Design of Implementation of a Cooperative Cache in a Cluster Based, Network Attached Storage Device Environment

Se-Chang Son
University of Wisconsin, Madison
sschang@cs.wisc.edu

Davin Sannes
University of Wisconsin, Madison
davin@cs.wisc.edu

Abstract

Reductions in network latencies and the availability of inexpensive processor are affecting the location and access of non-volatile storage by permitting efficient use of standalone storage devices attached to the network. Low latency networks have enabled clusters of workstations to access information stored in another machines memory faster than reading the information off of a local (or remote disk). This enables a cluster of workstations to cache information read from a network attached storage device globally resulting in both lower latency to user applications and fewer requests that the network attached disk must fulfill.

This paper examines the design and implementation of an in memory cooperative disk caching system/simulator, attempts to improve existing cooperative caching algorithms through simulation and examines the performance of each simulated scheme.

1. Introduction

Traditional computers have used local disks for providing non-volatile storage for information. The time to read data from a disk, or write data to a disk relative to the speed of the processor has increased greatly. In an attempt to amortize the cost of disk accesses, machines often store recently accessed data in main memory. On the first request for data off of the disk, the information is read from the disk into main memory and read from main memory into the processor. Subsequent requests to read the same information from the disk can be fulfilled by reusing the data from the first access that is still held in main memory.

As computer networks have enabled a disks to be shared among a group of computers, local caching of disk reads enabled the performance cost of a network disk access to be amortized across subsequent accesses by the local node. A machine that acting as a server, forwarding requests between the network and its local disk, was able to cache data read from its local disk to respond more quickly to subsequent requests from the network. By caching the data at both the disk's server and each node accessing the data, the average time to access data decreases and the number of nodes that can be supported by a disk server increases since it receives fewer requests.

Since accessing data on disk requires considerable time, a new type of disk was created to eliminate the strain caused by remote accesses to a machine that was acting as a server for its local disk, the network attached storage device (NASD) [Gibson 96]. A NASD is a disk with a built in processor and memory that handles requests from machines on the network. A NASD may work in conjunction with another file manager node, which will supply a remote process with a key to access particular information on the NASD. A NASD, a kind of smart disk, can support a variety of different services.

In the traditional disk sharing and caching environment, frequent accesses by each machine to common data, perhaps a system configuration file, cause information to be cached in memory on each node. Since network latencies have made accessing data in another computer's memory faster than reading information from disk, it may be sub-optimal to replicate data. For instance a machine that only accesses a very small number of bytes will have infrequently used data in its cache, while another machine accessing a large range of bytes might not be able to cache all of the information it repetitively accesses. If the machine with older information in its cache caches information for the machine with a large working set, the throughput of the system may increase and the number of requests sent to the disk server or NASD would be decreased.

Similarly, if accessing data from another machine's memory is fast, there is no need for every node to cache information. By sharing the memory of other machines in the cluster, each machine effectively has a memory the size of the sum of all the nodes on the cluster.

Section 2 discusses various approaches to sharing memory, focusing on caching disk accesses, in a cluster of workstations.

In Section 3, we describe the design and implementation of our system. Section 4 focuses on our simulations of various approaches. Section 5 discusses the simulation results and their implications.

2. Global Cooperation Schemes

Several previous studies have examined cooperatively sharing global memory resources among a cluster of workstations. [Wang93] provides a good description of some basic implementation issues, from the xFS projects implementation of a distributed file system. In [Dahlin94a], a wide variety of cooperative caching protocols are discussed, four schemes are simulated and compared to an optimal solution (using forward knowledge) and normal operations. Dahlin and Wang propose a near optimal solution called N-Chance. One shortcoming of the N-Chance algorithm is that it involves the use of a centralized server to coordinate the status of the cache, which limits the scalability of the system.

[Feeley95] describes and evaluates the insertion of a globally memory management scheme into the operating system at a low level, so that the whole system will benefit from sharing. Memory-intensive programs are able to perform 1.5-3.5 times faster when transparently using a globally managed memory. The implementation of this system requires the use of many global data structures that must be kept current so those local portions of the scheme can explicitly determine which machines are idle. This can cause multiple machines to locally reach the same decision about which nodes are idle, causing them to flood a node with concurrent requests.

We propose two schemes that attempt to combine the low complexity of the N-Chance scheme with the distributed nature of the global memory management scheme, without added complexity and load-balancing issues. The first of schemes is a distributed version of N-Chance, N-Chance-Dist. This scheme modifies the N-Chance algorithm by dividing the global object space amongst the managers (selected nodes). Each manager acts as an N-Chance server, keeping track of global state information for the objects that it is responsible for. In this paper, an object refers to a physical block in the NASD, typically 8KB, which is the granularity of caching that we chose for simplicity.

