Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
88(5) 507–541
© The Author(s) 2012
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/0037549711414152
sim.sagepub.com

A quantitative study of Web cache replacement strategies using simulation

Sam Romano and Hala ElAarag

Abstract

The Web has become the most important source of information and communication for the world. Proxy servers are used to cache objects with the goals of decreasing network traffic, reducing user perceived lag and loads on origin servers. In this paper, we focus on the cache replacement problem with respect to proxy servers. Despite the fact that some Web 2.0 applications have dynamic objects, most of the Web traffic has static content with file types such as cascading style sheets, javascript files, images, etc. The cache replacement strategies implemented in Squid, a widely used proxy cache software, are no longer considered 'good enough' today. Squid's default strategy is Least Recently Used. While this is a simple approach, it does not necessarily achieve the targeted goals. We simulate 27 proxy cache replacement strategies and analyze them against several important performance measures. Hit rate and byte hit rate are the most commonly used performance metrics in the literature. Hit rate is an indication of user perceived lag, while byte hit rate is an indication of the amount of network traffic. We also introduce a new performance metric, the object removal rate, which is an indication of CPU usage and disk access at the proxy server. This metric is particularly important for busy cache servers or servers with lower processing power. Our study provides valuable insights for both industry and academia. They are especially important for Web proxy cache system administrators; particularly in wireless ad-hoc networks as the caches on mobile devices are relatively small.

Keywords

cache replacement, proxy cache, simulation, Web

1. Introduction

With the debut of Web 2.0, many researchers were interested in studying their workload characteristics in order to design more efficient servers. YouTube is an example of a very popular Web 2.0 site and hence it was the choice for many to conduct such research.[1,2] Gill et al.[1] conducted an extensive analysis of YouTube workload and observed 25 million YouTube transactions over a 3-month period that included the downloads of 600,000 videos. They compared traditional Web workload characteristics with that of YouTube and found many similarities. They concluded that traffic characterization of YouTube suggests that caching popular video files on Web proxy servers reduces network traffic and increases the scalability of YouTube servers.[1]

There are several decisions that must be made, such as cache placement and replacement. Many proposals are covered in the literature regarding these Web cache decision processes. Our main focus will be the cache replacement problem, the process of evicting objects in the cache to make room for new objects, applied to Web proxy servers.

There are many replacement strategies to consider when designing a proxy server. The most commonly known cache replacement strategies are Least Frequently Used (LFU) and Least Recently Used (LRU). Research in this area has been active for at least 10 years. Podlipnig and Boszormenyi[3] provided a survey of Web cache replacement strategies. They have

Department of Mathematics and Computer Science, Stetson University, Deland, FL, USA.

Corresponding author:
Hala ElAarag, Department of Mathematics and Computer Science, Stetson University, 421 North Woodland Boulevard, Deland, FL 33327, USA
Email: helaarag@stetson.edu

done well to not only list well-known strategies, but also categorize the strategies into five groups.

Although both a survey and classification of these strategies are available, there is not a known record of results comparing the majority of the cache replacement algorithms. Balamash and Krunz[4] compared cache replacement policies qualitatively rather than quantitatively. Many of the works consulted for this paper presented results for different strategies against, at most, three to five other strategies. However, their results cannot be compared effectively because most of the literature devised their experiments with differences in their implementations, such as the use of auxiliary caching, representation of Web object characteristics, etc. There is also the difference in trace files between experiments, and a large range of different sizes used for the cache space. Lastly, different proposals used different criteria on when to cache an object such as ignoring *PHP* files, *cgi-bin* scripts, POST requests, or simply just using all requests presented in a particular trace file.

Wong[5] believes that Web caching is a crucial part of today's Internet and there are enough good replacement strategies for general proxy servers. Any new technique will only provide a very small improvement over existing techniques and hence there is no need to devise new policies for traditional environments.

In spite of the many cache replacement policies proposed in the past 10 years, the most popular Web proxy software, Squid, uses LRU as its default strategy. We believe that this is because there has not been a comprehensive study presented that compares these strategies quantitatively. As Wong[5] mentions 'all policies were purported to perform better than others, creating confusion as to which policy should be used'. Many of the new policies proposed divide the cache into parts where one part of the cache handles cache replacement using traditional methods.[6]

Our research studies cache replacement strategies designed for static Web content, as opposed to dynamic Web content seen in some Web 2.0 applications. Most caching that occurs with dynamic content occurs on the browser side and does not occur from the standpoint of proxy servers. As a result, these strategies have not been considered. We have researched how proxy servers can still improve performance by caching static Web content such as cascading style sheets, javascript source files, and most importantly larger files such as images (jpeg, gif, etc.).

Our work is particularly important in wireless adhoc networks. In such networks, mobile devices act as proxy servers for a group of other mobile devices. However, they have limited storage space. The extensive research on cache replacement policies we provide in this paper is crucial for such environments with small cache sizes and limited battery life

In this paper, we extend our previous work[7] and present *27* cache replacement strategies we simulated against different performance metrics. To the best of the authors' knowledge, this is the most comprehensive quantitative analysis of Web cache replacement strategies.

The rest of the paper is structured as follows. In Section 2, we define several terms used throughout the paper and their use in the simulations, and present some of the related work. We describe the cache replacement categorization, rationale and example policies in Section 3. Section 4 discusses our simulation and certain data structures used for the algorithms. We define the metrics used to measure the performance of each strategy, and also propose a new metric in Section 5. Section 6 presents the results and observations of our simulation. We present the conclusions in Section 7. Lastly, we suggest several areas of future research in Section 8.

2. Background Information and related work

It is necessary to have a clear and precise definition of when a Web object is allowed to be cached. A thorough, well-defined definition allows Web cache users to understand what requests they make could potentially be cached and, in addition, is necessary for system administrators as a good tool in improving Quality of Service (QoS) for end users. For this paper, a *Web object* is a term used for all possible objects (HTML pages, images, videos, etc.) transferred through the HTTP protocol that can be stored in a proxy cache.[3]

2.1. Web request

Web request is a reference made through the HTTP protocol to a Web object, primarily referenced by their Uniform Resource Locator (URL). Requests are also identified by the size of the requested Web object from the origin server (at the time the request was made), a HTTP return code, and the time the proxy received the request. We define a *cacheable request* to have the following criteria:

- There must be a defined size, in bytes, for the request, and that size must be less than the total size of the cache and greater than zero.
- The request must be a GET request and its status code must be one of the following, as set by the HTTP 1.1 protocol:[1] 200, 203, 206, 300, 301 or 410. Table 1 shows the status codes and their meanings.

Separate from *cacheable request*, we also ignore any requests with URLs containing '/cgi-bin/' as well as any

Table 1. HTTP status code meanings

Code	Meaning
200	Ok
203	Non-authoritative information
206	Partial content
300	Multiple choices
301	Moved permanently
410	Gone (synonymous with deleted)

URLs that are queries (those that contain a question mark in their URL after the last '/').

Once a request is known to be *cacheable* and is received by the proxy, several things will occur in a sequential order. In a basic proxy server model, if a *cache hit* occurs, then the object being referenced is in the cache and the data can be copied and sent to the client. On a *cache miss*, when no object in the cache matches the request, the Web object will be retrieved from the origin server and the *cache placement strategy* decides whether the object will be placed into the cache. If there is not enough room for the new object to be added, then the *cache replacement strategy* is invoked. However, in order to understand how these strategies work, we define several aspects of objects these strategies will consider.

2.2. Characteristics of Web objects

Web objects are identified by several different characteristics. Each replacement strategy requires usually a small sub-set of the characteristics; however, all Web objects must be identified by their URL, since this is the only unique factor. The most important characteristics of Web objects are as follows:[3]

- Recency: information relating to the time the object was last requested.
- Frequency counter: number of requests to the object.
- Size: the size, in bytes, of the Web object.
- Cost: the 'cost' incurred for fetching the object from the origin server. Also known as the miss penalty. This will be covered in detail later in Section 5.2.
- Request value: the benefit gained for storing the object in the cache. This is generally a heuristic based on other characteristics, since an actual request value of an object cannot be determined without a priori information.
- Expiration time: Also generally a heuristic, either defined by the proxy or by the origin server of when the object will become stale and should be removed or refreshed in the cache. Also known as the time-to-live (TTL).

Most strategies use a combination of these characteristics to make their replacement decisions. The expiration time is the only characteristic mentioned that was not utilized in our simulation and is primarily referenced when dealing with the problem of *cache consistency*, which is out of the scope of this research. The request value is an abstract characteristic, primarily used by *Function-based* strategies, and defined by a characteristic function that pursues a total, well-defined ordering of the objects. (Essentially, any characteristic could be the request value, if the algorithm makes its decision based on one variable that has a total, well-defined ordering.)

2.3. Related work

Many researchers are interested in Web caching and its associated problems.[9–13] Oritz et al.[14] exploited the large infrastructure of today's Web proxy caching systems to interactively transmit JPEG2000 images, while Liu and Xu[15] used them for media streaming. Luo et al.[16] focused on making proxy caching work for database-backed Web sites. Many proposals can be found in the literature regarding the Web cache decision processes. Houtzager et al.[17] proposed an evolutionary approach to find an optimal solution to the Web proxy cache placement problem, while Aguilar and Leis,[18] for example, addressed the replacement problem. Fagni et al.[6] proposed a static dynamic cache. They store the most popular Web queries in a static, read-only portion of a cache. The remaining cache entries are dynamic and store other queries that cannot be satisfied using the static cache. The dynamic cache is managed by *any replacement policy*. In 2009, Kaya et al.[19] devised an admission-control policy to screen documents based on a mathematical expression that is function of average delay per request. They use this policy to identify cacheable and non-cacheable documents then use LRU for cache replacement. Goyal et al.[20] developed Vcache: a client-side caching technique for dynamic objects. Kim et al.[21] used Squid cache server and studied the effect of large file transfer in network environments with high bandwidths. They proposed a P2P Web caching technique that solves the cache bandwidth problem.

Many cache replacement policies use artificial intelligence techniques for decision making. For example, Cobb and ElAarag[22] and Romano and ElAarag[23] use neural networks, while Sabegi et al.[24] and Calzarossa and Valli[25] use fuzzy logic for improving cache replacement decisions. A survey on applications of neural networks and evolutionary techniques in Web caching was presented by Venketesh et al.[26]

With the widespread use of mobile devices, wireless environments are specialized environments where cache replacement policies are used to improve performance. To mention a few woks in this area, Katsaros and Manolopoulos[27] propose a policy to not only reduce latency but conserve scarce network resources in mobile wireless networks. Kumar et al.[28] implement a prototype that utilizes multi-layered caching techniques to improve performance of mobile devices.

Abhari and co-workers[29,30] improved the performance of the Apache server, the most popular open source proxy cache. They used an effective prefetching technique that proved to reduce page latency. The technique prefetches images and multimedia components embedded in Web pages from the disk to the memory of the proxy cache. They used trace files provided by IRcache[31] and showed that their prefetching technique has a better cache hit ratio by 9.35% and better byte hit ratio by 2.93% over a period of 1 day access to Web objects. Abhari and Soraya[2] developed a workload generator to study the workload characteristics of YouTube that generated workload with the same characteristics as real YouTube videos. They proved that the distribution model of YouTube workload follows a Zipf-like behavior. After simulating several scenarios with variable workload parameters, similar to Gill et al.,[1] they proved that caching the most popular videos can reduce network traffic and increase scalability of YouTube Web site.

Content Distribution Networks (CDNs) have the same objectives as Web proxy caches. Their goal is to provide short access time and consume less network bandwidth.[32] However, they differ in design. In an attempt to improve the service quality, the main idea behind CDNs is to distribute a selective set of the contents from the origin server to servers scattered over the Internet and serves a request from a server closer to the client. That is, the CDN servers are geographically scattered but located at the edge of the network. Figure 1 shows how a client accesses a copy of the data nearest to it, as opposed to all clients accessing the same central server. This architecture can be compared with the architecture of proxy cache servers shown in Figure 2. Su et al.[33] focused on the Akamai CDN,[34] the most popular CDN with 15,000 servers operating in 69 countries. They investigated techniques for locating and utilizing quality Internet paths without performing extensive path probing or monitoring. Badam et al.[35] developed HashCache which they believe suitable for CDNs. HashCache provides the infrastructure for caching applications. It has a wide range of configurations that makes it very flexible to suit systems with scarce resources as well as high-end systems. They have also built on top of HashCache a Web proxy cache called HashCache Proxy. Kangasharju et al.[36] developed four heuristics to the NP complete problem of replicating objects in such a way as to traverse the minimum number of autonomous systems when the clients fetch objects from the nearest server. They also provided insight into P2P content distribution. Doyle et al.[37] studied an important problem, which they called the trickle-down effect: the heavy tail Zipf-like distribution of traffic loads, when using proxy servers and CDNs. They used traces from IRcache[31] to illustrate this effect and further illustrate the implications of the trickle-down effect on back-end servers. Gadde et al.[38] derived hit ratios for multi-level caches then extended their model for CDNs, especially a scenario that is equivalent to a hierarchical proxy cache. Their results showed some limitations to the benefits of CDNs when the leaf populations are large. They also used real-world data from IRcache[31] to validate their model.

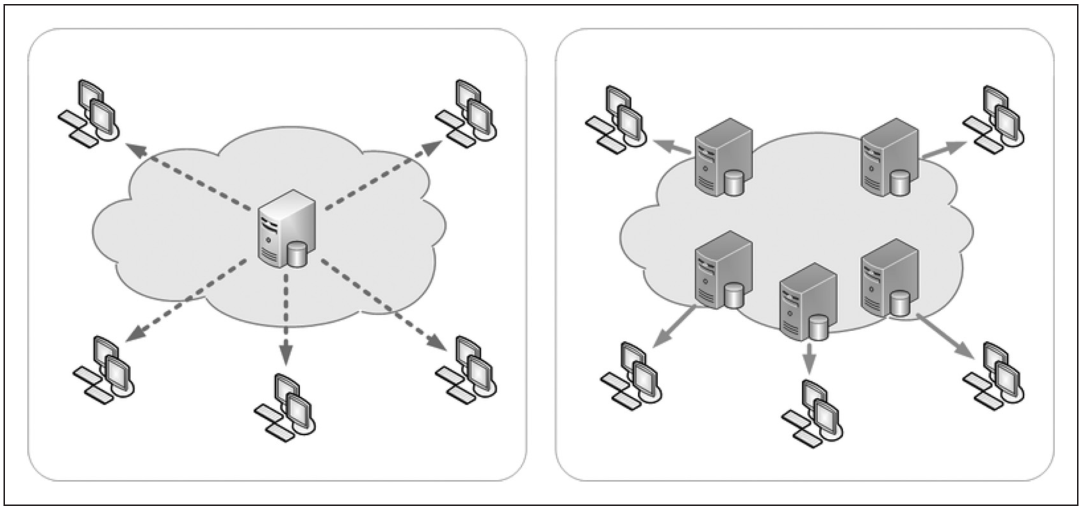

Figure 1. Content Distribution Network (CDN). Source http://cdn-comparison.com/.

3. Replacement strategies

In this research there is only one algorithm which makes its decision on two levels of characteristics; the rest decide primarily on one characteristic, or on a characteristic function (request value) which is a product of combined factors. Cache replacement strategies can be categorized into five groups.: frequency based, recency based, frequency/recency based, function based, and randomized.[3] Wong[5] also categorizes cache replacement strategies into five groups, but instead of recency/frequency category, a size category is suggested. We adopt the classification of Podlipnig and Boszormenyi[3] in this paper.

The first two groups, *recency* and *frequency*, are based mainly on LRU and LFU, respectively. *frequency/recency* strategies incorporate a mixture of an object's recency and frequency information together along with other characteristics to refine LRU and LFU. *Function-based* strategies have some defined method that accepts certain pre-defined parameters defining a request value to order the objects in the cache. The last group, *random*, essentially picks an object in a non-deterministic method. Owing to the inconsistent nature of the last category, we decided not to include any strategies that had a non-determinate replacement strategy. Table 2 presents the replacement categories, their rationale and some example of available replacement policies.

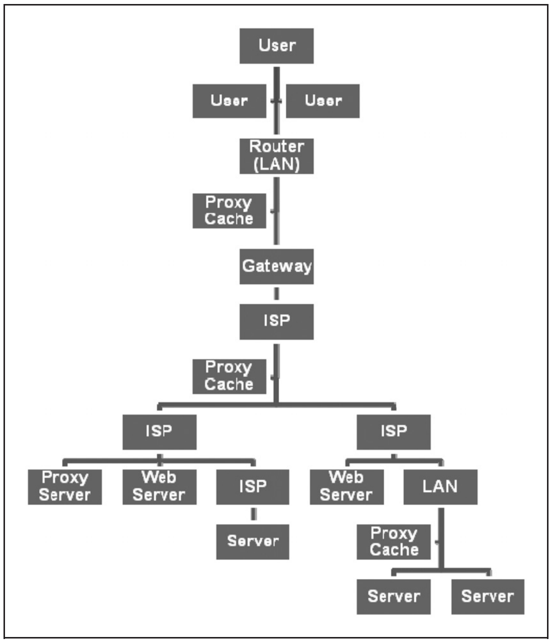

Figure 2. Possible locations for proxy cache servers in the Internet.

4. Simulation details

This section provides the details for our simulation and how some of the strategies were implemented. We also describe other Web cache strategies and components we used that were not part of our research, but integral to building a functional Web cache.

4.1. Web cache components

As discussed in Section 1, there are several decisions that must be made during the lifespan of a Web request. The first decision, covered in Section 2.1, is to decide whether the current request is cacheable. Once a request is known to be cacheable, it is searched among the objects in the cache to determine whether it already

Table 2. Cache replacement categories

Category	Rationale	Replacement policies
Recency based	Derived from a property known as *temporal locality*, the measure of how likely an object is to appear again in a request stream after being requested within a time span.[39]	LRU, LRU-Threshold,[40] Pitkow/Reckers strategy,[41] SIZE,[42] LOG2-SIZE,[3] LRU-Min,[40] Value-Aging,[43] HLRU[44] and Pyramidal Selection Scheme (PSS)[45]
Frequency based	Derived from an object's popularity where those that are most popular should be cached.[3,5]	LFU, LFU-Aging,[46] LFU-DA[46] and α – Aging,[43] and LFU*[47]
Frequency/ Recency based	Attempt to combine both *spatial* and *temporal locality* together maintaining their characteristics of the previous two classes.	Segmented LRU (SLRU),[48] Generational Replacement,[49] LRU*,[50] HYPER-G,[42] Cubic Selection Scheme (CSS),[51] and LRU-SP[52]
Function based	Use a general characteristic function to define a request value for objects.	Greedy Dual (GD)-Size,[53] GDSF,[54] GD*,[55] Taylor Series Prediction (TSP),[56] MIX,[57] M-Metric,[58] LNC-R-W3,[59] and LUV[60]

exists or not. If the object exists, it must be determined whether the data in the cache has passed its expiration time. If the expiration time is up or the object was not in the cache, the object is retrieved from the origin server, otherwise the Web object's information (recency, frequency counters, request value, etc.) is updated and the next request is served.

