Least Popularity-per-Byte Replacement Algorithm for a Proxy Cache

Kyungbaek Kim and Daeyeon Park
Department of Electrical Engineering & Computer Science
Division of Electrical Engineering,
Korea Advanced Institute of Science and Technology (KAIST), Taejon, Korea
E-mail: kbkim@sslab.kaist.ac.kr and daeyeon@ee.kaist.ac.kr

Abstract

With the recent explosion in usage of the world wide web, the problem of caching web objects has gained considerable importance. The performance of these web caches is highly affected by the replacement algorithm. Today, many replacement algorithms have been proposed for web caching and these algorithms use the on-line fashion parameters But, recent studies suggest that the correlation between the on-line fashion parameters and the object popularity in the proxy cache *are weakening due to the efficient* client caches. *In this paper, we suggest a new algorithm, called* Least Popularity Per Byte Replacement*(LPPB-R). We use the popularity value as the long-term measurements of request frequency to make up for the weak point of the previous algorithms in the proxy cache and vary the popularity value by changing the impact factor easily to adjust the performance to needs of the proxy cache. And we examine the performance of this and other replacement algorithm via trace driven simulation.*

1. Introduction

The recent increase in popularity of the World Wide Web has led to a considerable increase in the amount of traffic over the Internet. As a result, the Web has now become one of the primary bottlenecks to network performance. Consequently, web caching has become an increasingly important topic. Web caching aims to reduce network traffic, server load, and user-perceived retrieval delay by replicating popular content on caches that are strategically placed within the network. Caching can be implemented at various points in the network. In the server side, there is typically a cache in the *Web server* itself and there is a cache, which behave like a *web server accelerator*. In the opposite side, there are *client caches* in the Web browsers. And it is increasingly common for a university or corporation to implement specialized servers in the network called *caching proxies* [12]. In this paper, we shall discuss the cache replacement policies designed specifically for use by proxy web caches.

In the recent studies, we can find several important reasons why many people try to enhance the performance of proxy caches by using the efficient replacement policy. First, the growth rate of Web capacity is much higher than the rate with which memory sizes for Web caches are likely to grow [12] [11]. This tell us that we need much more storage cost to get more Hit Rate by using the same replacement policy. Second, recent studies have shown that Web cache Hit Ratio and Byte Hit Ratio grow in a log-like fashion as a function of the cache size [5] [6]. Thus, a better algorithm that increases Hit Ratio and Byte Hit Rate by only several percentage points would be equivalent to a several fold increase in cache size. Third, for an infinite sized cache, the Hit Ratio for a web proxy grows in a log-like fashion as a function of the client population of the proxy and of the number of requests seen by the proxy [5] [6]. In this case, we get the same result as the result from second reason. So, if we use a better replacement policy, the proxy can serve much more clients and requests than before. Finally, the benefit of even a slight improvement in cache performance may have an appreciable effect on network traffic.

Today many replacement algorithms have been proposed for web caching. These algorithms use the on-line fashion parameters like size, temporal locality and latency, to define the object popularity, rather than the object popularity value directly from the cache. But more recent studies [4] suggest that such relationships are weakening and hence may not be effective in capturing the popularity of Web objects in the *proxy cache*. One reliable reason for getting this result is the efficient *client caching*. Namely, the efficient client cache deals with the objects that have short-term frequency like temporal locality property. Then, in the proxy cache the objects that have long-term frequency are requested. So, if we use the previous algorithms in the *proxy cache*, we can not get the desirable performance.

In this paper, we suggest a new algorithm, called *Least Popularity Per Byte Replacement*(LPPB-R). This LPPB-R

0-7695-1153-8/01 $10.00 © 2001 IEEE

algorithm is a function-based algorithm. The LPPB-R algorithm uses the *popularity* value of the object from proxy caches directly, to make up for the weak points of the previous algorithms in the proxy cache. And this algorithm considers the *variable size* of objects, since the web objects are not homogeneous. The function of the LPPB-R is to make the popularity per byte of the outgoing objects to be minimum. Namely, when an object is to be inserted into the cache, the LPPB-R calculates the utilization of each object by using the popularity and the size. Then, LPPB-R evicts the object, which has smallest utilization.

We suggest the 2 models of calculating the popularity value, since how to set the popularity value determines the performance of this LPPB-R algorithm. One is the simple model by using the reference count, the other is the advanced model by using the reference count as the power term of the impact factor. According to these, we can accommodate a variety of performance goals by changing the β value which is the impact factor.

