Advancement Proposal
Storage Embedded Networks (SEN)
and
Adaptive Caching using Multiple Experts
(ACME)

Ismail Ari
ari@cs.ucsc.edu

UCSC-CRL-03-01
June 2, 2003

Department of Computer Science
University of California, Santa Cruz

ABSTRACT

The gap between CPU speeds and the speed of the technologies providing the data is increasing. This causes the performance of processes to be limited by the performance of the storage devices, the networks and the buses. Furthermore, the number of CPUs that share these data access resources is growing exponentially. Caching, prefetching and parallelism are some of the techniques used today to cope with I/O latency and system scalability to support more users.

This paper describes the two major contributions of our ongoing research on distributed data access. The first contribution is the design of the Storage Embedded Networks (SEN) architecture that aims to improve user response times and scalability on the Internet by better distribution of caches. SEN architecture is composed of trusted routers embedded with volatile and non–volatile storage that snoop bypassing objects for caching. Requests are checked by every hop, thus ensuring the transmission of the closest copy on the data path and load reduction at the upstream. The two main control overheads of other architectures, connection establishment and continuous cache communications, do not exist in SEN.

The second contribution is the design of adaptive caching schemes using multiple experts, called ACME, that manage the SEN caches and further improve the hit rates over static caching techniques. Machine learning algorithms are used to rate and select the current best policies or mixtures of policies via weight updates based on their recent success. Each adaptive cache node can tune itself based on the workload it observes. Since no cache databases or synchronization messages are exchanged for adaptivity, the clusters composed of these nodes will be exceedingly scalable and manageable.

We propose to extend our preliminary designs and analysis in two directions. The first is to compare the Storage Embedded Networks (SEN) with the existing hierarchical and distributed cache clusters in terms of user response times, network bandwidth usage, server load reductions and scalability. For this part we will run large scale simulations over realistic topologies using real web proxy, file system and raw disk traces. We will start these comparisons with static caching techniques. In the second part we will introduce the adaptive caching techniques to eliminate manual tuning and manual topological placement of static caches. We will measure the performance improvements gained by using the adaptive techniques and probe the performance limits by theoretical optimal algorithms. We will also quantify the time and space complexities of our schemes. Real system implementations will help us optimize our designs.

Keywords: Simulation, multi-level caches, web hierarchy, distributed, adaptive, topology generator, filtering, heterogeneous caching.

Contents

List of Figures

1. Introduction

The number of users connected to the Internet is growing exponentially. Satisfying so many users with fast response times or "low latencies" while transparently saving network bandwidth demands efficient distributed caching techniques. The data access latency problem in a single host is related to the discrepancy between the processor and disk I/O speeds [28, 86, 90, 89, 87]. In remote data accesses the network latency is added to the I/O latency at the servers [28, 7] further reducing the performance of the applications. Internet traffic analysis has shown that latency has improved, but not exponentially, from 500 ms to 100 ms and packet loss rates have dropped from 25% to 5% since 1993 [3].

Providing persistency of data along the path of traversal with a proper consistency vision enables reuse of the objects thus avoiding useless retransmissions. This approach reduces user response times as well as reducing the bandwidth usage in the network and the processing power usage on the server side, allowing resources that were once spent to do duplicative work for sending the same objects over and over to be allocated to provide richer content and higher quality service for all. Sources of redundancy in the Internet include undetected packet retransmissions, client sharing and poor version management [74]. The adverse effect of these redundancies will increase because of the increases in the number of mobile and wireless clients, file sizes [48] and percentage of dynamic objects. Exploiting any correlations and duplications between the requests is crucial.

SEN devices are routers with embedded volatile (DRAM) and non-volatile (MRAM [72], disk, MEMS [51]) storage to be used for object caching via object snooping in trusted routers. Requests are checked by every hop, thus ensuring the transmission of the closest copy on the path and load reduction at the upstream. Our vision is the use of storage physically embedded into the network device to save the overheads of extra messaging with external cache engines [36]. We use globally unique content–derived naming for object identification and define a new object transport protocol to carry the objects. There have been great efforts to provide scalable caching solutions that cooperate by exchanging messages with peers [33, 40, 45] or by inquiring a central database to locate cached copies of objects [67]. However, scalability has remained to be a major concern.

Enormous research efforts have also been put into characterizing the Web [7, 21] and file system [86] workloads and many static cache replacement policies have been invented. Today, robust static policies that work well with a wide variety of workloads are embedded into the systems [59, 20, 84]. Unfortunately, these policies cannot adapt to changes in workload and network topology and become suboptimal [99] when the conditions change.

Many factors increase the complexity of today's systems [15] in which caching is used. First, the characteristics of workloads change over short and long periods of time. Second, workloads mix when a system simultaneously serves multiple workloads generated by heterogeneous applications. Third, the characteristics of access to metadata and data are different. Finally, as the location of a cache node in the network topology changes the observed workload changes. This load is different from the load seen at the edges. This is called the "filtering effect" [8]. Recent research shows that these filtering effects [8] in a hierarchy of caches can change the nature of an otherwise predictable workload such that the higher layers are effectively useless [102, 27]. In these complex scenarios analytical modeling is daunting, manual tuning is tedious [14] and making wrong decisions has extreme monetary and performance costs.

Our machine–learning–based adaptive caching scheme (ACME) is motivated by these challenges of making caching decisions within complex systems in real–time and under dynamic conditions. We consider all previous cache replacement algorithms to be experts and register them into a pool with initially equal weights. When a new algorithm is invented we add it to our expert pool

and let it prove its success. We do not invent any new cache replacement algorithms, but use the existing ones more effectively. As the requests are made by the clients and the workload proceeds, the weights of experts are automatically changed by the computationally simple but powerful machine learning algorithms based on their success on selected metrics such as the *hit rate* or the *byte hit rate*. Hit rate is the percentage of all the documents accessed by the clients that are found in the cache and byte hit rate is the percentage of all the bytes accessed that are found in the cache. Each adaptive node is a self–governing or "autonomous" entity. Since no cache databases or synchronization messages are exchanged the clusters composed of these autonomous cache nodes will be exceedingly scalable and manageable. Machine learning algorithms [56, 70] have previously been successfully used in addressing non–trivial operating systems problems [54, 55] such as the disk spin–down problem in mobile computers.

In Section 2 we review the current state of caching in file systems and web proxies. In Section 3 we introduce our Storage Embedded Network (SEN) architecture. In Section 4 we focus on the adaptive cache design and in Section 5 we present some preliminary results. In Section 6 we propose extensions to our current work and we conclude in Section 7.

2. Related Work

Caching is used at all data access paths [101] and at all abstraction levels (file/record, block) in modern storage architectures as illustrated in Figure 2.1. However, most caches still depend on robust static cache replacement algorithms such as Least Recently Used (LRU) to decide on the objects to be ejected.

In this section we will review previous research in caching in four major groups. First, we will look at static cache replacement policies. Second, we will review caches in a single host. Third, we will examine some popular distributed file systems and the general issues with distributed systems. Fourth, we will analyze the hierarchical and distributed web caches. At the end we will have an overview of prefetching, which attacks the same problem as caching.

2.1 Static Policies

Table 2.1 lists some very popular and recently proposed criteria and the policies that use these criteria to make local replacement decisions. Random, First-In-First-Out (FIFO) and Last-In-First-Out (LIFO) do not require any information about the objects to be replaced. Time, frequency and object size are the most commonly used criteria for local replacement decisions. Least Recently Used (LRU) uses recency of access as the sole criteria for replacement, while Least Frequently Used (LFU) uses frequency of access. SIZE replaces the largest object and Greedy-Dual-Size (GDS) [59, 29] replaces the object with the smallest key $K_i = C_i/S_i + L$, where C_i is the retrieval cost, S_i is the size and L is a running age factor. GDS with Frequency (GDSF) [20] adds the frequency of access, F_i, into the same equation and replaces the object with the smallest key $K_i = (C_i \times F_i)/S_i + L$. LFU with Dynamic Aging (LFUDA) replaces the object with minimum $K_i = (C_i \times F_i) + L$ [20]. Lowest Relative Value (LRV) [84] makes a cost–benefit analysis using the access time, access frequency and size information about objects.

Hashing or more complex Bloom filters [45] on object IDs are often preferred for local decisions in the building blocks of a global system of caches. If the ID hash implies that the neighboring

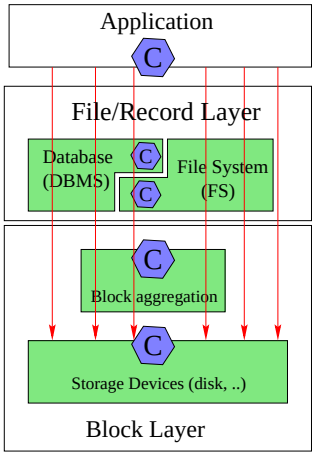


Figure 2.1: Access paths from applications to the storage devices. (Slightly modified version of SNIA Shared Storage Model access paths graph in Section 4.6 [101]). Caches are used at all layers. Today the *network* can also go between any of these layers and boxes.

criteria	algorithm
–	Random, FIFO, LIFO
time	LRU, MRU, GDS, GDSF, LFUDA, LRV
freq	LFU, MFU, GDSF, LRV, LFUDA
size	SIZE, GDS, GDSF, LRV
retrieval cost	GDS, GDSF, LFUDA, LRV
ID	Hash, Bloom filter
hop-count	–
QoS priority	Stor-serv

Table 2.1: An extended taxonomy of existing and proposed cache replacement policies.

node should be caching that object then it could be replaced quickly. Hop-counts provide another set of criteria that can passively provide an indication of the logical location of a cache without resorting to full location-awareness. Up-stream hop counts are a loose measure of how far a cache is from the closest data source, while down-stream hop counts indicate logical distance from clients. Recent research [103] points to the benefits of keeping a record of access latency history per object, providing yet another potential caching criterion (e.g., it s wise not to discard items from the cache that are very costly to retrieve). Stor-serv [35] proposes Quality of Service (QoS) ideas used in networking to be applied to storage systems for giving differentiated services.

Table 2.1 does not intend to cover all the proposed algorithms; rather, our goal is to show two things. First, the possible criteria and the ways to use them are endless, therefore we need a flexible design for integrating new criteria. Second, the trend in cache replacement algorithms is towards finding the functions that unite all the criteria in a single key or value. Other taxonomies of time, frequency and size based policies are presented in prior work [59, 34].

2.2 Caching and Adaptivity in Local File Systems

Linux has a dynamic cache space management [24] that uses the primary memory unused by the kernel and other processes. If the requirement for primary memory increases, the space allowed for buffering is reduced down to a minimum of 16 pages. File–oriented memory page caching is used for read operations and block buffer caching is used for write operations. Blocks are kept in *buffer cache*, which is a circular doubly linked LRU list [88]. Other caches are the *inode cache* that is used to look up *inode* structures using the device/inode number keys and *name cache* (also known as the *directory cache*) that associates inode numbers to filenames. Both caches are managed by LRU replacement algorithm.

Roselli *et al.* [86] found that even small caches can sharply decrease disk read traffic but even very large caches have limited effectiveness in reducing the read misses beyond a point and this point is workload dependent. Thus in general there is no support for the claim that disk traffic is dominated by writes when large caches are employed [89].

Hybrid Adaptive Caching (HAC) [32] combines the virtues of page and object caching by adaptively mixing them, while avoiding their disadvantages. Object caching discards objects in a page that are cold (*i.e.* not used) while keeping the hot objects. HAC compacts the hot objects freeing some memory pages, thus reducing the high bookkeeping overhead of object caching. HAC was shown to outperform object caching.

Khalid and Obaidat have recently proposed neural–network based cache replacement algorithms [79, 62] for eliminating inactive cache lines and achieved 8.71% improvement over LRU.

Jacob *et al.* give an analytical model for hardware related design of memory hierarchies [58]. Processor caches and hardware related optimizations are out of the scope of this research.

2.3 Caching and Adaptivity in Distributed File Systems (DFS)

Distributed file systems that allow clients to cache file data and also allow sharing need to provide *cache consistency*; "a coherent view of multiple copies of data and metadata" [101]. They also need to have complex *cache management* routines to decide "which cache should or does hold what" [101]. Another challenge for DFS is *availability* that refers to tolerating partial system failures.

The file caches of Sprite DFS [77] change dynamically in response to changes in virtual memory requirements. Measurements of Sprite by Baker *et al.* [22] in 1991 showed that about 60% of the data bytes were read from client caches and average file cache size was around 7 Mbytes out of 24 Mbytes of main memory.

Andrew File System (AFS) [57] has two separate caches for status and data and both are governed by the Least Recently Used (LRU) algorithm. Status cache holds information such as file sizes and modification times and responds quickly to *stat* system calls. AFS transfers chunks of files and caches them in the data cache on the client local disk to provide scalability.

