Dynamo: Design, Implementation, and Evaluation of Cooperative File Management in a Local Area Network

Jiong Yang, Wei Wang, Silvia Nittel, Vincent Busam, and Richard Muntz
Department of Computer Science
University of California, Los Angeles
{jyang, weiwang, silvia, vince, muntz}@cs.ucla.edu

December 7, 1998

Abstract

Cooperative caching and distributed file management have been shown to be scalable and effective in adapting to varying application resource requirements. In particular, significant gains are obtainable when some subset of individual working set sizes exceed the available memory on any one node, but the aggregate sum of working set sizes is less than the aggregate global memory available. In this paper, we extend previous work in several directions. First we provide a design that can adapt quickly and automatically to the number of nodes that are online. We argue that this is an important enhancement that can enhance fault tolerance, easy LAN management and other advantages. Related to this is the ability to migrate the management responsibility for subsets of pages. We show that this provides advantages in balancing CPU usage across collaborating machines. Finally we provide an improved algorithm for replacing pages from the local memory of one node by intelligently choosing which other node should cache the page. Performance studies on a prototype implementation of Dynamo are used to evaluate the effectiveness of the proposed improvements.

1 Introduction

Today, dozens of workstations and PCs form the typical local area network (LAN). Usually each of them is equipped with several hundred MHz CPU and around 100 or more MB of main memory. Often single workstations or PCs are not fully utilized by the applications running on them (e.g. “thin” applications such as text editors) resulting in idle processors and a significant large amount of free main memory. At the same time, a single node is often not powerful enough for efficiently running “fat” applications such as data mining which can result in saturated CPU or memory trashing. Often, a fat application is running on one machine while only a few thin applications are running on other machines with similar resources in the same LAN. The result is a very skewed distribution of CPU and memory consumption among nodes in a LAN.

Moreover, the identities of the saturated nodes change over time. For example, a user may run a network simulation program on his/her desktop which may last a few hours, and then terminate. By then another user may start a data mining program on his/her desktop. This behavior requires a system which not only can balance the resource usage, but also one which adapts relatively quickly, inexpensively, and automatically.

Further, laptops are becoming an affordable and powerful substitute for desktops. Although, a top-of-the-line laptop is technology-wise several years behind and 50% more expensive than its desktop counterpart, it is very powerful, indeed, and used more and more often because of its attractive mobility. Currently, a laptop equipped with 200+ MHz CPU and 80+ MB main memory is commonplace. People are increasingly using laptops at work, plugging them into the LAN environment and using them as their primary equipment, but then unplugging them from the network when they go home or travel.

In such a LAN environment, not only does the application load vary, but also the physical nodes present in the LAN environment changes relatively frequently. To deal effectively with changes in resource availability requires a distributed file system which can not only dynamically balance the CPU and memory usage over all existing nodes dynamically, but also adapt automatically, quickly, and efficiently to the changing makeup of the LAN's nodes.

Today, systems in a LAN such as file systems often employ a three tier architecture, including presentation, business logic, and data layers. In such an architecture, the system code is divided into the so-called *data* portion, which stores data and provides basic I/O services, and is usually referred to as data repository, the *business logic* portion, which provides basic buffer management and concurrency control and runs on a dedicated machine such as a workstation or SMP (usually it is called server machine), and the *presentation* portion which provides the API and executes on the machine on which the user application is running (client machine). Normally, a business logic server interacts with many client machines. Here, the scalability, adaptability, and performance of the overall file system significantly depends (mostly) on the compute power, aggregate bandwidth, etc. of the server machine, the server's scalability and the load and access patterns of clients.

Advances in technology, however, have made it possible for different software architectures to emerge. Mainly, three major trends have influenced the architecture of file systems: the available bandwidth in LANs, the emergence of network attached storage, and the increases in CPU power and main memory of server and client machines. At the beginning of the nineties, the typical bandwidth of a "fast" network was in the range of 10 Mbit/sec while a typical system bus's bandwidth was in the range of tens of MByte/sec leading to the fact that servers used directly attached disks and often processed data first and sent the result to clients, thus, reducing the load on the network. Today, point-to-point connected fibre channel is capable of transferring data at 100 MByte/sec and the industry projection is that its bandwidth will reach 400MByte/sec soon [FCA]. Thus, the network is no longer the bottleneck in a LAN environment making it also feasible to attach storage devices directly to the network instead of to a server machine. Today, the rate of bandwidth increase of a single disk is about 40% a year while the price per MB drops about 60% per year [Gro96]. By the year 2000, a megabyte of disk will cost about four cents and each individual disk will sustain a bandwidth on the order of 40 to 50 MB/sec. This trend suggests that systems will have much higher aggregate disk I/O bandwidth in the near future (400 MB/sec point-to-point) if attached to a fibre channel based network.

Based on technology trends and the observations discussed above, we introduce a highly scalable, adaptive, and distributed file management system: Dynamo (*Dynamic File Management System*). Dynamo provides an application interface and functionality similar to traditional local area file systems such as NFS and xFS [And96], which allows the storing and retrieving of data, buffer management, and consistency control of files. Dynamo has a layered architecture consisting of an I/O layer, a cooperative cache management layer, and a file management layer which provides consistency control. It differs, however, from the traditional client-server architecture of the file systems. In Dynamo, the file management and cooperative cache management layers reside on the client machine, and interact with a server on the server machine. However, the server part is much smaller in Dynamo compared to traditional file servers and acts as a coordinator rather than a complete server. At data access time, the necessary server code for cooperative cache management, and catalog information can be *dynamically downloaded* to the client machine and run on the client machine. (The dynamic download is only used to take advantage of available laptops and to avoid using up the laptop's disk space; we assume that the client parts are permanently installed on workstations and PCs.) The file management layer communicates with the coordinator on the server machine concerning the location of relevant data. However, instead of loading data through the server machine, the file management layer on the client machine interacts with Dynamo's I/O layer that resides on the disk controllers of the network-attached disks. This I/O layer performs physical and logical device management, and provides the abstraction of data pages to the file management layer. Requested data is directly retrieved from the network-attached disk and cached locally on the client machine, thus, eliminating a potential bottleneck caused by the server's bus bandwidth limitation.

In Dynamo, each client acts as a cache for local data that is shared with other clients, thus, providing a cooperative cache. Since the collective memory on all client machines is usually much larger than that on server machines, the cache hit rate can be improved, and disk I/O avoided. However, the cooperative cache management in Dynamo is different from existing cooperative caching systems such as [And96] [Cao95] [Cao96] [Fee95] [Fra92] [voe97] [Voe98]. In most of these systems, a client wanting to replace data in its own cache first tries to place the data in the cache of other nodes if space is existing at these nodes. If not, the data will be discarded or written back to disk. In determining the candidate remote node to which the local cache page will be evicted to, most cooperative caching systems consider the CPU and memory usage on a candidate remote node, however, neglect considering the likelihood that this node will use the page in the near future. Since accessing a page in a remote node is a few orders of magnitude slower than from local memory via fast local area network, evicting the local cache page to a remote node which is more likely to use the data could potentially increase the overall local cache hit rate, thus improving the overall system performance.

