Adaptive Cache Replacement:A Novel Approach

Sherif Elfayoumy
School of Computing
University of North Florida
Jacksonville, Florida

Sean Warden
School of Computing
University of North Florida
Jacksonville, Florida

***Abstract*—Cache replacement policies are developed to help insure optimal use of limited resources. Varieties of such algorithms exist with relatively few that dynamically adapt to traffic patterns. Algorithms that are tunable typically utilize off-line training mechanisms or trial-and-error to determine optimal characteristics.**

Utilizing multiple algorithms to establish an efficient replacement policy that dynamically adapts to changes in traffic load and access patterns is a novel option that is introduced in this article. A simulation of this approach utilizing two existing, simple, and effective policies; namely, LRU and LFU was studied to assess the potential of the adaptive policy. This policy is compared and contrasted to other cache replacement policies utilizing public traffic samples mentioned in the literature as well as a synthetic model created from existing samples. Simulation results suggest that the adaptive cache replacement policy is beneficial, primarily in smaller cache sizes.

Keywords—cache replacement policy; high performance computing; adaptive caching; Web caching

I. INTRODUCTION

Caching in computing has been a proven form of performance enhancement for some time, most notably in memory paging [1] [2]. The basic premise is that objects that are frequently used or most likely to be used can be stored in a location that provides performance benefits by virtue of being temporally or locally near the consumer of the object. Temporally meaning that an object may be served more quickly from a given location when compared to the service time from another host location, possibly due to reduced latency of access from faster resources (disk, memory, etc.) or increased service bandwidth. Because cache resources are finite in size, however, one problem in caching is that there exists a wide variety of algorithms for replacing objects in a filled cache.

Effective caching algorithms are necessary to insure users experience favorable performance benefits, especially in an environment as diverse and distributed as the Internet. Such performance benefits are a reduction in the delay between the time a user requests an object and its delivery to the user.

Since the Web has evolved, several caching algorithms have been suggested in literature; Balamash and Krunz's survey of replacement algorithms described no fewer than twenty different replacement schemes [3]. Each strategy utilizes different parameters to determine objects to replace – Balamash classified caching policies based on whether they utilized frequency, recency, or size information. While some algorithms utilize a single traffic trait, others employ a functional approach to compute a derived 'cost' of an object cache miss based on multiple parameters, thus removing the lowest 'cost' object when a replacement is required. Not surprisingly, the research has shown that functional approaches are generally more computationally complex than those based on a single attribute [3]. Due to the variety of approaches, metrics, and parameters utilized (sometimes singly and sometimes in combination), each algorithm has distinct performance characteristics.

Since access patterns are unique to each environment, certain algorithms are more suitable than others depending on the traffic situation, and one single algorithm is not best in class for all situations. A web server that functions as a search engine will have different traffic patterns, and thus unique cache architecture requirements, when compared to a university web portal, online encyclopedia, multimedia server, etc. [4]. Algorithm designers (such as [5]) have attempted to overcome this problem using one or more static parameters that can be tuned offline to optimize performance through analysis of historical object requests.

Web caches are designed to provide apparent speed benefits to object requestors by offloading objects from the web server itself to a location that is physically or logically closer to the requestor and/or decreases the amount of load experienced by the web server [6]. A web cache typically stores its objects in some form of memory or disk. Because these storage resources are finite, however, cache replacement policy algorithms are utilized to determine which objects to remove from the cache as new objects are accessed which are deemed more productive to cache [7]. Ideally, these replacement policy algorithms will always choose to keep the objects that will provide the best performance.

Developing an algorithm that is optimal for all traffic patterns is a challenging problem. Some algorithms attempt to overcome this problem using static tuning parameters that are set using offline training, trial-and-error, or possibly via an educated guess. Others attempt to adapt dynamically, using computationally complex dynamic parameters that are based on historical object requests. Because of the ubiquity of caching across a variety of computing environments – microprocessors, web, thin-clients, wireless devices, etc. – and the wide variety of possible differences in traffic and access patterns within and across these environments, researching methods for adaptive performance optimization is a worthy objective.

In this article, a novel adaptive approach to finding an efficient cache replacement policy for a given traffic pattern by sectioning the cache storage space into areas managed by

separate object replacement policies is introduced and assessed. The overall cache space then utilizes a tuning algorithm to allow the overall cache to choose the best policy based on current access patterns. This effort focuses on caching within a web environment. However, the approach is extensible to other caching environments. In section II an overview of the most prominent cache replacement policies is provided, and in section III the adaptive cache replacement policy is described. The assessment approach and preliminary results are described in section IV. Last, section V highlights the conclusions.