In the Serverless File System (xFS) [16] any machine can store, cache or control any block of data. If the data block is not cached locally the manager is consulted to query whether another client has cached this data. If the request could be satisfied from another client's cache then the blocks are directly forwarded from client to client to improve scalability. Otherwise, the correct stripe groups and the correct storage servers are found and the data is retrieved. Cooperative caching [40] by Dahlin *et al.* compares four different techniques for cooperation between clients and servers in a Local Area Network (LAN). These systems do not mention about adaptively changing their caching policy to track the changes in workloads as will be described in this paper.

Lots of focus has been made on providing consistency in DFS. In AFS the modifications to the file in the cache are only reflected to servers when the file is closed. The *callback mechanism* assures cache consistency and reduces the load on server by reducing the cache validation traffic. xFS has a *token–based* cache consistency on a per–block basis. Before modification of a block a client has to acquire its write ownership. In token–based approaches the server grants and recalls the read–write and read–only tokens. Client flushes its dirty blocks to the server upon recall of its token. Sprite [77] makes write–shared files uncacheable and flushes the caches when the cached files are opened by other clients. Zebra Striped DFS [53] follows the Sprite approach for consistency. Frangipani [95]-Petal [68] use locks for coherency and leases to deal with client failures. Many of these efforts conclude that write–sharing is rare enough that it is reasonable to pick the simplest consistency mechanism.

OceanStore [67] project aims to design is a global–scale persistent storage by elaborate replication, data location, consistency, access control and archival storage components. OceanStore design conceptually mentions about "promiscuously caching nodes, floating replicas of objects and a probabilistic algorithm attempting to find the objects near to the requesting machines". We believe, SEN architecture is a perfect match for providing such an infrastructure. However, SEN does not deal with deep–archival and concurrent update issues.

Parallel file systems also try to alleviate the I/O performance and scalability problems [64, 38, 78, 73], but are out of the scope of our current research.

2.4 Hierarchical and Distributed Web Caching

Web caches exploit reference locality [7] and allow shared data access. Hierarchical proxies such as Harvest Squid [33, 94] improve scalability over flat caches. They define parent-child relationships between the layers of the cache hierarchy thus forming a tree topology from bottom (leaf) to top layers. Clients connect to the leaf caches and directly send their requests to them. It is the duty of these leaf caches to find and return the requested objects. If the requested object is not found locally, the leaf cache connects to its parent cache and this parent to its own parent until the object is found; otherwise the last parent cache connects to the server on behalf of the client. The object is retrieved and transported back to the client via multiple store-and-forwards through the various proxy caches. Unfortunately, it usually takes long geographical distances and multiple network hops to go to the upper layers. The rule of thumb is [94] to avoid network hops because of the delay and uncertainty they introduce in the retrieval service. Upper layer caches can also easily become bottleneck nodes with long request queues and response times, since they support exponentially more clients [94]. Thus it is possible to get a hit in the caching system, but still perform poorly in response times due to the other overheads. SEN nodes do not setup connections and they forward requests as normal routers in case they do not have information about the requested object.

Misses become the ultimate worst case in hierarchical proxies, since all the time spent in making proofs of nonexistence. The rule violated here is that “misses should not be delayed” [94]. Tremendous research has been put into efficiently summarizing [45] and disseminating lists of cached objects among cache clusters, and some have been successful in improving over basic hierarchical proxy performance. Approaches that exchange information inherently limit scalability because every client action becomes a new piece of information or hint to be exchanged in the cluster.

By enabling persistence within the network nodes, SENs leave it to the clients to smoothly pull the data towards the edges, allowing unreferenced data to slowly move back to its ultimate persistence in the server. Even compulsory (first time) misses could be avoided by pushing popular content on some paths, making SEN of interest to Content Distribution Networks (CDNs) [1, 2] that currently provide persistence by establishing data centers all over the world. The two main control overheads, connection establishment and continuous cache communications, do not exist in SEN. The benefits of SEN are therefore similar but superior to those achieved by proxies.

Rodrigues *et al.* compare hierarchical and distributed web caching architectures [85]. They found that hierarchical caching achieves shorter connection times, reduces bandwidth usage, but can easily become highly congested at the higher layers. Distributed caching has shorter transmission times, but the connection times, bandwidth usage and administrative costs are increased. They propose a hybrid scheme where there is a caching hierarchy and certain number of caches that cooperate at each level of the hierarchy using distributed caching techniques.

Adaptive web caching [71] proposes that nearby caches self-configure themselves into a mesh of overlapping multicast groups and exchange messages to locate the nearby copies of requested data and to find out about topology changes. Scalability was a major goal in this design, but due to the vast amount of objects flowing on the Internet lots of control messages still need to be exchanged [94, 36]. There are also some other deployment problems with IP multicast [43]. In Summary Cache [45] and OceanStore [67] nodes use Bloom filters to summarize the contents of a group of caches.

Virtual cache management by Arlitt *et al.* [20] divides the proxy cache space into static partitions and lets a few successful policies work on separate partitions. Objects evicted from one partition go

to the next until they are moved out of the cache. Their results show that the performance is bound by the performance of the best partition [20].

Wong *et al.* [102] demonstrated the benefits of using demotions in a 2-level cache that represented client caches and a disk array cache. Demote operation moves ejected objects one more hop away from the client instead of discarding them, thus providing better *exclusive caching* to avoid useless duplications. This creates the effect of having one large unified cache. They also tried using different policies at different levels and found that LRU–MRU–Demotes was the most successful. However, demotions cause extra network overhead and are feasible in LAN or Storage Area Networks (SAN) with high-speed connections. Busari *et al.* also report that the use of heterogeneous policies [27] improves hit rates over usage of same static policy in multi–level caches [76].

In this paper, we confine our analysis to adaptive replacement policies for objects with static content. Detailed research on consistency issues in web caching can be found in related previous work [105, 30]. We also leave out the effects of distributed locking [25, 67] in file systems.

2.5 Prefetching

Prefetching is a technique to bring objects closer to the CPU before they are requested. Probabilistic techniques and frequency–based access histories have been widely used in the past to perform prefetching [52, 96]. Recently, program–based successor models have been proposed to do file prefetching. In these models names of applications are used as hints for prefetching [104].

On their web analysis Kroeger *et al.* [66] found that prefetching can offer more than twice the improvement of caching, but is still limited in its ability to reduce latency. With their workload 26% of the latency reduction was due to caching, 57% due to prefetching and 60% when both were employed.

Within the context of prefetching, adaptivity is used for buffer cache management to dynamically decide on the proportion of the memory to be allocated for the prefetched objects. If prefetching uses too much memory the buffer cache may be starved and if it uses too little memory then there is not enough space to benefit from prefetching.

Prefetching tries to improve the data access latencies just as caching. But, it is a subject orthogonal to caching and is out of the scope of this research. Details on prefetching [52, 96, 65, 104] and other types of file aggregations aggregations [9, 10] can be found in prior systems work.

3. Storage Embedded Networks (SEN)

Providing improved response times to exponentially increasing number of users is an ongoing research challenge. Latency is incurred either because the objects are physically stored far away or because they are highly popular and create hot spots of network and server load. Storage Embedded Networks (SEN) bring data closer to the clients and enable data sharing, thus reducing latency, network load and server load. Other clusters of caches also increase the performance over a single cache [17], however the trade–offs are the challenges in complexity, scalability, availability and administration [46].

SEN devices are routers with embedded volatile and non-volatile storage to be used for object caching in trusted routers. Requests are checked by every hop, thus ensuring the transmission of the closest copy on the path and load reduction at the upstream. Cache lookup is run in parallel with the route lookup, so that the basic forwarding task is not degraded. The operation of SEN routers is simple. The clients make requests using $< GUOID, offset >$ pairs via object transports described in Section 3.2. SEN routers snoop both the bypassing requests and data objects. If a local copy of the object exists, a SEN node responds by sending this local copy, otherwise it forwards the requests without delaying them as normal routers do. Similar to the Akamized [1] web sites, applications that want to make good use of SEN caching will have to include the Globally Unique Object ID (GUOID) of the embedded objects.

If caches on a path hold the same elements then a miss in one of them will also result a miss in the other ones. This is called *inclusive caching* [102]. We would like to achieve as much *exclusive caching* as possible between the collaborating caches, so the cluster has the effect of a one big unified cache to the users. We will used heterogeneous and adaptive caching techniques to provide exclusive caching. Many of the proposed architectures for caching clusters or hierarchies involve periodic message exchanges that may limit their scalability. A good cluster is formed when all unit caches in the cluster first do their best with the workload they observe and are able to change characteristics as their workload changes. After this infrastructure other intelligent techniques such as pushing or prefetching can also be utilized.

Adaptive caching techniques will be used to improve the hit rates of SEN caches over static caching. If the caching SEN node changes location by a change in the routing tables becoming an intermediate node, then adaptive policy will shift to satisfy the requirements of this new location. Therefore, adaptive caching is also very beneficial for mobile nodes. Since our scheme requires no explicit message or periodic database exchanges it is very scalable and allows the flexible construction of large SEN clusters.

For Content Delivery Networks (CDN) choosing the correct server that will lead to the fastest response to the client requests is a big challenge, especially “in the complexity of the real Internet” [61]. Measurements by Johnson *et al.* [61] show that neither of the two major commercial CDN services [1, 2] choose the optimal server consistently. We hope that SENs will provide a beneficial infrastructure for content delivery and push caching by both providing the persistence and ensuring the delivery of closest copies on the network paths. SEN infrastructure will alleviate the configuration, tuning and management complexities that CDNs face today.

3.1 Globally Unique Object Identification (GUOID)

To exploit correlations between client requests, a SEN object cache needs to identify all objects in a globally unique fashion, independent of sessions, connections, applications and protocol specific packet sequence numbers. Therefore, we choose to use the content-derived Globally Unique

Object Identification (GUOID) [6] to achieve connection independent naming. Content Derived Naming [6] scheme uses secure hash functions to derive an object's name from its content. Our calculations show that a 160 bit GUOID such as that generated by SHA-1 [18] could be used for long periods of time with very small probability of name clashes.

Whenever an object is modified, it essentially becomes a new object and is given a new GUOID value [81]. The clients make requests using $< GUOID, offset >$ pairs. As with the case of web pages, many objects have other embedded objects, mostly of static content, which do not change during the update operation of the initially retrieved indexing page. An update operation may only change the GUOID of the top-level object; if so, only that object would need to be retransmitted. Upon reception, clients can hash the contents and compare the result against the expected GUOID of the object to check *data integrity*.

3.2 Object Transports

We propose two possible transports for transferring and caching objects within SEN clusters. The first approach is an Open Systems Interconnection (OSI) Layer 4 (transport layer) solution that we call Object Transport Protocol (OTP). OTP runs on top of UDP/IP and carries objects identified by GUOIDs. OTP is not a totally new concept, but a generalization of the Real Time Protocol (RTP) [93], which is successfully being employed today to carry real-time traffic on the Internet. RTP introduces object awareness by tagging each packet with a globally unique Synchronization Source (SSRC) identifier and a time-offset for the real-time payload being carried. However, these specific fields make RTP suitable only for real-time traffic. Instead, the OTP header keeps a generic 20 byte GUOID for the object and an offset value for the packet being transmitted. Type and priority are other useful fields included in OTP header for application specific optimization and Quality of Service (QoS) differentiation, respectively.

The second approach is an OSI Layer 3 (network layer) solution that uses IP options for the exchange of GUOIDs and other useful information. An IP options solution was also proposed for providing Active Networking [100] services. The disadvantages of IP options solutions are quoted as the limited header space (40 bytes maximum) and the previous experience on the slow acceptance and slow deployment. The IP options solution, like OTP, is backward compatible. However, a caching service employing IP options solution will still need to be explicitly enabled by the hosts and routers just like the IP Explicit Congestion Notification (ECN) [82] service, until it becomes a common practice.

Backwards compatibility of new technologies is important, since it allows incremental deployment without disrupting the technologies in place. SEN routers are backwards compatible with the standard routers in use today, since current routers will forward IP packets as is without looking for OTP headers. This makes deployment of SEN devices an evolutionary process, where each added SEN router enhances the caching capabilities of the Internet. However, GUOID naming has to be used by those clients and servers that want to benefit from SEN caching.

3.3 Other SEN Related Technologies

We have seen hierarchical proxies and distributed systems in Section 2. This section reviews other specialized services that provide similar benefits to that of SEN generic architecture.

3.3.1 Packet Level Caching

Wireless links with high Bit Error Rates (BER), frequent packet losses, temporary disconnections and limited bandwidth can degrade transport performance dramatically. The situation gets worse when mobility is added; WAN traffic analysis [80] shows that even on wired links there are many forms of "pathological" network behaviors that require retransmissions. Unless packets are cached on the way, they will have to traverse the WAN links over and over again. There is potential for packet level reuse in this situation. Indirect TCP (I–TCP) [23], the Snoop protocol [12] and client-side TCP (C–TCP) [60] are some of the solutions proposed for network level packet caching and retransmissions. However, all these solutions are per TCP session and cannot capture correlations between applications, other local hosts and hosts distributed globally. SEN considers packets to be offsets of objects, therefore embracing support for caching at the packet level and reducing the need for WAN retransmissions. Compared to previous solutions, SEN is a generic, easy to deploy, cheap and effective solution that automatically encapsulates the specialized wireless and mobility solutions. There are also solutions that propose the delta encoding and compression of data [74].

