have reached the final storage after file close or after a file sync has been executed. In particular, MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. The atomic mode for independent I/O file accesses has not been implemented, i.e. sequential consistency is not guaranteed for concurrent, independent I/O with overlapped access regions. This approach is similar to the one taken in PVFS, and it is motivated by the fact that overlapping accesses are not frequent for parallel applications.

VI. Experimental Results

The experiments presented in this paper have been performed on the Surveyor Blue Gene/P system from Argonne National Laboratory. The system has 1024 quad-core 850 MHz PowerPC 450 processors with 2 GB of RAM each. All the experiments were run in Symmetric Multiprocessor mode (SMP), in which a compute node executes one process per node with up to four threads per process. The PVFS2 [21] file system is mounted on all I/O nodes and stripes files round-robin over four storage servers. We used a stride size of 1MBytes, which is equal to the page size of both levels of caching in our system.

A. Benchmarks

In the evaluations we use two benchmarks: SimParIO synthetic benchmark and NASA's BTIO benchmark.

SimParIO is a synthetic benchmark simulating the behavior of data-intensive applications, which have been shown to alternate the computation and I/O [18], [24], [29], [31]. The benchmark consists of a configurable number of alternating computation and I/O phases. The compute phases are simulated by idle spinning. In the I/O phase all the processes write non-overlapping records to a file. The configurable parameters of this benchmark are the following: the compute time, the number of phases, the record size, the number of records, and the access stride.

NASA's BTIO benchmark [34] solves the block-tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes. The execution alternates computation and I/O phases. Initially, all compute nodes collectively open a file and declare views on the relevant file regions. After every five computing steps, the compute nodes write the solution to a file through a collective operation. At the end, the resulting file is collectively read, and the solution is verified for correctness.

B. File write performance

This experiment evaluates the dependence of the file write-back performance on the high-water marks of the file caches on the compute node and I/O node. The size of client-side cache on a compute node is fixed at 64 MBytes. All the compute nodes cache data (i.e., act as aggregators). The size of the I/O node cache is 512 MBytes. The block size is 1 MByte. SimParIO was run with 64 and 256 processes (one process per compute node) and was configured to write a total of 2,560 MBytes, i.e. each process writes a record of 1 MByte in 40 phases for 64 processes and 10 phases for 256 processes.

The high water mark for client-side and I/O node-side caches was varied from 0% to 100%. A high water mark value of 0% signifies that flushing is always activated, while 100% value that flushing is activated when the whole cache is full. Figures 6 and 7 show the aggregate file write throughput for 64 and 256 processes with compute phases of 0 ms (left) and 500 ms (right). When computing the throughput, the time to close the file is included.

Note that in most cases the write throughput increases with the decrease of the high water mark on the I/O nodes. The best performance is obtained for 0%, i.e. when flushing is always activated. This indicates that a continuous write of dirty blocks from the I/O node to the file system is the best strategy. However, a 0% high water mark on the compute nodes does not bring performance benefits. The peaks are obtained for 6.25% and 12.5% for 64 nodes and 3.12% and 6.25% for 256 nodes. The values of the client-side high water mark for 256 node peak performance are smaller than those for 64 nodes, as the size of both client-side caches and I/O caches are scaled up by a factor of four, thereby reducing the data pressure in the write pipeline.

For compute phases of 500ms there is more potential for overlap between computation and I/O. However, when comparing with a 0ms compute phase, there is a significant performance increase only for small high water marks of the client-side caches. This is explained by the fact that small values of high water marks increase the probability of continuous flushing and, therefore, of overlapping I/O with computation. The efficiency of a flushing strategy can be estimated by the time to completely flush the file data at file close: the smaller

