Dynamo: design, implementation, and evaluation of cooperative persistent object management in a local area network

Jiong Yang[1*], Wei Wang[1], Silvia Nittel[2], Richard Muntz[2], and Vince Busam[2]

[1] *IBM T.J. Watson Research Center, 30 Sawmill River Road, Hawthorne, NY10532, USA*
[2] *Department of Computer Science, UCLA, Los Angeles, CA 90095, USA*

SUMMARY

In light of advances in processor and networking technology, especially the emergence of network attached disks, the traditional client-server architecture of persistent storage systems has become suboptimal for many computation/data intensive applications. In this paper, we introduce a revised architecture for persistent object management employing network attached storage: the dynamic object server environment (Dynamo). Dynamo introduces two main architectural innovations:(1) To provide high scalability, the object 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. Other systems have an a priori assignment of "object management" functions to specific nodes. (2) Dynamo uses a cooperative cache management which employs a decentralized lottery-based page replacement strategy with a novel technique which attempts to put a page replaced from one node into the cache of another node which is likely to access the page in the near future. 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.

KEY WORDS: Persistent object management, Cooperative caching, System architecture

Introduction

High speed local area networks and increasingly powerful desktop machines have led to the notion of clusters of PCs as servers. But servers, whether they be clusters or mainframes are configured a priori and, with growing workloads or in the case of workloads with "floating skew", a server can become a bottleneck. One approach to more scalable architecture is to use the client platforms to implement a cluster server; so each client executes not only the usual client side software but also, part of its resources are used to implement one node of a cluster server. The motivation is clearly that the "server resources" grow in direct proportion to the number of clients which leads to a scalable system. There are also challenges to be overcome to make this approach work in practice.

One challenge is to build a system that adapts to a changing configuration. Client machines may fail but also, new machines may be added or decommissioned and the system needs to automatically adapt to these changes. In addition, it is becoming more common for portable machines to be connected in the morning and disconnected in the evening. Particularly as this latter mode of operation becomes more popular

*Correspondence to: IBM T.J. Watson Research Center, 30 Sawmill River Road, Hawthorne, NY10532, USA

the system must be able to quickly and automatically adapt to configuration changes. Further challenges involve performance issues; in particular the balancing of workload across nodes. Most proposed designs for similar systems rely on a fine grained, static partitioning of data objects among nodes (e.g., hashing of data block IDs to determine the node responsible for the data block). With an unknown or changing system configuration and with shifting hot spots in data access, we suggest that a more flexible approach to adaptability is needed. Cache management presents some interesting characteristics in this type of system. For example, from the viewpoint of any given client there are three levels of memory: (a) local main memory, (b) main memory on another node, and (c) secondary storage. Working sets of individual clients can overlap in various ways. We have found, for example, that when a page is replaced from one node, its placement to another node can exploit knowledge of what relative affinity (i.e., likelihood of reference) for the page each node has in order to reduce fault rates.

Problem Statement

Here we briefly state the major assumption we make about the system environment.

1. *Data is stored on Network Attached Disks.* Network attached storage technology has been widely accepted for LAN and SAN (storage area network). Today, the price of a network-attached disk is almost the same as the PCI attached disk. Thus, LANs will increasingly employ network attached storage as their main secondary storage devices. As mentioned before, since data can be accessed almost equally efficiently from any node in a LAN, it is feasible to have a dynamic partitioning of the object management within a distributed object manager.
2. *Trusted peers.* In general, the machines in a LAN belong to the same administrative domain, thus it is reasonable to assume they can trust each other. Since security among peer nodes in a LAN is a less urgent issue, we assume that whether a node should assume the responsibility for managing a set of data can be determined solely on the basis of performance (e.g., whether this node has enough computer resources), reliability, etc.
3. *Symmetric access time.* We assume that the LAN is connected via a high speed communication network, such as Ethernet, Myrinet, Fibre Channel Arbitrated Loops, etc. In addition, the average number of nodes in a LAN ranges from ten to several hundred, and the topologies of the communication network is often straightforward. As a result, the access time for node A to access the data in node B's memory is more or less the same as to access the data in node C's memory. Moreover, the access time is several orders of magnitude less than to access data from a disk. This provides the motivation and justification for moving cached pages to other nodes' memory.

In the context of these assumptions about the computing environment, we address the design and performance evaluation of a persistent object management system in this paper. The overview section gives an overview of the proposed software architecture while the object management migration and cooperative caching sections go into the details of the design and the motivations for the design decisions. We discuss fault tolerance issues in the fault tolerance section. A description of the current prototype and the results of a detailed performance study are given in the experimental results section. The related work section discusses some previous work and a brief summary is given in the conclusion section.

Overview of Dynamo

The Dynamo architecture consists of four layers: the disk I/O layer, the cooperative cache manager layer, the object manager layer, and the coordinator layer. These four layers interact with each other as shown in Figure 1. We give an overview of the architecture from the bottom up.

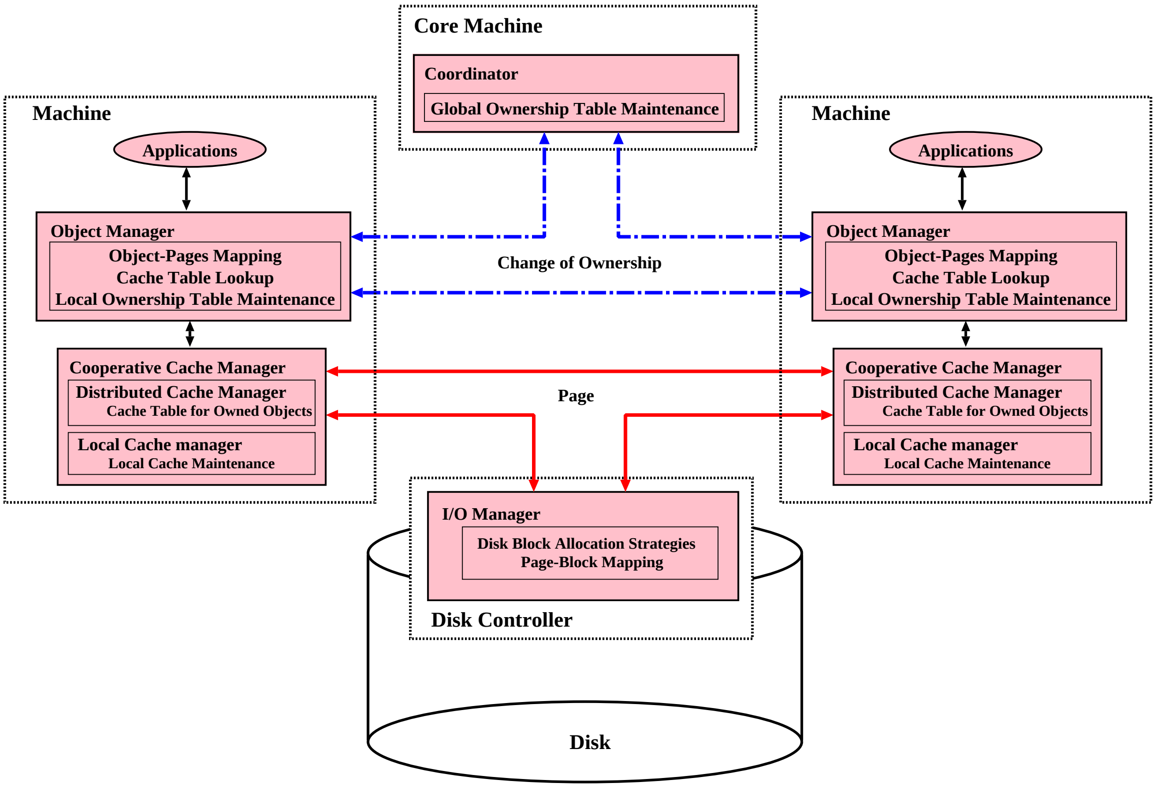


Figure 1. Dynamo Architecture

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 maps objects and data pages to storage locations on a storage device in a similar fashion to the I/O layer in a conventional object system. To obtain high bandwidth, an object may be striped over several devices (e.g., disks, disk arrays, etc.). A large object can be viewed as a sequence of pages. Different pages of an object are normally stored in contiguous physical storage space, but might also be stored on different devices if the object is very large. The disk I/O layer resides on the disk controllers of the network attached storage devices. The disk I/O layer determines the data block a page is written to. Each page is uniquely identified in Dynamo. The policy for choosing a block varies for different applications. In the UNIX environment, a log structured file system scheme might employed, so that the next block physically to the most recently written block is chosen to store the modified page. On the other hand, for database applications, it is often necessary to maintain data logically clustered to improve the retrieval time. Thus, the block allocation policy for data pages is application dependent.

