Discretionary Caching for I/O on Clusters*

Murali Vilayannur [†] Anand Sivasubramaniam [†] Mahmut Kandemir [†]

Rajeev Thakur [‡] Robert Ross [‡]

[†] Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
{vilayann, anand, kandemir}@cse.psu.edu
Phone: (814) 863-3627
Fax: (814) 865-3176

[‡] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{thakur, rross}@mcs.anl.gov
Phone: (630) 252-7162
Fax: (630) 252-5986

Abstract

I/O bottlenecks are already a problem in many large-scale applications that manipulate huge datasets. This problem is expected to get worse as applications get larger, and the I/O subsystem performance lags behind processor and memory speed improvements. At the same time, off-the-shelf clusters of workstations are becoming a popular platform for demanding applications due to their cost-effectiveness and widespread deployment. Caching I/O blocks is one effective way of alleviating disk latencies, and there can be multiple levels of caching on a cluster of workstations.

Previous studies have shown the benefits of caching — whether it be local to a particular node, or a shared global cache across the cluster — for certain applications. However, we show that while caching is useful in some situations, it can hurt performance if we are not careful about what to cache and when to bypass the cache. This paper presents compilation techniques and runtime support to address this problem. These techniques are implemented and evaluated on an experimental Linux/Pentium cluster running a parallel file system. Our results using a diverse set of applications (scientific and commercial) demonstrate the benefits of a discretionary approach to caching for I/O subsystems on clusters, providing as much as 48% savings in overall execution time over indiscriminately caching everything in some applications.

Index Terms: I/O, Clusters, File Systems, Compiler Optimizations, Caching.

1 Introduction

As processor speeds continue to advance at a rapid pace, accesses to the I/O subsystem are increasingly becoming the bottleneck in the performance of large-scale applications that manipulate huge datasets. This gap between CPU and I/O performance is exacerbated as we move to multiprocessor and cluster systems, where the compute power is potentially multiplied by the number of CPUs available to a problem. While one could argue that we can similarly use a large number of disks in parallel for improving I/O bandwidth, the latency of seeking to the appropriate location and performing the disk operation in addition to the overhead of the network transfer continues to hurt performance of many applications, especially those with non-sequential access patterns. Large buffers in memory (referred to as caches throughout this paper) are one way of alleviating this problem, provided we can achieve good hit rates. However, unlike the traditional instruction/data caches that are provisioned in the hardware of processor architectures, I/O caches are implemented in software and have much higher overheads. Further, the levels of I/O caching on some of the parallel environments (including clusters) can span machine boundaries, requiring network messages for cache accesses. It is thus very important to be able to determine what should go into an I/O cache and when we should avoid accessing it, apart from improving the hit rate itself. This paper addresses this important problem, presenting the design, implementation, and evaluation of a parallel file system's I/O subsystem that provides two levels of *discretionary caching*. The paper demonstrates the benefits of such discretionary caching mechanisms with compiler and runtime optimizations.

*Parts of this paper have appeared in the Proceedings of the 3rd IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid'03). This paper is an extension of these prior results, and includes a more extensive performance evaluation.

Clusters, put together with off-the-shelf workstations/PCs and networking hardware, are becoming the platform of choice for demanding applications because of their cost-effectiveness, upgradeability, and widespread availability. Clusters are finding their place in a plethora of environments, from academic departments to super-computing centers and even to the commercial world (e.g., for database, web and e-commerce applications). These platforms can benefit not only from the constantly improving processor/memory speeds, but also from the disk capacities and bandwidths. The multiple CPUs and their memories on these systems 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. One could either have disks attached to each cluster node with a SCSI-like interface (the corresponding node has to be involved in data transfers to/from such disks), or have disks accessible by everyone over a storage area network. While many of the issues/optimizations in this paper are applicable to both environments, we specifically focus on the former which is usually much cheaper, and thus more prevalent, alternative for the I/O subsystem (and are intending to investigate such issues for a storage area network in the future).

Many large-scale scientific applications are data-intensive, manipulating immense disk-resident data sets. These include applications from medical imaging, data analysis and mining, video processing, large archive maintenance, and so on. Commercial services such as web, multimedia, and databases on clusters are also demanding on the I/O subsystem. In addition, many high performance environments (particularly shared clusters within a department or a super-computing center) 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. All these issues make I/O optimizations an important and challenging problem for off-the-shelf clusters.

While the parallelism offered by the numerous disks in a cluster can alleviate the I/O bandwidth problem, it does not really address the latency issue which is largely limited by seek, rotational and network transfer costs. Caching data blocks in memory is a well known way of reducing I/O latencies, provided we can achieve good hit rates. I/O caching is typically implemented in software (not the disk/controller caches), and the overheads of cache lookup and maintenance can become quite high. Further, it has been shown [16] that we may need multiple levels of caching. For instance, in PPFS [16], a local cache at each node of the parallel system caters to the individual process requests at that node, and upon a miss goes to a shared global cache (running on one or more nodes of the cluster) which can possibly satisfy requests that come from different nodes. On such systems, the cost of going to the global cache — requiring a network message — and not finding the data there (before going to the disk) might be quite substantial. For instance, as our raw performance results will show, this approach turns out to be over twice as costly as directly getting the data from disk in several situations. Consequently, it becomes extremely important to intelligently determine what to place in the caches and when to avoid (i.e., bypass) the cache (particularly the caches whose look up costs are higher) on I/O requests. This largely depends on the data access patterns of the workload. To our knowledge, the issue of exploiting application behavior for such I/O cache optimizations on clusters has not been studied previously. There has been similar work (e.g., [18]) in the context of hardware data CPU caches, but the costs for I/O caching are of a much higher magnitude.

Rather than implementing all the APIs/feature of a full-fledged parallel file system to investigate these issues, we start with a publicly-available parallel file system — PVFS [5] — for Linux/Pentium clusters. We have considerably extended this system to incorporate a kernel level cache module at each cluster node to cater to all the requests (possibly different applications) coming from that node, which we refer to as the *local cache*. We also have implemented a shared *global cache* (between processes running on different nodes of an application, or even across applications) that runs on one or more nodes of the cluster. This global cache receives requests from the local cache and services them. If the lookup fails in the global cache as well, the request is forwarded to one or more nodes whose disks are used for striping the data. The experimental results presented in this paper are from a Pentium/Linux based cluster of workstations. 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. The global cache is run on one of the nodes that contains 384 MB of 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. Using this experimental system, this paper investigates/illustrates the following issues:

- We present several raw latency numbers for file reads and writes on this platform giving results for satisfying the requests from different levels in the cache hierarchy, and compare it to the original PVFS implementation which does not perform any explicit caching. Our results clearly demonstrate the benefit of caching. Even when missing from the local cache, going to the global cache, and fetching the data turns out to be better than the original PVFS in most cases. However, when we go via the global cache, only to find that the data is missing there, the overheads are significantly worse than not performing any caching altogether (as in the original PVFS). We also present some experimental data to show what hit rate is needed in the global cache to justify going through it.

- After pointing out the importance of discretionary data placement in the caches and bypassing them when needed,

we provide mechanisms within our system to explicitly specify whether a read/write should go through the local/global cache. The bypass capabilities can be conveyed to our caching layers through a kernel ioctl() call, and can be specified either by the application itself, or via the compiler or the runtime system.

- We show how simple compiler-based techniques are quite effective in benefiting from the caches, without incurring the extra overheads, for statically analyzable applications. We specifically present two techniques, one which determines what files should be accessed via the cache and what files should bypass the cache (which we refer to as coarse-grain optimizations), and the other which performs such discretionary accesses at a finer granularity.
- While compile time analysis can be employed in applications with statically analyzable code, we present a simple runtime approach for determining when to bypass the cache in situations where the codes are not readily analyzable or the sources are not available.
- All these optimizations are extensively evaluated with several applications/traces to show how they can be beneficial for improving cache behavior for parallel I/O.

The rest of this paper is organized as follows. The next section identifies some work related to this paper. Section 3 describes the system architecture and implementation details of our I/O subsystem on the Linux cluster, together with some raw performance numbers. The compiler-based and runtime-based optimizations are evaluated and compared in Sections 4 and 5. Finally, Section 6 summarizes the contributions of this paper and discusses directions for future work.

2 Related Work

Software work on high-performance I/O can be roughly divided into three categories: parallel file systems, runtime I/O libraries, and compiler work for out-of-core computations. A number of groups have studied automatic detection and optimization of I/O access patterns (e.g., see [22, 24, 11, 21] and the references therein). Others have proposed parallel file systems and I/O runtime systems that provide users/programmers with easy-to-use APIs [7, 35, 31, 26, 9]. While these systems allow users/programmers to exploit optimizations for I/O, it is still in general the user's responsibility to select which optimization to apply and determine the suitable parameters for it. Obviously, this puts a great burden on users, as in most cases it is not trivial to select what optimization(s) to use and the accompanying parameters. Our work instead tries to bring the advantages of I/O caching without much user effort.

