

A Quantitative Study of Recency and Frequency based Web Cache Replacement Strategies

Sam Romano and Hala ElAarag
Department of Mathematics and Computer Science
Stetson University
421 N Woodland Blvd, Deland, FL 32723
{sromano,helaarag}@stetson.edu

Keywords: Cache replacement, Simulation, Web, Proxy Cache

Abstract

There are many replacement strategies to consider when designing a web cache server. The most commonly known cache replacement strategies are Least Frequently Used (LFU) and Least Recently Used (LRU). Though comprehensive surveys exist, no known study has presented comparative performance measures of these strategies together. We will review and describe proxy cache replacement strategies based on the recency, frequency, and size attributes of web objects, and present two performance measures. Simulation results ranked by the two performance metrics, hit rate and byte hit rate, reveal strong inductions about current cache replacement strategies.

1. INTRODUCTION

The Web has become the single most important source of information and communication for the world. Proxy servers utilize the process of caching to reduce user (client) perceived lag and loads on the origin servers [1, 2]. Our main focus will be the cache replacement problem, the process of evicting objects in the cache to make room for new objects.

There are many replacement strategies to consider when designing a web cache server. The most commonly known cache replacement strategies are Least Frequently Used (LFU) and Least Recently Used (LRU). Until 2003, there had been no survey of known web cache replacement strategies. However, Podlipnig et al. [1] has done well to not only list well-known strategies, but also classify the strategies into different categories. However, there is not a known record of results comparing the majority of the algorithms presented by Podlipnig. Previous works compared results for different strategies against, at most, five other strategies. In this paper, we present quantitative comparisons of 19 different cache replacement strategies from three categories.

The rest of our paper is structured as follows. In section 2, we describe the classifications presented by Podlipnig et al [1], and a brief description of each strategy we simulated. Section 3 covers the performance metrics we used. Section

4 lists the results and observations of our simulation. We then summarized our paper in section 5.

2. REPLACEMENT STRATEGIES

There are many characteristics for web objects. Among them, recency, frequency and size are considered the three most important ones. 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. Due to this unique nature of the strategies surveyed by Podlipnig et. al [1], and used in this paper, there is a fairly clear classification.

The first two groups, *Recency* and *Frequency*, are based mainly on Least-Recently Used (LRU) and Least-Frequently Used (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*, picks an object in a nondeterministic method. Due to this inconsistent nature of the last category, we decided not to include it in our study. In this paper, we provide a comprehensive study of *Recency*, *Frequency* and *Recency/Frequency* algorithms. Due to space limitation, function based strategies and their results were not included in this paper.

Table 1 contains commonly used variables and their descriptions. Any use of the logarithmic function symbolized as *log*, is assumed to be of base 2.

Table 1: *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 requested
F_i	Frequency counter of object i
ΔF_i	Number of references to occur since last time object i was referenced
C_i	Cost of object i
R_i	Request value of object i
M	Size of the cache

2.1 Recency Based Strategies

This set of strategies are 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 [2]. They use recency information as a significant part of their victim selection process. Recency based strategies are typically straight forward to implement taking advantage of queues and linked lists.

1. LRU: One of the most commonly used strategies in many areas of cache management. This algorithm removes the least recently referenced object.
2. LRU-Threshold [4]: 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 [5]: 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 [6]: Removes the largest object first. If objects are of the same size, then their tie is broken by LRU.
5. LOG2-SIZE [1]: Sorts objects by their $\text{floor}[\log(S_i)]$, differentiating objects with the same value by LRU. This strategy tends to invoke LRU more often between similar-sized objects as compared to SIZE.
6. LRU-Min [4]: This strategy attempts to remove as little 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 works as follows:
 - a. Set T to S_i of the object being admitted to the cache.
 - b. Set L to L_o .
 - c. 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.
 - d. If the end of the list is reached, then divide T by 2 and repeat the process from step b.
7. Value-Aging [7]: 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:

$$R_i = C_t * \sqrt{\frac{C_t}{2}} \quad (1)$$

At each request, R_i is updated to:

$$R_i = C_t * \sqrt{\frac{C_t - T_i}{2}} \quad (2)$$

8. HLRU [8]: Standing for History LRU, this strategy uses a sliding window of h request times for objects. This strategy requires additional information to be held for each object, even after the object has been removed from the cache. The *hist* value is defined for an object x with n indexed request times, t_i , where t_i is equivalent to the i th request time of object x .

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

HLRU chooses the object with the maximum *hist* value. If multiple objects have *hist* values of 0, then they are sorted based on LRU.

