

Kernel-Level Caching for Optimizing I/O by Exploiting Inter-Application Data Sharing^{*}

M. Vilayannur M. Kandemir A. Sivasubramaniam
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802, USA.
{vilayann,kandemir,anand}@cse.psu.edu

Abstract

With applications becoming larger and the increasing load on high performance systems, it is important to tackle the I/O bottleneck problem from several angles. It is not only essential to optimize the I/O accesses of any one application, but also to be able to identify and exploit opportunities resulting from the sharing of datasets across applications. Clusters are rapidly becoming the platform of choice for demanding applications due to their cost-effectiveness and widespread deployment. Consequently, this paper attempts to optimize data sharing across applications concurrently executing on the cluster. Specifically, we propose and implement a kernel-level caching module at each node of a Linux cluster that can be used to service several processes of different applications. Using detailed evaluations on an actual Linux cluster, this paper demonstrates the benefits of this module in optimizing intra and inter-application I/O requests.

1. Introduction and Motivation

Today's parallel architectures comprise fast microprocessors, powerful network interfaces, and storage hierarchies that typically have multi-level caches, local and remote main memories, and secondary and tertiary storage devices. It is known that, in going from upper levels of a storage hierarchy to lower levels, average access times can increase significantly. Since disk technology has not kept pace with performance of the processors employed in parallel architectures, a large performance gap between secondary storage access times and processing unit speeds has emerged. Therefore, optimizing I/O performance is of critical importance.

Recent trends in computer architecture show that networks of workstations –also referred to as clusters– are emerging as a cost-effective solutions for high performance. This has been made possible in part due to the rapid advances in processor and networking technology (System Area Networks such as [3]). In these architectures, the multiple CPUs and their memories can provide

processing and primary storage parallelism, while the multiple disks (one or more at each workstation, or on a network) can provide secondary storage parallelism for both data access and data transfer. While there is a lot of previous work on the design and deployment of clusters (e.g., see [23, 16] and the references therein), there have been very few studies focusing specifically on the I/O-related issues [2, 14, 20]. Even most of these studies have been looking at issues specifically in the development of drivers and file systems.

Many large-scale scientific applications are data-intensive, manipulating large disk-resident data sets ranging from mega-bytes to tera-bytes. These include applications from medical imaging, data analysis and mining, video processing, large archive maintenance, and so on. In addition, many high performance environments not only handle one such application, but often have to deal with several (possibly I/O intensive) applications at the same time in a time-shared manner. One point to note, however, is that some of these simultaneously running applications can share disk resident data. An example of a typical computational science analysis cycle is shown in Figure 1. Such a requirement presents a challenging I/O problem, which can be defined as one of determining access and storage patterns to large disk-resident datasets shared by multiple applications. It is clear that an application programmer may be overwhelmed if required to solve this problem without any help. Most of the current approaches to optimizing I/O at application [18], compiler [5], runtime system [8], and file system [13] levels tend to consider (in general) each application individually.

Based on the above discussion, we believe that optimizing I/O in a cluster environment that run multiple applications simultaneously will be very important in the future. Consequently, in this paper, we try to address this problem by presenting a shared I/O cache implementation and reporting performance data showing its effectiveness. Specifically, we make the following major contributions:

- We present a kernel-level I/O caching strategy implemented on top of PVFS, a parallel file system for Linux clusters.
- We present performance data showing the effectiveness of this caching strategy for multiple, simultaneously-running, I/O intensive applications.

^{*}This research has been supported in part by NSF grants CCR-9988164, CCR-9900701, CCR-0097998, CCR-0130143, DMI-0075572, Career Award MIP-9701475, and equipment grant EIA-9818327.

- Study the impact of the number of nodes used by an application, the degree of multiprogramming (i.e., number of application instances running on each node), sizes of data read/written, the spatial/temporal locality of the application, and the degree of sharing exhibited across the applications instances to test the robustness of our strategy.

The rest of this paper is organized as follows. Related work on software-based I/O optimization is revisited in Section 2. Our implementation and rationale for our design decisions are discussed in Section 3. Section 4 presents experimental data showing the effectiveness of our approach. Finally, Section 5 summarizes the major contributions of this work and discusses ongoing work.

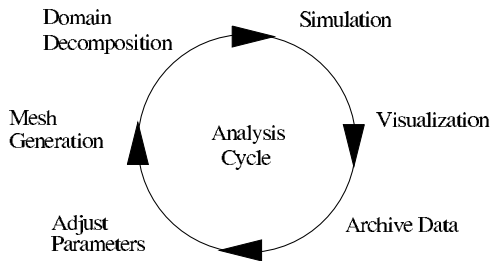


Figure 1. Computational science analysis cycle.

