Performance Fault Masking for Network-Attached Storage Devices

Omer Zaki and Brian Forney
University of Wisconsin-Madison
Computer Sciences Department
1210 W. Dayton St.
Madison, WI 53705
{ozaki, bforney}@cs.wisc.edu

Abstract

Distributed storage systems increase performance of storage subsystems yet introduce performance variability which can create bottlenecks in a storage system. This paper introduces performance fault masking for network-attached storage devices to decrease performance variations in parallel storage systems. Performance fault masking key components are a global cache and a cost-aware caching policy. Trace driven simulations of synthetic workloads have shown promise with this technique. Performance variations are reduced whereas average request times increase for most workloads.

1. Introduction

Distributed storage systems have lessened the gap between processing power and storage bandwidth. Systems such as RAID [Patterson88] have substantially increased the available parallelism. However, single disk performance has not increased at the rate of processor performance. Further, applications that distribute their workload across a distributed storage system, especially striped distributed storage systems, suffer from worst case performance of the slowest storage component. Single disk performance continues to be a limiting factor.

A slow single disk is an example of a performance variation. Performance variations in distributed storage systems can occur for various reasons. First, distributed storage systems can be heterogeneous. As a system grows, components may be added that have differing performance characteristics. Second, modern disks have variable zone densities. Transfer rates are higher from outer tracks of a modern drive than the inner tracks. Third, storage protocols like SCSI dynamically and transparently remap bad blocks. This leads to a logical decrease in track density, lowering performance of tracks with remapped blocks. Any solution to performance variations must consider these reasons.

As a solution, we propose an approach called *performance fault masking*. Performance fault masking for network-attached storage devices (NASD), or network-attached secure disks [Gibson97] uses caching to hide performance faults of the storage system from clients. Unlike traditional caching which works to provide data at a faster rate, assuming a constant retrieval cost from lower levels, performance fault masking assesses differing costs of data retrieval. Data retrieval costs are used to adapt the performance fault masking cache policy to favor those NASDs which have higher retrieval costs.

This paper is organized as follows. Section 2 discusses related work to performance fault masking including shared caching, adaptive storage, and cost-aware caching. The philosophy of and an algorithm for performance fault masking is presented in section 3. Section 4 discusses the simulator used to evaluate the proposed algorithm. Section 5 presents experimental results for three benchmarks. The paper finishes with future work and conclusions in Sections 6 and 7.

2. Related work

2.1 Shared caching

Several projects have studied the use of a shared cache between components of a cluster. The xFS filesystem [Anderson96] included a cooperative cache of file system blocks [Dahlin95]. Clients in the system maintain a logical shared cache. Blocks are given several chances in each client's portion of the shared cache before they are removed from the global cache. The Global Memory Service (GMS) project [Feeley95] at the University of Washington distributed virtual memory across a cluster of systems. Each system contains a global and local page cache with flexible allocation between each cache.

The cooperative caching in xFS and GMS differ from performance fault masking in two ways. First, they do not have a cost model for cache policies. xFS and GMS assume the cost of acquiring data is constant regardless of the source of the data. Second, they are concerned with clusters of clients and not clusters of storage devices.

2.2 Adaptive storage

Performance fault masking adapts to changes in the distributed storage system's performance characteristics. Two other systems have used adaptation in storage systems. The River system [Arpaci-Dusseau99] adapts the storage system workload according to performance availability of components. Adaptation occurs through the use of a distributed queue and graduated declustering interposed between clients and the storage system.

The River system differs from performance fault masking in several ways. River uses mirrored disks to provide a basis for adaptation. Performance fault masking uses caches to adapt the storage system. River has a data-flow programming environment, whereas performance fault masking uses a NASD storage interface at the operating system level.

The AutoRAID system [Wilkes95] adapts to workload changes by migrating data between mirrored and RAID 5 storage. The most recently used data is stored in mirrored disks for performance while older data resides in RAID 5. Performance fault masking uses copies of data that are temporarily stored in caches. Performance fault masking adapts to performance faults and, as a side effect, workload changes, while AutoRAID only adapts to workload changes.

2.3 Cost-aware caching

Cost-aware caching algorithms have been explored in web caches. The GreedyDual-Size algorithm [Cao97] uses web object size and retrieval cost in its cache replacement policy. When implemented in a web cache, this algorithm decreases download latency and reduces network traffic between the web cache and web servers. Performance fault masking uses the cost of retrieval and least recently referenced information to determine which pages to keep in its cache.

