ACME: Adaptive Caching Using Multiple Experts[*]

ISMAIL ARI
University of California, Santa Cruz

AHMED AMER
University of California, Santa Cruz

ROBERT GRAMACY
University of California, Santa Cruz

ETHAN L. MILLER
University of California, Santa Cruz

SCOTT A. BRANDT
University of California, Santa Cruz

DARRELL D. E. LONG
University of California, Santa Cruz

Abstract

The gap between CPU speeds and the speed of the technologies providing the data is increasing. As a result, latency and bandwidth to needed data is limited by the performance of the storage devices and the networks that connect them to the CPU. Distributed caching techniques are often used to reduce the penalties associated with such caching; however, such techniques need further development to be truly integrated into the network. This paper describes the preliminary design of an adaptive caching scheme using multiple experts, called ACME. ACME is used to manage the replacement policies within distributed caches to further improve the hit rates over static caching techniques. We propose the use of machine learning algorithms to rate and select the current best policies or mixtures of policies via weight updates based on their recent success, allowing each adaptive cache node to 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 scalable and manageable. We show that static techniques are suboptimal when combined in networks of caches, providing potential for adaptivity to improve performance.

[*]This work has been supported in part by Hewlett-Packard Laboratories Storage Technologies Department, the National Science Foundation under award CCR-9972212, and the USENIX Association.

Keywords

caching, adaptive systems, clusters, adaptive caching, static caching

1 Introduction

The number of users connected to the Internet is growing exponentially. Satisfying so many users with fast response times 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 [25, 26]. In remote data accesses, network latency is added to the I/O latency at the servers [1] further reducing the performance of the applications.

Enormous research efforts have been put into characterizing Web [1] and file system [25] 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 systems [19, 5, 24]. Unfortunately, these policies cannot adapt to changes in workload and network topology and become suboptimal when the conditions become more complex than the characterized cases [30].

Many factors increase the complexity of today's systems in which caching is used. First, the characteristics of the 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" [2]. Recent research shows that these filtering effects in a hierarchy of caches can change the nature of an otherwise predictable workload such that the higher layers are effectively useless [32, 7]. In these complex scenarios analytical modeling is daunting, manual tuning is tedious [3] and making wrong decisions has extreme monetary and performance costs.

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 multimedia (audio and video) as well as text. The bulk of the data that is transferred is still in static text, images, audio and video. In their extensive Web proxy workload characterization in 1999 spanning 5 months and 117 million requests Arlitt et al. [5] 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 [29] 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 [1, 7], which also indicates that there will still be sharing in the future and we will continue to benefit from caching. There are also proposed solutions for caching the dynamic content [10].

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 treat existing cache replacement algorithms as 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 propose 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 *hit rate*, the fraction of requested items found in the cache, or *byte hit rate*, the percentage of requested bytes that are found in the cache. Each adaptive node is a self-governing, or "autonomous," entity. Neither cache content information nor synchronization messages are exchanged between the peer caches; thus, the clusters composed of these autonomous cache nodes will be scalable and manageable. In this way, we can use machine learning algorithms [17] to improve caching just as they have been used in addressing non-trivial operating systems problems such as the disk spin-down problem in mobile computers [16].

2 Related Work

Caching is used on virtually all data access paths [31] and at all abstraction levels (file/record, block) in modern storage architectures. 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.

2.1 Existing Cache Replacement Algorithms

Table 1 lists some very popular and some 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 or popularity of access. Most Recently Used (MRU) and Most Frequently Used (MFU) are not successful when used alone, but may be beneficial in mixtures of policies. SIZE replaces the largest object and Greedy-Dual-Size (GDS) [19, 8] 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

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 1: An extended taxonomy of some existing and proposed cache replacement policies. Descriptions of the policies are in Section 2.1.

a running age factor. L is set to the key value of the objects that are replaced from the cache. GDS with Frequency (GDSF) [5] 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 $K_i = (C_i \times F_i) + L$ [5]. Lowest Relative Value (LRV) [24] makes a cost–benefit analysis using the access time, access frequency and size information about objects.