In the N-Chance scheme, when a non-replicated object (singleton) is evicted from a node's cache it is forwarded to random node. The receiving node will cache this object. If the cache is full an eviction must take place. If this cache contains any replicated objects, a replicated object will be evicted to make room. Otherwise, the least-recently-used object will be removed, but not forwarded to another node. In order to determine if an object is a singleton the server must be contacted or some type of callback system must be used to maintain accurate singleton status information. N-Chance-Dist forwards an evicted object to its manager, which discards it if it is replicated. If the object is a singleton, the manager checks to see if it manages any replicated objects, if so it forwards the singleton to a node with a replica and notifies the node to replace it. If it does not manage any duplicated pages it forwards the singleton to another manager (and decrements a hops-to-live field to prevent infinite page replacement). The node can choose whether or not to replace it (perhaps it's cache is not full, so it would prefer to simply add the singleton to its cache without evicting the other object), and notifies the server that it chose to cache both of objects (it must cache the singleton). This optimization would unfairly bias the comparison with normal N-Chance so the version of N-Chance we simulated uses this short-cut method. [Dahlin94] contains a description of the unmodified N-Chance Algorithm.

The second scheme that we propose, SeDa, modifies N-Chance two additional ways. First we used a least-often-used object replacement policy and second we forward an evicted singleton page to multiple nodes before evicting a page to make room for it.

Our proposed schemes focus on creating a distributed caching policy that works approximately as well as N-Chance forwarding. This is a reasonable goal since [Dalhin94a] show their approach to be near optimal. It is important to note that their approach achieves near optimality by offsetting the cost of randomly chosen suboptimal decisions by randomly chosen, locally superoptimal decisions. For example, an object may already reside in a local cache, on the first local access, partially offsetting the cost of not optimally placing the object on another node. This suggests that our approach may be able to use a distributed heuristic making fewer suboptimal decisions, or at least not perform considerably worse.