II. Cache Replacement Policies

A variety of cache replacement policy algorithms have been designed and evaluated in literature, with a goal of maximizing cache effectiveness as measured using performance metrics such as Hit Ratio. Each algorithm generally utilizes one or more of three pieces of information about requested objects: recency of access, frequency of access, or size. Algorithms range from those which are very simple in that they only use a single traffic parameter, such as LRU and LFU, to those which are very complex, using multiple parameters as well as statically tuned constants, such as Hybrid and GDSF. In the case of LRV, dynamic probability functions – built based on static algorithm analysis – are included but at the cost of implementation challenges and computational complexity.

Evaluating the performance of replacement algorithms is typically accomplished using real-world web logs (also known as web traces). These web logs are cleansed to remove non-cacheable requests (such as cgi or other dynamically generated content), then run through a program to simulate the replacement algorithm by 'playing back' the cleansed web log. The simulation process calculates benchmarks over a range of cache sizes. The most common metrics are hit ratio (HR), or the ratio of cache hits to all requests; byte hit ratio (BHR), or the ratio of bytes returned from the cache to all bytes requested; and latency ratio (LR), or the delay experienced by a user for objects retrieved from the cache verses that experienced if no objects were cached. Other effectiveness measures include reduced packets, or the ratio of network packets avoided due to caching to the total packets that would have otherwise been seen by the server; and reduced hops, a similar measure that focuses on network hops between client and server. Balamash and Krunz define the most common measures as follows [3]:

$$HR = \frac{\sum_{i \in R} h_i}{\sum_{i \in R} f_i} \qquad BHR = \frac{\sum_{i \in R} s_i \cdot h_i}{\sum_{i \in R} s_i \cdot f_i} \qquad LR = \frac{\sum_{i \in R} d_i \cdot h_i}{\sum_{i \in R} d_i \cdot f_i}$$

Where: s_i = size of object *i;* f_i = total number of object requests for object *i;* h_i = total number of cache hits for object *i;* d_i = average server retrieval delay for object *i;* R = set of all requested objects.

A. Least Recently Used (LRU) Algorithm

The Least Recently Used algorithm essentially uses a single parameter to decide which object to remove from the cache: time since last access. The basic premise of the algorithm is that those objects that are most likely to be accessed will have been accessed more recently than those that are not as likely. While simple to implement and requiring less computational power than most other algorithms, LRU has been outclassed by several other replacement algorithms: Balamash and Krunz's experiments showed that for large cache sizes, LUV, GDS, and Hyper-G produced better results for both HR and LR [3], while Bahn et al. found that for large cache sizes, LUV, Hybrid, Size, Mix, and sw-LFU performed better for HR and LUV was better for LR [5].

B. Least Frequently Used (LFU) Algorithm

Another relatively simple algorithm, Least Frequently Used utilizes a frequency counter for each object in the cache. Objects that are most frequently accessed are thus more likely to remain in the cache and presumably provide benefit to future users. Similar to LRU in terms of ease of implementation and complexity, this algorithm has also been surpassed by algorithms that are more efficient; Bahn et al. noted that LUV, LNC-R-W3, GDS, and LRV outperformed LFU for HR, BHR, and LR [5].

C. Size Algorithm

The Size algorithm is another simple algorithm - size is the only measure utilized for eviction evaluation [8]. When an eviction is required, Size removes the largest object currently in the cache with the idea that users are less likely to re-request larger objects. Smaller objects, then, are more likely to remain in the cache long-term [3]. Additionally, this allows for a larger number of objects to remain in the cache, potentially improving hit rates for some traffic patterns. Algorithm implementation is simple when compared to algorithms using multiple parameters, but exhibits generally poor performance: Balamash et al. found that the Size algorithm was a middle-of-the-pack performer for HR and absolute worst for BHR and LR using a simulated DEC trace and compared against LUV, GDS, Hyper-G, LRU, and Hybrid algorithms.

D. Least Unified Value (LUV) Algorithm

The LUV algorithm is a more complex functional algorithm that "trie[s] to get the benefit of both LRU and LFU in one unified scheme [3]". Each object in the cache is assigned a value that is used during a replacement operation; the object with the lowest value is removed. Values for each object (*i*) are assigned by the following formula:

$$V_i(k) \stackrel{\Delta}{=} \frac{C_i}{s_i} \sum_{n=1}^{k} \left(\frac{1}{2}\right)^{\lambda \Delta_{k,i}}$$

Where: C_i is the cost of object *i;* s_i is the size of object *i;* λ is a static parameter in the range $0 \leq \lambda \leq 1$; $\Delta_{k,i}$ is the time since the k^{th} reference to object *i.*

Bahn et al. did not describe a mechanism for determining λ, simply mentioning training as an approach without providing implementation details [5], while Katsaros and Manolopoulos suggested trial-and-error [9]. An obvious

drawback to either methodology is that the resulting parameter may not provide efficient object replacement as request patterns change. Interestingly, a value of zero for λ causes the algorithm to behave very similar to LFU, while behavior similar to LRU results when λ is set to one.