3. Performance fault masking

As previously stated, distributed storage systems can exhibit worst case behavior due to performance faults in components. Performance fault masking's goal is to hide performance faults from clients. Masking of performance problems occur through the use of a global cache and a cost-aware caching policy. The global cache allows NASDs to logically grow their cache as performance faults and workload changes occur. Cost-aware caching accounts for the difference in operation cost between NASDs. The remainder of this section describes the mechanisms and policies used for performance fault masking.

3.1 The global cache

All of the NASDs in the storage system cooperatively maintain a global cache. The global cache is logical in that it is distributed among the cooperating NASDs. Performance fault masking's global cache has similarities to Cooperative Caching [Dahlin94] and GMS [Feeley95]. Each NASD's physical cache has a logical local cache and, if needed, a portion of the logical global cache. The NASDs flexibly allocate their physical cache to the local and global cache as the system responds to changes in performance. Consequently, the two caches compete for space in each NASD's physical cache.

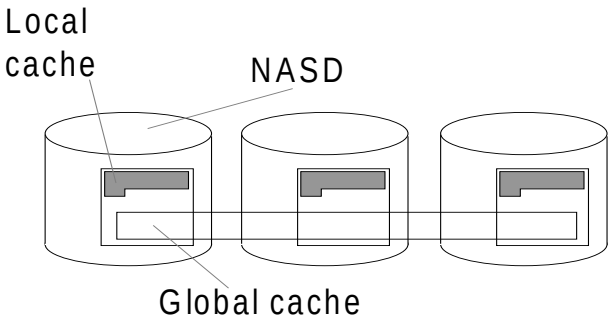

figure 1.

Figure 1 shows three NASDs each with a physical cache. The global cache is distributed among the physical caches. Each of the NASDs has a local cache in the figure. The NASD may not have a local cache if the global cache occupies the entire physical cache. Conversely, the local cache may occupy the full physical cache.

The global cache introduces the idea of data ownership. Each NASD *owns* data that it holds on stable storage. Copies of owned blocks may be in the global cache. Each NASD must maintain metadata to find the owned blocks since the global cache is distributed. The *remote block table* contains this metadata. Metadata is maintained as a tuple of <<object id, object offset>, caching NASD id>. The *caching NASD* id is the identifier for the NASD holding the block for the global cache. The remote block table is maintained on and by each owner NASD.

The remote block table is accessed for operations on blocks cached globally. To perform an operation on a globally cached block, the owner first looks up the block in the remote block table using the object id and offset as a synomyn for a block. If the block is found in the remote block table, the owner will then send a request to the remote system to perform the operation. If a client of the storage system makes a request of an owner NASD for a block cached globally, the request if *forwarded* to the remote NASD.

When a NASD evicts a block from its local cache, it may place the block in the global cache. The block is written to disk first, if dirty, and then sent to the global cache. The caching NASD will accept the block for the global cache, adding the block to global cache residing in its physical cache.

Just as blocks may be evicted from a local cache, the global cache can evict blocks. Upon eviction from the global cache, the block is discarded. The block could be passed around as in N-chance forwarding [Dahlin94], but for simplicity the block is discarded. Once the block is discarded the remote block table is inconsistent. Two solutions exist: explicitly update the remote block table at the owner or treat the remote block table as a set of hints. The first solution requires sending a message each time a block eviction occurs. It also must handle a race condition. The race condition is as follows:

1. The owner forwards a request for the evicted block
2. The caching NASD evicts a block and sends an eviction notice
3. The caching NASD receives the forwarded request
4. The owner receives the eviction notice

The caching NASD cannot satisfy the forwarded request during the race condition. A solution is to return the forwarded request to the owner and service the request at the owner. However, this solution requires two messages which adds complexity.

The second solution for the remote block table inconsistency allows the remote block table to become inconsistent and then handle the fallout. The remote block table then becomes a set of hints as to where blocks can be found. When a request is forwarded, the remote NASD services the request if the requested block can be found. If not, the forwarded request is returned to the owner. The owner then services the request and updates its remote block cache.

Because of simplicity, the second solution was used in our experiments. A future study could be performed; one solution likely better fits the usage of the global cache.

3.2 Cost computation

Each NASD periodically computes a *perceived bandwidth* and distributes it to all of the cooperating NASDs. The perceived bandwidth is the primary information used in the cache eviction policy. For each request, the total service time and bytes requested are added to running totals. When the period has expired, the NASD computes its perceived bandwidth by dividing the two running totals. The perceived bandwidth then does not directly account for the time required to retrieve data from the global cache - only the time required to process the request is accounted on the owner. Our current system uses a perceived bandwidth period of 100 milliseconds.