Once an object is retrieved from the origin server, a cache *placement* strategy is invoked, deciding whether the cache will accept the object or not before it is sent to the client. If not enough space exists in the cache the cache *replacement* strategy is invoked.

Most proxy caches also use two limits to their space available in the cache and make sure the cache can respond well in the case of sudden increases in the incoming number of requests. They set what are known as two watermarks. One is called a *low watermark*, and is often denoted by L; the other is known as the *high watermark* and is denoted as H. If the current space occupied by the cache exceeds H, then the replacement strategy, also known as the victim selection process, is invoked until the space is less than or equal to L. Typically, L is set to 60–75% of the cache's total space, and H set to 90–95% of the total cache space.

However, in the case of our simulation we ignored the watermark process, to see how much strain the Web cache could take as well as how well the replacement strategies worked when invoked under necessary conditions (no space being left) as opposed to being invoked prior to the cache being full.

In order to create these conditions, we also had to ignore the expiration process. Typically, either the object's time to live (TTL; or amount of time till the object is considered 'expired') is used, or a heuristic is generated based on the file's size, URL, etc. This was not the focus of our research, and since the trace files we utilized provided no information on the TTL of the object at the time the traces were recorded, it was out of our scope of this research to investigate known expiration policies. Last, but not least, our cache placement was a simple strategy. All objects whose requests were deemed *cacheable* were immediately intended to be cached.

Some of the literature makes a distinction in its cache placement strategy, admitting only those objects which will add an overall *benefit* to the cache so in the case that the replacement strategy must be invoked the *benefit* of the object being added must outweigh the *benefit* of those objects being removed from the cache.

In essence, this treats the *caching problem* as the famous *knapsack problem*, which is an excellent comparison. However, most definitions are ambiguous as to what the benefit is measured by. Also, the benefit should be dynamic and decreasing due to objects' expiration times. If expiration time is not a factor, then the benefit would need to consider recency information. However, in considering recency information, we also have to consider how recently the last request to the object being considered occurred, leading to more space overhead for objects not in our cache, which adds even more complexity to the entire process.

Without considering recency, expiration, or any factor involving durations of time, then the cache will suffer a similar pollution to what LFU tends to create, which is the caching of only those objects deemed to be beneficial overall. This is exactly the outcome one would expect in terms of the *knapsack* problem; but in terms of the *caching* problem, this is what we wish to avoid. Thus, we decided against considering more complex placement procedures due to there being no previous method available to calculate *benefit*, and to focus on the replacement process.

4.2. Simulation architecture

Our simulator was built as a discrete-event simulator where each request was treated as a discrete event. The simulation clock was advanced forward by the amount of time that passed from request to request so that the time in the simulation was artificially equivalent to the time that the request in the trace took place at. Trace files were parsed and cleansed prior to the simulation. We identified requests referring to the same URI and gave all unique URI's a specific unique identification number. Figure 3 illustrates the request life cycle and how simulation statistics are recorded.

4.3. Details of implemented strategies

Table 3 contains commonly used variables and their descriptions. If a strategy uses a variable not defined in the table, then it will be defined in its corresponding implementation detail. Any use of the logarithmic function, denoted as log, is assumed to be of base 2, unless otherwise specified.

Note that some strategies require the use of an auxiliary cache. An auxiliary cache holds information about all Web objects that have been seen by the proxy cache. Without this stored information, particular strategies tend to produce less-than-optimal results as explained later.

4.3.1. Implementation of recency-based strategies.

This set of strategies uses recency information as a significant part of its victim selection process. Recency-based strategies are typically straightforward to implement taking advantage of queues and linked lists.

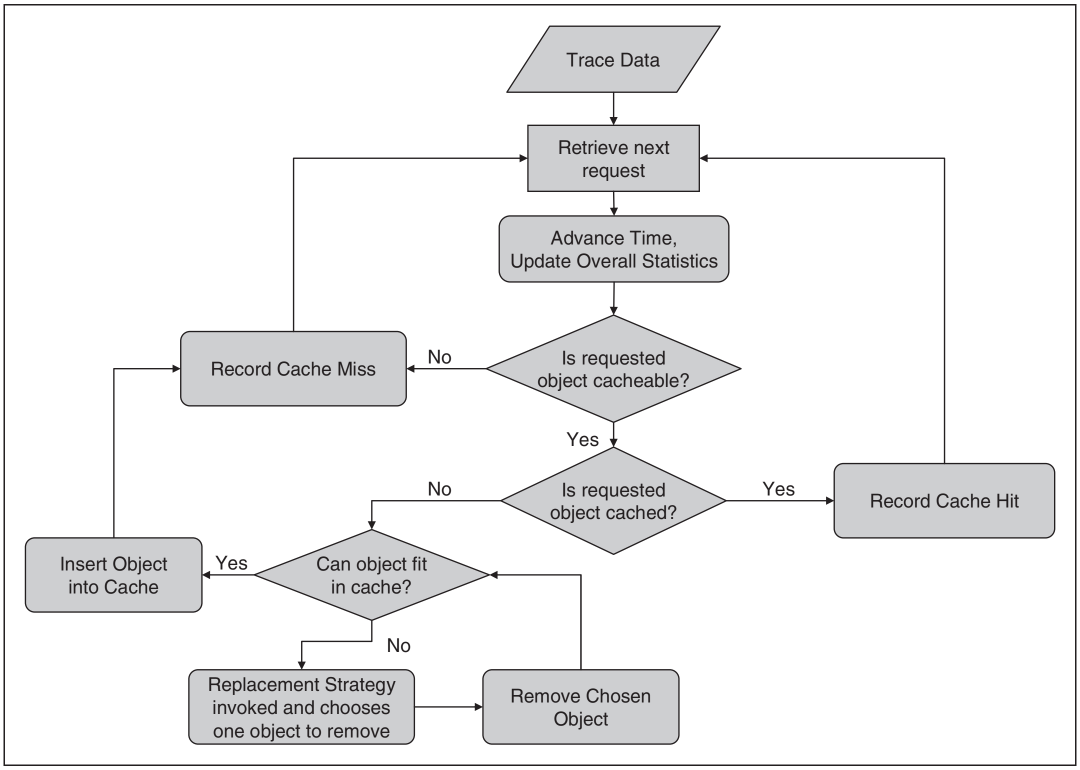
Trace Data
Retrieve next request
Advance Time, Update Overall Statistics
Is requested object cacheable?
No
Record Cache Miss
Yes
Is requested object cached?
No
Yes
Record Cache Hit
Can object fit in cache?
Yes
Insert Object into Cache
No
Replacement Strategy invoked and chooses one object to remove
Remove Chosen Object

Figure 3. Simulated request lifecycle.

Table 3. Characteristic symbols

Variable	Description
S_i	Size of an object *i*
T_i	Time object *i* was last requested
ΔT_i	Time since object *i* was last request
F_i	Frequency counter of object *i*
ΔF_i	Number of references to occur since last time object *i* was referenced
l_i	Access latency for object *i*
C_i	Cost of object *i*
R_i	Request value of object *i*
M	Size of the cache

1. **LRU**: One of the most commonly used strategies in many areas of data management. This algorithm removes the least recently used (referenced) object, or in other terms the object with the largest ΔT_i. A simple linked list (queue) allows this algorithm to be efficiently implemented and is referred as a LRU list.
2. **LRU-Threshold**:[40] Just like LRU, except an object is not permitted into the cache if its size, S_i, exceeds a given threshold.
3. **Pitkow/Reckers strategy**:[41] Objects that are most recently referenced within the same day are differentiated by size, choosing the largest first. Object references not in the same period are sorted by LRU. This strategy can be extended by varying the period of which objects are differentiated by their size (such as an hour, 2 days, 1 week, etc.).
4. **SIZE**:[42] Removes the largest object first. If objects are of the same size, then their tie is broken by LRU.
5. **LOG2-SIZE**:[3] Sorts objects by their *floor*[log(S_i)], differentiating objects with the same value by LRU. This strategy tends to invoke LRU more often between similar-sized objects as compared with SIZE.
6. **LRU-Min**:[40] This strategy attempts to remove as few documents as possible while using LRU. Let T be the current threshold, L_o be the least recently used object (tail of a LRU list), and L an object in the list. Then LRU-Min follows as in the following procedure:
   1. Set T to S_i of the object being admitted to the cache.
   2. Set L to L_o.
   3. If L is greater than or equal to T, then remove L. If it is not, set L to the next LRU object in the list and repeat this step again until there is enough space or the end of the list is reached.
   4. If the end of the list is reached, then divide T by *2* and repeat the process from step 2.

7. **Value-Aging**:[43] Defines a characteristic function based on the time of a new request to object i, and removes the smallest value, R_i. Letting C_t be the current time, R_i is initialized to

$$R_i = C_t * \sqrt{\frac{C_t}{2}} \tag{1}$$

At each request, R_i is updated to

$$R_i = C_t * \sqrt{\frac{C_t - T_i}{2}} \tag{2}$$

8. **HLRU**:[44] Standing for History LRU, this uses a sliding window of h request times for objects. This algorithm is fairly complicated compared with most algorithms in this set, requiring additional information to be held for each object, even after the object has been removed from the cache. It is entirely possible to implement the algorithm without an auxiliary cache, but we found the results to be far less than optimal. The author of the algorithm also defined the algorithm with an auxiliary cache in mind. The *hist* value is defined for an object x with n indexed request times, t, where t_i is equivalent to the ith request time of object x.

$$hist(x, h) = \begin{cases} t_{n-h} & n \geq h \\ 0 & n < h \end{cases} \tag{3}$$

HLRU chooses the object with the maximum *hist* value. If multiple objects have *hist* values of zero, then they are sorted based on LRU.

9. **Pyramidal Selection Scheme (PSS)**:[45] This classification makes what is known as a 'pyramidal' hierarchy of classes based on their size. Objects of a class j, have sizes ranging from 2^{j-1} to $2^j - 1$. Inversely, an object i belongs to the class $j = floor[\log(S_i)]$. There are $N = ceil\ [\log(M+1)]$ classes. Each class is managed by its own LRU list. To select the next victim during the replacement process, the recently used objects of each class are compared based on their value, $S_i{*}\ \Delta F_i$. In Section 4.3.5, we demonstrate an efficient way to calculate ΔF_i in $O(1)$ time without keeping record of the request stream.

4.3.2. Implementation of frequency-based strategies.

Obviously, this category is tied strongly to the frequency/access counts of Web objects. Unlike recency-based strategies, these simple algorithms require complex data structures, such as binary heaps (also known as priority queues) to help decrease the time overhead in making their decisions. Some strategies, such as SIZE and HLRU, also take advantage of binary heaps. However, we considered this to be a detail oddity compared to most recency-based strategies.

Most of these algorithms are an extension of the common algorithm LFU. There are two ways to implement these algorithms, one requiring use of an auxiliary cache, and the other not. Comparatively, most recency-based strategies only need to keep track of the most *recent* values seen by the proxy cache, simplifying the record of a Web object's data to the time it is in the cache even if it is removed and added repeatedly. However, frequency counts do not pertain only to the lifespan of a particular object in the cache, but can also be persistent across multiple lifetimes of the object. The persistent recording of data for an object's frequency counts is known as *perfect LFU*, which inevitably requires more space overhead. The tracking of data while the object is only in the cache is known as *in-cache LFU*.

In either *perfect* or *in-cache* implementations, the cache can suffer *cache pollution*, objects with high-frequency counts that persist in the cache despite no longer being popular. Objects accumulating high popularity in many bursts over long periods of time create a problem with *perfect LFU*. However, *in-cache* suffers from missing the objects that slowly accumulate popularity over a long period, caching only those that happen to accumulate high popularity in the short run. There are flaws with both implementations; some in this section and the next will seek to break those down.

Since there is space overhead with *perfect LFU*, we will assume the *in-cache* variants of these algorithms.

10. **LFU:** The base algorithm of this class, removes the least-frequently used object (or object with the smallest frequency counter).
11. **LFU-Aging**:[46] This strategy attempts to remove the problem of cache pollution due to objects that become popular in a short time period. To avoid it, this strategy introduces an aging factor. When the average of all of the frequency counters in the cache exceeds a given average frequency threshold, then all frequency counts are divided by 2 (with a minimum of 1 for F_i). There is also a maximum threshold set that no frequency counter is allowed to exceed.
12. **LFU-DA**:[46] Since the performance of LFU-Aging requires the right threshold and maximum frequency, LFU-DA tries to remove this problem. Upon a request to object i, its value, K_i, is calculated as

$$K_i = F_i + L \tag{4}$$

where L is a dynamic aging factor. Initially L is set to zero, but upon the removal of an object i, L is set to $K_{i.}$ This strategy removes the object with the smallest K_i value.

13. **α–Aging**:[43] This is a periodic aging method that can use varying periods and a range, [0, 1], for its aging factor, α. Each object in this strategy uses a value, K, which is incremented by 1 each cache hit, much like a frequency counter. At the end of each period, an aging factor is applied to each object:

$$K_{new} = \alpha * K, \quad 0 \leq \alpha \leq 1 \tag{5}$$

Changing α from 0 to 1, one can obtain a spectrum of algorithms ranging from LRU ($\alpha = 0$) to LFU ($\alpha = 1$). Of course this is only true if LRU is used as a tie-breaker.[3]

4.3.3. Implementation of frequency/recency-based strategies.

These strategies tend to be fairly complex in their structure and procedures.

14. **Segmented LRU (SLRU)**:[48] This strategy partitions the cache into a two-tier system. One segment is known as the *unprotected* segment, and the other as the *protected* segment. The strategy requires space set aside for the protected segment. Objects that belong to this segment cannot be removed from the cache once added. Both segments are managed by the LRU replacement strategy. When an object is added to the cache, it is added to the unprotected segment, removing only objects from the unprotected space to make room for it. There is an implicit size threshold for objects, where the minimum object size allowed to be cached is min{-*size of the protected segment, M – size of protected segment*}. Upon a cache hit of an object, it is moved to the front of the protected segment. If the object is in the unprotected segment and there is not enough room in the protected segment, the LRU strategy is applied until there is enough room for the object to be moved into it. Objects removed from the protected segment are moved to the head of the unprotected segment.
15. **Generational Replacement**:[49] This strategy uses n ($n > 1$) LRU lists. Each list is indexed, 1, 2… n. Upon being added to the cache, an object is added to the head of list 1. Upon a cache hit, an object belonging to list i is moved to the head of list $i + 1$, unless $i = n$, and then the object is moved to the head of list n. Victim selection begins at the end of list 1, and moves to the next consecutive list only when preceding lists have been depleted.
16. **LRU***:[50] This method combines a LRU list and what is known as a 'request' counter.[3] When an object enters the cache, its request counter is set to 1 and it is added to the front of the list. On a cache hit, its request counter is incremented by 1 and also moved to the front of the list. During victim selection, the request counter of the least recently used object (the tail of the list) is checked. If it is zero, the object is removed from the list; if it is not zero, its request counter is decremented by one and moved to the front of the list and the same process is applied until the new object can be added.
17. **HYPER-G**:[42] This strategy combines LRU, LFU and SIZE. At first the least frequently used object is chosen. If there is more than one object with the same frequency value, the cache chooses the least recently used among them. If this still does not give a unique object to replace, the largest object is chosen.

These next two strategies are extensions of the recency-based PSS strategy mentioned in Section 4.3.1.

18. **Cubic Selection Scheme (CSS)**:[51] As the name implies, CSS uses a cube-like structure to select its victims. Like PSS, CSS assigns objects to classes, except rather than being indexed only by size, CSS indexes classes by size and frequency counts. Each class, like PSS, is a LRU list. Objects in a class (j, k) have sizes and frequencies ranging from $2^{(j,k)\text{-}1}$ to $2^{(j,k)} - 1$. Inversely, an object i belongs to class $(j, k) = (\mathit{floor}[\log S_i], \mathit{floor}[\mathit{log}\ F_i])$. The width, which is the largest value of j is the same as in PSS, since it is based on cache size. However, in order to limit space overhead, there must also be a maximum frequency, $MaxF$, set to limit the height of the cube. Thus, the height of the cube is $\mathit{floor}[\mathit{log}\ MaxF] + 1$. CSS uses a complicated procedure to select its victims, considering the diagonals of the cube and the LRU objects in each list. There is also an 'aging mechanism' applied based on the $MaxF$ set for the cube.
19. **LRU-SP**:[52] Like PSS, this class utilizes classes managed by LRU and has the same number of classes as PSS, each managed by LRU. However, this class accounts for frequency counts as well. An object i is assigned to class $j = \mathit{floor}[\log(Si/F_i + 1)]$. Essentially, as an object is requested more frequently, it decreases the class it is in. When a victim is to be selected, all of the LRU objects of each list are compared based on the value $(\Delta T_i{*}S_i)/F_{i\cdot}$

4.3.4. Implementation of function-based strategies.

These functions use a general characteristic function to define a request value for objects. Most of these algorithms are straightforward requiring a binary heap to sort objects; however, several can become quite time consuming once the recency variable, ΔT_i, has been introduced, requiring a resorting/recalculation of objects when the cache replacement strategy must be invoked. If it is not stated, then we assume that the strategy always picks the object with the smallest request value.

20. **Greedy Dual (GD)-Size**:[53] Defines a request value, R_i, for the object which is recalculated upon insertion of on a cache hit:

$$R_i = \frac{C_i}{S_i} + L \tag{6}$$

where L is an aging factor like in LFU-DA described in Section 4.3.2, and initialized to zero. Whenever an object is removed, L is set to that removed object's R_i value. Here R_i is calculated upon an object request as it is placed in the cache. The factor, $C_i/S_{i,}$ is known as the 'normalized cost'. The normalized cost is meant to describe a proportion of an object's request cost to its size as opposed to a typical Landlord algorithm, which assumes uniform size.[55]

21. **GDSF**:[46] An extension of GD-Size, this uses the frequency information as well to define a value. The request value is defined as:

$$R_i = \frac{F_i * C_i}{S_i} + L \tag{7}$$

L is an aging factor used exactly like in GD-Size.