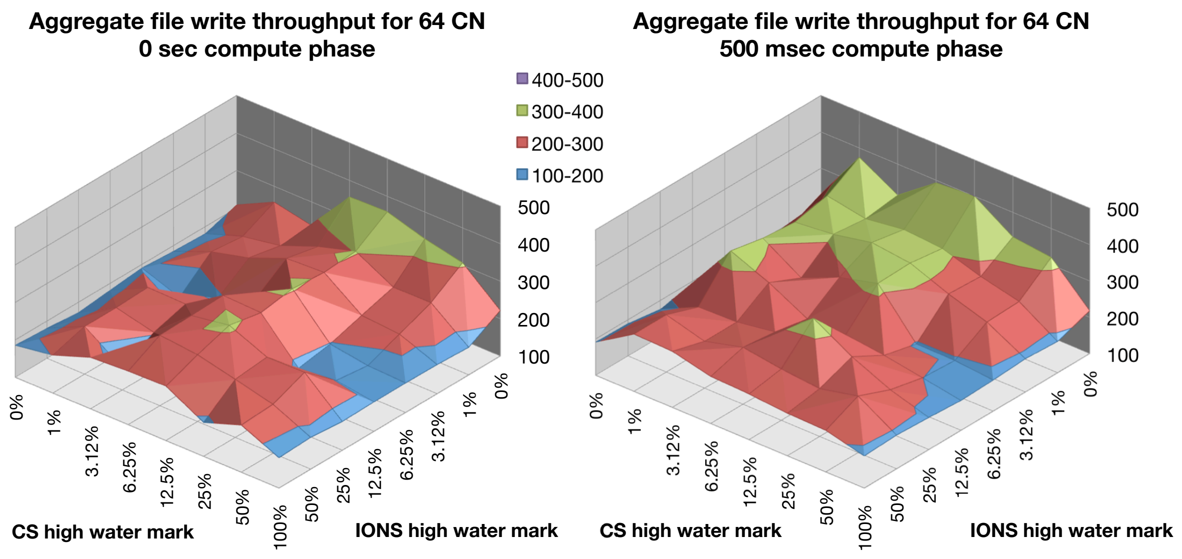


Fig. 6. File write performance for 64 processors, client-side cache size of 64 MBytes, I/O node-side cache size of 512 MBytes, variable high water mark, and 0 and 500ms compute phases. The scales of x-axis and y-axis are logarithmic. The time used to calculate the aggregate throughput includes the time to flush the caches on file close. The throughput increases with the decrease of the high water mark on the I/O nodes and is highest for client-side high water mark of 6.25% and 12%. In the presence of computation, the throughput increases for small, client-side high water marks.

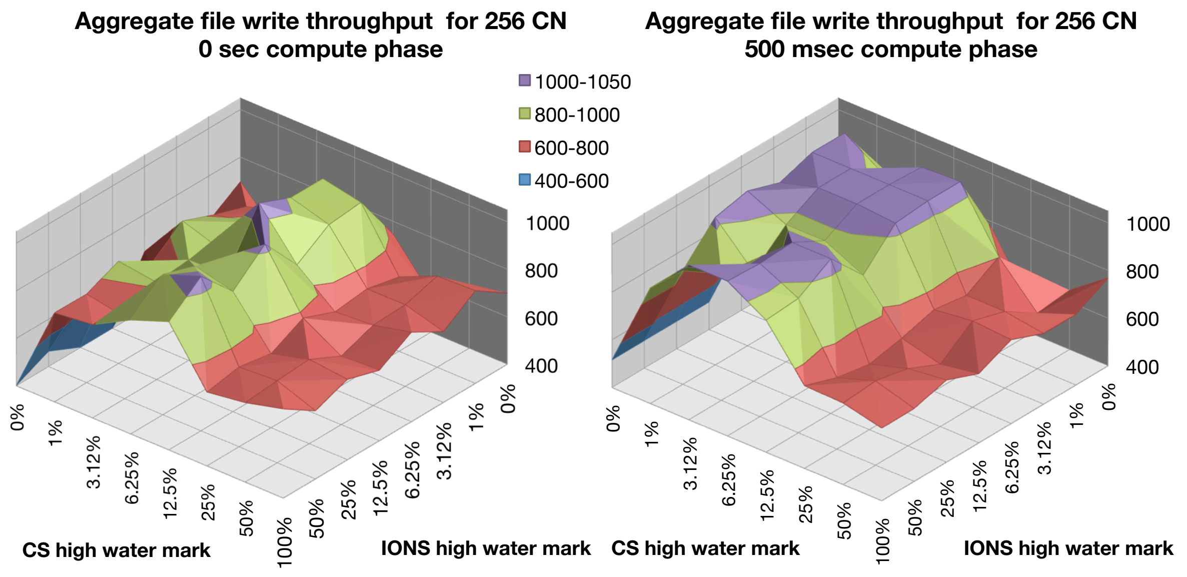


Fig. 7. File write performance for 256 processors, client-side cache size of 64 MBytes, I/O node-side cache size of 512 MBytes, variable high water mark, and 0 and 500ms compute phases. The time used to calculate the aggregate throughput includes the time to flush the caches on file close. The throughput increases with the decrease of the high water mark on the I/O nodes and is highest for client-side high water mark of 3.12% and 6.25%.

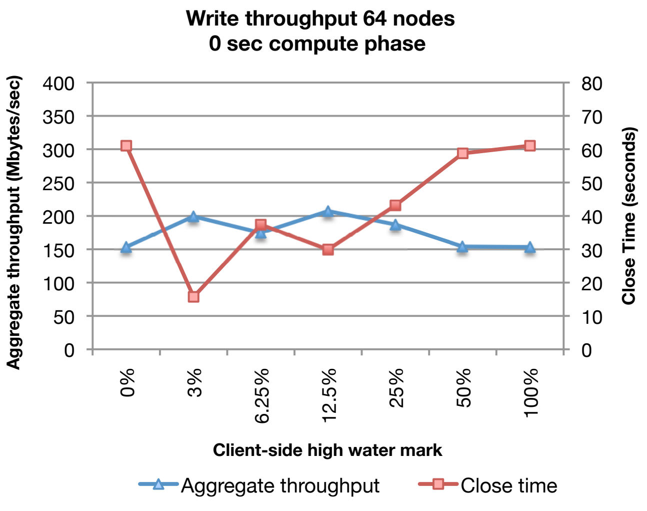
the close time, the more efficient the strategy. Figure 8 plots in parallel the aggregate write throughput and file close time for 64 nodes, 0s and 500ms compute phase, and 0% high water mark for I/O node-side cache. The figure confirms that the aggregate throughput is inversely proportional to the close time.