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 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 cache 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 consists of two components: the local cache manager and the distributed cache manager (DCM) which collaborates with other distributed cache managers. The cooperative cache manager has two major functions: First, it performs cache replacement of its local cache. When a new page is needed and the local memory is fully utilized, existing pages have to be replaced. The replaced pages can be evicted to the global cache (i.e., to some other node's memory), evicted to secondary storage (if the page is dirty), or simply discarded (if the page is clean, and maybe replicated). A distributed cache manager defines, in concert with other cache managers, a decentralized scheme for global cache management. Using intention sets and a lottery-based scheme, the distributed cache manager determines how and whereto local pages should be evicted.

The second function of the cooperative cache manager is the mapping between the logical address of an object, and its memory address.

Object Manager Layer

In a traditional persistent object management system, a large fraction of system's resources are allocated to server functions. These functions can be broken down into storage management, buffer management, page management, etc. In Dynamo, large parts of these 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 persistent object system: the *coordinator* and the *object managers*. The coordinator is a small remaining part which runs on one or more well-know core nodes, while the object managers execute on any nodes and perform most of the traditional data management functions.

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 object 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.

Dynamic Object Management

In Dynamo, object server functionality is

migrated to nodes in the LAN instead being performed on a centralized core node (or several core nodes). The main idea of Dynamo is to migrate the management and control of data *dynamically* to any machine (preferably less loaded or idle machines), and thereby, automatically adapt the persistent object management system to the available resources (CPU, memory, machines) in the LAN.

Coordinator and Object Management

Coordinator

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.

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. The coordinator has several responsibilities: (1) keep track of all network-attached storage (NAS) space in the LAN, (2) keep track of which object manager manages which data.

The coordinator responds to an initial data request by the local object manager on a node. The coordinator identifies the data 'chunk' (or management entity) to which the requested data belongs, and whether there is already an object manager assigned to manage this data. If not, the coordinator identifies a sizable 'chunk' of data, packages all necessary data management information for that data so that the object manager can independently perform all object management functions (e.g., consistency control, etc.), and sends this information to the requesting object manager. When the workload on the local node is too high and/or local application does not need some specific data any longer, the object manager can relinquish the data management responsibility for selected data either to the coordinator or to another node.

Object manager

The *object 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 Dynamo, an object manager does not manage a fixed portion of the system's data as in other systems. Instead, a dynamically chosen partition of the overall data is assigned to an object manager. The size of the data partition depends on the overall workload on the node. In general, a large amount of data is assigned to an object manager initially, but the object manager may relinquish the ownership of some data to other object manager(s) at a later time.

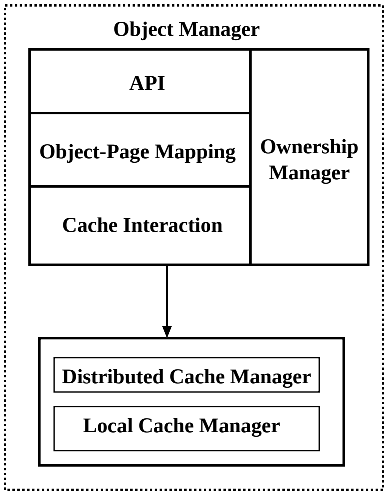


Figure 2. Detailed Object Management Layer

An *ownership manager* is the single module that handles data management for a specific data partition. In Dynamo, an object manager always performs application interaction, object to page mapping, and cache interaction, but it might only optionally run an ownership manager only when it owns some data (see Figure 2). If an object manager does not manage any data, then it does not have an ownership manager. The ownership manager also grants and manages access locks on the data. It is the enforcer of consistency control for the data which it owns. If exclusive access (e.g., write lock) or shared access (e.g., read lock) is required, then the owner of the data is in control of granting these permissions and also takes care of lock scheduling, dead-lock detection, etc. Due to space limitation, we will not discuss consistency control in detail here.

When an application writes, inserts or deletes data in the data partition that an owner manages, the ownership manager updates the secondary storage location information, and requests physical storage management from the NAS. In the case that the overall workload of a node increases and/or an increasing number of applications try to access data owned by an object manager, the owner can decide to relinquish parts of the management functions to other nodes in order to adapt to the environment.

Dynamo employs a data organization which supports efficiently relinquishing and re-assigning ownership management functions. We will describe the data organization in the next section.

Data Organization to Support Dynamic Object Management

To support the dynamic data management, Dynamo employs a data organization scheme that supports the flexible assignment of object management to nodes.

Data in Dynamo is viewed in two ways: the system's internal view and the users' view. The users' view is identical with the user interface of traditional persistent object management systems. Users can create, access and update persistent objects, and group persistent objects into clusters. A persistent object is identified via a logical descriptor (names, or persistent IDs). Persistent objects are mapped to data pages; several objects might fit onto one page, or an object can be stored on several data pages. The internal view, however, is specifically designed to support dynamic object management functions, and is based on data pages.

Internally, the entire data universe is viewed as sets of sets of data pages each set being uniquely identified by an internal identifier that is not known to the user. The data universe is organized in a hierarchical manner as shown in Figure 3. At the bottom level, a granule is a page, i.e., each entity represents a page. A page usually consists of 8 KB data. At the level above, each entity represents a set of pages with a varying number of pages per set. A granule at the next level up is called a *cluster*, a *set of clusters*, set of sets of clusters, and so on. The top level contains a few to a few hundred root entities each root entity representing a large set of data pages. This structure is used to group and identify pages and groups of pages. These groups are disjoint, i.e. a page can only belong to one group. Also, re-grouping functions are not supported for the time being.

Each entity in the hierarchical structure is uniquely identified. To keep track of the parent-children relationship within the hierarchy, 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 "1123000"; its children can be identified by keeping the same "1123" prefix in their identifier, i.e. an

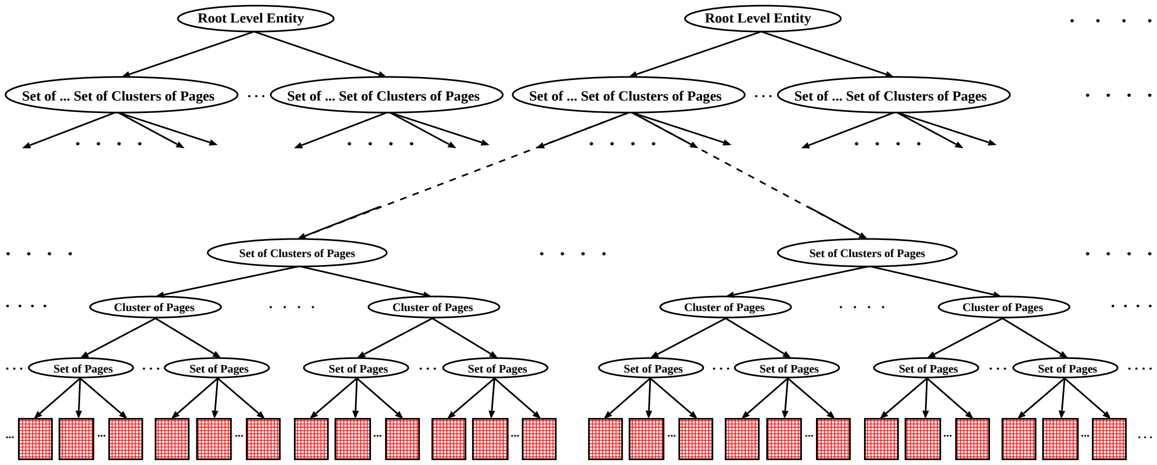


Figure 3. Hierarchical Data Organization

entity with the identifier "1123210" is a descendent of the former.

The prefix naming structure can support the flexible ownership management efficiently. Object managers and coordinators keep lists of entities† (in the hierarchical view). If no data page exists for an entity, then this entity is null. We use a field in an entity to record the exact number of children which its parent has. If an owner recognizes that it owns *all* children of an entity, then it merges these entities, and becomes the owner of the parent entity. Overall, the hierarchical data structure of helps to significantly compress the ownership management information. Without this structure, then there would be one entry for each page, resulting in a vast size of the management data.

In Dynamo, all pages of persistent object and all persistent objects of a user-defined cluster are physically grouped together. A user can choose which persistent objects should be put next to each other in the hierarchy. Once an ID is chosen for a data page, it remains the same for the lifetime of that page.

Each internal entity of the data organization structure is stored within one data page. Each entity has at most 255 entries; an entry for an entity consists of the node's data page id and the disk id of this page's storage location. To find the actual storage page location, an owner has to traverse the data hierarchy. It starts with the page storing the locations of all root entities, and finds the pages for the relevant root entity's children, etc. Decomposing an object identifier, the position of the relevant entry in the page can be directly accessed (between 1 and 255). Each entity in the hierarchy occupies one 8KB page. The maximum number of pages to be read before the data page is reached is the height of the hierarchy. The average number of disk I/O, which have to be performed, is much less than this because many internal entities may be cached by the coordinator and the object managers.

†An entity corresponds to an internal node in the hierarchical data organization.