9. Pyramidal Selection Scheme (PSS) [9]: 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 $j = \text{floor}[\log(S_i)]$. There are $N = \text{ceil}[\log(M + 1)]$ 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 a value defined as $S_i * \Delta F_i$.

2.2 Frequency Based Strategies

Frequency based strategies use a property of request streams known as *spatial locality*, the likelihood that an object will appear again based on how often it’s been seen before [2]. Unlike Recency-based strategies, these simple algorithms require complex data structures, such as binary heaps to help decrease the time overhead in making their decisions.

Most of these strategies are an extension of the commonly known algorithm *Least Frequently Used* (LFU). There are two ways to implement these algorithms, one requiring the 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*.

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

1. LFU: The base algorithm of this class, removes the least-frequently used object (or object with the smallest frequency counter).
2. LFU-Aging [10]: 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 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.

3. LFU-DA [10]: Since the performance of LFU-Aging requires the right threshold and maximum frequency, LFU-DA tries to avoid this problem. Upon a request to object i , its value, K_i , is calculated as:

$$K_i = F_i + L \quad (4)$$

where L , is a dynamic aging factor. Initially L is set to 0, 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.

4. α -Aging [7]: 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:

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

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

2.3 Frequency/Recency Based Strategies

These strategies attempt to combine both *spatial* and *temporal locality* together maintaining their characteristics of the previous two classes. As a result, they tend to be fairly complex in their structure and procedures.

1. Segmented LRU (SLRU) [11]: This strategy partitions the cache into a two-tier system. One segment is known as the *unprotected* segment and the other, the *protected* segment. The strategy requires space set aside for the protected segment, known as A . Objects that belong to this segment cannot be removed from the cache once added. Both segments are managed by 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 $\min(A, M - A)$. 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 space in the protected segment, the LRU strategy is applied to the protected segment, moving objects into the unprotected segment.
2. Generational Replacement [12]: This strategy uses n LRU lists, where $n > 1$. Upon being added to the cache, an object is added to the head of the first list. Upon a

cache hit, an object belonging to list i is moved to the head of list $i+1$, unless it's the last list, then the object is moved to the head of that list. Victim selection begins at the end of list 1, and moves to the next consecutive list only when preceding lists have been depleted.

3. LRU* [13]: This method combines an LRU list and what is known as a "request" [1] counter. 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 1 and moved to the front of the list.
4. HYPER-G [6]: This strategy combines LRU, LFU and SIZE. First, the least frequently used object is chosen. If there is more than one object with the same frequency value, the cache chooses the LRU object among them. If this still does not give a unique object to replace, the largest object is chosen.
5. Cubic Selection Scheme (CSS) [14]: As the name implies, CSS uses a cube-like structure to select its victims. Like PSS, CSS assigns objects to classes, indexed by size and frequency. Each class, like PSS, is an LRU list. Objects in a class (j, k) have sizes and frequencies ranging from $2^{(j, k)-1}$ to $2^{(j, k)} - 1$. The width, which is the largest value of j is the same as in PSS, since it is based on cache size. But, in order to limit space overhead, there must also be a maximum frequency, $MaxF$, set to limit the height of the cube. 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.
6. LRU-SP [15]: Like PSS, this class utilizes classes managed by LRU and has the same number of classes as PSS. However, this class accounts for frequency counts as well. An object i is assigned to class $j = \text{floor}[\log(S_i / F_i + 1)]$. Essentially, as an object is requested more frequently, it decreases the class it is in. When a victim is to be selected, all the LRU objects of each list are compared based on the value $(\Delta T_i * S_i) / F_i$.

3. PERFORMANCE METRICS

Performance metrics are designed to measure different functional aspects of the web cache. Hit-rate and byte hit-rate are by far the most common and stable measurements of web cache performance as a whole.

- *Hit-rate*: This metric is used to measure all generic forms of caching. This is simply the number of *cache hits* that occur to the total number of *cacheable requests* seen by the proxy.

- *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 rate is:

$$\frac{\sum_{i=0}^n h_i}{\sum_{i=0}^n r_i} \quad (6)$$

One might expect that hit-rate and byte-hit rate cannot be optimized for at the same time. No cache replacement strategy can provide the best results for both metrics 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.

4. EXPERIMENT SETUP AND RESULTS

4.1 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 [16]. IRCache gathers their trace files and other data on web caching from several different proxy servers located around the United States. The trace files each captured an entire week's worth of web requests from different locations around the United States. These particular trace files were chosen due to their differences in size and cacheable request rates.

