Dynamic Adaptive File Management in a Local Area Network

Jiong Yang, Wei Wang
T.J. Watson Research Center
IBM Research Division
{jiyang, ww1}@us.ibm.com

Richard Muntz, Silvia Nittel
Department of Computer Science
University of California, Los Angeles
{muntz, silvia}@cs.ucla.edu

Abstract

In light of advances in processor and networking technology, especially the emergence of network attached disks, the traditional client-server architecture of file systems has become suboptimal for many computation/data intensive applications. In this paper, we introduce a revised architecture for file management employing network attached storage: the dynamic file server environment (Dynamo). Dynamo introduces two main architectural innovations: (1) To provide high scalability, the file management functions are mainly performed cooperatively by the clients in the system. Furthermore, data is transferred directly to the client's cache from network-attached disks, thus avoiding copies from a disk to the server buffer and then over the network to the client. (2) Dynamo uses a cooperative cache management which employs a decentralized lottery-based page replacement strategy. We show via performance benchmarks run on the Dynamo system and simulation results how this architecture increases the system's adaptability, scalability and cost performance.

1 Overview of Dynamo

The Dynamo architecture consists of four layers: the disk I/O layer, the cooperative cache manager layer, the file manager layer, and the coordinator layer. These four layers interact with each other as shown in Figure 1.

Disk I/O Layer

The lowest layer of Dynamo is the I/O layer that provides data I/O from and to storage devices. We assume that the storage devices are disks or disk arrays. The I/O layer resides on the disk controllers of the network attached storage devices and maps files and data pages to storage locations on a storage device in a similar fashion to the I/O layer in a conventional file system.

Cooperative Cache Manager Layer

In a LAN, the working set size of local applications will vary over time resulting in time when the working set of an individual node exceeds the node's local physical memory; however, the aggregate size (i.e., of the union) of all working sets is often less than the aggregate nodes' combined main memory. Therefore, it is beneficial if local working sets can "spill over" to other node whose main memory is not fully utilized. To achieve this, Dynamo treats main memory from all nodes as a pool of global memory with a cooperative cache manager layer on each node. Each node treats its own memory as the *local cache* and memory on other nodes as a *remote cache*, intermediate between its local cache and secondary storage. The cooperative cache managers are responsible for managing the remote caches as well as their own local caches.

The cooperative cache manager (CCM) consists of two components: the local cache manager (LCM) and the distributed cache manager (DCM) which collaborates with other DCMs. The LCM performs cache replacement of its local cache while a DCM defines, in concert with other cache managers, a decentralized scheme for global cache management. Using a lottery-based scheme, the DCM determines how and whereto local pages should be evicted. CCM also maps between the logical address of a file, and its memory address.

File Manager Layer

In Dynamo, large parts of server functions are distributed to nodes in the LAN and are handled by them in a cooperative fashion. We distinguish two software components in Dynamo which provide the functionality of a traditional file system: the *coordinator* and the *file managers*. The coordinator is a small remaining part which runs on one or more well-know core nodes, while the file managers execute on any nodes and perform most of the traditional data management functions. We refer to the manager of a file as the owner of that file.

Coordinator Layer

The coordinator manages centralized information for all nodes, and coordinates which data partition is managed by which nodes, keeps track of which data is managed by which file manager, informs nodes about owners, and participates in ownership transfers between nodes as well as transaction management. Because of its central role, a coordinator (or a set of cooperating coordinators) run on a backbone of reliable, well-known nodes.

In the following sections, we describe the contributions and specifics of Dynamo such as the dynamic data management strategy and the decentralized cache management strategy for its cooperative cache in detail.