E. Hybrid Algorithm

Wooster et al., presented a cache policy that utilizes several objects and request traffic statistics in a functional computation that derives a cost (or value) for each member or potential member of the cache [10]. The Hybrid formula is defined as:

$$V_i = (rtt_s + W_b / b_s) \cdot (f_i)^{W_n / s_i}$$

Where: s_i= size of object *i;* f_i= total number of object requests for object *i;* rtt_s= round trip time from cache to server *s;* b_s= bandwidth from cache to server *s;* W_b and W_n= tuned parameters.

Balamash and Krunz noted that W_b is tuned based on the "importance of the connection time relative to the connection bandwidth [3]", while W_n is tuned based on the "importance of frequency information relative to the size of the object [3]." These parameters are static; though they can be tuned for each implementation, they do not change over time within the context of the implementation. Similar to LUV, the authors do not provide a methodology for determining the static parameters W_b and W_n other than experimentation utilizing trace files, which can result in less efficient cache performance.

F. Mix Algorithm

Niclausse et al. presented an extension to the Hybrid cache policy that adds a parameter of time since last access to the functional cost computation [7]. The Mix formula is defined as:

$$V_i = \frac{d_i^{r_1} \cdot f_i^{r_2}}{tref_i^{r_3} \cdot s_i^{r_4}}$$

Where: s_i= size of object *i;* f_i= total number of object requests for object *i;* d_i = average server retrieval delay for object *i;* $tref_i$= current date and time of last request for object *i;* r_1, r_2, r_3, r_4= tuned parameters.

The authors noted that tuning the parameters utilized by the algorithm is not a trivial task and did not present a methodology for doing so. However, their trace simulation experiments found that using a value for r_1 that was much smaller than r_2, r_3, and r_4 gave optimal performance, and their published results used 1, 0.1, 0.1, and 0.1 for the respective parameters. Though the algorithm utilizes several performance characteristics as well as tuned parameters in an attempt to create an efficient replacement algorithm, experimentation by Balamash found that the algorithm was one of the worst performers, generally bested by even LRU and being superior only to Size.

G. Greedy Dual Size with Frequency (GDSF) Algorithm

Jin and Bestavros proposed the GDSF algorithm, which is a functional algorithm similar in approach to Mix in that it utilizes several object statistics, but different in that it utilizes only one pre-tuned parameter [11]. The GDSF formula is defined as:

$$V_i = f_i \cdot c_i / s_i + L$$

Where: s_i = size of object *i;* f_i = total number of requests for object *i;* c_i= the cost of bringing object *i* into the cache; *L*= a runtime factor which starts at zero when the cache is initialized and represents the value of the most recent object to be replaced.

Arlitt et al. noted that the best HR is achieved when c_i is set to one [12]. The runtime factor *L* works as follows: when the cache is first initialized, *L* is set to zero. *L* remains zero until the cache becomes full and an object needs to be removed from the cache. The algorithm determines the object with the lowest value, V_i, and removes it from the cache. The runtime parameter *L* is set to the value of V_i for the ejected object. This process continues throughout the life of the cache such that *L* is an ever-increasing parameter.

Shi and Zhang, attempting to find single optimal algorithms for HR, BHR, and LR separately, instead found that GDSF performed best for all three metrics simultaneously when compared to LRU, LFU, and GDS [13].

H. Lowest Relative Value (LRV) Algorithm

After extensive analysis of web traces, Rizzo and Vicisano proposed a complex (and, according to Bahn et al., "difficult" to implement) algorithm called Lowest Relative Value [14][3]. The algorithm utilizes separate computations based on whether the object is being requested for the first time or a subsequent time:

$$V_i = \begin{cases} P_1(s_i) \cdot [1 - D(t)] \cdot c_i / s_i, n = 1 \\ P_n \cdot [1 - D(t)] \cdot c_i / s_i, otherwise \end{cases}$$

Where: s_i= size of object *i;* c_i= the cost of loading object *i* into the cache; *t*= time since last request for object *i;* P_n= probability of access of n+1 given access of n times; *D(t)*= cumulative distribution function of object inter-access time.

The probability functions P_n and *D(t)* are based on "extensive analysis of trace data [5]," but are computed dynamically [14]. Rizzo provides an estimation for computing *D(t)* as: $D(t) \approx c \cdot \log(\frac{t+\tau_1}{\tau_1})$ where τ_1 is a parameter that "accounts for the periodicity of frequent references to popular documents [14]" and *c* is further defined by the equation: $c = D(t_b) \cdot \log(\frac{t_2+\tau_1}{\tau_1})$

While Bahn, et al. did not examine the effectiveness of LRV, their treatment of the algorithm noted that the V_i equation needed to be calculated for every object in the cache each time an object was removed. Additionally, the LRV authors stated that the probability functions were iterative in nature. While the algorithm is adaptive to traffic, then, it is not only difficult to implement but also computationally complex.