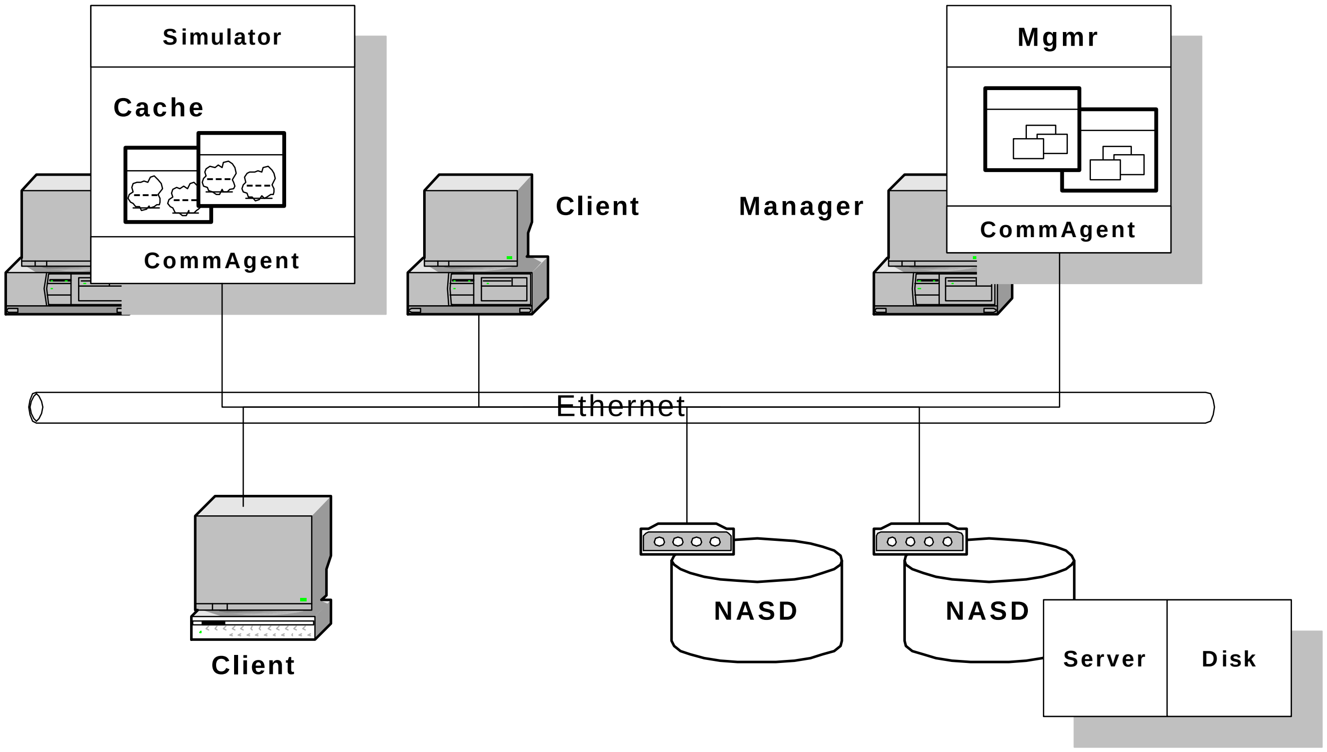
3. NASD Caching System

The network attached storage device framework typically involves 3 major components: server (device), file manager (issues keys to access the device), client (interface between NASD framework and normal file system). Our system integrates the cache into the NASD client, requiring no changes to the interface between the file system and the client. A typical NASD server does not require any changes. Although, we made significant changes to the NASD server the network interface and functionality supported was predefined and was not changed to reflect our cache. Our framework does not use a File Manager, however, integrating support for the use of a File Manager can be done in a few ways depending on the functionality that the NASD supports. One solution is to have the File Manager generate an extra key that the cache uses to verify if a request is authorized. Any functionality that is supported by the NASD server, can be easily integrated into the cache, by adding a translation layer, and may require a similar translation layer at the File Manager, similar to the above description.

3.1 System Design

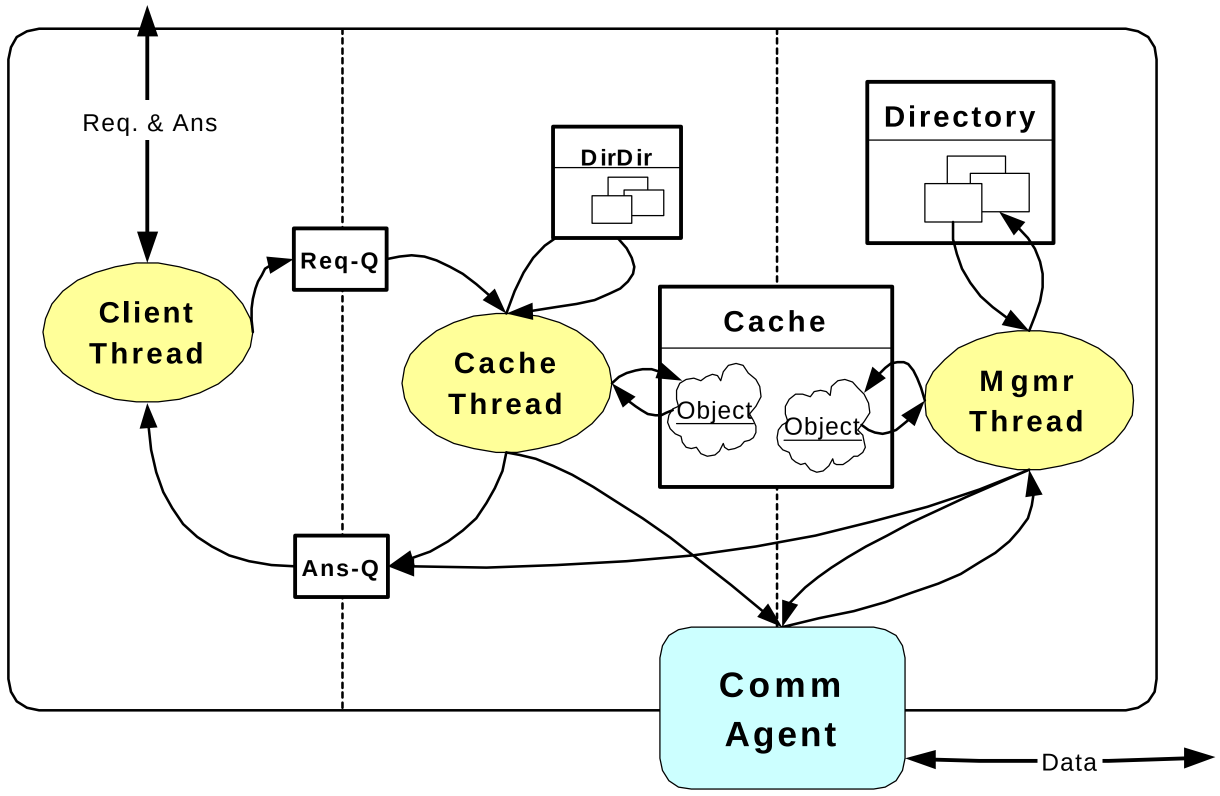
The following diagram provides a general overview of our system design. The system is composed of three major parts: NASDs, a high speed network (Ethernet) and nodes in the cluster of workstations. In general, there need not be a distinction between Manager and Client nodes. In our simulations all of the nodes in the cluster are homogeneous and therefore all nodes benefiting from the use of cooperative cache should share the cost. If it is desired that a node only act as a client, it need not register itself as manager. Each node is comprised of a communication agent (CommAgent), a cache/manager (Cache) and a client (Simulator). In the diagram above the client was omitted from the manager is omitted from one node for illustration purposes. Each NASD is comprised of both a NASD server and a physical disk drive. Typically, these are collocated inside of one physical device. In our simulations the server and disk were contained within one dedicated node in the cluster. The Ethernet displayed must be a low latency, high throughput connection. [Dahlin94a] suggests that 10Mps Ethernet switched Ethernet may be marginally acceptable, but focused on 155Mbps ATM. For the purposes of our discussion and simulations we assume a 100Mbps switched Ethernet is used.