22. **GD***:[54] An extension of *GDSF*, this method uses a predetermined calculation of temporal locality signified in a variable, β, known as the reference correlation. Reference correlation is measured by the distribution of reference interarrivals for equally popular objects.[3] The variable is meant to be figured optimally by using trace files. However, the authors of the algorithm found the optimal variable to be 1.66, which seemed to produce the optimal metrics for us as well. The request value is calculated as

$$R_i = \left(\frac{F_i * C_i}{S_i}\right)^{1/\beta} + L \tag{8}$$

where L is an aging factor used exactly like in GD-Size.

23. **Taylor Series Prediction (TSP)**:[55] TSP calculates the request value as follows:

$$R_i = \frac{F_i * C_i}{S_i * T_t} \tag{9}$$

where T_T is the 'temporal acceleration' of an object with respect to the current time, and its last and next to last request times. Here $T_T = t_p - t_c$, where t_p is the predicted time for the next request and t_c, the current time. The predicted time is solved using a second-order Taylor series. It should be noted that the variable T_T is similar in concept to $\Delta T_{i,}$ which means it must be recalculated before the cache replacement process begins.

24. **MIX**:[56] A heavily parameterized strategy, MIX is an all-around algorithm which can be tuned for any metric. There are four parameters, referenced as r_i $\{i \mid 1, 2, 3, 4\}$. The request value is calculated as follows:

$$R_i = \frac{l_i^{r1} * F_i^{r2}}{S_i^{r3} * \Delta T_i^{r4}} \tag{10}$$

According to Podlipnig and Boszormenyi,[3] the authors of the algorithm used $r_1 = 0.1$, and $r_{\{2,3,4\}} = 1$. There are no defined ranges, but adjusting the parameters greatly adjusts the factors making the algorithm fairly robust. The only exception is that no aging method is applied in this strategy, but an extension could introduce an aging factor much like GD-Size, etc.

25. **M-Metric**:[57] This strategy takes three parameters: f, s, and t. With these in mind, it defines the request value as

$$R_i = F_i^f * S_i^s * \Delta T_i^t \tag{11}$$

where f should be positive so as to give weight to popular objects. A positive s value will give higher weight to larger objects, while a negative value will give higher weight to smaller objects; t reflects how much recency is taken into account. A positive value gives weight to older objects, while a negative value will result in younger objects taking precedence over older. Based on the parameter values, this algorithm will decide in exactly the same manner as LRU ($f = 0$, $s = 0$, $t < 0$), LFU ($f > 0$, $s = 0$, $t = 0$), and SIZE ($f = 0$, $s < 0$, $t = 0$).

26. **LNC-R-W3**:[58] This method sets a parameter b, which is meant to change the importance of object sizes as in M-Metric. There is also a parameter K, which designates the past K request times to keep track of. Letting t be the current time, and t_k be the oldest request time in the sliding window of K requests for an object i, f_i is set as

$$f_i = \frac{K}{(t_c - t_k) * S_i^b} \tag{12}$$

f_i is then used to calculate the request value as

$$R_i = \frac{f_i * l_i}{S_i} \tag{13}$$

27. **LUV**:[59] Like the other function-based strategies, Least Unified Value (LUV) also defines a request value for each object *i*:

$$R_i = W(i) * p(i) \tag{14}$$

$$W(i) = \frac{C_i}{S_i} \tag{15}$$

where *W*(*i*) is known as the normalized, or relative, cost to fetch the object from its origin server. The other factor in the request value represents the 'probability' that an object *i* will be referenced again. Here *p*(*i*) is defined as

$$p(i) = \sum_{k=1}^{F_i} H(t_c - t_k) \tag{16}$$

$$H(x) = \left(\frac{1}{2}\right)^{\lambda x} \quad (0 \leq \lambda \leq 1) \tag{17}$$

where t_c is the current time, and t_k is the reference time in a sliding window of F_i request times. It should also be noted that *F*(*x*) can be any function, so long as the function is decreasing. We have only provided the function suggested by Bahn et al.[59] and used in our simulations.

4.4. Other implementation details

The implementation language we decided to use was Java SDK 6.0. Although we are mainly concerned with speed, Java performed well on the simulation machine in Ubuntu 7.04 with Intel Core 2 Duo E6600 (2.4 GHz) and 2 GB of RAM. We were able to parallelize the simulations, being able to take advantage of the dual core processor. The only limitations we encountered were memory intensive algorithms, which require use of an auxiliary cache in order to make their decisions. Our solution to the problem was to flag those algorithms which were memory intensive and run only one memory intensive algorithm at a time (memory intensive algorithms could run with other algorithms, as long as they were not also memory intensive).

The main reason we utilized Java was to minimize debugging time, and focus more on the development of the algorithms themselves. Unlike C++, Java's static Math class provided most of the functionality we needed in order to calculate the complex request values of the function-based algorithms. Also, with the addition of template programming and for-each loops in Java 5.0 and higher, it simplified the code for many strategies to a high degree. Lastly, Java's *Hashtable* and *PriorityQueue* classes supplied the most functionality.

Java's *PriorityQueue* class is an implementation of a binary heap. Many algorithms were able to take advantage of this data structure, decreasing the complexity for victim selection from Θ (n)to Θ (lg_2 n). All function-based algorithms took advantage of this process. However, many function-based algorithms which rely on the time that has *passed* since the last reference to an object had to be recalculated at each invocation of the replacement strategy. Most of these algorithms' literature did not discuss appropriate means to calculate their request values efficiently in a timely manner. Bahn et al.[59] proved an efficient way in calculating the request value for LUV (Section 4.3.3), which used all of the previous request times as opposed to a sliding window of request times like LNC-R-W3 (Section 4.3.3).

Algorithms which required a sliding window of *K* request times, or relied on a certain history of the past reference times presented another challenge in our implementation as well. These algorithms were HLRU (Section 4.3.1), TSP (Section 4.3.4), and LNC-R-W3 (Section 4.3.4). At first thought, it would be easy to develop a linked list per object, but the extra space and time overhead in manipulating pointers demonstrated much inefficiency. A simple circular buffer cleared these inefficiencies and worked relatively fast. In fact, since the buffer needs only to be filled one way, it is only necessary to know where the next request must be placed, leading to one array of length *K*, and a pointer to the oldest reference time in the buffer.

Two other algorithms, PSS (Section 4.3.1) and LRU-SP (Section 4.3.2), require a unique variable, ΔF_i, which measures the number of references that occurred since the last time an object *i* was referenced. Originally, a histogram was used and replayed back to see when and how many references occurred since the last time object *i* was referenced. There is sometimes a large time overhead in 'replaying' this information through. A rather simple solution of giving each *request* a unique number, which followed that of the previous request, brought the complexity down to Θ (1). Then, in place for less space and time, we kept track of the last request number that referenced object *i* and the current request ID and were able to calculate ΔF_i with a simple subtraction.

The last detail to cover is the use of auxiliary caches, storing of past information about objects not currently cached. Although we ran a complete simulation set without auxiliary caches on most algorithms, we found algorithms such as HLRU performed so poorly without the use of this information that they may have not existed at all. Thus, the auxiliary cache had to be implemented.

In our case, we did not delete information from the hash table, associated by the objects' URLs, when the object was evicted from the cache. This meant that all information about the objects was kept during the course of the simulation, which lead to large space complexities, and hence the memory-intensive situations.

Normally, the auxiliary cache has its own cache strategies it applies, deleting information of unpopular objects over a certain time period. However, due to the short period our trace files contained (1 week), we decided that the time span was too small to apply any significant strategies to this information.

The algorithms which utilized the auxiliary cache information, and deemed memory intensive, are *HLRU*, *CSS*, *LRU-SP*, *GDSF*, *GD**, *TSP*, *MIX*, *M-Metric*, *LNC-R-W3*, and *LUV*. No algorithms included in Section 4.3.2 made use of the auxiliary caches. Please refer to that section for a more thorough explanation.

5. Performance metrics and cost models

5.1. Performance metrics

Performance metrics are designed to measure different functional aspects of the Web cache. We used three measures. The first two have been used extensively before. The third is a measure of how often the algorithm was invoked.

- **Hit rate:** This metric is used to measure all generic forms of caching. This is simply the number of *cache hits*, as defined in Section 2.1, to the total number of *cacheable requests* seen by the proxy. It is important to realize that on average, 35–55% of the trace files we used were non-cacheable requests. It is also important to note that the ratio of cache hits to total number of *requests* will produce the same ranking of strategies relative to this metric, however the numbers are much smaller, and due to floating point errors, are harder to separate and rank.
- **Byte hit rate:** This metric is similar to hit rate, except it emphasizes the total bytes saved by caching certain objects. Letting h_i be the total number of bytes saved from all *cache hits* that occur for an object i and r_i be the total number of bytes for all *cacheable requests* to object i, and n, the total number of unique objects seen in a request stream, then the byte hit ratio is

$$\frac{\sum_{i=0}^{n} h_i}{\sum_{i=0}^{n} r_i} \tag{18}$$

- **Object removal rate:** This metric came about as an observation of LRU-Min (Section 4.3.1) compared with other similar algorithms. Most algorithms operate under the assumption that disk access or CPU time is far less than the network latency needed to send the data to the client. However, on proxy servers such as those for university campuses or small businesses, the time to sort through 10,000 or more Web objects during the cache replacement strategy may be comparable to the network latency to the client. The object removal rate is essentially the measure of the amount of times the cache replacement strategy removes an object from a cache to the number of cacheable requests that occur. In a cache where watermarks may be set, this measure may tend to be very similar among many objects. However, in a cache where the replacement method is invoked when only necessary, this shows some surprising results of how well objects are at making the right decisions and how often they have to. Letting k_i be the number of times object i was removed from the cache, and z_i be the number of times object i was admitted to the cache, the object removal rate is

$$\frac{\sum_{i=0}^{n} k_i}{\sum_{i=0}^{n} z_i} \tag{19}$$

It is important to keep in mind that hit rate and the byte hit ratio cannot be optimized for at the same time. No cache replacement strategy can be deemed as the best because there is a tendency in request streams for smaller documents to be requested more often than larger ones due to the download time it takes to gather these objects. Strategies that optimize for hit-rate typically give preference to objects in a smaller size range, but in doing so tend to decrease byte-hit rate by giving less consideration to objects not in a particular size range.

A high removal rate may suggest several possibilities: that many small objects are being removed for larger objects, or that a poor decision is being made in relating object size to the number of objects to remove for that object. If the algorithm has a complex process per decision, then it is an indication that the algorithm may not be decreasing the number of documents or outside references the proxy has to make. However, it may be decreasing network latency as it may be giving preference to objects that have high latency/network costs, which is a factor the cost models use.

5.2. Cost models

One of the most important characteristics of a Web object is the object's cost to fetch it from the origin server. There are several ways to relate a Web object to its cost. In this section, we note all of the different models we came across. Also, in the following, let S_i represent the size of a particular Web object i.

- **Uniform model:** This assumes that the cost to fetch all objects is the same and so sets the costs of all objects to some constant greater than or equal to 1.
- **Byte model:** This assumes that the network latency is the same for all servers, and so the only deciding factor in the cost for getting the object will be its size (as the size will decide the time to fetch the object). Thus, this method sets the cost to the size of the object.
- **Time model:** This uses the past latency (download time) information to predict the future download time for an object. Thus the object's cost is set to the predicted time it would take to fetch an object from a server. This has the unique advantage over the byte model in the rare occasion that small files that are on slow servers will have a higher cost, allowing them to have precedence over files that are similar in size. Since latency information cannot be gathered directly from the trace files, we used the *Time Model* technique as shown by Hosseini-Khayat,[60] which randomly assigns a time cost as

$$C_i = x(q) + \frac{S_i}{y} \tag{20}$$

  where x is an exponential random variable representing network latency with a mean θ and y, which represents network throughput.
- **Network cost:** This is similar to the byte model, but rather than predict future network latency and throughput, this simply estimates the number of packets that must be sent for a particular object. Thus, the cost based on the number of approximate packets for an object i is

$$C_i = 2 + \left\lceil \frac{S_i}{536} \right\rceil \tag{21}$$

6. Experiment setup and results

6.1. Trace files

In our experiment, we used trace files, which are files with recorded Web requests, to test each replacement strategy. The trace files were provided by IRCache.[31] These trace files are used in much of the previous proxy cache research.[29,30,37,38,61] IRCache gathers the trace files and other data on Web caching from several different proxy servers located around the United States. More than enough information is provided in these files to indicate which requests were cacheable.

Originally, the trace files were provided with data spanning only a day of recorded Web requests. While some researchers in the literature used 1 hour or at most 1 day of requests, we strung seven consecutive trace files together to create a week-long trace file from each proxy that the data came from. Once this was done, we then 'cleaned' the files to have only the cacheable requests (refer to Section 2.1 for a definition of a *cacheable request*) in them as to decrease the simulation time. We also exchanged each unique URL that appeared in the file with a unique integer identifier so that string comparison time could be decreased as well.

Table 4 presents statistics about the three traces we used for this simulation. These particular trace files were chosen due to their differences in size and cacheable request rates. Non-cacheable requests were extracted from the files prior to our experiment. Each trace represented varying levels of temporal locality, spatial locality, total bandwidth and number of requests testing the various limits of the replacement strategies. All trace files represent 1 week of recorded requests caught by IRCache from 12:00 A.M, Tuesday, 10 July 2007 to 12:00 P.M, Monday, 16 July 2007. This reflects realistic Internet traffic.

6.2. Simulation setup and parameters

The cost model we chose for our simulation was the time model, which produced similar results to both network and byte models. We used the same values as Hosseini-Khayat[60] suggested, with the mean of the exponential variable, x, set to 1000 ms, and y being a uniform random variable between 1 byte/ms to 1000 bytes/ms.

Some of the strategies presented in Section 4 had one or more parameters. Table 5 shows a list of these strategies and their corresponding parameters. We ran several simulations of each strategy with different values for each parameter. In Section 6.3, we present only the instances of the parameters that reflected the best result for the corresponding strategy. If there is more than one parameter, Table 5 also shows the order these parameters are listed in the graphs described in Section 6.3. For example, in Figure 4, AlphaAging(3600000/0.25) means that α–Aging performed best with Interval=3600000 ms (or 1 hour) and $\alpha = 0.25$.

We simulated the algorithms with varying limits of the cache size available on the disk. We started at 50 MB, then 100 MB, and finally ended with 200 MB. The reason for the small amount of disk space, when typical proxy cache servers might operate in terms of gigabytes was to engage the cache replacement algorithm as

Table 4. Trace file statistics for requests and bandwidth

Trace file	Urbana-Champaign, IL (UC)	New York, NY (NY)	Palo Alto, CA (PA)
Total requests	2,485,174	1,457,381	431,844
Cacheable requests	55.31%	51.70%	23.61%
Total bytes	71.99 GB	17.70 GB	5.601 GB
Cacheable bytes	95.62%	90.55%	88.30%
Unique requests	1,306,758 (52.58%)	708,901 (48.64%)	241,342 (55.89%)
Unique cacheable	73.78% of unique requests	73.71% of unique requests	33.89% of unique requests

Table 5. Order and description of parameters in the results

Strategy		Parameters
LRU-threshold	(Section 4.3.1)	*Threshold*: The maximum size threshold of the cache.
Pitkow/Reckers strategy	(Section 4.3.1)	*Interval*: Interval set to either daily or hourly describing when objects are differentiated by size.
HLRU	(Section 4.3.1)	*h:* The hist-value to use in a sliding window of *h*-requests.
LFU-Aging	(Section 4.3.2)	*Average frequency threshold*: The aging factor. *Maximum frequency threshold*: The maximum frequency counter of any given object.
α – *Aging*	(Section 4.3.2)	*Interval*: Interval, in milliseconds, of when the aging factor is applied. α: The aging factor.
Segmented - LRU	(Section 4.3.3)	*Protected Segment*: The size, as a percentage of the total cache size of the protected segment.
Generational replacement	(Section 4.3.3)	*Generations*: Number of generations used.
Cubic selection scheme	(Section 4.3.3)	*Max Frequency*: The maximum frequency counter, always a power of 2.
*GD**	(Section 4.3.4)	β: Parameter describing reference correlation.
MIX	(Section4.3.4)	r_1, r_2, r_3, r_4: Refer to Section 4.3.4 for more information.
M-metric	(Section 4.3.4)	*f*: Frequency weight *s*: Object size weight *t*: Recency factor.
LNC-R-W3	(Section 4.3.4)	*K*: Describes the size of the sliding window of past *k* requests. *B*: Object size weight.
LUV	(Section 4.3.4)	λ: Describes an exponential scaling factor for F(x)

frequently as possible. Note also that as the maximum disk size allowed increased, all of the replacement strategies performed better with respect to each metric. However, the general increase in performance did not significantly change the ranking indexed by a particular metric in any of the simulations. For this reason, we present the best instance of each strategy when the cache size was set to 200 MB. Significant differences for particular instances of strategies will be noted later.

Prior to running each test, we warmed up the cache with a smaller trace file from Boulder, Colorado. By using another trace file different from the others, we could guarantee that no files from that trace run would conflict with the other trace files. As a result, the cache would be filled by the time we started our simulation, putting our cache replacement strategies in effect immediately upon starting our tests. Therefore, all of the results presented in Section 6.3 are the full results of the cache replacement strategies.

6.3. Simulation results

First, we present the results of individual strategies categorized as set by Section 3 for each metric. Then, as a global comparison, we take the top three strategies from each category for a particular metric and compare them overall, to get a better sense of how the strategies from each category rank. After the presentation for each metric and trace file, we discuss the results as a whole. Second, we note peculiarities between different traces for individual strategies such as MIX (Section 4.3.4), M-Metric (Section 4.3.4), CSS (Section 4.3.3), etc. Many strategies that utilized parameters will also need further analysis to see how different combinations of parameters affect the overall functionality. Lastly, we discuss how certain attributes about the request streams affect certain strategies comparatively.

We have tested the reliability of these results by first validating our implementations of LRU and LFU. By running our simulation and verifying the results against a smaller version of a trace file and with smaller cache limits, we were able to compare the results against an expected result set.

The reader should note that the graphs presented in this paper do not start at an origin of zero and in fact many of the graphs start at different origins. This was done to be able to graphically demonstrate our results in an easy to see and comparative manner. If the origin is at zero, most results would not be seen clearly. While some results are close, it is important to note that slight differences in percentage points of these metrics can equate to thousands of missed requests or bytes.