Table 2 presents statistics about the three traces we used for our simulation. Each trace represented varying levels of temporal locality, spatial locality, total bandwidth and number of requests testing the various limits of the replacement strategies. Unique Requests describe the number of unique URLs that were seen in the trace file. Knowing the Unique *Cacheable* Requests, one can calculate the upper bound for hit-rate by subtracting the unique requests from unique cacheable requests.

Table 2. Trace File Statistics for Requests and Bandwidth

Trace File	Urbana-Champaign, Illinois (UC)	New York, New York (NY)	Palo Alto, California (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 Requests	73.78 % of Unique Requests	73.71 % of Unique Requests	33.89 % of Unique Requests

4.2 Simulation Setup and Parameters

Some of the strategies presented in section 2 had one or more parameters. Table 3 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 4, 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 3 also shows the order these parameters are listed in the graphs of section 4. For example, in Figure 2, AlphaAging(3600000/0.25) means that α - Aging performed best with Interval=3600000 ms (or 1 hour) and $\alpha = 0.25$.

The implementation language we decided to use was Java 6.0. Though concerned with speed, Java performed well on the simulation machine in Ubuntu 7.04. Lastly, Java's *Hashtable* and *PriorityQueue* classes supplied the most functionality. Upon handling a request, the web cache interface would attempt to add the object to the cache. If there was not enough space to add it, it would run the replacement strategy until there was just enough space.

We ran the simulations with different cache sizes. We started at 50 MB (megabytes), then 100 MB, and finally ended with 200 MB. As the cache size increased, all the replacement strategies performed better respective to each metric. However, the general increase of performance did not significantly change the ranking indexed by a particular metric in any of the simulations. For this reason, we will 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.

Table 3. Order and Description of Parameters in Results

Strategy	Parameters
<i>LRU-Threshold</i>	<i>Threshold</i> : The maximum size threshold of the cache.
<i>Pitkow/Reckers Strategy</i>	<i>Interval</i> : Interval set to either daily or hourly describing when objects are differentiated by size.
<i>HLRU</i>	<i>h</i> : The hist-value to use in a sliding window of h-requests.
<i>LFU-Aging</i>	<i>Average Frequency Threshold</i> : The aging factor. <i>Maximum Frequency Threshold</i> : The maximum frequency counter of any given object.
α - Aging	<i>Interval</i> : Interval, in milliseconds, of when the aging factor is applied. α : The aging factor.
<i>Segmented - LRU</i>	<i>Protected Segment</i> : The size, in percent of the total cache size of the protected segment.
<i>Generational Replacement</i>	<i>Generations</i> : Number of generations used.
<i>Cubic Selection Scheme</i>	<i>Max Frequency</i> : The maximum frequency counter, always a power of 2.

4.3 Simulation Results

Due to space limitations, we did not include the results for the NY trace file when testing the performance of each category since they produced similar results to the UC trace files for both performance metrics. However, we included the NY trace files when performing the overall comparisons of the three categories.

4.3.1 Hit Rate

Figures 1-9 show the hit-rates for our simulations. Figures 1-3 shows the results for the recency, frequency and recency/frequency categories, respectively, using the UC trace file. Figures 4-6 shows the results for the three categories using the PA trace. Figures 7-9 show the overall comparison of all strategies selecting the best three from each category, using The UC, PA and NT trace files,

respectively. Results of the recency category for the UC trace in Figure 1 have a smaller variance compared to the PA trace in Figure 4, demonstrating the effect of its high request rate.

In the recency category PSS performed the best for UC and PA traces as shown in Figures 1 and 4, demonstrating that the incorporation of size classes did in fact give it a higher edge than just considering size alone. However, the gain from its complicated decision process is questionable, as shown in the aforementioned figures, because simple algorithms such as SIZE performed almost just as well. Also in Figures 1 and 4, 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.

