DAChe: Direct Access Cache System for Parallel I/O

Kenin Coloma Alok Choudhary Wei-keng Liao
Electrical and Computer Engineering Department
Northwestern University
{kcoloma, choudhar, wkliao}@ece.northwestern.edu

Lee Ward Sonja Tideman
Scalable Computing Systems Department
Sandia National Laboratories
{lee, stidema}@sandia.gov

Abstract

One of the largest challenges in client-side caching in extremely large-scale environments is consistency and coherency. By handling a user-space cache, we can offer applications much closer control over our client-side cache and scale the cache with the size of the compute resources (i.e. compute nodes). Cache data is shared among each compute node analagous to a traditional shared memory machine. Our approach to maintaining the integrity of the distributed cache turns out to be quite scalable and offers potentially sizable performance gains.

1. Introduction

Caching is a well known and proven technique for masking latencies and bottlenecks in the memory hierarchy. In the context of I/O, caching can alleviate some of the problems with the inherent mechanical limitations of disk drives and limited number of I/O servers. Client-side file caching in large-scale parallel environments is challenging from the maintenance perspective. Depending on end-goals and priorities, a balance must be struck between semantics and scalability. Relaxing coherency and consistency can greatly improve scalability and performance. By default, DAChe enforces cache coherency and supports an optional sequentially consistent atomic mode as well. DAChe is designed and developed with three primary characteristics; scalablility, coherency, and a passive architecture.

As with any cache system, the usefulness of DAChe is somewhat mitigated by the I/O characteristics of the applications run. Any application that repeatedly accesses parts of a file either with reads, writes, or some mix, stands to gain from caching.

In section 2 we describe the potential benefits and challenges of file caching in large scale systems. Then in section 3 we touch on related research before continuing on to briefly describe a key enabling technology, Remote Memory Access (RMA) in section 4. DAChe in section 5, its performance is analyzed in section 6. Finally in section 7, we summarize what DAChe accomplishes and some exciting directions for further research.

2. Client-side Caching Issues

The core of the cache coherency problem is ensuring globally accessible data is up-to-date when used. Two processes, *p0* and *p1*, may read and cache some shared piece of data. Subsequently, *p0* may write to the data. When the *p1* rereads the data, it will read directly from cache and not the new data written by *p0*. This general issue arises at many levels of the memory hierarchy, and is attacked from different angles depending sprecific features at each level.

If there were no cache, data would always be up-to-date, but caching generally increases performance. In file systems, caching can mask the latency of accessing a local or even remote disk by allowing for quicker reads and writes. Another optimization made possible by client-side caching is write-behind in which write data is buffered in the cache while computation continues. For local file systems, caching masks the poor latency and throughput in-

volved with mechanical disk access speed, and there are no coherency issues since only local processes use the system.

Distributed file systems use caching to hide the latency and bandwidth of the network. With caching in a distributed environment, there can be multiple cached copies of data, but normal usage in such an environment is such that files are not usually operated on concurrently. A distributed file system really just needs to be able to let one client at a time cache file data. A typical solution distributed file systems use to deal with cache coherency is periodic checks with a central server to find out if some cached file is needed by another process. Since checks are only periodic, cached data can sometimes be in an incoherent state. The frequency of the checks determines the probability that data is stale. More frequent checks will lower that probability, but increase overhead associated with the constant checking. It is not unusual for a distributed file system to relax semantics to allow for intermittent incoherence to avoid the overhead of keeping to a strict set of semantics. Typical single-user usage of files allows checks with the file server to be rather infrequent. Expiring leases can also be used to ensure only one client at a time has file data cached. A more significant issue in distributed environments is the coherency of directory caches.

In a parallel environment, there are often many more clients than there are I/O servers. Client-side caching can not only provide cached data quicker, but it also reduces the load on and contention for I/O server resources. Parallel environments are similar to distributed ones in that there are multiple clients accessing a single file system. In a parallel environment however, many processes often work on a single problem and concurrently access an output or input file. Allowing a single client at a time to access and cache a file is unreasonable. Either a multiple client-side caches must be carefully maintained, or there should be know caching at all. Maintaining a client-side cache in a parallel environment is far more challenging than doing it in a distributed one.