I. Advanced Replacement Cache (ARC) Algorithm

Megiddo and Modha constructed a dynamically adaptive cache policy that attempted to strike a balance between object request frequency and recency. Their policy implements two

LRU cache areas: one is a list of LRU-ordered objects that have been requested only once and the other is a list of LRU-ordered objects that have been requested two or more times [15]. They define their algorithm as

ARC(*c*) Initialize $T_1 = B_1 = T_2 = B_2 = 0$, $p = 0$.
x - requested page.

Case I. $x \in T_1 \cup T_2$ (a hit in ARC(c) and DBL($2c$)):
Move x to the top of T_2.

Case II. $x \in B_1$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \min\{c, p + \max\{|B_2|/|B_1|, 1\}\}$.
REPLACE(p).
Move x to the top of T_2 and place it in the cache.

Case III. $x \in B_2$ (a miss in ARC(c), a hit in DBL($2c$)):
Adapt $p = \max\{0, p - \max\{|B_1|/|B_2|, 1\}\}$.
REPLACE(p).
Move x to the top of T_2 and place it in the cache.

Case IV. $x \in L_1 \cup L_2$ (a miss in DBL($2c$) and ARC(c)):
case (i) $|L_1| = c$:
if $|T_1| < c$ then delete the LRU page of B_1.
REPLACE(p).
else delete LRU page of T_1 and remove it from the cache.
case (ii) $|L_1| < c$ and $|L_1| + |L_2| \geq c$:
if $|L_1| + |L_2| = 2c$ then delete the LRU page of B_2.
REPLACE(p).
Put x at the top of T_1 and place it in the cache.

Subroutine REPLACE(p)

if ($|T_1| \geq 1$) and (($x \in B_2$ and $|T_1| = p$) or ($|T_1| > p$)) then move the LRU page of T_1 to the top of B_1 and remove it from the cache.

else move the LRU page in T_2 to the top of B_2 and remove it from the cache.

In their implementation, T_1 and T_2 represent the list of most recently requested objects in lists L_1 and L_2, respectively. Similarly, B_1 and B_2 represent the list of least recently requested objects in lists L_1 and L_2, respectively. The parameter p dynamically adjusts such that the overall cache contains "the p most recent [objects] from L_1 and c-p most recent [objects] from L_2." [15]. The authors' experimental results show that the ARC algorithm does outperform LRU in their trials. Unfortunately, the experiments were performed using traffic traces specific to their research facility rather than publicly available traces, making direct comparisons to other published algorithms impossible.

III. Adaptive Replacement Policy

Memory-based caching being finite in nature is typically governed by a single cache-replacement policy. A simplified and generalized architecture for a cache area x governed by replacement policy R is depicted in figure 1.

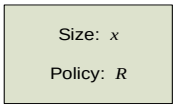

Fig. 1. Simple Cache Architecture

This article focuses on an adaptive cache replacement policy that allows for a short period of possibly less efficient cache performance while still providing some cache benefits, directly leading to an efficient algorithm choice without the need for trace file collection or offline training. The approach is similar to that of ARC, except that it is adaptable to replacement policy choices other than LRU and possibly extensible to more than two policies. The rest of this section outlines the new multiple-algorithm approach to dynamically choosing an efficient replacement policy.

The new approach modifies the simple cache architecture by splitting its finite area into n separate parts, each governed by a distinct replacement policy, R_i. In such an architecture, each partition would start with size x/n. Figure 2 depicts this generalized architecture.

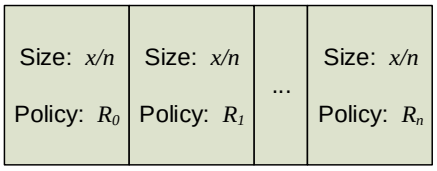

Fig. 2. Generalized n-Policy Cache Architecture

Considering a two-parts cache architecture, a second modification allows for the size of each area y and y' to adjust dynamically based on traffic patterns while their combined sizes are never greater than x, as illustrated in figure 3. Caches are initialized to be completely empty, so a cache system has no history on which to base the loading or removal of objects. The multi-algorithm approach *primes* each cache area with objects as they are being requested. The priming process alternates loading objects between the two caches, y and y', until both areas are full. Additionally, a cached object can exist in only one area at a time, not both simultaneously. During the load process, the cache policies continue to work normally – if a cached object is re-requested, it is provided by the cache area from which it resides. If one area becomes full during the priming phase, the other area continues to be filled until it, too, is completely primed.

Once primed, a secondary algorithm begins to dynamically adjust the size of the caches based on which area is serving the most cache hits. This is accomplished using a dynamic parameter ρ that is initially set to $x/2$ and is used to calculate the current size of each cache area, mathematically:

$$\|y\| = \rho \qquad \|y'\| = x - \rho$$

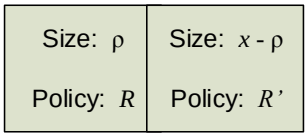

Fig. 3. Two-Policy Cache Architecture with Parameter

When area *y* experiences a cache hit, ρ is increased by the size of the object being requested. Alternatively, when area *y'* sees a hit, ρ is decreased by the requested object size. If the total size of all objects cached for an area exceeds the dynamic size, cache replacement policies kick in and objects are removed until the total object size is less than or equal to the dynamic allocation.