3.3.2 IP Multicast

IP multicast is another mechanism for the delivery of content with reduction in network traffic. IP multicast does not have any redundancy in terms of sending multiple copies of packets over the same link, since packets traverse to the edge routers and then get sent to the multicast address. SEN avoids redundant packet transmissions on the paths from servers to clients, as does IP multicast. However, IP multicast solutions proposed urge senders and receivers to be online at the same time on the same multicast IP address. This strict promptness and synchronization requirement violates the demographics of streaming media (audio, video), where customers may wish to start receiving the same content with various time–shifts. SEN allows insertion of time–shifts between requests and is therefore superior to IP multicast. This anytime, anywhere (i.e. ad-hoc) multicast capability is a beneficial side effect of the scalable persistency for objects in the SEN design. Diot *et al.* analyze other issues for the IP multicast service that have limited its commercial deployment [43].

3.3.3 Layer4 (L4) and Layer5 (L5) Switches

L4 switches [49] look deeply into the network packets to determine the types of requests (e.g. HTTP) and L5 devices look more deeply to see what messages are being carried by these requests (e.g. Uniform Resource Locators). These switches act as gateways to tunnel certain traffic types to the associated port numbers of external cache engines that are specialized for this type of traffic. These solutions focus on a few popular protocols and work at the edges of the network because of the processing overhead involved. There is also the overhead of extra messaging with the external cache engines [36]. SEN supports caching for different data types by identifying all types as generic objects.

Slice architecture by Anderson *et al.* [13] provides network file service in LANs with network–attached storage. μproxy is a component in Slice L5 protocol that provides content–based request switching. μproxy is implemented as an IP packet filter and can reside "within the networks", but it must still reside (logically) at the end of a connection.

4. Design of an Adaptive Caching Scheme

> *"But if he will not hear thee, then take with thee one or two more, that in the mouth of two or three witnesses every word may be established." Matt. 18:16*

Adaptivity to changing conditions requires multiple characteristics to be embedded in one system. This is also true for an adaptive caching system. Therefore, we will have a *pool* of static cache replacement algorithms with different characteristics to decide on how to behave based on the observed workload. The challenge is to join the relatively weak predictions of so many experts into one highly–accurate prediction [92]. Expert systems [69], specifically machine learning algorithms [56, 70] have been successfully used for this purpose in the past to solve non–trivial operating systems problems [54, 55].

4.1 Rationale

As the characteristics of the workload change over time (minutes, hours, days) the hit rates of the static policies become suboptimal. In caching research the performance of different static replacement policies are usually measured by keeping a cumulative running average for the hit rate or byte hit rates. These values are reported after the "warm-up" period as the performance of that static policy for a given cache size and workload. However, if we measure the hit rates of these policies in subregions of the request stream we see that the best policy for different subregions maybe different as illustrated in Figure 4.1 and we call this *switching*. Choosing the "best current" policy is preferable over choosing the "best overall" policy if the costs of achieving the former can be justified with its benefits. We define the difference between the hit rates of the best current policy and a particular static policy as "the loss" of that static policy. The cumulative results hide the recent successes or losses of static policies.

Figure 4.1 shows the existence of switching in real workloads using Digital Equipment Corporation (DEC) web proxy trace [4]. This proxy served 14,000 workstations in DEC in 1996. We used trace of date 9/16/96 for this test. Twelve policies were tested at the same time using each 64 MBytes of cache space. Only the three, four dominant policies became the best and extensively appeared in the graph, therefore we just show these policies. The byte hit rates are measured in windows of 500 requests and only the best policy is plotted for each window. We see that the best policy keeps changing for different time slots.

Figure 4.2 is similar to Figure 4.1, but it explicitly shows the byte hit rate of the best overall static policy (LRU). The colored spikes on top of the LRU byte hit rates indicate that very frequently some other policies were better than LRU. The cumulative average of the difference between the byte hit rate of best policy and particular static policies, *i.e.* the loss of static policies, were calculated. The byte hit rate loss was around 3% for LRU and 5% for both LFUDA and GDSF. Our goal is to develop an automated scheme that will be able to either select the current best static policy or create a more successful hybrid policy by mixing the available static policies.

It is vital that the opinion of each expert is heard and considered at all times. If a highly opinionated group or decision–maker ignores the decisions of the experts that have made weak or unsuccessful predictions in the past, then group may run into the danger of only following one strong static expert (i.e. *monopoly*). When the conditions change to favor the previously weak experts this "so–called adaptive" system is bound to collapse since the alternatives have been starved during the course of events.

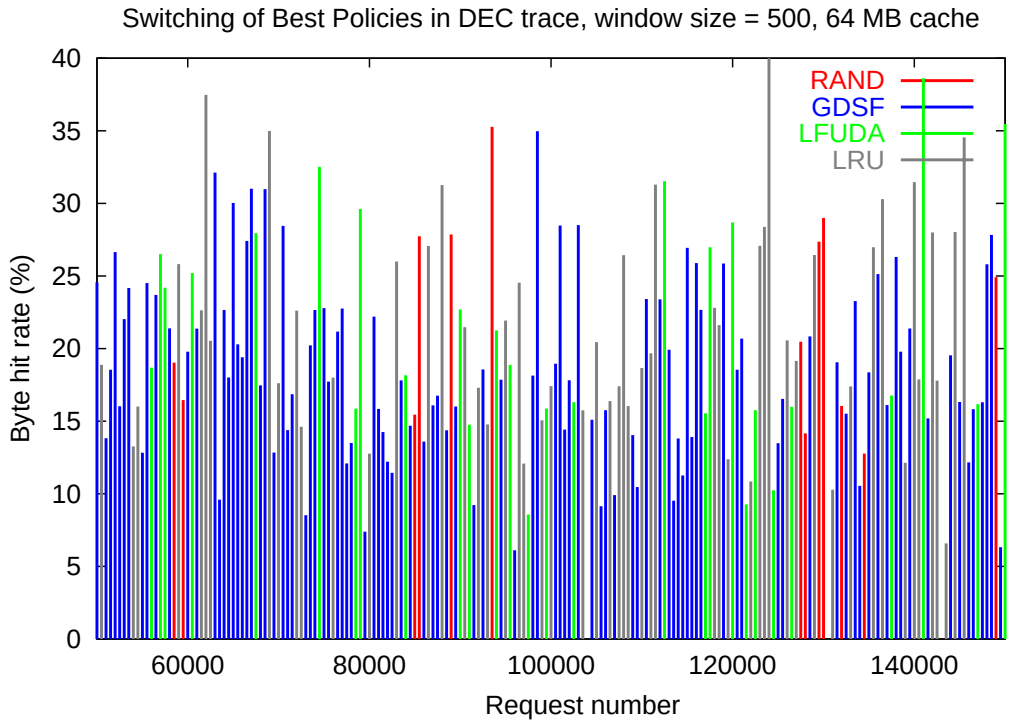


Figure 4.1: This graph shows the existence of switching of the best current policy in the DEC trace. The byte hit rate of the best policies with last 500 requests are plotted as bars. The cache size is 64 MBytes.

To illustrate this concept we wrote a simple synthetic request stream that favors LRU algorithm until 500 seconds and then changes characteristic to favor SIZE algorithm as seen in Figure 4.3. We see that a good adaptive algorithm implementation (Fig. 4.3a) looks at the recent success to quickly switch to using the SIZE policy maintaining a continuous high hit rate. Figure 4.3b shows that an implementation that only looks at the past performance cannot switch to the other good policies when the conditions change and is bound to be as good as the overall best fixed policy. SIZE policy has to exceed the overall maximum hit rate of the LRU policy for this switch to happen. The careful reader will notice that these two cases are actually the same except that the success history we look at in the second case is limited to a fixed number of requests instead on being all the history. How big or small the window or length of success history should be is an open research question that we will investigate.

Another concern is the amount of information in the workload. An adaptive algorithm based on learning will have its limits when the workload is completely random, since learning works whenever there is *at least some* information in the form of repetitive patterns. However, even with random request streams there is hope for improvement. To test this we created another synthetic load where 4096 unique objects were being requested with a random uniform distribution with one second inter-arrivals for one day or 86400 seconds. The sizes of variable size objects were uniformly distributed between [0,64 KBytes] and fixed size objects were all 64 KBytes. The cache size was chosen to be 4 MBytes, which is $1/64 \approx 1.563\%$ of the unique document space for fixed size case. Size–based algorithms are more successful with the variable object sizes in terms of hit rates (Fig. 4.4a), since they can replace big objects and hold more small objects. All policies perform similar when the objects are fixed size (Fig. 4.4b) and the hit rate is exactly *cachesize/unique-doc-space* $\approx 1.563\%$. An adaptive algorithm can exploit these facts without any human intervention.

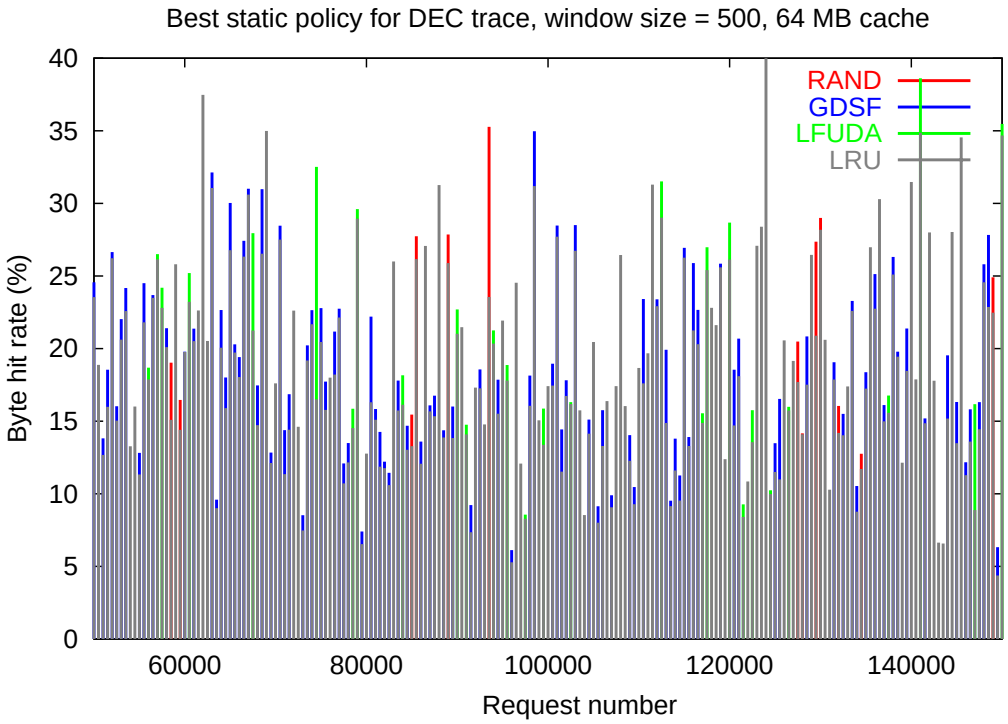


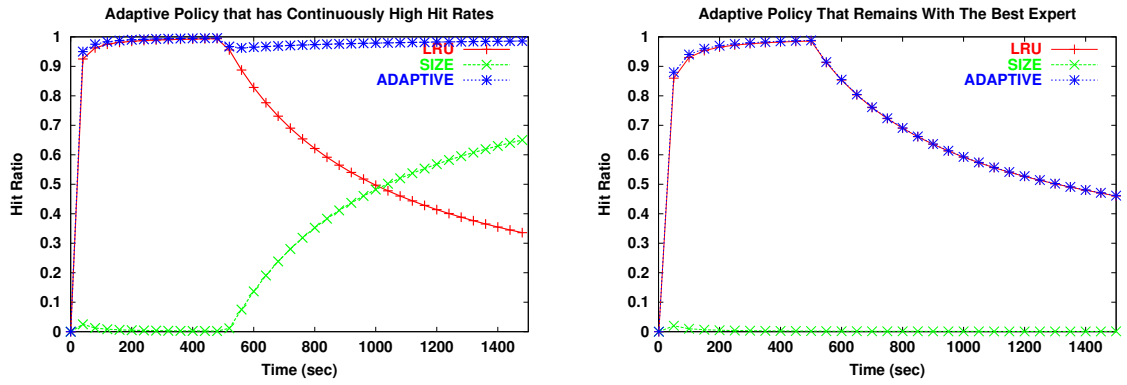
Figure 4.2: In this graph the byte hit rates of best static policy (LRU) is plotted. The colored spikes on top of the LRU byte hit rate bars show that there were some other policies better than LRU during that time period. The height of each spike indicates the percentage of byte hits we lost by using only LRU. Cumulative average loss for LRU was around 3%. For LFUDA and GDSF the average cumulative loss was around 5%. The cache size is again 64 MBytes.