An arbitrary number of nodes and NASDs can be supported by the cache, however, under heavy usage the network may become congested leading to poor performance, possibly worse than without the cooperative caching scheme. In a well-balance system, a system without such bottlenecks, this is a non-issue.



3.2 Detailed Design and Implementation

The NASD portion of system is comprised of a server and disk. This is based on an in kernel device driver implementation, which was ported to the user level, extended to support sequential accesses from multiple nodes. The server portion of the NASD, actually a C++ object, waits for requests from the network (via the CommAgent), calls the appropriate functions of the disk and sends the result back to the original requester. Since we are running the NASD as a user level process, defined a Disk C++ class which acts as a translation layer between the server and the underlying file system. The disk contains a group of objects that encapsulate the filename used by the underlying file system, and metadata about of the object that is retained in memory.



The main functionality of system is encapsulated inside of the cache. The Cache is a C++ class that interacts via an API with the clients. The Cache's API is the same as the Disk's API enabling the Cache to replace the Disk easily in user level code. The diagram above shows a conceptual view of how the cache works. The cache is composed of three threads:

1. Client thread: the execution of public functions that are called by the client
2. Cache thread: gets requests from the client thread, handles local hits and forwards local misses to the appropriate party via the CommAgent
3. Manager (Mgmr) thread: waits for events to come from the network and acts appropriately, sending replies and forwarding requests through the network.

Client Thread Execution:

Put request information into the Request- Q.
Wait for the answer to come into the Answer-Q.

The request and answer queues are implemented as synchronized classes in C++. They use condition variables and mutexes to provide mutual exclusion between two threads accessing a variety of shared memory locations. All of the cache API function calls are blocking, which ensures that at most two threads will attempt to access these queues. Our design is generalized to allow extension to a multithreaded client accessing the cache. At this stage each instance of the Cache class can only be accessed by one thread, i.e. it is not thread-safe.

The cache also contains a variety of objects handling various functions. The mapping of <drive, object> pairs to their appropriate managers in handled by function class to the DirDir object which keeps track of all the managers and hashes the <drive, object> pair to the IPv4 address of the node that manages the pair. Currently, the manager function does not support dynamic changes. To handle a dynamic change currently, requires notifying all nodes of the change, and flushing the global cache. Our design supports gracefully, dynamically changing the hashing function based on network load, however, this functionality has not be implemented.

The cache contains a data store denoted Cache in the diagram that maintains a least- recently-used (LRU) or least-often-used (LOU) list of objects currently in the cache. Our simulator uses both of these modes for various simulations. This object is synchronized to provide safe access between both the manager and the client thread. Internally objects are stored in a C++ STL map which provides efficient access to data and a C++ STL vector is used for maintaining the LRU and LOU lists.

The CommAgent is a thread safe C++ object that encapsulates packet marshalling/demarshalling, and network transport. Our implementation currently supports reliable TCP transfers and unreliable UPD transfers. We plan to implement reliable transmission over UDP in the future to increase the scalability of the system. We use TCP while running all of our simulations. The TCP connection setup/tear down overhead is also unacceptable in operational implementation of this system. The need for persistent TCP connections to avoid the setup/tear down costs is why TCP is not scalable, since each connection is defined by a file descriptor and only a limited number of file descriptors are allowed per process.

```
Cache Thread Execution:

 Wait for the answer to come into the Request-Q.

 If read: get object from the local data store and
                  place it into the Answer-Q (hit)
          lookup manager and forward request (miss)
 If write: update* the cache to contain the object
          lookup manager and forward data
 If other: take local action* and forward request

* includes notifying appropriate managers of changes
concerning where and object is cached
```