C. File cache sizes

This experiment evaluates the dependence of file write-back performance on the sizes of the file caches on the compute node and I/O node. The experiment was run on 64 compute nodes inside a pset. The high water mark for client-side cache was 6.25%, and for the I/O node-side cache 0%.

The size of client-side cache on a compute node was varied from 0 MBytes (no caching) to 64 MBytes. The size of I/O node-side cache was varied from 0 MBytes (no caching) to 512 MBytes. All the compute nodes cache data (act as aggregators). The file block size is 1 MByte. SimParIO was configured to write a total of 2,560 MBytes, i.e. each process repeatedly writes a record of 1 MByte in 40 phases.

Figure 9 shows the aggregate file write for compute phases of 0 ms (left) and 500 ms (right). When computing the throughput, the time to close the file is included. The graphs show that the client-side caches bring a substantial performance improvement. This improvement is almost independent of the size of the I/O caches. The best results are obtained



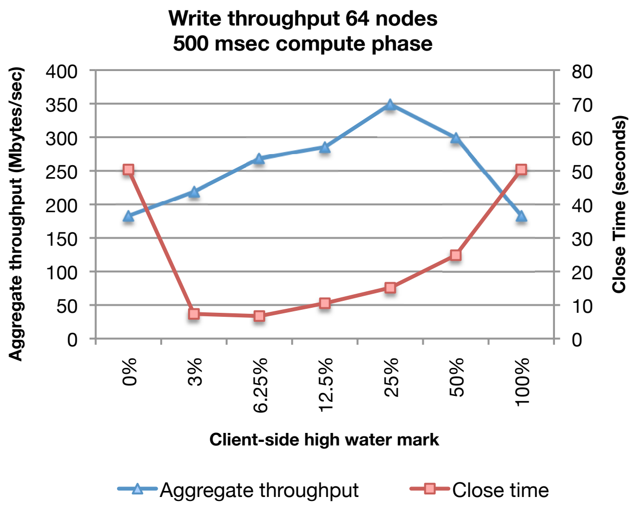
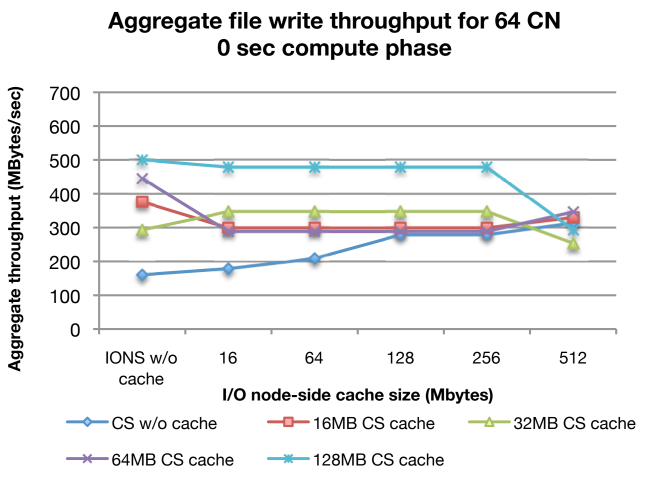


Fig. 8. The figure plots the aggregate write throughput and file close time for 64 nodes, 0ms and 500ms compute phase and 0% high water mark for I/O node-side cache. The close time can be seen as an efficiency metric of a flushing strategy: the graph shows that aggregate throughput is inversely proportional to the close time.



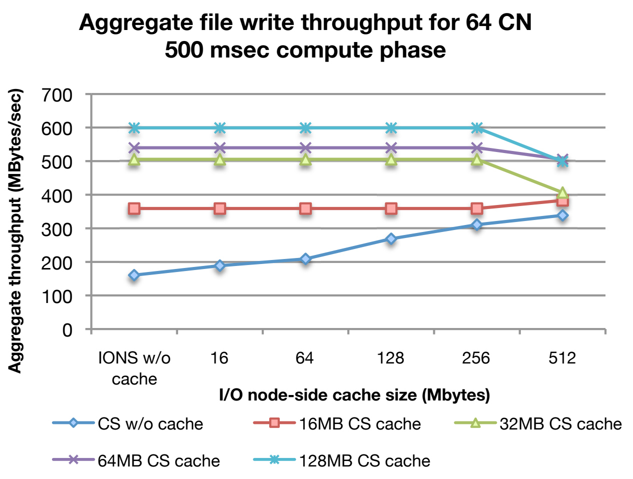


Fig. 9. Effect of file cache sizes on the aggregate file write throughput for a pset of 64 processors,6.25% high water mark for client-side cache, 0% high water mark for I/O node-side cache, 0ms and 500ms compute phase. The ratio between the best client-side cache size and I/O node-side caches size is 64, corresponding to the number of compute nodes in the pset and to the number of aggregators.

for client-side caches of 8 MBytes and I/O node-side caches of 512 MBytes. The ratio between the best client-side cache size and I/O node-side caches size is 64, corresponding to the number of compute nodes in the pset and to the number of aggregators. This result suggests the optimal size of the I/O node cache to be equal to the sum of the client-side caches in the corresponding pset. Further increases of this cache appear even to worsen the performance. This could be explained by the fact that a larger cache may take a longer time to be flushed when the file is closed.