Several modifications were made to a straightforward implementation of the perceived bandwidth computation. The modifications handle changes in workload that may be dramatic. These are discussed next.

NASDs may be idle during a perceived bandwidth period and compute a perceived bandwidth of infinity. Thus using this system may appear cheap, so other NASDs may cache blocks in the global cache on the idle NASD. This may be an unfortunate decision. The idle NASD may become busy in the next period and become very slow if the global cache displaces the local cache contents during the idle period. One solution is to throttle the change when a system becomes idle. Our implementation doubles the perceived bandwidth from the previous period.

For non-idle NASDs, the new perceived bandwidth is a weighted average of the straightforward computation and the previous perceived bandwidth. A weighted average smoothes dramatic changes in the perceived bandwidth. The current implementation uses weights of 80% new and 20% previous:

New smoothed perceived bandwidth =
*0.8 * new straightforward perceived bandwidth +*
*0.2 * previous perceived bandwidth*

The weights were determined by a few empirical trials; further study of the weights could be performed. Removing dramatic changes in the perceived bandwidth is believed to help the performance fault algorithm.

The perceived bandwidth contains a rough measure of the throughput of each NASD. If the throughput of a NASD is higher, the cost of requesting data often will be lower. However, throughput and latency are orthogonal. A highly contended NASD will have a higher request latency while the throughput remains the same due to queuing effects. Since many clients of storage systems amortize the cost of latency by batching request and caching, latency is less important and a throughput measure is sufficient for most clients.

3.3 Cache policies

The cache policies can be divided into two parts: evicting a block or blocks and the destination of the evicted blocks. The eviction policy is impacted by the non-uniform request cost in the system. The use of a global cache affects the placement policy for evicted blocks.

The eviction policy has two pieces of information to choose blocks to evict. The perceived bandwidth of each NASD is the primary information used. Using the perceived bandwidth as primary information, allows the storage system to adapt to performance faults and workload changes. Secondary information comes from the time the block was last accessed. Generally, past behavior can be used as a fairly good future predictor. However, this is suboptimal when streaming through data sets larger than the cache. A technique like EELRU[Smaragdakis99] may help.

To choose a block to evict, the cache uses a priority based scheme. The policy will remove entries from a NASD's physical cache that are for owners faster than the NASD. The physical cache finds the fastest owner in the distributed storage system with one or more entries in the NASD's physical cache using the perceived bandwidth information. Entries for that owner will be evicted. If the new blocks will still not fit in the cache, the next fastest owner with entries in the physical cache will be evicted. This process continues until either a sufficient number of blocks have been evicted or all of the owners faster than the NASD have had their blocks evicted. If the physical cache still has insufficient space for the request, then the request is trimmed to fit in the cache.

Once blocks have been evicted, they must be *placed*. Blocks evicted from the global cache are not placed. Only blocks from the local cache are placed. Placement occurs in a priority scheme similar to eviction. Blocks are placed in the fastest NASD's portion of the global cache. The fast NASD caches the evicted blocks. This process continues using slower and slower NASDs until all evicted blocks have been placed.

3.4 Invalidations

Writes in performance fault masking occur in two ways. If the block to be written resides in the local cache, the data is written into the local cache in a write back fashion. Writes to blocks in the global cache incur invalidations. A write then will invalidate the block in the global cache, sending a message to the caching NASD to release the block. Before completing the write request, the block is written into the local cache of the owner.

3.5 Fragmented requests

A request for blocks may be satisfied from any combination of the owner's disk, the owner's local cache, and the global cache. When the local cache and global cache satisfy blocks requested, the request is fragmented. The client will then combine the fragmented request responses.

A performance fault masking storage system hides the vast majority of its complexity from the client. The illusion is only broken for fragmented requests. The client must know that its request could be returned as multiple response messages. This hiding simplifies the clients, allowing for a wider variety of clients to use a performance fault masking storage system.

4. Simulation methodology

To study the feasibility of our approach, we built an event based simulator using C++. We use a model that resembles the CMU NASD system paradigm [Gibson97] very closely. Since we felt that simulating the file manager and the storage manager would not aid us in studying the effects of cooperative caching, we decided to omit them from our model. We have a number of clients that are connected by a switched network to NASDs. A trace file of requests is fed into the simulator for each client. The trace files for the clients in our simulator thus contain object level read and write requests. The clients consume these requests and submit them to a central scheduler. A request is simulated as a series of discrete events where each event gives rise to the other that follows it in logical time. The central scheduler contains a time ordered heap of events from all active requests and simulates the action of every event at the appropriate time. Figure 2 shows the components in the system.