The last thread that runs inside of the Cache object is the Manager thread. This thread listens for events from the network (via CommAgent) and takes appropriate actions depending on the event. It accesses the Directory keeping track of which nodes are caching objects that it manages. This directory is only contains information about currently cached objects, not all objects in general. This supports an arbitrary number of drives and objects (actually, the 96-bit globally unique object identifier limits this). Just as the Cache is the most considerable part of the overall, the manager thread is the most considerable part of the cache.

```
Manager Thread Execution:

 Wait for an event to come from the network.
 If read: get object from the local data store and
                  place it into the Answer-Q (hit)
          lookup to see if it is cached elsewhere and
                  forward request (semi-hit)
          send the request to drive (miss)

              <continued in next column>
```

```
Manager Thread Execution < continued >:

 If write: update the local data store if it contains the
                  object.
          send an write packet to each node that
                  currently is caching the object and
                  send a write packet & data to the
                  drive (manager)

 If add:  add the corresponding <object, node> pair
                  to the list of pages (manager)
          lookup and forward the add to the correct
                  manager (not manager)

If evict: remove source from list of nodes caching
                  the object (manager)
          if the object is a singleton ignore it
          if there is room here add it and notify
                  manager of object. If no room run
                  page replacement algorithm
          if this the evict_N_to_make_room flag is
                  set, evict object N (not manager)
          if it manages any duplicated objects then
                  send evict_to_make_room to the
                  node and remove from cache list
           if only manages singletons then choose a
                  random node and send to node
                  (all manager nodes)
when evicting a page from the data store at any time
          decrease the nchance value by one and send
          to the manager.
```

The general algorithms described above vary depending upon the type of caching scheme in use. The Cache object and its component objects support 4 modes of operation: no cache, N-Chance cache, N-Chance distributed caching, and SeDa caching (the second variant of N-Chance caching).

The Simulator is a modified client that dynamically varies the kinds of requests made on the caching system. It reads setup data from configuration files and writes trace information to result files. It allows the specification of working set size, the range of the initial working set, what percentage of requests lie outside of the initial working set (if random > 0, then the working set changes over time), what percentage of requests should be writes, how often to log summary data to result files on disks. Another configuration file specifies what drives exist in the system and what objects already exist on that drive. We also have created a manual simulator which allows us to specify the type of cache to use, what other nodes are running and to interactively send read requests, write requests, and dump the status of the cache (all the current objects cached and run time statistics). We used the manual simulator to do initial verifications that our schemes were working as described.

4. Simulation Methodology

On a cluster of 36 Linux workstations (dual Pentium III-550, 1 GB RAM) connected by 100Mbps switched Ethernet. The cluster of workstations was concurrently used to run a variety of scientific applications and database functionality. After initially running our simulations using 16 nodes and 5 drives, we scaled our simulations down to 6 nodes and 3 drives to make the amount of data easier to analyze. Our simulations were designed to help us compare the 4 different caching schemes, and analyze the performance of the two new caching schemes that we are testing, namely N-Chance-Dist and SeDa caching.

In our approach we ran four synthetic workloads under each of the four caching schemes. The workloads that had the following characteristics:

1. Local Reuse: Each node accesses objects that no other node accesses.
2. Global Reuse: All nodes access the same set of objects.
3. Random Reuse: Each node randomly accesses objects.
4. Mixed Workload: Each node accesses a different combination of objects, some random, some globally shared, some locally shared.

For each simulation run we traced various information concerning how each node performed. We kept track of how long each request took based on simulated times, the actual amount of time it took, where the read requests were filled from (local hit, manager hit, global hit, disk hit or a disk hit based on stale manager information) as well as, how many reads request were made. We made use of simulated times and actual times to account for variances caused by other jobs running on the cluster other than our simulator.

Our simulated times are similar to those used in [Dahlin94a]. We changed a few of the times to account for running over 100Mbps switched Ethernet. The times described in [Dahlin94a] for Ethernet were very optimistic, our values are less optimistic for network transport time but are optimistic. We reused the memory access times from [Dahlin94a] but we found that our memory access times had improved twenty percent in some cases. In the local reuse simulation it was noticeable when data was stored in a memory cache as opposed to main memory.

4.1 Simulation Results

After running each simulation, we combined the results from each of the six nodes and calculated the average times that each read request required. This provides with a view of how total system performance is affected by each scheme. We ignored write requests since all request complete in constant time and the affects of writes show up in the read request times.