2. Related Work

Software solutions for high-performance I/O include optimizations by the file system, runtime I/O libraries, and even by the compiler. Several studies have focussed on runtime detection and optimization of I/O access patterns [13, 15, 11, 10]. A number of research groups have proposed runtime systems with user-friendly interfaces for array-based computations [8, 22, 19]. These routines help distribute disk-resident data sets across parallel processors and to reuse the limited memory space in a near-optimal fashion. However, in general, it is the user’s responsibility to decide which I/O calls to use, set up parameters, and insert them at appropriate points in the code. Compilation of I/O-intensive codes using explicit I/O has also been the focus of some research [6, 4, 17]. Brezany et al. [6] have developed a parallel I/O system called VIPIOS that can be used by an optimizing compiler. Bordawekar et al. [4] focussed on stencil computations that can be re-ordered freely due to lack of flow-dependences. They present several algorithms to optimize communication and to indirectly improve the I/O performance of parallel out-of-core applications. Paleczny et al. [17] incorporate I/O compilation techniques in Fortran D to choreograph I/O from disks along with the corresponding computation. Many of these studies, however, specifically target massively parallel processors (MPPs) and do not try to perform any inter-application optimization taking into account the data sets shared by multiple applications. Our work is different from those as we focus on a cluster environment and attempt to

optimize accesses to data sets shared by multiple applications.

Three prior systems –MPI-IO [1, 9], PVFS [7], and PPFS [12]–are more closely related to our work. MPI-IO [1] is a new API for parallel I/O as part of the MPI-2 standard and contains features specifically designed for I/O parallelism and performance. This API has been implemented for a wide variety of hardware platforms including networks of workstations [21]. The main optimizations in MPI-IO are for non-contiguous parallel accesses to shared data, mainly at the user-level. As a result, the user needs to have a thorough understanding of the application to glean the data access pattern, and be familiar with the numerous calls to invoke the appropriate optimization routine. Since implementations of the MPI-IO standard do not provide caching functionality, its response time is largely determined by the caching capabilities provided by the underlying file system.

PVFS [7] is a parallel file system for Linux clusters that presents three different APIs, and accommodates frequently used UNIX shell-commands. The work presented in this paper augments PVFS with a caching capability and quantifies its benefits. PPFS [12] is a user-level I/O library that has been implemented for numerous parallel machines and clusters. This system differs from the other two in that it offers runtime/adaptive optimizations as well in terms of caching, prefetching, data distribution and sharing. In comparison, the strategy discussed in this paper is oriented more toward optimizing accesses to data sets shared by multiple applications.

3. System Description

In the following discussion, we first give a quick overview of PVFS on which our system is built. This is followed by a description of our system architecture and details of its implementation.

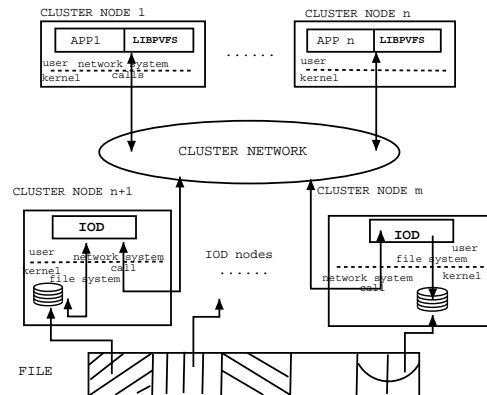


Figure 2. PVFS architecture. In this example, cluster nodes 1 through n run application processes, and nodes $n + 1$ through m hold data.

3.1. PVFS

The Parallel Virtual File System (PVFS) [7] aims to provide a high-performance parallel file system for applications running on Linux clusters. It provides several nice features such as a shared name space across the cluster, facilities for distributing/stripping the file data across the disks of the cluster nodes, and mechanisms for easy access to this data. In addition, it provides seamless transparent access to several existing utilities on normal file systems.

PVFS is a mainly user-level implementation, i.e., there is a library (*libpvfs*) linked to application programs which provides a set of interface routines (API) to distribute and retrieve data to/from the files striped across the cluster nodes (see Figure 2). In addition to the library, PVFS uses two other components, both of which run as daemons on one or more nodes of the cluster. This includes a meta-data server (*mgr*), to which *libpvfs* sends requests for meta-data information (access rights, directories, file attributes, etc.). There is one such instance of the meta-data server across the entire cluster. In addition, there are several instances of a data server daemon (called *iod*), one on each of the machines whose disk is being used to store the data. This daemon (again running at the user level) listens on sockets for requests from *libpvfs* functions to read/write data from/to its local disk. There are well defined protocols for exchanging information between *libpvfs* and an *iod*. The reader is referred to [7] for further details on the functioning of PVFS.

