SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:1179–1188
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.755

Self-tuning caching: the Universal Caching algorithm

Ganesh Santhanakrishnan, Ahmed Amer*,† and
Panos K. Chrysanthis

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.

SUMMARY

A cache replacement policy is normally suited to a particular class of applications, or limited to a set of fixed criteria for evaluating the cache-worthiness of an object. We present *Universal Caching* as a mechanism to capture the generality of the most adaptive algorithms, while depending on a very limited set of basic criteria for cache-replacement decisions. Our testing was limited to Web workloads, where we tested the *Universal Caching* policy using real-world traces from both the server-side and client-side proxies. Using a self-tuning mechanism, combined with a generalization of the criteria employed in GD-* Web caching, Universal Caching was able to consistently outperform any other fixed choice of algorithm we tested. Copyright © 2006 John Wiley & Sons, Ltd.

Received 19 April 2004; Revised 11 April 2006; Accepted 12 April 2006

KEY WORDS: caching; adaptive; automated; self-tuning; self-optimizing

1. INTRODUCTION

One of most prevalent techniques for improving storage system, network, and device performance is caching. There have been many studies into developing the best caching algorithms, focusing on both improving the choice of item to replace, as well as developing techniques to model data-access behavior and prefetch data. In this work, we present a mechanism to automatically adapt the replacement policy in response to the observed workload. We depart from prior art by utilizing simple criteria, in contrast to schemes that simply combine multiple arbitrary algorithms. We also differ from simpler adaptive

*Correspondence to: Ahmed Amer, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.
†E-mail: amer@cs.pitt.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: ANI-0325353

Copyright © 2006 John Wiley & Sons, Ltd.

schemes, such as those employed for handling page evictions, in the universal nature of our model. For this reason we named our scheme *Universal Caching* (UC).

UC is suitable for managing object caches of variable-sized objects, efficient enough to be implemented with minimal computational costs, and can be easily extended to capture all observable criteria influencing the performance of the cache-replacement scheme. Criteria are essentially any readily observable properties of a request for data, typical examples of which include the recency of prior requests for the same data, or the frequency of such requests.

Before we go on to describe UC in more detail, in the next section we provide an overview of existing caching schemes. We introduce UC in Section 3. In order to evaluate our algorithm, we have developed a detailed simulator and tested it using real-world traces from both the server-side and client-side proxies. In Section 4 we present results for these trace-driven tests.

2. BACKGROUND AND RELATED WORK

Ghost (or shadow) caches are a known technique of tracking more entries than the cache capacity, greedy-dual (GD) algorithms differ from simpler caching schemes such as least recently used (LRU) and least frequently used (LFU) in that they use a cost function to form a priority list of items in the cache. Other algorithms have also used such techniques to rank in-cache data. More recently, self-adaptive caching policies have also been proposed [1,2]. Our earlier GD-GhOST policy [3] is actually an adaptive caching algorithm based on a dynamic combination of GD-Size variants [4] as component algorithms. Our UC policy is also an adaptive caching algorithm but based on a dynamic combination of criteria.

2.1. Simple caching, simple criteria

The least-normalized-cost replacement (LNC) [5] policy inserts the documents into a priority queue with a priority key. The problem with this policy is that it has tunable parameters that are workload dependent. Another policy known as the low interference recency set (LIRS) [6] maintains a variable size LRU stack whose LRU page is the *L(lirs)*-th page that has been seen at least twice recently, where *L(lirs)* is a parameter. As suggested in the paper, the setting of *L(lirs)* to 1% of the cache size will be good for independent reference model (IRM) workloads, but it does not perform as well for LRU stack depth distribution (SDD) workloads.

The frequency-based replacement (FBR) [7] policy combines both the frequency of access and recency of access. It divides the LRU list into three sections, and maintains a counter for every document in the cache. The algorithm has several tunable parameters. In order to prevent cache pollution from stale pages with high reference counters, all of the reference counters must be periodically rescaled. A class of policies known as least recently/frequently used (LRFU) [8] were shown to subsume the LRU and LFU policies. These policies have an update rule which is a form of exponential smoothing that is widely used in statistics. It effectively balances both the LRU policy and LFU policies. There is an adaptive version of the policy known as adaptive least recently/frequently used (ALRFU) [9] policy. This policy basically has a mechanism to dynamically adjust the parameter used in the basic LRFU policy. Again, both of these LRFU schemes require tunable parameters.