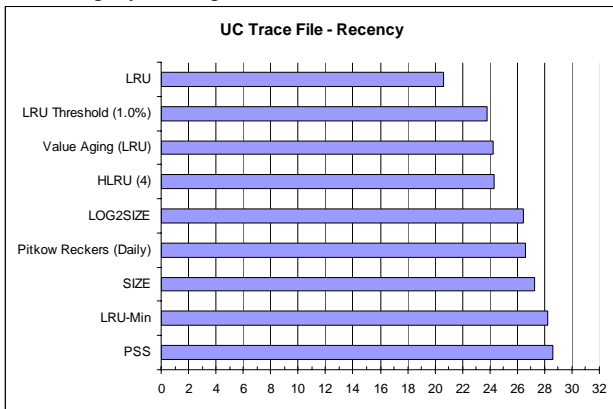


Figure 1. Hit-Rate for Recency using UC Trace File

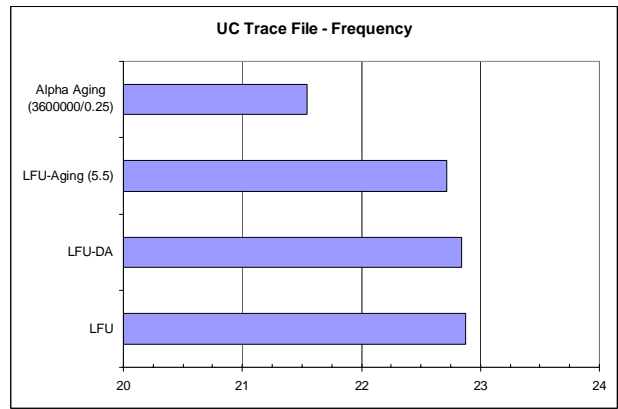


Figure 2. Hit-Rate for Frequency using UC Trace File

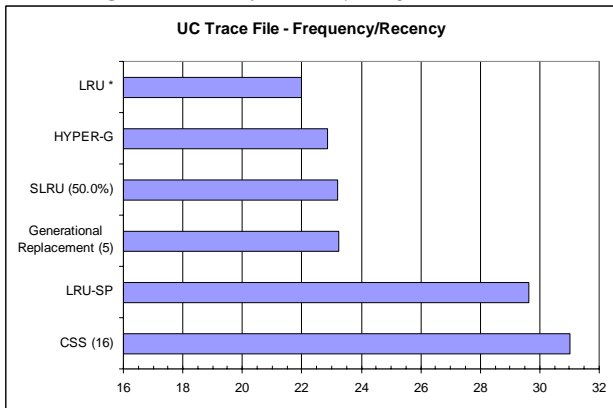


Figure 3. Hit-Rate for Frequency/Recency using UC Trace File

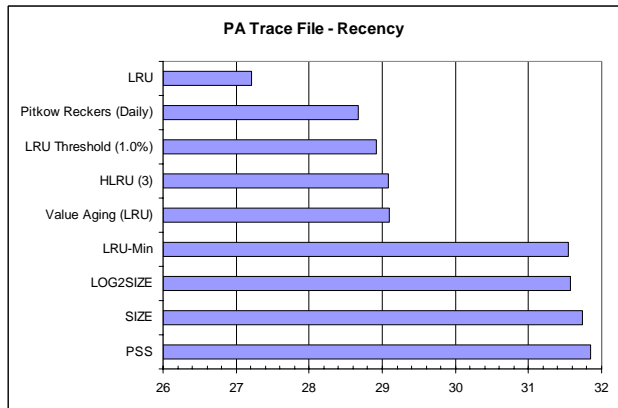


Figure 4. Hit-Rate for Recency using the PA Trace File

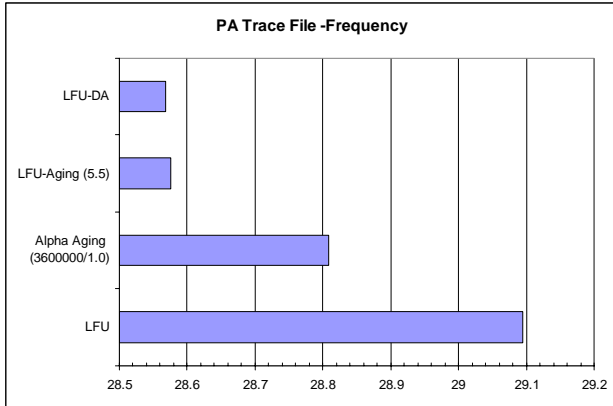


Figure 5. Hit-Rate for Frequency using the PA Trace File

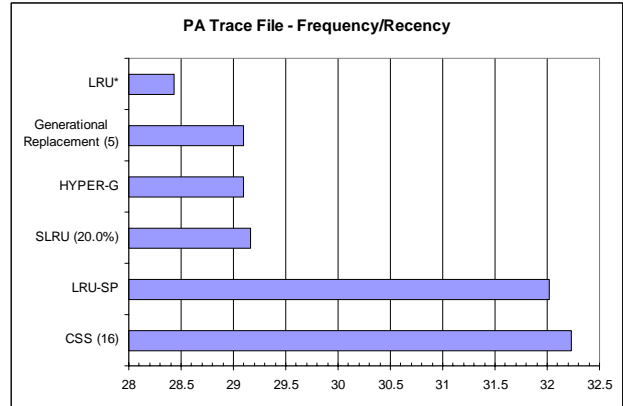


Figure 6. Hit-Rate for Frequency/Recency using the PA Trace File

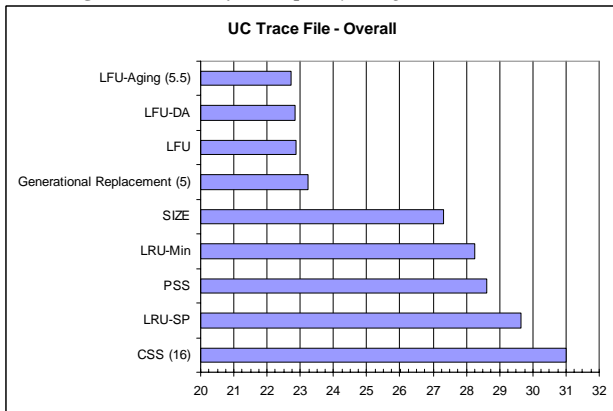


Figure 7. Hit-Rate for Overall using UC Trace File

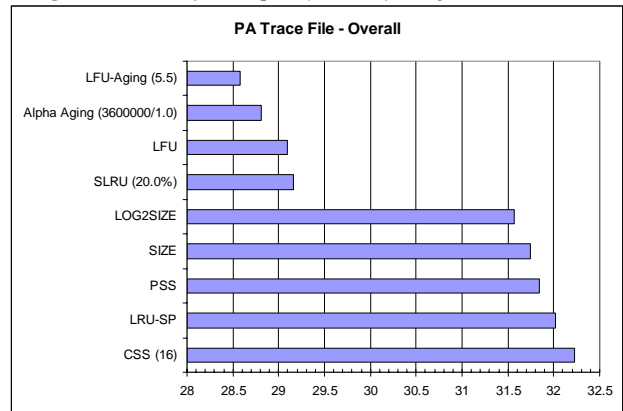


Figure 8. Hit-Rate for Overall using PA Trace File

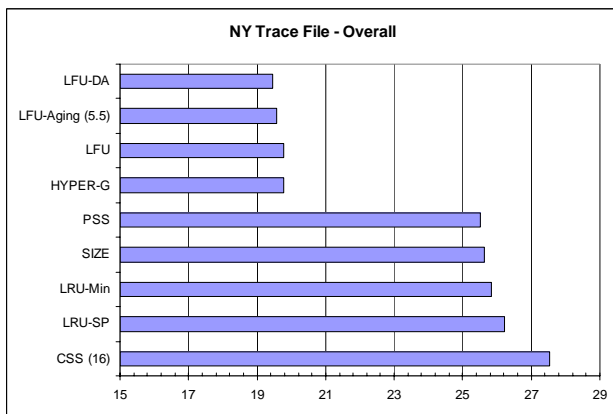


Figure 9. Hit-Rate for Overall using NY Trace File

One can also notice from these 2 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; what is important to note is that when recency (LRU) is used as a base, derivative algorithms will generally do far better.

This observation, however, does not apply to the Frequency-based strategies. LFU as shown in Figures 2, and 5, always outperformed its derivative 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, though 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. Due to this flaw, the Frequency strategies are always outperformed by the other categories' best in the overall charts as shown in Figures 7-9.

From figures 3, and 6, it is clear that for Frequency/Recency strategies, LRU-SP and CSS did the best consistently. Though it is not displayed here, CSS for any parameter generally did the same with an incredibly small variance (this is also true for the byte-hit rate metric as well). LRU-SP generally did as well as PSS or a little better. With the exception of HYPER-G, all the algorithms did outperform LRU in hit-rate, holding our earlier observation valid.

Overall, Figures 7-9 show that CSS outperformed all other strategies consistently by utilizing a strong balance between size, frequency and recency information to make its decisions. When modified from LRU and SIZE, recency strategies clearly outperformed the frequency strategies despite that LFU consistently outperformed LRU. Frequency/Recency strategies held the most consistent results, generally outperforming their respective parent strategies.

4.3.2 Byte Hit Rate

In previous literatures, 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 fact that larger web objects are accessed less often because these files are updated less frequently and have high latency to acquire. Generally the network cost to access a large 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 trade off between saving bandwidth and decreasing user perceived lag. In the latter, the users will feel the effects of the proxy cache, where as in the former, the origin servers will witness a cut in bandwidth costs.