3.2. System Architecture and Implementation Details

As mentioned earlier, we would like to build on the existing capabilities provided by PVFS to leverage off its rich API and features. Further, we would like to provide our caching infrastructure in a fairly transparent fashion so that it is not even apparent to PVFS, leave alone the application. This makes our system fairly modular and scalable, making it easier for the caching benefits to be available as PVFS undergoes further revisions/improvements. This underlying rationale implies that we need to intercept all of the socket calls that *libpvfs* makes, and provide caching at that point. It should be noted that our cache is meant only for *iod* requests, and we do not cache any meta-data information (they necessarily go to the meta-data server) at this time. We decided to implement interception in the kernel since it can serve as a serializing point to track access patterns of different processes residing at the same node.

The cache at each node in our system is implemented as a dynamically loadable Linux kernel module (see Figure 3). The socket system calls made by *libpvfs* are directed to this module which takes on the job of first checking to see if the request can be fully serviced by the cache that it maintains, and if so the system call returns the necessary data. If the request cannot be fully satisfied by the cache, then the kernel module sends messages (using sockets) to the appropriate *iods* as before (note that it is possible that a part of the request may be satisfied in which case the external request is for only the missing data).

Intercepting and servicing the requests in the kernel module in a transparent manner to the existing PVFS infrastructure introduces several issues that we would like

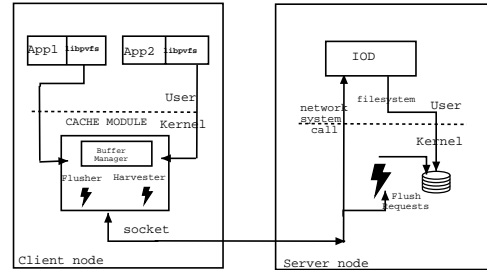


Figure 3. Our implementation.

to point out. We will illustrate this for the read protocol here (write protocol needs similar steps). The *libpvfs* read protocol aggregates all the reads to each *iod* and issues a socket request for each of them before waiting for an acknowledgment and the data packets from each *iod*. Consequently, our module has to first check which blocks are already present and discount these in the request(s). This can sometimes result in more than one request being sent to an *iod* if there is a cached block in the middle of a set of contiguous block requests. After issuing these requests the kernel module needs to return control back to *libpvfs* after modifying state to indicate that there are pending requests for certain blocks. When *libpvfs* makes receive socket calls for acknowledgments subsequently, our kernel module fakes these acknowledgments locally (without really getting any messages from the *iods*). On the actual data receive requests that *libpvfs* makes next, the module checks to see if the data has arrived in the buffers for which it has marked transfers as pending. If so, the data is copied to the application. Else, it waits for the data messages from the *iods*. One can envision our kernel module as maintaining a finite state machine for each socket; transitioning between states is based on the socket calls that *libpvfs* makes on that node and the incoming messages from the corresponding *iods*.

Our system implements a full-fledged buffer manager of blocks, requiring the implementation of hash tables, free list and dirty list. The used cache blocks (a block size of 4 KB is used in this study to make it equal to page size on which a lot of memory allocation routines are based) are chained in a hash table (open hashing) for faster retrieval and access. The kernel module implementing caching incorporates several background activities (kernel threads) in addition to those requested explicitly by the application such as read and write calls. We would specifically like to point out two kernel threads - *flusher* and *harvester* (Figure 3). On a write, cache blocks are not immediately propagated to the server (the write is performed on the cache and control is returned back to *libpvfs*). The blocks are marked dirty (kept in a dirty list), and this list is flushed periodically to the *iod* nodes. A server version of this *flusher* thread runs on the *iod* nodes, which listens on a separate socket for the flushes and writes those blocks to disk using direct local file system calls. Rather than allocate/free blocks on demand, which can incur higher latencies at those points, we have a *harvester* thread that becomes active whenever the number of blocks in the free list falls below a certain threshold. This thread frees up blocks till the free list size reaches a high water mark. We use

an approximate LRU replacement algorithm to free up the blocks (since exact LRU can result in a significant overhead at each read/write invocation), and preference for replacement is given to clean blocks over dirty ones. Since all these data structures are shared and manipulated/read by several concurrent activities, locking is needed for concurrency control. We have attempted to keep the granularity of locking to a fine level to increase the concurrency of the execution. It is to be noted that this locking is only for guarding accesses to data structures maintained within the kernel, and is not explicitly available to user processes (they are not file locks).