Compilation of I/O-intensive codes using explicit I/O has also been the focus of some research (see [3, 1, 28] for example techniques that target out-of-core datasets). Brezany et al. [3] have developed a parallel I/O system called VIPIOS that can be used by an optimizing compiler. Bordawekar et al. [1, 2] have focussed on stencil computations that can be re-ordered freely due to lack of flow-dependences. They have presented several algorithms to optimize communication and to indirectly improve the I/O performance of parallel out-of-core applications. Palecnzy et al. [28] have incorporated I/O compilation techniques in Fortran D. The main philosophy behind their approach is to choreograph I/O from disks along with the corresponding computation. Many of these studies have, however, specifically targeted massively parallel processors (MPPs) and do not deal with selective data placement in caches. DPFS [32] is a parallel file system that collects locally distributed unused storage resources as a supplement to the internal storage of a parallel system. In contrast, our work is targeted for cluster environments with multiple levels of caching, that does not only benefit the processes of one application, but can also benefit several applications sharing datasets (through a global cache).

There has been a considerable amount of prior work on optimizing I/O and I/O caches [30, 10, 27, 29, 19, 20, 23, 33, 17, 4, 25, 13], some of which has been on clusters as well. Recently, [6, 40] have focused on buffer cache management policies in a multi-level buffer cache system. [40] proposes primitives for maintaining exclusivity in multi-level buffer caches, while [6] uses higher level cache eviction information to guide the placement of blocks in lower levels. Maybe the most closely related work to ours are the approaches presented in three prior systems, namely, MPI-IO [14, 8], PVFS [5], and PPFS [16]. MPI-IO [14] is an 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 clusters [34]. 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 numerous programming interfaces to invoke the appropriate routines. Since MPI-IO itself does not specify any caching functionality, its response time is largely determined by the caching capabilities provided by the underlying file system or the MPI-IO implementation. PVFS [5] is a parallel file system for Linux clusters that presents three different APIs, and accommodates frequently used UNIX file tools. Its optimizations for non-contiguous data are perhaps less powerful than MPI-IO's optimizations. The work presented in this paper augments PVFS with a local and global caching capability, benefiting from its rich original APIs. PPFS [16] is a user-level I/O library that has been implemented for several parallel machines and clusters. This

system differs from the other two in that it offers runtime/adaptive optimizations (not just an API) as well in terms of caching, prefetching, data distribution and sharing. The differences of our work from PPFS are in that we are examining the benefits of compiler/runtime directed cache bypassing towards optimizing the hit rates of one or more applications running on the cluster.

3 System Architecture

Our system builds on the architecture of the Parallel Virtual File System (PVFS) [5] since we did not want to re-invent the APIs and mechanisms for providing a shared name space across the cluster, and facilities for distributing/striping the file data across the disks of the cluster nodes. PVFS also provides seamless transparent access to several existing utilities on normal file systems. Since all these provisions are already provided in a publicly distributed parallel file system, we have opted to build upon this system in this work rather than re-implement all these features. We briefly go over some key architectural features of PVFS and then discuss our contributions.

3.1 PVFS

The original 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. In addition to the library, PVFS uses two other components, both of which run as daemons on one or more nodes of the cluster. One of these is a meta-data server (called *mgr*), to which libpvfs sends requests for meta-data information (access rights, directories, file attributes, etc.). 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 on clients to read/write data from/to its local disk using normal Linux file system calls. There are well-defined protocols for exchanging information between libpvfs and IODs. For instance, when the user wants to read file data that is striped across several IODs, libpvfs converts this request into several requests (one for each IOD involved), sends these requests to the IODs using sockets, waits for an acknowledgment from each of them, following which it waits for the data sent by the IODs. This data is then collated and returned to the application process. On a write, libpvfs sends out the requests, following which the relevant data is sent to each IOD. Each IOD then sends back an acknowledgment indicating how much data was actually written to check for error conditions. The reader is referred to [5] for further details on the functioning of PVFS.

3.2 Overview of System Architecture

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 wanted to provide our caching infrastructure in a fairly transparent fashion so that it is not even apparent to a large part of the PVFS implementation, leave alone the application. This 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 at this time (i.e., they always go to the meta-data server).

Our system provides two levels of caching — a *local cache* at every node of the cluster where an application process executes, and a *global cache* that is shared by different nodes (and possibly different) applications across the cluster, with the possibility of skipping either of them as illustrated in Figure 1. The design and implementation of the local cache at each node is described in an earlier work [38], and here we quickly go over it for completeness, and then concentrate on the global cache.

Local Cache

There are two alternatives for implementing the local cache at each node. One option is to implement the caching within the library that is linked with the application (user-level). However, with this approach we do not have the flexibility of sharing cache data between application processes running on the same node. This is the reason why we opted to implement the local cache within the Linux kernel (a dynamically-loadable module), that can be shared across all the processes running on that node. Only when the request misses in this cache (either all or some of the request cannot be satisfied locally), is an external request initiated out of that node, either to the global cache or to the IODs as will be explained later. This cache is implemented using open hashing with second chance LRU replacement. There is a dirty list (which keeps track of all the cache frames which have been modified while in cache), a free list (which keeps track of all the unused cache frames), and a buffer hash to chain used blocks for faster retrieval and access. The hashing

function takes as parameters the inode number of the file and the block number to index the buffer hash table. There are two kernel threads called *flusher* and *harvester* in the implementation. Writes are normally non-blocking (except the *sync_write* explained later), and the flusher periodically propagates dirty blocks to the global cache/IOD. The harvester is invoked whenever the number of blocks in the free list falls below a low water mark, upon which it frees up blocks till the free list exceeds a high water mark. A block size of 4K bytes is used in our implementation. Note that such a kernel implementation automatically allows multiple applications/processes to share this local cache, thus making more effective use of physical memory.

Global Cache

The global cache, as explained earlier, adds one more level to the storage hierarchy before the disk at the IOD needs to be accessed. There are numerous questions/alternatives when implementing the global cache and we go over them in the following discussion, explaining the rationale behind the choices we make specifically in our implementation:

- *Should there be a global cache for each file, or should all files share the same cache?* While there may be some possibility for detecting access patterns across datasets for optimizations, our current system uses a separate global cache for each file. If there is little file sharing across applications, or even across parallel processes of the same application, then the requests would automatically distribute the load more evenly with this approach.

- *Should each application have its own global cache, or should we share a global cache across applications?* Since we would also like to be able to perform inter-application optimizations based on sharing patterns, we have opted to share the global cache across applications. This can help one application (even its cold references) benefit from the data brought in earlier by another from the cache. There is, however, the fear of worse miss rates if there is interference because of such sharing, and these are points that our cache bypass mechanisms will address later. This feature is one key difference between our system and PPFS [16] where the global cache is intended for optimizations within the processes of a single application. At this point, we would like to mention that our system does suffer from scalability issues and performance may start to drop beyond a particular number of client nodes due to the centralized nature of the global cache. However, the focus of this work is on techniques for intelligent caching of data in file-system caches and we are looking at scalable techniques as part of our future work. Further, providing a separate global cache for each file as explained above can ease some of this bottleneck.

- *Should we distribute the global cache across the cluster?* While distribution is a good idea in terms of alleviating contention, there are a couple of drawbacks. First, depending on the granularity of distribution, it may be difficult to perform certain optimizations (such as prefetching) if one node is not the repository for all the file data. Second, two levels (one between the IODs and the global caches, and one between the global caches and local caches) of multiplexing and demultiplexing the data may be needed. We, instead, opted to have a centralized global cache for each file. However, since we have a separate global cache for each file, we can have separate global caches on different cluster nodes serving different files, and that can alleviate some of the contention problems which may arise.

- *Should the global cache be implemented as a user process or as a kernel module?* The reason for a kernel level implementation for the local cache is due to the need for trapping all application requests coming at that node from the different processes via the PVFS calls. However, with the global cache, TCP/IP sockets are being explicitly used for sending messages to it from the individual local caches regardless of which application process is making a call. The convenience and flexibility (option of busy-waiting) of a user-level implementation has led us to implement the global cache for a specified file as a stand-alone, user-level daemon running on a specified node of our cluster.

Each global cache in our system is, thus, a user level process serving requests to a specific file running on a cluster node, to which explicit requests are sent by the local caches, and is shared by different applications. The internal data structures and activities of the global cache are more or less similar with those for the local cache that were described earlier. One could designate such global caches on different nodes (for each file), particularly on those nodes with larger physical memory (DRAM). Consequently, this architecture is also well suited to heterogeneous clusters where one or more nodes may have larger amounts of memory than the others.