The initial theory behind the new multi-algorithm replacement policy hypothesized that a cache area *x* would adapt to changing traffic patterns in a manner similar to that of ARC. Instead, another interesting result emerged from the simulation results: the overall area *x* generally converges to either replacement policy *R* or *R'*, depending on which policy yields best results for the traffic pattern. The multi-algorithm policy is outlined as follows:

```
Given:
      L1 is list using LRU policy
      L2 is list using LFU policy
      x is total cache size
      0 < |L1| ≤ ρ  and  ρ < |L2| ≤ x
LFRU3 (x) Initialize L1 = L2 = 0, ρ = x/2.
          o - requested object.
Case I. o ∈ L1 (a miss in LFU, a hit in LRU):
          If  L1  is  primed,  L2  is  primed,  ρ>
              sizeof(o), and ρ +sizeof(o)< x
              Adapt ρ = ρ + sizeof(o).
              Move o to the top of L1.
          If |L2| > x - ρ, evict LFU objects until
              |L2| ≤ x - ρ and mark L2 as primed.
Case II. o ∈ L2 (a miss in LRU, a hit in LFU):
          If L1 is primed, L2 is primed, x - ρ >
              sizeof(o), and ρ - sizeof(o) > 0
              Adapt ρ = ρ - sizeof(o).
          Increase frequency of o in L2.
          If |L1| > ρ, evict LRU objects until |L1|
              ≤ ρ and mark L1 as primed.
Case III. o ∉ L1 ∪ L2 (a miss in LRU and LFU):
          If ((L1 is not primed and L1 has fewer
              entries than L2)
              or (L1 is not primed and L2 is primed)
              or (L1 is primed and L2 is primed and ρ
              < x/2))
              and ρ + sizeof(o) < x
              (LRU is doing better or needs primed)
              add o to the top of L1.
          Else
              If ((L2 is not primed and L2 has fewer
                  entries than L1)
                  or (L2 is not primed and L1 is
                  primed)
                  or (L1 is primed and L2 is primed))
                  and ρ - sizeof(o) > 0
                  (LFU is doing better or needs
                  primed)
                  add o to L2 with frequency of 1.
                  ρ remains unchanged.
              If |L1| > ρ, evict LRU objects until
                  |L1| ≤ ρ and mark L1 as primed.
              If |L2| > x - ρ, evict LRU objects
                  until |L2| ≤ x - ρ and mark L2 as
                  primed.
```

IV. Preliminary Results

An experiment was designed to assess the effectiveness of the adaptive policy using two algorithms where the cache was split into two parts. LRU algorithm was chosen for policy *R* and the LFU algorithm was chosen for policy *R'*. These policies were chosen due to their computational simplicity [16] and the fact that many if not most existing policies utilize recency and/or frequency in their design [3].

Two traces from the Internet Traffic Archive [17] were chosen to carry out the validation process: one from Digital Equipment Corporation (DEC) and another from the Environmental Protection Agency (EPA). The trace files were chosen primarily based on how they performed under simulation; DEC generally performed better using the LRU policy whereas EPA generally showed a preference for the LFU policy. Additionally, the DEC trace files were previously utilized in algorithm research by in [5], [16], [11], [3], and [14] while the EPA trace files were used in [18]. Finally, a third trace file was created to alternate requests from the DEC and EPA trace files in order to craft a synthetic trace file for analysis that mimicked two simultaneous unique traffic patterns.

These web traces were applied using a cache simulator written by Pei Cao, known as Uniform [19]. Employed in other research ([20], [21], and [22]) and written in C, this application readily facilitated the simulation of the LRU replacement policy. The simulator was enhanced by implementing the LFU policy as well as the adaptive policy, LFRU3, in order to simulate these policies along with the already available LRU. Additionally, each policy utilized a threshold mechanism that existed in the initial simulator and was implemented in newly added policies (LFU and LFRU3). The threshold process refused to cache objects above a certain size (250kB in this experiment) so that one large object could not 'pollute' the cache, meaning that a request for one large object would not cause many smaller and possibly more beneficial objects to be removed.

The performance of the three primed with thresholding versions of the three algorithms were compared to determine the effectiveness of LFRU3. Figures 4-6 depict the performance using the DEC, EPA, and synthetic DECEPA trace files, respectively. For the DEC trace the adaptive policy, LFRU3, has nearly the same performance as the best algorithm (LRU) for smaller cache sizes and very close in performance at the largest cache size, as shown in figure 4. This indicates that LFRU3 successfully converged to the better of the two algorithms in all test cases.