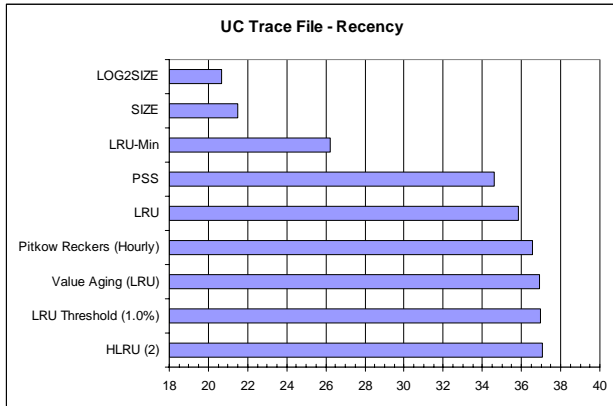


Figure 10. Byte Hit-Rate for Recency using the UC Trace File

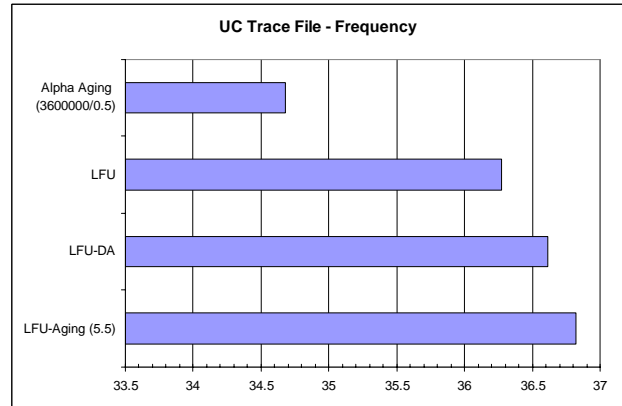


Figure 11. Byte Hit-Rate for Frequency using the UC Trace File

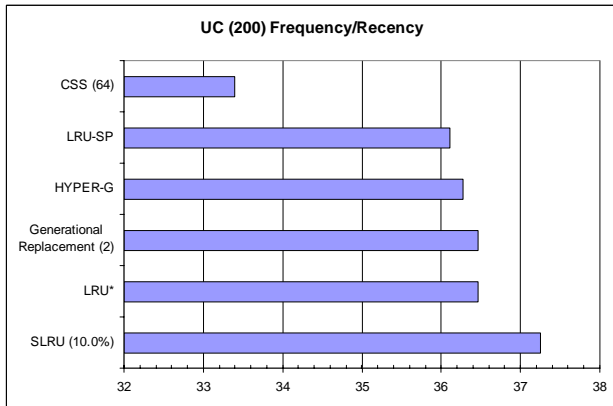


Figure 12. Byte Hit-Rate for Frequency/Recency using the UC Trace File

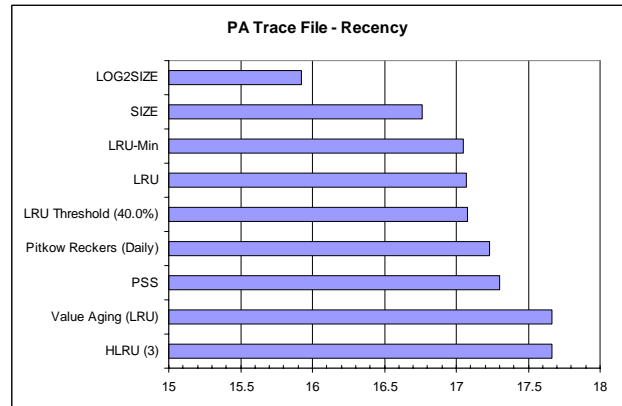


Figure 13. Byte Hit-Rate for Recency using the PA Trace File

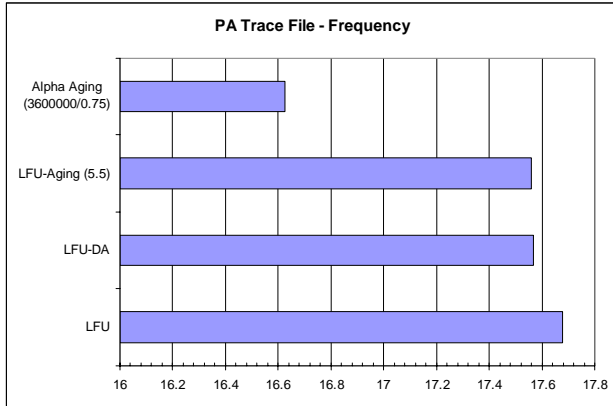


Figure 14. Byte Hit-Rate for Frequency using the PA Trace File

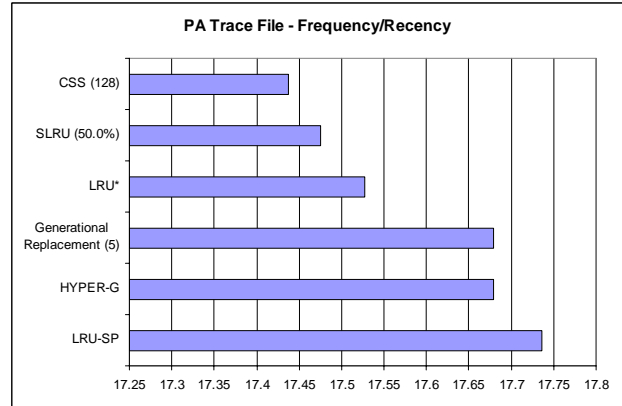


Figure 15. Byte Hit-Rate for Frequency/Recency using the PA Trace File

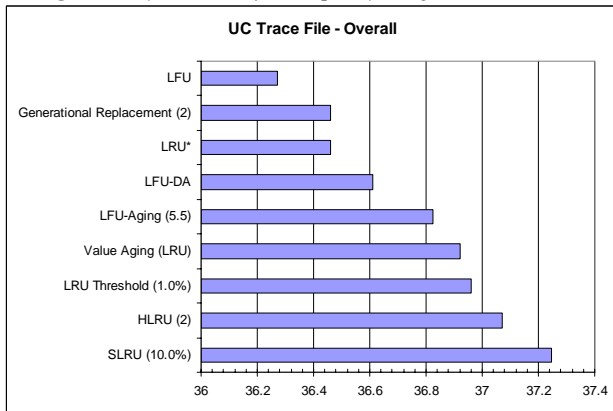


Figure 16. Byte Hit-Rate for Overall using the UC Trace File

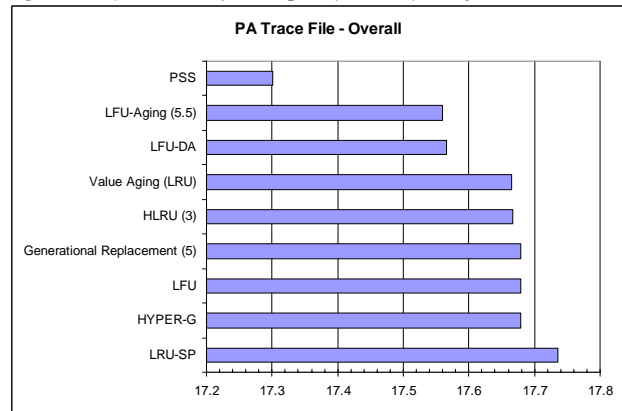


Figure 17. Byte Hit-Rate for Overall using the PA Trace File

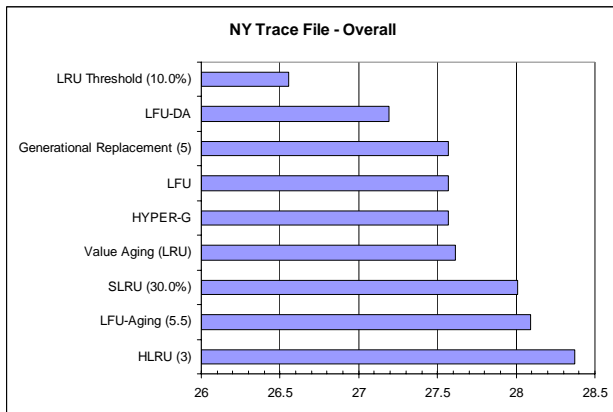


Figure 18. Byte Hit-Rate for Overall using NY Trace File

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 10. The one exception occurs in the PA trace file, Figure 13. In fact, the exception occurs again in comparison to other categories in the PA trace results of Figure 17 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.

We also observed that HLRU does well in the NY trace and UC trace, Figure 10, and also manages to do the best for the PA's Recency set, Figure 13. This may suggest that considering the rate of requests is relevant to the size of objects. Value-Aging also did well in comparison with other recency strategies, but did only mediocre overall, Figures 16-18. 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 the frequency class, the frequency-based methods did worse overall, but we cannot rule out frequency as being an irrelevant characteristic, as LFU still outperforms LRU each trace. In fact, HYPER-G, from the recency/frequency category which base its decision on frequency among other characteristics, does well consistently and inversely does better as the rate of web requests decreases.

Also, the aging factors for LFU-DA and LFU-Aging, which were a problem for hit-rates, actually work to the advantage of larger 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;

thus the objects with higher frequencies have a higher risk to being selected as victims during removal as larger objects.

