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

‡ Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{thakur, rross}@mcs.anl.gov

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. 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 33% savings over indiscriminately caching everything in some applications.

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. 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 cluster-based, parallel file system with an 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.

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 and rotational 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 [10] that we may need multiple levels of caching. For instance, in PPFS [10], 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. 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., [11]) 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 [1] — 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 differ-

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

ent 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 rest of this paper is organized as follows. Section 2 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 presented and evaluated in Sections 4 and 5 respectively. 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 [14, 13] and the references therein). Others have proposed parallel file systems and I/O runtime systems that provide users/programmers with easy-to-use APIs [2, 16, 4]. 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.

There has been a considerable amount of prior work on optimizing I/O and I/O caches [5, 17, 12, 15, 6], some of which has been on clusters as well. Maybe the most closely related work to ours are the approaches presented in three prior systems, namely, MPI-IO [8, 3], PVFS [1], and PPFS [10]. MPI-IO [8] 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 [18]. 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 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. PVFS [1] 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 [10] 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. Many of the ideas from PPFS for prefetching, distribution, and sharing can be used in conjunction with what is presented here.

3. System Architecture

Our system builds on the architecture of the previously proposed Parallel Virtual File System (PVFS) [1] 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.

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. The reader is referred to [1] for further details on the functioning of PVFS.

3.2. Overview of System Architecture

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. The design and implementation of the local cache at each node is described in an earlier work [21], and here we quickly go over it for completeness, and then concentrate on the global cache.

3.2.1 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. The details of our local cache implementation can be found in [21].

3.2.2 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 scope for detecting access patterns across datasets for optimizations, our current system uses a separate global cache for each file.

- *Should each application have its own global cache, or should we share a global cache across applications?* Since one of our goals is 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 [10] where the global cache is intended for optimizations within the processes of a single application.

- *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. Consequently, this architecture is also well suited to heterogeneous clusters where one or more nodes may have larger amounts of memory than the others.

3.2.3 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, 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). 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. More details can be found in [22].

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.

3.3. Performance of Primitives

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 these 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 is used in all our experiments.

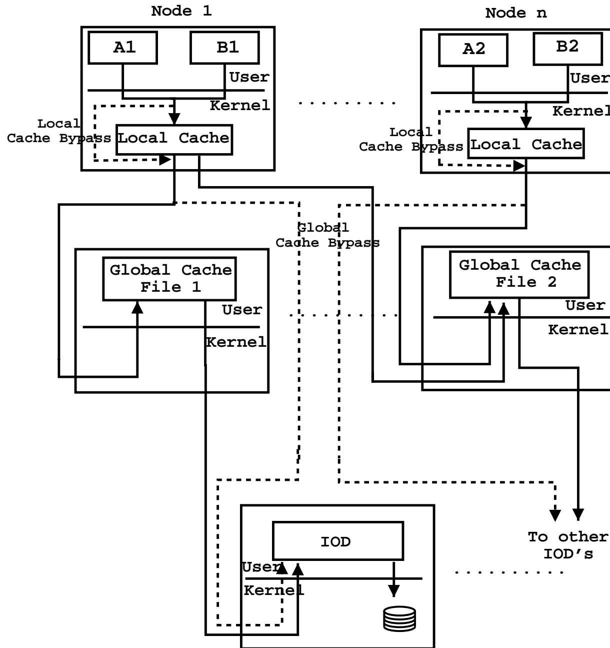

Figure 1. System architecture. Nodes 1..n are the clients where one or more application processes run, and have a local cache present. Upon a miss, requests are either directed to the global cache (one such entity for a file), or is sent directly to IOD node(s) containing the data in the disk(s).

3.3.1 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 original PVFS for all request sizes because of the overheads in looking up the local cache. This is also suggestive that 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

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

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

Table 2. Write times (msec) 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

the global cache can lower access times, provided the data is present there, compared to the original PVFS without caching in 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 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. 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.

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 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).

We have also conducted experiments with micro-benchmarks and their results can be found in [22]. These results also reiterate that it is important to not only cache at each node locally, but also at a global point, since this can cover some working sets larger than the local cache. However, such caching turns out to be a problem when there is poor locality, performing worse than not caching at all.

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 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 is 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 each 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 done 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. The rest of this paper explores the benefits of cache bypassing, and ways of initiating such bypass with the compiler and the runtime system.

4. Compiler-directed Cache Management

An optimizing compiler can help us identify what data should be brought into the global cache. It can achieve this using at least two strategies. 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 (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., [9, 7]) can be used here for this purpose. After doing such analysis, the compiler uses the global cache for reads/writes to the files 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.