In the EPA simulations the new two-algorithm policy chose the better (LFU) algorithm in two scenarios – cache sizes of 0.05% and 10% of maximum (5 simulations). For these five simulations, then, LFRU3 successfully converged to the better of the two algorithms 40% of the time. It is encouraging to note that at the 0.05% cache size, where the LRU algorithm and LFU algorithm showed the most dramatic performance differential, the adaptive policy successfully converged to the better of the two algorithms, as shown in figure 5.

The synthetic simulation showed the most promise. In this simulation the new approach was actually superior to the other algorithms for smaller cache sizes. It delivered an 8% HR improvement over LFU and 48% improvement over LRU for the 0.05% cache size. At 0.5% cache size, the results were less pronounced but still superior: 1.4% HR improvement

over LFU and 9.6% better than LRU, as shown in figure 6. Figures 7 and 8 depict the HR for these two cache sizes over simulation time.

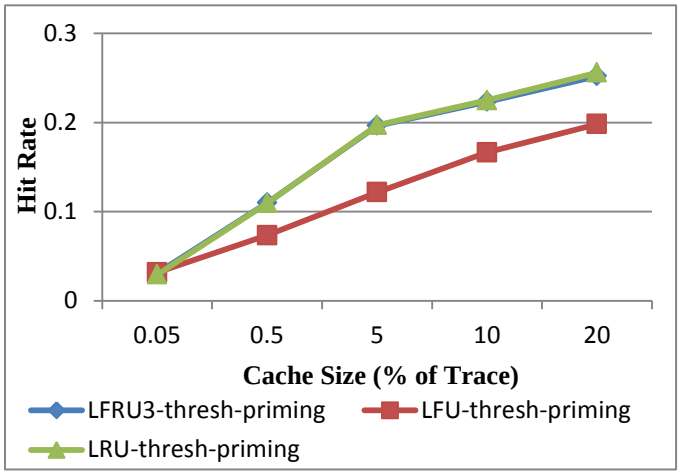

Fig. 4. HR vs. Cache Size (percentage of DEC Trace)

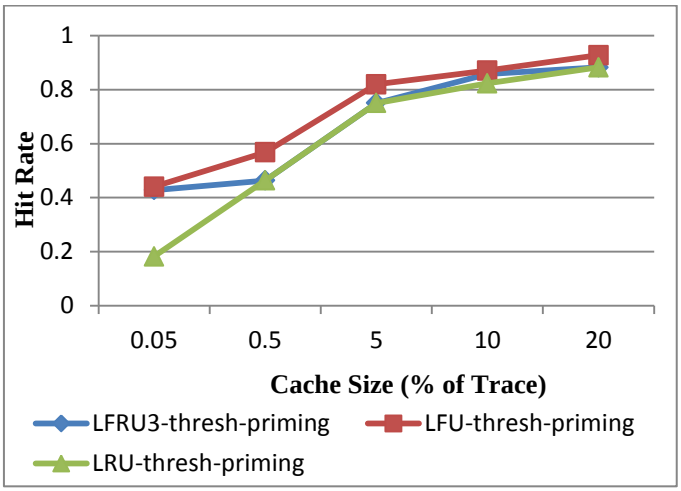

Fig. 5. HR vs. Cache Size (percentage of EPA Trace)

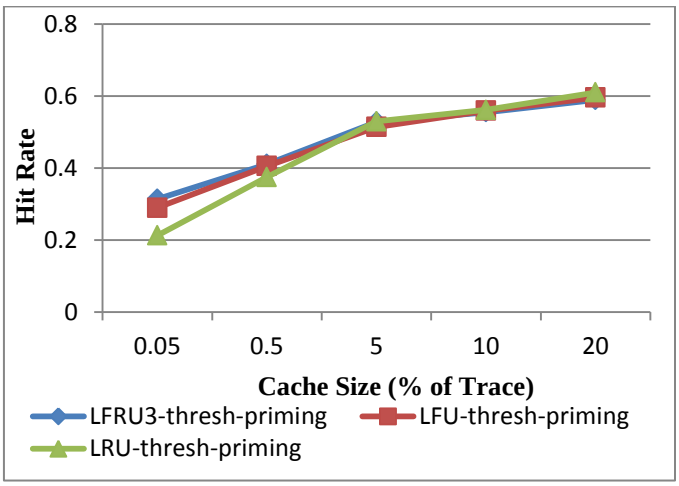

Fig. 6. HR vs. Cache Size (percentage of DECEPA)

Results in Figure 7 illustrate that for the smallest cache, LFRU3 performs consistently better than the other algorithms, and all three algorithms have consistent performance throughout the life of the simulation. Thus, the convergence to superior performance over the other algorithms occurs very early in the lifecycle of the cache and remains constant throughout, a very encouraging result.