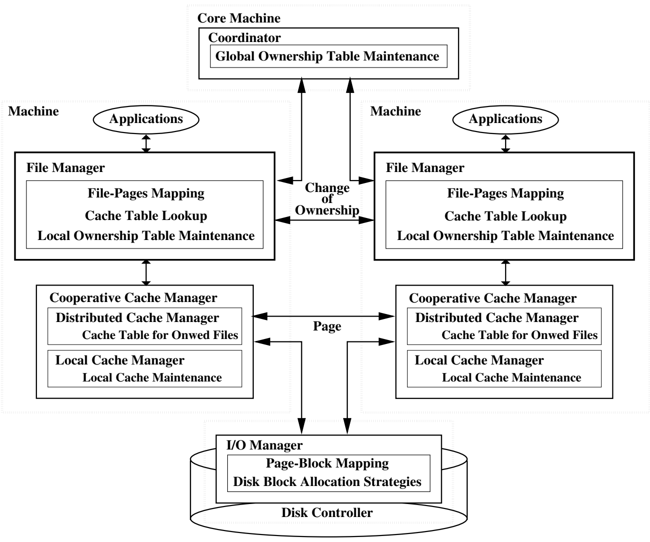


Figure 1: Dynamo Architecture

2 Data Organization

To support the dynamic data management and cooperative caching, Dynamo employs a data organization scheme that supports the flexible assignment of file management to nodes. In a UNIX file system, directories, subdirectories, and files form a natural hierarchy structure. DynamO also utilizes this hierarchy. The parent-child relationship in the hierarchy represents the directory-subdirectory relationship within the file system. To keep track of this parent-child relationship, we use a *prefix-based naming structure*. Each level in the data management hierarchy scheme is represented by one byte in the identifier; the maximum fanout of a node is 255. The length of the identifier can be chosen by the system administrator, and depends on the overall data size of the system. Typically, the identifier size is 8 bytes, and it can address 10^{13} GB of data. For example, an entity of level 4 in the hierarchy has the identifier "11230000"; its children can be identified by keeping the same "1123" prefix in their identifier, i.e. an entity with the identifier "11232100" is a descendent of it.

3 Cooperative Caching

In this section, we describe the details of cooperative cache management in Dynamo.

3.1 Page Retrieval

Page retrieval from either external storage or remote caches is the most basic functionality of any cooperative caching scheme. In Dynamo, page retrieval includes some of the following procedures as illustrated in Figure 2.

1. Cache discovery. This procedure is used to discover the node on which a cached copy of the desired data exists if there exists one.
2. Local page retrieval. If the cached copy of data is stored in the local cache, then this procedure will be invoked to load the page into the application user space.
3. Remote page retrieval. This procedure is designed to fetch a cached copy of the data from a remote node to the local cache if the data is not cached locally.
4. Disk page retrieval. If there is no cached copy anywhere in Dynamo, then this procedure has to be invoked to fetch the page from disk(s).
5. Page replacement. If there is no space on a node to load the data, then some existing page(s) have to be replaced.

Each of these five procedures is described next in some detail.

3.1.1 Cache Discovery

Figure 4 describes the operations involved in cache discovery. Assume that an application on node A tries to access data owned by the file manager on node B. The application first contacts the local file manager with the ID of the desired file and the offset, length of the data in the file (Step 1). The file manager[1] first performs file to page mapping and identifies the desired page(s). Then the file manager checks whether it already knows the owner of the data (Step 2). If not, it then contacts the coordinator for this information (Step 3).

As the name implies, the coordinator has a delegating, coordinating, and book keeping role in Dynamo. It manages the information that has to be kept centrally for all nodes; however, access to this information is kept minimal to avoid the analog of the server bottleneck[2]. As we will discuss later, the coordinator has several responsibilities: (1) keep track of all network-attached storage (NAS) space in the LAN, (2) keep track of which file manager manages which data.

The coordinator responds to an initial data request by the local file manager on a node by looking up its *global ownership table* (as illustrated in Figure 3(b)) to identify the owner (Step 4), and returns the ID of the owner and the highest level

[1] We will explain the functionality of the file manager in detail in next section.

[2] According to the size, the system can employ a single, central coordinator or a distributed coordinator. In the case of a distributed coordinator scheme, the set of nodes running a coordinator is known to all nodes in the LAN.

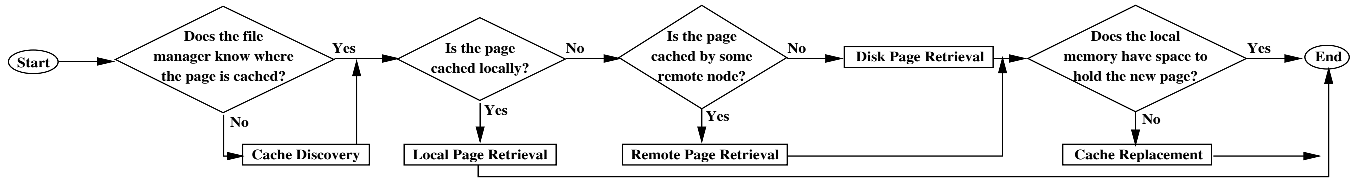


Figure 2: State Diagram of Page Retrieval

homogenous entity[3] which contains the data page to the file manager on A (Step 5)[4]. Note that if the owner of some data has become known to the local file manager previously, the above process is omitted. Ptherwise, the file manager on A directly sends a message to its peer who owns the data (i.e., the file manager on B) to request the proper lock[5] and ask for the ID of the node where the desired data is cached (Step 6).

pageID	status	nodeID
00112233	single	131.179.99.79
00331127	replicate	131.179.99.79 131.179.99.69
. .	. .	. .