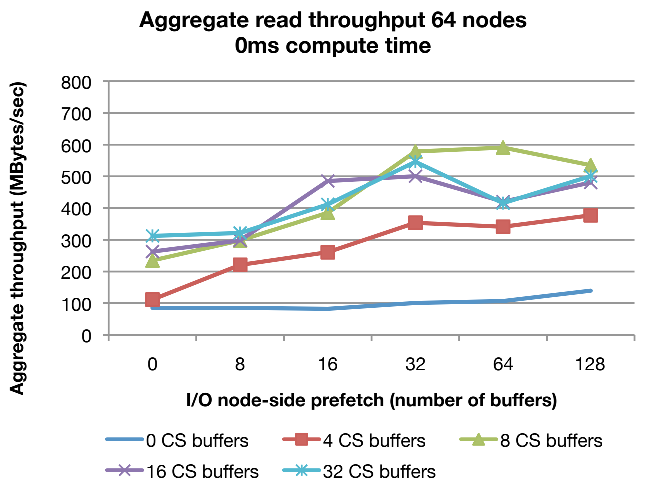
Expectedly, the size of the cache closer to the application (client-side cache) appears to influence the performance in a stronger way than a remoter cache (I/O-side cache). The comparison of the two graphs for 0ms and 500ms shows that a better potential to overlap computation brings only a marginal performance benefit for client-side caches larger or equal to 16 MBytes.

D. *Prefetching*

This experiment evaluates the performance of prefetching into the client-side and I/O node-side caches. The experiment was run on 64 compute nodes inside a pset. The size of the client-side cache on a compute node was fixed at 64 MBytes and the size of the I/O node-side cache at 512 MBytes. All the compute nodes cache data (act as aggregators). The file block size is 1 MByte. SimParIO was configured to read a total of 2,560 MBytes, i.e. each process repeatedly reads a record of 1 MByte in 40 phases. No views were used; a further evaluation of the prefetching based on views is presented in the next section using the BTIO benchmark.

We evaluate prefetching configurations, varying the number of prefetched file blocks. For client-side caches the number of prefetched blocks was 0 (no prefetching), 4, 8, and 16. On the I/O nodes this number was varied from 0 (no caching) to 256. Figure 10 shows the aggregate file read for compute phases of 0ms (left) and 500ms (right).

As expected, the client-side prefetching has a stronger influ-



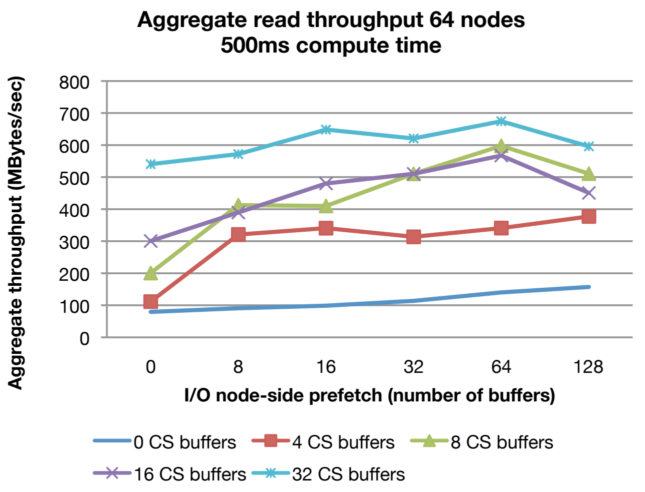
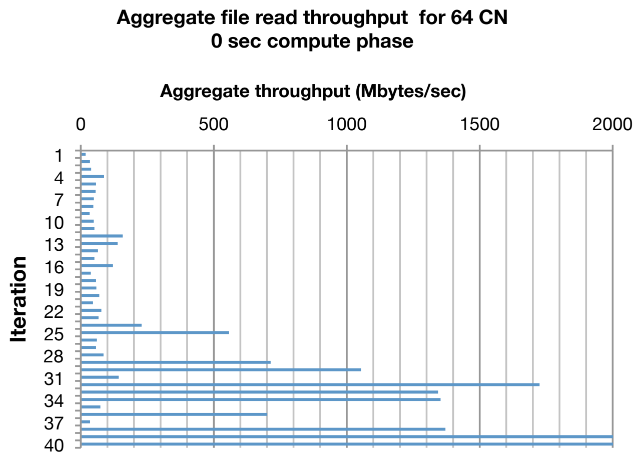


Fig. 10. Effect of prefetching window on the aggregate file read throughput for a pset of 64 processors, for 0 , 4, 8, and 16 prefetched client-side caches blocks and 0, 8, 16, 32, 64, 128, 256 prefetched I/O node-side caches blocks. Client-side prefetching has a stronger influence on the read performance than I/O-side prefetching. It brings up to one order of magnitude improvement in presence of computation.