One should note that the high hit rates in variable size case do not guarantee high *byte hit rates*, which may be a better metric for representing the improvements in user response times. In fixed size case, hit rates and byte hit rates will also be the same in percentage. The selection of the metric that leads improved *user response times* and thus improved user satisfaction is crucial. We will investigate different metrics.

Our implementations follow two intuitive directions. The first direction is a voting mechanism that emphasizes on finding a *democratic* or compromised solution to the caching problem [69]. The second direction is motivated by the games played in nature and depends on survival of the fittest, where *fitness* is determined by the success of the cache policy in reducing the mean client response times. Ideas from machine learning are used in all the directions followed. The simulated rollover algorithm described at the end of this section combines all the good features of these concepts.

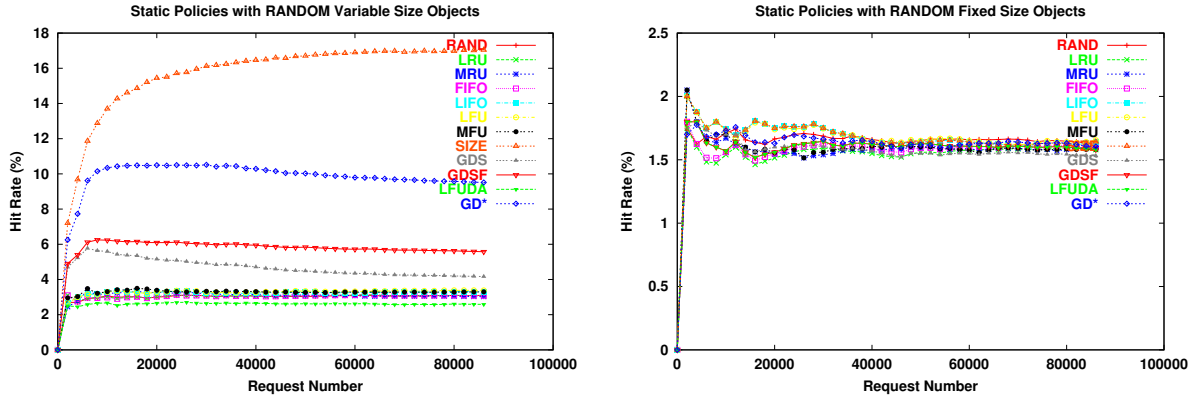
4.2 Voting Mechanism

Figure 4.5 illustrates the major components of our weighted voting–based adaptive design. We define a pool of *virtual caches* each simulating a single static cache replacement policy and an object ordering. Virtual caches act as if they own the whole physical or real cache, but they only keep object header information; not the actual data. On each request they indicate their *predictions* to the ACME (Adaptive Caching using Multiple Experts) module [19]. In the current implementation virtual caches simply say whether they would have got a hit (1) or miss (0) if they were the real cache



(a) (b)

Figure 4.3: Hit rate results for a synthetic workload that switches characteristic after 500 seconds. (a) Adaptive schemes that look at recent success can quickly switch to currently successful policies and provide continuous high performance (b) Schemes that look at the cumulative success will stick with the overall best policy suffering performance when this policy is no more favored by the workload



(a) (b)

Figure 4.4: Hit rate results of 12 static policies for random uniform workload with (a) variable size and (b) fixed size objects.

and this is considered as that policy's prediction. Although objects are ordered with the highest weighted–vote in this implementation, the true outcome is only compared to the hit/miss prediction, but not the weighted–vote. Both the caching and replacements are done based on votes. The objects with the highest weighted–votes stay in the cache. The policies that predict the workload well are rewarded by an increase in their weight and the policies that lead to wrong decisions are punished by a decrease in their weight using the machine learning algorithms described in this section. Over time the real cache ordering will look like the ordering of virtual caches with the highest weights, but will still be a mixture of multiple policies.

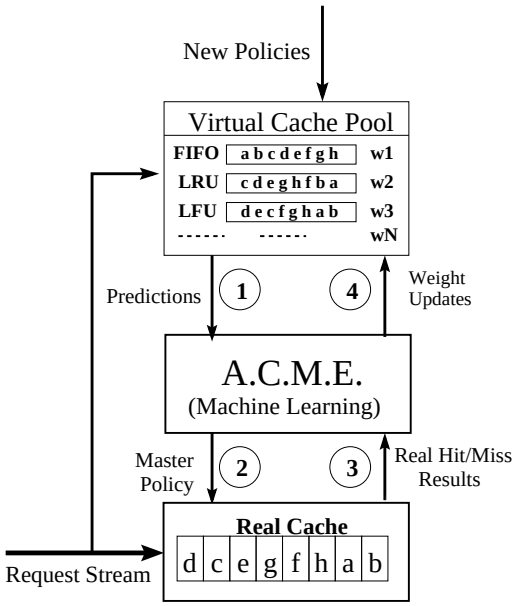


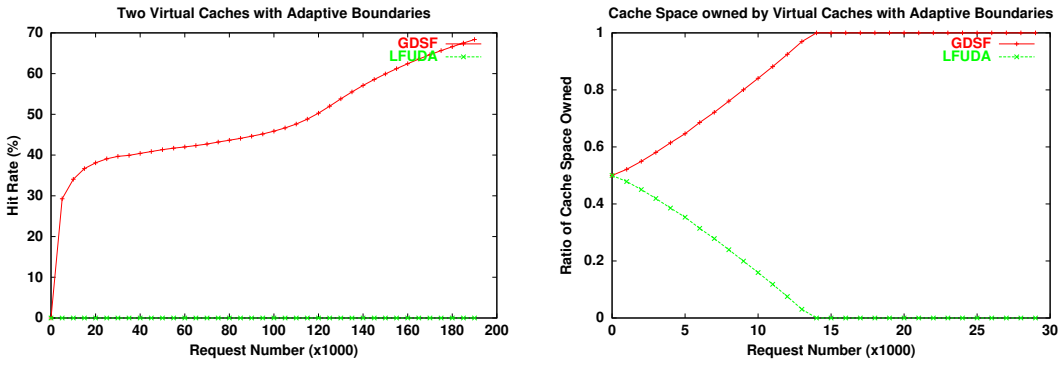
Figure 4.5: Design of Adaptive Caching using Multiple Experts (ACME). Virtual caches make predictions on whether objects should be cached or replaced. A weighted average of these predictions defines the master policy that manages the real cache. The real outcomes are compared to the predictions and used for weight updates of the virtual policies.

4.3 Game Theoretic Approaches

We also apply various ideas from the Game Theory to the adaptive caching problem. For example, a *dynamic boundary adaptation* is achieved by placing two cache replacement policies in one shared cache space and letting them play a *tug of war* or *rope pulling* game to adjust the boundary between them. The strength of a policy is determined by its hit rate or byte hit rate performance. Every time a policy gets a hit it grabs a certain percentage of the opponent's cache space. If the other policy also gets a hit it can get its space back. In this experiment we found that if one policy is good for enough time to starve the others by taking their spaces then there is no return, since the inferior policies cannot prove their success anymore. Figure 4.6 show the cache space ownership and hit rate results for a 2 policy game. As GDSF takes over the cache it becomes much harder for LFUDA to regain control.

Policies that perform well in terms of hit rates may be worse in terms of byte hit rates. For example, in cases where we measured success based on the hit rate as ultimate performance measure we have noticed that our byte hit rates suffered. If loss is calculated based on the *sizes* of objects that were hit or missed, then the policies with better byte hit rates would be favored.

Another game possibility is to define a third party or *resource manager* to determine the amount of cache space owned by each policy. Imagine all cache policies as *species* competing for food (documents or objects) in a *habitat* (cache). The *fitness* of a species is based on how well it eats. This will be related to its hit rate or byte hit rate success. The frequency of a species in the population depends on its fitness and highly fit species may populate, thus starving the others if there is no controlling force over them. Preserving the variety is crucial, since a drastic change in the environmental conditions may wipe-out a previously highly fit species favoring the previously weaker ones. In nature this is done by the *predators* that probabilistically prey on the most frequent or easy to catch species. Predators protect diversity or mixing among species by avoiding the most



(a) (b)

Figure 4.6: Dynamic boundaries may cause the initially inferior policy to be quickly starved to zero weight.

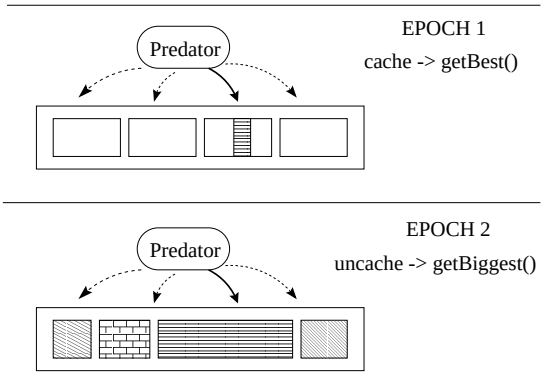


Figure 4.7: There are several trade–offs in the predator design. A good design both has to avoid duplications of objects in the cache and avoid assigning the unsuccessful policies the most popular objects as this would cause an unfair inflation of their weight making them look like successful policies.

fit species from starving the others. Our predator or *Resource Manager* implementation illustrated in Figure 4.7 assigns the objects to caches and manages cache spaces. However, there is a trade–off between these two duties. Since the cache space is valuable we cannot allow duplications, therefore we have to decide which policy gets to manage certain pages. Unfortunately, a lucky draw at the beginning may assign a popular page to a weak policy and cause this policy's weight to be unfairly inflated due to numerous hits. To avoid these problems we define a pool of *virtual caches* each simulating a single cache policy and object ordering. Virtual caches act as if they have the whole cache, but they only keep object header information; not the actual data.

4.4 Machine Learning Algorithms

In machine learning terminology "experts" are algorithms (*e.g.* LRU) that make predictions, denoted by the vector x_t. In the current implementation caching experts simply say whether they would have got a hit (1) or miss (0) if they were the real cache and this is considered as that policy's prediction. We refer the users to Section 4.5 for implementation details, since this section intends to give a generic overview of experts, updates and other machine learning concepts. Weights of experts

(w_t) represent the quality of predictions. The *Master Algorithm* [55] predicts with a weighted average ($y_t^{'}$) of the experts' predictions:

$$y_t^{'} = x_t.w_t \tag{4.1}$$

The weights of the policies are generally initialized equally as $w_i = 1/N$, where N is the number of experts. Instead of a constant initialization, past experience on the success of policies may also be used to bias the initialization vector. However, one should make sure that this process is done in an automated fashion, but not via manual tuning.

Depending on the true outcomes (y_t), which in caching case is hits and misses, we incur *loss*. For example a simple loss function may be: $Loss(y_t^{'}, y_t) = (y_t^{'} - y_t)^2$, called the *square loss*. Then this loss is used to update the weights. Many forms of *loss update* have been proposed in the literature [70, 97]. The *Vovk Update* [97] given below is a generalized version of *Weighted–Majority* algorithm by Littlestone and Warmuth [70].

$$w_{t+1,i} = w_{t,i} \frac{e^{-\eta \times \text{Loss}_i}}{\sum_{i=1}^{N} w_{t+1,i}} \quad \text{for } i = 1, \ldots, N \tag{4.2}$$

where parameter η is called the *learning rate* and the summation in the denominator provides normalization of weights between [0,1]. The weights at time t are multiplied with the exponential factor to obtain the new weights to be used at time $t+1$.

Because of the exponential factor in the formula it is claimed that the loss updates learn too fast, but do not recover fast enough [56]. Therefore, with loss updates the weights of many experts can quickly become zero or very close to zero. Left with very little trust or cache space these inferior algorithms are never ever given a chance to prove their success in the future. *Share algorithms* [56] try to ensure that weights do not quickly become zero, so that an inferior policy can recover its lost weight if it starts performing well. In the Share algorithm each policy is forced to contribute a loss–proportional part of their weight into a pool:

$$pool = \sum_{i=1}^{n} \left(1 - (1-\alpha)^{\text{Loss}_i}\right) w_{t+1,i}. \tag{4.3}$$

where α denotes the *sharing rate*. After this sharing, the pool is redistributed by giving equal shares to all policies:

$$w_{t+1,i} = (1-\alpha) w_{t+1,i} + \frac{1}{N-1}(pool - \alpha w_{t+1,i}) \text{ for } i = 1, \ldots, N \tag{4.4}$$

However, because of the additional operations needed the Share Algorithm is computationally more intensive than the simple loss updates. A cost–benefit analysis will be required for comparison.