(a) Data Page Cache Table

entity	node ID
13400000	131.179.99.79
12000000	131.179.99.69
20000000	0
. . .	. . .

(b) Global Ownership Table

Figure 3: Two Tables

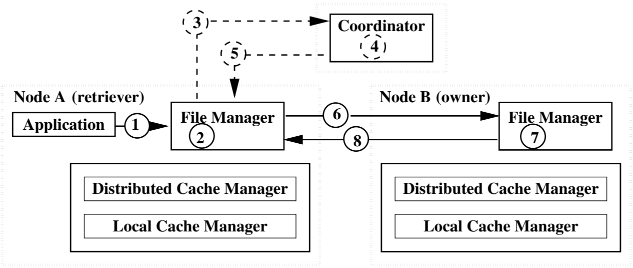


Figure 4: Cache Discovery

The owner of the page(s) uses its *data page cache table* to determine where the desired pages are located (Step 7). A "data page cache table" is maintained by each DCM. On each node, the data page cache table records which node has a cached copy of a locally owned data page. As shown in Figure 3(a), each entry in the data page cache table consists of three fields: *pageID*, *status*, and *cached node ID* (represented by the IP address of the node). The status of a page could be one of the follows. (1) *single*: Only a single copy of the page resides in the memory of some node. (2) *replicated*: The page is cached by multiple nodes. The *cached node ID* field is a linked list of the IDs of the nodes which cache a copy of the page identified by *pageID*.

If the desired page is cached, then the owner will return the ID of the node which holds the cached page (Step 8). Otherwise, the ID of the disk which stores the page will be returned with the page ID.

3.1.2 Local and Remote Cache Page Retrieval

If the page resides in the local memory, then the file manager requests the local cache manager (LCM) via the DCM to locate and load the page into the application user space. This is shown in Figure 5.

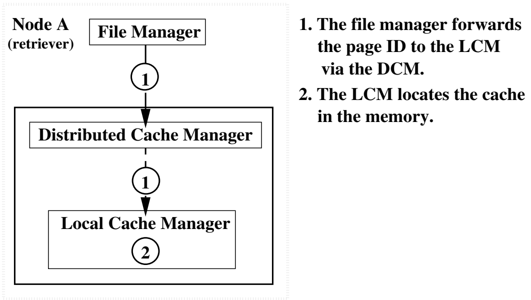


Figure 5: Retrieve a Page Locally

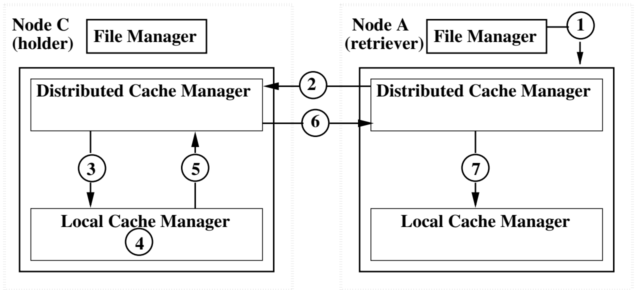


Figure 6: Retrieve a Page from a Remote Node

If the page resides on other node, the remote cache retrieval procedure has to be invoked. The DCM plays an important role in retrieving a remote cached page. The entire procedure is illustrated in Figure 6. After determining the location of the

---

[3] A highest level homogeneous entity is the entity whose owner is different from at least one of its siblings.

[4] A becomes the owner/manager of this entity and all its descendants.

[5] In Dynamo, we assume the page-oriented locks unless otherwise specified.

cache holding the desired page, the file manager forwards the page ID and the node ID of the cache holder to its DCM (Step 1). The DCM sends a request with the page ID to its peer on the cache holder (say node C) (Step 2). After obtaining the page ID, the DCM on node C contacts its LCM to locate the page in its memory (Step 3 — 5), and then sends back the page to its peer on node A (Step 6). Finally, the DCM on A stores the page in local memory via its LCM (Step 7).