Whenever considering client-side caching, it is necessary to also consider what kind of semantics to follow. The strict nature of POSIX consistency semantics make them ill-suited for parallel and distributed computing environments. Rigid enforcement of POSIX consistency in these environments is usually at the expense of performance, either disallowing efficient caching or centralizing cache management. MPI semantics are by default slightly more relaxed. After write completion, data must be visible to all processes in the same communicator. Unless atomic mode is explicitly set, concurrent access to conflicting parts of a file are undefined. The problematic thing is that in atomic mode, an atomic access may be noncontiguous. If not for the last bit about atomic noncontiguous accesses, a POSIX compliant filesystem could provide all that MPI requires semantically.

Cache coherency provides fairly intuitive results. With sequential consistency so hard to provide at the library or file system level, this onus is better left to the application developer who is much more well equiped to handle ordering than any library could possibly be.

3. Related Work

Previously, Coloma *et al.* [3] worked with the collective I/O operations in MPI to partition files in various ways and assign I/O responsibilities for each region to one process across multiple I/O operations. By managing file access at the library level, file data cached individually by any process could be guaranteed the latest version. This solution does not actually implement a cache system, but uses the system client-side file cache.

IBM's General Parallel File System (GPFS) manages cache coherency with its distributed lock manager [1]. On extremely large scale systems, the overhead and coordination of the file system level lock manager make it quite a performance hinderance.

Panasas [8] maintains cache coherency through a callback system based on its metadata servers. While file data flow is directly between I/O servers and clients, a client must register its read/write intentions with the metadata servers so callbacks can be made should a region of file become dirty.

DAChe is quite similar to the work on active buffering done by Ma *et al.* [6], the primary differences being the lack of locally attached disks and thread support. DAChe is targeted at very large computers running light weight operating systems. In the process of trimming down such a specialized operating system, thread support is often left out. Both active buffering and DAChe have the effect of deferring writes, however active buffering uses threads to overlap I/O with application execution. Local disks in active buffering gives each process virtually unlimited cache space (relative to memory). Again, trends in large scale architectures make attaching a disk to each compute node infeasible.

4. Remote Memory Access

Remote Memory Access (RMA) interfaces provide what can be thought of as shared-memory emulation for a distributed memory environment. RMA in its truest form allows for the movement of data to or from a remote process without its active participation. The primary functions of any RMA interface mirrors shared-memory:

- Get
- Put
- Atomic test-set, atomic swap, or lock

In DACche, RMA is used to remotely access cache metadata and cache data. Two RMA interfaces were considered for use in DACche. One was MPI-2 RMA interface, and the other was the Portals interface.

The MPI-2 RMA interface provides two modes. Active one-sided communication requires the use of collective fence functions to ensure communications are complete. The collective nature of this mode rules out its use in DACche. Because read and write operations are independent, the only places to call the fence functions are `dache_open` and `dache_close`. Between open and close, there is no way to be sure any one-sided communication has completed. The second mode MPI-2 provides is passive one-sided communication. In this mode, remote memory can be remotely locked by another process in the communicator. In MPICH2, both modes are thread-based for portability, and at the time of development, the passive mode was not yet complete.

Portals is an open source message-passing library described further by Brightwell *et al* in [2]. While primarily intended as a low-level library foundation for an MPI implementation, file systems and other subsystems are allowed to hook directly in. Portals is built on a one-sided communication model. The implication of this for MPI-2 is its RMA interface should be easily developed on top of Portals. Though the Portals v3.3 API specifically calls for an atomic swap function, only Portals v3.0 has been implemented at the time of development.

DACche currently uses Portals for RMA. While portability is definitely a factor, the Portals interface is more mature and exists on specific large-scale platforms of interest. Besides which, the platforms of interest do not provide a thread-safe environment, making the present implementation of MPICH2 infeasible.

5. DACche design

From a design perspective, DACche can be architecturally divided into 3 primary subsystems: cache metadata, locking, and cache management. Its modular implementation makes it quite easy to port and experiment with given that the basic RMA requirements are met. One over-arching theme always under consideration during the design of DACche is to keep all aspects of cache management as decentralized as possible. A secondary theme is minimization communication where possible. The main tenet of DACche is that only a single copy of any file data can reside in any file cache. This single-copy rule ensures cache coherency and removes the task of maintaining state for replicated data. Another way to think of it is that there is never more than one usable copy of any given file page. The use of RMA keeps I/O operations in DACche passive, but requires that `dache_open` and `dache_close` be collective in order to set things up and break things down safely. After `dache_open` completes, all communication is one-sided unless it is with a mutex server described in more detail in the Mutual Exclusion 5.2 subsection.