With the presence of multiple copies for data blocks, there is the issue of coherence/consistency. Our default read/write mechanisms do not worry about consistency, and a read simply returns the value in a version of the block that it finds (i.e., the write is only propagated to the cache and IOD — any subsequent read to the cache/IOD will get this value, but a read from a node that already has this block in its cache will not get this latest value). While this may not pose a problem for many applications, where read-write sharing is not common (as compared to read sharing) or where consistency is explicitly managed by the application itself, there are certain applications where ensuring consistency is critical and should be preserved by the underlying system. Consequently, in our system, we also provide a special version of the write, called `sync_write`, which not only propagates the writes to the cache/IOD, but also invalidates the caches which have a copy (so that subsequent reads on those nodes can go out on the network and get the latest copy). Coherence is maintained at a block granularity, and thus requires a directory entry per block (at the IOD) to keep track of the caches that have a current copy of that block.

4. Experimental Evaluation

4.1. Platform and Micro-Benchmark

The platform on which we implemented and evaluated the system-level caching is a Pentium/Linux cluster. Each node on this cluster has a 800 MHz Intel Pentium-III (Coppermine) microprocessor with 32KB of L1 cache, 256KB of L2 cache, and 128MB of PC-133 main memory. Each node is also equipped with a 20GB Maxtor hard disk drive and a 32bit PCI 10/100Mbps 3-Com 3c59x network interface card. All the nodes are connected through a Linksys Etherfast 10/100Mbps 16 port hub. For the experiments in this paper, we used a 6 node cluster configuration.

To test the effectiveness of the caching strategy, we used a customizable micro-benchmark which generates different access patterns depending upon the command line values. At the high level, this benchmark is a parallel application in which multiple processors execute read/write requests of specified sizes on shared (or private) file(s) at different offsets. The command line parameters include the size of the file, the size of each I/O request (denoted d), the number of nodes over which the application is parallelized (p), and a variable indicating whether read or write is to be performed. In addition, the I/O requests are performed within a loop whose iteration count can be modified (i.e., it is also a command line parameter). This loop allows

the user to read the desired amount of data. Another (command line) parameter, which we refer to as the degree of locality (denoted l), gives the user the flexibility of ensuring a pre-specified cache hit ratio in I/O accesses. In our experiments, we used a single instance of this micro-benchmark as well as multiple instances. We use the terms application instance and micro-benchmark instance interchangeably. Finally, the user can also specify the desired degree of data sharing between applications (denoted s). For example, $s = 50\%$ means that the application instances share 50% of their data references. Experimenting with different values (for these parameters) allows us to evaluate the robustness of our implementation. The results presented here are explicitly for those cases where each processor/node in an application accesses a distinct portion of the file (completely data parallel).

In the following, we compare two different PVFS implementations. The first of these is the original implementation without caching. This implementation is referred to as *no caching version*. The second implementation is the one that includes our system-level caching strategy. We call this the *caching version*.

4.2. Results

We have conducted extensive experiments varying the number of nodes used by an application, the degree of multiprogramming (i.e., number of application instances running on each node), the read and write patterns, sizes of data read/written, the spatial/temporal locality of the application, and the degree of sharing exhibited across the applications. In the interest of space, we present salient and representative results under four categories. Our first experiment examines how much overhead our caching implementation incurs compared to the original PVFS, considering just a single application executing on the cluster. Second, we examine the benefits of caching even with just one application executing. The third set of experiments look at caching benefits with more than one application executing with different degrees of sharing. Finally, we investigate the trade-offs between caching and parallelism which can be very useful in scheduling jobs on high-performance clusters. A cache size of 1.2MB at each node is used in all the experiments. Note that the use of a relatively small cache size allows us to better evaluate the continuing trend of large increases in dataset sizes. We would like to point out that the cost of the extra actions (cache lookup and then copying the required block to user space) on a socket call introduced by our cache implementation over the original PVFS socket code is less than 400 microseconds for a block of 4K bytes.

4.2.1. Caching Overhead

To investigate how much overhead is introduced by our caching module when there is really no locality in the workload to be exploited, we have run a simple experiment, taking as input one instance of the micro-benchmark described in Section 4.1 with $p = 4$. This micro-benchmark instance is executed with a zero degree of locality; that is, $l = 0$, meaning that all requests will result in cache misses, and with different values of read/write request sizes (that

is, the parameter d). The resulting time for each read and write request are plotted in Figures 4(a) and (b), respectively, as a function of the amount of data read/written by each request. The dotted lines show the performance of the original PVFS code and the solid lines show the performance of our caching version.