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2.2. Self-adaptive caching policies

A recently proposed self-tuning caching policy is known as the adaptive replacement cache (ARC) [1]. It maintains two variable sized LRU lists. It captures both recency and frequency of access well, and provides an elegant, efficient and effective mechanism for combining them. It is intended as a page replacement policy, and so it does not consider the different delays in fetching a document, and variable object sizes, which are important factors in general object caching, as with Web content caching.

Our general approach is most similar to adaptive caching using multiple experts (ACME) [2]. ACME uses a mixture of arbitrary policies that are treated as experts. Machine learning algorithms are applied to combine the recommendations of the different policies based on their ordering. We differ from ACME in two ways: by providing a more direct evaluation of each element's relative importance, and allowing the evaluation of an arbitrary selection of performance criteria. By using GD-Size variants as component algorithms we are able to proportionally scale the credits based on the normalized cost functions. ACME restricts policy information to orderings in exchange for a greater generality in terms of applicable policies. This means that GD-GhOST introduced in [3] produces a new evaluation function instead of a switching among policies or a rigid mixed-weighting policy.

The Greedy-Dual * (GD-*) Web caching algorithm [10] is known to be superior to many other Web cache replacement policies, but our approach differs in that we attempt to optimize a user-specified combination, or selection, of performance goals.

As with GD-* we also differ from ACME in our ability to incorporate arbitrary combinations or selections of performance criteria. Finally, a motivating factor for these dynamic schemes, as for multi-queue algorithms [11], is to adapt policies automatically for cases where multiple caches can have adverse interactions. A recent work that addresses harmful cache interactions is that of Wong and Wilkes [12], where demotions were used as a mechanism to explicitly ensure cache exclusivity. Demotions require communication between these multiple caches, an assumption we do not make.

2.3. GD-GhOST caching

Combining multiple algorithms can be performed by combining a multi-expert machine learning algorithm with virtual caches, and using different replacement algorithms for each virtual cache. This approach was first presented with ACME [2]. For machine learning algorithms, such as 'weighted majority' and 'share' [13], the term expert refers to any mechanism for offering an answer to the question. It can be any arbitrary algorithm for predicting the correct value, or even a sample answer *per se*. For cache replacement, the answer we seek is the identity of the object in the cache with the least likelihood of subsequent future access. The actual answer offered by the full algorithm is, in fact, a weighted combination of each expert's individual answer, and the contribution of the machine learning algorithm is to provide a mechanism to update these weights based on the relative performance of the individual experts. For ACME, the individual experts were full replacement algorithms, applied to virtual caches, and their performance was estimated based on the observed performance of these virtual caches.

GD-GhOST (goal-oriented, self-tuning) caching is based on a combination of several GD-Size variants, and attempts to satisfy a given performance goal using a fully adaptive combination of these individual component algorithms [3]. Similar to ACME, GD-GhOST combines individual component algorithms using a master-algorithm approach [2]. For cache eviction decisions, the items with the

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

lowest combined H values (weighted by combined credit values for each component algorithm) are the first selected for eviction. Initially we assign equal credits to each of the policies. Based on the performance of the component algorithms, we distribute the credits for each algorithm in the following manner.

$$Credit_{\mathrm{c}} = (Perf_{\mathrm{o}} - Perf_{\mathrm{i}}) \cdot Credit_{\mathrm{p}}$$

where $Credit_{\mathrm{c}}$ is the current credits for the algorithm, $Credit_{\mathrm{p}}$ is the previous credits for the algorithm, $Perf_{\mathrm{o}}$ is the overall combination/selection of hit ratio and byte hit ratio, $Perf_{\mathrm{i}}$ is algorithm i's performance based on the combination/selection of hit ratio and byte hit ratio.