In addition, we use the LFU-based multi queues by managing the objects and the meta information in the cache, to make the LPPB-R algorithm more practical. And because our LPPB-R algorithm has the LFU property by using the popularity value, we must consider the cache pollution phenomenon. So, we suggest a technique for managing objects to avoid the cache pollution phenomenon.

The paper is organized as follows. In section 2, we describe several related works about the replacement algorithm. Section 3 introduce our new replacement algorithm, LPPB-R. The simulation environment and the performance evaluation are given in section 4. Finally, We conclude the paper in section 5.

2. RelatedWork

There are many replacement algorithm to obtain better performance of proxy caching, from simple to complex. They attempt to maximize various cost metrics, such as hit rate, byte hit rate, and reduced latency. We classify these algorithms into three categories; Traditional algorithm, Key-based algorithm, and Function-based algorithm.

2.1 Traditional Replacement algorithm

There are Least Recently Used (LRU), Least Frequently Use (LFU), First In First Out (FIFO), and etc. in this category. These algorithms are used widely, because of their simplicity and robust feature. But these algorithms are suggested for homogeneous paging caching, so these are not satisfied the web environment, whose traffics have the non-homogeneous feature, the variable latency, the dynamic relative frequency of request, and etc. In other words, the difficulty with the algorithms in this category is that they fail to pay sufficient attention to the characteristic of Web.

2.2 Key-based Replacement algorithm

The algorithms in this category enhance the performance of proxy caching by applying the web characteristics to the traditional algorithms. The main idea in key-based policies is to sort objects based upon a primary key, break ties based on a secondary key, break remaining ties based on a tertiary key, and so on. There are size, latency, reference count, last access and etc that are used as the keys. The idea in using the key-based algorithms is to prioritize factors. However, such prioritization may not always be ideal. So these algorithms are only good for a specific metric. For example, size parameter is good for Hit Rate, but it is not good for other metrics. So when we use LFF for the proxy cache, we get very high Hit Rate, but get poor Byte Hit Rate.

2.3 Function-based Replacement algorithm

There are Greedy Dual-Size [6], Hybrid [10], Lowest Relative Value [8], Least Normalized Cost Replacement [9], Bolot/Hoschka [7], Size-Adjust LRU [2] and etc. in this category. The idea in function-based algorithms is to employ a potentially general function of the different factors such as time since last access, entry time of the object , transfer time cost, object expiration time, object size and so on. Namely, these algorithms use the utilization of each object which is calculated by many parameters that are related to web characteristics. This utilization indirectly infers the popularity. In the most algorithms, this utilization is calculated by estimated value like a size, a latency, a last access rather than the direct popularity value of objects like the relative access frequency or reference counts through a proxy. Especially in the case of Size-Adjust LRU, this algorithm use the size and the last access number to get the object utilization. Size-Adjust LRU assumes that both of them can indicate the future popularity of the object.

But recent studies [4] suggest that such relationships are weakening. One reliable reason for getting this result is the efficient client caching. If the client caches are used, when the client has a request for one object, first the client checks the client cache whether the object is in the client cache. Then if the object is in the cache, uses that object, but if not, the client sends the request to the proxy cache. Nowadays, the client such as a web browser has the client cache which has sufficient size to behave efficiently. So, the efficient client caching deals with short-term measurement of request frequency like temporal locality property. So, if Size-Adjust LRU, which uses temporal locality and size, is used in proxy cache, we can not get the desirable performance.

3. Least Popularity Per Byte Replacement Algorithm

According to these related works, we must consider that the correlation between temporal locality or size and popular objects is weakening by the efficient client caching. So, to get better performance of the proxy cache, the proxy cache should deal with the *long-term measurement* of request frequency. In this section, we present Least Popularity Per Byte Replacement algorithm(LPPB-R). This LPPB-R additionally uses object popularity parameter, which is not considered by previous algorithms, from proxy cache directly.

3.1 Overview of LPPB-R

When an object is to be inserted into the cache, more than one object may need to be removed in order to create sufficient space. In this case, LPPB-R follows the behavior of function-based algorithm. Like all other function-based algorithm, LPPB-R calculates the utilization of each object, then evict the object which has smallest utilization. This utilization $U(j)$ of an object j is calculated by the following model:

$$U(j) = P(j)/S(j)$$

where $P(j)$ is the popularity of the object j and $S(j)$ is the size of the object j. We describe the detail of the popularity in section 3.2. In this model, the utilization of an object, $U(j)$ represents the popularity per byte. So when the proxy cache wants to the replacement for a new object, the cache evicts the object, which has the smallest popularity per byte value.