Assignment and Release of Object Ownership

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.

Initial ownership assignment

As described, the initial ownership assignment is performed by the coordinator (which might be central or distributed). The coordinator is the only component that knows about all data in the system, about owners, and which data is owned by which owner. It manages the global ownership table entries which consist of the IP address of the owner (0 represents no owner) for each highest level homogeneous entity‡ (as shown in Figure 4). 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 to support the efficient splitting and merging of ownership in Dynamo.

entity	IP address
13400000	131.179.99.79
12000000	131.179.99.69
20000000	0
.	.
.	.

Figure 4. Global Ownership Table

Initially, an object manager gets an object request from an application, and determines the data page(s) on which the object is stored. The object 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 object manager as the owner. In order to minimize this kind of request, the coordinator assigns a much larger partition of the data to the object manager; however, since only parts of the additional data will be accessed, the object manager gets only a minimal additional workload. 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. At system start-up, this is a root entity within the hierarchy. The object-to-page-to-block mapping information table for the whole entity is stored on a NAS disk, and the coordinator

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

sends the storage location of this mapping table to the object manager which is the owner. Additionally, it enters the IP address of the node into the global ownership table determining the ownership relationship for the data entity.

Although, an owner is responsible for managing any further updates to the object-to-page-to-block information table, it will regularly update the table to the previous NAS disk to provide for stable information. The owner is responsible for writing back the mapping information to external storage before it releases ownership either to another object manager or the coordinator.

Re-Assigning Object Management at Run Time

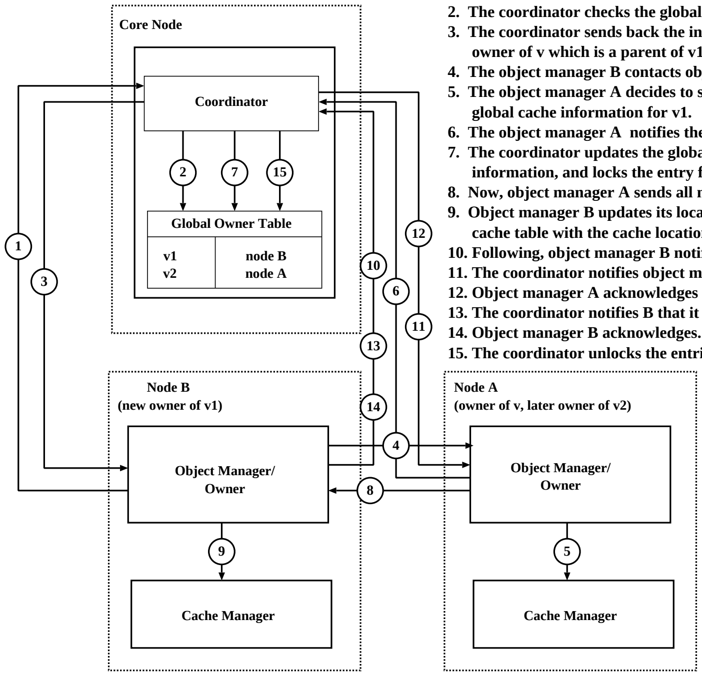
When object manager A accesses data that is currently owned by object manager B, A and B can decide whether the ownership of the data should be transferred to A. Criteria for the ownership transfer are the workload on both machines, the intended duration of data usage, and the estimate further usage of the data by the owner. An object 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, B decides to transfer ownership of a subset of its data to object manager A. The ownership migration process is illustrated in Figure 5.

We assume that the requested entity is v'; v' is a descendant of v which is an entity that is owned by object manager B. B decomposes v into a set of child entities; one of the children contains v'. B checks whether it needs the child entity of v that includes v'. If not, B transfers the ownership of the child entity to A. If there are locks on parts of the hierarchy, the locks are also transferred to the new owner as part of the ownership transfer.

If the owning object manager needs parts of the child entity, the owner decomposes the child entity. The process is repeated until B finds an entity that is a descendant of v, contains v', and which B estimates it will not use in the near future. The ownership of this entity is transferred to A by B. If such an entity does not exist, no ownership transfer will occur.

If object manager B wants to transfer the ownership of v_1 to A, it notifies the coordinator. In turn, the coordinator starts an ownership transfer process similar to a two-phase commit ensuring that A, B, and the coordinator have stable and actual information about the ownership transaction. This is done as follows: B intends to release entity v_1 to A. Therefore, object manager B decomposes v into a set of disjoint entities v_1, v_2, …, v_k where $\bigcup_{i=1}^{k} v_i = v$. B removes v_1 from this set, and sends all information associated with v_1 (e.g., global cache information, NAS location information, and lock information) to A.

Object manager A updates its ownership table by adding v_1. It prunes its ownership table to see whether it also owns v_2, v_3, …, v_k. If it owns these data entities, A would remove v_1, v_2, …, v_k from its ownership table and put back v and continue the pruning process until no more sub-entities can be removed. The goal of the pruning process is to keep a minimal list of entities in the ownership table. After receiving the notification of ownership change, the coordinator updates its global ownership table



1. Object manager on node B contacts the coordinator to determine the owner of data entity v1.
2. The coordinator checks the global cache table for the owner of v1.
3. The coordinator sends back the information that the object manager on Node A is the owner of v which is a parent of v1.
4. The object manager B contacts object manager A to request ownership of v1.
5. The object manager A decides to split v into v1 and v2 (the only fanout of v), and packages the global cache information for v1.
6. The object manager A notifies the coordinator about the impending ownership transfer of v1 toB.
7. The coordinator updates the global ownership table, splits v into v1 and v2, updates the owner information, and locks the entry for the time being.
8. Now, object manager A sends all necessary NAS, cache and lock information to object manager B.
9. Object manager B updates its local tables, stores the NAS information, and updated its distributed cache table with the cache location of the newly owned data of v1.
10. Following, object manager B notifies the coordinator that it finished updating its information.
11. The coordinator notifies object manager A to delete the ownership information of v1.
12. Object manager A acknowledges the deletion to the coordinator.
13. The coordinator notifies B that it is now the owner of v1, and
14. Object manager B acknowledges.
15. The coordinator unlocks the entries of v1 and v2 in the global ownership table.

Figure 5. Ownership Migration

using essentially the same procedure as A and B. Figure 5 shows the major steps involved in ownership transfer. During the transfer process, B is considered as the owner of the data. However, any request to the data being transferred is blocked until the transfer process is completed. The standard two phase commit recovery is used if during the transfer, A, B, or the coordinator crashes.

Cooperative Caching

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

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 6.

1. Cache Discovery. As the name indicates, 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. This procedure is invoked 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.

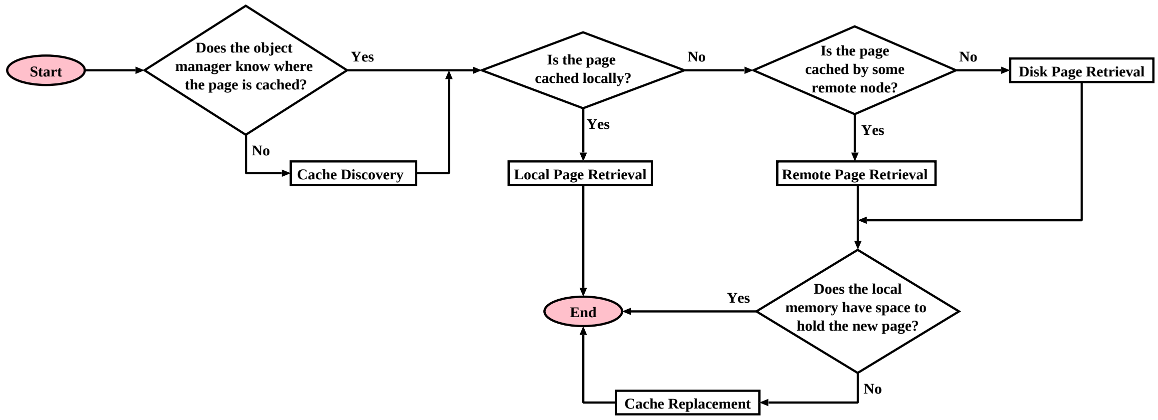


Figure 6. State Diagram of Page Retrieval

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

Cache Discovery

Figure 8 describes the operations involved in cache discovery. Assume that an application on node A tries to access data owned by the object manager on node B. The application first contacts the local object manager with the ID of the desired object and the offset, length of the data in the object (Step 1). For example, an application may ask for 100 bytes of data starting at the 200th byte of object "foo". The object manager first performs object to page mapping and identifies the desired page(s). Then the object 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). The coordinator of the data is always on a well known core node. This can be achieved by assigning data pages to coordinators based on page IDs, e.g., hashed page ID. The coordinator, then, looks up its *global ownership table* (as illustrated in Figure 4) to identify the owner (Step 4), and returns the ID of the owner and the highest level homogenous entity which contains the data page to the object manager on A (Step 5). For example, if the requested data page is '1234211' and the object manager B owns '123000', then object manager B will return '123000' to A. Note that if the owner of some data has become known to the local object manager previously, the above process is omitted. If object manager A requests data page '1234111' later, it does not need go to the coordinator for the ownership information. Then, the object manager on A sends a message to its peer who owns the data (i.e., the object manager on B) to request the proper lock[§] and ask for the ID of the node where the desired data is cached (Step 6). If the desired data is large, then there might be multiple owners, each of which owns part of the data. Without loss of generality, we assume that only one owner owns the requested data.