6.3.1. Hit rate.

Figures 4–18 show the hit rates for our simulations. Figures 4–7 shows the results for the frequency, recency, recency/frequency, and function-based categories, respectively, using the UC trace file. Figures 8–11 shows the results for the four categories using the PA trace, while Figures 12–15 shows those for NY trace files. Figures 16–18 show the overall comparison of all strategies selecting the best three from each category, using the UC, PA, and NY trace files, respectively. Results of the recency category for the UC trace

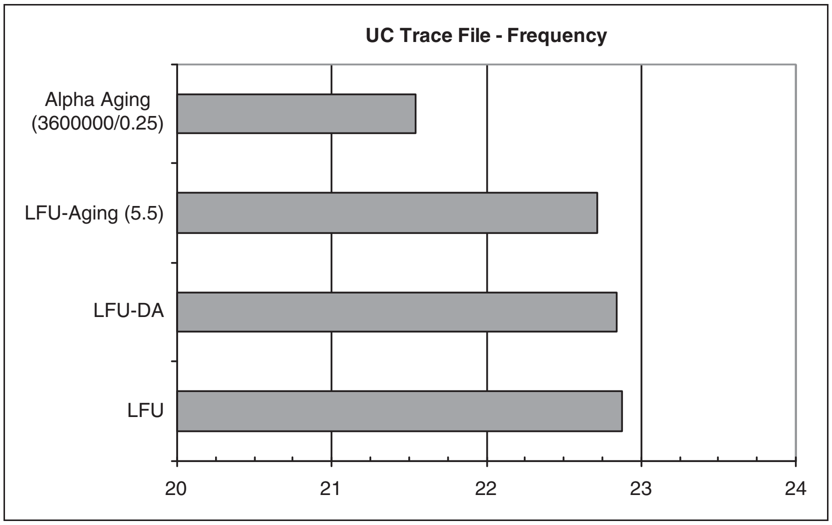

Figure 4. Hit rate for frequency using the UC trace file.

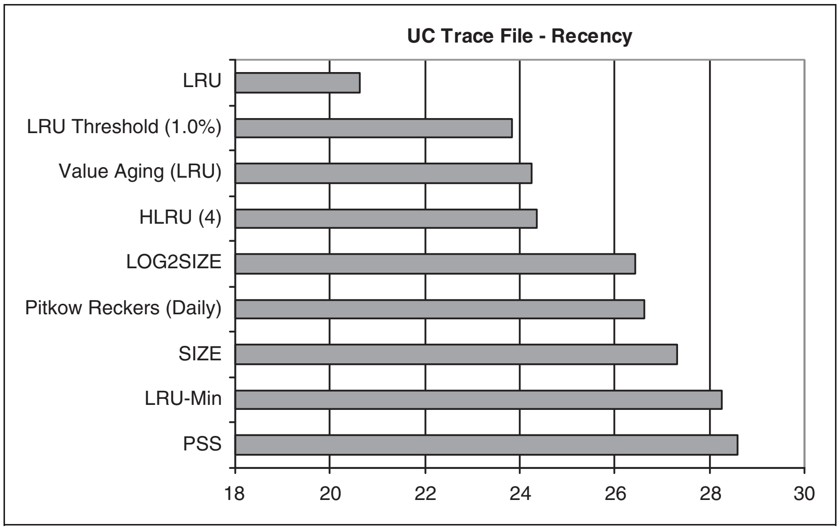

Figure 5. Hit rate for recency using the UC trace file.

in Figure 5 have a smaller variance compared with the PA trace in Figure 8, demonstrating the effect of its high request rate.

In the recency category PSS performed the best for UC, PA, and NY traces as shown in Figures 5, 8, and 12, demonstrating that using recency along with grouping similar objects by size demonstrated its ability to intelligently remove the correct objects. However, the gain from its complicated decision process is questionable, as shown in the aforementioned figures, because simpler algorithms such as SIZE performed almost just as well. By examining Figures 5, 8 and 12, one can clearly notice that among the recency category the four algorithms: PSS, SIZE, LOG2SIZE, and LRU-Min did well consistently and demonstrate that when considering recency, size should also be considered at the same time for hit rate.

One can also notice from the figures that LRU, the parent strategy of the recency category, consistently did the worst. This is by far a revealing development because LRU is so widely used commercially in place of many of these other strategies. Simply considering the object size or using a little more complicated strategy such as LRU-Min gains a considerable amount of performance over LRU; in conclusion, when recency is used as a base factor, derivative algorithms on other object characteristics will generally do far better.

This observation, however, does not apply to the frequency-based strategies. LFU as shown in Figures 4, 9, and 13 always outperformed its derivative

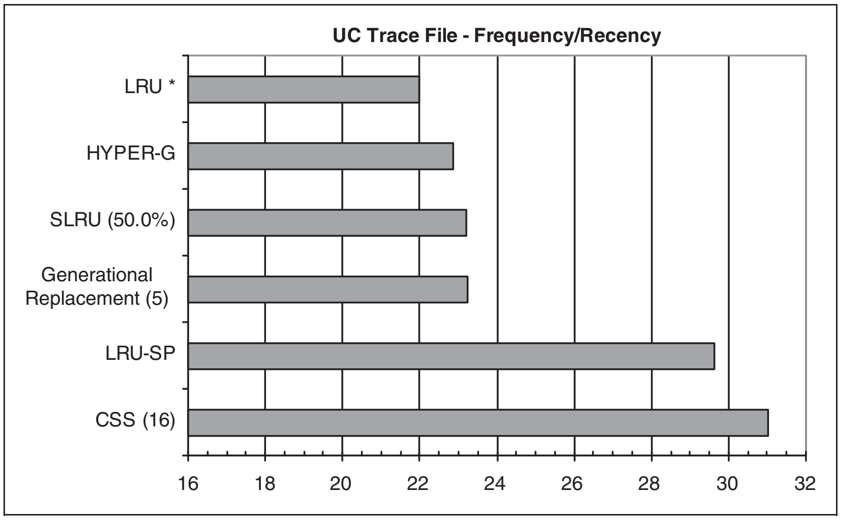

Figure 6. Hit rate for frequency/recency using the UC trace file.

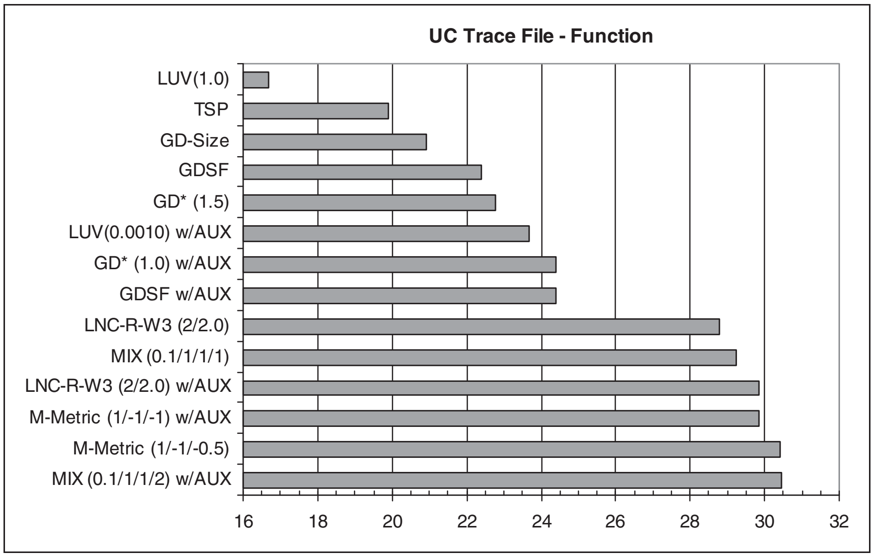

Figure 7. Hit rate for function using the UC trace file.

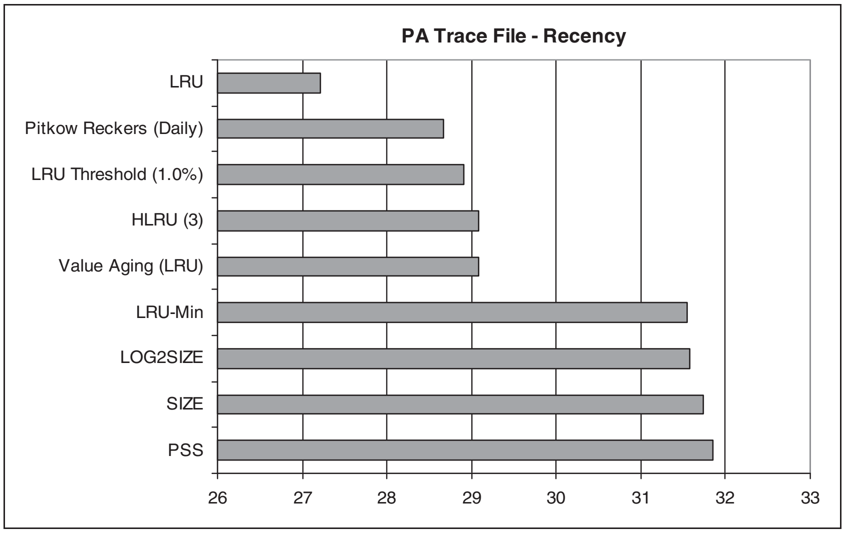

Figure 8. Hit rate for recency using the PA trace file.

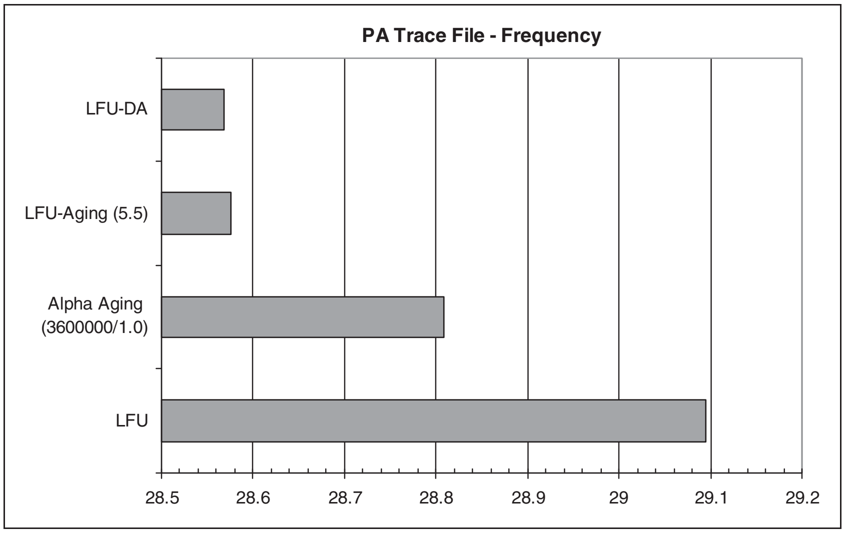

Figure 9. Hit rate for frequency using the PA trace file.

strategies. One reason may be that over the course of the simulated week, aging the frequency counters may not be needed since we used in-cache frequency. In that respect, when an object is removed, and if it should enter the cache again, it would have to accumulate its frequency count again; essentially this is an aging factor in itself, although instead of being applied globally as LFU-DA and LFU-Aging attempt to do, it is applied when the object is removed; applying global aging factors on top of in-cache frequency may actually lead to an imbalanced weighting of frequency counts. Owing to this flaw, the Frequency strategies are always outperformed by the other categories' best in the overall charts as shown in Figures 16–18.

From Figures 6, 10, and 14, it is clear that for frequency/recency strategies, however, LRU-SP and CSS did the best consistently. Although it is not displayed here, CSS for any parameter generally did the same with an incredibly small variance (this is also true across all other metrics as well). LRU-SP generally did as well as PSS or a little better. With the exception of HYPER-G, all of the algorithms did outperform LRU in hit-rate, holding our earlier observation valid.

Figures 7, 11, and 15 show the results of hit rate for function-based strategies. These strategies hold the widest range of results. With the idea of auxiliary caches, along with several different parameters, it should be of no surprise that these are the most

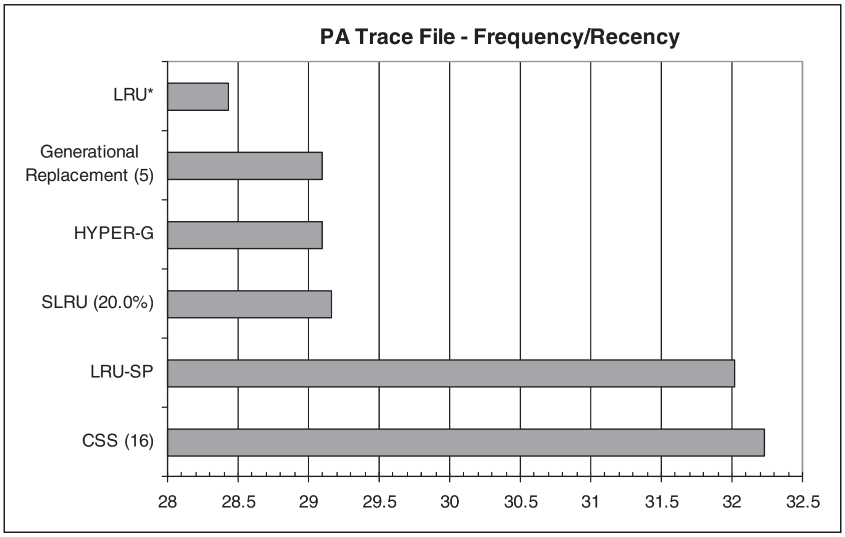

Figure 10. Hit rate for frequency/recency using the PA trace file.

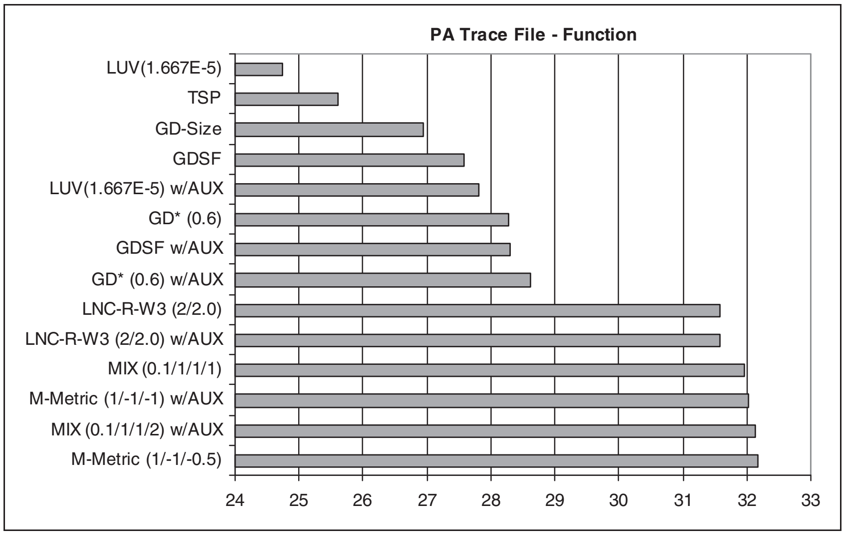

Figure 11. Hit rate for function using the PA trace file.

complicated to implement and can require long durations of parameter tuning. An entire body of literature could explain the effect of each parameter on each of the appropriate algorithms, and yet, with the addition of a weighting factor such as that in GD* from GDSF, that it could be grounds for an entirely new strategy in of itself. In fact, MIX and M-Metric are so similar that in a uniform cost model, either could be used to model the other with the appropriate parameters.

Owing to the nature of the parameters, what would perform well on one trace would almost certainly not be the same for the other traces. In an actual environment, this would require system administrators to be carefully tuning the metrics and be able to understand the parameter's relations to characteristics of the request stream. This would also require use of some type of metric to measure characteristics of request streams, which although not discussed in this literature, are as opposed to hit rate and byte hit rate for most cases.

Some general cases can be made for function-based strategies. The use of an auxiliary cache to keep further information on Web objects generally added to the hit rates of many strategies. The only exception to this was the M-Metric strategy. It is unclear as to why this oddity would occur in M-Metric and not in MIX with similar values, but the consideration of latency in MIX seems to have been enough where the addition of the auxiliary cache made all the difference.

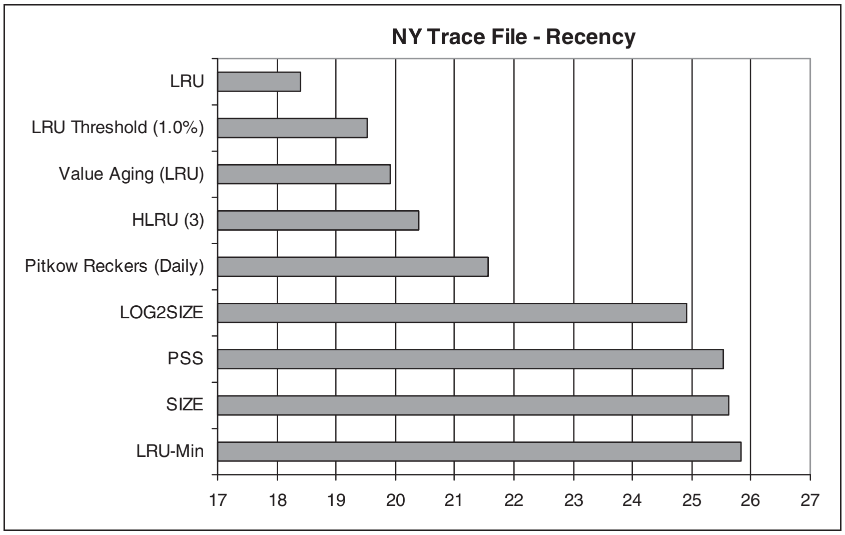

Figure 12. Hit rate for recency using the NY – 200 MB trace file.

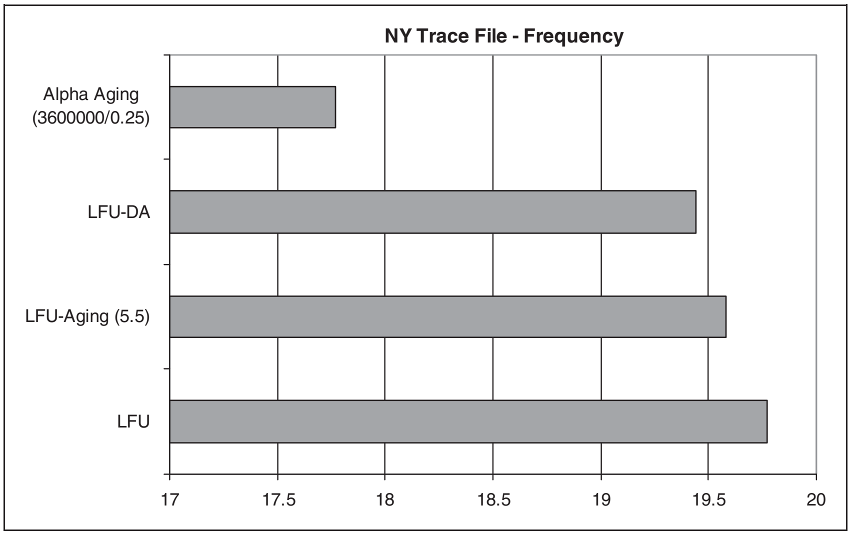

Figure 13. Hit rate for frequency using the NY – 200 MB trace file.