Machine learning algorithms [56, 70] have successfully been used in addressing non–trivial operating systems problems [55] such as the disk–spin down problem in mobile computers. To conserve precious battery power, mobile computer hard drives are spun down after a certain time-out period. Unfortunately, spinning a disk back up consumes more energy per unit time than normal operation. The optimal adaptive on-line algorithm would spin down the disk immediately if the upcoming idle period would exceed the spin down cost of the disk. This would provide the maximum power savings for a given workload, while never spinning down a disk for idle periods that are too short to justify the additional cost of spinning it up again. The goal of adaptive disk spin-down algorithms is to approach the behavior of this optimal algorithm by observing the disk activity and dynamically adjusting the disk time–out. Helmbold *et al.* used Share algorithm to

attack the problem and this work resulted in the most power-efficient adaptive algorithm to date and was "often using less than half the energy consumed by a standard one minute time–out" [55, 54]. This success motivated our research in handling adaptive caching in complex, dynamic systems. Other details can be found in our previous work [50] and other machine learning literature [97, 56, 70, 54, 55].

4.5 Simulated Rollover Algorithm

Switching the control of physical cache from one best policy to another quickly between the two consecutive regions of a request stream is a challenging task. There is some latency between the time when the current best algorithm is found and the time when the contents and the ordering of the real cache closely resemble that of the current best policy [50]. Simulated rollover algorithm [50] tries to minimize this latency as much as possible.

We have two separate Virtual Policy Pools (VPP), VPP1 and VPP2. Each pool has the same set of policies all with equal virtual cache sizes which are also equal to the physical cache size. The virtual policies only keep the metadata for the objects they would cache along with an identifier for the physical objects in the physical cache. The metadata overhead of this implementation is considerable, but could be improved.

The first set of virtual policies, those in VPP1, act in the same way as those described in Figure 4.5 and are only used for weight update purposes. Whenever they miss an object they are punished with a weight decrease and then all the weights are renormalized to add up to one as shown in Figure 4.8. Over time the successful policies will have larger weights. All policies in VPP1 directly observe the request stream and may choose to keep metadata for different objects.

The second set of virtual policies, those in VPP2, are used to cooperatively act as the *master policy* that governs the physical cache space. The policies in this pool only keep metadata for those objects in the physical cache, but are allowed to order their metadata independently. During the warm-up period, the caches fill with the objects that are initially missed (*i.e.* compulsory misses) and retrieved from the server. Since all policies in VPP2 were assumed to have the same virtual cache size, the set of objects cached at this period will also be exactly the same. Whenever the physical cache is full, some objects need to be replaced from the physical cache to make room for the incoming ones. The weight distribution line shown in Figure 4.8 is *randomly sampled* to choose a policy. We replace from the physical cache by the rule of this policy. Since this method allows us to impose multiple virtual orderings on the same set of physical objects that can be quickly changed, we call this algorithm "simulated rollover" [50]. In a sense we make the physical cache look as close as possible to the physical cache of the best current virtual policy cache. Successful policies have larger weights and are thus more likely to be selected as the policy that will govern the physical cache at any given instant. The selected policy indicates which object or objects should be ejected and then all the other policies in VPP2 obey its choice releasing the record of the selected objects from their queues. This algorithm is the closest implementable algorithm to an "optimally switching" algorithm with infinite lookahead power. Initial simulation results show that its performance is much better than the other adaptive schemes. Figures 4.9 and 4.10 give the pseudo–code describing this algorithm.

4.6 Experimental Setup and Description of Workloads

We started implementing our adaptive caching algorithms as a module in the *ns* network simulator [44] to be able to easily construct complex cache topologies and to make good use of the

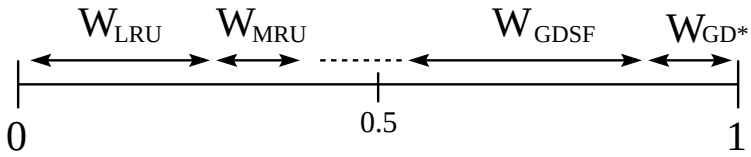


Figure 4.8: Weights of policies are attached to each other and renormalized after updates to add up to one. A random sampling of this line between (0,1) will select one of the algorithms. Good algorithms have higher weights thus probabilistically higher chance of being selected.

Definitions For Simulated Rollover Algorithm

Physical cache = *Physical memory where real data for objects is stored*
Virtual Policy (VP) = *a policy that gives an ordering of objects by using only the headers of objects*
VPP1 (Virtual Policy Pool 1) = *the set of VPs where each VP orders objects seen in the request stream*
VPP2 (Virtual Policy Pool 2) = *the set of VPs where each VP orders objects kept in the physical cache*

Figure 4.9: Definitions for the simulated rollover algorithm described below.

validated TCP/IP and Ethernet models. Although this implementation was realistic for measuring client response times the simulations quickly became a bottleneck, because of the computational needs. Even the simplest simulations were taking long periods of time. Therefore we chose to implement our own simple cache simulator in C++. We implemented 12 different cache replacement policies in our simulator including RAND, LRU, MRU, FIFO, LIFO, LFU, MFU, SIZE, GDS, GDSF, LFUDA, GD*. This long list was implemented for completeness although some of these policies are never used in modern systems. Inferior policies may be useful in mixtures. Table 2.1 summarizes most of these policies. In addition to these we implemented LFUDA [20] and Greedy-Dual* [59] policies.

We are using various workloads to test the performance of our adaptive caching algorithms. The random workload used for Figure 4.4 was generated by simple scripts. For web tests we are using both synthetic and real proxy traces. ProWGen workload is a synthetic Web proxy workload generated by the ProWGen program developed by Busari and Williamson [26, 27]. RTP trace [5] is a one day log of HTTP requests to a major proxy cache at Research Triangle Park (RTP) in the Squid national caching hierarchy by National Lab of Applied Network Research (NLANR). We used the trace of date July-05-2001 with 198,453 requests accessing to 330 MBytes of unique data and HR_∞ of 52%. DEC traces were described in Section 2.1. In our trace archive we also have other proxy (UCB, NLANR, EPA, NASA) and file system traces. For example, the Berkeley trace [86] we used in previous research [50] includes file system level calls collected from a mix of Unix and NT machines in undergraduate labs of Berkeley CS department and is also called an “instructional workload”. We are only looking at the READ calls to test the effectiveness of caching on reads. Most of the reads are small (a few hundred bytes), since for most of the time it is the file meta-data that gets read.

```
Simulated Rollover Algorithm

Add N VPs to VPP1;
foreach VP ∈ VPP1 set weight W_VP ← 1/N;
Add N VPs to VPP2;

foreach request
    id ← document requested;
    if id ∉ physical cache then
        fetch id;
        while insufficient space for id in physical cache do
            randomly choose VP using current weights;
            VP chooses object x to evict from physical cache;
            foreach VP ∈ VPP2 evict x;
        foreach VP ∈ VPP2 cache id;
    else
        foreach VP ∈ VPP2 inform VP of a hit on id;

    foreach VP ∈ VPP1
        if id ∉ VP then
            Loss_VP ← 1;
            while insufficient space for id in VP cache do
                VP evicts the least desirable object from VP cache;
            VP caches id;
        else
            Loss_VP ← 0;
            Inform VP of a hit on id;
    foreach VP ∈ VPP1
        Update W_VP using Loss_VP and formula 2;
```

Figure 4.10: Pseudo–code for the adaptive algorithm that uses two virtual policy pools and simulated rollover. A list of the abbreviations used in this figure is given in Figure 4.9.

5. Preliminary Results

In this section we present the hit rate results and comparisons of static and adaptive policies using real Web proxy and file system traces. We first review and extend the efforts in heterogeneous caching.

5.1 Static Heterogeneous

Heterogeneous caches improve total hit rates in multi–level caches without any communication between the peers. Our autonomous caches address the same problem. However, choosing good policy pairs manually can be complicated even in a simple 2–level cache topology as we will see in this section. This motivates our goal of making these decisions in an automated fashion.

Figure 5.1 shows a simple 2–level cache that can be extended to any *N* levels. If caches are of the same size and if they hold exactly the same elements then a miss in one of them will also result a miss in the other ones. This is called *inclusive caching* [102] and makes upper levels useless. This is usually the case when the same cache replacement policy is used at all levels. We would like to achieve as much *exclusive caching* [102] as possible between the collaborating caches, so that the cluster has the effect of a one big unified cache to the users. Using heterogeneous caches has been demonstrated to improve exclusivity in multi–level caches by Busari and Williamson [27] and Wong *et al.* [102]. Wong *et al.* also demonstrated the benefits of using demotions in a 2-level cache that represented client caches and a disk array cache.

Demote operation moves ejected objects one more hop away from the client instead of discarding them, thus resulting in different objects to be cached in different but topologically close caches. This creates the effect of having one large unified cache. They found LRU–MRU–Demote combination to be the most successful triple in combinations of only LRU and MRU policies. However, they recommend the demotions to be used in Local Area Networks (LAN) or Storage Area Networks (SAN) with high-speed connections, since it requires extra network resources to move objects between caches.

We extended the work of Busari and Williamson [27] and tested all permutations of 12 different policies in our expert pool in a simple 2–level cache each 4 MBytes in size as shown in Figure 5.1. We used their ProWGen workload for compatibility. ProWGen workload is a synthetic Web proxy workload generated by the ProWGen program developed by Busari and Williamson [27] and used in their previous web caching research. We used this tool to generate a workload including 200,000 requests using Zipf slope of 0.75 and Pareto tail index of 1.3 [27].

Table 5.1 shows the results for 5 of these policies. First column gives the hit rate for the first level caches. GDSF has the highest hit rate (54.41%) with the ProWGen workload described above. Note that when the same policy is used at the second level (*e.g. LRU–LRU*) the hit rates are minimal. The third column shows the policy that matched well with the policy at the first level and performed the best at the second level. Our results agree with the previous results and the best match is always a policy different than the policy in the first level. For example, with this workload using

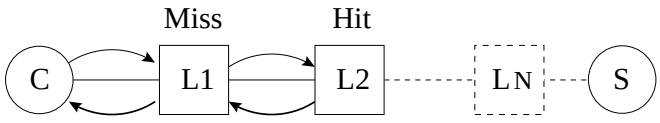


Figure 5.1: A simple N level cache. The object request of client resulted in a *hit* in the second level and was responded locally.

Policy	HR-Level1	HR-Level2 Same Policy	HR-Level2 Best Other	Best Total
LRU	42.70	0.37	7.76 (GD*)	50.46
LFU	36.79	5.25	19.36 (GDSF)	56.15
GDSF	54.41	1.72	1.75 (GDS)	56.16
LFUDA	46.75	2.49	8.47 (GD*)	55.22
GD*	52.98	0.63	2.36 (GDSF)	55.34

Table 5.1: Hit rate results in a 2–level cache using ProWGen workload with 200,000 requests. Hit Rate (HR) of policies at the first level cache is given in column 2. Column 3 lists the second level HR when the same policy is used. Column 4 shows the benefit of using a different policy at the second level by giving the results of best other policy. The best other policy is always different than the first level policy and provides considerable improvements over the usage of same policy in both levels.

GD* at the second level of a 2–level cache with LRU at the first level improves overall hit rate by 7.76%. Also note that as the hit rate of the policy in the first level approaches to maximum possible hit rates (HR∞) the hit rates at the second level drop drastically. A 12×12 matrix of all combinations and the total hit rate results in the fourth column revealed that there are many good and bad combinations and manual tuning or guessing these pairs is hard even in a simple 2–level cache. The success of pairs is also workload dependent. Therefore, we are motivated to use automated processes employing machine learning algorithms.

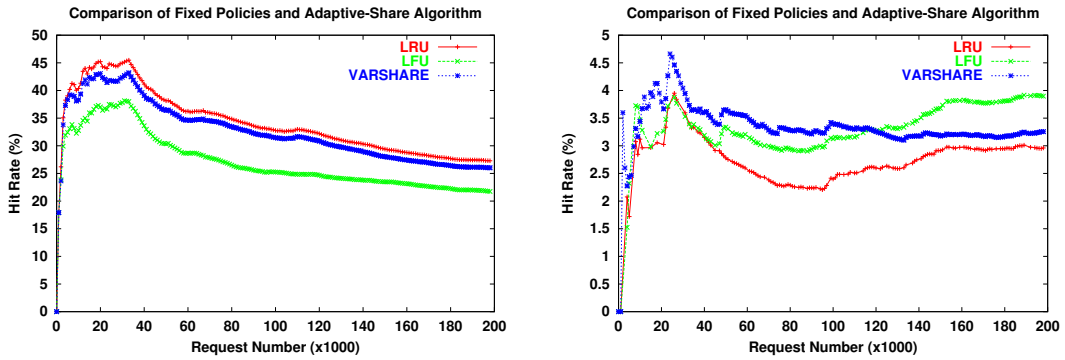
5.2 Web Proxy Trace Results

All the results presented in this section use the Simulated Rollover Algorithm with various updates such as Vovk loss update and Variable Share pools as described in Section 4. For the web tests we are using real proxy traces described in Section 4.6. Figure 5.2 shows the RTP trace hit rate results as the workload proceeds in the 2–level cache for Variable Share adaptive policy and with 2 policies (LRU & LFU) in each level's pool. Although we had no statistical information about the data the adaptive policy performed almost as good as the better policy (LRU) at the first level and averaging as good as both policies at the second level. Static policies are results of long workload analysis research and we use their expertise effectively in our design. Also note that LFU was better than LRU at all times at the second level, which relates to the benefits of using heterogeneous policies [27] in multi–level caches described in the previous section.