3.1.3 Disk Page Retrieval

As illustrated in Figure 7, if the page is not cached, the file manager forwards the page ID and disk ID to the LCM via the DCM (Step 1). The LCM then sends a request to the I/O manager on the disk controller (Step 2). The I/O manager, in turn, locates the page on its disk (Step 3) and returns it to the LCM of the retriever (Step 4).

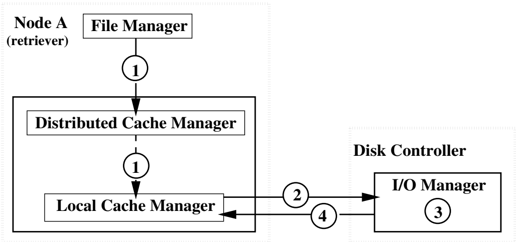


Figure 7: Retrieve a Page from Disk

Every time a page is fetched from a disk or a remote node, the LCM stores the page in its memory. However, if the local memory is full, some page has to be replaced.

3.2 Page Replacement

If a page in memory has to be replaced, this page can either be evicted to another node or simply be discarded. This process is illustrated in Figure 8.

When the DCM on B receives a request on pages status, it will check the data page cache table and return back the requested information (Step 4, 5). In turn, the DCM on A forwards this information to its LCM (Step 6). The LCM then chooses a page owned by B to be replaced according to the following criteria (Step 7): If there is a page whose status is *replicated*, then choose a *replicated* page according to the LRU principle. Otherwise, if there is a *clean* page whose status is *single*, then choose a *single clean* page by LRU. Otherwise, choose a *single dirty* page by LRU.

If the replaced page is dirty (the third case), then the LCM contacts the I/O manager of the disk to write back the page (Step 8). (This step is not necessary if the page is clean.) A lottery-based algorithm is used to choose a node to which to evict the page. The objective is to maintain a "near optimal" cache hit ratio with little overhead (as demonstrated in Section 5.1). Intuitively, for a node, it does not matter which other node caches the evicted page for it as long as the page can be cached by some node. Therefore, a restricted global LRU (as implemented in PGMS [12]) is not necessary due to the large overhead incurred. Instead, we use a lottery-based approach to maintain the same remote cache hit ratio with little overhead. Each node has some probability to be chosen. Each probability is proportional to the size of available memory at the node. For example, a node with 10 MB spare memory is twice likely to be chosen as a node with 5MB spare memory. In order to achieve this, each node multicasts the size of its available memory whenever such size changes by a certain percentage.

Let C be the selected node. The LCM on A forwards the node ID of C and the address and page ID of the replaced page to its DCM (Step 9). The DCM will then send the page to its peer at node C (Step 10). The DCM on C stores this page via its LCM (Step 11 — 13) and sends back an acknowledgment to its peer at A (Step 14). Finally, the DCM sends a message to its peer at the owner (node B) to update the data page cache table accordingly (Step 15, 16). Note that if node C does not have space to hold the page, this page would be simply discarded.

4 Dynamic File Management

In Dynamo, file server functionality can be migrated *dynamically* to any machine (preferably less loaded or idle machines), and thereby, automatically adapt the file management system to the available resources (CPU, memory, machines) in the LAN.

The *file manager* performs most of the traditional data management functions in Dynamo. It interacts with application(s) and cooperates with the cache manager on the same node. In general, a large amount of data is assigned to a file manager initially, but the file manager may relinquish the ownership of some data to other file manager(s) at a later time. In this section, we describe the process of initial ownership assignment via the coordinator and the exchange of ownership between two nodes in more detail.

4.1 Initial Ownership Assignment

The coordinator plays the bookkeeping role of the ownership transfer of all data via the global ownership table. Each entry consists of the ID of the owner (0 represents no owner) for each highest level homogeneous entity (as shown in Figure 3(b)). The initial table consists of all root entities; if a root entity is split (because of ownership transfer), the old entry is deleted and the new entries of its children are inserted. The same procedure follows for further splits for the ownership of an entity. If all existing children of an entity has the same owner, the entries for children entities are deleted, and the entry for the parent entity is inserted. This table structure is designed