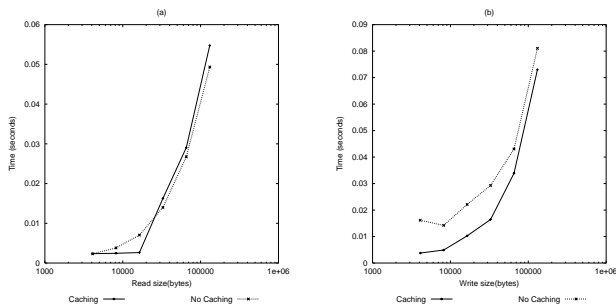


Figure 4. Overhead of caching for (a) reads and (b) writes in single application instance case with $p = 4, l = 0$

Since there is no locality to exploit and cache lookups will fail, it was expected that the caching version of PVFS would perform worse than the original version, however we find in Figure 4(a) that the differences between the two are not very significant. This suggests that the overhead of caching in our implementation is very small. When we examine the write performance in Figure 4(b), we find that the caching version performs better than the original version (with the differences being much more prominent for smaller d values). Remember that writes require only copying the data into the cache and returning back to the application as long as there is space in the cache. Propagating these writes to the iods takes place in the background by the flusher thread, hiding a lot of the network and I/O latencies for the application in the caching version. When d becomes large, the writes may need to block for availability of cache space, lessening the differences between the two versions at those d values.

4.2.2. Impact of Application Locality

In the previous section, the execution used $l = 0$ which depicts the worst case situation for the caching version. We now set $l = 1.0$ which is the best case situation for our caching strategy, and Figures 5(a) and (b) show similar graphs as before for this case.

We find substantial benefits from caching in these experiments with the differences between the caching and no-caching versions very prominent. The benefits are obtained for both reads and writes in this case, and they increase with larger request sizes. It should be noted that an individual request size cannot exceed the cache size in these experiments. At very small request sizes (8K or less), the overheads of caching become more significant, and this overhead is compensated for with the benefits of caching at larger request sizes.

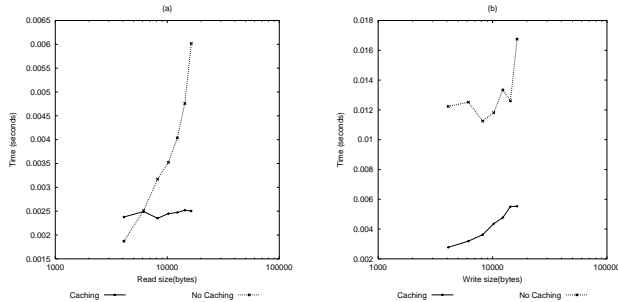


Figure 5. Comparison of caching and no caching for (a) reads and (b) writes in single application instance case with $p = 4, l = 1$.

4.2.3. Caching Benefits Across Applications

The earlier two experiments used only one instance of our micro-benchmark running at each node, and the cache is used to serve only one process. As was mentioned early on, one of the main motivations behind our caching PVFS implementation is the ability to service multiple applications from the same cache, especially with applications sharing data. To investigate this issue, we have next run an experiment with the same micro-benchmark as before instantiated twice; i.e., there are two instances of the micro-benchmark running on the same p processors (each node now runs two processes). We vary the number of processors used by each application instance ($p = 2, 4$), the degree of data sharing between the application instances ($s = 25\%, 50\%, 75\%, 100\%$), and the degree of locality in the application instance ($l = 0, 0.5, 1.0$) to capture the influence of each of these factors.

Figures 6 and 7 show the read performance for this experiment with the applications running on 4 and 2 processors respectively. In each of these figures, the first graph shows the performance with no locality exhibited by the application ($l = 0$) and the third graph shows the performance with excellent locality (i.e., we always find the necessary blocks in the cache). The second graph captures the scenario when around 50% of the references result in cache hits. The y-axis in all these graphs shows the total time for the application to complete as a function of different sizes of data read in each invocation to the file system. Note that as we increase the amount of data read per invocation in these experiments, the total number of calls to the file system itself goes down (since we are keeping the total amount of data read by the application constant). This is the reason why all the executions trend towards lower times with larger read sizes per invocation (on the x-axis).