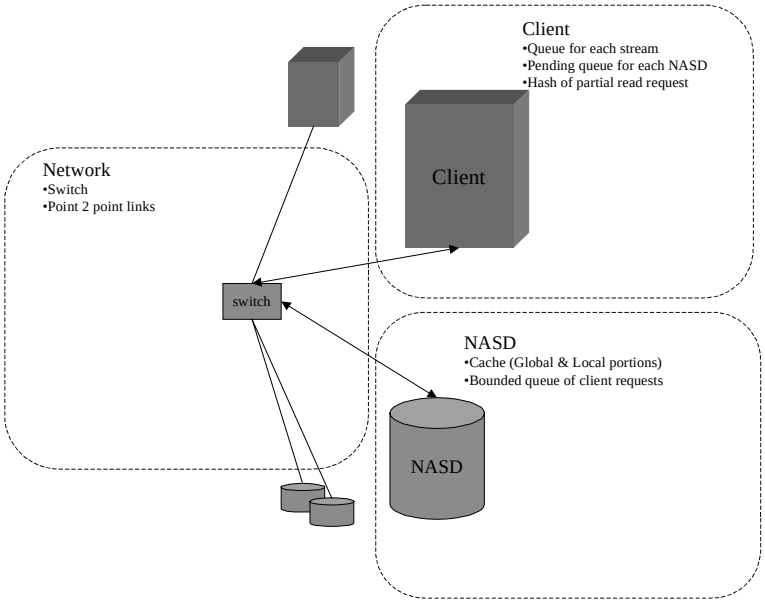

Figure 2. Overview of simulator

4.1 Configuration parameters

The simulator allows many of the environment parameters to be changed. The following parameters can be set using a configuration file:

- Number of NASDs
- Number of clients
- NASD parameters
  - Physical cache size
  - Transfer bandwidth, including time at which the transfer bandwidth occurs for bandwidth fault injection
  - Seek time
  - Block size
  - Queue length
- Network parameters
  - Packet header size
  - Transfer bandwidth
- Client parameters
  - Trace file

4.2 Trace driven

Our simulator services a trace of object read and write requests for every client. Since we do not model the file manager or the storage manager our simulator has no control over where the physical data is placed on the physical disks. This information is specified by the trace generating program. Striping policies would need to be implemented by the trace generating program.

4.3 Network

All links in our network have a configurable bandwidth. For most of our simulation studies we set it to 100 megabit (12.5 megabyte/sec) ethernet. We do not multiplex the link and at any given time we allow, at most a single request to utilize the link. We do not differentiate between the direction of the requests. A single request monopolizes the entire bandwidth of the link. A request is scheduled to use the link based on its arrival time. A queue is used to arbitrate access to the link. Also, the transfer of data from one end of the link takes place in one atomic transfer. We assume that when a transfer is to take place it can do so in one atomic step, and we assume that there will be sufficient buffer storage on the other end of the link.

All components in our model are attached to a central switch by the above mentioned links. Every request that traverses the switch spends some fixed time there.

4.4 NASD

Each NASD in our model represents a disk with a cache. The cache consists of two parts as described in section 3.1. By making every read and write go through the cache, we model the asynchronous interaction between a bus interface and the disk mechanism. At a given time only a single request can use the NASD. Other requests are placed in a bounded queue that is partitioned equally among the clients. We model a random seek to a block using the average seek time and the average rotational delay. We model sequential access. All writes go straight to disk. Dirty, remotely cached blocks are invalidated.

4.5 Credit based flow control

We limit the number of requests a particular client can have outstanding for a given NASD with the help of a credit based scheme. This models flow control over the client requests. At start up time, every client is allocated a fixed equal share of every NASD's bounded queue in the form of credits. The client can continue to send requests to the desired NASD until it exhausts its credits. Upon this exhaustion the client will have to suspend sending further requests to this NASD until its active requests complete.

4.6 Dependencies in trace requests

We allow dependency constraints to be applied to trace requests. If, for instance, a given request can only be issued upon the completion of a previous request we place it in a queue. When the active request completes we issue this suspended request. This way we are able to model streams. We identify each stream with a streamId. Different streams are independent of each other.