We expected the results for cost-based algorithms to be lower than the rest for a rather simple reason when it came to hit-rate: Users do not make judgments based on the latency cost of acquiring an object because in terms of those objects which are most frequently accessed (the smaller ones usually), the user may only feel a half a second lag. Relative to having to acquire a larger file, most users will ignore the delay for those objects which would affect hit rate the most. Thus, information pertaining to *latency* and/or *network* costs of acquiring an object will generally decrease the hit rate when used as a factor in the decision process.

Overall, Figures 16–18 show that CSS outperformed all other strategies consistently by utilizing a strong balance between size, frequency and recency information to make its decisions. Followed closely by function-based strategies, which after parameter tuning will generally do almost as well at balancing characteristics, and as we will see later have the ability to capture more than just hit rate. When modified from LRU and SIZE, recency strategies clearly outperformed the frequency strategies (keep in mind that LFU outperformed LRU greatly). Frequency/recency strategies held their own, outperforming straight recency strategies on average.

6.3.2. Byte hit rate. In previous literature, it has been noted that byte hit rate tends to be inversely related to hit rate. If a strategy increases its hit rate, generally it will decrease its byte hit rate. This is mainly due to the

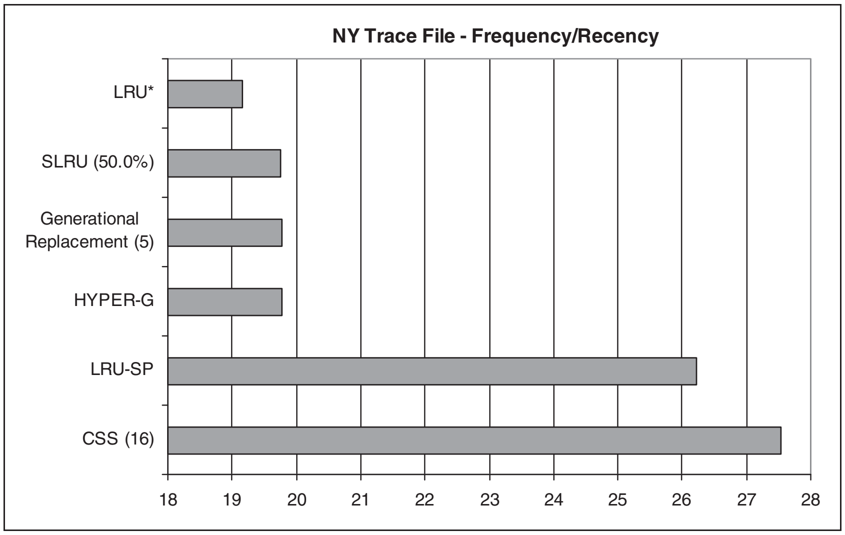

Figure 14. Hit rate for frequency/recency using the NY – 200 MB trace file.

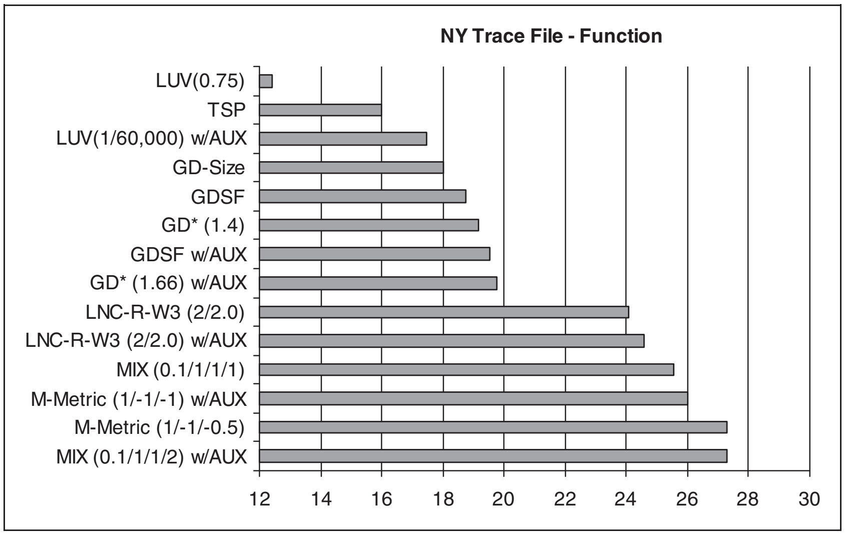

Figure 15. Hit rate for function using the NY – 200 MB trace file.

fact that larger Web objects are accessed less often because these files are updated less frequently and have high latency to acquire. Also, generally the network cost to access the object one time is much larger than most other files.

However, this is also an advantage to proxy caches because they can save large amounts of bandwidth with these assumptions as well. Objects with high cost and large size are generally targets for system administrators trying to cut down on bandwidth costs for servers. Thus, there is a tradeoff between saving bandwidth and decreasing user perceived lag. In one, the users will feel the effects of the proxy cache, where as in the other, the origin servers will witness a cut in bandwidth costs.

Thus, it should be of no surprise that LOG2SIZE, SIZE, LRU-Min, and PSS, which did well under hit rate, perform the worst in byte hit rate shown in Figure 19. The one exception occurs in the PA trace file, Figure 20. In fact, the exception occurs again in comparison with other categories in the PA trace results of Figure 21 as well. LRU-SP, derived from PSS also has similar effects. These out of line occurrences may be due to the fact that the PA trace file has a sparse request stream with less than a quarter of cacheable requests. Strategies that relatively compare objects' characteristics adapt to the density of request streams as opposed to completely falling out. Also, strategies that compare static characteristics, characteristics that do not vary

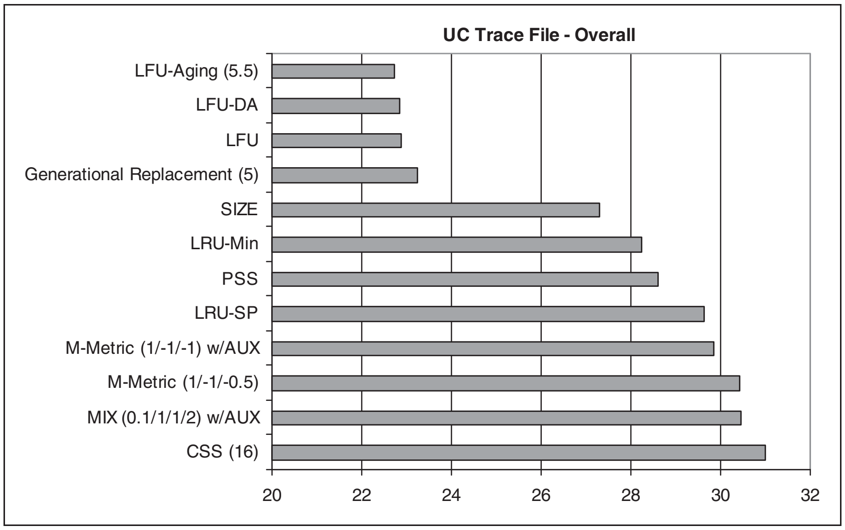

Figure 16. Hit rate for overall using the UC trace file.

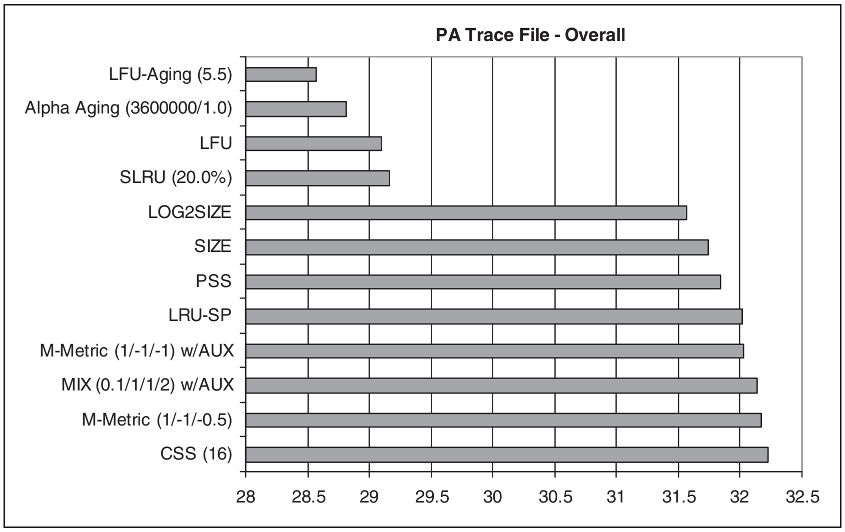

Figure 17. Hit rate for overall using the PA trace file.

much over the course of the simulation, also tend to do well on sparse request streams. For instance, M-Metric and MIX represented in Figure 22 are outperformed by the Greedy-Dual derivatives and LNC-R-W3. However, it should be noted that the majority of function-based strategies greatly outperform many of the other categories, and as well have a low deviation from one another.

Another observation is that HLRU does well in the UC trace, Figure 19, and NY trace, Figure 23, and also manages to do the best for the PA's Recency set, Figure 20. This may suggest that considering when the past *h*th request (Section 4.3.1) occurred is somehow relevant to the size of objects. Value-Aging also did fairly well in comparison with other recency strategies, but did only mediocre overall. This is most likely due to the fact that Value-Aging slowly increases as the time grows, which is an advantage to larger objects, which tend to have long periods between requests.

In terms of function-based strategies Figures 22, 24, and 25 all represent a different ranking between one another. For UC Figure 24, MIX on top and PA, Figure 22, with GD* and GDSF and NY, Figure 25, with LNC-R-W3 and GD*, all had LNC-R-W3 in the top three. This makes sense since HLRU did well in recency, which utilizes a sliding window scheme of *h* requests like LNC-R-W3. However, LNC-R-W3 also uses information about frequency and size, which

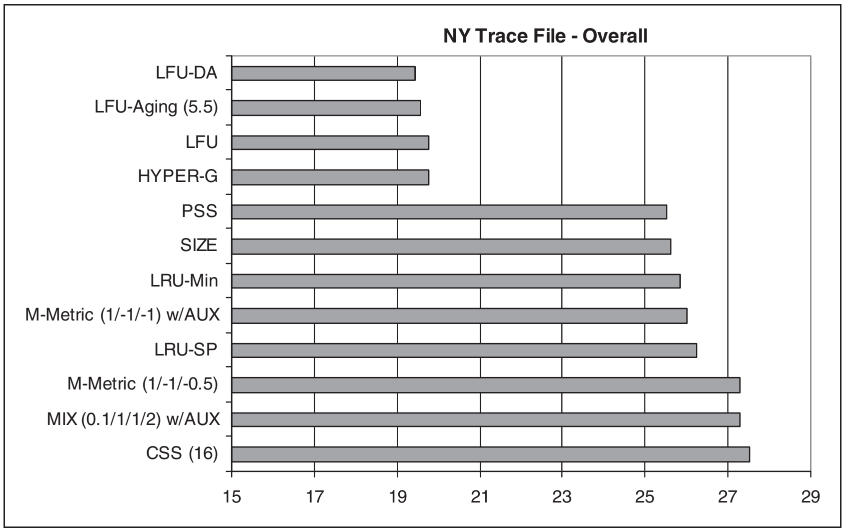

Figure 18. Hit rate for overall using the NY trace file.

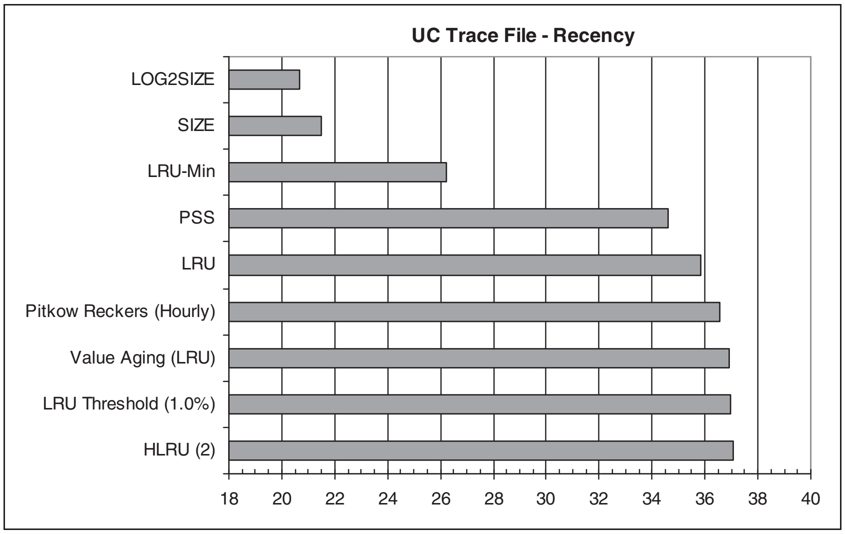

Figure 19. Byte hit rate for recency using the UC trace file.

enhances its comparisons over HLRU to optimize primarily for byte hit rate.

In terms of the frequency class, Figures 26–28, LFU-aging seems to perform well. Again, the frequency-based methods did worse overall, Figures 21, 29, and 30, but it is still too little data to rule out frequency as being an irrelevant characteristic, as LFU still outperforms LRU each simulation. Also, the aging factors for LFU-DA and LFU-Aging, which were a problem for hit rates, actually work to the advantage of large objects under byte hit rate. In this condition, since no frequency counter can be less than 1, usually the aging factors have no effect on large objects when the aging factors have been applied. Thus, the objects with higher frequencies are brought down in comparison with the large objects giving some larger objects an equal chance to stay in the cache as their smaller counterparts.

6.3.3. Removal rate.

The removal rate highlights a third tradeoff with respect to the proxy server itself. As stated in Section 5.1, removal rates can be a significant indicator of CPU usage and storage accesses. For a high removal rate, generally we can assume that the strategy on average exchanges smaller objects for larger objects. A low removal rate suggests the strategy removes larger objects first in exchange for the placement of many smaller objects.

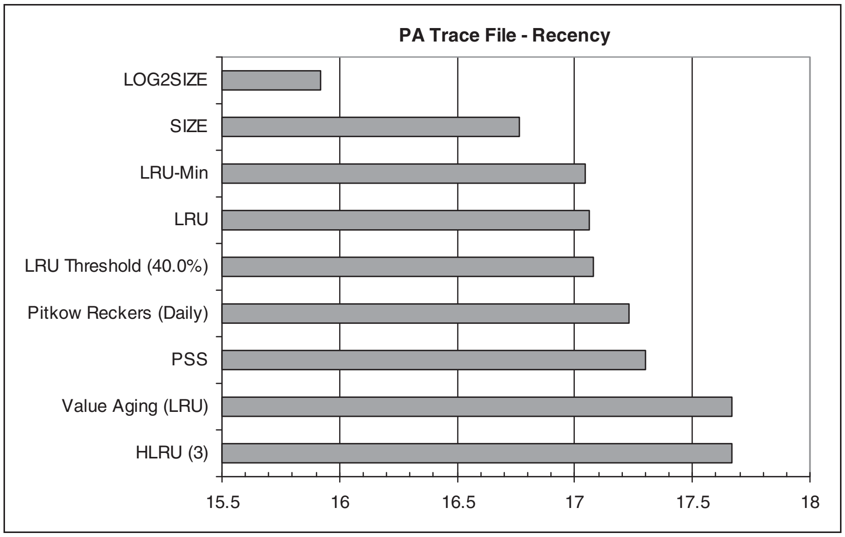

Figure 20. Byte hit rate for recency using the PA trace file.

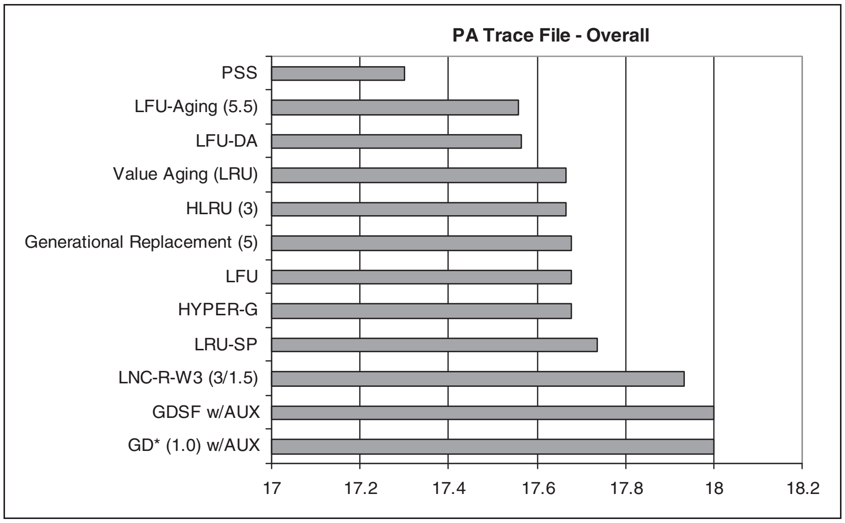

Figure 21. Byte hit rate for overall using the PA trace file.

Clearly, under these assumptions, SIZE and other similar strategies that performed well under hit rate but not so well in byte hit rate will perform optimally here. This suggests that hardware will be in more use to serve the users more than save on bandwidth, which has a greater effect on the origin servers. In this strategy, smaller objects are constantly swapped for other equally sized objects, while larger objects are removed first before smaller ones. Figures 34–48 show the removal rates for our simulations. Figures 34–37 shows the results for the recency, frequency, recency/frequency and function based categories, respectively, using the UC trace file. Figures 38–41 shows the results for the four categories using the PA trace, while Figures 42–45 shows those for NY trace files. Figures 46–48 show the overall comparison of all strategies selecting the best three from each category, using The UC, PA and NY trace files, respectively.

In the function-based strategies Figures 37, 41 and 45, the strategies follow the general rankings as set by the byte hit rate. However, an odd occurrence in PA's trace file, Figure 41, shows that the deviation between the highest and lowest simulations is between 50% and 60%. It is an interesting occurrence because in hit rate, Figure 11, the greatest deviation is about 5–7% of the highest and lowest, and the byte hit rate, Figure 22, has an even smaller deviation. NY trace results have a much larger deviation on both, and yet a rather stable removal rate.