The three subsystems in DACche warrant a closer look to understand how DACche works. Figure 1 illustrates the basic interactions between these subsystems.

5.1. Cache Metadata

Cache metadata maintains basic state for each page in the file. Most importantly, this metadata provides the whereabouts of any given file page. File page refers to the logical partitioning of the entire file into blocks of a size matching the page size of the cache. If a page is cached on any process the metadata reflects the caching process as well as an index location into that process's cache. cache metadata is remotely accessible through RMA and distributed across the application nodes in a deterministic fashion; in this case a basic striping algorithm. By striping the metadata array, or table as it will be called, across nodes deterministically, not only is a potential bottleneck avoided, but there is also no communication required to find the metadata associated with any file page.

Creating metadata for each logical file page brings up the issue of metadata allocation. This allocation process requires explicit coordination amongst the application processes, and this is only available at the collective `dache_open` and `dache_close` functions. Ideally, one would want the size of the metadata table to be directly related to the size of the file. What this basically entails is some level of cooperation among processes for growing or shrinking the table size during run time. Since the write operations are independent, however, there is no opportunity to coordinate all the processes in order to modify the size of the table. Without this coordination opportunity, the last resort would be the ability to remotely allocate globally accessible memory. Needless to say, remote memory allocation brings its own set of challenges. For all the above reasons, the default maximum file size is assumed to be 2 GB, with this value being settable by the user.

Given the distribution and passive nature of cache metadata, one crucial element is enforcing mutually exclusive access to it.

5.2. Mutual Exclusion

The purpose of the mutual exclusion subsystem is to ensure safe access to read and modify cache metadata. Ideally, mutual exclusion is directly supported in the RMA interface. With RMA support for mutual exclusion, the lock-

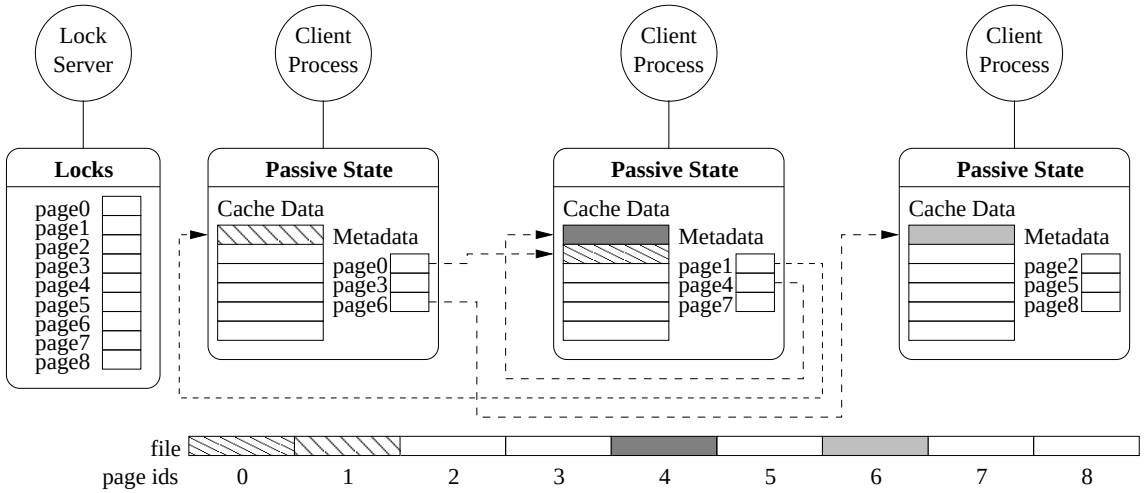

Figure 1. DACche architecture with passive metadata and cache servers on each client. Metadata is striped across clients, but a file page can be cached on any client.

ing subsystem can be also distributed across the application nodes as passive remote accessible state.

MPI-2 explicitly provides the `MPI_Win_Lock` and corresponding unlock functions. Gropp *et al* [5] describe how to use `MPI_Win_Lock` for traditional locking. Since the current test platform for DACche is a threadless environment, however, this rules out the use of the MPICH2's RMA interface. MPICH2 is focused on portability, and threads are more commonly available than hardware supported one-sided communication. At the same time, the atomic swap operation in the Portals library has yet to be implemented.