Within each graph, we have run experiments with the two application instances sharing different amounts of data expressed as a percentage (degree of data sharing); i.e., a 25% data sharing indicates that a quarter of all the data read by the application is to a shared file (that is read by the other application instance as well) and the other 75% is to a private file. The percentage of sharing results in differential performance only for our caching version since the original (no caching) version will always issue network re-

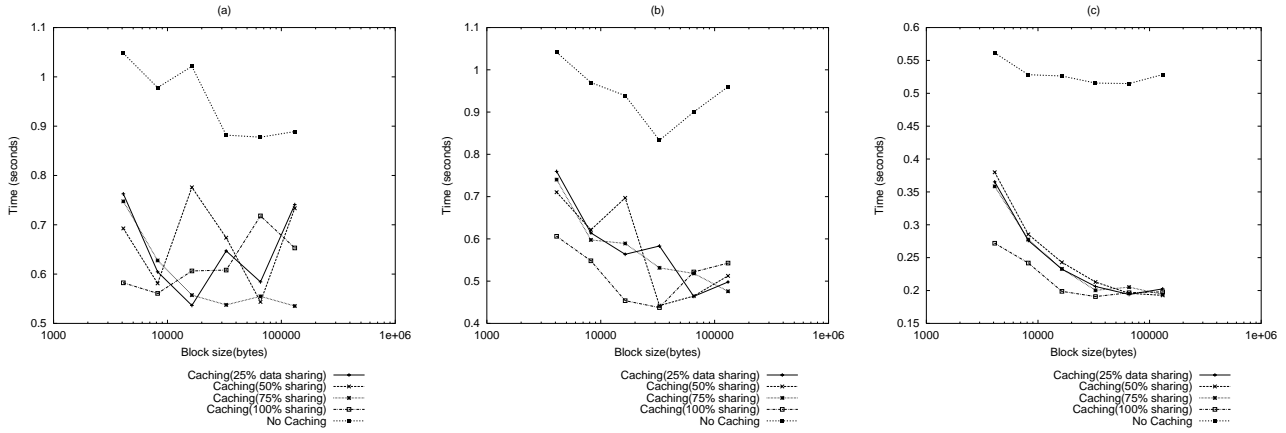


Figure 6. Two application instances performing reads ($p = 4$). (a) $l = 0$, (b) $l = 0.5$, (c) $l = 1.0$.

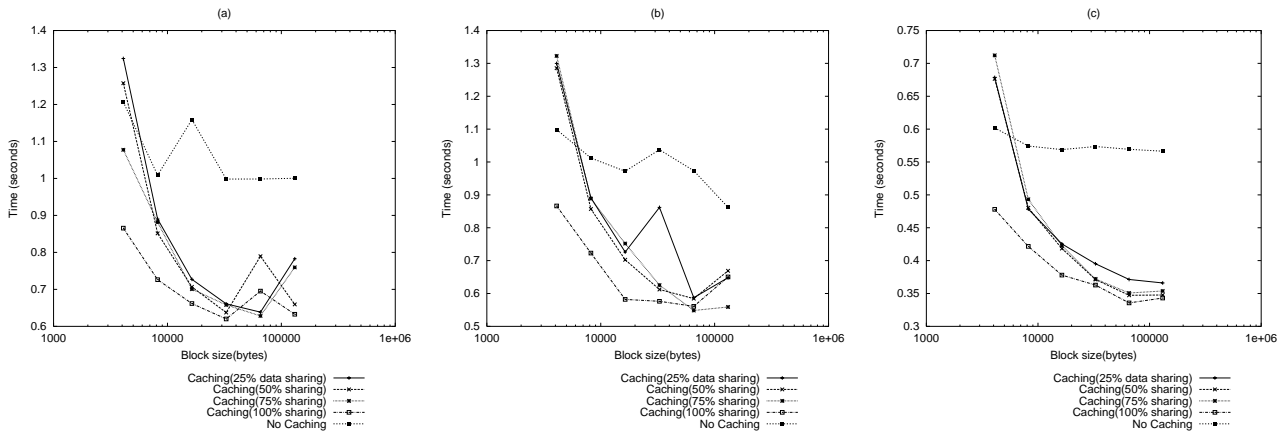


Figure 7. Two application instances performing reads ($p = 2$). (a) $l = 0$, (b) $l = 0.5$, (c) $l = 1.0$.