Reads/Writes

Figure 1 gives a schematic overview of our system. Let us now briefly go over a typical read operation (there could be some differences when one or more levels of caching are disabled as will be discussed later) to understand how everything

works when an application process on a node makes a read call, possibly to several blocks that span different IODs. The original PVFS library on the client aggregates the requests to a particular IOD, before making a socket request (kernel call) to the node running that IOD. Our local cache intercepts this call in the kernel and checks to see if all or even a part of it can be satisfied locally. If the entire request can be satisfied without a network message, then the data is returned to the PVFS library and the application proceeds. Otherwise, the local cache module accumulates a list of requests that need to be fetched. A subsequent message is sent to the global cache with these requests (Note that this may change, and the requests are directly sent to IODs if the global cache is bypassed). The multi-threaded global cache keeps listening on a dedicated socket for requests, and upon receiving such a message looks up its data structure. If it can satisfy the requests completely from its memory, it returns the data to the requesting local cache. Otherwise, it sends a request message to each of the IODs holding corresponding blocks, stores the blocks in its memory when it gets responses from the IODs, and then returns the necessary data to the requesting local cache. A write operation works similarly except that the writes are propagated in the background (using the flusher thread described earlier), and control is returned back as soon as the writes are buffered.

The above read and write operations are the most common, and can benefit significantly from spatial and temporal locality in the caches. However, with the presence of multiple copies for data blocks, there is the issue of coherence/consistency. The above 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 global cache and IOD — any subsequent read to the global cache/IOD will get this value, but a read from a node that already has this block in its local 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. Consequently, in our system, we also provide a special version of the write, called *sync_write*, which not only propagates the writes to the global cache/IOD, but also invalidates the local 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 to keep track of the local caches that have a current copy of that block. We maintain this directory at both the global cache and the IOD. The need for the latter would be more clear when we discuss global cache bypassing later. The actual set of local caches with a copy would involve merging these two directory entries for a block. On a system where there is no global cache bypassing (all requests go via the global cache), the directory at the IOD would be completely null. Local caches that bypass the global cache would update the directory at the IOD rather than at the global cache. A sync_write is thus an additional overhead (over normal writes), involving looking up the directory entries, and invalidating any copies, in addition to propagating the write itself. It would thus be more prudent to use the normal writes as far as possible, and use sync_write only when coherence is needed (or when one is not sure).

3.3 Performance of Primitives and Micro-benchmarks

Before we go any further into our optimizations, we would first like to present some raw latency numbers and micro-benchmark results for read and write performance with the presence/absence of local/global caches. For these experiments, the local cache size was fixed at 2MB (500 data blocks), while the global cache size was fixed at 40MB (10000 data blocks). Also, a stripe size of 32KB was used in all our experiments.

Raw Latencies for Reads/Writes

In the first set of results (see Table 1), we give the read latencies for a file striped over different number of IODs (1 to 4). In these tables, `Pvfs` denotes the read latency of the original PVFS system which does not use any caching (local or global). `Local Hit` indicates the latency when the access is satisfied from local cache and `Local Miss` is the latency when the access misses in the local cache and is satisfied from one or more IODs. The latter case thus captures the execution on a system without a global cache. `Global Hit` and `Global Miss`, on the other hand, denote the cases when the access misses in the local cache (i.e., a local cache lookup is still needed) and hits and misses, respectively, in the global cache.

From these numbers, we clearly see that the local cache hits (`Local Hit`) can substantially lower read costs compared to the original PVFS implementation. On the other hand, if the locality is not good, causing us to miss in the local cache (i.e., `Local Miss`), the performance becomes worse than the original PVFS for all request sizes because of the overheads in looking up the local cache. Therefore, it is not only important to improve the hit behavior of the local cache, but it is also meaningful to bypass the local cache on certain lookups if we feel that it is going to miss.

When we next move to the scenarios with the accesses to global cache (misses in local cache), we first see that the global cache can lower access times, provided the data is present there, compared to the original PVFS without caching in

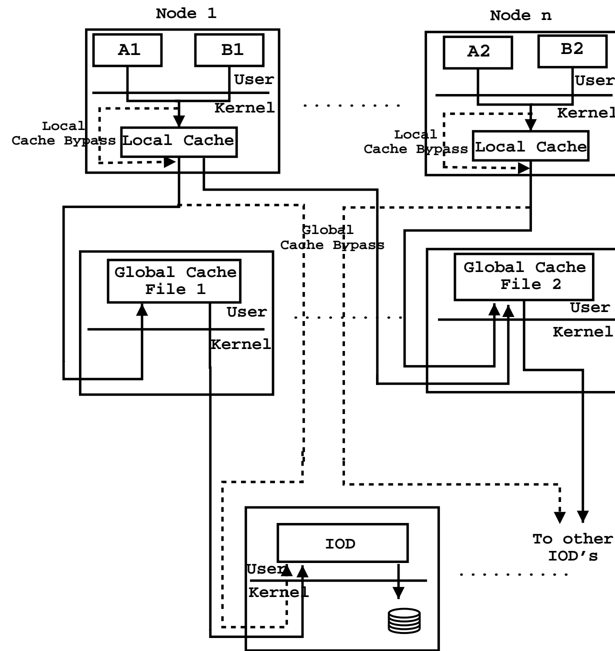


Figure 1: System architecture. Nodes 1..n are the clients where one or more application processes run, and have a local cache present. Upon a miss, requests are either directed to the global cache (one such entity for a file), or is sent directly to IOD node(s) containing the data in the disk(s).

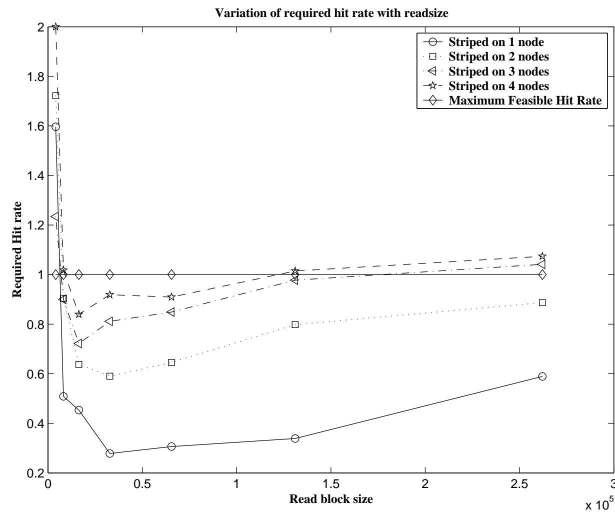


Figure 2: Graph showing the minimum required hit-rate at global cache for good performance

Table 1: Read times (in ms) for different request sizes and number of IODs ($|IOD|$).

Request Size ⟶	4K	8K	16K	32K	64K	128K	256K
	$\|IOD\| = 1$						
Pvfs	1.09	2.27	4.31	9.48	19.04	38.52	54.04
Local Hit	0.67	0.68	0.72	0.80	0.97	1.59	2.85
Local Miss	1.25	2.28	4.61	9.54	20.77	44.23	67.54
Global Hit (Local Miss)	1.43	1.71	2.44	4.26	8.14	15.28	25.91
Global Miss (Local Miss)	2.00	2.85	5.86	11.49	23.85	50.42	94.38
Required HR	1.59	0.50	0.45	0.27	0.30	0.33	0.58
	$\|IOD\| = 2$						
Pvfs	1.12	1.99	3.82	7.84	14.16	24.09	47.79
Local Hit	0.74	0.83	1.03	1.38	2.43	4.34	8.56
Local Miss	1.32	2.08	4.36	8.07	18.59	36.49	52.92
Global Hit (Local Miss)	1.51	1.85	2.62	5.01	8.32	17.77	39.92
Global Miss (Local Miss)	2.05	3.31	5.93	11.91	24.78	49.06	109.36
Required HR	1.72	0.90	0.63	0.58	0.64	0.79	0.88
	$\|IOD\| = 3$						
Pvfs	1.08	1.83	3.52	6.17	12.00	20.04	36.86
Local Hit	0.75	0.84	1.01	1.41	2.42	4.50	8.67
Local Miss	1.31	2.35	4.48	8.19	18.96	26.90	40.63
Global Hit (Local Miss)	1.23	1.66	2.45	4.80	8.67	19.26	39.38
Global Miss (Local Miss)	1.87	3.36	6.30	12.06	30.71	54.14	100.34
Required HR	1.23	0.90	0.72	0.81	0.84	0.97	1.04
	$\|IOD\| = 4$						
Pvfs	1.08	1.63	3.33	5.32	10.64	19.06	33.79
Local Hit	0.76	0.84	1.01	1.40	2.41	4.68	8.89
Local Miss	1.32	2.18	4.50	8.61	14.47	21.76	38.96
Global Hit (Local Miss)	1.48	1.67	2.80	4.70	9.10	19.50	38.87
Global Miss (Local Miss)	1.88	3.87	6.10	12.33	26.07	49.62	107.63
Required HR	2.00	1.01	0.83	0.91	0.90	1.01	1.07

Table 2: Write times (in ms) for different request sizes and number of IODs ($|IOD|$).