This suggests that while deviations of hit rates and byte hit rates may be low, there is no clear pattern between removal rate and the other two metrics *despite*

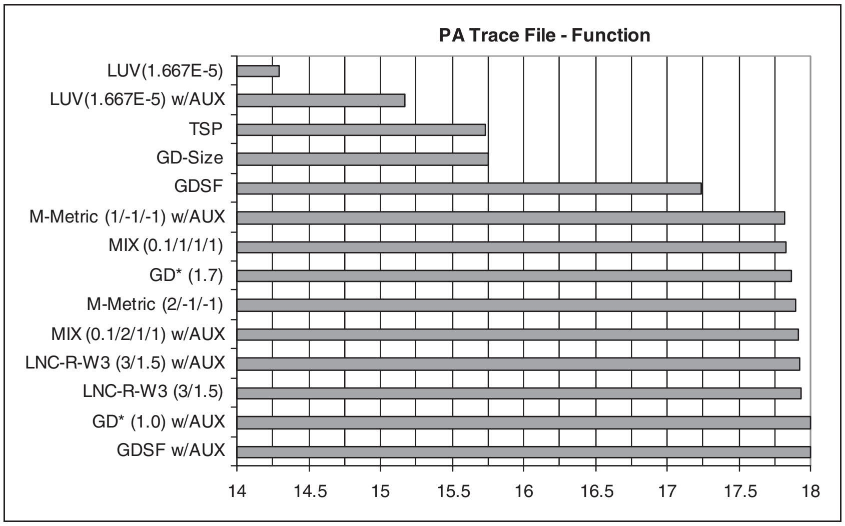

Figure 22. Byte hit rate for function using the PA trace file.

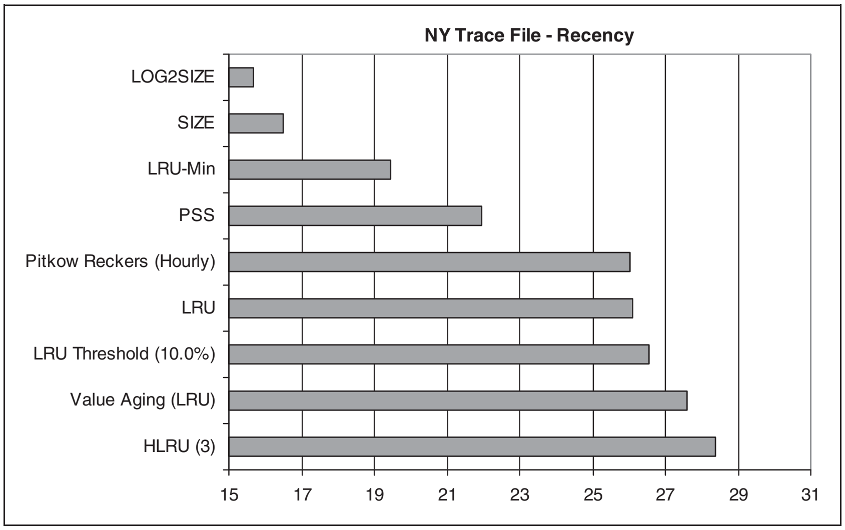

Figure 23. Byte hit rate for recency using the NY – 200 MB trace file.

that it generally may be otherwise. In the case of PA's results, it would be a much wiser decision based on a sparse request stream, to implement M-Metric and MIX because they will have less drive access, making smarter decisions about which objects to replace, while having almost identical byte hit rate to Greedy-Dual's derivations and doing far better in hit-rate than other function-based methods.

7. Conclusion

Proxy servers have been used extensively to reduce network traffic and improve the availability and scalability of the network. We have shown how one aspect of the Web cache, the cache replacement strategy, can adversely affect the performance. This paper has provided an exhaustive quantitative analysis of cache replacement strategies based on three metrics. The metrics are very important as they indicate the amount of bandwidth (or network traffic), user perceived lag (or delay time), and CPU usage (or disk access).

A comprehensive study of 27 algorithms was included along with details of their implementation. First, we presented the results of individual strategies categorized as recency, frequency, recency/frequency, or function based for three different trace files and three different metrics. Then, as a global comparison, we took the top three strategies from each category for a particular metric and compared them overall, to get a better sense of how the strategies from each category rank. Second, we noted peculiarities between different traces for individual strategies. Many strategies that

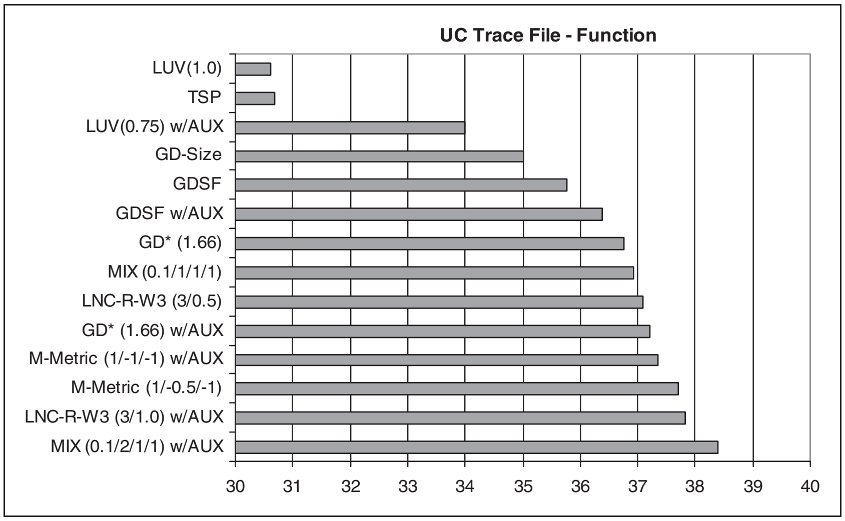

Figure 24. Byte hit rate using the UC trace file.

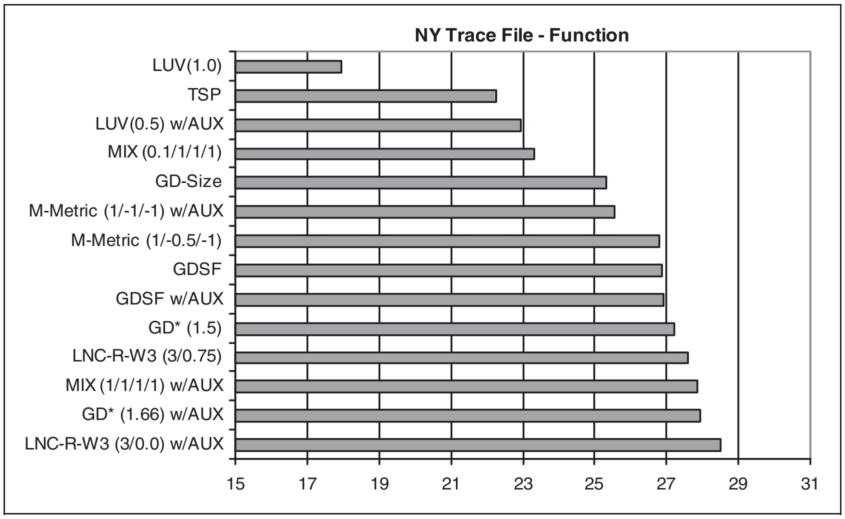

Figure 25. Byte hit rate for function using the NY trace file.

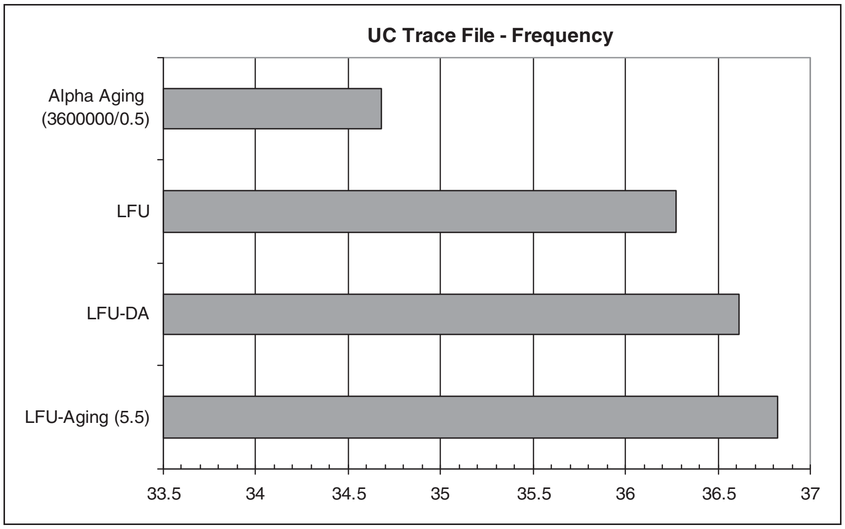

Figure 26. Byte hit rate for frequency using the UC trace file.

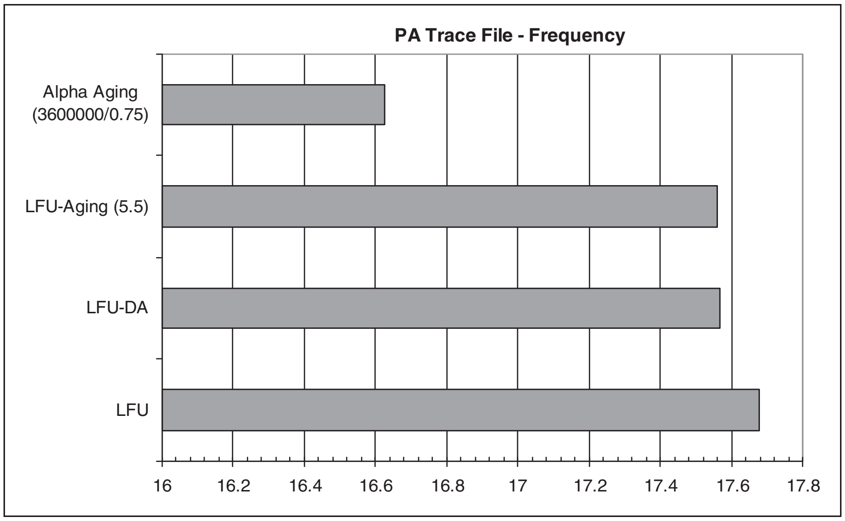

Figure 27. Byte hit rate for frequency using the PA trace file.

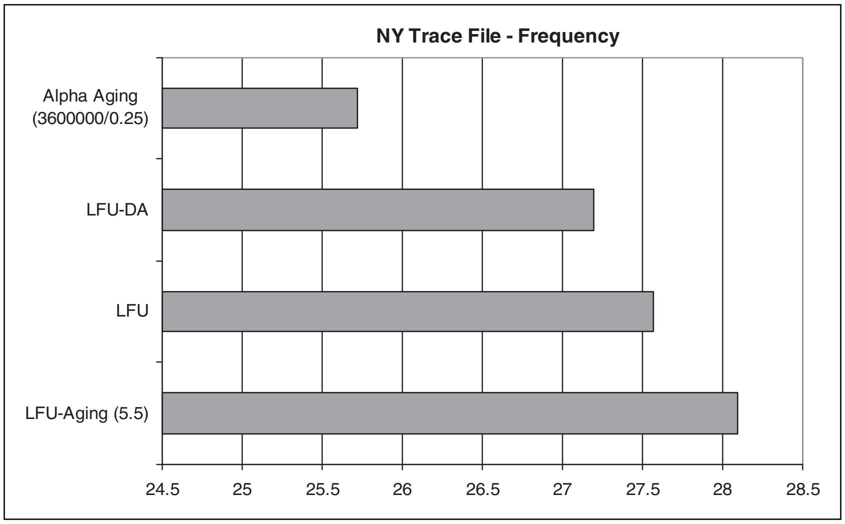

Figure 28. Byte hit rate for frequency using the NY – 200 MB trace file.

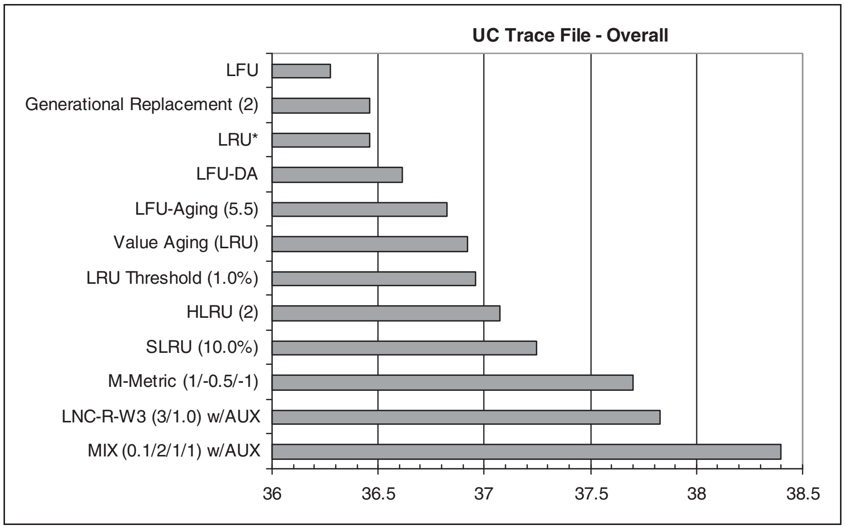

Figure 29. Byte hit rate for overall using the UC trace file.

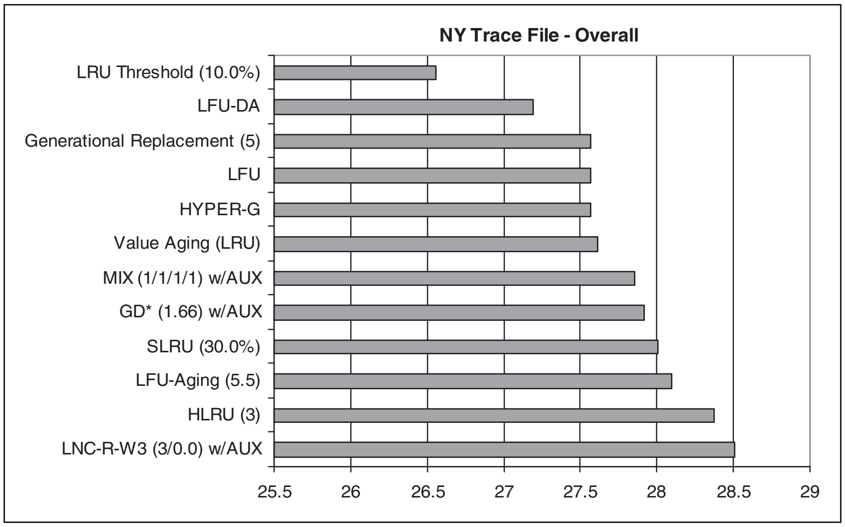

Figure 30. Byte hit rate for overall using the NY trace file.

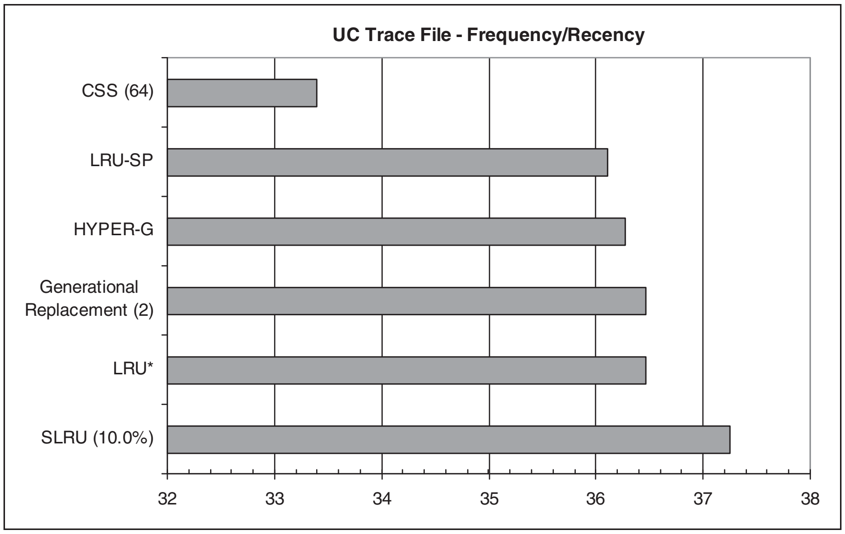

Figure 31. Byte hit rate for frequency/recency using the UC trace file.

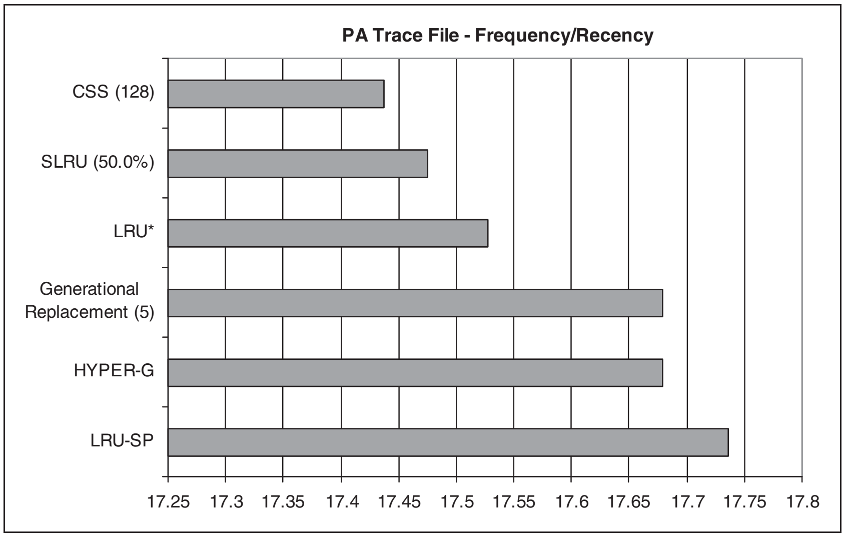

Figure 32. Byte hit rate for frequency/recency using the PA trace file.