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 distributed cache manager. 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 7, each entry in the data page cache table consists of four fields: *pageID*, *status*, *cached node ID* (represented by the IP address of the node), and *intention set*. 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*.

pageID	status	nodeID	Intention Set
00112233	single	131.179.99.79	131.179.99.79
00331100	replicate	131.179.99.79 131.179.99.59 131.179.99.49	131.179.99.59 131.179.99.49
⋮	⋮	⋮	⋮

Figure 7. Data Page Cache Table

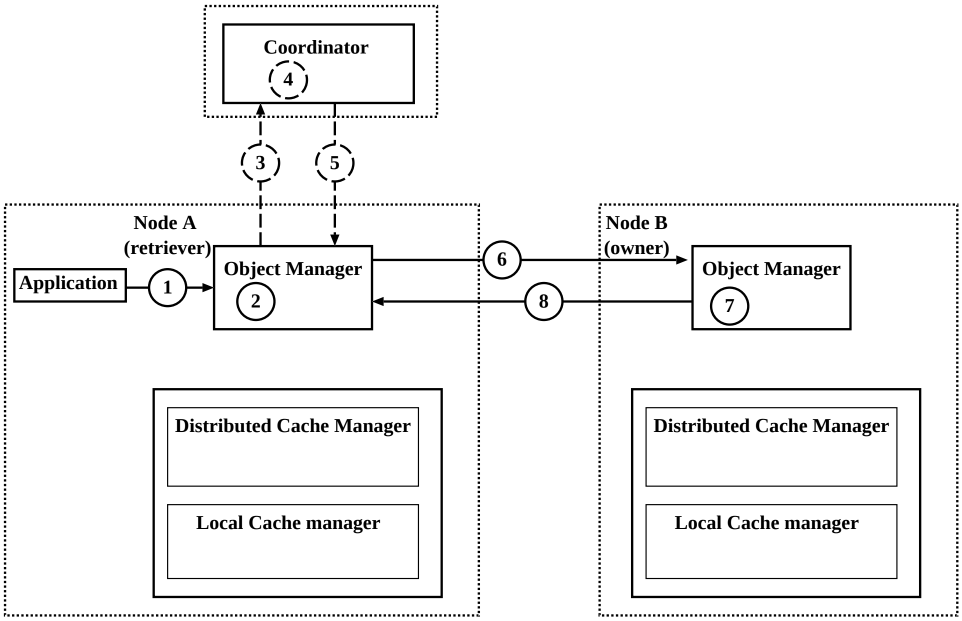
If the desired page is cached, then the owner will return the IP address of the node which holds the cached page (Step 8). Otherwise, the ID of the disk which stores the page will be returned with the page ID.

This cache discovery procedure is invoked if the object manager on A does not know where the page is cached. However, if this information is already known by the local object manager, it will directly retrieve the page as described in the following subsections.

Local and Remote Cache Page Retrieval

If the page resides in the local memory, then the object manager requests the local cache manager (via the DCM) to locate and load the page into the application user space. This is shown in Figure 9.

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



1. The application sends a request to the object manager.
2. The object manager checks its local cache to see whether it already knows the owner.
3. If the owner is unknown, the object manager on node A sends a request to the coordinator to obtain the address of the onwer of a page.
4. The coordinator checks the global ownership table to locate the owner.
5. The coordinator returns the ID of the owner (e.g., node B) to the object manager on node A.
6. The object manager on node A sends a request to the object manager on node B to ask which node has a cached copy of the page.
7. The object manager on node B checks the cache table to find the node which caches the page.
8. The object manager on node B returns the ID of cache holder to the object manager on A if the page is cached at some node. Otherwise the ID of the disk which stores the page is returned.

Figure 8. Cache Discovery

If the page resides on other node, the remote cache retrieval procedure has to be invoked. The distributed cache manager plays an important role in retrieving a remote cached page. The entire procedure is illustrated in Figure 10.

After determining the location of the cache holding the desired page, the object manager forwards the page ID and the node ID of the cache holder to its distributed cache manager (Step 1). The distributed cache manager 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 distributed cache manager on node C contacts its local cache manager to locate the page in its memory (Step 3 — 5), and then sends back the page to its peer on

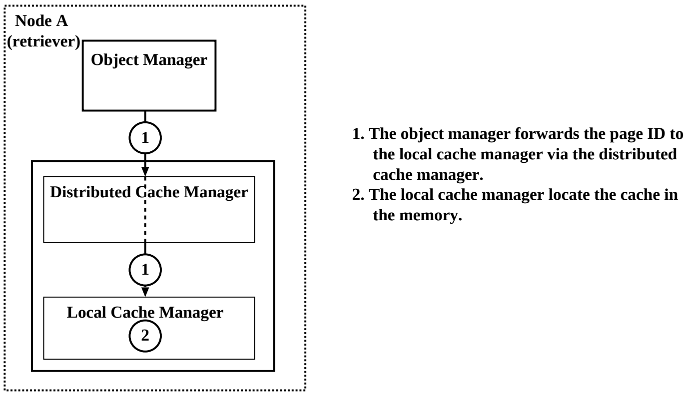
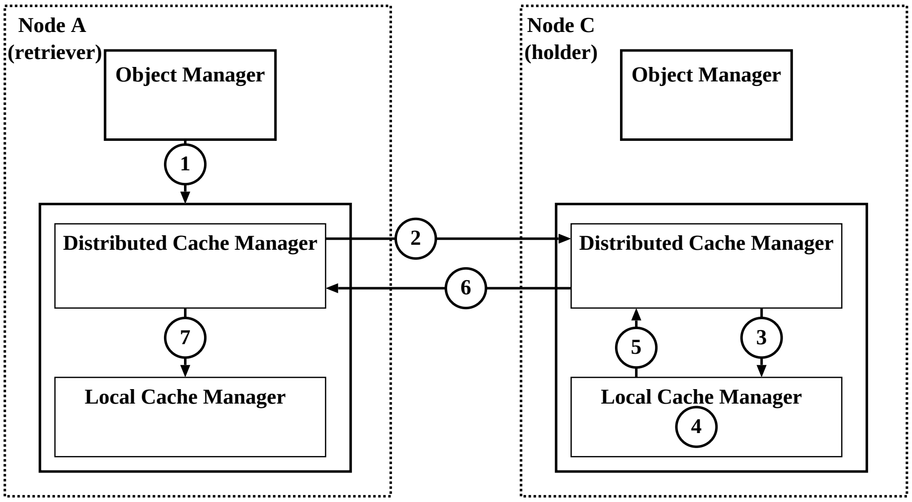


Figure 9. Retrieve a Page Locally



1. The object manager on node A forwards the page ID and the node ID of the cache holder (e.g., node C) to the distributed cache manager on node A.
2. The distributed cache manager on node A sends a request (with the page ID) to the distributed cache manager on node C.
3. The distributed cache manager on node C asks its local cache manager for the page.
4. The local cache manager on node C locates the page in its memory.
5. The local cache manager on node C returns the address of the page to the distributed cache manager.
6. The distributed cache manager on node C returns a copy of the page to the distributed cache manager on node A.
7. The distributed cache manager store the copy in the memory through the local cache manager.

Figure 10. Retrieve a Page from a Remote Node

node A (Step 6). Finally, the distributed cache manager on A stores the page in local memory via its local cache manager (Step 7).

Disk Page Retrieval

As illustrated in Figure 11, if the page is not cached, the object manager forwards the page ID and disk ID to the local cache manager via the distributed cache manager (Step 1). The local cache manager 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 local cache manager of the retriever (Step 4).