Request Size ⟶	4K	8K	16K	32K	64K	128K	256K
	$\|IOD\| = 1$						
Pvfs	0.68	1.03	1.97	3.95	7.83	15.94	31.09
Caching	0.55	0.56	0.60	0.96	1.05	1.76	3.15
	$\|IOD\| = 2$						
Pvfs	0.68	1.27	1.90	3.77	9.86	15.44	29.61
Caching	0.60	0.67	0.84	1.43	2.04	3.70	7.19
	$\|IOD\| = 3$						
Pvfs	0.68	1.04	1.85	3.62	8.23	15.74	29.40
Caching	0.59	0.68	0.87	1.37	2.08	4.01	7.79
	$\|IOD\| = 4$						
Pvfs	0.68	1.02	1.95	3.58	8.18	15.87	29.09
Caching	0.60	0.68	0.90	1.55	2.17	4.30	8.02

many cases (i.e., requests larger than 4KB). It is also better than fetching the data directly from IODs upon a local cache miss (`Local Miss`). However, global cache miss costs are substantially higher than any of the other cases because of the additional message hop and serialization overhead that occurs in the critical path and the associated lookup costs. This suggests that if we want to incorporate and benefit from the global cache, it is very important to keep its hit rate quite high. In fact, the `Required HR` rows in Table 1 give the minimum hit rates that are needed (for each request size) to tilt the balance in favor of the global cache compared to the original PVFS. A value larger than 1 in these rows indicate that it is impossible to generate better results than the original PVFS using that request size and the number of IODs. Figure 2 shows the same behavior plotted as a graph. This again means that we need to be very careful on what to put in the global cache and when to avoid going through it. Further, we can observe that the benefits of global caching (look at the last row showing required hit rate) are most significant when request sizes are not at either extreme. At very small request sizes, the overhead of global caching itself is more significant. At the other end, large amounts of data can cause more capacity misses, leading to poor temporal locality. Another point to note is that when the number of IODs involved in the access increases, the cost of a global cache miss becomes even more significant. This is because the global cache has to amass the data coming in from different IODs and then send them sequentially to the requester, while all the IODs could have potentially sent them in parallel to the requester if the global cache was not involved.

Table 2 gives the times for write operations to return back to the application after they are issued with different number of IODs involved. We compare the performance of the original PVFS code (denoted `Pvfs`) with our system having a local cache (denoted `Caching`). We are not separately giving the costs as in the read table (Table 1) for the other scenarios as they are comparable to the scenario with a local cache (the writes are simply accumulated in the local cache, and a background activity — flusher — propagates these writes to either the global cache or the IOD). We do not buffer writes

of an application, when there is not enough space left on the local cache. Hence the cost of writes whose sizes are greater than the local cache size is comparable to the cost of the original PVFS implementation. We can see that write stall times are significantly lower because of this feature as is to be expected. It is to be noted that the savings that will be presented later in this paper with our optimizations are not a result of these non-blocking writes, since we show savings even over the scenarios that cache everything in the local/global caches (which also performs non-blocking writes).

Micro-benchmark Results

While our later experiments will evaluate caching using real benchmarks, we wanted to stress the system along different dimensions, and employed a micro-benchmark to do so. Our micro-benchmark is parameterized based on s (the maximum size for a read/write operation in blocks, where a block is defined to be the same size as the granularity of the caches) and o (the maximum offset within a file in blocks from which the next read/write is initiated). The micro-benchmark program iteratively goes over a number of operations, randomly picking whether it is a read or a write with equal probability. The size of this operation is also picked randomly between 1 and s blocks, and the starting offset within the file for the operation is picked, again randomly, between 0 and o blocks. Note that a small value of o will automatically provide good locality, and we can tune these parameters to mimic different access patterns.

Instead of presenting all the results, we discuss here one representative case with one IOD being employed, $s = 2$, and for three different values of o: 10, 600, and 25000 (see Figure 3). Note that the locality progressively gets worse from $o = 10$ to o=25000. When the locality is very good (o=10), the working set is contained well within the local cache, and the schemes which use the local cache perform much better than those without it. We also note that the global cache alone scheme still does turn out to be better than the scheme without any caching. Even though the hit rates are quite high for the global cache, its overheads cause it to perform much worse than for schemes with a local cache. At the other end of the spectrum, when the locality becomes very poor (o=25000), the working set is not well exploited by any of the caching schemes, and their associated overheads cause them to perform worse than a scheme without any caching. The more interesting results are those for o=600, where the working set overflows the local cache, but is captured by the global cache (which is larger). Consequently, the two schemes which use a global cache provide much better performance than a scheme without any caching or a scheme with just a local cache.

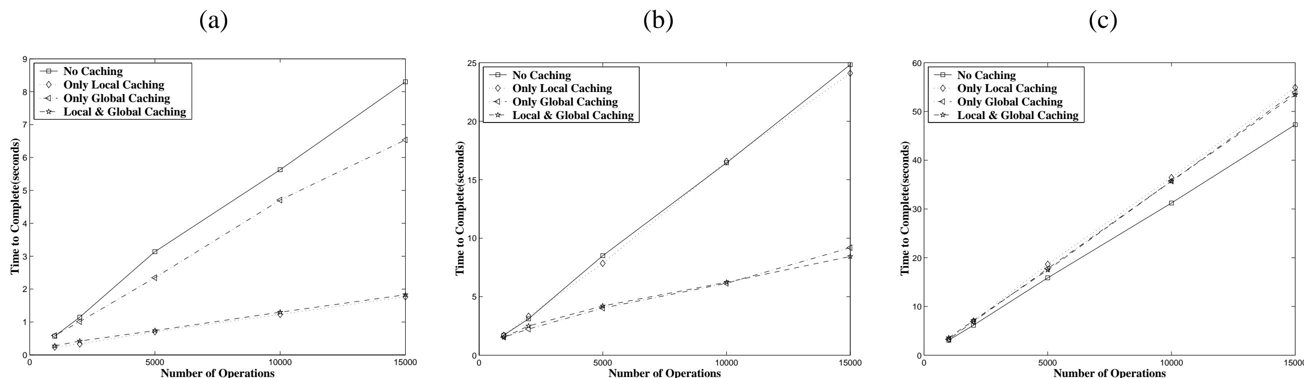


Figure 3: Micro-benchmark running on 1 node, File striped on one IOD, s = 2, (a)o = 10, (b) o = 600, (c) o = 25000.

In the earlier experiment, the micro-benchmark is run on a single node. We have also run the same micro-benchmark on different nodes, and the data being striped across different IODs (using a stripe size of 32KB). In Figure 4, we show the results for one such scenario with three IODs used to distribute the data. We observe similar trends to those that we saw earlier. The only slight difference with the poor locality situation (o=25000) is that the local cache alone execution is not much worse than without any caches because the local cache overheads are not too significant.

3.4 Cache Bypass Mechanisms

The results in the previous subsection indicate that it is important to provision a local and a global cache for good performance. However, our results also show that it is equally important to be very careful in deciding what data to place in these caches and when to avoid/bypass them.

Our system provisions mechanisms for bypassing the local and/or global caches for a read or write. Our system does not require any different read/write calls to specify that a cache needs to be bypassed since that can get cumbersome, and

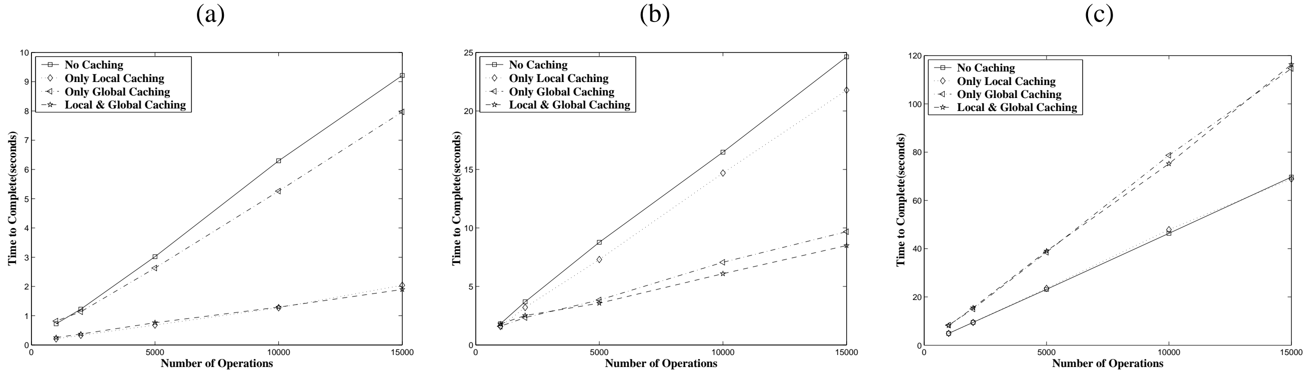


Figure 4: Micro-benchmark running on 2 nodes, File striped on three IODs, s = 2, (a) o = 10, (b) o = 600, (c) o = 25000.

it is not clear how such a mechanism can be effectively used by application programmers. Instead, we provide the notion of a *segment* — a certain number of contiguous file blocks (unless explicitly stated otherwise, a segment of 4 blocks was used in the experiments) — with a set of bits determining what actions to be performed on a read/write. For each operation (read or write), we have two bits, one for specifying whether that operation for the segment needs to go through the local cache and another for whether it needs to go through the global cache. We thus provide a segment-level granularity for cache bypassing.