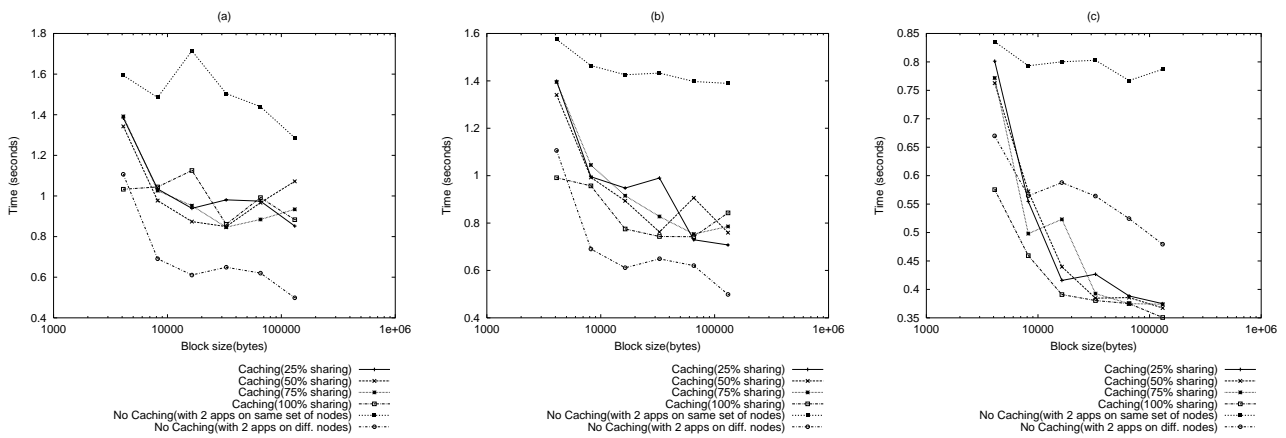


Figure 8. Comparing no caching version running applications on entirely different nodes (taking 6 nodes) with the caching version running them on same nodes (taking 3 nodes). (a) $l = 0$, (b) $l = 0.5$, (c) $l = 1.0$.

quests. Consequently, we show only one line for the original PVFS (no caching) version in these graphs, while the caching version is shown for the different data sharing percentages.

We first note that even in the $l = 0$ case where caching was not helpful in the single application instance execution earlier, the caching version does better than the original PVFS for nearly all non-zero percentages of data sharing. Even if one of the application instances incurs cache misses for all its blocks, the cache is able to meet many of the demands of the other instance running on the same node. This clearly illustrates the necessity for inter-application optimizations and the importance of the shared cache at each node. In general, as the percentage of sharing between the application instances increases, the benefits of the cache become more prominent.

When we move to higher degree of locality in the application ($l = 0.5, 1.0$), the benefits of the caching version becomes even more visible. Further, we find that when we compare the experiments for $p = 2$ and 4, the caching benefits for the larger p are more significant, suggesting the need and scalability of caching for large scale problems/parallelism and large clusters.

4.2.4. Can Caching Compensate for Any Loss in Parallelism?

Having demonstrated the benefits of caching in not only meeting the needs of any one application but also in executing applications sharing data concurrently, we next move on to an important issue that can have significant ramifications on scheduling applications on the cluster. Let us consider two applications sharing data, and assume that we want to schedule their processes on the cluster. If we have enough free nodes, then one option is to give each application its own set of nodes. With this approach, we are trying to improve the parallelism in the execution. The other approach is to time-share the nodes between these two applications. In this option, we can possibly reap the benefits of inter-application caching, with a loss of parallelism in the execution itself. If we find that the benefits of inter-application caching are significant enough to tilt the balance in favor of the second option, then the scheduler/system administrator can potentially have better chances for filling the other nodes (keep them free for any other incoming job). To investigate this issue, we have run two instances of our micro-benchmark using three nodes with these two options. We compare the caching version of the execution with both instances running on the same set of nodes (taking up only 3 nodes in all) with the no caching version of the execution running each instance on entirely different nodes (taking up 6 nodes in all) as well as running them all on the same set of nodes (again taking only 3 nodes in all). The resulting performance results are shown in Figures 8(a), (b), and (c) for $l = 0, 0.5$, and 1.0 , respectively. As before, we have considered different degrees of sharing between the application instances.

Note that the caching versions are expected to perform better than the no caching version running on the same set of nodes (one could think of these as an upper bound from the results of Section 4.2.3) which is the case here. On the other hand, we would like the caching versions to perform better than the no caching version running on differ-

ent nodes. In the $l = 0$ case, though we are doing much better than the no caching version running on the same nodes, the parallelism benefit that one gets from running the instances on different nodes is much higher than the inter-application caching effects. It should be pointed out that this is indeed a worst case scenario for our caching implementation, since an instance does not by itself exhibit any locality. With higher l , the effects of caching are coming into the picture to offset parallelism loss. When we move to $l = 1.0$, we find that caching benefits offset any loss of parallelism by executing the instances on the same node. This is a very important result that has not been addressed in-depth in previous work, which can have wide ramifications on scheduling and resource management for clusters. As is to be expected, the caching benefits favor these executions even further as the degree of sharing between the instances increases.