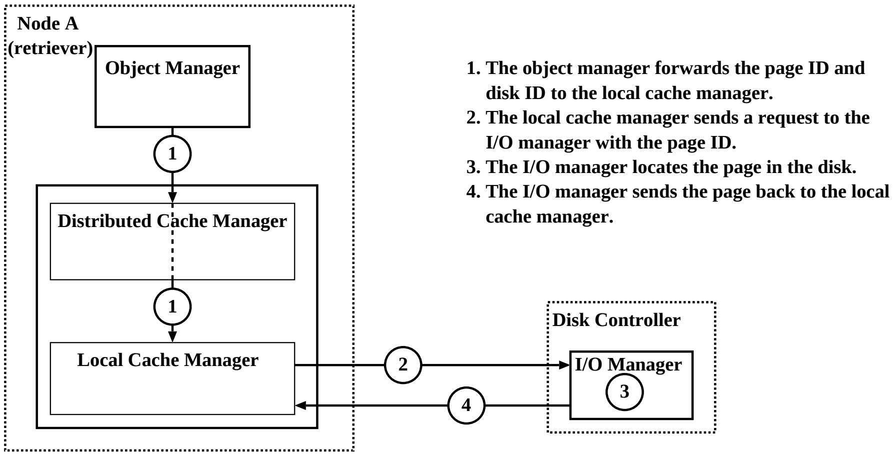


Figure 11. Retrieve a Page from Disk

Every time a page is fetched from a disk or a remote node, the local cache manager stores the page in its memory. However, if the local memory is full, some page has to be replaced. This page replacement procedure is discussed in the next subsection in detail.

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 12.

First the local cache manager on A first generates the local candidate set S for page replacement according to rules such as LRU (Step 1). All pages in S are replaceable. Let B be the node which owns the most pages in S. Then the local cache manager on A forwards to its distributed cache manager the ID of B and the set of pages owned by B in the candidate set S (Step 2). The distributed cache manager on A, in turn, sends a request to its peer on node B to ask the status and the intention sets of these pages (Step 3).

The intention set of a data page consists of the nodes which have accessed the page more than a certain number of times within a short period (e.g., past 10 minutes).

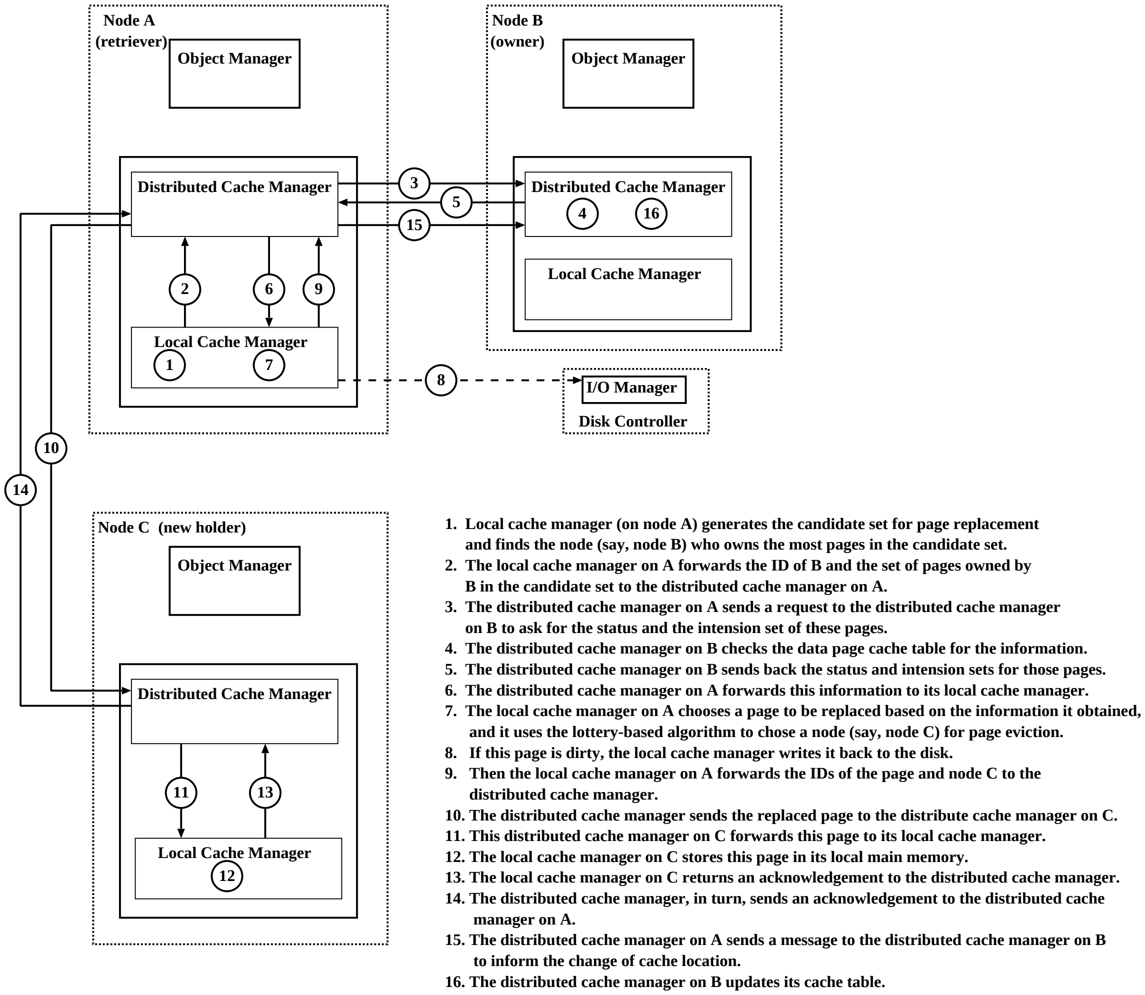


Figure 12. Page Replacement

The goal is to improve the local cache hit ratio of nodes within the intention set (as shown in the experimental results ection). Intuitively, nodes in the intention set of a data page are those which tend to access this page frequently. For any node, it is preferable to cache locally the pages which it accesses frequently. In Dynamo, when the local cache manager decides to replace a page, this page will be evicted to the memory of a node in the intention set if possible. The motivation is that nodes in the intention set have higher probability to access the evicted page than other nodes and thus eviction to a node in the intention set will tend to increase the probability of a local cache hit. The intention set information is maintained in the data page cache table by the object manager who owns the page. As we mentioned before, every access to a page has to go through the object manager (to request a proper lock), so little additional overhead is required for the object manager to maintain the intention set. For any page, its intention set could be empty or consist of one or more nodes. If the intention set is empty or no node in the intention set has available memory, then this page is evicted to some other node¶.

When the distributed cache manager on B receives a request on status and the intention set of pages, it will check the data page cache table and return back the requested information (Step 4, 5). In turn, the distributed cache manager on A forwards this information to its local cache manager (Step 6). The local cache manager 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 local cache manager 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.) Let IS be the intention set of the replaced page. A lottery-based algorithm is used to choose a node in IS to which to evict the page. The objective is to maintain a "near optimal" remote cache hit ratio with little overhead (as demonstrated in the experimental result section). 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 [22]) 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 in the intention set 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 local cache manager on A forwards the node ID of C and the address and page ID of the replaced page to its distributed cache manager (Step 9). The distributed cache manager will then send the page to its peer at node C (Step 10). The distributed cache manager on C stores this page via its local cache manager (Step 11 — 13) and sends back an acknowledgment to its peer at A (Step 14). Finally, the distributed cache manager 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. The experimental results section presents results from a performance study (measurement and simulation) of these procedures which demonstrates their effectiveness.

¶Note that this scheme can be easily fit into any general scheme which supports cooperative caching.

Fault Tolerance

In a LAN system, any node may fail at any time. Therefore, fault tolerance and failure recovery becomes a crucial issue in Dynamo. Coordinators are only located on core nodes, but object managers and cooperative cache managers may be located on any nodes. We consider core node failure and non-core node failure separately. We use a replicated coordinator scheme to guard against core node failure. In other words, the set of data managed by coordinators overlaps. For a given data item, there is a prime coordinator and a secondary coordinator, which are located on different core nodes. When the prime coordinator fails, the secondary coordinator will assume the responsibility of the prime coordinator. This requires that when ownership transfer occurs, both coordinators need to participate in the process. The object manager and cache manager may also exist on core nodes, the recovery of the object manager and cache manager on core nodes is the same as the recovery process of these managers on non-core nodes.

Another type of failure node is a non-core node. In this case, the object manager, the distributed cache manager, and the local cache manager on that node are no longer available. However, other surviving nodes may still request the data owned by the crashed object manager, and of course, these requests can not be served. We apply a crash recovery protocol to handle this situation. When an object manager requests data from another object manager, it has a timeout mechanism. If the object manager does not respond within the timeout period, the requesting object manager will report to the coordinator that the other object manager could have crashed. Then, the coordinator will contact the specified object manager. If the object manager still does not respond, the coordinator will revoke all ownership of the specified object manager, and assign its entities to the non-owner table. Also during normal operation, each object manager maintains logs and periodically writes new log entities to network attached storage so that during a crash, the coordinator can fetch the log (the location of the log on the disks is known to the coordinator) and roll back the changes to the last consistent state. This procedure is similar to that used in distributed server failure recovery.

Since the entire node crashes, all information maintained on that node is assumed to be unavailable. If a surviving node wants to access an object owned by the crashed node, then it has to go through the coordinator and retrieve the data page from disks.