In the absence of an RMA solution to mutual exclusion, one or more processes from the allocated user processes are siphoned off from the main group to act as mutex servers. They are spun off during `dache_open` and returned during `dache_close`. During this time, the mutex servers cannot execute any user application code. So while passive RMA mutual exclusion is preferred, DACche can still be evaluated using dedicated mutex servers. Later, these mutex servers will become passive elements on the application processes.

Since mutex responsibilities will eventually be moved to the client, the mutex servers are intentionally kept quite simple. Lock responsibility for each file page is spread across the mutex servers in the same way cache metadata is spread across application processes. The mutex servers service locks in the order they come, queueing requests to the same cache metadata. Polling and simple queueing are both implementable when the mutex servers become passive state on remote processes. A process must block until its lock request is fulfilled by the mutex server.

Typically, cache metadata is not "held" for extended periods of time. It is locked only briefly for modification. It is not held for the duration of access to the actual cached data. Minimizing the time that the metadata is locked, should reduce the amount of simultaneous lock requests to the same metadata. The exception to short lock times is when a particular file page is being brought into the cache or a cache page is being evicted. In either case, the cache metadata must be locked for entire I/O phase in order to prevent early or late cache accesses, respectively.

5.3. Caching

Pages cached on one process are globally accessible to any other process through RMA. Although access to metadata is carefully mediated, remote access to cache data is basically a free-for-all. Since any file page can be cached in at most one cache, all accesses to that page are coherent.

Cache management and eviction is handled locally with one exception to be discussed a little further along. What data to cache is determined by the local procces's I/O accesses. If a process accesses a file page that is not yet cached on any of the other processes, it caches it itself and updates the corresponding cache metadata to reflect this change. Should a process run out of cache space locally, it must evict a page based on some local policy such as a least recently used (LRU) policy. Remember that during eviction, a page's metadata cannot be accessed at all. Another precaution alluded to earlier is that processes accessing a remotely cached page must "pin" the page in cache so the page cannot be evicted while being accessed. This pin is a semaphore contained in the cache metadata along with location information. While a page is pinned, the process on which the

cache page resides cannot evict the page, and must either wait until the page is "un-pinned" or try to evict a different page. New data is not written to disk until it is either evicted or written out at *dache_close*.

6. Performance Implications

DACheis evaluated using a synthetic I/O access pattern that should, by design, benefit from cached data. It is roughly based on the regular noncontiguous access patterns often found in scientific applications and is meant to test the performance of DACheThe benefits a real application may derive from DACheare also clearly of interest. The basic labeling convention is as follows:

- mtx-n where n is the number of mutex servers
- DACheclients is the number of clients actually caching data (non-mutex servers)
- 50:50 refers to an equal mix of clients and mutex servers

6.1. Machine Configuration

All of our tests were run on ASCI Cplant [4] at Sandia National Laboratory. Cplant is an Alpha Linux cluster with each compute node configured with one 600 MHz Alpha EV-6, 512(Ross) MB of RAM, no disk, and a 64-bit Myrinet card. Each compute node runs Red Hat 6.x with kernel 2.4.x.

6.2. Sliding Window Benchmark

The sliding window application uses a repetitive I/O access pattern, and the underlying caching library uses the lock service to gain exclusive access to cache page metadata. Figure 2 describes the access pattern of the synthetic benchmark. Each process accesses a contiguous chunk of data. In each subsequent iteration, processes circular shift their accesses until all chunks are read before sliding to the next set of four chunks. Since there is one lock per page, and the benchmark accesses are page-aligned, the number of meta data locks is tied directly to the number I/O operations. The cyclic access pattern makes the amount filesystem calls increase along a second order function rather than linearly, as seen in figure 3. Direct filesystem calls that the sliding window application follows the the "direct" curve, while the number of filesystem calls as mediated by DACheflows the "DACHe" curve. This reduction is achieved transparently from the application point of view. The same could be achieved using explicit communication in application, but increases application complexity.