4.7 Limitations of the simulator

When a NASD interacts with another NASD it does not obtain adequate credits like what the client components of the simulator have to do. However, the policy that a NASD can only serve a single request at a time is still adhered to.

4.8 Experience with the simulator

We have had mixed experiences with our simulator. The time it took to complete a basic simulator was more than what we had anticipated (5 weeks vs. 2 weeks.) This gave us lesser time to obtain and analyze the results for our simulations. The simulator as it stands now is roughly over 7500 lines of commented C++ code. Writing the simulator has taught us how to write a better simulator in the future. We are pleased to have it working as well as it does in the limited time we have had.

5. Experimental results

To evaluate our algorithm, we ran synthetic traces through our simulator. Real traces of NFS servers were available, but all of the requests are sent to one server, requiring a mapping between NFS files and NASDs and NASD objects. Mapping may affect the outcome of studies. To avoid the additional variable of mapping, we focused on synthetic traces which are more easily characterized than traces from production systems.

Three workloads were run: a locality microbenchmark, web, and modified TPC-B. Each workload was run with local caching and with performance fault masking, called global caching in our results. The locality microbenchmark was used to evaluate the performance fault masking algorithm when locality for a few files was held high. The web benchmark approximated the workload a storage system may see from a web server that does not have a buffer cache. The benchmark was used to evaluate read intensive workloads with mostly small files. The TPC-B benchmark [TPCB] is a database benchmark with many read-modify-writes. TPC-B is used to evaluate the workloads with writes. All traces are striped with 10KB stripes.

5.1 Locality microbenchmark (trace rank)

The locality benchmark rank orders files from most to least referenced. Files are chosen for output to the trace using a lottery scheduling [Waldspurger94] like algorithm. The distribution of the file sizes approximate a normal distribution with 12,288 bytes as the median file size.

For evaluation of this benchmark, we scaled the number of clients from one to five while fixing the number of NASDs at three. The trace file only used two of the three NASDs. One NASD had a slow seek time of 30 ms and also received a bandwidth performance fault at three seconds decreasing its bandwidth to 3 MB/second. All NASDs start at 8.3 MB/second and have 128 KB caches. 1000 files were requested by each client. The working set size was approximately 768 KB.