The credits are distributed among the algorithms every time we evaluate their relative performance. This is not done with every access, but at a variable interval. There is no need to manually set this interval, as it is automatically adjusted based on variations in relative algorithm performance. If the workload is such that there is a consistent combination of algorithms to maximize performance, then the period is automatically lengthened. As updates are needed more rapidly, the period is automatically reduced. This on-line update of credits ensures that at any instant in time, we are most likely to follow the leader among the three algorithms, for the metrics that are considered most important. When the best performing algorithm degrades in performance, the redistribution of credits ensures that it does not degrade the overall performance. We have actually observed that GD-GhOST can follow the best performance of the three component algorithms, and frequently exceeds it.

With the GD-GhOST algorithm [3], and subsequently UC, we have moved away from the use of arbitrary collections of component algorithms, and focused on a reduced number of complimentary algorithms. GD-GhOST caching limited the component algorithms to variants of GD-Size. This had the advantage of using component algorithms that offered a numeric evaluation of each object in the cache. In this manner, it was possible to weight these numeric values directly, rather than infer expert answers based solely on the ordering of items in the cache. This approach demonstrated notable performance improvements, using only three component algorithms [3]. UC, which we describe in the following section, offers further performance improvements, but its most notable feature is that it relies on a set of three properties of each cached object. These properties, or base criteria, were derived from the intuitions expressed in GD algorithms, and can be directly calculated or estimated for each cached object.

3. THE UC POLICY

We now present our proposed UC algorithm, which offers a self-optimizing replacement algorithm that is usable for general object caching, and yet is based on the simplest base criteria. As discussed above, prior work has demonstrated the feasibility of constructing adaptive caching algorithms, but such efforts have either been restricted to page replacement and limited criteria, as with ARC [1], or else based on combinations of arbitrary replacement algorithms, as with ACME [2]. In contrast, UC uses a very simple set of object properties to select which objects (of varying size) will be removed from the cache.

GD cache replacement algorithms are based on ranking objects by their cost of retrieval (H-values), and replacing the objects with the lowest retrieval cost. To account for variable page sizes the retrieval

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

cost is divided by the page size in GD-Size algorithms. For an object p, the H-value is calculated as

$$H(p) = \frac{cost(p)}{size(p)} \tag{1}$$

The division by the page size accounts for the intuition that replacing larger pages frees up more space in the cache than removing (the same intuition behind ‘SIZE’ replacement that evicts the largest objects first). Different variants of GD-Size use different estimates of retrieval cost. For example, GD-1 assumes all objects have an equal replacement cost ($cost(p) = 1$), while GD-Size Frequency (GDSF) assumes that the cost of replacement is proportional to the frequency of access of the object ($cost(p) = frequency(p)$). Yet another variant, GD-Size Packets, estimates the cost of retrieval as proportional to the number of packets that must be sent to transfer the object ($cost(p) = 2 + size(p)/network\ packet\ size$).

All variants of GD-Size algorithms are essentially providing different estimates for the same basic formula for evaluating the merit of retaining objects in the cache. This formula can be expressed as

$$H(p) = \frac{L(p) \cdot C(p)}{B(p)} \tag{2}$$

where $L(p)$ is the access-likelihood of p, $C(p)$ is the cost of retrieving p and $B(p)$ is the benefit from evicting p.

All cache replacement algorithms can be interpreted as different simplifications and estimates of this formula. The benefit from evicting an object is best described by the cache capacity freed by such an eviction. This does not differ among the GD-Size variants, but from Equation (2), it should be apparent that increases in the replacement cost of an object, weighted by the likelihood of future access, are equivalent to *expected* costs incurred by evicting an object. The variations among the GD-Size variants appear in their estimates of this expected replacement cost. More importantly, we suggest that there is no cache replacement policy that does not perform an estimate of Equation (2) when deciding what to evict.

The UC algorithm draws on this small set of object properties (likelihood $L(p)$, cost $C(p)$ and benefit $B(p)$), and directly combines estimates of them to provide a self-optimizing and potentially comprehensive caching policy. For the UC algorithm, estimators for these values are drawn from simple observable criteria of the access workload. Assuming that we have three estimators (criteria) that accurately describe these three properties, a possible mechanism to vary their combination is to provide a weight for each property. Unfortunately, it would be much simpler to automatically adjust such weights if these properties were a sum of ‘expert’ values, and not a product. This is easily achieved by taking the log-sum of the product in Equation (2), giving us

$$H(p) = w_L \cdot \log(L(p)) + w_C \cdot \log(C(p)) - w_B \cdot \log(B(p)) \tag{3}$$

In the following section we see that using logarithms has a distinct impact on algorithm performance. For Equation (3) to be the basis of an adaptive caching algorithm, we need to select estimators of likelihood, cost and benefit, as well as a mechanism to update the weights.