These (segment) bits can be set via a system call that updates a data structure in the underlying kernel module (implementing the local cache) at each node. When a read/write call is made, this bitmap data structure is consulted to find out whether to look up the local cache, and whether to route the request to the global cache or directly to the IOD. The system call to set these bits can either be explicitly invoked by the application program or be invoked by instructions inserted into the code by the compiler. These bits can also be set by the runtime system based on previous execution characteristics. In the default configuration, all operations go via the local and global caches for all segments. The rest of this paper explores the benefits of cache bypassing, and ways of initiating such bypassing with the compiler and the runtime system. While it is also possible to adopt a user-based strategy where the application programmer sets these bits explicitly, we believe that such an approach would be very difficult for the user (investigating profile based techniques and tools for doing this is part of our future research agenda). Also, we specifically focus on bypass mechanisms for the global cache in this paper, whose overheads on a miss are much more significant than the corresponding overheads for the local cache.

4 Compiler-directed Cache Bypass

Previous discussion emphasized the importance of careful management of the global cache space. An optimizing compiler can help us identify what data should be brought into the global cache. It can achieve this using at least two different strategies. We assume here that the data for each array corresponds to a different file. In the first strategy, the compiler adopts a coarse-granular approach and determines the arrays that are used frequently program wide. It achieves this by estimating (at compile time) the number of accesses to each array in the code. More specifically, for each loop nest, the compiler counts the number of references to each array and multiplies these counts by the trip counts (the number of iterations) of all enclosing loops. If there is a conditional flow of control (e.g., an if-statement) within the loop, the compiler conservatively assumes that all possible branches are equally likely to be taken. Note that if we have profile data on branch probabilities, it is straightforward to exploit it for obtaining a more accurate estimate. Another potential problem is the compile-time unknown loop bounds. In such cases, the compiler can estimate the number of accesses symbolically. Note that previous symbolic manipulation techniques (e.g., [15, 12]) can be used here for this purpose. After doing such analysis, the compiler uses the global cache for reads/writes to the files (arrays) with the most references (depending on how many such files can fit in the global cache).

An important drawback of this coarse-granular strategy is that it fails to capture short-term localities. For example, in a given large, I/O-intensive application, an array might be accessed very frequently in the first half of the application and is not accessed in the second part. However, the strategy described above can continue to cache the segments of this array in the second part of the application if the overall (program wide) access count of this array is larger than those of the others. Our second strategy tries to eliminate this drawback of the coarse-grain method by managing the global cache space in a nest basis focusing on segment granularity.

Specifically, in our second strategy, the compiler determines the blocks that will be accessed in each nest separately.

The id's of a subset of these blocks are then recorded at the loop header. This subset contains the most frequently used blocks in the nest. By doing this, the second strategy tries to capture short-term localities and manages the global cache space at a finer granularity. Then, the segments corresponding to the most frequently used blocks are cached. Note that this approach can be expected to result in better global cache hit ratio than the first strategy. It should also be noted that determining the blocks that will be accessed by a loop nest is possible as in our applications there is a one-to-one correspondence between arrays declared in the program and disk-resident files (i.e., our applications use a separate file for each array that they manipulate). Therefore, the compiler can associate the array elements with the blocks. Also, as in the case of coarse-grain approach, this approach can take advantage of profile data (e.g., on branch probabilities) where available. Further, again as in the previous case, it can employ symbolic expression [15, 12] manipulation when loop trip counts are not known at compile time.

We implemented both these strategies using the SUIF compiler infrastructure [39] and evaluated them using codes where data access patterns are statically analyzable. SUIF consists of a small, clearly documented kernel and a toolkit of compiler passes built on top of the kernel. The strategies that were described above have been implemented as SUIF passes that performs the required analysis and writes the output to a file. We present here results with I/O-intensive versions of two Spec benchmarks: `tomcatv` and `vpenta`. While the original codes manipulate arrays directly in memory, we extended them to read/write these arrays from data files explicitly, before manipulating them in memory. The results are shown for `tomcatv` in Figures 5 and 6(a) as a function of the problem size (local cache size of 400KB, global cache sizes of 20 MB and 200 MB) and as a function of the global cache size (keeping the problem size fixed at 1500 - this corresponds to matrices of size 1500*1500 doubles manipulated in the application), respectively. The corresponding results for `vpenta` are given in Figures 7 and 6(b). In each of these figures, we compare the performance of four different executions: (a) a scheme with no caching (and hence no compiler optimizations for I/O); (b) a scheme with local and global caches without any compiler optimizations for I/O; (c) a scheme with local and global caches in conjunction with coarse-grain (file level) compiler optimizations, and (d) a scheme with local and global caches in conjunction with fine-grain compiler optimizations.

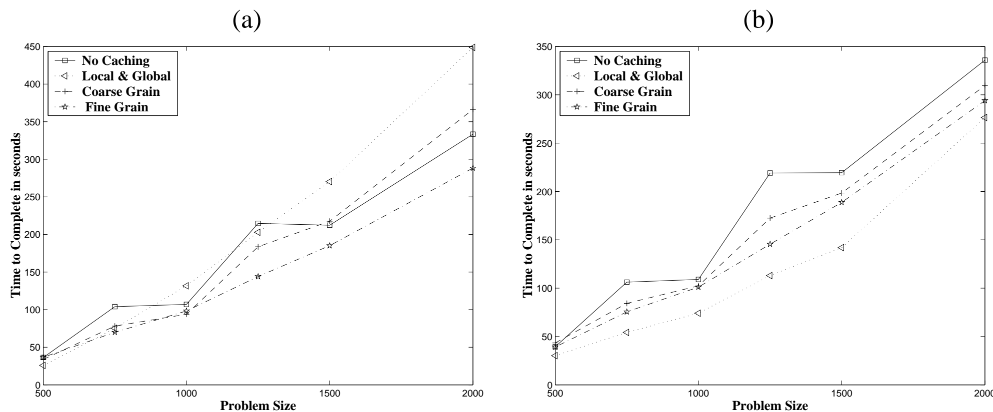


Figure 5: `tomcatv`: impact of problem size (a) Global cache is 20MB, (b) Global cache size is 200MB.

Examining Figure 5(a), we find evidence in the earlier arguments that blindly caching everything in the local and global caches can sometimes worsen performance. Specifically, we observe that the `No Caching` alternative does better than the `Local & Global` option (i.e., caching everything indiscriminately), especially at larger problem sizes. The overheads of going to the global cache and not finding the required blocks in it contribute to this behavior. Performing compiler optimizations at the coarse (file) granularity does give better performance than caching everything, but it still does worse than not caching anything. However, we can see that the fine granular approach, gets the benefits of the global cache, and does turn out to be a better alternative than not caching (because it avoids consulting the global cache when it feels the data may not be present). This benefit improves as the problem size gets larger (relative to the global cache size). Evidence for the last statement is further substantiated when we examine the executions with a much larger global cache in Figure 5(b). Here, the hit rates in the global cache are much higher, and the always cache option is a better idea. As the global cache gets larger, the selectively cache option can possibly limit some data from benefiting from this compared to caching everything. All these observations are reiterated when we look at the impact of global cache capacity for a fixed problem size in Figure 6(a). The benefits of selective caching/bypassing is much more significant at small cache sizes, and the always cache option becomes better only with larger global caches. The results for `vpenta` (given in Figures 7 and 6(b) are similar to many of those observed with `tomcatv`, except that the magnitude of the differences are less