Another major difference between Dynamo and other existing file systems is its adaptability to both dynamic workloads and dynamic changes to the configuration of machines on the LAN. Most previous research has focused on scalability, i.e., the performance of a file system when the number of nodes in a LAN is large. However, they do not exploit the *adaptability* of a file system, especially adaptability with respect to changes in the makeup of the nodes on LAN. Dynamo quickly reacts to a newly added node since a node can download the necessary code for the file manager and cooperative cache manager and start to share the workload with other nodes. This is achieved via Dynamo's flexible control: due to the usage of network attached storage, any node can be the manager of any data in the system. Therefore, it is not necessary to have a fixed partitioning of data among the Dynamo's file managers. During the lifetime of a data item it can belong to many different managers which reside on different machines. Considering the balance of CPU consumption and available memory space, Dynamo dynamically determines the subset of data assigned to a file manager. Similarly, the subset of data managed by a node can be transferred to any other node(s) if that node is to be shut down. For the time being, we consider that dirty pages are kept on PCs or workstations which are not unexpectedly taken off the LAN while laptops are used to extend the cache for non-modified pages. In case of an unexpected shutdown of a laptop, the system only loses the online availability of the machine's cache. If a node's CPU becomes saturated, part or all of the data management performed by that node can be moved to other nodes. Thus, Dynamo not only adapts well to the workload changes of each node, but also to the configuration of machines on the LAN.

In Dynamo, we assume that all nodes in a LAN are trusted. We believe that nodes in a LAN can usually be viewed as trusted nodes because they usually belong to the same administration domain and users for those nodes belong to same group or company.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the Dynamo architecture. The experimental results are analysis in Section 4. We draw the conclusion in Section 5.

2 Related Work

The work done in Dynamo is motivated by different systems such as NASD [Gib97] which employed the 'third-party transfer' of data from network-attached storage devices directly to client machines. Furthermore, Dynamo was influenced by the Condor system [Tan95] which envisions machines in a LAN as a pool of machines (each consisting of processor, memory and disk resources) that are dynamically used according to their available resources, as well as by the concept of 'thin clients' that introduced the idea of dynamically migrating functionality to other 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 [Fra92]. Here, a server was introduced that extensively used the available client cache for keeping data pages in memory so that the server was able to service other clients requests directly from a client 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 [And96] as a 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 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 are run directly on the CPU of network-attached devices assuming that these devices are primarily used for storing data in the LAN environment. 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. availability of laptops) 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 [Dah94, Lef91, Voe98]. The second research direction focuses on global cache replacement strategies that are managed by clients in a decentralized manner [Dah94, Sar96]. Voelker et al. [Voe98] 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 some potential problems: theoretically, leader election on a general distributed computer systems is impossible [Lyn96]. 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 before the end of an epoch, thus a new epoch has to begin. Therefore, a significant overhead may occur and the scalability can be impacted.

In the original paper on cooperative caching by Dahlin et al. [Dah94], several techniques for cooperative caching are described, and the authors chose the decentralized *N-Chance Forwarding* algorithm, used subsequently in xFS [?]; the algorithm tries to avoid evicting pages from the cooperative cache that are the only copy within the cooperative cache, and rather evicts 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 most likely to access the page.

Sarkar and Hartman [Sar96] 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. [Cao95, Cao96] 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 [Car94] system, and in Thor [Lis96] and Hac [Cas97].

3 The Dynamo Architecture

To better balance the use of computer resources (e.g., CPU and memory) and achieve higher performance, we have implemented the Dynamo architecture. The main idea of Dynamo is to dynamically move file server functionality to client (machines) for execution, dynamically migrate the management and control of data to idle or less loaded machines, thus, automatically adapting the file system to the availability of resources (CPU, memory, machines) in the LAN, and last but not least dynamically "extend" a local cache with the available memory of other machines. We make the following three assumptions for today's LANs.

1. *Data is stored on Network Attached Disks.* Fibre Channel 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, we believe that more and more LANs will 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 unnecessary to have a fixed partitioning of the data management within a distributed file manager. A static partitioning is a good choice in a machine-attached disks environment since the data has to flow into the machine's main memory, then go out to other machines.

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 or not a node should assume the responsibility to manage a set of data can be determined on the basis of performance (e.g., whether this node has enough computer resources), reliability, etc.

3. *More or less symmetric access time.* A LAN is usually connected via a high speed communication networks, Ethernet, Myrinet, Fibre Channel Arbitrated Loops, etc. In addition, the average number of nodes in a LAN is ranging 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 on the disks. This provides the motivation and justification for moving cached pages to other nodes' memory.

The Dynamo architecture emerges from the above three assumptions. It consists of four layers, disk I/O layer, active cooperative cache management layer, file management layer, and coordinator layer. The disk I/O layer code resides at the network disk controllers. It provides space allocation and page-id to disk id mapping functionality. For example, when a data page has to be written to disk, the page id and the data is given to the disk controller and the disk controller will find the proper disk block(s) in which to store the data. It also maintains the page to disk block translation. Disk I/O layer research has been pursued by many [Gib97]. Due to space limitations, we will not discuss this layer in detail in this paper. We will, however, discuss data organization, file manager and coordinator, active cooperative cache manager, and fault tolerance in more detail. The overall architecture is illustrated in Figure 1.

3.1 Hierarchical Data Organization

Data in Dynamo can be viewed in two ways: the internal view and the users' view. Users identify data via logical descriptions such as file names, directory names and file offsets. The system internally 'sees' all available data as a set of data pages each of which is uniquely identified. The universe of data pages is organized in a hierarchical manner in Dynamo, as shown in Figure 2. At the bottom level, the granule is a page, i.e., each entity represents a page. A page usually consists of 4k or 8k bytes data. At the level above, each entity represents a set of pages