The choice of estimators can be debated, but for our experiments we selected the simplest possible set. The benefit from evicting object p was taken as the size of the object, while the cost of retrieval was also estimated as the size of the object. It should be pointed out that such conflicting uses of the same value are perfectly reasonable. If the cost of retrieving object p outweighs the benefit from evicting it,

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

this can be reflected by increasing w_C relative to w_B. It should also be made clear that the estimator need not be an accurate estimate, merely that it should be proportional to the value being estimated.

Estimating the likelihood of future access was done using two different estimators: frequency of access, and recency of last access. As these two criteria were used as two distinct experts, we avoid any need to balance the mix of the two criteria (as is done manually in LRFU algorithms, and automatically in ARC [1]).

4. EXPERIMENTAL RESULTS

We conducted simulation-based experiments on real-world traces to evaluate the UC policy and the effect of Web-cache filtering effects on the GD-GhOST policy. We tested the ability of the UC policy and the GD-GhOST policy to match the performance of the best policy for the selected performance metric. Specifically, we tested the ability of the UC and GD-GhOST algorithms to maximize hit ratios and byte hit ratios, which we will present in this section through minimizing miss rates and byte miss rates. Both algorithms are compared against the most competitive GD algorithms for the selected workloads.

4.1. Workload description

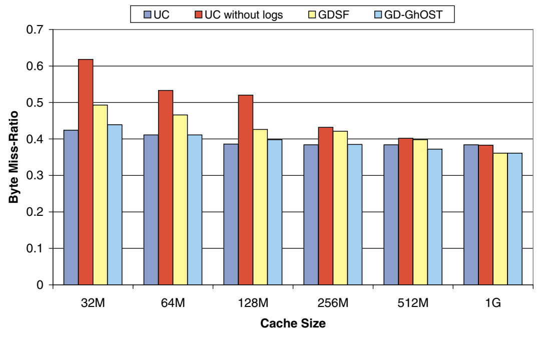
Experiments were conducted using traces run against a cache simulator implemented in Java. The three GD-Size algorithms (GD-Size(1), GD-Size(packets), and GD-Size(frequency)) were implemented, along with the GD-GhOST and UC policies. For GD-GhOST and UC the credits for the components (algorithms or criteria) were updated on-line using the proportional weighted averaging described in Section 3.

For UC, the four criteria were implemented with each operating on its own ghost cache. This was a compromise to simplify testing and comparison with GD-GhOST, but ultimately the goal will be to completely eliminate the need for ghost caches by evaluating criteria values in relation to future accesses.

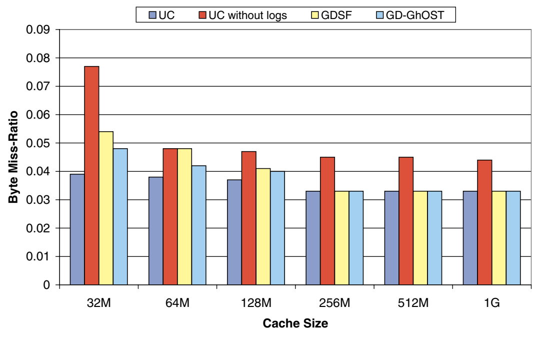
The four criteria were also implemented without using their log based components. We refer to the UC policy thus implemented as the UC policy without logs. The credits for the individual replacement criteria were updated on-line using the proportional weighted averaging described in Section 3. As described above, the trial period for updating the credits was dynamically adjusted and required no *a priori* settings.