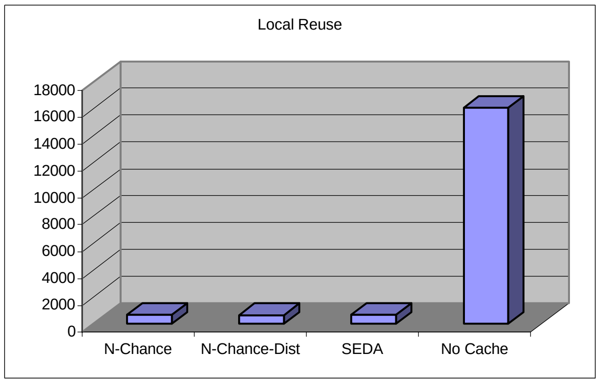


Figure 1: Average Read Times under N-Chance, N-Chance-Dist, SeDa and No Caching

In Figure 1 it is apparent that under all each of the caching schemes there is a very large improvement in access time compared to not caching. Figure 2 is the same data without scaling to show No Cache.

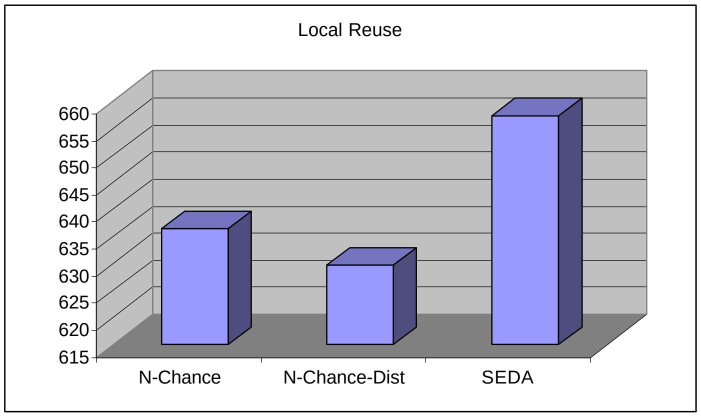


Figure 2: Average Read Times under N-Chance, N-Chance-Dist and SeDa Caching

Without showing the No Cache option it appears that there is a significant difference between the average times, however, one should note the there is only a 5% variation in the times spent. We attribute the difference between N-Chance and N-Chance-Dist to be the distributed management nature. With distributed control, more pages are cached at the server which reduces the access time by one network hop. SeDa caching's slightly worse performance is partially attributable to it object eviction policy, since objects were accessed randomly within the working set it and attempting to find an idle node. The latter would cause pages to migrate away from its manager more than either of the N-Chance strategies.

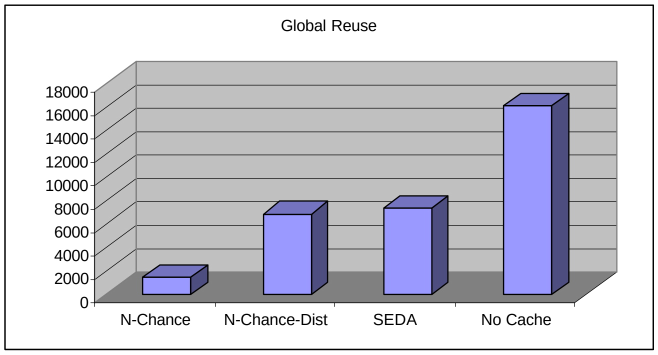


Figure 3: Average Read Times for Global Reuse

In Figure 3 we see that under a global reuse workload N-Chance performs much better than other two works loads. This is because of N-Chance's centralized server. The centralized server caches most of the data since requests are repeatedly set for the same data. In the N-Chance-Dist and SeDa schemes data is moved throughout the cluster and managed with only local knowledge, by evicting pages that haven't been used recently (or used often) these schemes force data away from the server. Unfortunately, the global working set was not large enough to determine whether or not this was truly the case. Further testing would helps us to verify this fairly large difference.