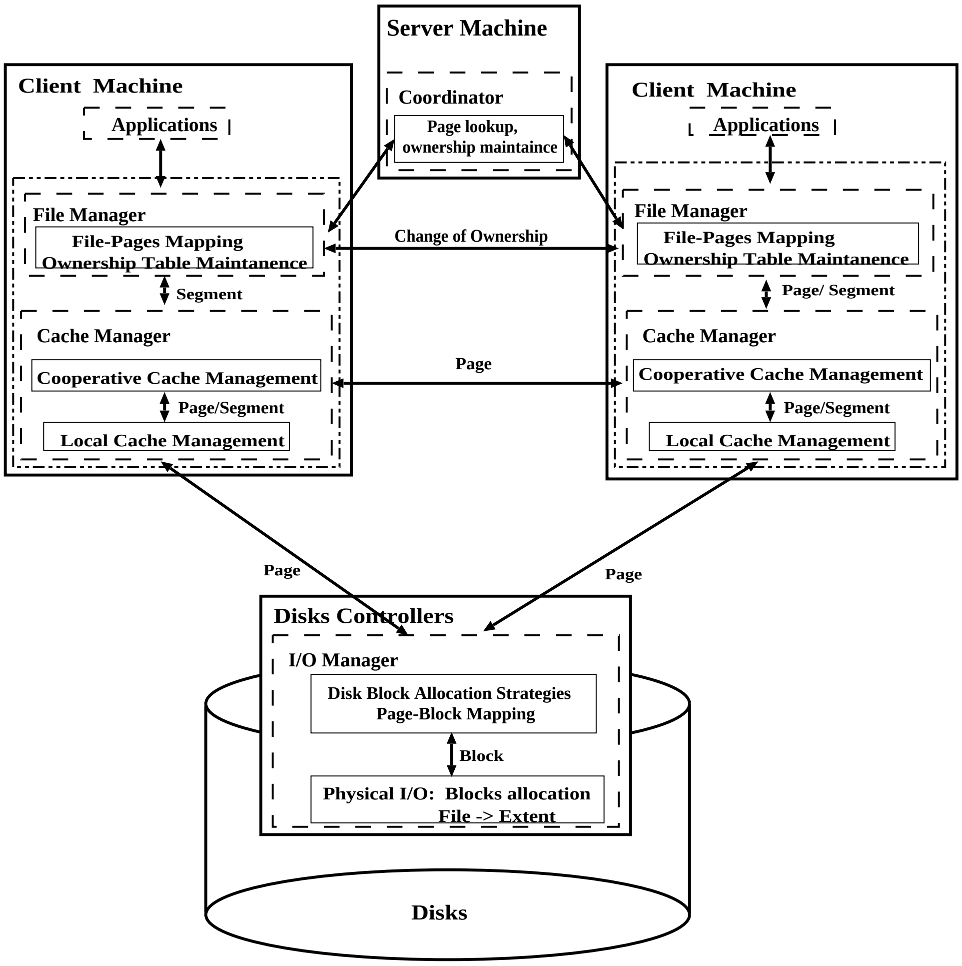


Figure 1: Dynamo layers

with a varying number of pages per set. The granules at the next level up are called clusters, set of clusters, set of sets of clusters, and so on. On the top level, there are from a few to a few hundred root entities each root entity encompasses a large set of data pages. This structure is used to group and identify pages and groups of pages.

Each entity in the hierarchical structure has an id. We use a *prefix naming structure* to keep track the parent and children relationship in the hierarchy. For example, if the ids of two entities in the hierarchy are 1123000 and 1123210, respectively, then the latter one is a descendent of the former. In addition, the entity with id 1123100 will be a descendent of the former, but not an ancestor of the latter. Each level is represented by a byte in the entity id. The length of entity id can be chosen by the system administrator. The length of the id can be between six to eight bytes, or more, based on how large the data set is. The maximum fanout of an internal node is 255. If the id is of eight bytes long, i.e., the hierarchy is eight levels, then Dynamo can address up to 10^{13} GB data.

When a file is created, a unique id is assigned to each of its pages. At this point, the user can choose with which files the newly created file should be 'clustered'; this is also possible if the related files are stored in another user directory, but often used with the newly created file. Thus, the internal hierarchical structure mirrors how data is used rather than the users' directory structure. Dynamo generates the ids for newly created file pages according to the rule that all pages of a file and all files in a directory are grouped together. However, if a file consists 1000 pages,

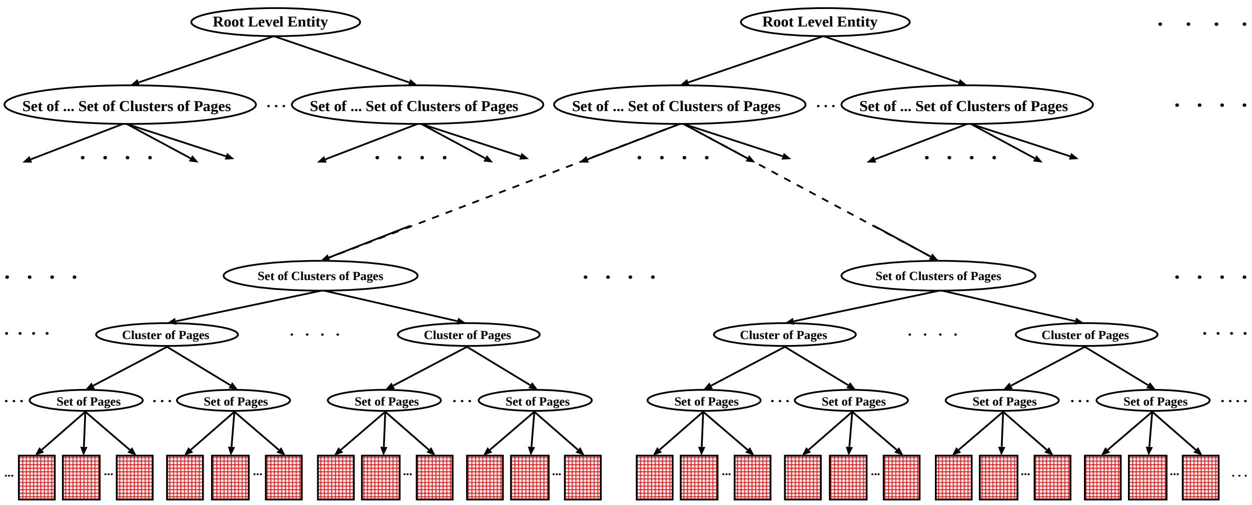


Figure 2: Hierarchical Data Organization

then, four entities at level two are created, all of them are children of one entity at level three, and each of them consists of 250 entities at level one (page). This means that the id of an entity determines the position of that entity in the hierarchy. Once an id is chosen for a data page, it normally remains the same for the lifetime of that page.

Each internal node is stored via one page. For example, each node has at most 255 entries. Each entry consists of the page id of the associated child node and the disk id for the disk on which the page representing the child node is stored. For a given data page (with the page id), the position of the page in the hierarchy can be known easily with the prefix naming structure, e.g., the ascendant of this page. Thus, it is always possible to find the disk on which the page is stored by traversing the hierarchy from the root. The maximum number of pages to 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 nodes may be cached by the coordinator and the file managers.

In our current scheme, it is quite cumbersome to migrate one entity to another location in the hierarchy. For example, if we want to migrate entity 1123100 to be a child of 1124000, then we have to rename all the descendants of 1123100 to reflect this change. If the migrating entity is at a very high level, then there is likely to be a significant number of entities that need to be renamed, and this could be an expensive operation. However, this inflexibility is compensated for by supporting the dynamical transfer of ownership. Looking ahead, a file manager can transfer the ownership (we will explain the term of ownership shortly) of an entity at any level so that it can alleviate hot spots. The downside is that, a file manager could end up with a large number of disassociated low level entities and the coordinator has to keep track all these information. However, if the application exhibits some locality of reference (as in many applications), then the file manager should manage only a small number of high level entities. In turn, the bookkeeping information will not be too much.

3.2 File Manager and Coordinator Layers

The management of data pages, data access, and concurrency control is mainly handled via the coordinator and file managers in Dynamo. The coordinator is the remaining server portion of Dynamo; most of the work is done by file managers running on available client machines.

3.2.1 Coordinator

The coordinator plays a bookkeeping role in the Dynamo architecture. It has two responsibilities: (1) it keeps track of which data set is owned by which file manager and performs all logical data address to page id mapping. (2) the coordinator keeps track which data is *owned* by which file manager. If a file manager *owns* data in Dynamo, it is responsible for the data management of this data chunk, i.e. it manages the information table of the data's storage location (disk id and node id if the data is cached on that node), provides this information to other file managers, grants and keeps track of locks and concurrency control. The coordinator assigns the actual data and its ownership to a file manager initially. During runtime, this data management 'load' is automatically balanced as we will describe later. The coordinator is located on one or more machines. If a distributed coordinator scheme is employed, then the set of nodes on which the coordinator is running should be known to all machines in the LAN. We assume there is a set of nodes in a LAN which are the *core* nodes, i.e., they are highly available nodes in the LAN. Looking ahead, newly added nodes will share the file management and active cooperative cache management responsibilities, but they do not share the coordinator responsibilities. However, since the work performed by the coordinator is only a small fraction of the overall work load, it does not impact the scalability of Dynamo.