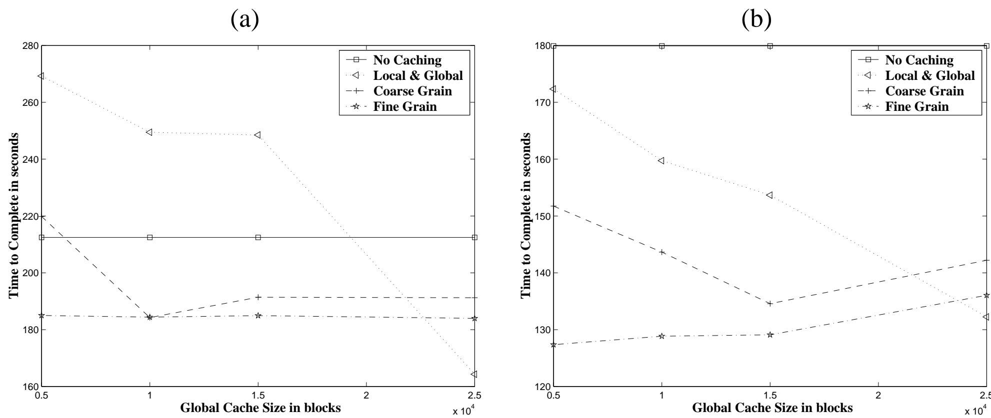


Figure 6: Impact of global cache size for a problem size of 1500.(a) tomcatv, (b) vpenta

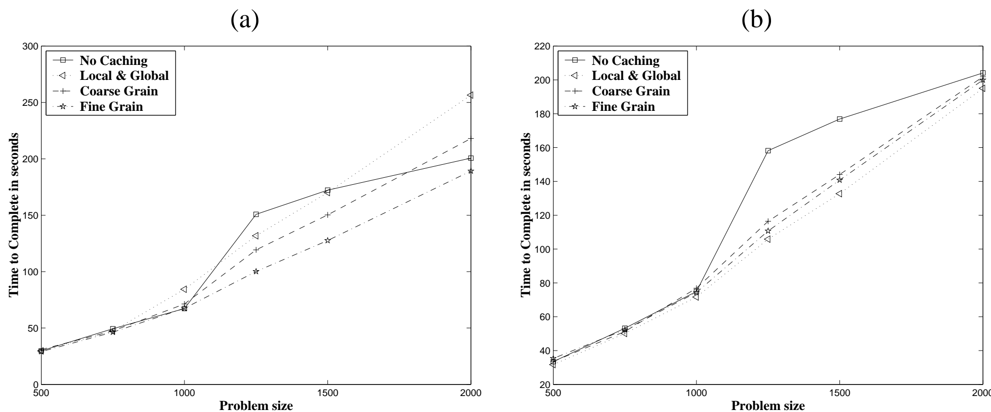


Figure 7: `vpenta`: impact of problem size (a) Global cache size is 20MB, (b) Global cache size is 200MB.

pronounced because its I/O traffic is lower.

In summary, we find that discretionary caching becomes very important when the problem sizes of applications get large enough, and the working sets cause more thrashing in the global cache. We find that a compiler based technique for modulating what to place/bypass in the global cache can alleviate some of these thrashing problems and help us reap the benefits of a global cache. Of the two different policies that we tried, we find that a finer granularity of control is a better option than file level control. This is because not all blocks within a file may have the same access pattern or access frequency.

5 Runtime Cache Bypass

So far, we have evaluated two compiler-based strategies (coarse-grain and fine-grain) where our compiler decided what to place in the global cache and when to bypass it. There are many cases where such a compiler-based strategy may not be desirable or even applicable. For example, when we do not have the source code of the application, we cannot analyze the program and determine its access pattern statically. Similarly, in some cases, the application code might be available but the access pattern it exhibits may not be amenable to compiler analysis (e.g., due to array-subscripted array references, non-affine subscript functions, or pointer arithmetic). However, in these and similar cases, it might be still possible to optimize the application using a runtime technique. A runtime technique tries to evaluate block access frequencies at runtime and makes cache bypassing decisions dynamically.

In this section, we investigate the effectiveness of a runtime strategy for managing global cache. Along similar lines, there has been prior work [18] in the context of processor data caches for runtime bypassing using access counters. However, in this study, we examine a much simpler strategy since there are some problems when implementing schemes such as (e.g., [18, 36]) on our platform where we have multiple levels of caches and a miss from the local cache may not at all go through the global cache. Our strategy is based on the idea of having counters with segments. Specifically, we

associate a counter with each segment that keeps the number of times the segment is accessed. These counters are called *segment counters*. When a block needs to be brought into global cache, its segment counter is compared with a pre-set *threshold value.* If the value of the segment counter is larger than the threshold, the block is placed into the global cache; otherwise, the cache is bypassed. When the local cache gets this block, it is told (either in the read response or the write acknowledgment) to avoid going through the global cache if it needs to be bypassed subsequently. The rationale behind this approach is that when a block is not accessed frequently enough, placing it into the global cache can cause a useful (i.e., more frequently used than the block in question) block to be discarded. It should be noted that we do not perform any checks when the block is accessed for the first time (counter reads zero), and only subsequently does this scheme kick in. When a new block is accessed, the harvester on the global cache examines all currently residing blocks to find a candidate for replacement whose counter is below the threshold (and does some aging of counters when doing so). Finally, in our current implementation, the decision for a block (whether to bypass or not) is made only once and we do not re-evaluate the choice once we decide to bypass the global cache for a block.

The results with this strategy are given in Figure 8 for a global cache size of 20 MB with two different threshold values — high (20) and low (3) for the same two applications examined earlier. We find that the runtime strategy improves the performance of global caching for both these extremes. The benefits are better at larger problem sizes where cache thrashing becomes more significant and we need to be careful on what to put in the global cache. This is also the reason why when we go to larger problem sizes, the more aggressive runtime approach (i.e., the one with the higher threshold value) does better than the one with the smaller threshold.

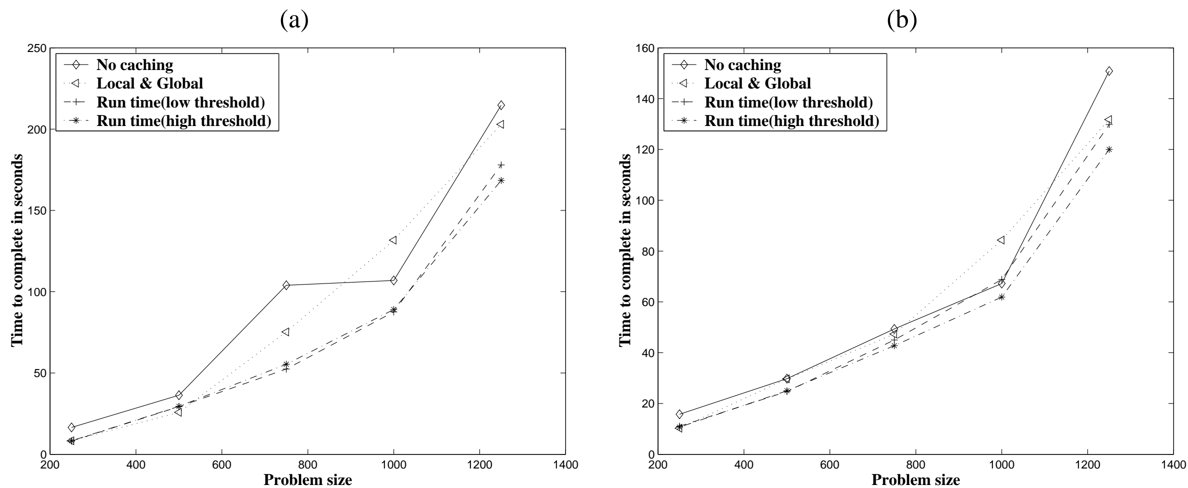


Figure 8: Runtime cache bypassing (global cache size is 20 MB) (a) `tomcatv`. (b) `vpenta`.

Next, we perform a sensitivity study of the runtime technique that depends on two tuneable parameters namely – threshold value and segment size. Figure 9(a) captures the performance of the runtime strategy as a function of the threshold value for `tomcatv`. We observe that typically threshold values in the range of 20-50 lead to better performance since they are more effective in weeding out what should not be put in the global cache, without defaulting to the `No Caching` strategy. Consequently, we use threshold values in this range in the next few experiments.

Recall that so far we have fixed segment size to be four blocks. To study the sensitivity of our runtime strategy to the segment size, we conducted another set of experiments where we used different segment sizes ranging from 2 blocks to 64 blocks. The results are illustrated in Figure 9(b) for `vpenta`. Note that each bar in these graphs is normalized to the 4 block segments. These results indicate that selecting a suitable segment size is important. In particular, working with very small or very large segment sizes may not be a good idea. When the segment size is very large, the blocks in a given segment do not exhibit uniform locality, therefore, a segment-wide decision might be the wrong (suboptimal) choice for many blocks in the segment. Similarly, if the segment size is very small, we witness an increased traffic through the global cache (which in turn hurts the performance). It should also be stressed that a small segment size means more bookkeeping and more runtime overhead. Similar results have been obtained with other applications as well and they are not explicitly given here.