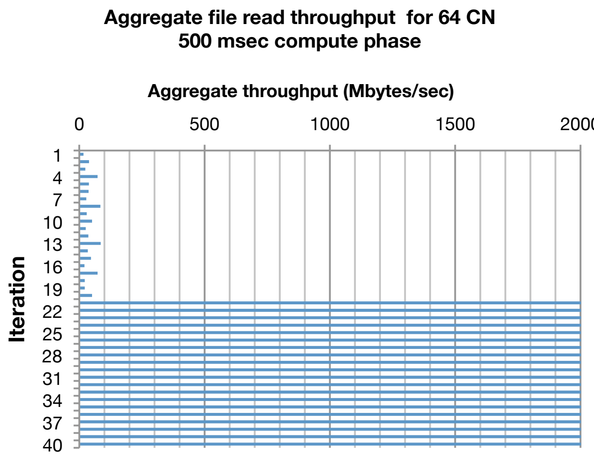


Fig. 11. Histogram of the 40 phases of file read for the 16 blocks read-ahead, for 0ms and 500ms compute phase. Prefetching starts to pay off in phase 24 for no compute phase and in phase 20 for a 500ms compute phase.

ence on the read performance than I/O-side prefetching. When no client-side prefetching is used, the I/O node prefetching does not appear to bring any performance benefit. Client-side prefetching brings more than one order of magnitude improvement, especially when the compute phase to be overlapped is increased to 500ms.

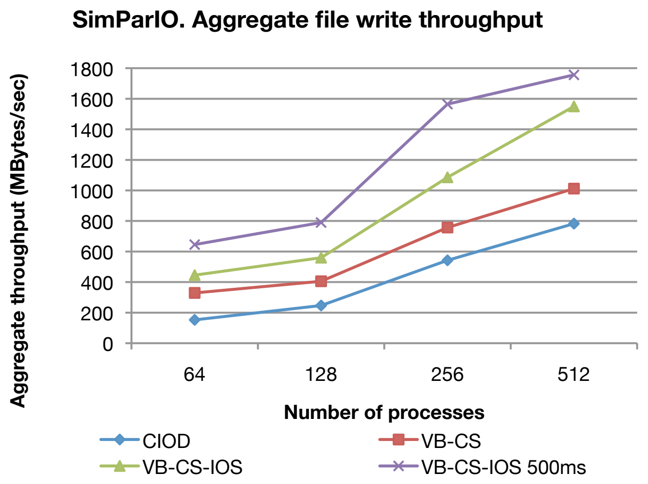
In order to better understand the impact on prefetching we plot in Figure 11 the aggregate read throughput of the 40 individual read phases for two cases from Figure 10, with 16 prefetched blocks on the I/O node and both 0 ms and 500 ms compute phase. Prefetching begins to pay off in phase 24 for no compute phase and in phase 20 for a 500 ms compute phase. The prefetching is substantially more efficient when it is overlapped with computation: all the phases after phase 20 appear to be serviced from the client-side cache.

E. Scalability

This evaluation aims to test the solution scalability in terms of file size and number of compute nodes. Figure 12 shows the results of running SimParIO benchmark on 64 to 512 compute nodes. The size of client-side cache is fixed at 128 MBytes. All the compute nodes cache data (act as aggregators). The size of each I/O node cache is 512 MBytes. The high water mark for client-side cache was 12.5%, and for the I/O node-side cache 12.5%. Compute nodes perform 40 iterations. In each iteration, the file access pattern is strided with a record size of 64 KBytes and a stride size of 4 MBytes. The maximum file size produced by 512 processes was 0.5 TBytes. We evaluate the file writes of the SimParIO benchmark for three different setups: two-phase I/O over IBM solution (CIOD), view-based I/O with client-side caching (VBIO-CS), and view-based I/O with both client-side and I/O node-side caching (VBIO-CS-IONS). The graphs show that file access performance scales well with the number of compute nodes for both read and write operations. The performance obtained is higher when both cache levels are employed and in presence of computation.

F. BTIO benchmark

In this section we evaluate our data staging approach for the BTIO benchmark. The client-side cache on each compute node