3.2.2 File manager

The responsibilities of a file manager is to keep track of caching information, and to enforce consistency control on all the data it owns. An application trying to access a data item passes the logical data address to the file manager as illustrated in Figure 1. If the file manager does not know how to translate the logical data address to a page id, it means that the file manager does not own the data. In this case the file manager sends a request to the coordinator specifying the logical data address. The coordinator locates the page id of the data in the Dynamo hierarchy and id of the file manager that owns the data. This information is returned to the requesting file manager. If the data is not owned by any file manager, the coordinator will make the requesting file manager the owner of the data pages, i.e., the highest level entity in the hierarchy which includes the data and it is not currently owned by any file manager. Also the coordinator will send back the disk id for the disk on which the index page of the entity resides.

If an file manager is the owner of the data, it knows where the cached data is regardless of which node has it cached. The owner of the data is not required to hold all the cached data in its local memory. It is theoretically possible that all the data it owns is cached on some other nodes in the LAN because the application running on the local node, does not reference any of the data it owns. Thus, it can either get the cached data (if a cached copy exists) or fetch it from the disk where the data resides. If the file manager does not own the data, it contacts the owner of the data, and might be able to retrieve the data from another cache or the disk.

The file manager is also 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 will be in control of granting these permissions and will also take care of lock scheduling, dead-lock detection, etc. Due to space limitation, we will not discuss the consistency control in detail here.

3.2.3 Ownership Transfer

When file manager A accesses data currently owned by file manager B, A and B will, as a byproduct, also decide whether the ownership of the data should be transferred to A. If it is determined that an ownership transfer is desirable (e.g., file manager B thinks it will not need to access the data in near future, and its machine is highly loaded) then B can decide to transfer ownership of a subset of its data to A. To illustrate this process more clearly, let us assume that the entity which is requested is v' and v' is a descendant of v, an entity owned by file manager B. B decomposes v into a set of children entities, and one of the children contains v'; B checks whether it needs the particular child entity of v that includes v'. If yes, then the child entity is decomposed and this 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. This is one possible scheme which we do not claim to be optimal. For example, the algorithm described above does not account for the total size of entities owned by B or the frequency of access for each entity, etc.

If file 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, file manager B decomposes v into a set of disjoint entities v_1, v_2, ..., v_k where $\bigcup_{i=1}^{k} v_i = v$. Then B removes v_1 from this set. B sends all information associated with v_1 (e.g., which client have cached data in v_1, which client has read/write locks on data in v_1, etc.) to A.

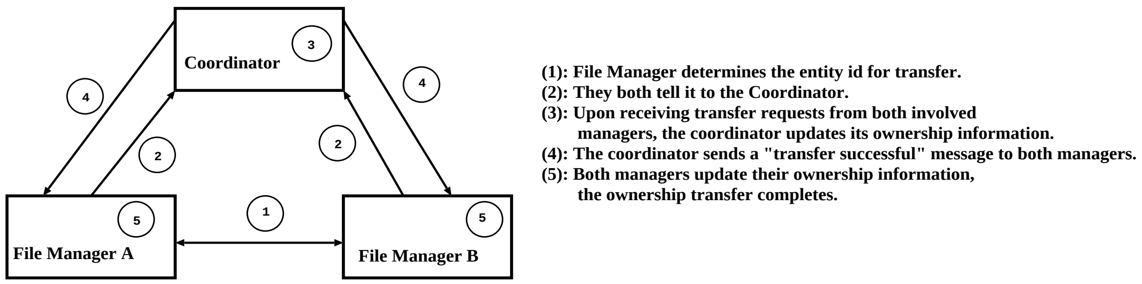


Figure 3: Ownership Transfer

File manager A, in turn updates its ownership table by adding v_1. Then, 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. Then, A downloads the code for consistency control from the server if A does not have this code. (In some environment, different data set may require different consistency control protocols, e.g., optimistic, pessimistic, etc.) After receiving the message of ownership change, the coordinator updates its own coordinator table which contains both client A and B's ownership tables using essentially the same procedure as A and B. Figure 3 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 in transfer 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.

3.3 Active Cooperative Cache Management Layer

The file manager also cooperates with the local active cooperative cache manager which consists of a distributed cache manager which communicates with other caches, and is responsible for data transfers, and the local cache manager that performs regular buffer management. The file manager in cooperation with the distributed cache manager maintains the page id to node id (who has a cache of the page) mapping for the data it owns. If another file manager requests data, the file manager returns the node id(s) for the data. For requested data that is locally cached, the file manager retrieves the physical memory address for the page from the cache manager for requests from local applications and remote file managers. (Requests from remote file managers originated from remote applications.)

The active cooperative cache manager provides another major functionality: if the cache on a local node is exhausted and new data is requested, a set of cached pages has to be replaced. In a traditional cache manager, the old pages are evicted either to disk or discarded if no modifications were made in order to accommodate the new data. Since the retrieval of a page from a network neighbor machine is usually several orders of magnitude faster than the disk retrieval, Dynamo attempts to place the eviction candidate in other nodes' cache instead of writing it back to disk.

To free cache space, the local cache manager selects a set of candidate replacement pages which are chosen to be the pages that have not been referenced for the longest time locally. Then the cache manager sends a message to the owner(s) of the data. The cache manager's message also includes an indication of when the page may be needed by its local application(s) again. In turn, the owner checks whether the page is cached anywhere else and updates the time-stamp to indicate when the cached page may next be needed by an application. Then the owner returns a message to indicate either (1) the page has been cached elsewhere, (and if so, where), (2) this is the only cached page and no node intends to reference it in the near future, or (3) this is the only cached copy of the page and someone intends to use it in the near future. The prediction information is maintained by a timestamp on each cached page. When the responses have been received we can classify each page in the candidate set as type 1, 2, or 3 depending on the type of response for that page. The cache manager will evict pages of type 1 first and pages of type 2 as needed. After discarding the pages of type 1 and 2, the cache manager may have to replace the pages of type 3. Before discarding a type 3 page, the cache manager will attempt to find a node to cache the page which has indicated that will be re-referenced in the not too distant future.

There are several sources for prediction of future references. (1) If there is application level knowledge on data access pattern, e.g., multimedia streams, then the application can advise the cache manager via file manager. (2) We can use an approximation algorithm, e.g., if a page has not been referenced for T time units in the past, we assume the page will not been referenced for T time units in the future. (3) If the page is only used by one application and that application has terminated, then we can assume it will not be referenced in the foreseeable future.

In determining nodes to hold pages that will be re-referenced, cache managers broadcast the available local cache size to all its peers when the local cache size changes significantly from the last broadcast (e.g., from 30 MB to 10 MB or from 0MB to 15MB) or a significant amount of time has elapsed (e.g., 2 to 5 minutes). The available cache size is determined as a factor of the CPU usage and the free memory. (Serving pages to other nodes also consumes a significant number of CPU cycles.) When an application terminates, all pages that are only accessed by that application are either discarded or written back to disk. In addition, if a page is not going be used for a significantly long time, a cache manager may decide to discard it after obtaining the confirmation from the owner.