According to this utilization and the cache policy, we can have the following model for the LPPB-R replacement algorithm in general.

$$Minimize \sum_{j \in C} D(j) \times U(j)$$

$$such\ that \sum_{j \in C} D(j) \times S(j) \geq R$$

$$and\ D(j) \in \{0,1\}$$

where C means the group of object in the cache when the replacement event occur. R is the required size for the new object that is to be inserted. Let $R \geq 0$ denotes the amount of additional space in the cache, which must be created in order to accommodate the new object. And $D(j)$ is the decision variable for object j defined to be 1 if we wish to purge it, and to be 0 if we want to retain it. We can analyze above formulas by using these notations. If the replacement event occur (i.e $R \geq 0$), the proxy cache selects the object which is evicted. In this case, the proxy cache makes the sum of objects which are removed to be larger than the R and makes the sum of popularity per byte to be as small as possible.

3.2 Getting the Popularity Value

In LPPB-R, the popularity value is the most important parameter to get the utilization value. And this popularity value must consider the long term measurement of request frequency, which is neglected in the other algorithms. We use the reference count value to get the popularity value. The reference count value is highly variable over short time scales, but this is much smoother over long time scales. [3] This property makes the popularity value to deal with the long term measurement of request frequency.

We suggest the 2 models to get the popularity value One is the simple model, and the other is the enhanced model. We describe the detail of these two type value. In the following models, $P(j)$ is the popularity of an object j.

$$(1)\ P(j) = R(j)/T$$

where $R(j)$ is the reference count number of object j (i.e. $R(j)$ shows how popular the object j) and T is the total request number through the proxy cache at the time when the new object is to be inserted into the cache. In this model, we don't use the reference count number directly, but we use the normalized reference count number. This prevents the popularity value being more weighted than other factors. If we use $R(j)$ to get the $P(j)$ directly, the utilization value varies heavily by changing the popularity value slightly. Namely, the popularity value has more priority than the others, and this property prevents the LPPB-R algorithm getting the reasonable performance in the all metrics. So, we use the normalized value.

$$(2)\ P(j) = 1/(\beta)^{R(j)},\ (0 < \beta < 1)$$

where $R(j)$ is the reference count number of object j, and β is the constant for the impact factor. This model is in a exponent -like fashion as a function of the $R(j)$. And the β value adjusts the raising point of the graph. If the β value is near zero, the raising point is near zero (i.e. y axis). And if the β value is near one, the raising point is near infinity. By using this model, we weight the popularity parameter partially. This means that we have the popularity threshold. If the popularity of the object is larger than the threshold, the popularity value is weighted, then this object is evicted very seldom.

For example, if $\beta = 0.1$ and the difference between $R(j)$ and $R(i)$ is 1, the difference of the utilizations become 90. But if $\beta = 0.9$ and the difference between $R(j)$ and $R(i)$ is 1,the difference of the utilizations becomes 0.1. If $\beta = 0.9$ and we want to make the difference of the utilizations 90,

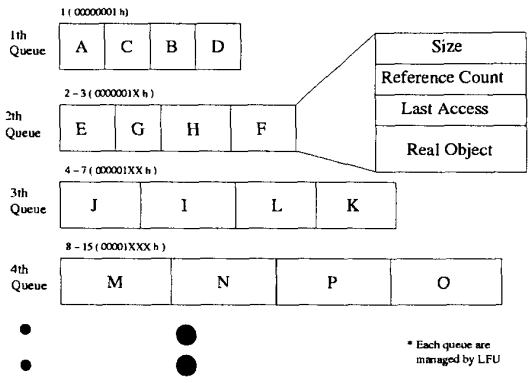


Figure 1. Multiqueue structure used by LPPB-R

the difference between $R(j)$ and $R(i)$ must be more than 65. So, because β value affects the popularity value heavily, we can accommodate a variety of performance goals by changing the β value. In this paper, we set the β value on the range from 0.3 to 0.5 to get the good performance in the all metrics.

3.3 Managing the objects

Strictly following the LPPB-R has the terrible overhead to calculate the utilization values. When the cache needs the replacement event, the LPPB-R calculates the utilization values of the all objects. Then, it evicts the object that has the smallest utilization value until the cache has sufficient space. This operation needs $O(k)$ time, where k is the object number in the cache. So, the overhead is proportional to the cache size and this method is not practical.