Hashing or more complex Bloom filters [15] 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 a peer node should be caching that object then it may 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 source of an object, while down-stream hop counts indicate logical distance from clients. Recent research [33] points to the benefits of keeping a record of access latency history per object, providing yet another potential caching criterion (*e.g.*, it is wise to keep items items in the cache if they are very costly to retrieve). Stor-serv [12] proposes Quality of Service (QoS) ideas used in networking to be applied to storage systems for giving differentiated services to users.

Table 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 extremely varied and subject to change, requiring 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. However, a single function cannot be successful at all times with different workloads and their mixtures. Other taxonomies of time, frequency and size based policies are presented in prior work [19].

2.2 Adaptivity in Systems

The term “adaptivity” has different meanings in different systems. For example Linux has “dynamic” cache space management [6] 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. However, the cache replacement policies are static. The buffer cache, the inode cache and the name (or directory) cache are managed by LRU algorithm.

Hybrid Adaptive Caching (HAC) [11] 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 to free memory pages, thus reducing the high bookkeeping overhead of object caching. HAC was shown to outperform object caching.

The file caches of the Sprite distributed file system [22] change dynamically in response to the changes in virtual memory requirements. The Andrew File System (AFS) [18] has two separate caches for status and data and both are governed by the Least Recently Used (LRU) algorithm. In the Serverless File System (xFS) [4] any machine can store, cache or control any block of data. Adaptive web caching [21] 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. These systems do not mention about adaptively changing their caching policy to track the changes in workloads.

In this paper, we confine our design to the use of adaptive replacement policies for objects with static content. Detailed research on consistency issues in file systems [4, 18, 28] and web caching can be found in related previous work [9, 34]. Many of these efforts conclude that write-sharing is rare enough that it is reasonable to pick the simplest consistency mechanism.

2.3 Static Heterogeneous Caching

Figure 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* [32] and makes upper levels useless. This situation often occurs when the same cache replacement policy is used at all levels. We would like to achieve as much *exclusive caching* [32] as possible between the collaborating caches, so that the cluster has the effect of a one big unified cache to the users. Using heterogeneous policies has been demonstrated to improve exclusivity in multi-level caches by Busari and Williamson [7] and Wong *et al.* [32]. Our analysis in Section 4 confirms and extends these results. It is crucial to note that the unified cache effect is achieved without any communication between the peers. However, choosing good

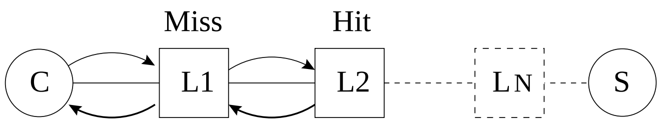


Figure 1: A simple N level cache. The object request of client resulted in a *hit* in the second level and was satisfied there.

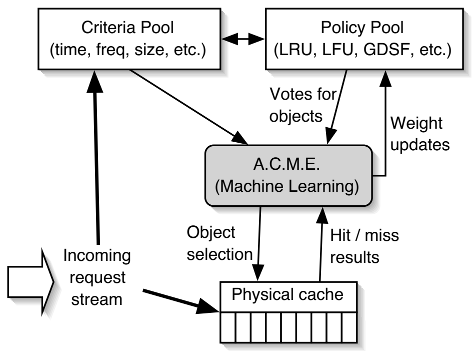
policy pairs manually can be complicated even in a simple 2-level cache topology. This motivates our goal of making these decisions in an automated fashion.

Wong *et al.* [32] also demonstrated the benefits of using demotions in a 2-level cache that represented client caches and a disk array cache. A demote operation moves ejected objects one hop further from the client instead of discarding them, thus resulting in different objects to be cached in different but topologically close caches. 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.

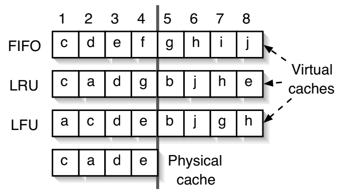
3 Design of an Adaptive Caching Scheme

Adaptivity to a variety of and possibly changing conditions requires multiple algorithms to be embedded in one system. This is also true for an adaptive caching system. Therefore, our design uses a *pool* of static cache replacement algorithms with different characteristics to decide how to behave based on the observed workload. The challenge is to join the relatively weak predictions of many different policies into one highly-accurate prediction [27], deciding which objects to keep in the cache. Expert systems, specifically machine learning algorithms [17] have been successfully used for this purpose in the past to solve non-trivial operating systems problems [16].