During the node selection, a local active cooperative cache manager uses a random number generator to determine which of all the nodes available should be chosen. The choice of the random number generator is based on the last broadcasted available cache size. In Dynamo, a node i will be chosen with the probability N_i/N where N_i is the last broadcasted available cache size on node i and $N = \sum_{j \in allnodes} N_j$, which is the total available cache from the latest broadcasts. Therefore, a random number generator is used that creates numbers in the range from $[0, 2^{16}]$, and divide the range into sections based on availability of a node (e.g. node A has twice as much cache space than B, therefore, its number range is twice as large as B making it twice as likely that a page gets evicted to A). When a page is to be evicted, a random number is created, and according to that number, a node is selected. If a node C sends a page to a remote node D and D has no more available cache, then C will update its random number generator ranges to reflect this and repeat this process until it pushes the cache to other node or a certain number of trial has failed. In the latter case, the page will be discarded. In this scheme, no centralized global cache manager is required. In addition, there is no synchronization necessary among the distributed cache managers in determining where the cache should be sent. Moreover, the cached pages tend to be proportionally distributed with respect to the available cache size on each node.

Intention Set

When a page is evicted from one node's local cache to a remote node (rather than to disk), there are potentially two benefits: 1) the node (which evicts the page) or other nodes can retrieve the page from the global cache instead of the disk for a future reference. 2) the remote site only needs to perform a *local memory access* if the evicted page is used by the applications on that remote site. As mentioned before, most cooperative cache systems only focus on the first part. However, the second benefit could be substantial if there is a significant possibility that the remote node will use that page in the near future.

To exploit the second benefit, Dynamo employs the concept of an *intention set*. For any data entity (e.g., pages, clusters of pages, etc.), there is an intention set associated with it. An intention set consists of a set of nodes. If a node is in an intention set for a given data entity, then the node is very likely to reference pages in the data entity in the near future. An intention set can be formed by using measured behavior, e.g., past references. In current implementation, if the frequency of a node references pages in a given file exceeds certain threshold during past ten seconds, then the node is added into the intention set for that file. Once a page needs to be replaced from a node's local cache, instead of evicting it to the global cache with the only consideration being cache space and CPU load, the cache manager first tries to send this page to a node in the intention set of that page (this occurs only if no replication of the page exists in any other node). If no space is available in any node of the intention set, then the page is evicted to any available node.

A file manager can dynamically join or leave any intention set. For example, when an application opens a file, it is likely that this application will read pages from the file, then this node should join the intention set for the file. Upon closing of the file, the node leaves the intention set. During a cache eviction cycle, the probability is high that the cached page of a files will be evicted to the node in its intention set which can significantly increase the local cache hit rate, reduce not only the memory access time, but also the CPU load on each node. In section 4, we use the trace data from a World Wide Web (WWW) multimedia search engine on Dynamo with and without the intention set technique, and results demonstrate a performance improvement of 25%.

If the access time to nodes is highly asymmetrical, e.g., it is much faster for node A to retrieve data from node B's cache than node C's cache, then a weight is assigned to each remote node during the node selection process. It is possible, a node may prefer to extend its "logical" cache to a particular node with less available memory because it is *closer*. During the transfer of the cache elements, the owner of the data should be notified of this change so that when some other file managers or applications request this data, it knows where the elements reside.

In the current implementation, we only push clean pages to the cache of other nodes, because if a node holding the dirty page crashes, the file manager that generated the dirty page can not get the data back. However, we also assume that the core nodes tend to be available in the LAN for a long period of time and they will send, most often, notice before they are shutdown, allowing for proper finishing of transactions.

3.4 Fault Tolerance

In Dynamo, a user may "unexpectedly" remove his/her laptop from the LAN at any time. From the point of view of the rest of the nodes in the LAN, the removed node just "failed". Such occurrences may become commonplace if we migrate the cache manager and file manager onto laptops. Moreover, any machine may fail due to hardware or software. Therefore, failure recovery becomes a crucial point in Dynamo.

We use a replicated coordinator to guard against coordinator failure. In other words, the set of data managed by each coordinator may overlap. For a given data item, there is one prime coordinator and another secondary coordinator. When the prime coordinator fails, the secondary coordinator will assume the responsibility of the prime coordinator. This requires that when the ownership transfer occurs, both coordinators need to participate in the process.

When a file manager crashes, then other surviving nodes may request the data owned by the crashed manager, and of course, these requests can not be served. We apply a crash recovery protocol to eliminate this kind of problem. When a file manager requests data from another file manager, it has a timeout mechanism. If the file manager does not respond within the timeout period, the requesting file manager will report to the coordinator that the other file manager could have crashed. Then, the coordinator will contact the specified file manager. If the file manager still does not respond, the coordinator will revoke all ownership of the specified file manager, and assign its entities to the non-owner table. Also during normal operation, each file manager writes logs and periodically puts it on the 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. When a client machine recovers from a crash, the file manager on that client will contact the coordinator to retrieve the data that it needs.

In this scheme, the load on the server machine is minimal. In fact, only the coordinator remains on the server, and the coordinator only tracks who is the owner of each entity. Therefore, the congested server problem is eliminated.

In addition, each data entity is owned by at most one owner. Therefore, cache coherency can be maintained by the owner. This avoids complicated distributed algorithms for cache updating. Moreover, the hierarchical object model makes the partitioning of cache management easy and fair. If one client accesses a large amount of data, it could serve a large amount of data to other clients. Furthermore, if an file manager requests some data from the coordinator, it may become the owner of a very large root entity. However, if other clients request data in this root entity, this client machine can transfer the ownership of some entities (which are descendant of the root entity) to other clients. As a result, the service partition is based on dynamic usage rather than on some static apriori partition, such as used in xFS. Especially in the environment that machines can join in and leave a network frequently, (e.g., people brings their notebooks to the office and plug them into the network in the morning and take them home at night), a static partition may not work well. In addition, since there may be more than one server in a network, Dynamo lets a client download the coherency control code from the server. A client can download different coherency control protocols from different servers for different data and all of them can work concurrently and correctly as long as there is only one owner and one coherency control protocol for a given data entity.

When an active cooperative cache manager fails, all the pages stored in its cache will not be available to other nodes any more. For clean pages, the file manager may contact the owner and report the unavailability of the cache manager and the owner may find another cache or return the disk id which the data is located. The application may be re-executed if a dirty page is lost due to the cache manager failure.

4 Experimental Results

Our Dynamo prototype is implemented in C running on Unix and Linux. We performed the experiments on Ultra-Sparc1 workstations (Solaris 2.5 operating system) each with 80 MB main memory and one 167 MHz CPU. The workstation cluster is interconnected via 100 Mbit/sec Ethernet. The page size is 4KB and we use 8 byte ids for each page and internal entity. The data is stored on a 7200RPM Quantum Atlas III SCSI disk drive with 7.8ms average seek time. In these experiments we use four versions of Dynamo which are tailored to evaluate various aspects of Dynamo such as path length, adaptability to changes, etc. The first Dynamo version represents the traditional client server model. Here, the client simply sends a request to the server which returns the result to the client. We denote this version as **C-S Dynamo**. The second version of Dynamo employs the traditional cooperative cache (without intention set) (**C-C Dynamo**). The third version of Dynamo uses cooperative caching and dynamically migrates file managers to different nodes according to the CPU usage and availability of new nodes (**M-M Dynamo**). The fourth version is *M-M* Dynamo combined with the intention set technique; it is denoted as **I Dynamo**. Through the following experiments, we intend to determine the benefit of the intention set based cache management and page management migration.