We tested with different cache sizes and using both client and proxy Web traces. Specifically, we used traces from Boston University [14] and the 1998 World Cup [15]. The Boston University traces contain records of the HTTP requests and user behavior of a set of Mosaic clients running in the Boston University Computer Science Department, spanning from 21 November 1994 through 8 May 1995. During the data-collection period, a total of 9633 Mosaic sessions were traced, representing a population of 762 different users, and resulting in 1 143 839 requests for data transfer. There were 5–32 workstations. The World Cup 98 data set consists of all the requests made to the 1998 World Cup Web site between 30 April 1998 and 26 July 1998. During this period of time the site received 1 352 804 107 requests.

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

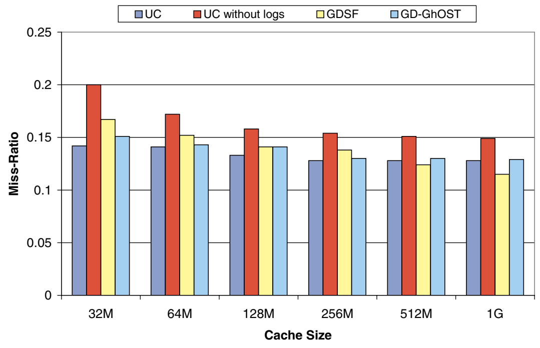


(a)

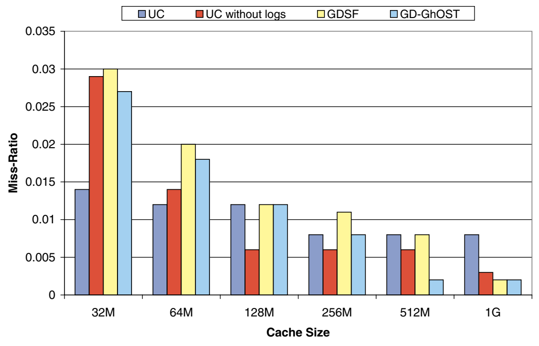


(b)

Figure 1. Average byte miss ratio results for varied cache sizes: (a) Boston University trace; (b) World Cup 98 trace.



(a)



(b)

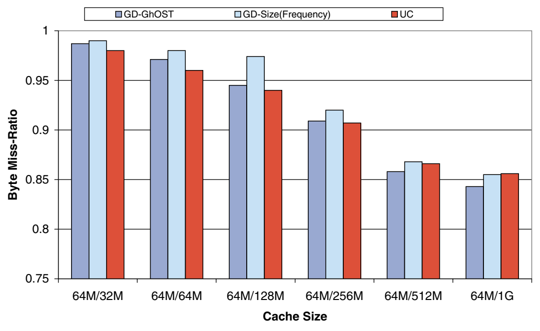
Figure 2. Average miss ratio results for varied cache sizes: (a) Boston University trace; (b) World Cup 98 trace.

4.2. UC results

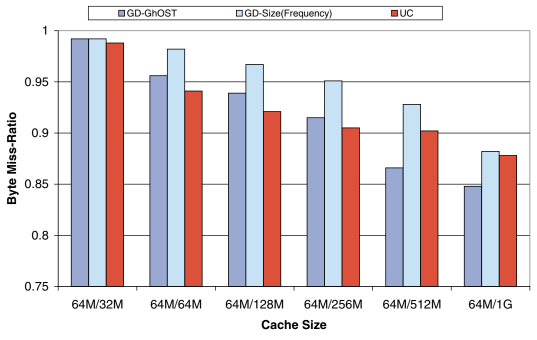
We present the miss ratios for the UC policy in Figure 2 and the byte miss ratios in Figure 1. These results are for cache sizes of 32 MB, 128 MB and 1 GB. We compare the performance of the UC policy based Equation (3) (with logs), to that of the UC policy without logs (Equation (2)), the GD-GhOST policy [3], and the GDSF policy [16].

We see that our UC policy (with logs) based on byte hit-ratio, performs on par with the other competing variants exceeding them on several occasions. When compared based on hit-ratio, it performs among the top two policies while performing better than others on more than a couple of occasion's. The UC policy not based on logs performs slightly poorer than that based on logs.

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(a)



(b)

Figure 3. Web cache filtering results for level-1 filtered Boston University trace: (a) LRU-filtered level-1; (b) GDSF-filtered level-1.