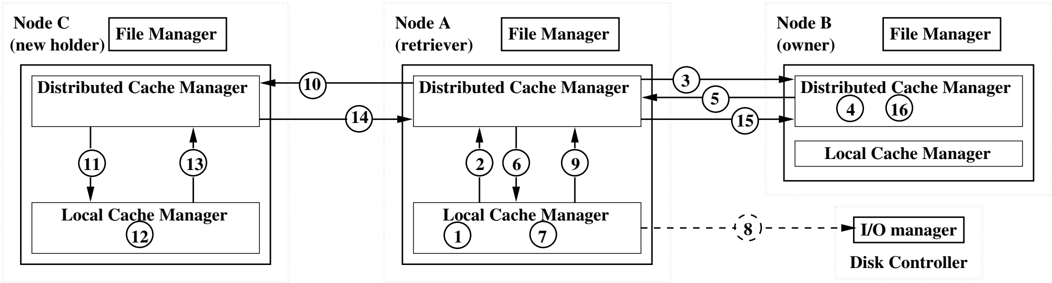


Figure 8: Page Replacement

to support the efficient splitting and merging of ownership in Dynamo.

The initial ownership assignment is performed by the coordinator (which might be central or distributed). Initially, a file manager gets a file request from an application, and determines the data page(s) on which the file is stored. The file manager tries to find the owner of the data pages via the coordinator. If the coordinator determines that the data is not 'owned' yet, it assigns the requesting file manager as the owner. In order to minimize this kind of request, the coordinator assigns a much larger partition of the data to the file manager. The coordinator identifies the data entity containing the requested data pages in the data hierarchy, and traverses the hierarchy upward to determine the largest subtree of the hierarchy containing the requested pages, which is not owned yet[6]. The owner is always responsible for managing the file-to-page-to-block information until it releases ownership either to another file manager or the coordinator.

4.2 Re-Assigning File Management at Run Time

When file manager B accesses data that is currently owned by file manager A, B and A can decide whether the ownership of the data should be transferred to B. Criteria for the ownership transfer are the workload on both machines, the intended duration of data usage, and the estimated further usage of the data by the owner. A file manager will be interested in keeping ownership of data if it expects to use the data in the near future because of shorter round-trips to find, get, and manage data than if the data would be managed by another node. If an ownership transfer is desirable, A decides to transfer ownership of a subset of its data to file manager B.

We assume that the requested entity is v'; v' is a descendant of v which is an entity that is owned by file manager A. A decomposes v into a set of child entities; one of the children contains v'. A checks whether it needs the child entity of v that includes v'. If so, A decomposes this child entity recursively until an entity v_1 is reached so that A will not use in the near future and v_1 contains v'. The ownership of v_1 is transferred to B by A. The file manager A notifies the coordinator. In turn, the coordinator starts an ownership transfer process similar to a two-phase commitensuring that B, A, and the coordinator have stable and actual information about the ownership transaction. Figure 9 shows the major steps involved in ownership transfer.

Assume that v was divided into v_1 and v_2. At a later time, if the ownership of v_2 is also transferred to B before it relinguish the ownership of v_1, B will merge them into a single entry v in its onwership table and inform the coordinator. The goal of this process is to keep a minimal list of entities in the ownership table.

5 Experimental Results

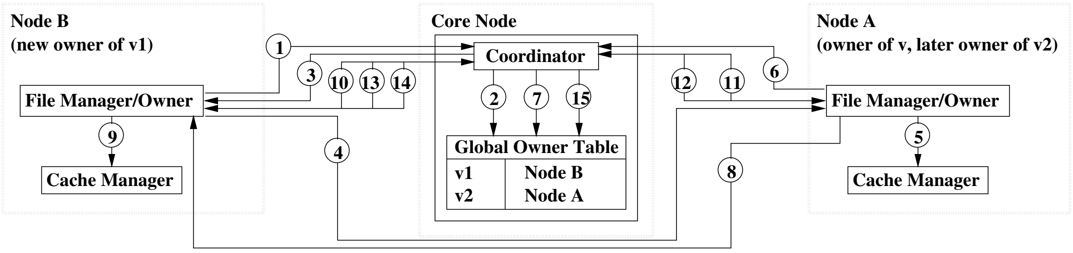
Our Dynamo prototype is implemented in C and is running on a workstation farm running the Sun Microsystems Solaris 2.6 operating system. The cluster of UltraSparc machines with 167 MHz CPU and 80 MB main memory are interconnected via 100 Mbit/sec Ethernet. The page size is 8KB and we use an 8-byte ID for each page and internal entity in the hierarchy. The cooperative cache management algorithm is implemented on top of the UNIX memory management layer. For the lottery-based page evictor algorithm, the UNIX function **random()** is used to generate random numbers. The ownership table and cache tables are implemented using a hash table. The data is stored on a 7200RPM Quantum Atlas III SCSI disks drive with 7.8ms average seek time. The disks are attached to an idle workstation to simulate the effects of network attached storage.