One drawback of Share algorithm is its additional computational overhead over the Vovk loss update. Although excluded in Figure 5.2 for clarity Vovk adaptive algorithm performed just as good as the Share algorithm without the extra overhead. This was confirmed by other workloads we tested. Therefore, we only used the simple Vovk update for the rest of the experiments in this paper.

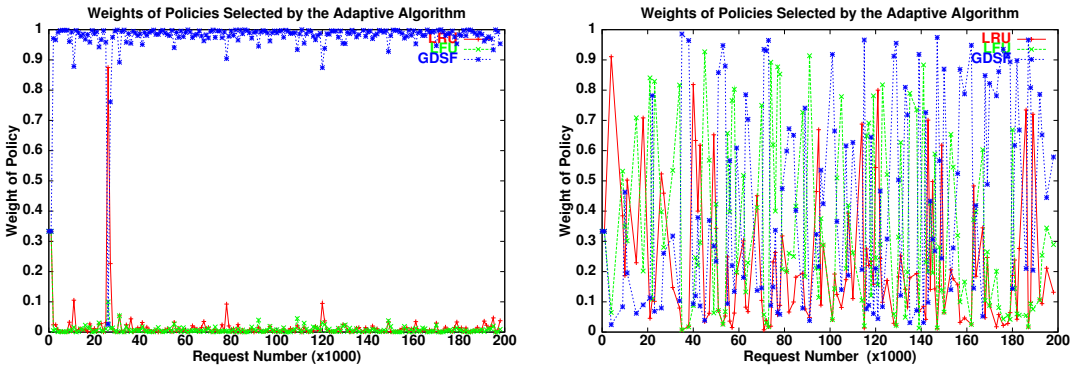
Weight adjustments in the 2–level adaptive cache that adds the GSDF policy to the previous LRU and LFU pool is given in Figure 5.3 for RTP workload. We see that while in the first the level the adaptive choice was very decisive on GDSF policy, the choice was a mixture of the workloads in second level because of the change in workload character after the first level. We could not have guessed and used any simple static or static–heterogeneous policies at this level.

In Figure 5.4 we see the hit rate results of the realcache with GDSF, LRU and LFU in the virtual policy pools using the RTP workload. We understand that the learning rate η improves the hit rate whenever there is something new to learn. So in the first level where GDSF was a clear winner learning faster or slower did not change this fact thus the performance of real cache much. In level two we see that the hit rates slowly go up and then down, but not much, as we tune η. This parameter



(a) Level 1 Hit Rates (b) Level 2 Hit Rates

Figure 5.2: In a 2–level cache with RTP workload the adaptive algorithm was able to perform almost as good as the best policy in the first level and slightly better until request number 120,000 in the second level without any statistical knowledge of the data.



(a) (b)

Figure 5.3: The machine learning algorithms provide automated response to workloads via weight adjustments of the static experts. In part (a) GDSF was a clear winner in terms of hit rates, so it was assigned a high weight for most times. In part (b) the workload did not favor any specific policy so a mixture was chosen by the adaptive scheme.

may need to be tuned for specific workloads. This is undesirable, since it violates the spirit of our fully–automated design goals.

In Figure 5.5 we see that the sharing rate α does not have an affect in the first level performance with RTP workload and only had small improvements in the second level. This conclusion says that for this workload Share algorithm is not much different than the simple Vovk update and the additional computational costs exceed its benefits.

In Figure 5.6 we see that as α increases from 0.1 to 0.5 the weights of the three policies start to converge around 0.33, a perfect 1/N sharing for 3 policies. One could also compare these two to Figure 5.3b, where the share rate was 0.001.

Figure 5.7 shows the Digital Equipment Corporation (DEC) web proxy trace [4] hit rate results. We used the trace of date Sep-16-1996 with 1,245,260 requests (524,616 unique) accessing to approximately 6 Gbytes of unique data with HR∞ of 57.87%. Adaptive policy using Vovk update managed to perform a few percents better than the best fixed policy. This is due to the capability of

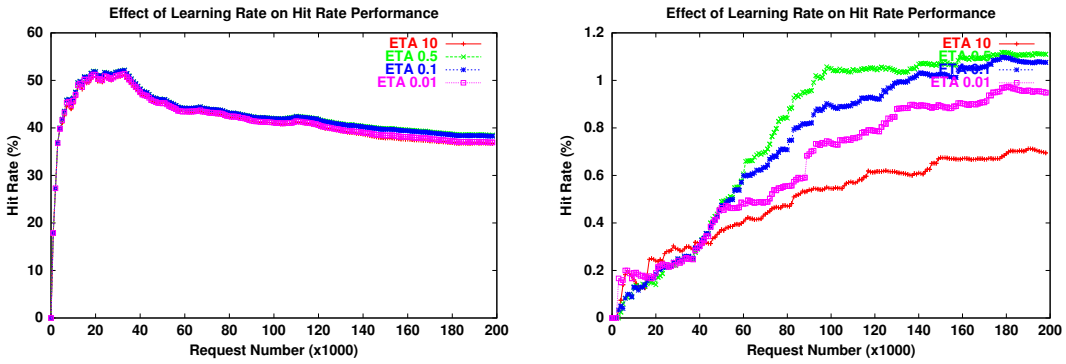


Figure 5.4: Effect of η, the learning rate, on performance. (a) Since GDSF was a clear winner in the first level cache the learning rate did not affect the results much. (b) In the second level the hit rate slightly increased and then reduced as η was increasing. This indicates that there may be benefit from adjusting η based on the arrival rate of the workload when there is not any clearly winning policy.

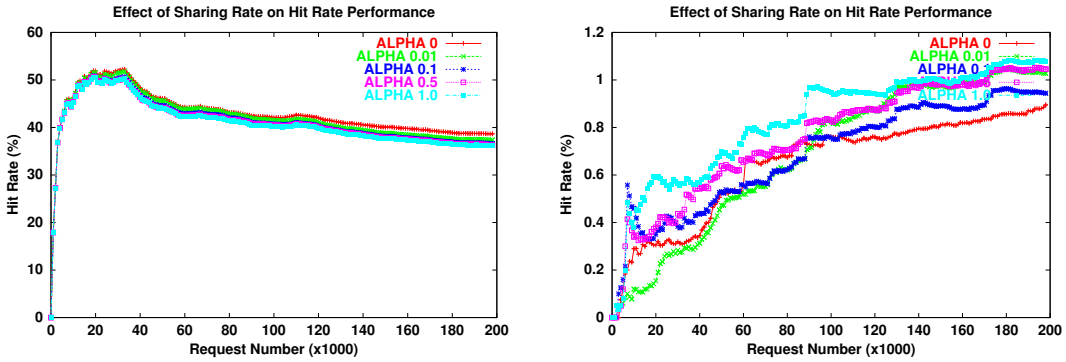


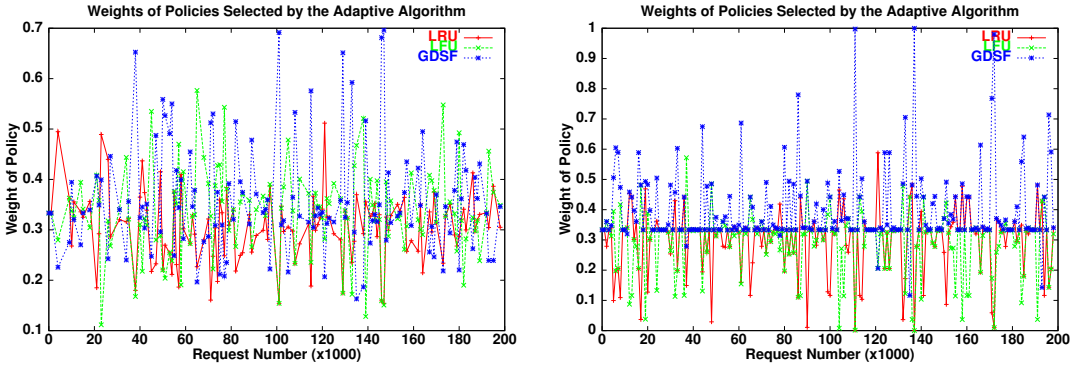
Figure 5.5: Effect of α, the sharing rate, on the hit rate. There is no clear pattern on the affect of α.

adaptive policy to switch to other policies in short time periods when another policy is favored by the workload.

Figure 5.8 shows the final hit rate results of the same DEC trace for different cache sizes. 64 Mbyte cache size accounts for 1% of the total unique data space. We show that the adaptive policy made its selections automatically and even achieved to be a little better than the best fixed policy with all cache sizes and at both levels of the 2–level cache.

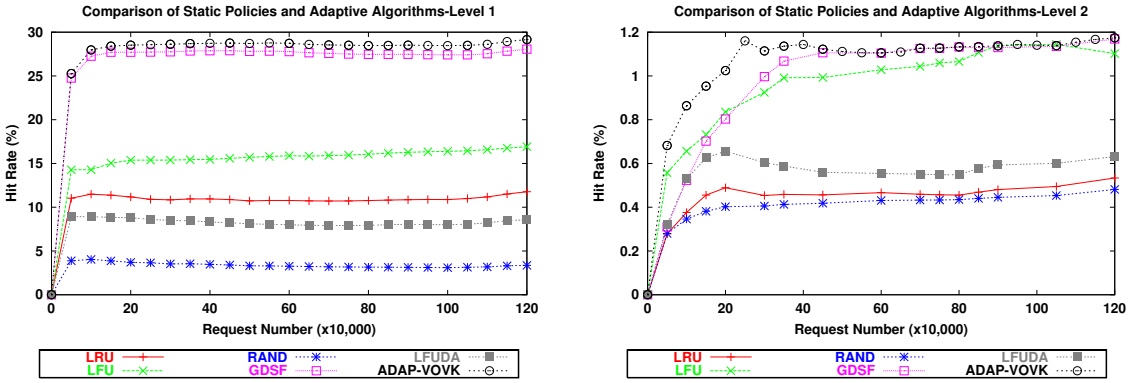
5.3 File System Trace Results

File system traces were gathered using Carnegie Mellon University's DFSTrace system [75]. The tests covered five systems for durations ranging from a single day to over a year. The traces represent varied workloads, particularly *Mozart* a personal workstation, *Ives*, a system with the largest number of users, *Dvorak* a system with the largest proportion of write activity, and *Barber* a server with the highest number of system calls per second. We will only present the results for



(a) (b)

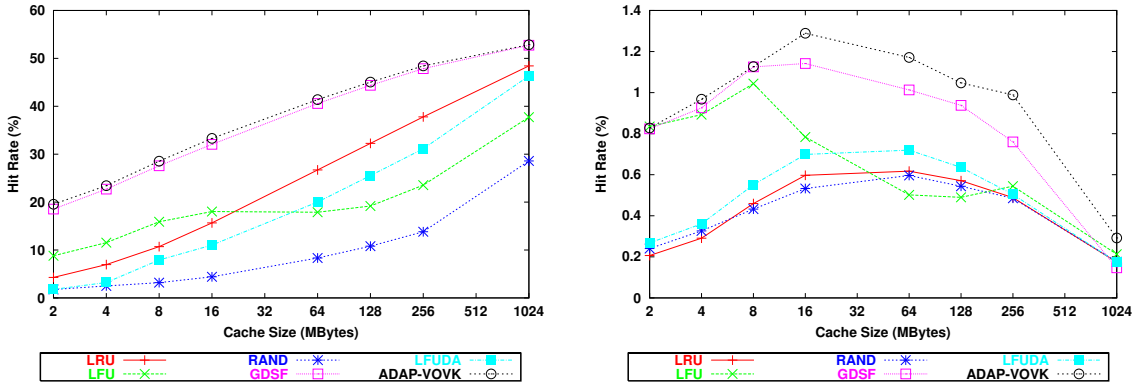
Figure 5.6: As share rate increases from 0.001 in (a) to 0.5 in (b) weights of policies converge to a perfect 1/N parameter, N being the number of policies in the pool. In this case we have 3 policies in the pool, thus the weights start to converge to 0.33.