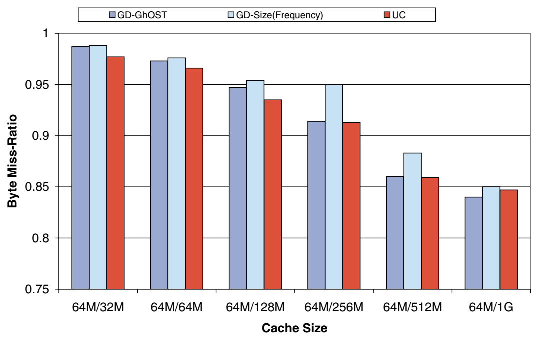
The striking result from Figure 1 is that our UC policy consistently outperforms the GDSF policy in terms of byte hit ratios. From Figure 2, we see that for simple hit ratios, where all object misses are equal regardless of object size, UC performs only slightly worse for the largest of cache sizes. While the overall value ranges differ for the two workloads, it is important to note that the trends are consistent regardless of whether the workload is from a server or from a client-side proxy.

4.3. Web cache filtering results

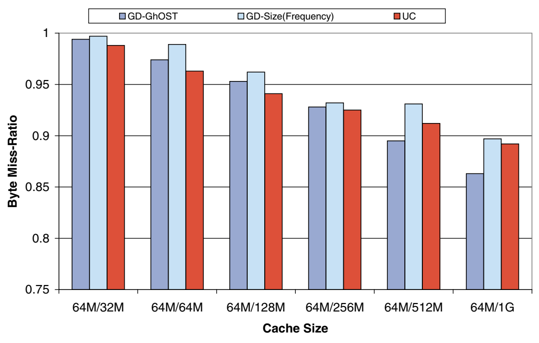
Recognizing that in most configurations, the request path from clients to servers can involve multiple caches, at least two at browser/proxy and at the server, that is why we performed experiments with Web cache filtering. The Web cache filtering effects described here were simulated as follows: we allowed the Boston University and World Cup 98 traces to pass through a cache employing a specific policy. Then we ran the filtered trace through the cache employing the GD-GhOST policy, its constituents and the UC policy. For clarity and space we present the results for the Boston University trace, and limit the constituents to the GD-Size(Frequency) policy, which was the best performing GD-Size variant for the workloads tested. While the results presented here are representative, we felt that filtering the Boston University traces would most realistically reflect a server-caching scenario where client requests have passed through prior caching levels.

We performed the simulation for the Web cache filtering effects using varying cache sizes. We first experimented with LRU filtered caches of size 32 MB, 64 MB, 128 MB, 256 MB, 512 MB and 1 GB with the same cache sizes for algorithm. We then fixed the cache size of the filtering cache as 64 MB since we felt that it was representative. We also experimented with the GD-Size(1), GD-Size(Packets), and GD-Size(Frequency) filtered caches of size 64 MB using cache sizes of 32 MB, 64 MB, 128 MB, 256 MB, 512 MB and 1 GB. We also experimented with two levels of cache filtering employing the same policies, and using the same cache sizes as described above.

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License



(a)



(b)

Figure 4. Web cache filtering results for level-2 filtered Boston University trace: (a) LRU-filtered level-2; (b) GDSF-filtered level-2.

We compare the results for byte miss ratios for the UC, GD-GhOST and GD-Size(Frequency) policies. Results for both levels of filtered workloads are shown in Figures 3 and 4. The miss ratios are generally much higher under these conditions, but this is true regardless of policy, and the UC results are again very encouraging. Our policy consistently outperforms GD-Size(Frequency) and the GD-GhOST policy for small and medium sized caches, while performing only slightly worse for the largest cache capacities.

5. CONCLUSIONS

Our major contribution in this paper is an adaptive caching policy (UC) that demonstrates its ability to use a set of simple criteria as a basis for auto-tuning the cache behavior. In summary, we can see that UC performs very well, achieving the highest byte hit ratios among fixed selections of GD algorithms, and even against a self-optimizing combination of these algorithms. The comparison was based on Web workloads, but they differed greatly in their nature. While one was a client-side proxy trace, the other was a busy server log. Regardless of the origin of the trace, UC consistently performed well, demonstrating its ability to use a set of simple criteria as a basis for auto-tuning the caching policy. Our testing was limited to Web workloads, where we tested the UC policy using real-world traces from both the server-side and client-side proxies, but we are currently investigating its applicability and usefulness in different domains, including resource-constrained environments such as mobile computing [17].