Since there are several contributions of Dynamo, i.e., lottery-based page eviction, migration of management, we distinguish each technique in the subscript. For example, $Dynamo_l$ denotes the Dynamo implementation with the Lottery-based eviction algorithm while $Dynamo_{l,m}$ denotes the Dynamo implementation with lottery-based eviction and migration of management techniques.

To characterize the performance of Dynamo, we used two real applications:

1. **OO7** is an object-oriented database benchmark that builds and traverses a parts-assembly database [2].

[6] At system start-up, this is a root entity within the hierarchy.



1. File Manager on node B contacts the coordinator to determine the owner of data entity v1. 2. Coordinator checks the global ownership table for the owner of v1. 3. The coordinator sends back the information that the file manager on node A is the owner of v which is the parent of v1. 4. The file manager B contacts file manager A to request the ownership of v1. 5. The file manager A decides to split v into v1 and v2 (the only fanout of v), and packages the global caching information for v1. 6. The file manager A notifies the coordinator about the impending ownership transfer of v1 to B. 7. The coordinator updates the global ownership table, splits v into v1 and v2, updates the owner informatiuon, and locks the entry for the time being. 8. The file manager A sends all necessary NAS, cache, and locking information to file manager B. 9. The file manager B updates its local cache tables, stores the NAS information, and updates the data page cache table. 10. The file manager B notifies the coordinator that it has finished updating its information. 11. The coordinator, then notifies the file manager A to delete the ownership informaiton of v1. 12. The file manager A acknowledges the deletion to the coordinator. 13. The coordinator notifies B that it is the owner of v1. 14. File Manager B acknowledges. 15. The coordinator unlocks the entries of v1 and v2 in the global ownership table.

Figure 9: Ownership Migration

Our experiments traverse an existing 650MB database mapped into memory.

2. **ScourNet** is a trace of a public web search engine "scour.net". scour.net uses an 1 GB database to answer queries.

5.1 Lottery-based Page Eviction

In this set of experiments, we are evaluating the benefits of employing the technique of lottery-based page eviction by comparing it against random eviction and the PGMS model [12].

In the random eviction scheme, when a page is evicted, the page will be evicted to any node in the LAN with the *same* probability. If the node to which the page is evicted has no available memory, then the page is simply discarded. Since each node does not have a global picture of the memory usage at other nodes, the overall cache hit ratio could be impacted when there are a significant number of nodes in the LAN without any available memory. (Pages evicted to these nodes will be discarded.)

In PGMS, the least loaded node in the system is chosen as the leader periodically. The leader receives the buffer information of all nodes, and centrally computes the replacement set of pages for each node. The length of each epoch is usually between 5 to 10 seconds. The goal of PGMS is to utilize the available global cache so that the average access latency can be minimized. However, it requires the synchronization of all nodes. Moreover, with an increasing number of nodes, the leader has to spend more CPU cycles to compute the possible number of pages can be replaced on each node and it is more likely that the number of pages which can be replaced on a node will change more rapidly and epochs have to be restarted. Therefore, a significant overhead may occur and scalability can be impacted.

In the following subsections, we investigate the cache hit ratio and response time of these schemes in detail. All experiments were performed with 5 nodes but varying memory size per node across experiments in this subsection.

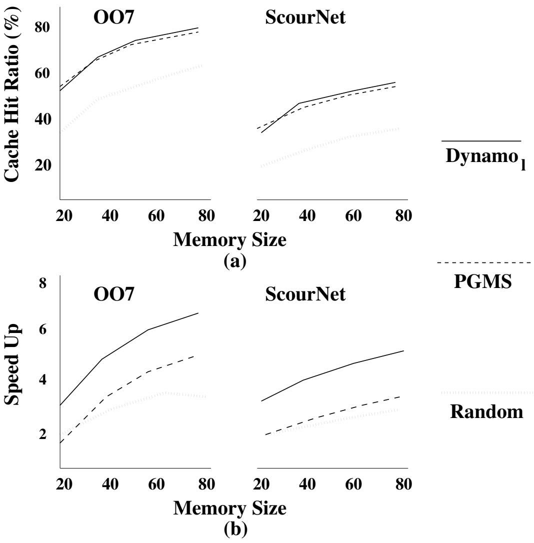


Figure 10: Three Cache Eviction Schemes

5.1.1 Cache Hit Ratio

Figure 10(a) show the average cache hit ratio. The cache hit ratios of $Dynamo_l$ and the PGMS model are similar and are significantly higher than that of the random eviction scheme. The main difference between $Dynamo_l$ and the PGMS model is the selection of the node from which a page will be evicted.