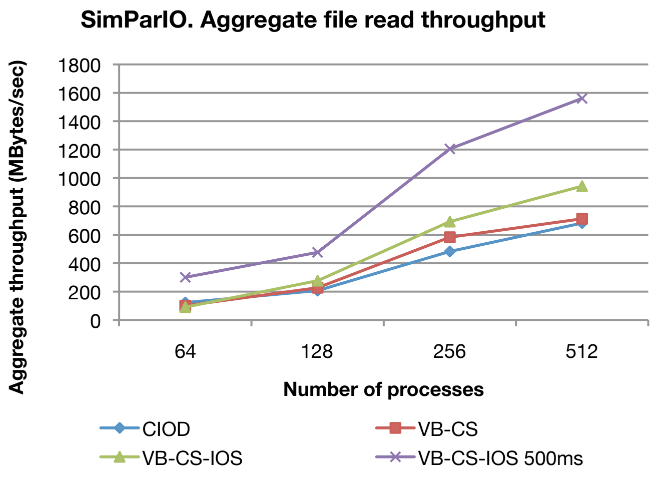


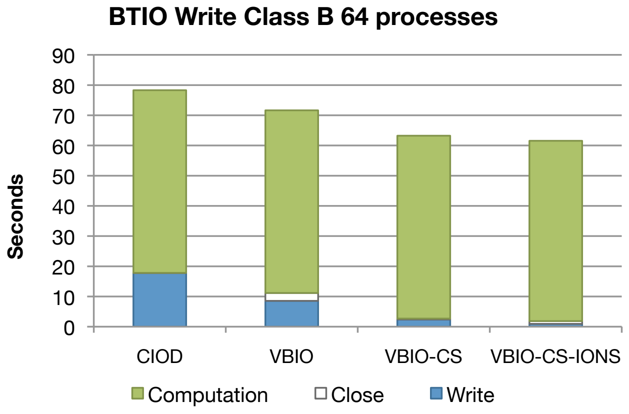
Fig. 12. Scalability in terms of file size and number of compute nodes. Aggregate file write and read throughputs for 64 to 512 processes, and 0ms to 500ms compute phases.

is fixed at 64 MBytes, while the I/O node-side cache is fixed at 512 MBytes. All application nodes acted as aggregators. We report the results for BTIO class B producing a file of 1.6 GBytes.

1) File writes: We evaluate the file writes of the BTIO benchmark for four different setups: two-phase I/O over IBM solution (CIOD), view-based I/O with no caching (VBIO), view-based I/O with client-side caching (VBIO-CS), and view-based I/O with both client-side and I/O node-side caching (VBIO-CS-IONS). Figure 13 shows the total time breakdown into compute time, file write time, and close time for BTIO class B. The close time is relevant because all data are flushed to the file system when the file is closed. We notice that in all solutions the compute time is roughly the same. VBIO reduces the file write time without any asynchronous transfers. VBIO-CS reduces both the write time and close time, as data are asynchronously written from compute node to I/O node. For VBIO-CS-IONS, the network and I/O activity are almost entirely overlapped with computation. We conclude that the performance of the file writes gradually improves with the increasing degree of asynchrony in the system.

2) File reads: BTIO performs all forty read phases in sequence, without any interleaving compute phases. In order to evaluate the effect on prefetching in presence of computation, a computation phase was inserted between consecutive read phases.

Figure 14 displays the file read performance without prefetching (for two-phase I/O and view based I/O) and with prefetching for 0ms, 500ms, and 1000ms compute phases. We note that prefetching pays off when the client-side prefetching pool has at least 8 blocks and computation is present. The worst time was obtained for 2 prefetched file blocks and no computation and the best for 16 prefetched blocks and 1 second compute phase. Figure 15 shows the measured times of the 40 file read operations for 64 processors for the these two cases. We note that, in the worst case depicted on the left, the phase time decreases starting with phase 19, and in the best case shown on the right with phase 11. This indicates the timing when the read accesses start to hit the cache. In



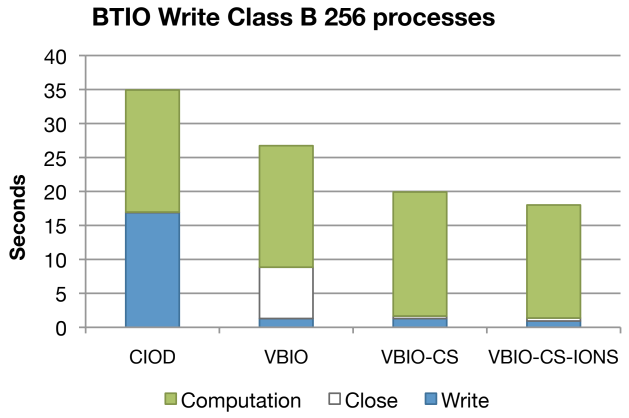


Fig. 13. BTIO class B file write times for 64 and 256 processors. The performance of the file writes gradually improves with the increasing degree of asynchrony in the system: the best overlap is achieved when both client-side cache and I/O node-side cache are used.

the best case, the presence of computation causes a more uniform distribution and a reduction of access times in the initial phases.

VII. Discussion

Our results demonstrate that a significant performance improvement can be obtained from multiple level data staging. The employment of client-side and I/O-node caches helps

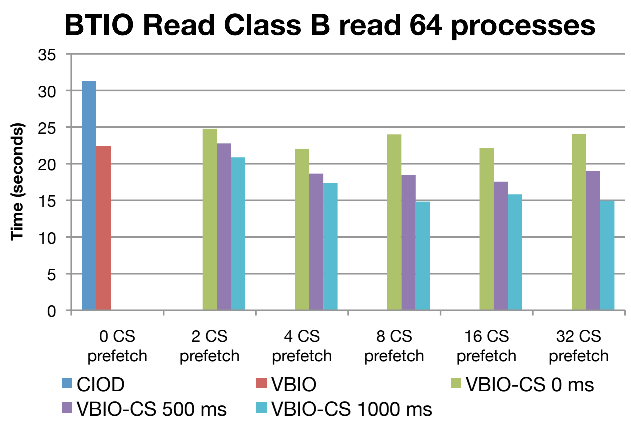


Fig. 14. BTIO class B file read times for 64 processors for client-side prefetching pool sizes of 0, 2, 4, 8, 16 file blocks. Prefetching is improved only in the presence of computation and for client-side prefetching pools of at least 8 blocks.