Figure 2 illustrates the major components of our initial weighted voting–based adaptive design. We define a pool of *virtual caches*, each of which simulates a single static cache replacement policy by maintaining an object ordering as if it owned the entire physical cache. To save space, each virtual cache only keeps object header information, not the actual data for the object. On each request, each virtual cache reports whether it would have gotten a hit (scored as 1) or miss (scored as 0) if it were the real cache. This information is used to adjust the weights of the policies by increasing the weight of policies that would have kept the object and decreasing the weight of those policies that would have discarded the object. Future implementations may use more sophisticated mechanisms to “reward” and “punish” policies; for example, they might consider how highly an



(a) Design of Adaptive Caching using Multiple Experts (ACME).



(b) Virtual caches and physical caches.

Figure 2: Virtual caches in the policy pool assign values to each object they "cache." 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.

object was valued in calculating the weight change rather than simply using binary value.

Both caching and replacement are done based on votes. Each virtual cache votes on the objects it wants to keep, assigning higher values to objects that it believes are most worth keeping. The objects with the highest weighted vote total stay in the cache. Over time, the real cache ordering will probably resemble the ordering of virtual caches with the highest weights, but will still be a mixture of multiple policies.

One potential limitation in this design is the limitation that virtual caches only keep as many objects as will fit in the physical cache. If this is the case, a virtual cache with space for n objects will be penalized equally for not containing the objects ranked $n+1$ and $n+100$, where a rank of 1 is assigned to the most "valuable" object. We believe it is better to reward caches that rank reused objects highly even if the objects could not be kept in the cache. Thus, we use virtual caches that are larger than the physical cache, as shown in Figure 2b. Using this strategy, an object X_1 ranked $n+1$ in all virtual caches might be chosen over an object X_2 ranked $n-1$ in one cache and unranked in every other cache. If virtual caches were the same size as physical caches, X_1 would be totally unknown and thus ineligible for ranking. However, it is likely that X_1 is more desirable than X_2 and would receive a higher vote because so many policies rank it relatively highly.

We expect that keeping track of more objects than the physical cache has space for will not present an overly large burden on an ACME cache. Objects can be tracked with relatively few bytes, and we believe the improved performance will justify the little incremental space required.

4 Preliminary Analysis and Results

In this section we present performance results and comparisons of static policies and a simple adaptive policy using real Web proxy and file system traces. We simulated the performance of two-level hierarchies using all combinations of policies at each of the two nodes, showing that heterogeneous policies outperform homogeneous policies. We next found that the loss due to using a static policy on even a single node could approach 20% or more of the potential hits. We then simulated the behavior of a simple adaptive algorithm that chooses between two algorithms based on either recent history or overall performance, showing that adaptivity can be used to improve caching performance.

Our experiments use a cache simulator, written in C++, that implements 12 different cache replacement policies: RAND, LRU, MRU, FIFO, LIFO, LFU, MFU, SIZE, GDS, GDSF, LFUDA [5], and GD* [19]. We implemented all of these policies for completeness—though some of these policies are never used in modern systems, inferior policies may be useful in mixtures. These policies are summarized in Table 1.

4.1 Static Heterogeneous

We extended the work of Busari and Williamson [7] 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 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 [7] 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 [7].

Table 2 shows the results for 5 of these policies. The first column gives the hit rate for the first level caches. GDSF has the highest first level 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 very low. 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 policy at the second level is always different than the policy in the first level. For example, with this workload using 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 ($HR\infty$) 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