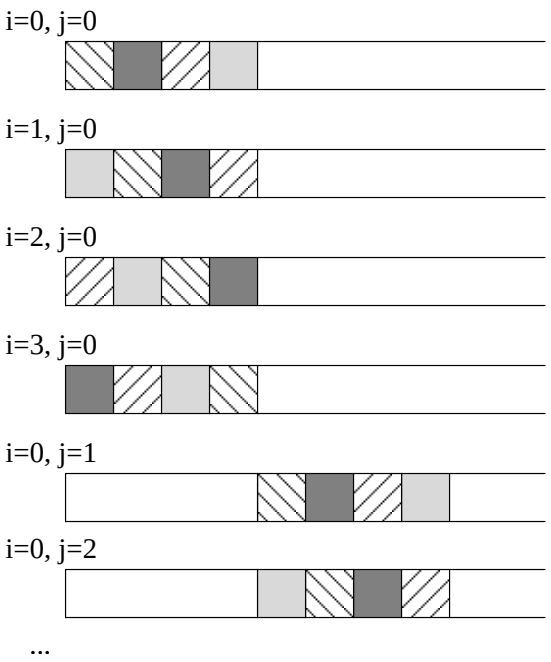

Figure 2. Several iterations of the sliding window I/O pattern for 4 processes.

Cache Size/proc	4 MB
Page Size	32 KB
Chunk Size	64 KB
loops	40

Table 1. DACheand sliding window parameters

It is important to note that since the DACheSystem is the primary subject of evaluation. As such, actual I/O is removed from DACheto prevent interference from any specific filesystem. The sliding window access pattern is such that once a file page is brought into cache, it remains there throughout the execution and is only remotely accessed over the network. Though actual I/O would have been performed to bring in the file data, all subsequent accesses really access cached data, possibly on remote nodes. Throughput is calculated based on the number of I/O calls that would have been made to the file system. Since a fixed number of lock requests are generated per cache page access, the number of locks requested by each process is directly proportional to the amount of I/O done.

Since scalability is the primary goal of the DACHe, it was tested with various numbers of mutex servers running. In-

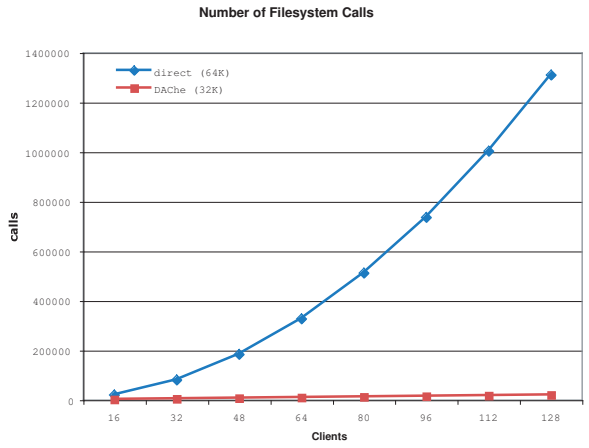

Figure 3. The amount of I/O grows as a second order function on the number of clients.

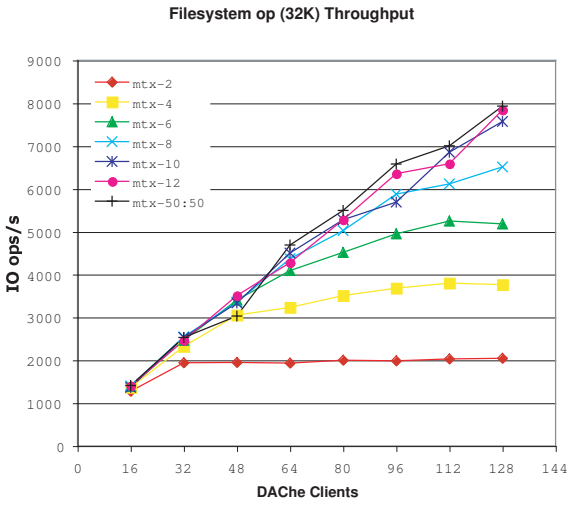

Figure 4. The number of clients at which throughput knees over is dependent on how many lock servers there are.