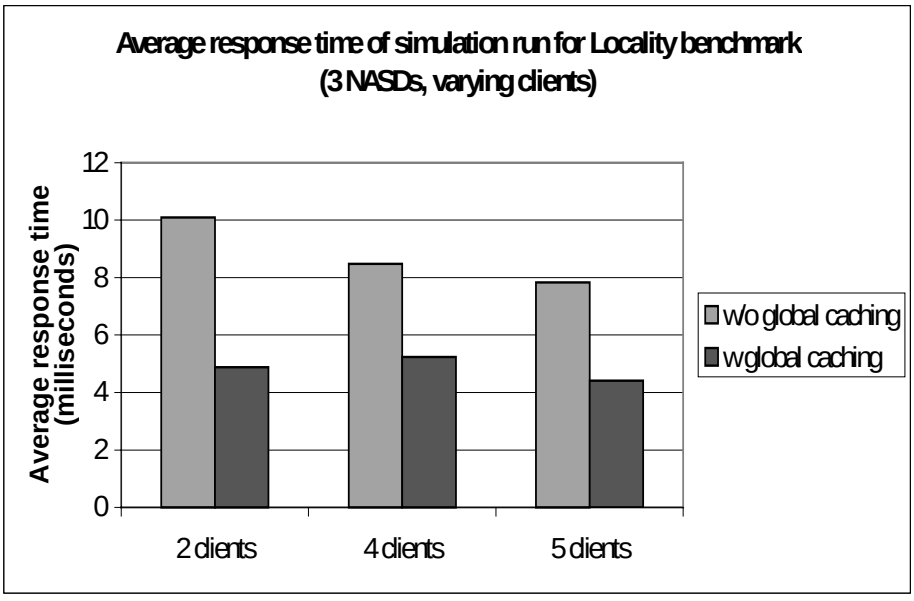

figure 4. Average response time for the locality benchmark

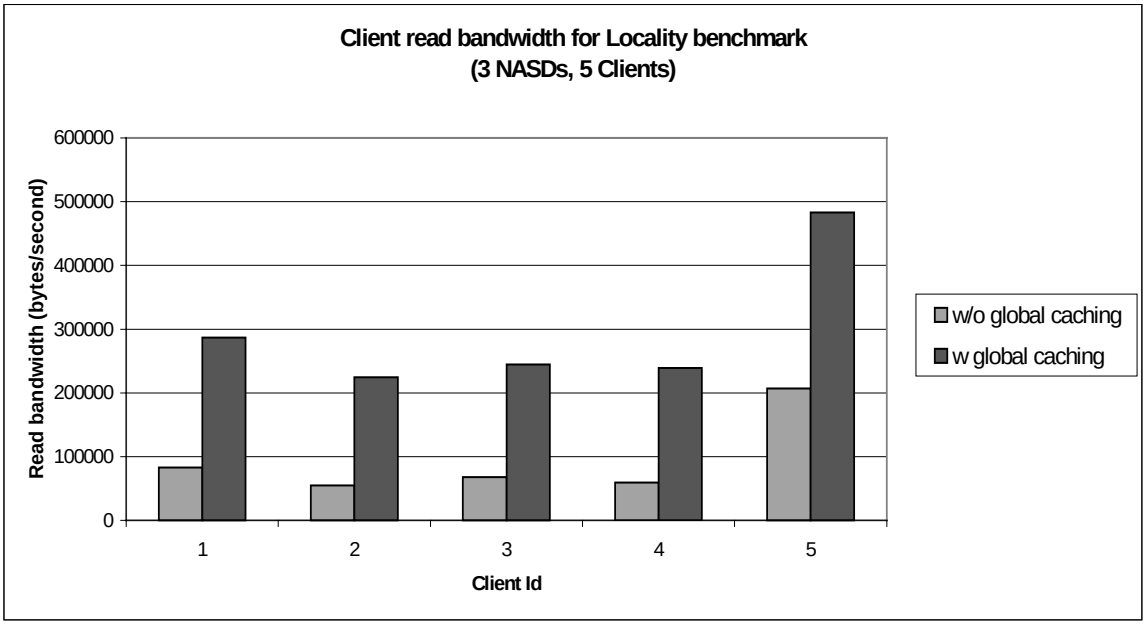

figure 5. Bandwidth at the client for the locality benchmark with 3 NASDs and 5 clients.

The locality benchmark showed that average response times decreased with global caching as shown in figure 4. The bandwidth seen by the client as shown in figure 5 increased due to greater hits in the global cache. Since one NASD was idle during the benchmark, the global cache was allowed to grow large and did not see contention from the local cache on the idle NASD. Other benchmarks showed more contention for the physical cache between logical caches as all NASDs serviced requests. Thus, contention for the physical cache is an issue.

5.2 Web benchmark

The web benchmark mimics a web workload. Files are distributed among different file types, like HTML and images, as discussed in [Arlitt96], but randomly chosen. Within each of these file types, the file size is fixed to the average transfer size in [Arlittl96].

We ran this benchmark by varying the number of clients and NASDs. The clients were varied from one to five, and the NASDs from two to five using all combinations of client and NASDs. All trace files had 1000 file requests with a working set size that increased proportional to the number of NASDs. The slow NASD was modeled after a Quantum Fireball lct15 [Quantum00] and the fast NASDs model an IBM Ultrastar 9LP [IBM00]. The major differences between these disks are seek time (12 ms. vs. 7.5 ms) and transfer rate (8.25 MBs/sec vs. 11.525 MBs/sec). No faults were injected for this benchmark; the performance fault of heterogeneity was present from the beginning of the simulation.

Overall, the web benchmark performed worse with global caching. The global cache was used infrequently for most experiments. The average request time and the total run time increased with global caching. However, the read bandwidth was almost consistently higher across all experiments. The increase in both the average read bandwidth and the average request time appears counterintuitive. With global caching, the average request time will increase due to greater network queuing and service time on the owner NASD because a lookup in remote block table is needed. Additionally, extra queuing which affects response times may occur since requests can be fragmented. Previous studies have shown that global caching can have greater burstiness. Since the bandwidth measurement is an average, the greater burstiness may lead to higher average bandwidths.

The disparity between each NASD as seen in figure 6 decreased when global caching was introduced which was expected. The reason for the effect, however, could be due to one of two reasons: the performance fault masking algorithm or network contention as a byproduct of the algorithm. However, since the global cache was used infrequently, few forwarded requests and invalidations occur. The algorithm appears to smooth disparities even though the global caching studies showed lower average bandwidth.