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

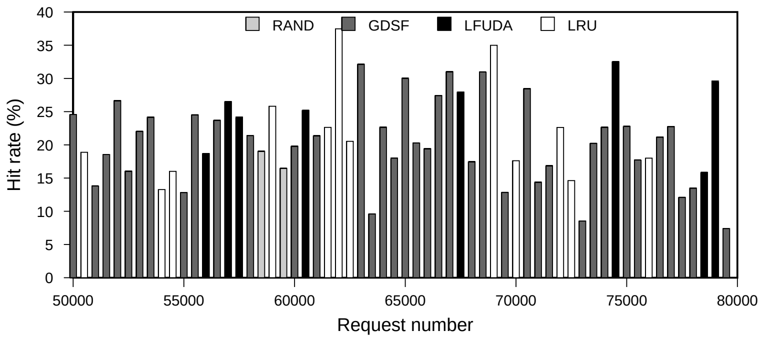
Table 2: Hit rate results in a 2-level cache using a ProWGen workload with 200,000 requests. The hit rate (HR) of policies at the first level cache is given in column 2. Column 3 lists the second level HR when the same policy is used. Column 4 shows the benefit of using a different policy at the second level by giving the results of the 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.

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.

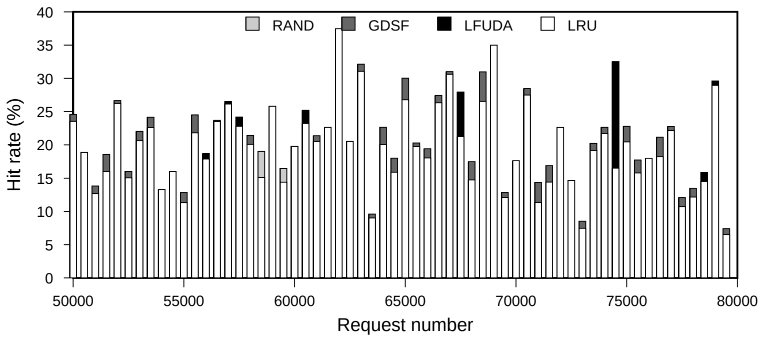
4.2 Rationale for Adaptive Caching

As the characteristics of the workload change over time, performance 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 hit rate or byte hit rate. These values are reported after the "warmup" 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 may be different, as shown in Figure 3. We term this behavior *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 3 shows the existence of switching in real workloads using a web proxy trace gathered at Digital Equipment Corporation (DEC) [14]. This proxy served 14,000 workstations, and was taken on September 16, 1996. The trace contains 1,245,260 requests for 524,616 unique items, consisting of approximately 6 GBytes of unique data, for a $HR\infty$ of 57.9%. We tested twelve policies on this trace, giving each 64 MBytes of cache space. Only a few policies dominated for



(a) This graph shows the existence of switching of the best current policy in the DEC trace for a segment of the trace; different shadings correspond to different policies doing best in each 500 request interval.



(b) This graph shows the loss due to the use of a static algorithm—LRU—during each interval. The shaded area above the LRU bar shows the improvement that could have been obtained by using a different algorithm. LRU was chosen for the baseline because its cumulative loss was around around 3%. For LFUDA and GDSF, the average cumulative loss was around 5%.

Figure 3: These graphs show the byte hit rate of various policies on a Web cache trace over 500 request intervals using a 64 MByte cache. The top graph shows only the best policy for each interval, and the bottom graph plots the same data showing both LRU and whichever algorithm performed best for the interval.

more than 1–2 intervals; these are the policies shown in Figure 3. The byte hit rates are measured in intervals of 500 requests; Figure 3a shows the policy that “won” for each interval. The graph shows that the best policy keeps changing for different time slots even after warmup period of a single well-characterized workload.

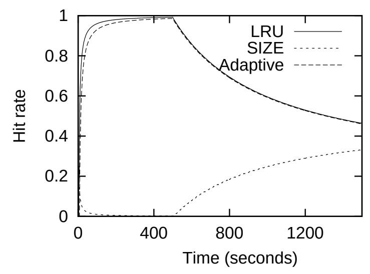
Figure 3b shows the same data, but also plots the performance of the best overall static policy, LRU, for each interval. The shaded bars on the top of the LRU byte hit rates indicate that, for many intervals, other policies were better than LRU. The cumulative average of the difference between the byte hit rate of best policy and particular static policies, *i.e.* the loss due to using static policies, was around 3% absolute for LRU and 5% absolute for both LFUDA and GDSF. Since average byte hit rate was under 20%, using adaptive policies could increase the number of bytes provided by the cache by 15%-25% or more, with corresponding reductions in bandwidth and response time.