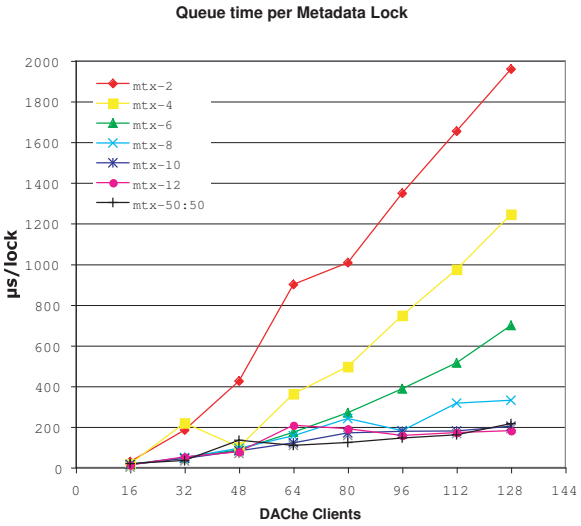

Figure 5. Since the number of locks grows as a second order function, increasing the number of mutex servers with the number of clients should yield a linear increase in Queue time.

tuitively, the heavier the lock system is taxed, the more mutex servers are needed to accomodate the increased load. This is illustrated clearly in figure 4 where a larger number of mutex servers allows a larger number of clients without severely hindering performance when more than the minimum number of mutex servers is used. From a cost efficiency perspective, one would like to stay on the outside curve using the minimum number of processes at each point. This client to mutex server ratio is highly dependent on the properties of the specific machine. A 50:50 mix where there are an equal number of clients and mutex servers allows the mutex service to scale with the application size. As expected from the growth rate of filesystem calls, this outer bandwidth curve is roughly a square root function. The usefulness of this will become more apparent in section 7. The most important point from Figure 4 is the number of mutex servers determines at which number of clients performance will plateau.

Another demonstration of the lock system's scalability can be extracted from Figure 5 which describes the queue time for a lock. An obvious bottleneck, increasing the number of mutex servers with the number of clients decreases the average queue time for each lock request. Increasing the number of lock servers with the number of clients keeps queue times reasonable. Because the problem size increases at second order rate, the 50:50 mix ratio of clients to servers

Abbreviation	Function
Meta Lock	Obtain access to cache metadata
Meta Unlock	Release access to cache metadata
Eof	(Un)Lock the End-of-file for writing
Get Meta	RMA retrieving cache metadata
Put Meta	RMA writing cache metadata
Rmt xfr	Data transfers to/from remote caches
Application	Application + silent functions

Table 2. DACHe and sliding window parameters

yields the expected linear increase in queue times. The client service time includes network latency and bandwidth as well any lock contention generated by the sliding window application itself.

Figure 6 provides a breakdown of how much time is spent executing a subset of functions in DACHe. Table 2 provides a brief explanation of each function included in the timing figure. Indicated by the quickly rising meta lock times, two mutex servers is obviously not sufficient beyond 16 processes. While the proportion of time spent on locking increases as the number of processes gradually increases, it can be explained by the second order increase in locks generated by the sliding window application. Since the other functions in DACHe utilize a fairly constant proportion of the run time, and all include one-sided communication, the network is not being saturated. It is worth noting that since the sliding window application only tests I/O, its non-io related computation is extremely limited. The dramatic difference in relative communication costs are indicative of the need to scale mutex services with the computation size.

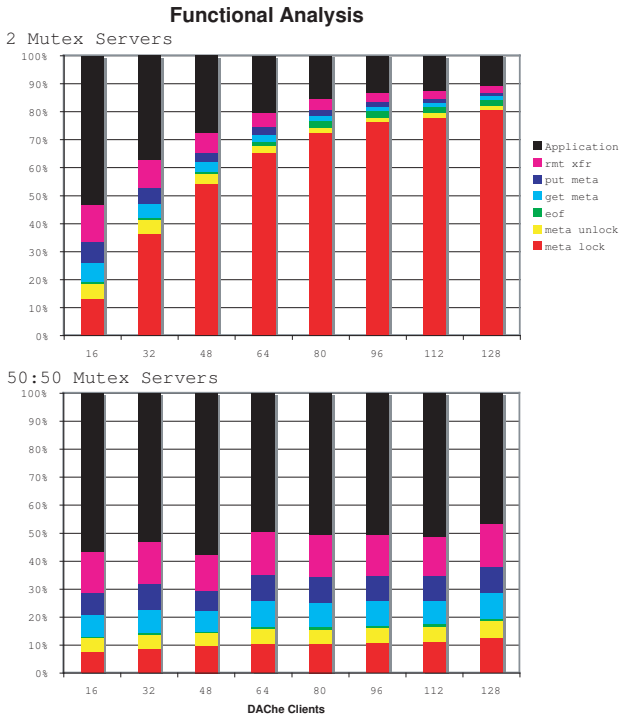

Figure 6. Relative amounts of time spent on individual DACHe functions for 2 mutex servers and 50:50 mutex servers.