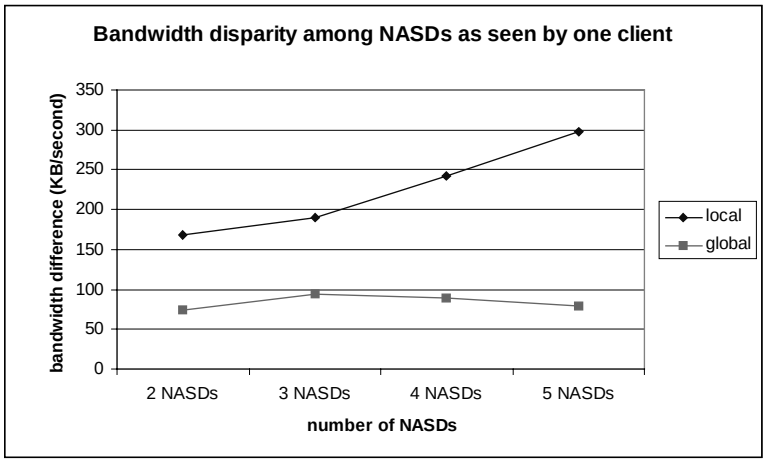

figure 6. The difference between the maximum and minimum average read bandwidths as seen by one client of the web study

5.3 TPC-B

The TPC-B benchmark used in our project is a modified version of the standard TPC-B. TPC-B simulates bank employees using a database. The benchmark has many read-modify-writes providing a tool to evaluate a more write intensive workload. Our TPC-B benchmark uses a uniform distribution between types of bank employees rather than the weighting used by the standard TPC-B benchmark. The size of the requests were modified to be 1024 bytes rather than 100 bytes to fit simulator constraints. The increased size of requests may be more realistic for today's databases as record sizes have increased since the TPC-B benchmark was obsoleted in 1995. The TPC-B benchmark has a specific distribution of the relations of the database across the five disks. We ignore this and stripe all relations on five NASDs using a simple round robin policy.

We ran the benchmark with five NASDs and varied the number of clients from one to six. Each NASD modeled an IBM Ultrastar 9LZX disk. We however changed the cache size from 512K to 100K to make the global cache size smaller than the working set size of our benchmark that was approximately 1 MB. By doing this, we tried to ensure that reasonable global cache reuse takes place. Two of the five NASD were made to have a higher average seek times. (30 milliseconds vs 7.5 milliseconds) Also a performance fault was introduced into one NASD decreasing its bandwidth from 8.25MB/s to 1 MB/s. We also ran the benchmark keeping the 5 NASDs and 1 client fixed and varying the cache size. What we hoped to discover was some interesting interaction between the global cache size and the working set size. The results obtained showed no such significant interaction and we do not report it here.

Our results for varying the number of clients indicate that the read bandwidth of each client in the global caching case was significantly higher than in the local caching case as seen in figure 7. However, global caching's average request time was also higher. This counterintuitive result was seen in the web benchmark also. Another interesting result obtained showed that the write bandwidth was lower than the read bandwidth in the local caching case. The writes tend to complete faster in the local caching case because we model a write back cache. Such a behavior is not seen in the global caching case because of the associated overhead that may result. A write may cause an invalidation to take place.

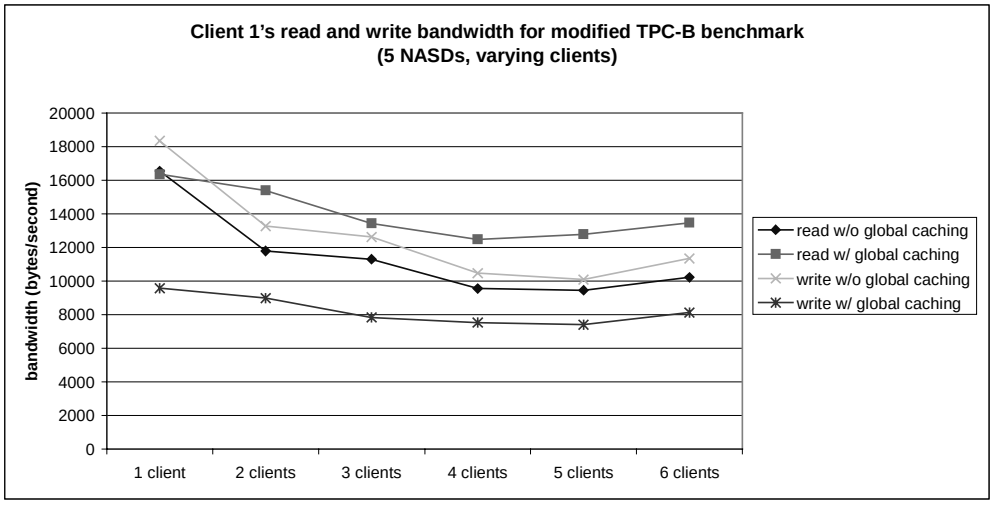

figure 7. Average read and write bandwidth of client one when the number of clients are increased for the TPC-B benchmark and the number of NASDs are fixed.