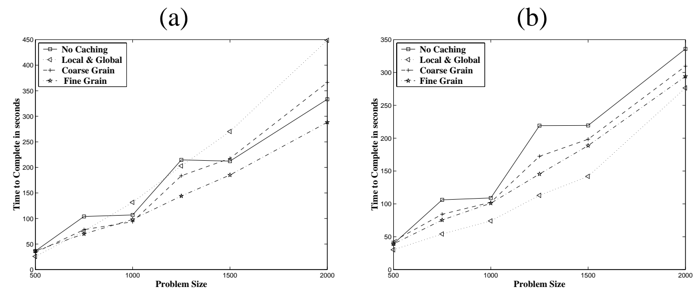

Figure 2. `tomcatv`: impact of problem size (a) Global cache is 20MB, (b) Global cache size is 200MB.

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

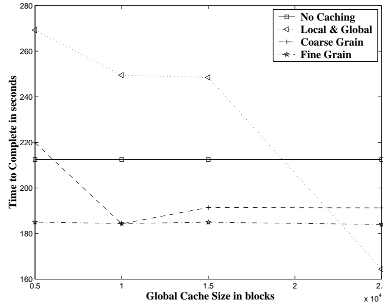

Figure 3. `tomcatv`**: impact of global cache size for a problem size of 1500.**

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 [9, 7] manipulation when loop trip counts are not known at compile time.

We implemented both these strategies using the SUIF compiler infrastructure [23] and evaluated them using codes where data access patterns are statically analyzable. 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 2 and 3 as a function of the problem size (local cache size of 400KB, global cache size of (a) 20 MB and (b) 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 manipulated in the application), respectively. 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.

Examining Figure 2(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 compiler optimizations at a fine granularity, 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 2(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 3. 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` are similar to many of those observed with `tomcatv` and are shown in [22].

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 patterns 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 [11] 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., [11, 19]) 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 4 for a global cache size of 20 MB with two different threshold values — high (20) and low (3). 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. We observed 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 used threshold values in this range in our experiments.

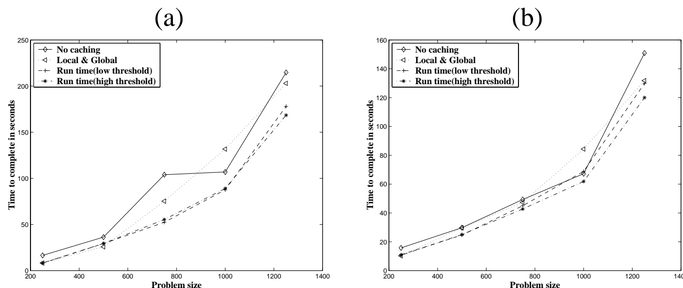

Figure 4. Runtime cache bypassing (global cache size is 20 MB) (a) `tomcatv`. (b) `vpenta`.

Another sensitivity parameter is the segment size. 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 witnessed 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. These trade-offs can be found in [22].

In the next set of experiments, we measured the impact of our runtime approach using a set of application traces, where such a runtime optimization is the only choice. The traces used in this part of our experiments are from [20], which capture several diverse set of application executions (scientific and commercial). We evaluated the runtime strategy using the traces for the following six applications (due to long run-times, only a part of the traces were executed):

- `LU`: This application computes the dense LU decomposition of an out-of-core matrix. It performs I/O using synchronous read/write operations. We executed the first 1000 out of approximately 8500 read/writes to demonstrate the benefits of discretionary caching.
- `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. We executed the complete trace (which contains around 400 read/write operations).
- `Titan`: This is a parallel scientific database for remote-sensing data. We executed all read/write operations in the trace.
- `Mining`: This application tries to extract association rules from retail data. We executed all read/write operations in the trace. The first 1000 out of approximately 6000 read and write operations were executed.
- `Pgrep`: This application is a parallelization of agrep program from the University of Arizona. We executed all read/write operations in the trace.
- `DB2`: This is a parallel RDBMS (Relational Database Management System) from IBM. The first 60000 read/write operations (out of nearly 500000 operations) were executed.

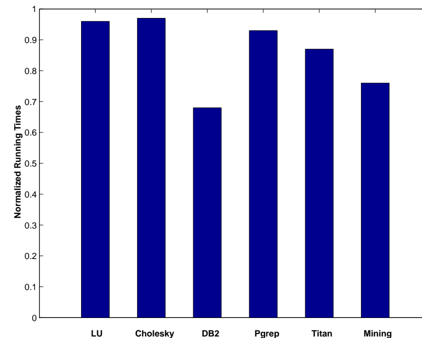

Figure 5. Benefits of runtime bypassing on application traces.

Figure 5 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 six applications, with the benefits (reductions in execution times) ranging between 4% and 33%. The benefits are particularly significant in applications with poor locality (such as `DB2` and `Mining`). These results reiterate the importance of managing/bypassing the global cache with an effective runtime strategy.

6. Concluding Remarks

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 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.

7. Acknowledgments

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] T. Cortes, S. Girona, and J. Labarta. Design Issues of a Cooperative Cache with no Coherence Problems. In H. Jin, T. Cortes, and R. 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.

[6] M. Dahlin, R. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, 1994.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In H. Jin, T. Cortes, and R. 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Discretionary Caching for I/O on Clusters. Technical Report CSE-02-018, Pennsylvania State University, PA, 2002.

[23] 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.