Different static policies may be more successful with different workloads, therefore choosing a single static policy will result in different losses with different workloads. 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.

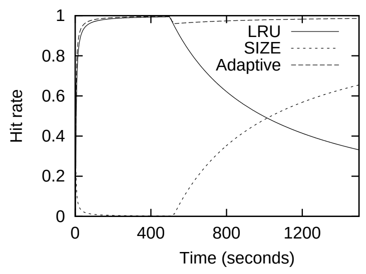
It is vital that the opinion of each expert is heard and considered at all times. If a highly opinionated group or decision-maker ignores the decisions of the experts that have made weak or unsuccessful predictions in the past, then group may run into the danger of only following one strong static expert (*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.

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. Figure 4b shows that an implementation that only looks at overall 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. However, a good adaptive algorithm implementation, shown in Fig. 4b, can look at recent success to quickly switch to using the SIZE policy maintaining a continuous high hit rate.

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 request streams that appear random, there is hope for improvement. Workloads in which references made to randomly-chosen objects will likely favor algorithms that cache smaller objects because they can cache more of them, perhaps gaining additional hits from keeping more objects. Similarly, algorithms that cache distant objects may do better on



(a) Schemes that look at cumulative success will stick with the overall best policy, causing performance to suffer when the workload changes.



(b) Adaptive schemes that look at recent success can quickly switch to currently successful policies and provide continuous high performance.

Figure 4: Hit rate for a synthetic workload that changes after 500 seconds.

such workloads because they provide the highest benefit when a hit *does* occur.

5 Future Work

Our current design and implementation constitute only proofs of concepts. In order to make adaptive caching an effective technique, we must discover which machine learning algorithms best adapt to Web and file system workloads. Simply choosing the best algorithms is not enough, however, unless caching can be embedded in networks with little performance penalty.

Real implementations will enforce us to minimize space and computational overheads and make performance trade-offs. Actual implementations will also require the use of more efficient data structures, such as B-trees [13] and B+-trees [23] used in file systems and databases. For example, a Unified Buffer Management (UBM) scheme for the FreeBSD file system was implemented and tested by Kim *et al.* [20] and user response times were improved by 67.2% (with an average of 28.7%). Adaptive caching must use similar optimizations or suffer from unacceptably high costs to make good predictions.

6 Conclusions

We presented adaptive caching schemes applicable to single and multiple processor systems. Adaptive caching helps with the management of distributed 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.

Acknowledgments

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 members of the Storage Systems Research Center as well as Manfred Warmuth and the members of the Machine Learning Group at UC Santa Cruz. Finally, we thank the participants of WDAS for helping us refine the ideas in this paper.

References

[1] ALMEIDA, V., BESTAVROS, A., CROVELLA, M., AND DE OLIVEIRA, A. Characterizing reference locality in the WWW. In *Proceedings of the 1996 International Conference on Parallel and Distributed Information Systems (PDIS '96)* (Dec. 1996).

[2] AMER, A., AND LONG, D. D. E. 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.

[3] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002).

[4] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Y. Serverless network file systems. *ACM Transactions on Computer Systems 14*, 1 (Feb. 1996), 41–79.

[5] ARLITT, M., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., AND JIN, T. Evaluating content management techniques for web proxy caches. In *Proceedings of the 2nd Workshop on Internet Server Performance (WISP '99)* (Atlanta, Georgia, May 1999).

[6] BECK, M., BOHME, H., DZIADZKA, M., KUNITZ, U., MAGNUS, R., AND VERWORNER, D. *Linux Kernel Internals*, 2nd ed. Addison–Wesley, 1998.

[7] BUSARI, M., AND WILLIAMSON, C. 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)* (Cincinnati, OH, Aug. 2001), IEEE, pp. 379–388.

[8] CAO, P., AND IRANI, S. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)* (Dec. 1997), pp. 193–206.

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

[10] CAO, P., ZHANG, J., AND BEACH, K. Active cache: Caching dynamic contents on the web. In *Proceedings of the 1998 Middleware Conference* (1998).

[11] CASTRO, M., ADYA, A., LISKOV, B., AND MYERS, A. C. HAC: Hybrid adaptive caching for distributed storage systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (1997), pp. 102–115.