The page eviction scheme in PGMS model employs a restricted global LRU algorithm. On the other hand, in the lottery-based eviction algorithm, a set of candidate pages for replacement will be chosen. However, there is no strict order of replacing these pages. For the page evictor, it does not matter which node caches the evicted page. For the cacher, it does not matter which pages it caches for other nodes as long as it does not loose any useful pages. With the lottery-based eviction scheme, the probability that the evicted page will be discarded is very low when there is any available memory. Therefore, $Dynamo_l$ and the PGMS model have a similar cache hit ratio.

5.1.2 Average Performance

The speed up factor is chosen to describe the average response time of retrieving a data page under the three schemes, respectively. We use the non-cooperative caching scheme as the base. The speed up factor is the ratio of the average time to fetch a page in a non-cooperative caching system over the average time to fetch a page with either random eviction, PGMS model, or the lottery-based eviction scheme. Figure 10(b) shows the speed up factor of the three cooperative caching scheme with respect to the non-cooperative caching model. When the collective overall memory size is less than the working set, the speed up factor increases because the cache hits increase. As a result, fewer page faults occur[7].

5.2 Scalability of Dynamo Caching Scheme

In this section, we analyze the scalability of $Dynamo_l$, i.e., performance as the number of nodes in the LAN increases. In this set of experiments, we fix the size of memory on each node to 60MB and vary the number of nodes. Figure 11 (a) and (b) show the cache hit ratio and average performance, respectively. $Dynamo_l$ and the PGMS model have a similar cache hit ratio while $Dynamo_l$ has the best performance because of the high cache hit ratio and the low overhead.

5.3 Migration of Management

Here we examine the effects of migration of management by comparing $Dynamo_{l,m}$ (the Dynamo implementation which employs dynamic migration of management with lottery-based page eviction scheme) against the Dynamo implementation which employs the lottery-based page eviction with static partition of management, denoted as $Dynamo_l$. We evaluate the benefits of migration of management in the aspects of scalability (increasing number of applications on each node) and adaptability (changing data access pattern). In this experiment, we fix the memory on each node to 60MB and the number of nodes to 5.

[7]When the collective overall memory size is larger than the collective working sets, the cache hits remain same since all working sets can be cached in global memory.

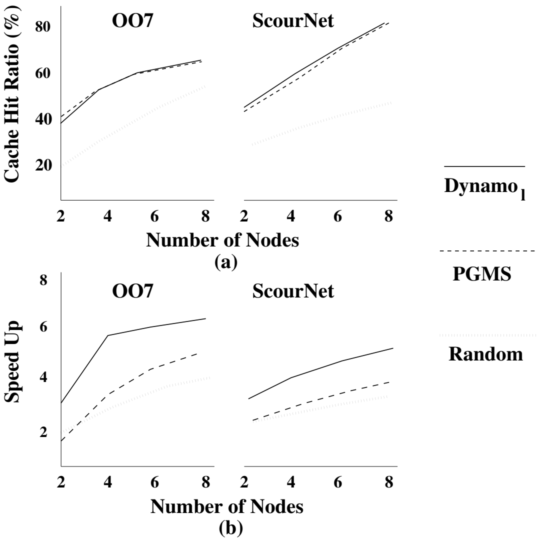


Figure 11: Scalability of Three Schemes

5.3.1 Scalability

The workload is generated as follows: an application requests data initially managed by one node with 40% probability and requests data initially managed by each of the other nodes with 15% probability. The average size of data in a request is 2KB and after receiving the requested data, it takes the application 5 seconds to process the data on average.

Figure 12(a) shows the average response time for an application to receive the data as a function of the number of applications on each node. When the number of applications is large, the average response time of data retrieval with $Dynamo_l$ is increasing at a much faster pace than that of $Dynamo_{l,m}$ because uneven distribution of management makes $Dynamo_l$ saturate faster.