overlap the latency for both file writes and reads and may contribute to up to a five-hold increase for writes and an order of magnitude for reads, depending on various parameters. As expected, the client-side cache contribution to the performance improvement is predominant. Applications access the client-side cache over the torus network, contributing a decrease in the number of small transfers over the tree network and to better distribution of transfers over time.

The write-back performance shows a strong dependence on the flushing high water mark of both client-side and I/O node-side cache. The performance difference between best and worst figures for these two parameters can be as high as two-fold. In the considered cases, the best policy appears to be a combination of continuous flushing on the I/O nodes (0% high water mark) and more bursty flushing on the compute nodes (high water mark greater than 0). The size of the client-side cache may also cause a performance difference as high as two-fold. However, the size of the I/O node-side cache seems to have a weak effect on performance.

Prefetching brings performance benefits of up to one order of magnitude depending on the prefetching pool sizes on both compute nodes and I/O nodes. Prefetching into the client-side cache is critical for performance in all cases. Prefetching into the I/O node is important when application read operations are not interleaved with computation. In this case the I/O node prefetching works in parallel with on-demand prefetching on the compute node, increasing the pipeline parallelism. The obtained results suggest suitable policies for data staging. The results from Figure 6 indicate that the write-back policy should be chosen by the following rule of thumb: the farther from the storage the cache, the lower the high watermark. On the other hand, Figure 10 shows that a simple prefetching policy employed by client-side aggregators efficiently propagates reads to the next levels and provides large performance benefits. The interaction between levels is crucial in order to optimize the performance. In this work we have mainly studied the interaction between different levels of either write-back or prefetching in isolation. A further analysis is necessary in order to better understand the cross-interactions between write-back and prefetching at different levels.

The client-side cache scales with the number of aggregators. The results suggest that, for efficient pipelining, client-side and I/O-side caches have to be sized in such a way that the I/O node cache size should be at least equal to the sum of the client-side caches in the corresponding pset.

The solution presented in this paper was implemented on Blue Gene systems, but it can be easily extended to other systems. The client-side and I/O modules are generic and portable, and the implementation can be used unmodified on clusters or any other supercomputers. This can be achieved either by extending the ZOID back-end to these systems or, alternatively, through an ADIO module for file systems mounted on the I/O nodes.

To summarize, a coordination policy for multi-layer caching must take into account various aspects related to the file I/O pipeline including application requirements, size of the caches, flushing water marks, and prefetch window sizes. The evaluation in this paper, based on a subset of this large parameter space, demonstrated the huge potential for performance improvement using this approach.

VIII. Conclusions and Future Work

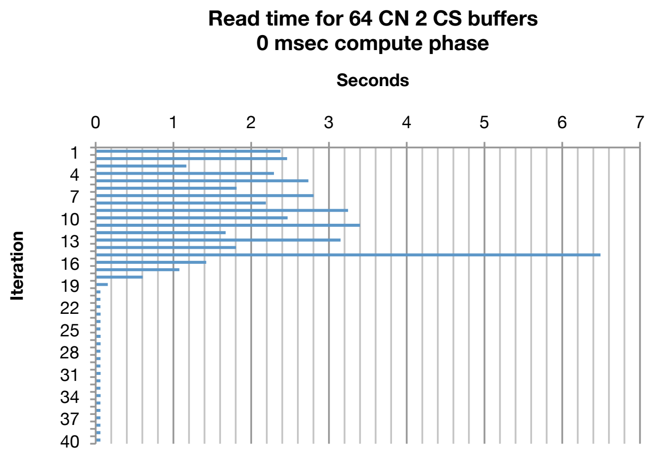
This paper presents the design, implementation and evaluation of multiple level data staging for Blue Gene systems. The data staging is based on a two-level hierarchical caching system, consisting of an application-close client-side cache and a storage-close I/O node-side cache. The experimental results prove that both write-back and prefetching strategies provide a significant performance benefit, whose main source comes from the efficient utilization of the Blue Gene parallelism and asynchronous transfers across storage system hierarchy.

The paper shows that the performance may significantly vary with configuration parameters such as cache sizes, high and low water marks, and prefetch pool size. Future work will target automating the parameter selection for performance. Further, larger-scale evaluations are needed, and we are currently beginning this work on the larger Intrepid system at Argonne National Laboratory. Additionally, we plan to perform more extensive evaluations of data staging in a congested system, focusing on torus and tree congestions and file system load. The goal is to design novel, adaptive data staging policies that address changes in network congestion, I/O node load, and file system response time.