4.1 Operational Path Length

In the first experiment, we used the *I Dynamo* version, and measured the end-to-end path length, as well as single segments of the path between clients, and the Dynamo components. All path length measurements are the average of 50 iterations. Table 1 shows the average path length of each operation in the process of *retrieving data from a* *remote node*. Here, we assume the owner, the requester, and the "cacher" (the node that caches the requested data) are in three different nodes. The requester node sends a message to the owner of the data for the data. The owner in turn determines the "cacher" and tells the cacher to send the data to the requester. Upon receiving the request from the owner, the cacher sends the page from its main memory to the requester. Finally the requester installs the page into its memory. Here, we assume that there is free memory space on the requester's node so that it does not have to evict any pages. The overall elapsed time is approximately 750 μs, which is more than 10 times faster than retrieving a page from disk. In addition, 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 will soon be common place, and the overall access time should be much faster with one of these networks.

Operation	Node	CPU time (μs)	Net (μs)	total (μs)
Request	Requester	21	65	86
	Owner	17	65	82
Retrieve	Cacher	17	488	505
Install	Requester	82	N.A.	82

Table 1: Path length of retrieval data from remote node's main memory

If the requester does not have any free space, it has to *evict pages* to another node. Table 2 shows the path lengths in evicting a page to a remote node in Dynamo. The node which wants to evict a page (evictor) contacts the owner of that page. The owner notifies the evictor about the eviction type of the page (see section 3.3, e.g. is the page cached somewhere else, etc.). This is referred to as "Start up" operation in Table 1. The evictor chooses a node as the receiver and sends the page to it. The receiver stores the page in its main memory.

Operation	Node	CPU time (μs)	Net (μs)	Total (μs)
Start up	Evictor	20	65	85
	Owner	18	65	83
Evicting	Evictor	22	488	510
Receiving	Receiver	82	N.A.	82

Table 2: Path length of cache replacement

Table 3 shows the path lengths associated with *ownership migration* in Dynamo. Manager 1 first determines the entity of which it will relinquish its ownership, and then the identity of the file manager (manager 2) which will be the next owner. Manager 1 sends a request to the candidate file manager. If manager 2 agrees to the ownership transfer, they exchange the ownership information (labled as "Initialization" operation in Table 3), and both managers send a request to the coordinator for ownership transfer. The coordinator updates its ownership table and acknowledges that the transfer procedure has completed.

Operation	Node	CPU time (μs)	Net (μs)	Total (μs)
Initialization	Manger 1	15	65	80
	Manager 2	13	65	78
Two Phase Commit	Manger 1	7	65	72
	Manager 2	7	65	72
	Coordinator	12	65	77

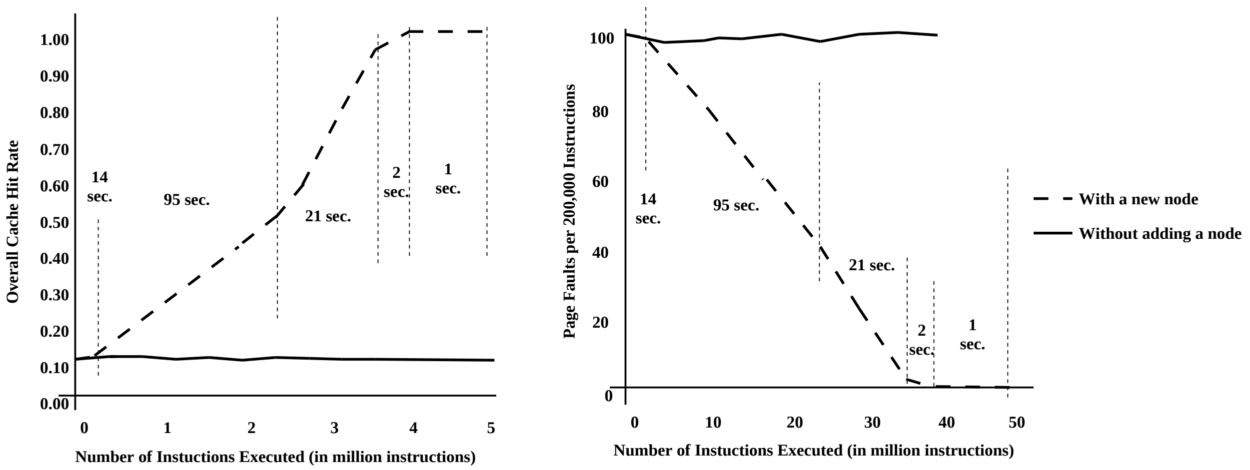
Table 3: Path length of migrating ownership from manager 1 to manager 2

4.2 Dynamic Change of Configuration

We use a data mining application to test the capability of Dynamo to adapt dynamically to the changes of the configuration of a LAN. For this experiment, we have used the *M-M Dynamo* version. Specifically, we use an association rule discovery program [Agr93] for this experiment. Typically, an association rule discovery program is highly CPU intensive. The program in our experiment spends approximate 95% time in computation with only 5% in data access (e.g., fetching data from disk), assuming the working set can fit in the main memory. The input data to the association rules program in these experiments is trace data from a web search engine run by Scour Inc. Scour [Scour] is a World Wide Web multimedia search engine that indexes web pages from the Internet that contain multimedia content such as images, audio or video. Scour's database currently contains 2.5 Million index entries for web pages with multimedia content, and is about 1.2 GByte in size. The search engine receives about 300,000 database queries (600,000 total page views, 6 million total HTTP hits) per day, and runs on 4 Dual PII Linux servers. The Scour web server maintains a large index, and retrieves the answer set from its database, returning the results to be displayed by the user's browser. Each user is identified by his/her IP address. A transaction contains identifiers of the set of web pages a user visits within a specified time frame. Scour keeps trace logs about users' access to its database and contents which we have used for this experiment. The association rules indicate likely pages a user will access after he/she has accessed a certain set of pages in the same session. This is especially useful for web server memory management in deciding which pages are better to keep in the main memory cache.

There are 40 trace logs in the Scour trace data set each of which represents one day of activity. Each consists of about three to four million hits with a size of about 35 MB. We executed one association rule discovery program on three trace logs on one Ultra-Sparc1 workstation with 80MB main memory. The working set of this program is approximately 120MB which exceeds the main memory. Then we added another node with the same resources. Figure 4 (a) and (b) demonstrate the effect of Dynamo on the cache hit rate (includes both local cache hits and global cache hits) and page faults, respectively.