(a) Level 1 Hit Rates (b) Level 2 Hit Rates

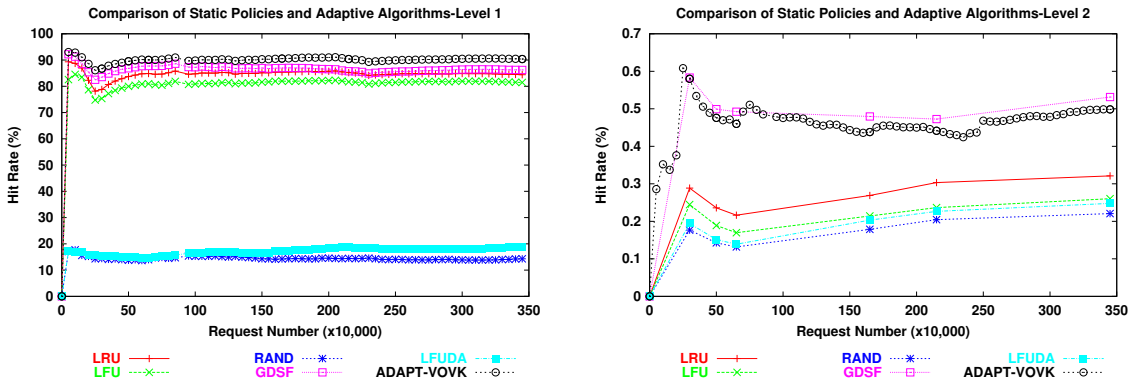
Figure 5.7: Hit rate comparisons of the static policies and the adaptive policy for DEC-9/16/96 trace using 8 MBytes of memory at each level of the 2–level cache. Adaptive policy was even slightly better than the best static policy.

the year long *Ives* trace, which includes 3,464,314 requests. The results for others looked very similar. These traces provide information at the system-call level, and represent the original stream of access events not filtered through any intervening caches. For these CMU traces we are measuring file accesses based on file open requests. This assumes a data-object granularity for the analysis, we focus on patterns of file requests and are not concerned with intra-file access patterns.



(a) Level 1 Hit Rates (b) Level 2 Hit Rates

Figure 5.8: Hit rate comparisons of the static policies and the adaptive policy for DEC-9/16/96 trace using different cache sizes. Adaptive policy tracks and even beats the performance of the best fixed expert.



(a) Level 1 Hit Rates (b) Level 2 Hit Rates

Figure 5.9: Hit rate comparisons of the static policies and the adaptive policy for Ives file system trace using 4 MBytes of memory at each level of 2–level cache.

6. Proposed Work

Our current designs and implementations constitute only proofs of concepts. We will extend tour current research in two major areas. The first area includes comparison of Storage Embedded Networks (SEN) using heterogeneous and adaptive caching with hierarchical caches. For this part we will run large scale trace–driven simulations with realistic network topologies. We will compare the average client response times, network bandwidth usage, server load reductions , scalability and load distribution of these two systems. The second area is to implement adaptive caching in the memory management modules of real file system and web proxy caches. Real implementations will enforce us to minimize space and computational overheads and make performance trade–offs.

6.1 Storage Embedded Networks versus Hierarchical and Distributed Caching

We have discussed SEN and other caching architectures in the previous sections. To summarize, hierarchical proxies have performance bottlenecks at the upper layers because of the exponentially increasing number of users they serve. Distributed caches also have a hierarchical structure, but the data is only kept at the edges by the clients. Upper layers just handle the control messages carrying object advertisements and object locations. The data transfers occur between clients. Although client to client transmissions take shorter time the connection times and administrative costs are higher. Adaptive web caching proposes that caches form multicast groups to exchange the object location information. SENs do not establish hierarchies and do not have the connection establishment and communication costs.

To compare the advantages and disadvantages of using SEN to other architectures for distributed data access we will run large scale simulations using realistic network topologies such as UCSC network topology in Figure 6.1a and national caching hierarchies in Figure 6.1b. We will parse real web proxy and file system traces to obtain request streams per client and play requests coming from hundreds to thousands of clients at the edges of the network. We will measure user response times, network bandwidth usage and server load of both architectures.

We will also create the *flash–crowd* scenarios, which occur frequently on the Internet. Flash–crowd is an unexpected flood of requests experienced by a web site. In our simulations, we will generate the same situation by sending a flood of requests to a single server. We will measure and compare the load distribution capabilities of SEN and the hierarchies.

The raw disk workload we are planning to test over SEN is of special interest, because of its benefits for emerging network–attached storage systems [47] and the Internet SCSI (ISCSI) [91] transport. We will use SEN and ACME for reliable transport of ISCSI packets carrying raw disk data and measure the throughput of data transfers.

ISCSI [91] is a transport protocol for SCSI block command set [11] that operates on top of TCP/IP. ISCSI maps or encapsulates the SCSI commands into TCP/IP packets and then carries them over to the remote network–attached targets. SCSI protocol was designed for short distance and reliable buses. Now there is a question on how ISCSI will perform over Wide Area Networks (WAN) with high Round Trip Times (RTT) and high jitter, *i.e.* variance in latency.

In SENs RTT of the lost or corrupted packets will be reduced to the RTT of few hops because of the persistence on the path. Like many applications network–attached storage devices and ISCSI require reliable transmission over WAN, but do not want to suffer the performance overhead of TCP. However, because of the amount and content of the data they transfer they are less flexible for losses and performance degradations. SEN can reliably carry raw SCSI data over IP with no Forward Error

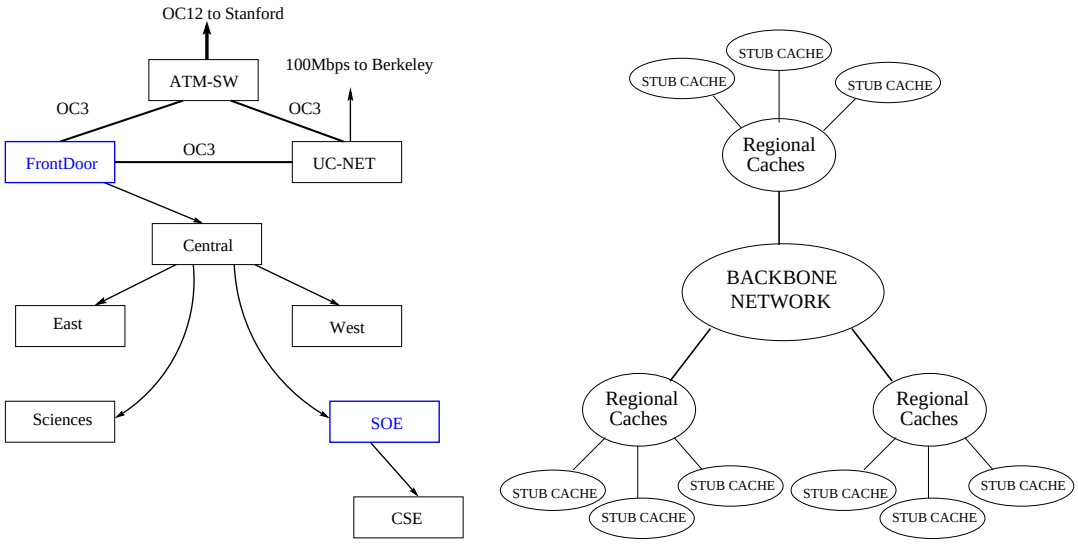


Figure 6.1: Various network topologies to be tested for user perceived performance and scalability analysis. (a) UCSC network topology comparison with seven SEN routers versus two proxies in FrontDoor and School of Engineering (SOE). (b) Another topology: national hierarchical caches simplified from a prior work of Danzig *et al.* [41].

Correction (FEC). A scheme similar to FEC by DigitalFountain Inc. [42] demonstrates an order of magnitude improvement of bulk data transfer throughput over WAN.

6.2 Implementation of Adaptive Caching in Linux

Our preliminary results indicate that adaptive caching can be implemented using multiple techniques and even the simplest weight update mechanisms provide enough adaptivity to the system. However, the benefits gained from more complex algorithms are not worth their cost in some cases. The time complexity of our current implementation is $N.O(n)$, where N denotes the number of policies in the pool. The complexities of all virtual policies were assumed to be same and $O(n)$, since they all use linked lists of objects. Intuitively, this much increase in cost would also require a factor of N improvements in the user response times to be feasible. However, CPU cycles are less expensive than the user's time and are getting cheaper. Even a factor of two improvement in user response times would be a significant achievement. There is also a similar space overhead of our techniques, but this issue is less problematic because of the small size of object headers. To understand all the detailed design issues we will implement machine learning based adaptivity in memory management module of Linux or FreeBSD operating systems. This will give us more insight about the benefits of adaptivity in a single host.

To improve the computational overhead some inferior policies may be switched off after a while. However, this would take us back to using a few static policies. The only difference from static policies is that the mixture obtained would be well tuned for the initially observed workload. Indeed, for a workload that is guaranteed to stay the same this is the best approach. However, the workloads we see in SEN will be more dynamic than a simple steady request stream. Therefore, we will have to continuously and automatically observe the workload, summarize the information we receive, periodically update the weights and reflect the changes to the master policy. If our periods are too

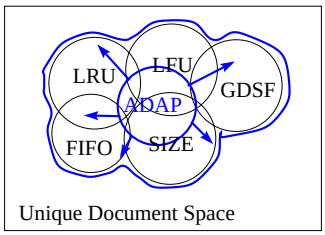


Figure 6.2: It is possible to parse request streams based on their algorithmic content and then use this information to recreate request streams of the same nature, but of adjustable length. This approach may lead to close to realistic synthetic trace generation that is especially useful for adaptivity benchmarks.

long then the adaptive policy sticks to the policy with the highest maximum hit rate in that period of the request stream. Switching between policies becomes harder. If the periods are too small, (*e.g.* per request updates), then the updates are too erratic and the adaptive algorithm cannot decide which policy to chose.

Real implementations will also require the use of more efficient data structures than simple linked lists. We may resort to using the B–tree [37] and the B+-tree [83] structures used in file systems [16] and databases. A Unified Buffer Management (UBM) scheme for FreeBSD was implemented and tested by Kim *et al.* [63] and user response times were improved by 67.2% (with an average of 28.7%).

The selection of the metric that leads improved user response times is also crucial. Until this point our focus was on defining a basic design whose success could be evaluated. We started with a discrete hit (1) and miss(0) criteria and the *objective function* was to "always get hits". The difference between the real and objective functions was called the loss and was used as a feedback to adjust the system. However, not all hits and misses are equal. We are already investigating different functions of the *object size* for improved byte hit rates and observing the effects of different functions on user response times.

There is ongoing research in our machine learning group for obtaining *hybrid policies* via complex mixing or "meta–mixing" methods. The most significant outcome of this research would be if hybrids only using simple policies such as LRU, LFU could outperform more complex policies such as GDSF. We will collaborate with them to understand the nature and limits of hybrid policies.

6.3 Workload Characterization and Benchmarking

Another goal is to use the results of ACME policy mixing to understand the nature of various workloads and to reconstruct similar synthetic workloads of adjustable length for trace–driven simulations. We may be able to use the weight update information to run a backwards induction and through usage of "stack distances" [7, 59] create sequences of object requests that would generate similar hit rate results. These types of workloads will especially be useful for benchmarking the adaptivity of new algorithms, since the characteristics could be changed in a controlled fashion.

Figure 6.2 shows the unique document space covered by the static policies using the same cache size. The documents that will be selected by multiple policies will fall in the intersection regions of the sets. For example, a document that is getting requests very frequently will both be selected by LRU and LFU algorithms due to its recency and frequency of access, respectively. The adaptive policy in this figure can be thought of as a circle of the same size, but that can move around following the request stream in the unique document space.

7. Conclusions

We presented Storage Embedded Networks (SEN) architecture and adaptive caching schemes applicable to single and multiple processor systems. SEN provides improvements over hierarchical and distributed caches and adaptive caching helps with the management of SEN caches when complex dynamic workloads are serviced. Our autonomous caches use machine learning algorithms to collaborate with a pool of caching experts to tune themselves to the observed workload. Since no cache databases or synchronization messages are exchanged the clusters composed of these autonomous cache nodes will be scalable and manageable. Our methods will be useful for all distributed Web, file system, database and content delivery services.

We proposed to extend our work in two major areas, first the comparison of SEN with existing distributed caching and second the real system implementation and improvement of adaptive algorithms. Both contributions of this paper are novel techniques. We will explore the possible interactions between our techniques and other emerging networked–storage systems such as NASD and ISCSI. This research may also lead to interesting workload characterization and adaptivity benchmarking tools.

Some researchers and businesses predict that all caching systems will be useless due to the immense customization of web content by both the clients and the content providers. However, we believe that the dynamic part of the content constitutes mostly the text portion of the documents composed of many images and text. The bulk of the data that is transfered is still in static images and static text. In their extensive Web proxy workload characterization in 1999 spanning 5 months and 117 million requests Arlitt et al. [20] reported that 92% of all the requests accounting for 96% of the data transferred was cacheable and high hit rates were achieved by proxies. Surveys of WWW [98] from 1997 to 1999 showed that the size of the static content on the web has grown exponentially (approximately 15% per month). It is also known that web workloads follow a Zipf popularity distribution [39, 7, 21, 27], which also indicates that there will still be sharing in future and we will continue to benefit from caching. There are also proposed solutions for caching the dynamic content [31].