Having examined both compiler (static) based and runtime optimizations for the same two applications, one could ask how the two compare in terms of effectiveness. We plot the local and global cache hit rates for different problem sizes for the same two applications under four different execution scenarios, (a) a scheme which blindly caches everything without any optimization for I/O, (b) a static compiler-driven scheme that caches file blocks at a coarse granularity (file level), (c)

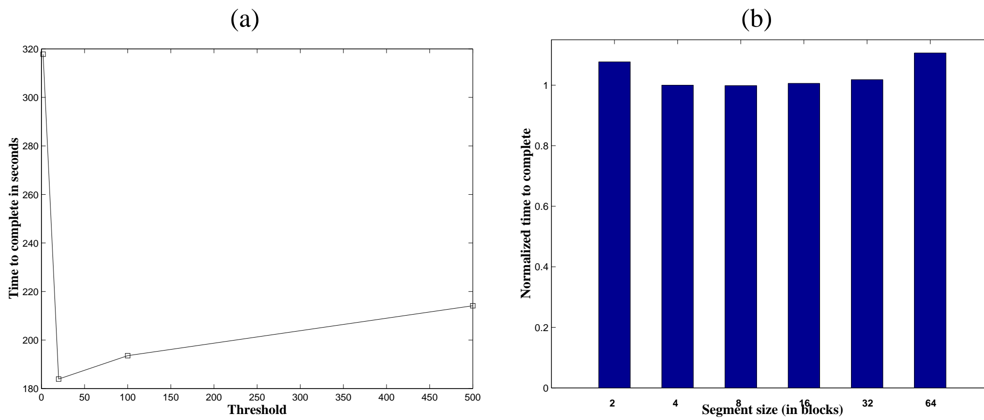
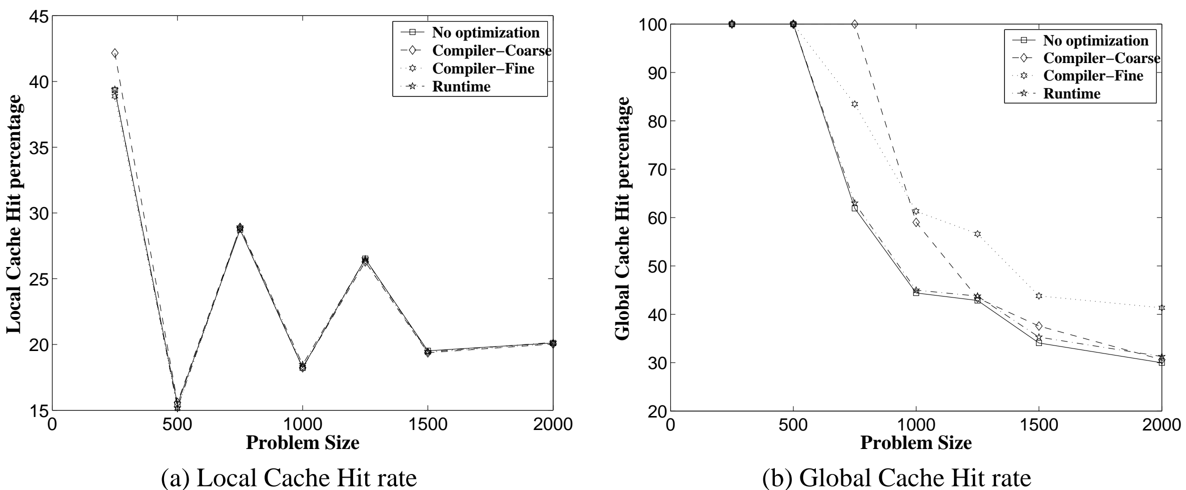


Figure 9: (a) Impact of the threshold value for `tomcatv`. (b) `vpenta:` Impact of segment size.

a static compiler driven scheme that caches file blocks at a finer granularity (block level), and (d) a scheme which makes cache bypassing decisions at runtime, and the results are given in Figures 10 and 11 where the hit rates in the two caches are given for `tomcatv` and `vpenta`. As is to be expected, in such applications where all the information can be statically gleaned, the compiler based techniques can be anticipated to perform better than their runtime counterpart, since the latter requires a warm-up period before it attempts bypassing. However, the benefits of the runtime approach will be felt more in non-analyzable applications, or those in which we do not have source codes to perform these optimizations. We illustrate this by studying the effectiveness of the runtime optimizations on a set of parallel I/O traces, where this option is the only choice.

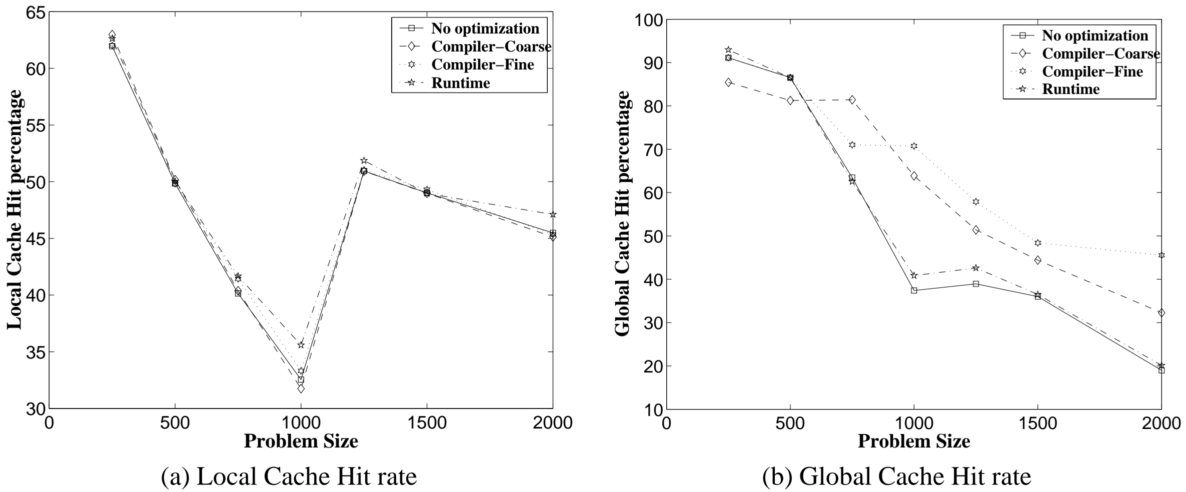


(a) Local Cache Hit rate (b) Global Cache Hit rate

Figure 10: `tomcatv`:Variation of Cache Hit rates with problem size

The traces used in this part of our experiments are from [37], which capture several diverse set of application executions (scientific and commercial). We evaluated the runtime strategy using the traces for the following six applications :

- `LU:` This application computes the dense LU decomposition of an out-of-core matrix. It performs I/O using synchronous read/write operations.
- `Cholesky:` This application computes Cholesky decomposition for sparse, symmetric positive-definite matrices. It stores the sparse matrix as panels. This application performs I/O using synchronous read/write operations.
- `Titan:` This is a parallel scientific database for remote-sensing data.
- `Mining:` This application tries to extract association rules from retail data.
- `Pgrep:` This application is a parallelization of agrep program from the University of Arizona.



(a) Local Cache Hit rate (b) Global Cache Hit rate

Figure 11: `vpenta`:Variation of Cache Hit rates with problem size

- `DB2:` This is a parallel RDBMS (Relational Database Management System) from IBM.

In the above experiment, we fixed the size of the local cache to 2MB, and the size of the global cache was fixed at 4MB and the threshold values were selected between 10 and 25. Figure 12 shows the execution time of the runtime optimized system normalized with respect to the system that uses local and global caching without runtime bypass. We can see that the optimized system benefits all but one of the six applications, with the benefits (reductions in execution times) ranging between 4% and 48%. The benefits are particularly significant in applications with poor locality (such as `DB2` and `LU`). These results reiterate the importance of managing/bypassing the global cache with an effective runtime strategy.

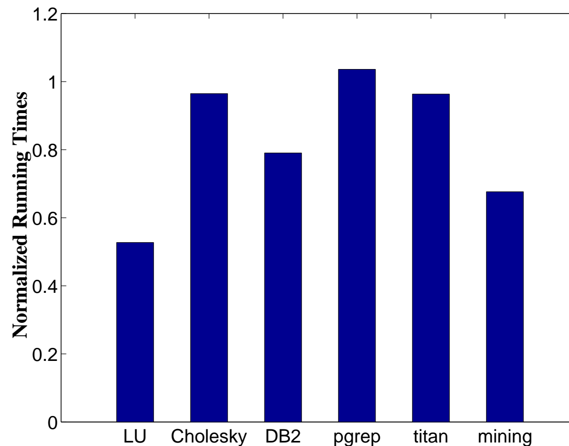
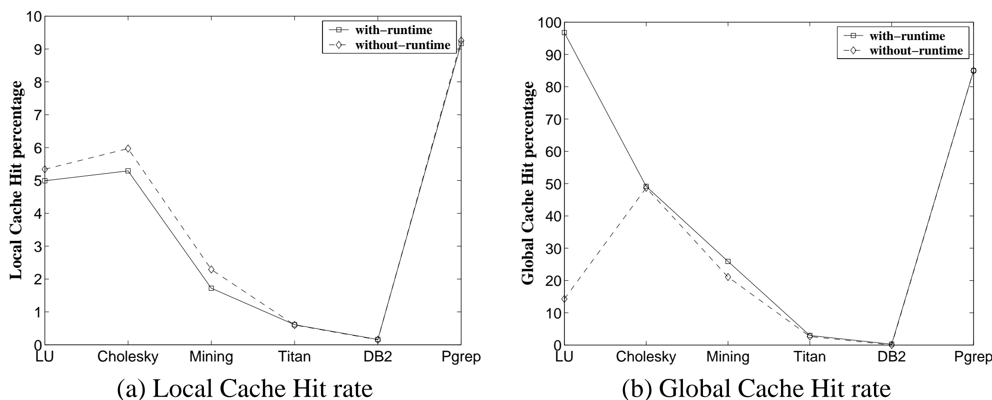


Figure 12: Benefits of runtime bypassing on application traces.