It is evident that for the first 2 million instructions after adding a new node, there is no significant effect on page faults and cache hit rate because most of the referenced pages are still residing on disk. However, after 2 million instructions, we can see that the cache hit rate rises and page faults decreases because more and more referenced pages will be cached at the newly added node. Finally, after four million instructions, all pages in the working set are in either the local main memory or the remote node's main memory. Thus, a 100% hit rate and 0 page faults are achieved with 2 machines. The results depicted in Figure 4 show the elapsed time to achieve the 100% cache hit



(a) Overall Cache Hit Rate **(b) Page Faults**

Figure 4: Adapt to the change of Configuration

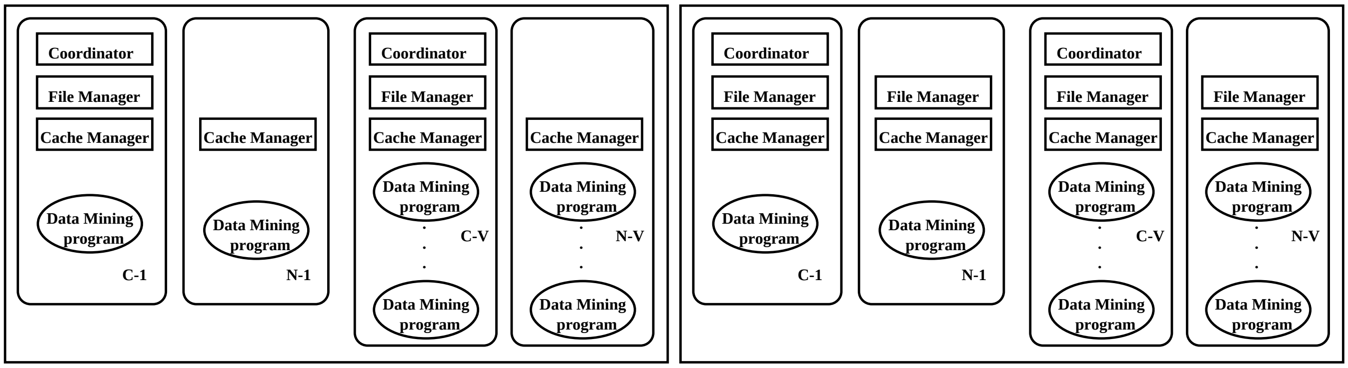
rate. It takes approximate 132 seconds to reach 100% hit rate and 0 page faults. Before adding a new node, it takes approximate **7 seconds** to execute one millions instructions (most time is spent on fetching data from *disk*). After adding a new node, it takes about **0.1 second** to execute 1 million instructions (most time is spent to fetching data from the *remote node*). This experiment demonstrates that a speed up factor of 70 is achieved by adding one node, and automatically making use of the system's resources.

4.3 CPU Intensive Applications

In these experiments, we examine the effects of Dynamo on CPU intensive applications. Here, we have chosen the same data mining application as the previous experiment as our test bed, and employ four Ultra-Sparc1 workstations, each with 80 MB main memory. To exploit the performance impact of the differences between the four versions of Dynamo, we configure the system slightly differently. As depicted in Figure5, For the *C-S* and *C-C Dynamo* testcase, we use the four machines as following: two nodes are configured as file managers and coordinators each one owns (i.e., manages) 20 log files while the two other machines are only used for caching. For the *M-M* and *I Dynamo* testcase, we configured 2 nodes as coordinators while the file and cache managers are dynamically migrated to any node for execution.

In both testcases, we execute a single data mining program on two machines. At the same time, we vary the number of data mining programs running on the other two machines. We run the same set of programs on all four implementations of Dynamo. We name the four nodes: C-1 (coordinator with 1 program), C-V (coordinator with variable number of programs), N-1 (no coordinator with 1 program), and N-V (no coordinator with variable number of programs).

Figure 6(a), (b), (c), (d), and (e) show the average performance of programs on each type of node with respect to the number of programs running on the variable nodes, respectively. We also ensure that there is no overlap of input data files for all concurrent executed programs so that data portions of the working sets are disjoint. The results show that *I Dynamo* has very close to same performance as *M-M Dynamo* for this experiment because intention set does not bring too much overhead.



(a) Setup for C-C and C-S Dynamo **(b) Setup for M-M and I Dynamo**

Figure 5: Setup for C-S, C-C, M-M, and I Dynamo

The performance of the program running on N-1 node with *C-S Dynamo* does not vary with the change of workload on other nodes because the node is “isolated” from other nodes. The performance of the program with *C-C Dynamo* is impacted slightly because the N-1 node will serve as a cache manager for programs on other nodes. The *M-M Dynamo* slightly outperforms *C-C*, due to the fact that the requests to the manager are communicated via IPC instead of message passing. *M-M Dynamo* outperforms the *C-S* when the number of programs on the variable nodes is small because the IPC versus message passing benefit outweighs the overhead of serving cache pages to other nodes, but it is outperformed slightly by the *C-S* model when the number of programs on the variable nodes is large because it has to serve more pages to the programs running on the variable nodes.

The average performance of the program running on C-1 node with *C-S Dynamo* and *C-C Dynamo* degrade to a certain degree due to the greater demands from the variable nodes, as shown in Figure 6 (b). On the other hand, the performance of programs with *M-M* and *I Dynamo* degrades at a slower pace because the file manager only manages the set of data it is currently accessing.

Now let’s examine the programs running at the N-V node as illustrated in Figure 6 (c). A node can hold at most 2 data mining programs’ working set at a time, so thrashing occurs for more than two programs. However, thrashing does not occur for *C-C Dynamo* until four applications are running. *M-M Dynamo* outperforms the *C-C* because of the benefits of IPC vs. message passing.

Figure 6 (d) depicts the performance on the C-V node. *C-C Dynamo* degrades fastest because memory thrashing begins to occur when more than 2 programs are running. *C-S Dynamo*’s performance degrades at a slower pace because the pages will be evicted to other nodes first. *M-M Dynamo* outperforms *C-C Dynamo* for two reasons: (1) the application to file manager interface is an IPC instead of message passing; this effect is also shown at other nodes. (2) The number of files owned by the manager on C-V is less. For our data mining program, we found that 8% of the overall path length is spent in the file manager.

Last but not least, the overall average response time of all programs for each implementations of Dynamo with respect to the number of programs on the variable node is illustrated in Figure 6 (e). It is clear that the most benefit comes from the cooperative caching (27% when the number of programs on each variable node is 3) , but the migration of management (*M-M Dynamo* over *C-C Dynamo*) also presents a significant advantage (25% when the number of programs is 4 on each variable node.)