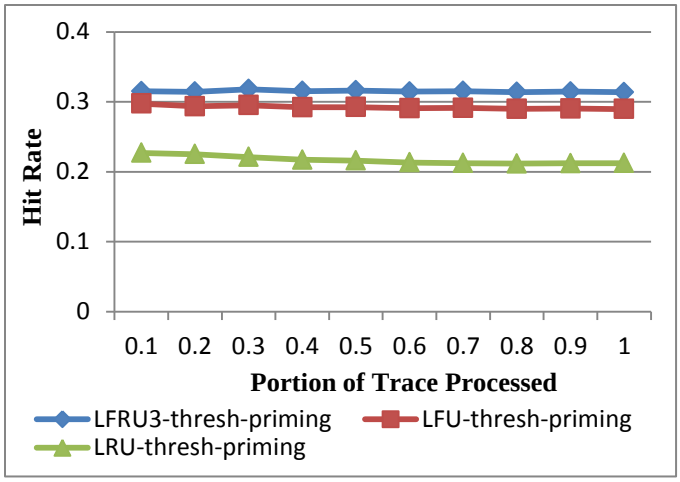

Fig. 7. HR vs. Sim. Time Size (0.05% Synthetic Trace)

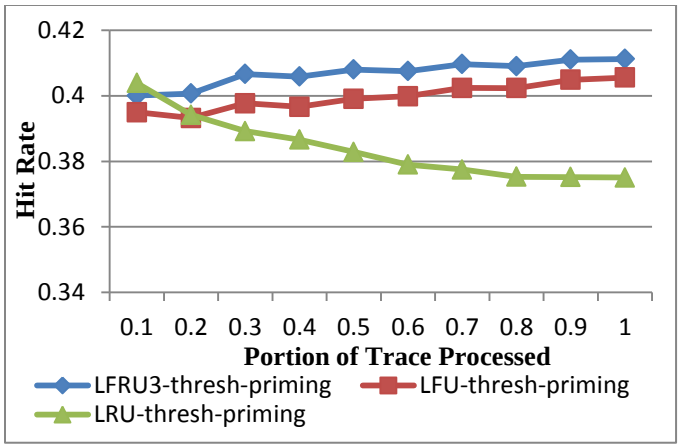

Fig. 8. HR vs. Sim. Time Size (0.5% Synthetic Trace)

The next largest cache shows LRU starting strong but quickly being surpassed by both LFU and the new approach, with LFRU3 consistently better than LFU throughout the examination period, as shown in figure 8. While in this case the new policy is not superior at the first examination period (10% of the trace), it quickly exceeds the performance of the initial best performer and shows gradual improvement and superior performance throughout the remainder of the cache lifecycle.

For Byte Hit Rate (BHR), the performance gain was not as pronounced. As Figure 9 illustrates, the LFRU3 policy slightly edges the next-best performer, LFU, at the smaller size, but performance at other sizes varies. Reduced latency (LR), a measure of the reduction of time spent waiting for objects, shows promise.

Figure 10 shows that LR is nearly identical to that for HR – LRFU3 is the best LR performer for the two smallest caches.

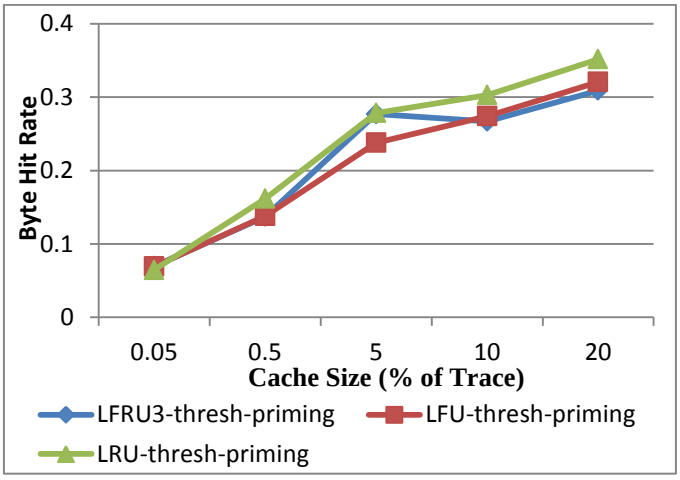

Fig. 9. BHR vs. Cache Size (percentage of Striped Trace)

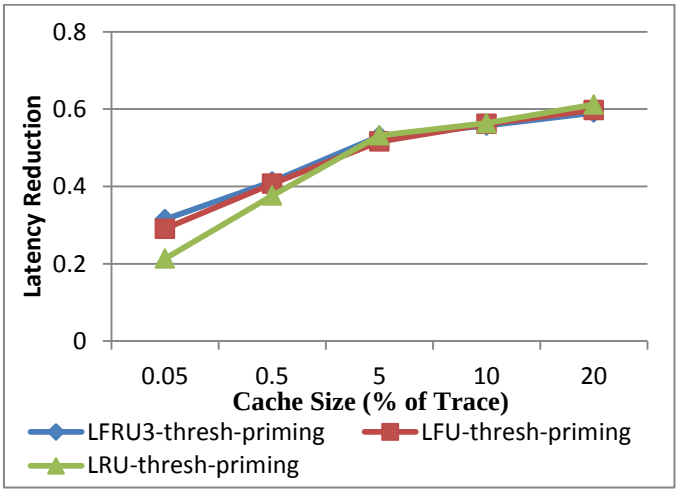

Fig. 10. LR vs. Cache Size (percentage of Striped Trace)