When a node recovers from a crash, the object manager on that node does not own any objects initially. As time goes on, the object manager may regain ownership of some objects and the cooperative manager may rebuild the cache through normal data access.

Experimental Results

Implementation Issues

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. This is the environment in which we conducted our experiments.

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. To implement the intention set, each object manager keeps track of the number of requests for each object from each node. Time is divided into 10 minutes epochs. At the end of each epoch, the object manager reevaluates the intention set for each object. A node will be put into the intention set of an object if one of the following two criteria are satisfied.

1. The node has requested some access lock for (part of) the object more than 10 times in the past epoch.
2. The node has held the lock for (part of) the object more than 5 minutes.

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, intention set, migration of management, we distinguish each technique in the subscript. For example, $Dynamo_{l,i}$ denotes the Dynamo implementation with the Lottery-based eviction algorithm and intention set technique while $Dynamo_{l,m}$ denotes the Dynamo implementation with lottery-based eviction and migration of management techniques.

Path Length Measurement

In this section, we present measurements of the path lengths of different operations in Dynamo. All the path length measurements are the average of 50 iterations. Table I, shows the average path length of each operation in cache discovery process. The operation label corresponds to the label in Figure 8. The latency between any two nodes is approximately $35\mu s$.

Operation Label	Node	CPU time (μs)	Net (μs)	total (μs)
2	retriever	1	35	36
3	coordinator	3	0	3
4	coordinator	1	35	36
5	retriever	1	35	36
6	owner	3	0	3
7	owner	1	35	36

Table I. Path length of cache discovery process

After discovering where the cache is, the remote page retrieval process (as illustrated in Figure 10) is invoked if there exists a cached copy of the data in remote memory, but not local memory. Table II shows the average path length of each step in the remote page retrieval process.

Operation Label	Node	CPU time (μs)	Net (μs)	Total (μs)
1	retriever	2	0	2
2	retriever	2	35	37
3	holder	2	0	2
4	holder	2	0	2
5	holder	1	0	1
6	holder	14	785	799
7	retriever	78	0	78

Table II. Path length of remote page retrieval

When a new page has to be loaded into main memory, some page has to be replaced if the memory is fully utilized. The page replacement process was described in Figure 12. Table III shows the average path length of operations involved in the page replacement process.

Operation Label	Node	CPU time (μs)	Net (μs)	Total (μs)
1, 2	retriever	5	0	5
3	retriever	1	35	36
4	owner	3	0	3
5	owner	1	35	36
6, 7, 9	retriever	6	0	6
10	retriever	15	788	803
11, 12, 13	new holder	44	0	44
14	current holder	2	34	36
15	retriever	5	35	40
16	owner	8	0	8

Table III. Path length of page replacement

From the above tables, we can see that the majority of the elapsed time is spent in sending the data page (8KB) across the network. We used a 100 Mb/s Ethernet in this experiment, but faster networks (e.g., Myrinet, Fibre Channel, etc) which exceed 1 Gb/s bandwidth are becoming common place now, and the overall access time will be much faster as these are deployed.

Benchmarks

To characterize the performance of Dynamo over a wide range of workloads, we used a number of real applications and synthetic benchmarks, and ran them on the Dynamo prototype:

1. **OO7** is an object-oriented database benchmark that builds and traverses a parts-assembly database [5]. Our experiments traverse an existing 650MB database mapped into memory.

2. **Data-mining** application is an association discovery program applied to a 200MB market basket data file. Our experiments perform several full sequential scans of the entire data set. Between each consecutive scans, the confidence and support of each potential association rule are computed [2]. This is a data and compute intensive program.
3. **ScourNet** is a trace of a public web search engine "scour.net". scour.net uses an 1 GB database to answer queries.
4. **Random** randomly access all pages in a 200MB file.

Lottery-based Page Eviction and Intention Set

In this set of experiments, we are evaluating the benefits of employing the technique of lottery-based page eviction and intention set by comparing it against random eviction and the PGMS model [22].

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. However, 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, this approach has some potential problems: theoretically, leader election on a general distributed computer systems is impossible [17]. Although the leader may be elected successfully in the current local area network environment with very high probability, 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. In addition, with a larger node population, 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. To estimate the overhead for the PGMS model, we assume that all nodes have to send a message about its cache status (e.g., when the last time it is accessed) of every page in the cache to the leader at the beginning of each epoch and the leader has to scan at least one of these messages once. For example, if each node has 64MB cache, then it has to send a 8KB message to the leader (assuming it takes one byte to encode the cache status of a cached page and the page size is 8KB). 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 8.3.

Cache Hit Ratio

There are two kinds of cache hits in a cooperative caching system, local cache hit (when the desired data page is located in the local cache) and remote cache hit (when the desired data page is located in the remote cache). Figure 13(a) and (b) show the average local and remote cache hit ratio, respectively.

For the four workloads investigated here, LRU is used as the local page replacement algorithm. It is evident that $Dynamo_{l,i}$ has a significant higher local cache hit ratio than the other two schemes except for the random page access workload. The improvement mostly comes from intention set technique because it is more likely that a page is evicted to a node that might access the page in the near future (which will result a local cache hit). However, in the random page access workload, since every node has the same probability in accessing each page, then there is no benefit to use the intention set technique.

The remote cache hit ratio is the ratio of the number of remote cache hits over the number of remote cache requests. With the OO7 and ScourNet workloads, the remote cache hit ratio of $Dynamo_{l,i}$ and the PGMS model is similar. The page eviction scheme in PGMS model considers the page access pattern globally. It computes a queue of candidate pages for replacement on a centralized machine, based on the latest reference time (i.e., global LRU algorithm). As a result, when a page needs to be replaced from the global cache, the page at the head of the queue will be chosen. 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. The main difference between $Dynamo_{l,i}$ and the PGMS model is the selection of the node from which a page will be evicted. 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, $Dynamo_{l,i}$ and the PGMS model have a similar remote cache hit ratio.

Moreover, the remote cache hit ratio of $Dynamo_{l,i}$ and the PGMS model is higher than that of the random eviction scheme. If half of nodes have available memory while the other half have no available memory, with the random eviction scheme, a page will be evicted to a node which has no available memory with 50% probability, and thus the page will be discarded from the global cache with 50% probability. However, with the PGMS and $Dynamo_1$ scheme, the page will be evicted to a node with available memory, and hence, the page will be kept in the global cache.

With the data mining workload, when the memory is small (i.e., the entire data set can not be fit into the global cache), the PGMS model and $Dynamo_{l,i}$ scheme have the similar remote cache hit ratio which is higher than the random eviction scheme due to the same reason as in the ScourNet and OO7 case. When the memory on each node is large enough to hold the working set in the global cache, the remote cache hit ratio becomes constant.

However, with the random page access workload, all three schemes have a similar remote cache hit ratio because all pages have the same probability to be accessed in the