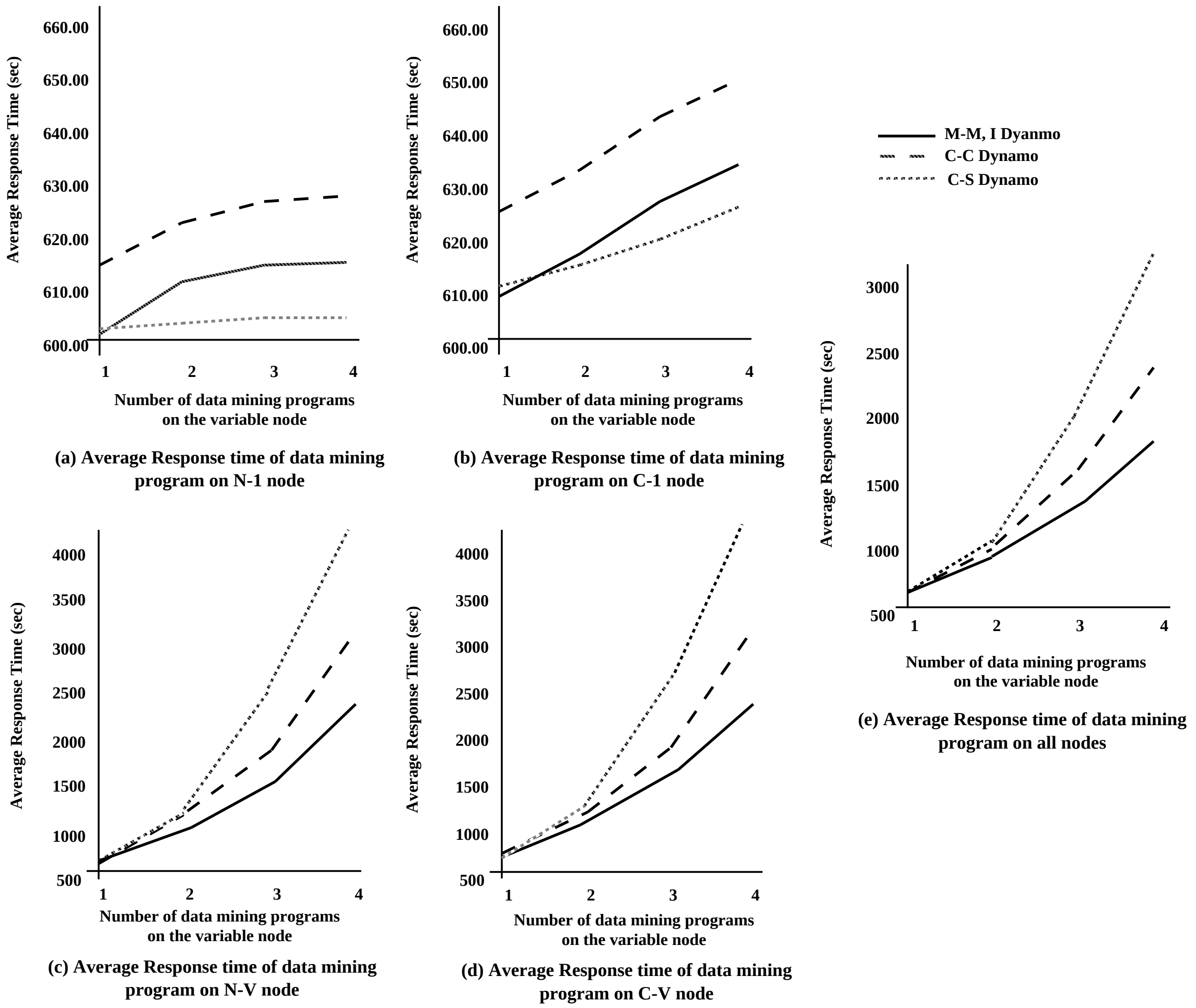


Figure 6: Performance of Data Mining Programs

Figure 7 shows the average performance of the data mining programs as a function of the ratio of the aggregate size of all working sets to the global memory size (W-G ratio). When the W-G ratio is small, i.e., each node's working set can fit in its own local cache, the *C-C* is slightly outperformed by *C-C Dynamo* due to the overhead of cooperative cache. With an increase of W-G ratio, the *C-C* and *I Dynamo* begin to show its benefit. *C-C Dynamo* has its largest benefit when the W-G ratio is almost 1, then the benefit decreases slightly because memory to disk trashing occurs. *M-M* and *I Dynamo* have the best performance overall because they not only employ cooperative cache, but also migrate the management so that the manager is located closer to the applications.

4.4 World Wide Web Caching

From the previous sections, we can see the benefits of management migration. In this section we analyze the benefits of intention set based cooperative caching. We used the Scour trace log that recorded the queries submitted to Scour search engine within a period of four and half hours from midnight to 4:30am of October 4, 1998. The log consists 10,000 queries. Scour search engine fetches on average 14 pages per query.

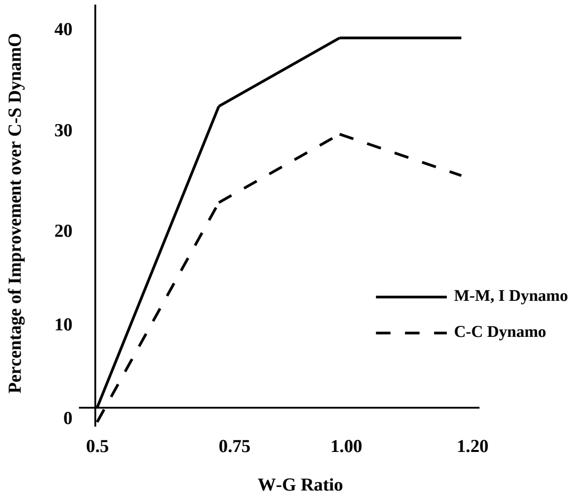


Figure 7: Working Set Size vs. Global Memory Size

We run the same set of queries from the log against the four different Dynamo implementations. For the experiment with *C-S Dynamo* when a query comes in, it is assigned to a server machine with the least CPU load. This routing technique is used by the current Scour web server.

The second experiment was run on *C-C Dynamo*. Here, an incoming query is routed via a hash function to a node. The hash function hashes the first key word used in the query to a set of nodes; the incoming query is then assigned to the server with the least CPU load in the set of candidate nodes. The routing scheme for the experiment with the *I Dynamo* is the same as the *C-C Dynamo*. For a given key word, there is a set of data associated with it. Then the nodes to which the key word hashes will form an intention set for the set of data associated with the key word. For example, if a page associated with the key word "car" is to be evicted from one node's main memory, the system tries to evict the page to a node within the intention set for "car" if there is some cache available on these nodes.

	C-S Dynamo	*C-C Dynamo*	*I Dynamo*
Local Cache Hit Rate	30%	45%	58%
Global Cache Hit Rate	N.A.	55%	42%
Response Time (sec.)	4.5	1.2	0.9

Table 4: Average Performance of Scour Search Engine

Table 4 shows the average response time of each query with three implementations of Dynamo. The *M-M Dynamo* has virtually the same performance as *C-C Dynamo* so that we did not show it in Table 4. It is evident that *C-S Dynamo* has the worst average performance due to the low cache hit rate. *C-C Dynamo* has a significant benefit over *C-S Dynamo* due to the cooperative cache. It improves over 70%. In addition, the *I Dynamo* has an additional 25% improvement over *C-C Dynamo*. The improvement comes from the fact that local cache hit rate improves significantly.

5 Conclusion

In this paper, we present a new adaptive, distributed local area network file system with network attached storage, Dynamo. It employs intention set technique so that the local cache hit rate can be improved. Furthermore, Dynamo can dynamically migrate management from one node to another so that it can adapt to the change of physical configuration quickly. We show that Dynamo can outperform the file system with the traditional cooperative cache with a significant margin.

References

[And96] 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.

[Agr93] 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.

[Cao95] 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.

[Cao96] 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.

[Car94] 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.

[Dah94] 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.

[Dias89] 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.

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

[Fee95] 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.

[Fra92] M. Franklin, M. Carey, and M. Livny. Global memory management in client-server DBMS architectures. *Proc. of the 18th VLDB Conference*, 1992.

[Gib97] 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.

[Gro96] 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.

[Lef91] 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.

[Lis96] 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.

[Cas97] 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.

[Lyn96] Nancy Lynch. *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.

[Sar96] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. *Proceeding of Third Symposium on Operating System Design and Implementation (OSDI)*, 1996.

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

[Tan95] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobb's Journal*, February 1995.

[voe97] 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.

[Voe98] 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.