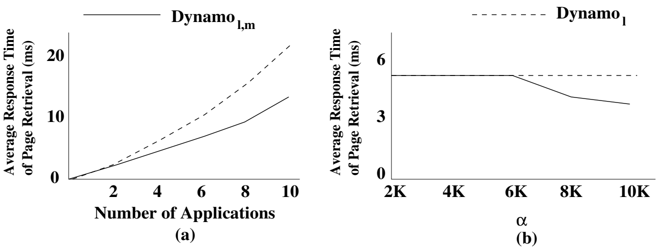


Figure 12: Effects of Management Migration

5.3.2 Adaptability

The workload in this experiment is generated in the same manner as in the previous experiment except for one difference: the heavily loaded node changes with time. For the first α requests, the heavily loaded node is A; for the next α requests, the heavily loaded node is B, then C, then D, and so on.

Figure 12(b) shows the average response time for a page request. Here α can be viewed as the duration of one page access pattern. When α is small (i.e., the hot spot changes very rapidly), migration of management does not help because the migration itself takes a significant amount of resources and time. (The migration of management is invoked only when a significant change of workload or data access pattern is detected.) When the α is larger (i.e., the change of data access pattern is more permanent), $Dynamo_{l,m}$ outperforms $Dynamo_l$ significantly due to the more even distribution of workload on each node with $Dynamo_{l,m}$.

6 Related Work

The design of Dynamo is motivated by several systems such as NASD [6] which employs 'third-party transfer' of data from network-attached storage devices directly to client machines.

Initial work on a more scalable approach to client-server architectures was done by Franklin, Carey and Livny [5]. The serverless network file system (xFS) was developed at the University of California at Berkeley [1] as part of the NOW (Network of Workstations) project to address the problem of a highly scalable file system in a distributed environment using cooperative caching.

There are two research directions related to overall global cache management: the first focuses on a centralized management component that has a central view of the caches of all clients and optimizes the page replacement strategy with respect to the needs of all clients [4, 7, 12]. The second one focuses on global cache replacement strategies that are managed by clients in a decentralized manner [4, 10]. Voelker et al. [12] proposed a globally managed prefetching and caching system, the PGMS system.

Related work in the area of cooperative caching system for object-based systems (persistent objects, object-oriented database management systems) has been done in the Shore [3] system, and in Thor [8] and Hac [9].

7 Conclusion

We have presented the complete design for a scalable, fault-tolerant, cooperative file management system for LAN environments that is dynamically adaptive to configuration changes. In addition, the design can adapt to shifting workloads and hotspots in file access. Extensive performance studies have shown that this extension can result in significant improvements in page fault response times by increasing the hit rate on remote nodes (rather than having to make a disk access).

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, and others. Serverless network file systems. *ACM Transactions on Computer Systems*, Feb. 1996, vol.14, (no.1):41-79.

[2] M. Carey, D. DeWitt, and J. Naughton. The oo7 benchmark. *Proceedings of the ACM SIGMOD Conference*, 1993.

[3] M. Carey, D. DeWitt, M. Franklin, N. Hall, et. al. Shoring up persistent applications. *Proceedings of the ACM SIGMOD Conference*, 1994.

[4] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: using remote client memory to improve file system performance. *Proc. of 1st Symp. on Operating Systems Design and Implementation*, 1994.

[5] M. Franklin, M. Carey, and M. Livny. Global memory management in client-server DBMS architectures. *Proc. of the 18th VLDB Conference*, 1992.

[6] G. A. Gibson, D.F. Nagle, K. Amiri, F. W. Chang. File Server Scaling With Network-attached Secure Disks. *Performance Evaluation Review*, June 1997, vol.25, (no.1):272-284.

[7] A. Leff, P. S. Yu, J. L. Wolf: Policies for Efficient Resource Utilization in a Remote Caching Architecture. *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS)*, 1991.

[8] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A.C. Myers, L. Shrira: Safe and Efficient Sharing of Persistent Objects in Thor, SIGMOD, 1996.

[9] M. Castro, A. Adya, B. Liskov, A.C. Myers: HAC: Hybrid Adaptive Caching for Distributed Storage Systems. Proc. of the ACM Symposium on Operating System Principles (SOSP'97), 1997.

[10] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. *Proceeding of Third Symposium on Operating System Design and Implementation (OSDI)*, 1996.

[11] Scour Search Engine. http://www.scour.net

[12] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system *Proc. of the ACM SIGMETRICS Conf.*, 1998.