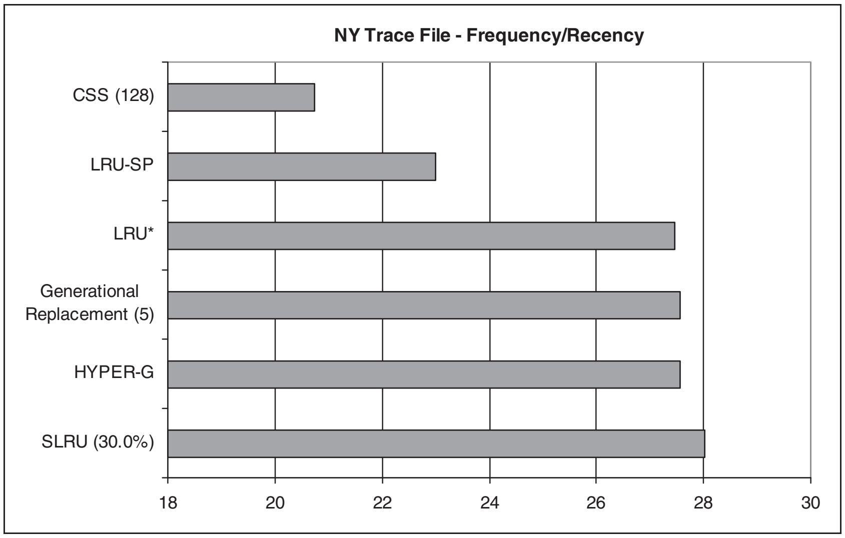

Figure 33. Byte hit rate for frequency/recency using the NY trace file.

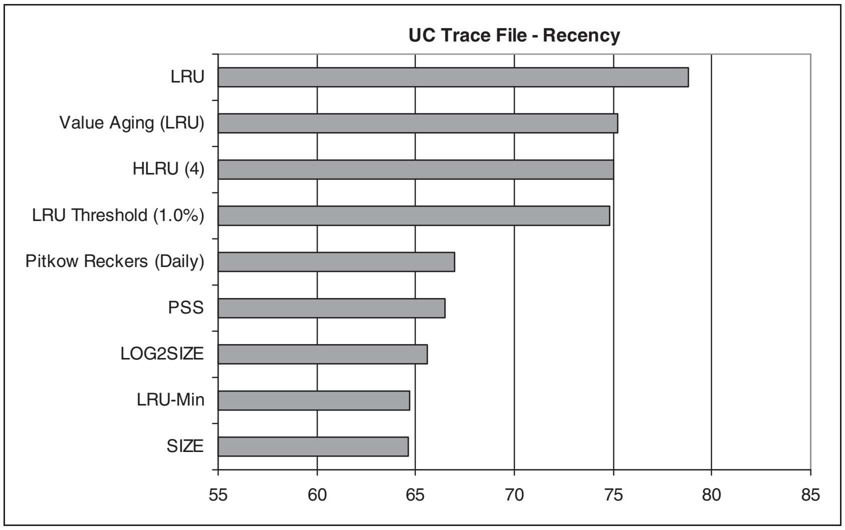

Figure 34. Removal rate for recency using the UC trace file.

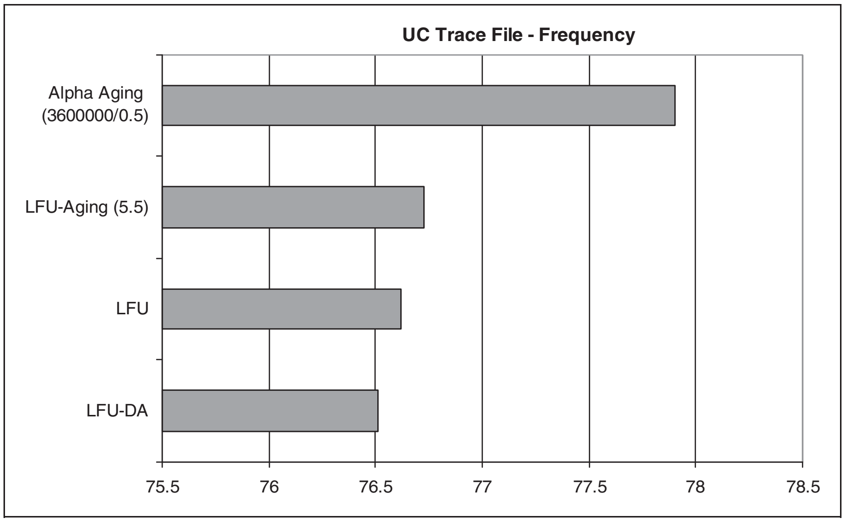

Figure 35. Removal rate for frequency using the UC trace file.

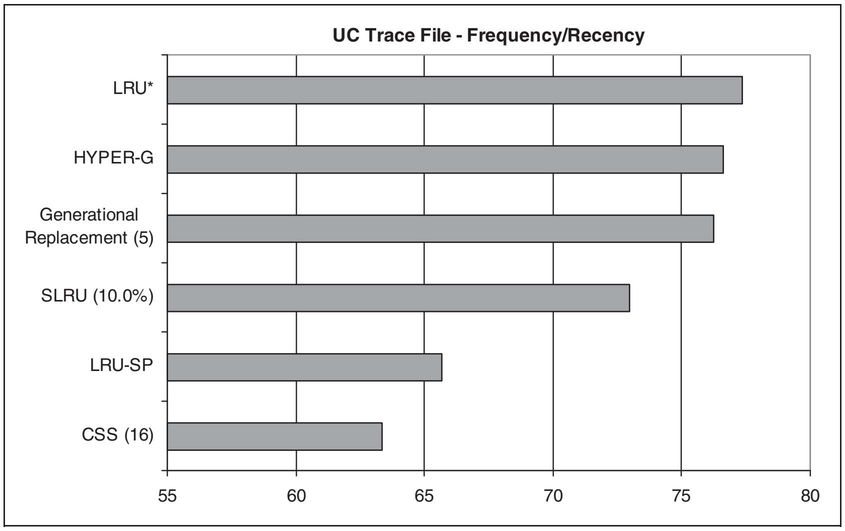

Figure 36. Removal rate for frequency/recency using the UC trace file.

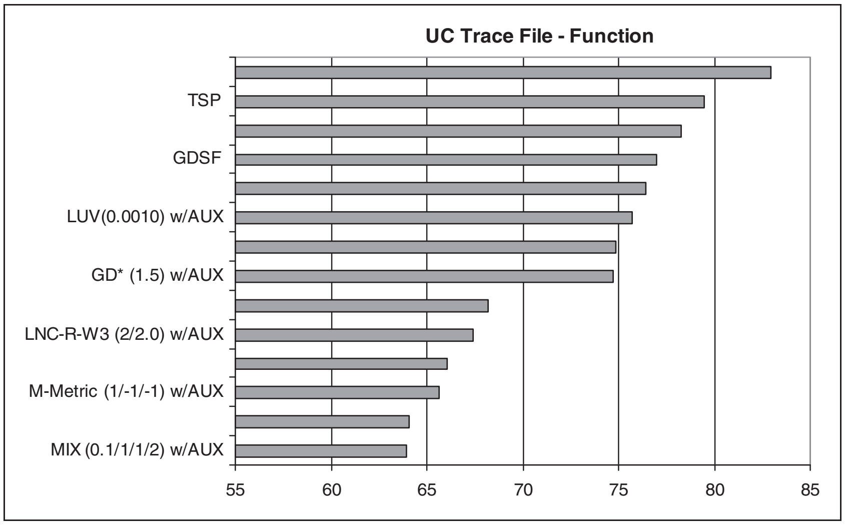

Figure 37. Removal rate for function using the UC trace file.

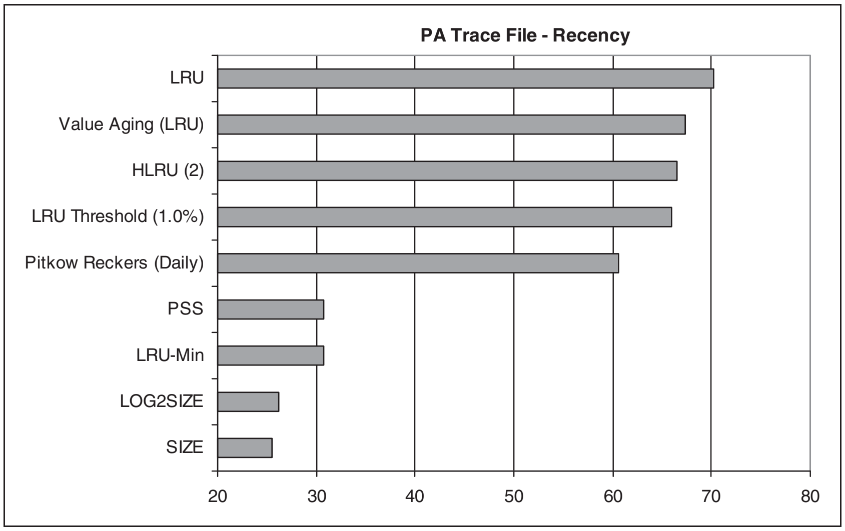

Figure 38. Removal rate for recency using the PA trace file.

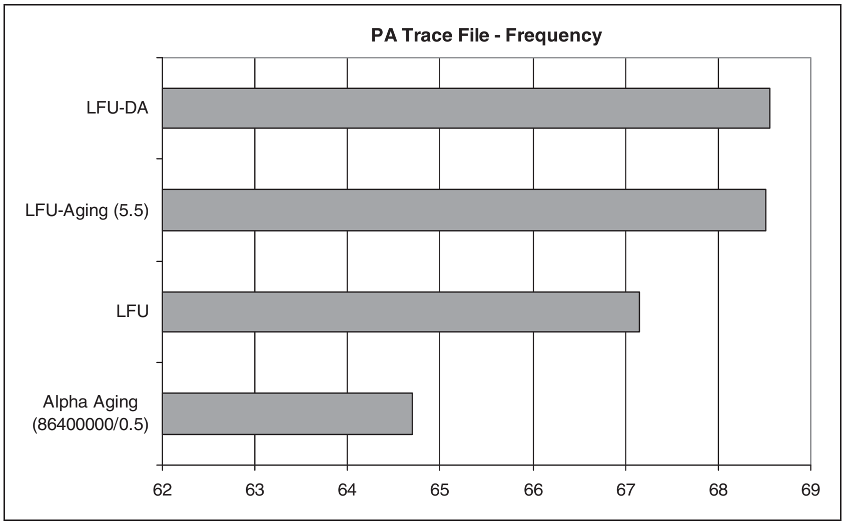

Figure 39. Removal rate for frequency using the PA trace file.

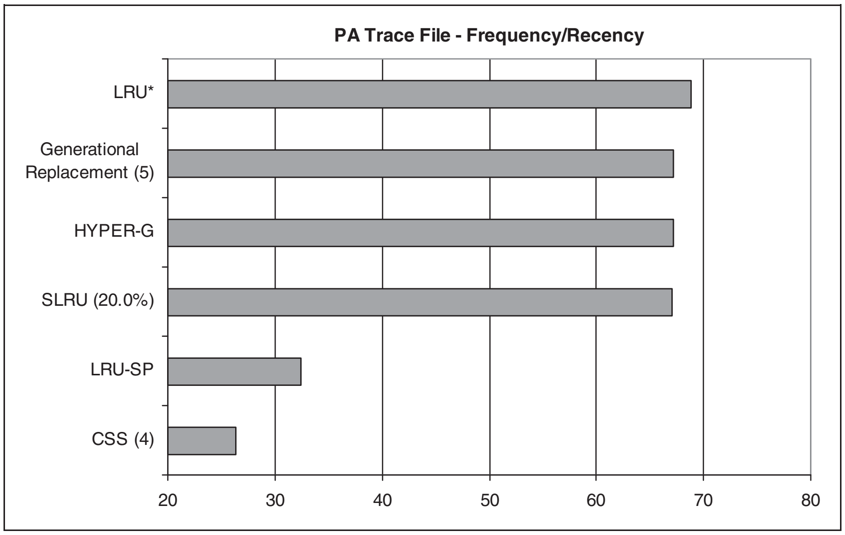

Figure 40. Removal rate frequency/recency using the PA trace file.

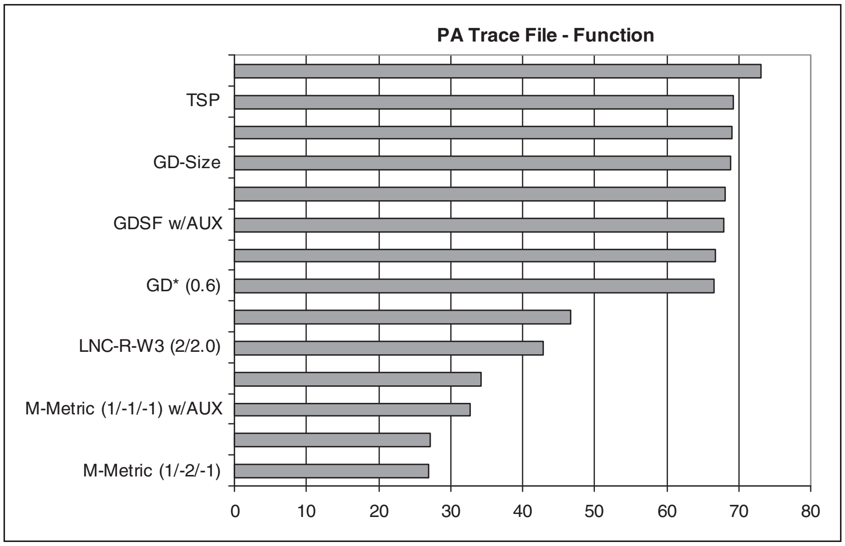

Figure 41. Removal rate for function using the PA trace file.

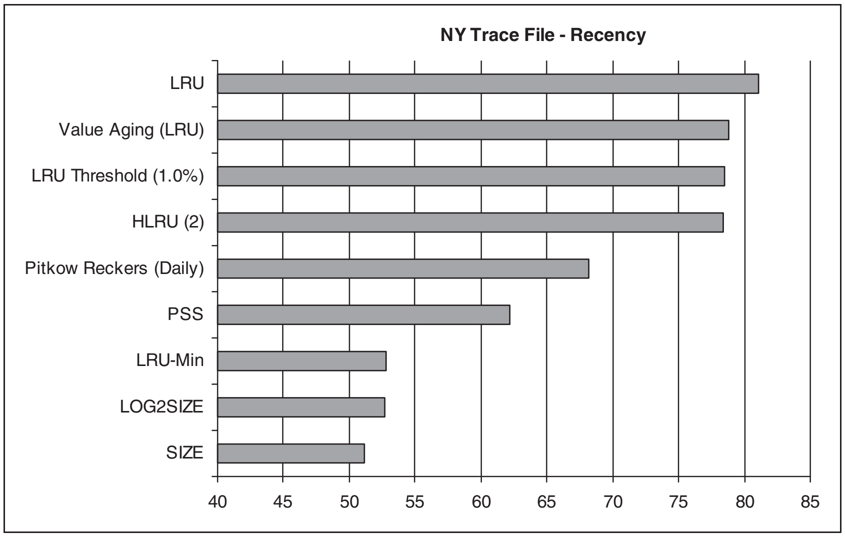

Figure 42. Removal rate for recency using the NY trace file.

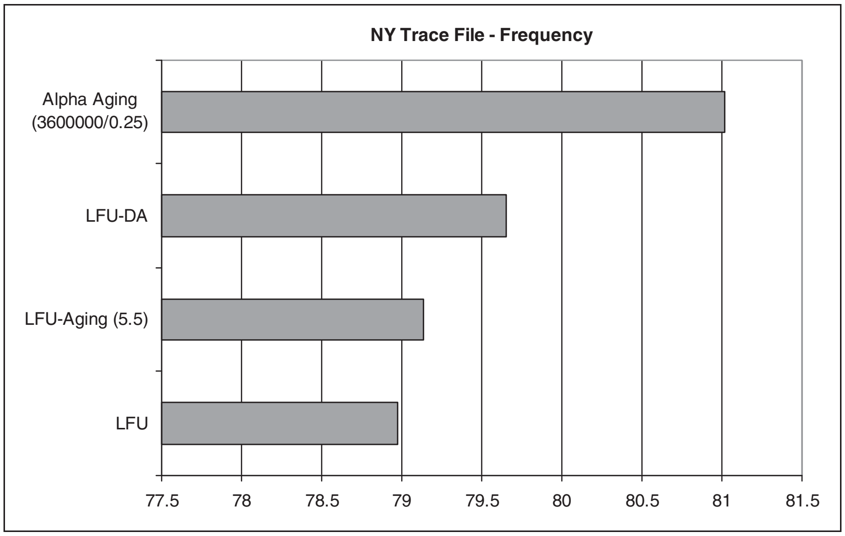

Figure 43. Removal rate for frequency using the NY trace file.

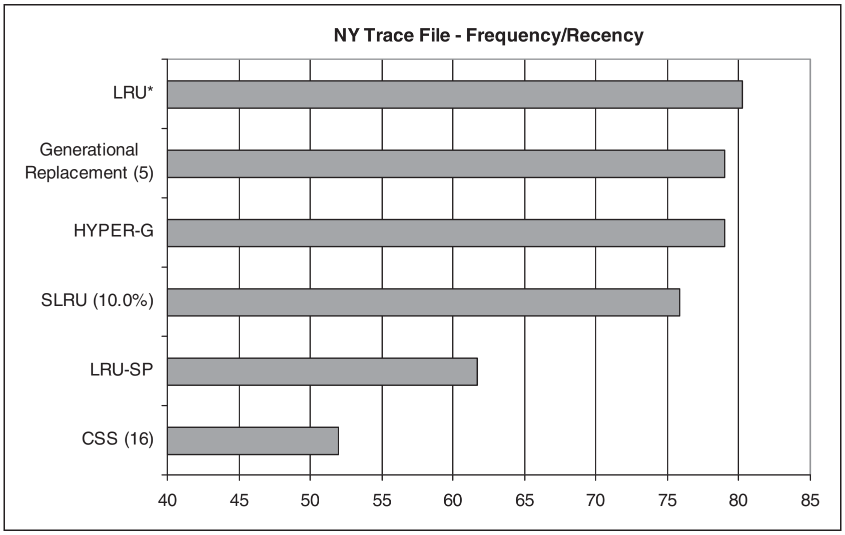

Figure 44. Removal rate for frequency/recency for the NY trace file.

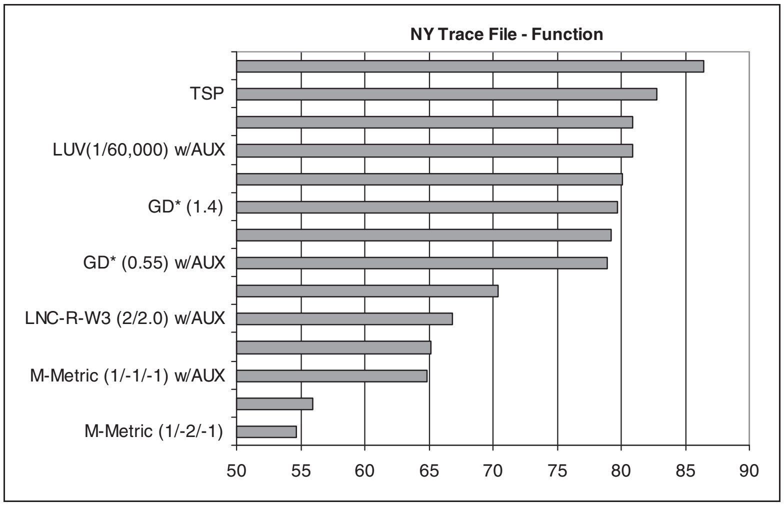

Figure 45. Removal rate for function using the NY trace file.