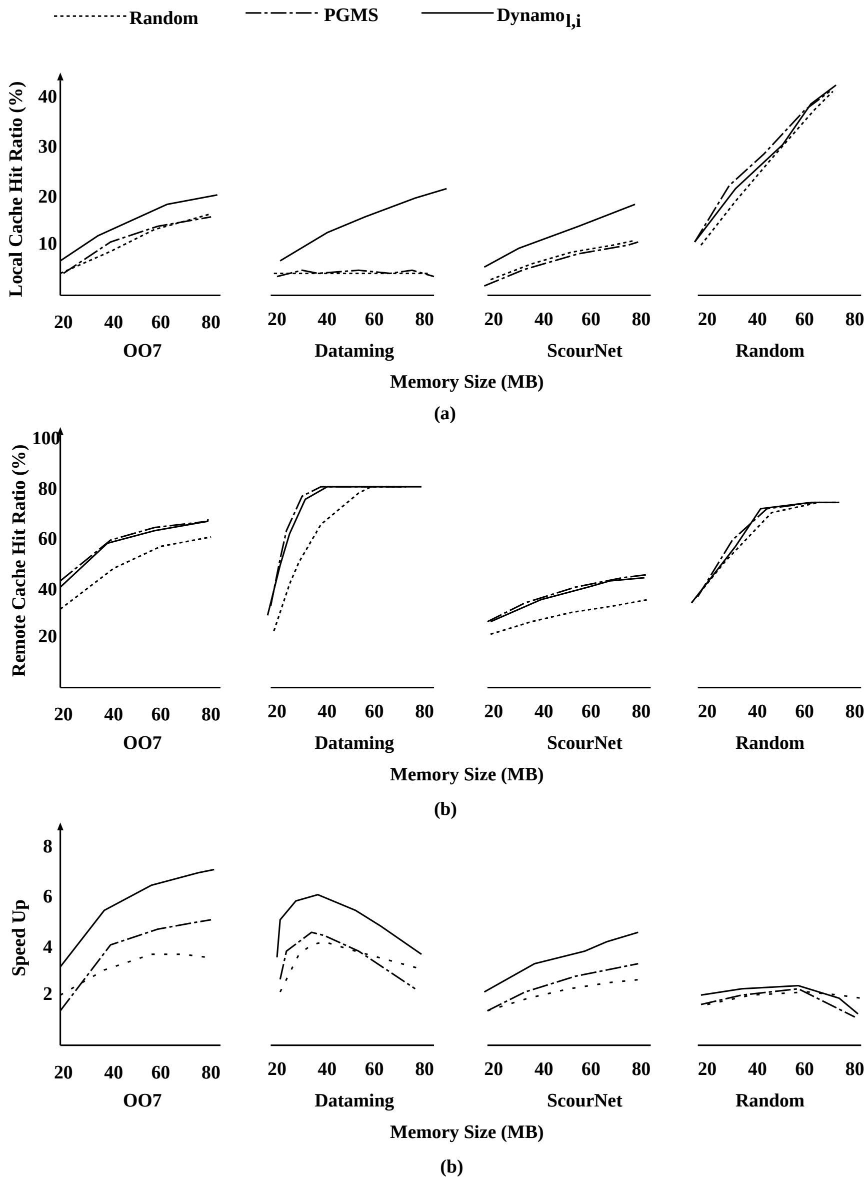


Figure 13. Comparison of Three Cache Eviction Schemes

near future. As a result, it does not make any difference on which page is replaced out from the global cache. When the collective working set size is less than the collective memory on all nodes, the remote cache hit ratio remains constant with respect to the increase of memory because the entire working set can be fit into the global cache.

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 13(c) shows the speed up factor of the three cooperative caching scheme with respect to the non-cooperative caching model.

The speed up factor changes with memory size on each node. When the collective overall memory size is less than the working set, the speed up factor increases because the remote cache hits increase. (The remote cache hits are one important benefit of cooperative caching.) As a result, fewer page faults occur. However, when the collective overall memory size is larger than the collective working sets, the overall cache hits remain same since all working sets can be cached in global memory. With more memory on each node, more data can be cached locally, and less data is cached remotely. Thus the benefits of cooperative caching decrease. As a result, the speed up factor decreases. This is illustrated in the data mining and random page accessing workload.

In general, when the collective memory of all five nodes is less than the working set, the random eviction scheme has the lowest speed up factor because the random scheme has the lowest overall cache hit ratio. Although $Dynamo_{l,i}$ has a slightly lower overall cache hit ratio compared to that of the PGMS model, it has the highest speed up factor due to its lower overhead than that of PGMS model in this scenario.

Individual Effects

Figure 13 shows the combined benefits of intention set and lottery-based page eviction techniques. We choose the ScourNet workload as the example to show the individual effects of intention set and lottery-base page eviction schemes (as illustrated in Figure 14).

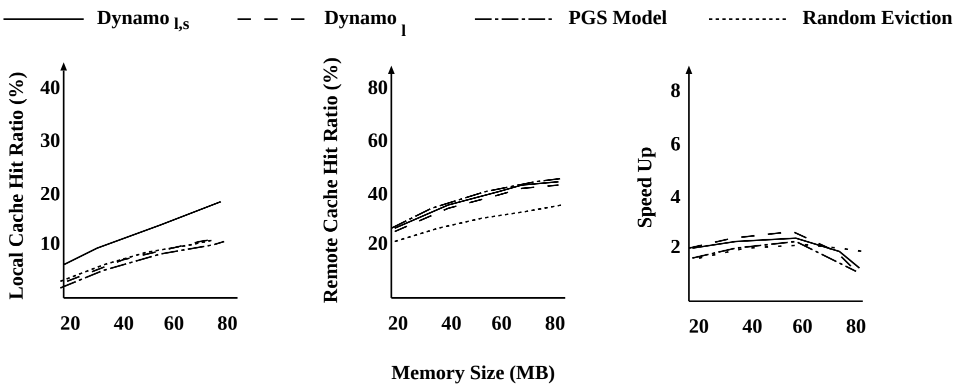


Figure 14. Individual Benefits of Lottery-base Page Eviction and Intention Set

We can see that the lottery-based page eviction scheme can achieve a similar remote cache hit ratio as the PGMS model as the reasons explained before, however, it has a lower overhead as the PGMS model. On the other hand, the intention set has a significant impact on the local cache hit ratio. As a result, on average $Dynamo_{l,i}$ outperforms $Dynamo_l$, which slightly outperforms the PGMS model.

Scalability of Dynamo Caching Scheme

In this section, we analyze the scalability of $Dynamo_{l,i}$, 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 15 (a) and (b) show the overall cache hit ratio[||] and average performance, respectively.

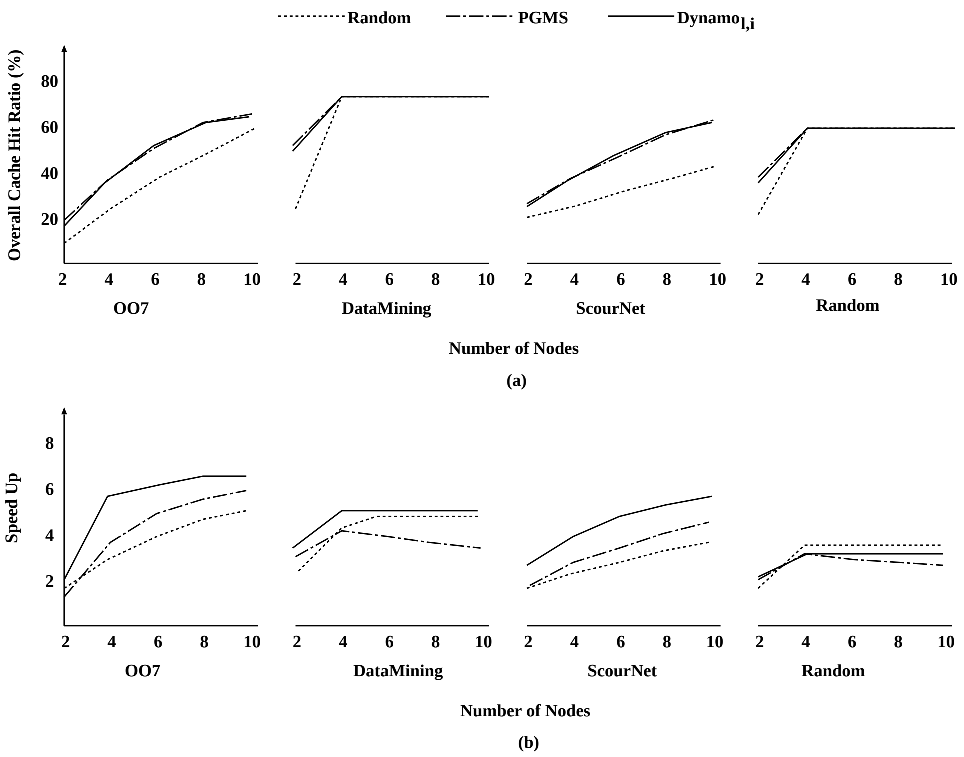


Figure 15. Scalability of Four Schemes

With the OO7 and ScourNet workload, $Dynamo_{l,i}$ and the PGMS model have a similar overall cache hit ratio, which is significantly higher than the random eviction scheme. This is explained with similar reasoning as in the previous subsections. Since the entire working set of the random access workload and data mining workload can

[||]Overall cache hit ratio is the ratio of the local cache hits and remote cache hits over the overall number of requests.

fit into the main memory of four nodes, all schemes have the same overall cache hit ratio when the number of nodes is larger than 4.

Performance is measured in terms of the speed up over the non-cooperative caching scheme. With OO7, data mining, and ScourNet workloads, $Dynamo_{l,i}$ has the best performance. However, with the random page accessing workload, the random eviction scheme may outperform others (when the entire data set can be contained in the main memory). This is mainly due to that there is no useful page access pattern and the random page eviction scheme has the least overhead.

Migration of Management

Here we examine the effects of migration of management by comparing $Dynamo_{l,i,m}$ (the Dynamo implementation which employs dynamic migration of management with lottery-based page eviction and intention set schemes) against the Dynamo implementation which employs the lottery-based page eviction and intention set with static partition of management, denoted as $Dynamo_{l,i}$. In other words, in $Dynamo_{l,i}$, a given object is managed by a given node. 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).

Scalability

In this experiment, we fix the memory on each node to 60MB and the number of nodes to 5. All objects are partitioned into five non-overlapping sets; a node is assigned to manage a set of objects. In $Dynamo_{l,i}$, the assignment is permanent, however, in $Dynamo_{l,i,m}$, the assignment of management may change at execution time. The workload is generated as follows: an application requests data initially managed by node A 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 16(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 on each node is small (i.e., less than 4 applications), $Dynamo_{l,i}$ and $Dynamo_{l,i,m}$ have the same average response time for page retrieval because the system is lightly loaded. When the number of applications is large, the average response time of data retrieval with $Dynamo_{l,i}$ is increasing at a much faster pace than that of $Dynamo_{l,i,m}$ because uneven distribution of management makes $Dynamo_{l,i}$ saturate faster.