V. CONCLUSIONS

Caching for Web documents is a hugely beneficial function, and much research has been published on cache replacement policies. While the replacement algorithms themselves may factor in one, a few, or many parameters related to the objects and their request history, a multi-algorithm policy has not been attempted. While this article focuses on a two-policy architecture, it is expected that it can be effective for more than two policies.

Simulations of the multi-algorithm cache replacement policy shows that it is a viable approach that can adapt itself to the better of two replacement policies in many instances, and provide superior performance in some others. The empirical results support that the adaptive policy works particularly well in environments with limited cache sizes.

REFERENCES

[1] Smith, A. J., "Bibliography on Paging and Related Topics", Operating Systems Reviews, Vol. 12, 1978.

[2] Smith, A. J., "Second Bibliography for Cache Memories", Computer Architecture News, Vol. 19, No. 4, 1999.

[3] Balamash, A. and M. Krunz, "An Overview of Web Caching Replacement Algorithms", IEEE Communications Surveys, Vol. 6, No. 2, Second Quarter, 2004.

[4] Baeza-Yates, Ricardo, et al., "Design Trade-Offs for Search Engine Caching", ACM Transactions Web, Vol. 2, No. 4, October, 2008.

[5] Bahn, Hyokyung, et al., "Efficient Replacement of Nonuniform Objects in Web Caches", IEEE Computer, Vol. 35, No. 6, June, 2002.

[6] Chankhunthod, A., et al., "A Hierarchical Internet Object Cache", Technical Report 95-611, Computer Science Department, University of Southern California, Los Angeles, California, 1995.

[7] Niclausse, N., et al., "A New and Efficient Caching Policy for the World Wide Web", Proceedings of Workshop Internet Server Performance (WISP 98), 1998.

[8] Williams, S., "Removal Policies in Network Caches for World Wide Web Documents", Proceedings of ACM SIGCOMM Conference, Stanford University, Aug. 1996.

[9] Katsaros, D. and Y. Manolopoulos, "Caching in Web Memory Hierarchies", Proceedings of the 2004 ACM Symposium on Applied Computing, 2004.

[10] Wooster, R. and M. Abrams, "Proxy Caching that Estimates Page Load Delays", Proceedings of the 6th International World Wide Web Conference, Santa Clara, CA, Apr. 1997.

[11] Shudong and A. Bestavros, "Popularity-Aware GreedyDual-Size Web Proxy Caching Algorithms", Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS 2000), 2000.

[12] Arlitt, M., et al., "Evaluating Content Management Techniques for Web Proxy Caches", SIGMETRICS Performance Evaluation Review, Vol. 27, No. 4, March, 2000.

[13] Shi, Lei and Y. Zhang, "Optimal Model of Web Caching", Conference Record of the 2008 Fourth International Conference on Natural Computation, 2008.

[14] Rizzo, L. and L. Vicisano, "Replacement Policies for a Proxy Cache", IEEE/ACM Transactions on Networking, Vol. 8, No. 2, 2000.

[15] Megiddo, N. and D. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm", IEEE Computer, Vol. 37, No. 4, April, 2004.

[16] Bahn, Hyokyung, "Web Cache Management Based on the Expected Cost of Web Objects", Information and Software Technology, Vol. 47, April, 2005.

[17] The Internet Traffic Archive, http://ita.ee.lbl.gov/html/traces.html, last updated April 9, 2008.

[18] Cheng, A. M. K. and Z. Zhang, "Improving Web Server Performance with Adaptive Proxy Caching in Soft Real-time Mobile Applications", Journal of VLSI Signal Processing, Vol. 47, 2007.

[19] Cao, Pei, Uniform Cache Simulator, download from ftp://ftp.cs.wisc.edu/pub/cao/webcache-simulator.tar.z.

[20] Cao, P. and S. Irani, "Cost Aware WWW Proxy Caching Algorithms", Proceedings of the 1997 UXENIX Symposium on Internet Technology and Systems, 1997.

[21] Shi, Lei, et al., "An Applicative Study of Zipf's Law on Web Cache", International Journal of Information Technology, Vol. 12, No. 4, 2006.

[22] Buijtendijk, V., "Module Deployment and Management within the Grid-based Virtual Laboratory for e-Science", Thesis Paper, Universiteit van Amsterdam, May, 2005.