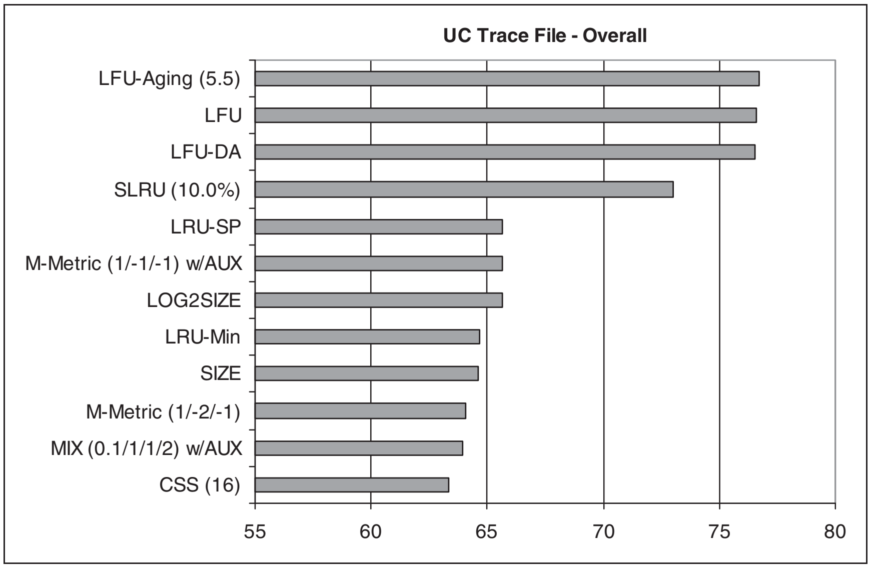

Figure 46. Removal rate for overall using the UC trace file.

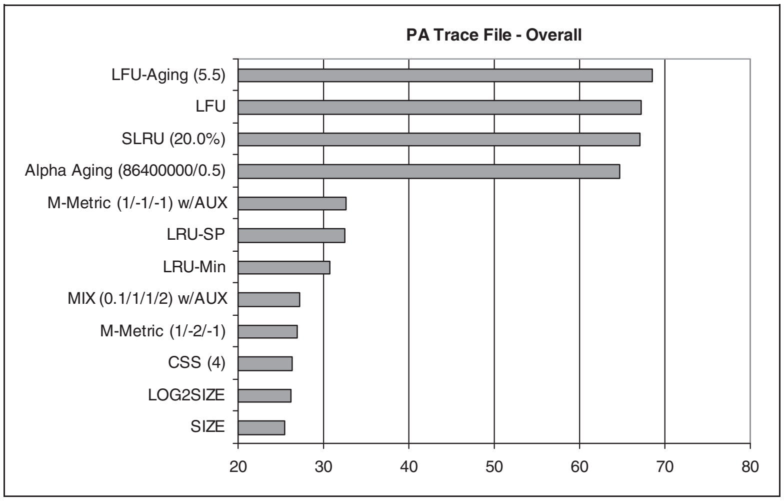

Figure 47. Removal rate for overall using the PA trace file.

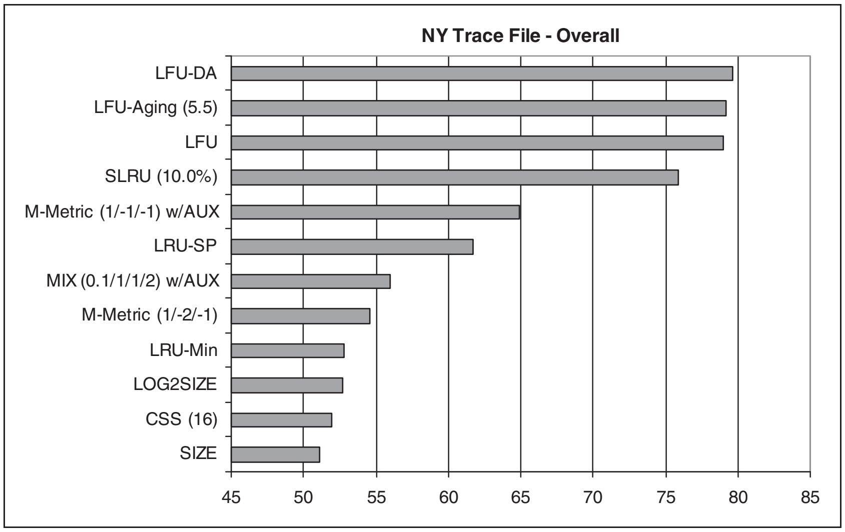

Figure 48. Removal rate for overall using the NY trace file.

utilized parameters needed further analysis to see how different combinations of parameters affect the overall functionality. We discussed how certain attributes about the request streams affect certain strategies comparatively. Several explanations were provided detailing various performance issues of the strategies individually and as a category. We also demonstrated that the sparseness of request streams has a large effect on an algorithm's performance and that some algorithms thought to be resilient to temporal locality, such as many function-based methods, were far more sensitive than previously thought.

The Squid proxy server provides a choice between LFU-DA, GDSF, a heap-based variation of LRU, and by default enables a linked-list version of its LRU variant. Based on the research presented in this paper, it is obvious that algorithms such as PSS, CSS, M-Metric, MIX, and GDSF would allow system administrators greater control over their proxy servers. System administrators should configure Squid to use one of the more advanced strategies, GDSF or LFU-DA, instead of the default LRU that was clearly demonstrated to perform the worst consistently in our research.

Most of the strategies we covered are relatively simple to implement and incorporate a relative low CPU and space overhead and should be deployed in commercial proxy cache servers to allow system administrators and ISPs greater control over the QoS of their services. Our results are particularly important in ad-hoc wireless networks where mobile devices have limited cache size.

8. Future Work

Cache replacement strategies that work in collaboration with multiple proxy servers should also be researched. Cooperation of multiple clusters could lead to a generally faster Web on the whole. Replacement strategies based on neural networks, genetic algorithms and fuzzy logic should also be analyzed in hopes of devising strategies that are resistant to changes in the request stream.

Cache placement strategies should be surveyed in a similar way as replacement strategies were. Many models are based on probability and heuristics, also incorporating expiration, and other general characteristics that are not normally considered in replacement strategies. Once analyzed, combinations of placement strategies with replacement strategies can be studied to find how these two decisions affect one another.

Likewise, cache replacement strategies dealing with dynamic Web content should be surveyed and simulated in a similar manner. As the Web evolves, new dynamic strategies will need to be devised and this paper can be used as the basis to analyze how these replacement strategies perform.

Funding

This research has been partially funded by a SURE grant from Stetson University.

Acknowledgment

The authors are very grateful to the three anonymous reviewers and the associate editor for their comments and critique which improved the quality of this paper.

References

1. Gill P, Arlitt M, Li Z and Mahanti A. YouTube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, San Diego, CA, 2007, pp. 15–28.

2. Abhari A and Soraya M. Workload generation for YouTube. *Multimedia Tools Appl* 2010; 46: 91–118. DOI 10.1007/s11042-009-0309-5.
3. Podlipnig S and Boszormenyi L. A survey of Web cache replacement strategies. *ACM Comput Surveys* 2003; 35: 374–398.
4. Balamash A and Krunz M. An overview of Web caching replacement algorithms. *Commun Surveys Tutorials* 2004; 6: 44–56.
5. Wong K. Web cache replacement policies: A pragmatic approach. *IEEE Network* 2006; 20: 28–34.
6. Fagni T, Perego R, Silvestri F and Orando S. Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans Inform Syst* 2006; 24: 51–78.
7. Romano S and ElAarag H. A quantitative study of recency and frequency based Web cache replacement strategies. In *11th Communication and Networking Symposium (CNS.08), Spring Simulation Multiconference*, Ottawa, Canada, 14–17 April 2008, pp. 70–78.
8. Hypertext Transfer Protocol—HTTP/1.1, 14 August 2007. Available at: http://www.w3.org/Protocols/rfc2616/rfc2616.html.
9. Gan Q and Suel T. Improved techniques for result caching in Web search engines. In *Proceedings of the 18th International Conference on World Wide Web*, Spain, 2009, pp. 431–440.
10. Pallis G, Vakali A and Pokorny J. A clustering-based prefetching scheme on a Web cache environment. *Comput Electrical Engng* 2008; 34: 309–323.
11. Kumar C and Norris JB. A new approach for a proxy-level web caching mechanism. *Decision Support Syst* 2008; 46: 52–60.
12. Chiang IR, Goes P and Zhang Z. Periodic cache replacement policy for dynamic content at application server. *Decision Support Syst* 2007; 43: 336–348.
13. Kastaniotis G, Maragos E, Dimitsas V, Douligeris C and Despotis DK. Web proxy caching object replacement: frontier analysis to discover the 'good-enough algorithms'. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, October 2007, pp.132-137.
14. Ortiz JPG, Ruiz VG, Lopez MF and Garcia I. Interactive transmission of JPEG2000 images using Web proxy caching. *IEEE Trans Multimedia* 2008; 10: 629–636.
15. Liu J-C and Xu J-L. Proxy caching for media streaming over the Internet. *IEEE Commun Mag* 2004; 42(8): 88–94.
16. Luo Q and Naughton J. Form-based proxy caching for database-backed web sites: Keywords and functions. *VLDB J* 2008; 17: 489–513.
17. Houtzager G, Jacob C and Williamson C. An evolutionary approach to optimal Web proxy cache placement. In *IEEE Congress on Evolutionary Computation* 2006.
18. Aguilar J and Leis EL. A coherence-replacement protocol for web proxy cache systems. *Int J Comput Appl* 2006; 28: 12–18.
19. Kaya CC, Zhang G, Tan Y and Mookerji V. An admission-control technique for delay reduction in proxy caching. *Decision Support Syst* 2009; 46: 594–603.
20. Goyal V, Sanyal S and Agrawal DP. Vcache: Caching dynamic documents. In *CoRR,* 2010.
21. Kim H-C, Lee D, Chon K, Jang B, Kwon T and Choi Y. Performance impact of large file transfer on Web proxy caching: A case study in a high bandwidth campus network environment. *J Commun Networks* 2010; 12(1): 52–65.
22. Cobb W and ElAarag H. Web proxy cache replacement scheme based on back propagation neural network. *J Syst Software* 2008; 81: 1539–1558.
23. Romano S and ElAarag H. A neural network proxy cache replacement strategy and its implementation in the Squid proxy server. *Neural Comput Appl* Vol. 20, 2011, pp. 59–78, Springer-Verlag.
24. Sabegi M and Yaghmaee M. Using fuzzy logic to improve cache replacement decisions. *Int J Comput Sci Network Security* 2006; 6(3A): 182–188.
25. Calzarossa M and Valli G. A fuzzy algorithm for Web caching. *Simulation* 2003; 35: 630–636.
26. Venketesh P and Venkatesan R. A survey on applications of neural networks and evolutionary techniques in web caching. *IETE Tech Rev* 2009; 26: 171–180.
27. Katsaros D and Manolpoulos Y. Web caching in broadcast mobile wireless environments. *IEEE Internet Comp* 2004; 8(3): 37–44.
28. Kumar N, Gangopadhyay A and Karabatis G. Supporting mobile decision making with association rules and multi-layered caching. *Decision Support Syst* 2007; 43: 16–30.
29. Lewycky N, Benhan B and Abhari A. Improving the performance of the Squid proxy cache. In *9th Communications and Networking Simulation Symposium (CNS 2006)*, Huntsville, AL, 2–6 April 2006.
30. Serbinski AA and Gusic. Improving the performance of Apache web server. In *Proceedings of the 2007 Spring Simulation Multiconference*, Volume 1, 2007, pp.166–169.
31. IRCache Home. Available at: http://www.ircache.net/
32. Peng G. *CDN: Content Distribution Network*. Technical Report TR-125, Experimental Computer Systems Lab, Stony Brook University, February 2008.
33. Su A-J, Choffnes DR, Kuzmanovic A and Bustamante FE. Drafting behind Akamai: Inferring network conditions based on CDN redirections. *IEEE/ACM Trans Networking* 2009; 17(6): 1752–1765.
34. Akamai CDN. Available at: http://www.akamai.com.
35. Badam A, Park KS, Pai VS and Peterson LL. HashCache: cache storage for the next billion. In *NSDI '09: 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.
36. Kangasharju J, Roberts J and Ross K. Object replication strategies in content distribution networks. *Comput Commun* 2002; 25: 376–383.
37. Doyle RP, Chase JS, Gadde S and Vahdat AM. The trickle-down effect: Web caching and server request distribution. *Comput Commun* 2002; 25: 345–356.
38. Gadde S, Chase J and Rabinovich M. Web caching and content distribution: A view from the interior. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

39. Davison B. A Web caching primer. *IEEE Internet Comput* 2001; 5(4): 38–45.
40. Abrams M, Standridge CR, Abdulla G, Williams S and Fox E. Caching proxies: limitations and potentials. In *Proceedings of the 4th International World Wide Web Conference*, 1995.
41. Pitkow J and Recker M. A simple yet robust caching algorithm based on dynamic access patterns. In *Proceedings of the 2nd International World Wide Web Conference*, 1994, pp. 1039–1046.
42. Williams S, Abrams M, Standridge CR, Abdulla G and Fox EA. Removal policies in network caches for world-wide web documents. In *Proceedings of ACM SIGCOMM*. New York: ACM Press, 1996, pp.293–305.
43. Zhang J, Izmailov R, Reinniger D and Ott M. Web caching framework: Analytical models and beyond. In *Proceedings of the IEEE Workshop on Internet Applications*. Piscataway, NJ: IEEE Computer Society, 1999.
44. Vakali A. Proxy cache replacement algorithms: A history-based approach. *World Wide Web* 2001; 4: 277–297.
45. Aggarwal CC, Wolf JL and Yu PS. Caching on the World Wide Web. *IEEE Trans Knowedge Data Eng* 1999; 11: 94–107.
46. Arlitt MF, Cherkasova L, Dilley J, Friedrich RJ and Jin TY. Evaluating content management techniques for Web proxy caches. *ACM SIGMETRICS Performance Evaluation Rev* 2000; 27: 3–11.
47. Arlitt M. *A Performance Study of Internet Web Servers*. MSc Thesis: University of Saskatchewan, 1996.
48. Arlitt MF, Friedrich RJ and Jin TY. *Performance Evaluation of Web Proxy Cache Replacement Policies*. Technical Report HPL-98-97(R.1), Hewlett-Packard Company, Palo Alto, CA, 1999.
49. Osawa N, Yuba T and Hakozaki K. Generational replacement schemes for a WWW proxy server. In *High-Performance Computing and Networking (HPCN'97)* (*Lecture Notes in Computer Science*, Vol. 1225). Berlin: Springer-Verlag, 1997, pp. 940–949.
50. Chang C-Y, McGregor T and Holmes G. The LRU* WWW proxy cache document replacement algorithm. In *Proceedings of the Asia Pacific Web Conference*, 1999.
51. Tatarinov I. *An Efficient LFU-like Policy for Web Caches*. Technical Report NDSU-CSORTR-98-01, Computer Science Department, North Dakota State University, Wahpeton, ND, 1998.
52. Cheng K and Kambayashi Y. A size-adjusted and popularity-aware LRU replacement algorithm for Web caching. In *Proceedings of the 24th International Computer Software and Applications Conference (COMPSAC)*. Piscataway, NJ: IEEE Computer Society, 2000, pp.48–53.
53. Cao P and Irani S. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 193–206.
54. Jin S and Bestavros A. GreedyDual*: Web caching algorithms exploiting the two sources of temporal locality in Web request streams. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop,* 2000.
55. Yang Q, Zhang HH and Zhang H. Taylor series prediction: A cache replacement policy based on second-order trend analysis. In *Proceedings of the 34th Hawaii International Conference on Systems Sciences*. Piscataway, NJ: IEEE Computer Society, 2001.
56. Niclausse N, Liu Z and Nain P. A new efficient caching policy for the World Wide Web. In *Proceedings of the Workshop on Internet Server Performance* 1998, pp. 119–128.
57. Wessels D. *Intelligent Caching for World-Wide-Web objects*. Boulder, CO: MSc thesis, University of Colorado at Boulder, 1995.
58. Scheuermann P, Shim J and Vingralek R. A case for delay-conscious caching of Web documents. In *Proceedings of the 6th International WWW Conference*, 1997.
59. Bahn H, Koh K, Min SL and Noh SH. Efficient replacement of nonuniform objects in Web caches. *IEEE Computing* 2002; 35: 65–73.
60. Hosseini-Khayat S. *Investigation of Generalized Caching*. PhD Dissertation, Washington University, St Louis, MO, 1997.
61. Sajeev GP and Sebastian MP. A novel content classification scheme for web caches. *Evolving Syst* 2010; 2(2): 101–118.

Sam Romano is a software engineer at Lockheed Martin Corporation, Orlando, Florida. He obtained his BSc in computer science from Stetson University in 2008. He has multiple publications in the area of Web proxy cache replacement. His current research interests include artificial intelligence, knowledge base and rule engines, task scheduling problem analysis with algorithm design, concurrent performance on distributed system architectures, and Web application software architecture. His work at LMCO is mostly based on logistics and training Web applications for more thorough management in military logistics and training programs across the world.

Hala ElAarag is an associate professor of computer science at Stetson University, DeLand, Florida. She received her PhD degree in Computer Science from the University of Central Florida in 2001, her MSc and BSc from Alexandria University in 1991 and 1989, respectively. Her research interests include computer networks, operating systems, evolutionary computation, modeling and simulation and performance evaluation. She has numerous publications in prestigious international journals and conference proceedings. She obtained Stetson University research award in 2005 and Best Paper Award at the 11th Communication and Networking Symposium (CNS'08). She was co-general chair of Communication and Networking Simulation Symposium 2009. She is Program Chair and board member of Consortium of Computing Sciences in Colleges Southeastern Region. She is Vice General Chair and General Chair of Society for Modeling and Simulation International (SCS) Spring Simulation Multiconference 2011 and 2012, respectively. She serves on the technical committee for many international conferences and reviews for multiple journals