7. Conclusions and Future Directions

The most glaring problem with the current implementation of DACHe is the use of separate processes as mutex servers. A fully implemented RMA system which provides one of the necessary atomic operations should remedy this. RMA allows the lock responsibilities of the lock servers to be taken up by client nodes, eliminating the extra cost of seperate lock processes and leveraging the performance characteristics of single-sided communication, see Figure 7. The algorithms involved have been explored chiefly by Mellor-Crummey *et al.* [7] in the context of shared memory machines. Another reason for the move to a passive RMA solution is the architectural trend of parallel computers towards running stripped down threadless operating systems on compute nodes. A drawback of moving to a completely passive lock system is implementing distributed queues in

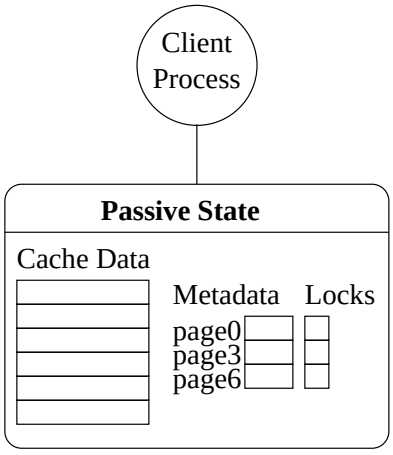

Figure 7. With proper support in the RMA interface, it will be possible to implement locks passively on the clients.

an efficient manner. A distributed waiting queue will result in additional communication costs, but these cost will hopefully be more than compensated for by performance enhancemence elsewhere in the system. This passive goal is the reason behind the lock system's straight-forward design. A passive environment prevents any computation on the server side and limits the server to remote accessable state.

The least recently used cache eviction policy is commonly accepted as the best general purpose policy. In the current implementation, the LRU queue is stored locally, and is only affected by local accesses to cache pages. Not only should the basic eviction performance of DAChe be evaluated, but further research is planned in how taking remote accesses into account in the eviction policy may affect overall cache performance. A further extension would be to migrate cache pages frequently accessed by a particular process to that node for dynamic load rebalancing. Adaptive applications may intermittently redistribute working sets and domains, causing substantial shifts in the I/O access patterns of individual processes.

Tighter integration with MPI is also planned with support for MPI-2 RMA. By moving away from the Portals interface, DAChe will benefit from the portability of MPICH2.

Preliminary performance results for DAChe suggest scales well. The extremely distributed characteristics of DAChe get around a number of potential bottlenecks. The most interesting feature of DAChe is its efficient use of the increasingly common one-sided communication architecture.

Acknowledgements

This work was supported in part by Sandia National Laboratories and DOE under Contract 28264, DOE's SCiDAC program (Scientific Data Management Center), award number DE-FC02-01ER25485, NSF's NGS program under grant CNS-0406341, and NSF/DARPA ST-HEC program under grant CCF-0444405.

References

[1] An introduction to GPFS 1.2. `http://www-1.ibm.com/servers/eserver/clusters/software/gpfs.html`, December 1998.

[2] R. Brightwell, B. Lawry, A. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication.

[3] K. Coloma, A. Choudhary, W. Liao, L. Ward, E. Russell, and N. Pundit. Scalable high-level caching for parallel i/o. In *Proceedings of the 2004 IEEE International Parallel and Distributed Processing Symposium*, April 2004.

[4] CPLANT: A commodity-based, large-scale computing resource. `http://www.cs.sandia.gov/cplant`.

[5] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[6] X. Ma, M. Winslett, J. Lee, and S. Yu. Faster collective output through active buffering. In *Proceedings of the 2002 IEEE International Parallel and Distributed Processing Symposium*, April 2002.

[7] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report TR 342.

[8] D. Nagle, D. Serenyi, and A. Matthews. The panasas activeScale storage cluster - delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Supercomputing Conferencence*, November 2004.