6. Future work

The work done for this project was preliminary and many more questions could be explored. Much of the time was spend developing a simulation environment, which was new for us. The simulator currently has less capabilities than needed for thorough studies of performance fault masking. Additionally, the simulator cannot run all workloads we wanted to study. We would like to remedy these shortcomings.

Several steps could be taken for further study. First, low contention of the physical cache by the two logical caches appeared as a strong reason why the locality study showed bandwidth and response time improvements. This conclusion is based on a small study and should be expanded.

Second, all studies used synthetic traces. Synthetic traces help gain an understanding of how the algorithm behaves in a controlled environment. However, real world traces would add a valuable understanding of the value of performance fault masking.

Third, a priority based algorithm was used. Other algorithms, like lottery scheduling [Waldspurger94], may perform better in certain domains than a priority based approach.

Finally, if further simulations provide greater insight into performance fault, a prototype implementation should be build to verify simulations and study the interactions of the algorithm on read hardware and software.

7. Conclusions

Performance fault masking has been able to decrease performance variability which was its goal. Using a global cache and cost-aware caching are key to achieving this goal. We have described the algorithm and the simulation environment. Additionally, some of the design tradeoffs have been briefly discussed.

Performance fault masking with this initial study shows promise. The locality microbenchmark shows that average request time is decreased when locality is high and contention for the global cache is low. Bandwidth is also increased. The TPC-B and the web studies revealed that average bandwidth and response time increased. This appears counter intuitive but the behavior is due to increased network activity and the burstiness of the storage system with performance fault masking. The web study shows performance disparities between NASDs decreases with performance fault masking. Finally, global caching and cost-aware caching did not help for the web and TPC-B benchmarks degrading performance, whereas the locality benchmark saw improved bandwidths and request times.

Acknowledgements

We would like to thank a few individuals for helpful conversations and suggestions during this project. In no particular order: Matt McCormick, Sambavi Murthukrishnan, Jonathan Ledlie, Andrea Arpaci-Dusseau, Ina Popovici, and Remzi Arpaci-Dusseau.

References

[Anderson95] Anderson, T. E. et al, "Serverless Network File Systems," Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 1995.

[Arlitt96] Arlitt, M. F. and C. L. Williamson, "Web Server Workload Characterization: The Search for Invariants," SIGMETRICS '96, May 1996.

[Arpaci-Dusseau99] Arpaci-Dusseau, R. H. et al, "Cluster I/O with River: Making the Common Case Fast," IOPADS '99, May 1999.

[Cao97] Cao, P. and Sandy Irani, "Cost-Aware WWW Proxy Caching Algorithms," Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems, December 1997.

[Dahlin94] Dahlin, M.D. et al, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," Proceedings of the First Symposium on Operating Systems Design and Implementation, November 1994.

[Feeley95] Feeley, M. J. et al, "Implementing Global Memory Management in a Workstation Cluster," Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 1995.

[Gibson97] Gibson, G. A. et al, "File Server Scaling with Network-Attached Secure Disks," SIGMETRICS '97, June 1997.

[IBM00] http://www.storage.ibm.com/hardsoft/diskdrdl/prod/9lp18xp.htm

[Patterson88] Patterson, D. et al, "A Case for Redunant Arrays of Inexpensive Disks (RAID)," Proceedings of SIGMOD, June 1988.

[Quantum00] http://www.quantum.com/downloads/pdfs/fireball_lct15.pdf

[Smaragdakis99] Smaragdakis, Y. et al, "EELRU: Simple and Effective Adaptive Page Replacement," SIGMETRICS '99, May 1999.

[TPCB] http://www.tpc.org/bspec.html

[Waldspurger94] Waldspurger, C.A. and W. E. Weihl, " Lottery Scheduling: Flexible Proportional-Share Resource Mangement," Proceedings of the First Symposium on Operating Systems Design and Implementation, November 1994.

[Wilkes95] Wilkes, John et al, "The HP AutoRAID Hierarchical Storage System," Proceedings of the 15th Symposium on Operating Systems Principles, December 1995.