In conclusion, our results for the three categories we simulated demonstrate that there is generally a trade-off between byte hit-rate and hit-rate. This is mainly due to the fact that the characteristics, recency and frequency, are generally inversely related to the object's size.

5. SUMMARY

This paper has provided an exhaustive quantitative analysis of cache replacement strategies based on two metrics. A comprehensive review of the Recency, Frequency, and Frequency/Recency based strategies was also included. We have provided several explanations of the results detailing various performance issues of the strategies individually and compared to other strategies. By simulating different instances of strategies such as α -Aging, and CSS, we have demonstrated how the parameters can be fine tuned to obtain the best results for a certain metrics for that particular strategy. Comparing strategies within each of their categories and overall, we have demonstrated the trade-off between byte hit-rate and hit-rate when considering an object's recency, frequency and size characteristics.

We also demonstrated that commonly used methods are generally outperformed by their derivative strategies. By combining web object characteristics together, the cache replacement strategies chose better victims in their decision process. Most of the strategies we covered are relatively simple to implement and incorporate relative low CPU and space overhead and should be deployed in commercial proxy cache servers rather than the naive LRU strategy.

REFERENCES

- [1] S. Podlipnig and L. Boszormenyi, "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys*, vol. 35, no. 4, pp. 374-398, 2003.
- [2] B. Davison, "A Web Caching Primer," *IEEE Internet Computing*, vol. 5, no. 4, pp. 38-45, 2001.
- [3] "Hypertext Transfer Protocol -- HTTP/1.1" [Online Document] [Cited Aug. 14, 2007] Available. WWW: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [4] Abrams, M., Standridge, C. R., Abdulla, G., Williams, S., and Fox, E. "Caching Proxies: Limitations and Potentials," *Proceedings of the 4th International World Wide Web Conference*. 1995.
- [5] Pitkow, J. and Recker, M. "A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns," *Proceedings of the 2nd International World Wide Web Conference*, pp. 1039-1046, 1994.
- [6] Williams, S., Abrams, M., Standridge, C. R., Abdulla, G., and Fox, E. A. "Removal Policies in Network Caches for World-WideWeb Documents," *Proceedings of ACM SIGCOMM*. ACM Press, New York, NY, pp. 293-305, 1996.
- [7] Zhang, J., Izmailov, R., Reinniger, D., and Ott, M. "Web Caching Framework: Analytical Models and Beyond," *Proceedings of the IEEE Workshop on Internet Applications*. IEEE Computer Society, Piscataway, NJ. 1999.
- [8] Vakali, A. "Proxy Cache Replacement Algorithms: A History-Based Approach", *World Wide Web*, v.4 n.4, p.277-297, 2001.
- [9] Aggarwal, C. C., Wolf, J. L., and Yu, P. S. "Caching on the World Wide Web," *IEEE Trans. Knowledge. Data Eng.* 11, 1 Jan., pp. 94-107, 1999.
- [10] Arlitt, M. F., Cherkasova, L., Dille, J., Friedrich, R. J., and Jin, T. Y. "Evaluating Content Management Techniques for Web Proxy Caches," *ACM SIGMETRICS Performance Evaluation. Rev.* 27, 4 Mar., pp. 3-11, 2000.
- [11] Arlitt, M. F., Friedrich, R. J., and Jin, T. Y. "Performance Evaluation of Web Proxy Cache Replacement Policies," Tech. rep. HPL-98-97(R.1), Hewlett-Packard Company, Palo Alto, CA. 1999.
- [12] Osawa, N., Yuba, T., and Hakozaki, K. "Generational Replacement Schemes for a WWW Proxy Server," *High-Performance Computing and Networking (HPCN'97)*. Lecture Notes in Computer Science, vol. 1225. Springer-Verlag, Berlin, Germany, pp. 940-949. 1997.
- [13] Chang, C.-Y., McGregor, T., and Holmes, G. "The LRU* WWW Proxy Cache Document Replacement Algorithm," *Proceedings of the Asia Pacific Web Conference*. 1999.
- [14] Tatarinov, I. "An Efficient LFU-like Policy for Web Caches," Tech. Rep. NDSU-CSORTR-98-01, Computer Science Department, North Dakota State University, Wahpeton, ND. 1998.
- [15] Cheng, K. and Kambayashi, Y. "A Size-Adjusted and Popularity-Aware LRU Replacement Algorithm for Web Caching," *Proceedings of the 24th International Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, Piscataway, NJ, pp. 48-53. 2000.
- [16] "IRCache Home" [Online Document] [Cited Nov. 11, 2007] Available. WWW: <http://www.ircache.net/>