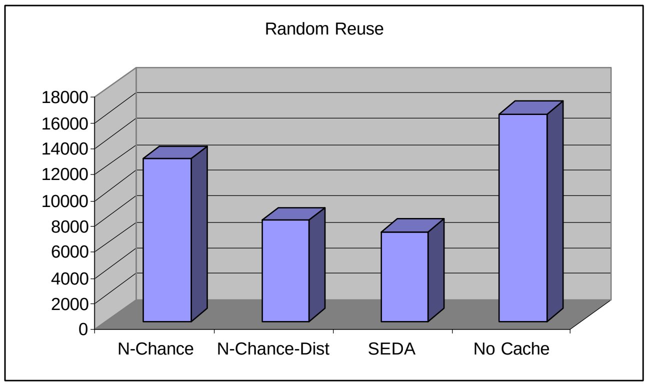


Figure 4: Average Read Times for Random Reuse

Under a random reuse workload SeDa outperforms the other two schemes. This is expected because SeDa does a better job of keeping pages alive, by migrating pages until finding an idle node. In both N-Chance strategies when a singleton page evicts another singleton page, the new singleton page is evicted. N-Chance-Dist likewise N-Chance-Dist outperforms normal N-Chance since more pages are evicted at the server in N-Chance. Unfortunately, this problem is particular to our optimization of N-Chance which reduces control traffic on the network. By forwarding all objects to the manager, which then randomly selects a node, we cause the server in N-Chance to discard more objects.

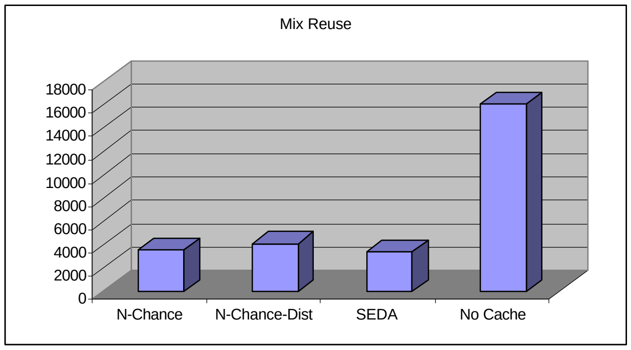


Figure 5: Average Read Times for Mixed Reuse

Actual workloads running on the cluster will probably not fall into one of the previous categories of data locality. The mixed simulation combines nodes running each of the previous workloads and runs them together on the cluster. We see in this example that all three schemes perform fairly well. SeDa outperforms the other two schemes because under a mixed workload environment it keeps singleton objects alive longer and takes usage into account. So one node accessing a small number of objects, will not have its objects evicted for recently, infrequently used objects. These results are not surprising given the previous data. Although, N-Chance out performs N-Chance-Dist in terms of global throughput, the node running the centralized server becomes swamped with only 13 nodes running.

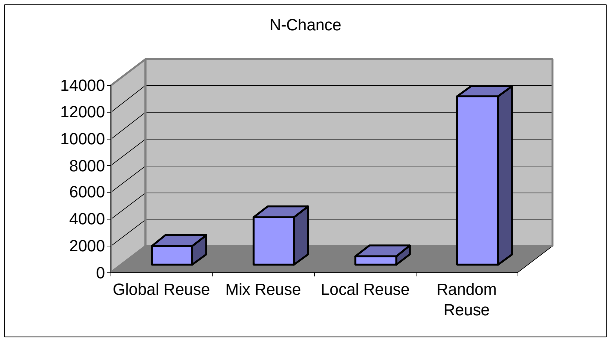


Figure 6: Average Read Times for N-Chance

Figure 6 shows the performance of N-Chance of each of the workloads. The relative performance on each workload makes sense given the centralized nature and the crippling affects that our optimization has. Further testing without the optimization is required to determine how N-Chance proposed in [Dahlin94a] would compare under our simulation loads.

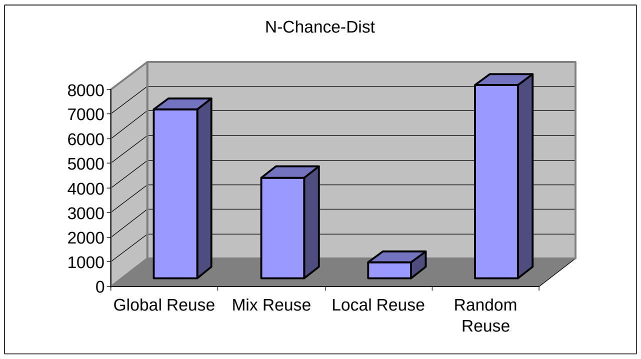


Figure 7: Average Read Times for N-Chance-Dist

Figure 7 illustrates the performance of N-Chance-Dist under each of the workloads. The performance under Global Reuse is surprising. In order to determine why global reuse does not perform better we will have to conduct addition testing.