[12] CHUANG, J., AND SIRBU, M. Stor-serv: Adding quality-of-service to network storage. In *Proceedings of Workshop on Internet Service Quality Economics* (Cambridge MA, Dec. 1999).

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

[14] DIGITAL EQUIPMENT CORPORATION (DEC). Digital's web proxy traces. Available from ftp://ftp.digital.com/pub/DEC/traces, Sept. 1996.

[15] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking 8*, 3 (2000), 281–293.

[16] HELMBOLD, D. P., LONG, D. D. E., AND SHERROD, B. 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)* (Rye, New York, Nov. 1996), ACM, pp. 130–142.

[17] HERBSTER, M., AND WARMUTH, M. K. Tracking the best expert. In *Proceedings of the 12th International Conference on Machine Learning* (Tahoe City, CA, 1995), Morgan Kaufmann, pp. 286–294.

[18] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WES, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[19] JIN, S., AND BESTAVROS, A. 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).

[20] KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. 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)* (San Diego, CA, Oct. 2000), pp. 119–134.

[21] MICHEL, S., NGUYEN, K., ROSENSTEIN, A., ZHANG, L., FLOYD, S., AND JACOBSON, V. Adaptive Web caching: towards a new global caching architecture. *Computer Networks and ISDN Systems 30*, 22-23 (1998), 2169–2177.

[22] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the Sprite network file system. *ACM Transactions on Computer Systems 6*, 1 (1988), 134–154.

[23] RAO, J., AND ROSS, K. A. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, TX, May 2000), pp. 475–486.

[24] RIZZO, L., AND VICISANO, L. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking 8*, 2 (2000), 158–170.

[25] ROSELLI, D., LORCH, J., AND ANDERSON, T. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000).

[26] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (Mar. 1994), 17–29.

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

[28] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (1997), pp. 224–237.

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

[30] WEIKUM, G., KONIG, A. C., KRAISS, A., AND SINNWEL, M. Towards self-tuning memory management for data server. *Data Engineering Bulletin 22*, 2 (1999), 3–11.

[31] WILKES, J., RICKARD, W., GIBSON, G., ANDERSON, D., AND BLACK, D. Shared storage model. a framework for describing storage architectures. Technical Council Proposal Document draft-june5, SNIA, June 2001.

[32] WONG, T. M., GANGER, G. R., AND WILKES, J. My cache or yours? Making storage more exclusive. Tech. rep., CMU-CS-00-157 Carnegie Mellon University, Nov. 2000.

[33] WOOSTER, R., AND ABRAM, M. Proxy caching that estimates page load delays. In *Proceedings of the 6th International World Wide Web Conference* (Santa Clara, CA, Apr. 1997), pp. 325–334.

[34] YU, H., BRESLAU, L., AND SHENKER, S. A scalable web cache consistency architecture. In *Proceedings of SIGCOMM99* (1999), pp. 163–174.

Ismail Ari is a Ph.D. student in the Computer Science Department at the University of California, Santa Cruz, where he does research as a member of the Storage Systems Research Center. E-mail: `ari@cs.ucsc.edu`.

Ahmed Amer will receive his Ph.D. in Computer Science from the University of California, Santa Cruz in Summer, 2002. He will join the Computer Science Department at the University of Pittsburgh in Fall, 2002. E-mail: `amer4@cs.ucsc.edu`.

Robert Gramacy is a Ph.D. student in the Computer Science Department at the University of California, Santa Cruz, where he is a member of the Machine Learning Group. E-mail: `rbgramacy@cs.ucsc.edu`.

Ethan Miller is an Assistant Professor in the Computer Science Department at the University of California, Santa Cruz, where he is a member of the Storage Systems Research Center. E-mail: `elm@acm.org`.

Scott Brandt is an Assistant Professor in the Computer Science Department at the University of California, Santa Cruz, where he is a member of the Storage Systems Research Center. E-mail: `sbrandt@cs.ucsc.edu`.

Darrell D. E. Long is a Professor in the Computer Science Department at the University of California, Santa Cruz, where he is the Director of the Storage Systems Research Center. E-mail: `darrell@cs.ucsc.edu`.