For the above experiment, the hit rates of the local and global caches are shown in Figure 13. As before, we don't see too much of variance in the local cache hit rates, and the performance improvement can be attributed to improved global cache hit rates with the runtime technique.

6 Concluding Remarks and Future Work

Caching for I/O is widely recognized as being critical for performance enhancements in large codes. Such caching is typically done at multiple levels — at the client nodes, at the server nodes, and perhaps even in between. Each has its advantages and drawbacks. This paper has shown that one should not indiscriminately cache all data at all levels of the caching hierarchy. We have demonstrated this by extending an off-the-shelf parallel file system for clusters, with a local cache at each node and a shared global cache. We have also provisioned mechanisms for bypassing each of these



(a) Local Cache Hit rate (b) Global Cache Hit rate

Figure 13: `Application traces`:Variation of Cache Hit rates

caches for a read/write operation at a fine granularity. One could use such mechanisms either explicitly by the application (perhaps some profile based tools could be useful here), or could be exploited by the compiler or the runtime system. In this paper, we have presented both compile-time and runtime based strategies to exploit global cache bypassing. Using both statically analyzable codes, as well as several public-domain I/O traces, coming from diverse domains, we have demonstrated the benefits of discretionary caching with these techniques. It should be noted that several of the previously proposed I/O optimizations such as prefetching, data striping/distribution, etc. can be used in conjunction with the ideas and discussions in this paper.

There are several interesting directions for future work. As mentioned previously, the scalability of the global cache with additional client nodes may turn out to be a problem and we are currently looking at scalable solutions to see if we can apply the techniques presented here at the I/O server nodes. We have only presented and evaluated a simple runtime strategy, and even that has turned out to be quite effective. We are currently exploring more sophisticated runtime schemes with this approach. We have used a shared nothing architecture for the experimental studies, and it would be interesting to study the applicability and benefits to systems with a shared storage architecture (perhaps including a storage area network). An important goal of our future optimizations is to be able to detect access patterns across different simultaneously running applications for I/O and cache optimizations. We are also interested in developing performance monitoring and profiling tools to better determine what, when, and where to cache data blocks. Finally, work is also underway in extending our compiler analysis to capture I/O access patterns inter-procedurally and applying more aggressive (global) optimizations.

7 Acknowledgments

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

References

[1] 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.

[2] 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.

[3] 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.

[4] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 31–44, Berkeley, CA, 2000.

[5] 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.

[6] Z. Chen, Y. Zhou, and K. Li. Eviction Based Cache Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[7] 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.

[8] 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 Hai Jin, Toni Cortes, and Rajkumar 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.

[9] 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.

[10] T. Cortes, S. Girona, and J. Labarta. Design Issues of a Cooperative Cache with no Coherence Problems. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 259–270. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[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] N. Stavrako et al. Symbolic Analysis in the PROMIS compiler. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, 1999.

[13] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems. In *Proceedings of the First International Conference on File and Storage Technologies (FAST)*, 2002.

[14] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[15] M. R. Haghighat and C. D. Polychronopoulos. Symbolic Analysis: A Basis for Parallelization, Optimization and Scheduling of Programs. In *1993 Workshop on Languages and Compilers for Parallel Computing*, pages 567–585, Portland, OR., 1993. Berlin: Springer Verlag.

[16] 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 Hai Jin, Toni Cortes, and Rajkumar 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.

[17] K. Hwang, H. Jin, and R. Ho. RAID-x: A New Distributed Disk Array for I/O-Centric Cluster Computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 279–287, Pittsburgh, PA, 2000. IEEE Computer Society Press.

[18] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-time Cache Bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, 1999.

[19] M. Kallahalla and P. J. Varman. Optimal Prefetching and Caching for Parallel I/O Systems. In *Proceedings of the Thirteenth Annual ACM symposium on Parallel algorithms and architectures*, pages 219–228. ACM Press, 2001.

[20] T. Kimbrel, P. Cao, E. Felten, A. Karlin, and K. Li. Integrating Parallel Prefetching and Caching. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 262–263. ACM Press, 1996.

[21] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

[22] T. M. Kroeger and D. E. Long. Predicting File-System Actions from Prior Events. In *Usenix Annual Technical Conference*, pages 319–328, 1996.

[23] 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.

[24] T. M. Madhyastha. *Automatic Classification of Input Output Access Patterns*. PhD thesis, UIUC, IL, 1997.

[25] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.

[26] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O library for out-of-core computations. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204. IEEE Computer Society Press, 1996.

[27] B. Nitzberg and V. Lo. Collective Buffering: Improving Parallel I/O Performance. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 148–157. IEEE Computer Society Press, 1997.

[28] 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.

[29] 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.

[30] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, 2002.

[31] 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.

[32] X. Shen and A. Choudhary. DPFS: A Distributed Parallel File System. In *Proceedings of the International Conference on Parallel Processing*, Spain, 2001.

[33] S. R. Soltis, T. M. Ruwart, G. M. Erickson, K. W. Preslan, and M. T. O'Keefe. The Global File System. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 10–15. IEEE Computer Society Press and John Wiley & Sons, 2001.

[34] 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.

[35] 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.

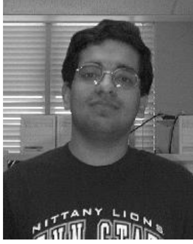
[36] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 93–103, 1995.

[37] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report CS-TR-3802, University of Maryland, College Park, MD, 1997.

[38] M. Vilayannur, M. Kandemir, and A. Sivasubramaniam. Kernel-level Caching for Optimizing I/O by Exploiting Inter-application Data Sharing. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[39] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, C. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[40] T. M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proceedings of the USENIX Annual Technical Conference*, pages 161–175, 2002.



Murali Vilayannur is a Ph.D student in the Department of Computer Science and Engineering at The Pennsylvania State University. His research interests are in High-Performance Parallel I/O, File Systems, Virtual Memory Algorithms and Operating Systems.

Anand Sivasubramaniam received his B.Tech in Computer Science from the Indian Institute of Technology, Madras, in 1989, and the M.S and Ph.D. degrees in Computer Science from the Georgia Institute of Technology in 1991 and 1995 respectively. He has been on the faculty at The Pennsylvania State University since Fall 1995 where he is currently an Associate Professor. Anand's research interests are in computer architecture, operating systems, performance evaluation, and applications for both high performance computer systems and embedded systems. Anand's research has been funded by NSF through several grants, including the CAREER award, and from industries including IBM, Microsoft and Unisys Corp. He has several publications in leading journals and conferences, and is on the editorial board of IEEE Transactions on Computers and IEEE Transactions on Parallel and Distributed Systems. He is a recipient of the 2002 IBM Faculty Award. Anand is a member of the IEEE, IEEE Computer Society, and ACM.

Mahmut Kandemir received the B.Sc. and M.Sc. degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the Ph.D. from Syracuse University, Syracuse, New York in electrical engineering and computer science, in 1999. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a member of the IEEE and the ACM.

Rajeev Thakur is a Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. He received a B.E. from the University of Bombay, India, in 1990, M.S. from Syracuse University in 1992, and Ph.D. from Syracuse University in 1995, all in computer engineering. His research interests are in the area of high-performance computing in general and high-performance networking and I/O in particular. He was a member of the MPI Forum and participated actively in the definition of the I/O part of the MPI-2 standard. He is the author of a widely used, portable implementation of MPI-IO, called ROMIO. He is also a co-author of the book ”Using MPI-2: Advanced Features of the Message Passing Interface” published by MIT Press.

Robert Ross received his Ph.D. in Computer Engineering from Clemson University in 2000. He is now an Assistant Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. His research interests are in message passing and storage systems for high performance computing environments. He is the primary author and lead developer for the Parallel Virtual File System (PVFS), a parallel file system for Linux clusters. Current projects include the ROMIO MPI-IO implementation, PVFS, PVFS2, and the MPICH2 implementation of the MPI message passing interface.