Adaptability

As in the previous experiment, the memory on each node is set to 60MB and there are a total of 5 nodes in the LAN. The workload in this experiment is generated in the same manner as in the previous experiment except for one difference: the heavily

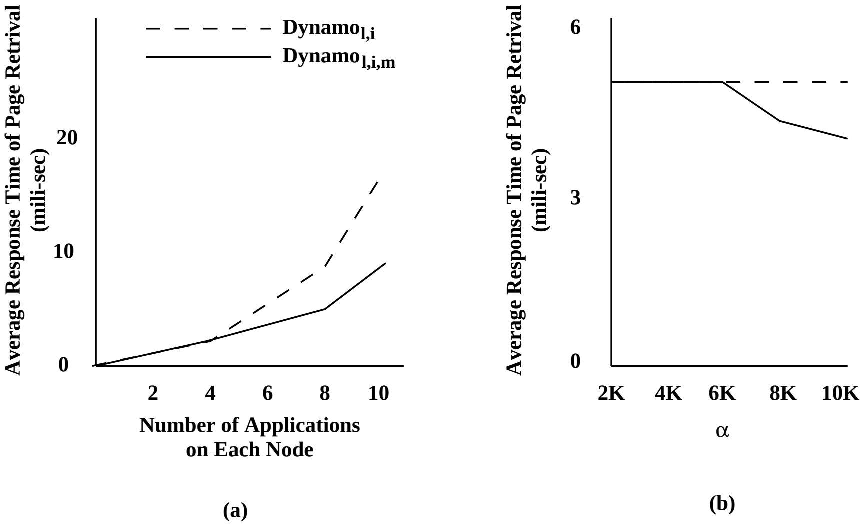


Figure 16. Comparison of Three Cache Eviction Schemes

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 16(b) shows the average response time for a page request. Here α can be viewed as the duration of one page access pattern phase. 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), $Dynamo_{l,i,m}$ outperforms $Dynamo_{l,i}$ significantly due to the more even distribution of workload on each node with $Dynamo_{l,i,m}$.

Brief Summary

From these experiments, we can see that Dynamo is especially beneficial when the entire data can not be fit into local cache and there exists some locality in the page access pattern. We believe that most applications fall into this category. In such cases, the intention set and lottery-based eviction scheme will be beneficial.

Moreover, in general there is a skew in object access pattern and the hot spots change over time, thus, creating a bottleneck in the object management. Dynamic migration of management can eliminate the bottleneck in persistent object management.

Related Work

The work done in Dynamo is motivated by several systems such as NASD [12] which employs 'third-party transfer' of data from network-attached storage devices directly to client machines. In the remainder of this section, we will compare Dynamo with related file system architectures, and cooperative caching strategies.

Initial work on a more scalable approach to client-server architectures was done by Franklin, Carey and Livny [11]. Here, the server observes what the clients are doing and uses the fact that a page might be in some client's main memory so that the server is able to serve other clients' requests directly from a client's cache instead of its own limited buffer or disk resources.

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. In NOW, workstations are connected by a fast LAN and disk devices are attached to all workstations. Part or all of the client workstations can act cooperatively as a file manager or storage server, or both, and xFS employs a cooperative cache. Similar to Dynamo, the file manager and storage server are not a centralized system, but their role can be taken by any client machine, and the work load is shared between clients. In Dynamo, however, storage managers run directly on the CPU of network-attached devices. This fact allows Dynamo to dynamically assign data management tasks to any machine in contrast to xFS which statically assigns pages of a file to be managed by a particular file and storage manager. Furthermore, the problem of how to utilize frequently changing resources (e.g. new machines added to the LAN) in the most effective way is not addressed in the xFS system.

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 [7, 14, 22]. The second research direction focuses on global cache replacement strategies that are managed by clients in a decentralized manner [7, 18]. Voelker et al. [22] proposed a globally managed prefetching and caching system, the PGMS system. In PGMS, a node is selected as a leader which is the least loaded node in the system. The leader receives the buffer information of all nodes, and centrally computes the replacement set of pages for each node. This is repeated every 5 to 10 seconds. The goal of PGMS is to utilize the available global cache so that the average access latency can be minimized. The approach, however, has several potential problems. First, theoretically, leader election on a general distributed computer systems is impossible [17]. Although the leader may be elected successfully in the current local area network environment with very high probability, 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 number of pages can potentially be replaced on each node. In addition, with a larger node population it is more likely that the number of pages which can be replaced on a node will change more rapidly causing the frequency of new epoch to increase. Therefore, a significant overhead may occur and the scalability can be impacted.

In the original paper on cooperative caching by Dahlin et al. [7], several techniques for cooperative caching are described, and the authors chose the decentralized *N-Chance Forwarding* algorithm, used subsequently in xFS [1]; the algorithm tries to avoid evicting pages from the cooperative cache for which there is no other copy within the cooperative cache, but rather is biased toward evicting replicated pages. Instead of writing the last copy to disk, it is forwarded to a random peer client cache. In Dynamo, we introduce the concept of *intention sets* so that page eviction candidates are not forwarded to a random client, but to a client that is more likely to access the page.

Sarkar and Hartman [18] introduce a decentralized global caching algorithm based on hints. In contrast to the N-Chance Forwarding algorithm which is based on facts about the global cache state, the hints-based algorithm is based on approximations of the global state. In this work, it is predetermined which pages get evicted to the global cache (master copies) and which are not (replicas), so that a client does not need to contact a centralized manager to make the eviction decision. Instead, each client makes guesses about the system's oldest blocks that it keeps in its memory, compiling a global oldest block list which approximates a global LRU. Also, clients requesting a page receive an access token from the server and a list of hints of the most likely cache locations of the page. In Dynamo, exact information about cache location is kept and provided to a client, so that 'misses' are avoided. Still, since the cache location management is done by clients, it is still highly efficient and scalable.

Cao et al. [3, 4] also did work on cooperative file caching, and investigated techniques for efficient prefetching of data into a cooperative cache. Prefetching, however, has not yet been considered in Dynamo. 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 [6] system, and in Thor [15] and Hac [16].

Conclusion

We have presented the complete design for a scalable, fault-tolerant, cooperative object management system for LAN environments that is dynamically adaptive to configuration changes. In addition, the design can adapt to shifting workloads and hotspots in object access. The prototype system has been used to provide path length measurements. We also proposed and evaluated an extension to cooperative cache management algorithms in which the system attempts to place a page in a node that is likely to access that page when it has to be replaced from the node on which it is currently cached. 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).

Future work includes deployment and measurement of the system under actual workloads. In addition, application specific "hints" for management will be explored to see if significant further performance improvements are achievable. Further work on security issues are also warranted to extend this design to environments in which it is not practical assume individual systems are "well behaved".

ACKNOWLEDGEMENTS

This work was done when the first and second authors were in the University of California, Los Angeles. Discussion with Greg Ham are also gratefully acknowledged.

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] . R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between set of items in large databases. *Proc. ACM SIGMOD Conf. on Management of Data*, 207-216, 1993.

[3] . P. Cao, E. Felton, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1995.

[4] . P. Cao, E. Felton, A. Karlin, and K. Li. Implementation and performance of integrated applicaiton-controlled file caching, prefetching, and disk scheduling. *ACM Trans. on Computer Systems*, 14(4), November, 1996.

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

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

[7] . M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: using remote client memory to improve file system performance. *Proceedings of First Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[8] . Dias, D.M., Balakrishna, R.I. Robinson, J.T., Yu, P.S., Integrated Concurrency-Coherency Controls for Multisystem Data Sharing. IEEE Transactions of Software Engineering, Vol. 15, No. 4, April 1989.

[9] . Fibre Channel Association. *http://www.amdahl.com/ext/CARP/FCA/FCA.html.*

[10]. M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levey, and C. Thekkath. Implemention global memory management in a workstation cluster. *Proc. of the 15th ACM Symposium on Operating System Principles*, December, 1995.

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

[12]. 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.

[13]. E. Grochowski and R. F. Hoyt. Future Trends in Hard Disk Drives. *IEEE Transactions on Magnetics*, May 1996, vol.32, (no.3, pt.2):1850-1854.

[14]. Avraham Leff, Philip S. Yu, Joel 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.

[15]. 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'96, Montreal, 1996.

[16]. 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), Saint-Malo, France, October, 1997.

[17]. Nancy Lynch. *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.

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

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

[20]. T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobb's Journal*, February 1995.

[21]. G. Voelker, H. Jamrozik, M. Vernon, H. Levy, and E. Lazowska. Managing server load in global memory systems. *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1997.

[22]. 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. on Measurement and Modeling of Computer Systems*, 1998.