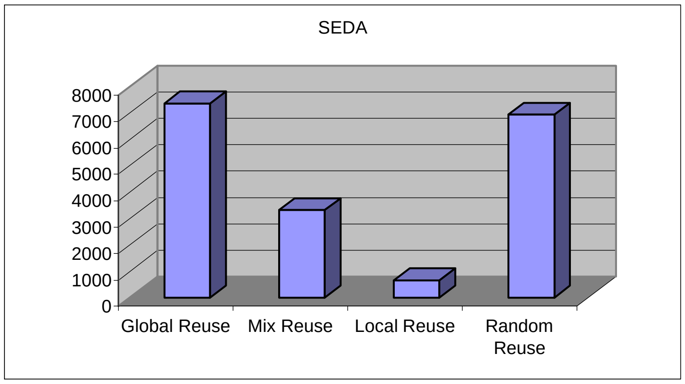


Figure 8: Average Read Times for SeDa Caching

Figure 8 shows the performance of SeDa caching over each of the workloads. The performance of SeDa caching is what we had expected, with the exception of the global case, since forwarding evicted singleton objects and basing the eviction policy on how often an object is accessed causes better performance under heterogeneous workloads and poor performance under random reuse. It is surprising that SeDa performs better in a wholly random environment than in an environment with global reuse. To determine how SeDa cache can be improved to take this into account will require more simulations to be run.

5. Conclusions and Future Work

We feel that we have successfully shown that a simple distributed version of the N-Chance algorithm with a optimized object migration policy, performs fairly well in some cases. However, our results are mixed concerning which caching scheme is more likely result is better performance on real workloads. Running more simulations will help us to better understand how each of the algorithms will be affected by real workloads.

The main contributions of our research is that although N-Chance performs fairly close to an off-line optimal global caching policy, there are certain workloads under which a different policy can perform significantly better.

Our research questions that claim made in [Dahlin94a] that LRU is not much different than a usage based scheme. Further research into how each replacement policy performs under real workloads is warranted to quantify the improvements. We feel that further simulation and analysis will help to clear up the questions that various results generated.

In the future, we plan to run more simulations using our simulator to fully understand the SeDa and N-Chance-Dist cases that were hard to analyze. We would like to run the [Dahlin94a] N-Chance algorithm without our optimizations to see how they affect the overall performance of the system. We would like to analyze our results on a per node basis to see how each caching scheme affects individual nodes. We feel that simulations are no replacement for real workload data. We plan to implement our simulator fully so that we can integrate it as a device driver at kernel level.

Although further research is needed to determine how the NASD environment will affect cooperative caching schemes, we feel that our cache can easily be extended to accommodate more complex drives.

Bibliography

Cooperative Caching

[Voelker98] Geoff Voelker, Eric Anderson, et al. "Implementing Cooperative Prefetching and Caching in a Global Memory System". *Proc. of the 1998 ACM Sigmetrics Conference on Performance Measurement, Modeling, and Evaluation*, June 1998.

[Voelker97] Geoffrey Voelker, Herve Jamrozik, Mary Vernon, et al. "Managing Server Load in Global Memory Systems". *Proc. of the 1997 ACM Sigmetrics Conference on Performance Measurement, Modeling, and Evaluation*, June 1997.

[Dahlin94a] Michael Dahlin, Randolph Y. Wang, Thomas E. Anderson and David A. Patterson. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance". *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

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

[Jamrozik96] H.A. Jamrozik, M.J. Feeley, G.M. Voelker, J. Evans II, A.R. Karlin, H.M. Levy and M.K. Vernon. "Reducing Network Latency Using Subpages in a Global Memory Environment". *Proc. of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[Anderson95] Tom Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, Randy Wang. "Serverless Network File Systems". *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[Neefe97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randy Wang, Tom Anderson. "Improving the Performance of Log-Structured File Systems with Adaptive Methods". *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[Dahlin94b] Michael Dahlin, Clifford Mather, Randolph Wang, Thomas Anderson, David Patterson. "A Quantitative Analysis of Cache Policies for Scalable Network File Systems". *SIGMETRICS '94* , 1994.

[Wang93] Randolph Y. Wang, Thomas E. Anderson. "xFS: A Wide Area Mass Storage File System". White Paper, University of California, Berkeley, 1993.

Network Attached Storage Devices

[Gobioff99] Howard Gobioff, "Security for a High Performance Commodity Storage Subsystem", PhD Dissertation, CMU-CS-99-160, July 1999.

[Gibson99] Garth A. Gibson, et al. "NASD Scalable Storage Systems". *USENIX99*, June 1999.

[Gobioff98] Howard Gobioff, David F. Nagle*, Garth A. Gibson. "Integrity and Performance in Network Attached Storage". CMU SCS Technical Report, CMU-CS-98-182, December 1998.

[Gibson97] Garth A. Gibson, et al. "File Server Scaling with Network-Attached Secure Disks," *Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, June, 1997.

[Gibson96] Garth A. Gibson, et al. "A Case for Network-Attached Secure Disks". CMU SCS technical report, CMU-CS-96-142, September 1996.