7.1 Summary of Benefits

Adaptive and autonomous caching improves:

- *Performance*: SEN and ACME together provide the best possible fixed or mixed current strategy anytime, anywhere thus giving the users fastest possible response times.
- *Adaptivity*: our systems govern themselves based on their location in the network topology and the observed workload. Our biggest contribution will be the elimination of tedious manual tuning from modern storage systems. *Manageability* and *Flexibility* are also improved at the same time.
- *Scalability*: this is the key to success in a global scenario. Our proposed methods do not have any communication overheads, therefore provide scalable growth on the Internet. The backward compatibility vision in SEN routers allow its deployment to be *evolutionary*.
- *Efficiency*: a new system should not incur overheads and costs that exceed its benefits. We believe the simple, but powerful machine learning algorithms will provide a strong base for efficient adaptive caching design.

Acknowledgements

Ismail Ari was supported by a grant from Hewlett-Packard Laboratories. We thank Dick Henze, Terril Hurst, Rich Elder, Ran-Fun Chiu and Scott Marovich in the Storage Technologies Department of Hewlett-Packard Laboratories for supporting our research and providing helpful comments. We are also grateful to Ethan Miller, Darrell Long, Scott Brandt, Patrick Mantey and other members of the Storage Systems Research Center, Manfred Warmuth and the members of the Machine Learning Group; Robert Gramacy, Jonathan Panttaja and Clayton Bjorland in our Game Theory Group at University of California, Santa Cruz for helping us refine the ideas in this paper and in our on-going work.

References

[1] Akamai, http://www.akamai.com.

[2] Digital Island, http://www.digitalisland.com.

[3] Matrix.net. The state of the internet, http://average.miq.net.

[4] Digital Equipment Corporation, Digital's Web Proxy Traces. ftp://ftp.digital.com/pub/DEC/traces, August–September 1996.

[5] National Lab of Applied Network Research (NLANR). Sanitized access log. Available at ftp://ircache.nlanr.net/Traces/, July 1997.

[6] K. Akala, E. Miller, and J. Hollingsworth. Using content-derived names for package management in Tcl. In *Proceedings of the 6th Annual Tcl/Tk Conference*, pages 171–179, San Diego, CA, Sept. 1998. USENIX.

[7] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems (PDIS '96)*, Dec. 1996.

[8] A. Amer and D. D. E. Long. Adverse filtering effects and the resilience of aggregating caches. In *Proceedings of the Workshop on Caching, Coherence and Consistency (WC3 '01)*, Sorrento, Italy, June 2001. ACM.

[9] A. Amer and D. D. E. Long. Aggregating caches: A mechanism for implicit file prefetching. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, pages 293–301, Cincinnati, OH, Aug. 2001. IEEE.

[10] A. Amer and D. D. E. Long. Noah: Low-cost file access prediction through pairs. In *Proceedings of the 20th IEEE International Performance, Computing and Communications Conference (IPCCC '01)*, pages 27–33. IEEE, Apr. 2001.

[11] American National Standard for Information systems(ANSI). Small Computer System Interface SCSI-2. Standard X3.131-1994, Jan. 1994.

[12] E. Amir, H. Balakrishnan, S. Seshan, and R. H. Katz. Efficient TCP over networks with wireless links. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[13] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2000.

[14] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[15] E. Anderson, R. Swaminathan, A. Veitch, G. A. Alvarez, and J. Wilkes. Selecting RAID levels for disk arrays. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[16] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb. 1996.

[17] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, Feb. 1995.

[18] Anonymous. Secure hash standard. FIPS 180-1, National Institute of Standards and Technology, Apr. 1995.

[19] I. Ari, A. Amer, E. Miller, S. Brandt, and D. Long. Who is more adaptive? ACME: adaptive caching using multiple experts. In *Workshop on Distributed Data and Structures (WDAS 2002)*, Paris, France, Mar. 2002.

[20] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. In *Proceedings of the 2nd Workshop on Internet Server Performance (WISP '99)*, Atlanta, Georgia, May 1999.

[21] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. In *Measurement and Modeling of Computer Systems*, pages 126–137, 1996.

[22] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Oct. 1991.

[23] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS '95)*, May 1995.

[24] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison–Wesley, 2nd edition, 1998.

[25] R. Burns, R. Rees, and D. D. E. Long. Consistency and locking for distributing updates to web servers using a file system. In *Workshop on Performance and Architecture of Web Servers*. ACM, June 2000.

[26] M. Busari and C. Williamson. On the sensitivity of web proxy cache performance to workload characteristics. In *Proceedings of IEEE INFOCOM'2001*, pages 1225–1234, Anchorage, Alaska, Apr. 2001.

[27] M. Busari and C. Williamson. Simulation evaluation of a heterogeneous web proxy caching hierarchy. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, pages 379–388, Cincinnati, OH, Aug. 2001. IEEE.

[28] L.-F. Cabrera and D. D. E. Long. Swift: A distributed storage architecture for large objects. In *Digest of Papers, 11th IEEE Symposium on Mass Storage Systems*, pages 123–128, Monterey, Oct. 1991. IEEE.

[29] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 193–206, Dec. 1997.

[30] P. Cao and C. Liu. Maintaining strong cache consistency in the World Wide Web. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, 1997.

[31] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of the 1998 Middleware Conference*, 1998.

[32] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 102–115, 1997.

[33] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, 1996.

[34] K. Cheng and Y. Kambayashi. Advanced replacement policies for WWW caching. In *Proceedings of Web-Age Information Management, First International Conference, WAIM*, pages 21–23, Shanghai, China, June 2000.

[35] J. Chuang and M. Sirbu. Stor-serv: Adding quality-of-service to network storage. In *Proceedings of Workshop on Internet Service Quality Economics*, Cambridge MA, Dec. 1999.

[36] M. Cieslak, D. Forster, G. Tiwana, and R. Wilson. Web Cache Coordination Protocol (WCCP) V2.0, Internet-Draft , draft-wilson-wrec-wccp-v2-00.txt, July 2000.

[37] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[38] P. F. Corbett and D. G. Feitelso. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[39] M. E. Crovella and A. Bestavros. Self–similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[40] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Nov. 1994.

[41] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of SIGCOMM93*, pages 239–248, 1993.

[42] Digital Fountain Inc. White Paper, Comparison of Digital Fountain File Transfers to FTP, http://www.digitalfountain.com, 2001.

[43] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, 2000.

[44] K. Fall. Network emulation in the Vint/NS simulator. In *Proceedings of IEEE Symposium on Computers and Communications (ISCC'99)*, July 1999.

[45] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[46] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 78–91, Oct. 1997.

[47] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[48] T. J. Gibson and E. L. Miller. Long-term file activity patterns in a UNIX workstation environment. In *Proceedings of the 6th Goddard Conference on Mass Storage Systems and Technologies / 15th IEEE Symposium on Mass Storage Systems*, pages 355–372, College Park, MD, Mar. 1998.

[49] M. Goulde. Network Caching Guide: Optimizing Web Content Delivery, Inktomi Corporation Whitepaper, http://www.inktomi.com, Mar. 1999.

[50] R. Gramacy, I. Ari, J. Panttaja, and C. Bjorland. Adaptive caching by mixing. CMPS 272 Evolutionary Game Theory Class Project Report, University of California Santa Cruz, Mar. 2002.

[51] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Modeling and performance of MEMS-based storage devices. In *Proceedings of the 2000 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 56–65, June 2000.

[52] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 197–207, 1994.

[53] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[54] D. P. Helmbold, D. D. E. Long, T. L. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *ACM/Baltzer Mobile Networks and Applications (MONET)*, 5(4):285–297, 2000.

[55] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking 1996 (MOBICOM '96)*, pages 130–142, Rye, New York, Nov. 1996. ACM.

[56] M. Herbster and M. K. Warmuth. Tracking the best expert. In *Proceedings of the 12th International Conference on Machine Learning*, pages 286–294, Tahoe City, CA, 1995. Morgan Kaufmann.

[57] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. Wes. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[58] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. *ACM Transactions on Computer Systems*, 45(10):1180–1194, 1996.

[59] S. Jin and A. Bestavros. GreedyDual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[60] E. N. Johnson. A protocol for network level caching, (C-TCP). M.E.E.C.S Thesis, MIT, May 1998.

[61] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek. The measured performance of content distribution networks. *Computer Communications*, 24(2):202–206, 2001.

[62] H. Khalid and M. S. Obaidat. Kora–2: a new cache replacement policy and its performance. In *Proceedings of 6th IEEE International Conference Electronics, Circuits and Systems (ICECS)*, volume 1, pages 265–269, 1999.

[63] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 119–134, San Diego, CA, Oct. 2000.

[64] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, 1997.

[65] T. M. Kroeger and D. D. E. Long. Predicting file-system actions from prior events. In *Proceedings of the Winter 1996 USENIX Technical Conference*, pages 319–328, San Diego, Jan. 1996.

[66] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 13–22, Monterey, 1997. USENIX.

[67] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[68] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Cambridge, MA, 1996.

[69] J. Liebowitz. *The Handbook of Applied Expert Systems*. CRC Press LLC, 1st edition, 1998.

[70] N. Littlestone and M. K. Warmuth. The Weighted Majority Algorithm. *Information and Computation*, 108(2):212–261, Feb. 1994.

[71] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web caching: towards a new global caching architecture. *Computer Networks and ISDN Systems*, 30(22-23):2169–2177, 1998.

[72] E. L. Miller, S. A. Brandt, and D. D. E. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 83–87, Schloss Elmau, Germany, May 2001.

[73] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, 1997.

[74] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of SIGCOMM97*, pages 181–194, 1997.

[75] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. *Software—Practice and Experience (SPE)*, 26(6):705–736, June 1996.

[76] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems. Technical Report 91-3, University of Michigan Center for IT Integration (CITI), Aug. 1991.

[77] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[78] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.

[79] M. S. Obaidat and H. Khalid. Estimating neural networks-based algorithm for adaptive cache replacement. *IEEE Transactions on Systems, Man, and Cybernetics*, 28-B(4), Aug. 1998.

[80] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[81] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In D. D. E. Long, editor, *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[82] K. K. Ramakrishnan and S. Floyd. A proposal to add explicit congestion notification (ECN) to IP. Request For Comments (RFC) 2481, IETF, Jan. 1999.

[83] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, Dallas, TX, May 2000.

[84] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.

[85] P. Rodriguez, C. Spanner, and E. Biersack. Web caching architectures: hierarchical and distributed caching. In *Proceedings of the 4th International WWW Caching Workshop*, 1999.

[86] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, June 2000.

[87] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[88] A. Rubini. *Linux Device Drivers*. O'Reilly and Associates, 1st edition, Feb. 1998.

[89] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 405–420, San Diego, CA, Jan. 1993.

[90] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, Mar. 1994.

[91] J. Satriani. ISCSI. Internet-Draft draft-ietf-ips-iscsi-09.txt, IETF, Nov. 2001. http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-09.txt.

[92] R. E. Schapire. A brief introduction to boosting. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1401–1406, 1999.

[93] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real time applications. Request For Comments (RFC) 1889, IETF, Jan. 1996.

[94] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the Internet. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99)*, pages 273–284, 1999.

[95] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, 1997.

[96] V. Vellanki and A. Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proceedings of Supercomputing '99*, Portland, OR, Nov. 1999. ACM Press and IEEE Computer Society Press.

[97] V. Vovk. Aggregating strategies. In *Proceedings of the 3rd Annual Workshop on Computational Learning Theory*, pages 371–383, Rochester, NY, 1990. Morgan Kaufmann.

[98] J. Wang. A survey of web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5):36–46, Oct. 1999.

[99] G. Weikum, A. C. Konig, A. Kraiss, and M. Sinnwel. Towards self-tuning memory management for data server. *Data Engineering Bulletin*, 22(2):3–11, 1999.

[100] D. J. Wetherall and D. L. Tennenhouse. The ACTIVE IP option. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[101] J. Wilkes, W. Rickard, G. Gibson, D. Anderson, and D. Black. Shared storage model. a framework for describing storage architectures. Technical Council Proposal Document draft-june5, SNIA, June 2001.

[102] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, Monterey, CA, June 2002. USENIX.

[103] R. Wooster and M. Abram. Proxy caching that estimates page load delays. In *Proceedings of the 6th International World Wide Web Conference*, pages 325–334, Santa Clara, CA, Apr. 1997.

[104] T. Yeh, D. D. E. Long, and S. Brandt. Performing file prediction with a program-based successor model. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, pages 193–202, Cincinnati, OH, Aug. 2001. IEEE.

[105] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. In *Proceedings of SIGCOMM99*, pages 163–174, 1999.