ACKNOWLEDGEMENTS

We would like to thank our colleagues at the University of Pittsburgh: Milos Hauskrecht for valuable discussions, and Huiming Qu and Dan Li for early contributions to this work. We would also like to thank members of the Storage Systems Research Center, and the Machine Learning Group at the University of California, Santa Cruz, for valuable discussions.

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

REFERENCES

1. Megiddo N, Modha DS. ARC: A self-tuning, low overhead replacement cache. *Proceedings of the USENIX File and Storage Technologies Conference (FAST)*, March 2003. USENIX Association: Berkeley, CA, 2003.
2. Ari I, Amer A, Gramacy R, Miller EL, Brandt S, Long DDE. Adaptive caching using multiple experts. *Proceedings of the Workshop on Distributed Data and Structures*, Paris, 20–23 March 2002. Carleton Scientific, 2002.
3. Santhanakrishnan G, Amer A, Chrysanthis PK, Li D. GD-GhOST: A goal-oriented self-tuning caching algorithm. *Proceedings of the 19th ACM Symposium on Applied Computing*, March 2004. ACM Press: New York, 2004.
4. Cao P, Irani S. Cost aware WWW proxy caching algorithms. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997. USENIX Association: Berkeley, CA, 1997.
5. Scheuermann P, Junho Shim RV. A case for delay-conscious caching of Web documents. *Computer Networks and ISDN Systems* 1997; **29**(8–13):997–1005.
6. Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *Proceedings of the ACM SIGMETRICS Conference*, January 2002. ACM Press: New York, 2002.
7. Robinson JT, Devarakonda MV. Data cache management using frequency-based replacement. *Proceedings of the ACM SIGMETRICS Conference*, May 1990. ACM Press: New York, 1990.
8. Lee D, Choi J, Kim J, Noh S, Min S, Cho Y, Kim C. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers* 2001; **50**(12):1352–1361.
9. Lee D, Choi J, Kim J-H, Noh SH, Min SL, Cho Y, Kim CS. On the existence of a spectrum of policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. *Proceedings of the ACM SIGMETRICS Conference*, June 1999. ACM Press: New York, 1999.
10. Jin S, Bestavros A. GreedyDual* Web caching algorithm: Exploiting the two sources of temporal locality in Web request streams. *International Journal on Computer Communications* 2001; **24**(2):174–183.
11. Zhou Y, Philbin JF, Li K. The Multi-Queue replacement algorithm for second level buffer caches. *Proceedings of the USENIX Annual Technical Conference*, June 2001. USENIX Association: Berkeley, CA, 2001.
12. Wong TM, Wilkes J. My cache or yours? Making storage more exclusive. *Proceedings of the USENIX Annual Technical Conference*, June 2002. ACM Press: New York, 2002.
13. Herbster M, Warmuth MK. Tracking the best expert. *Machine Learning (Special Issue on Context Sensitivity and Concept Drift)* 1998; **32**(2):151–178.
14. Cunha CA, Bestavros A, Crovella ME. Characteristics of WWW client traces. *Technical Report TR-95-010*, Department of Computer Science, Boston University, Boston, MA, April 1995.
15. Arlitt M, Jin T. 1998 World Cup Web site access logs. http://ita.ee.lbl.gov/html/traces.html [9 June 2006].
16. Cherkasova L. Improving WWW proxies performance with Greedy-Dual-Size-Frequency caching policy. *Technical Report*, Hewlett-Packard, Palo Alto, CA, November 1998.
17. Santhanakrishnan G, Amer A, Chrysanthis PK. Towards universal mobile caching. *Proceedings of the 4th ACM Workshop on Data Engineering for Wireless and Mobile Access*, June 2005. ACM Press: New York, 2005.

Copyright © 2006 John Wiley & Sons, Ltd.

1097024x, 2006, 11-12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.755 by University Of Pittsburgh, Wiley Online Library on [11/11/2025]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License