5. Conclusions and Ongoing Work

With clusters taking on demanding roles and often being subjected to high loads in multiprogrammed environments, it is important to be able to maximize system throughput by looking at relationships between applications concurrently executing, and exploit these relationships. Caching is a very well known approach for boosting performance of the storage hierarchy, and has been demonstrated to be very useful for parallel file system performance. However, the benefits of caching for multiple co-existing applications has not been addressed in prior work. Towards this goal of optimizing the execution of co-existing applications that share data, this paper has addressed one important issue of providing a shared cache to meet the demands of several applications at the same time. We have implemented a full-fledged buffer (cache) manager as a Linux kernel module atop a publicly available parallel file system, which can be used to service multiple applications at each node.

Apart from the detailed implementation, this study has also conducted extensive evaluations of the system using micro-benchmarks to study the impact of access patterns, size of reads/writes, the locality in the application, the locality across applications, and the impact of multiprogramming. Most of our experiments clearly illustrate the benefit of caching - not just for a single application but also across applications. In fact, one of our results suggests that inter-application locality can be exploited to such an extent that it may sometimes supplant even the benefits of parallelism, i.e. sometimes it may be better to co-locate two different applications to the same set of cluster nodes (so that each node is multiprogrammed) for cache locality, than running them on entirely different set of nodes (where they do not need to contend with each other or with anyone else for the CPUs). Such locality benefits can have important ramifications on scheduler designs.

We would like to point out that we have only begun to scratch the surface of inter-application optimizations and systems support for parallel I/O in this paper, and there are several ongoing and future efforts planned:

- We are extending the current system to also include a global cache that can be shared by all the nodes (the current cache is shared only by the application processes at a given node) before disk opera-

tions are really invoked. Though there are systems which currently do this, our goal is to introduce inter-application optimizations in this global cache.

- We plan to classify different sharing patterns and develop different I/O optimizations for each type of pattern. In particular, we are interested in addressing this issue from the viewpoint of inter-application sharing.
- We are also examining compilation techniques and runtime support to detect and exploit inter-application sharing patterns, for possible combining of I/O requests, prefetching, and other optimizations.

These issues need to be evaluated not only with the micro-benchmarks such as the one used here, but with real applications/datasets. While there are some I/O benchmarks available in the public domain for experiments, there is a lack of benchmarks containing groups of applications sharing data. Identification and characterization of such benchmarks is also an interesting topic that we intend to investigate in the future.

6. Acknowledgements

We would like to thank the anonymous referees for their helpful suggestions on improving the presentation of the paper.

References

- [1] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. <http://www.mpi-forum.org/docs>.
- [2] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.
- [3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.
- [4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the ACM-SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Santa Barbara, CA, 1995. ACM Press.
- [5] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic optimization of communication in compiling out-of-core stencil codes. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 366–373, Philadelphia, PA, 1996. ACM Press.
- [6] P. Brezany, T. A. Muck, and E. Schikuta. Language, Compiler and Parallel Database support for I/O Intensive Applications. In *Proceedings on High Performance Computing and Networking*, Milano, Italy, 1995.
- [7] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000.
- [8] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS-636, Syracuse University, NY, 1994.
- [9] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO Parallel I/O Interface. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 477–487. IEEE Computer Society Press and Wiley, New York, NY, 2001.
- [10] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. D. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, 1995.
- [11] C. S. Ellis and D. Kotz. Prefetching in file systems for MIMD Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:306–314, St. Charles, IL, 1989. Pennsylvania State Univ. Press.
- [12] J. V. Huber, Jr., C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFs: A High Performance Portable Parallel File System. In H. Jin, T. Cortes, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 330–343. IEEE Computer Society Press and Wiley, New York, NY, 2001.
- [13] T. M. Kroeger and D. E. Long. Predicting File-System Actions from Prior Events. In *Usenix Annual Technical Conference*, pages 319–328, 1996.
- [14] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.
- [15] T. M. Madhyastha. *Automatic Classification of Input Output Access Patterns*. PhD thesis, UIUC, IL, 1997.
- [16] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing 95*, 1995.
- [17] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, McLean, VA, 1995.
- [18] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.
- [19] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing 95*, San Diego, CA, 1995. IEEE Computer Society Press.
- [20] N. Talagala, S. Asami, D. Patterson, and K. Lutz. Tertiary Disk: Large Scale Distributed Storage. Technical Report CSD-98-989, UCB, 1998.
- [21] R. Thakur, E. Lusk, and W. Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Argonne National Labs, 1997.
- [22] S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a Portable Library for Scalable Out-of-Core Linear Algebra Computations. In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, 1996.
- [23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.