Acknowledgments

This work was supported in part by Spanish Ministry of Science and Innovation under the project TIN 2007/6309, by the U.S. Dept. of Energy under Contracts DE-FC02-07ER25808, DE-FC02-01ER25485, and DE-AC02-06CH11357, and NSF HECURA CCF-0621443, NSF SDCIOCI-0724599, and NSF ST-HEC CCF-0444405.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.



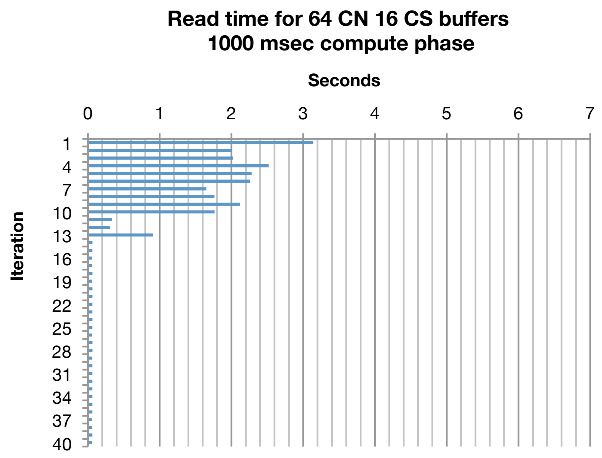


Fig. 15. Histograms of the 40 file read operations of BTIO class B times for 64 processors for the best and worse performing cases from Figure 14. The reads start to hit the cache in phase 19 and 11, respectively.

References

[1] *International Exascale Software Project*. http://www.exascale.org.

[2] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. Datastager: scalable data staging services for petascale applications. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 39–48, New York, NY, USA, 2009. ACM.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In *Proceedings of IEEE Conference on Cluster Computing, New Orleans, LA*, September 2009.

[4] J. G. Blas, F. Isaila, J. Carretero, R. Latham, and R. B. Ross. Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems. In *PVM/MPI*, pages 164–173, 2009.

[5] J. G. Blas, F. Isaila, D. E. Singh, and J. Carretero. View-Based Collective I/O for MPI-IO. In *CCGRID*, pages 409–416.

[6] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC '08*, pages 1–12, 2008.

[7] F. Chang and G. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of OSDI*, 1999.

[8] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. In *SC '08*, pages 1–10, 2008.

[9] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.

[10] F. Isaila, J. Garcia Blas, J. Carretero, R. Latham, S. Lang, R. Ross. Latency hiding file I/O for Blue Gene systems. In *CCGRID '09*.

[11] http://fuse.sourceforge.net. *FUSE Homepage.*, 2009.

[12] http://www unix.mcs.anl.gov/zeptoos/. *ZeptoOs Project.*, 2008.

[13] http://www.top500.org. *Top 500 list*.

[14] http://www.unix systems.org/. *The Portable Operating System Interface*, 1995.

[15] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating Collective I/O and Cooperative Caching into the “Clusterfile” Parallel File System. In *Proceedings of ACM International Conference on Supercomputing (ICS)*, pages 315–324. ACM Press, 2004.

[16] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *PPoPP '08*, pages 153–162, 2008.

[17] M. Kallahalla and P. Varman. PC-OPT: optimal offline prefetching and caching for parallel I/O systems. *Computers, IEEE Transactions on*, 51(11):1333–1344, Nov 2002.

[18] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.

[19] W. K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel, and S. Tideman. Collective Caching: Application-Aware Client-Side File Caching. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[20] W. K. Liao, K. Coloma, A. N. Choudhary, and L. Ward. Cooperative Write-Behind Data Buffering for MPI I/O. In *PVM/MPI*, pages 102–109, 2005.

[21] W. Ligon and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[22] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. In *IPDPS*, pages 22–26, 2003.

[23] J. Moreira and et al. Designing a highly-scalable operating system: the Blue Gene/L story. In *SC '06*, page 118, 2006.

[24] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File Access Characteristics of Parallel Scientific Workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, pages 1075–1089, Oct. 1996.

[25] C. M. Patrick, S. Son, and M. Kandemir. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO. volume 42, pages 43–49, New York, NY, USA, 2008. ACM.

[26] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29(5):79–95, 1995.

[27] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM Press.

[28] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[29] P. C. Roth. Characterizing the I/O behavior of scientific applications on the Cray XT. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 50–55, New York, NY, USA, 2007. ACM.

[30] Y. H. Sahoo, R. Howson, and et all. High performance file I/O for the Blue Gene/L supercomputer. *HPCA*, pages 187–196, 2006.

[31] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[32] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[33] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel Scripting for Applications at the Petascale and Beyond. *Computer*, 42(11):50–60, 2009.

[34] P. Wong and R. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical report, NASA Ames Research Center, 2003.

[35] W. Yu and J. Vetter. ParColl: Partitioned Collective I/O on the Cray XT. *ICPP*, pages 562–569, 2008.

[36] W. Yu, J. S. Vetter, and R. S. Canon. OPAL: An Open-Source MPI-IO Library over Cray XT. In *SNAPI '07*, pages 41–46, 2007.