To turn this ideal theory to practical algorithm, we use the multi queue to manage the objects. ith queue manages the objects whose size is from 2^{i-1} to $2^i - 1$. Thus, there will be $N = \lceil log(M+1) \rceil$ different queues of objects, where M is the cache size. Additionaly, If we divide the queues by $\lceil log_2(size) \rceil$, when the cache manages the objects, the cache gets more advantages in determining object size. Because the bitwize operation which uses the size of object is possible. For example, if the size of the object j is 10byte we present this size 000a by using 32bit. Then, we find the 1 in the 4th bit and we insert this object to the 4th queue. For the cache, the objects in each queue i are maintained as a separate LFU list. And the meta information like reference count, size, latency, and etc are attached to each object in every queue. Figure 1 shows the general structure of the multi queues. Using these queues, when cache finds the objects which are evicted, cache does not search the whole queue, but compares the utilization values for the least frequently used objects of each list. Finally, the time which is needed to process one request decreases from $O(k)$ to $O(\lceil log(M+1) \rceil)$.

If we use this practical algorithm, we have some disadvantages to find unpopular objects which are evicted. But this disadvantage has the limit. If $U(i)$ is the minimum of the utilization value by the ideal LPPB-R theory and $U(p)$ is the minimum of the utilization by the practical LPPB-R algorithm, $U(p)$ has the same value or the less value than $2 \cdot U(i)$. Namely, this means the following model:

$$U(p) < 2 \cdot U(i)$$

The detail description for this model is following: First, we assume that the any one queue has the object i which has the minimum of the utilization value by the ideal LPPB-R theory and the LFU list header object for this queue is object j. In this case, $S(i) < 2 \cdot S(j)$ and $2 \cdot U(i) > U(j)$ is true. (if this condition is not true, the i and j can not exist.) Then, if the object p has the minimum of the utilization value by comparing the LFU list header objects for the whole queue, $U(j) > U(p)$ is true. According to these facts, $2 \cdot U(i) > U(j) > U(p)$. This happens in the worst case. But in reality, the value $U(p)$ is so close to $U(i)$ that the performance difference between the ideal theory and the practical LPPB-R algorithm.

Notice that the LOG2-SIZE discussed in [11] and SLRU discussed in [2] bears at least some resemblance to the independently derived LPPB-R scheme. The LOG2-SIZE scheme always chooses the least recently used items in the nonempty stacks corresponding to the largest size range. And in the SLRU scheme, the objects in the queue are maintained as a LRU list. In contrast, the LPPB-R scheme applies LFU algorithm to each queue and looks at the least frequently used objects of each stack, and among these picks the object which has the least utilization.

3.4 Avoiding the cache pollution phenomenon

In LPPB-R algorithm, the most important parameter among the many factors is popularity and this value is determined by the object reference count which can infer the long-term frequency of an object. This reference count value has an advantage, that the cache gets the popularity information of the objects directly by the requests through the cache. But this has a bad characteristic that decreases the performance. That is the request is bursty. So, in the short-term period, the LFU has less performance than the LRU. For example, in the short-term period, the reference count value of the popular objects is much higher than the others that are not popular. So, though these objects are not popular any more and these must be evicted, cache can't

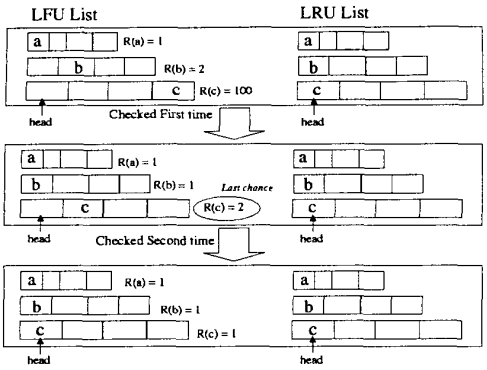


Figure 2. Cache pollution avoidance mechanism

evict these objects because their reference count values have been very high. This phenomenon is called *cache pollution phenomenon*. The cache pollution phenomenon can occur not only in the short-term period, but also in the long-term period. Consequently, to make LPPB-R more efficient than other algorithms, we must avoid the cache pollution phenomenon.

We use the LRU list to avoid the cache pollution phenomenon. Namely, the proxy cache uses two lists, LRU list and LFU list. The LFU list is used to calculate the utilization value and to select the objects which is removed, and the LRU list is used to avoid the cache pollution phenomenon.

In this paragraph, we describe the detail about the mechanism for avoiding the cache pollution phenomenon. First of all, the proxy cache checks the least recently used objects which are in the whole LRU list periodically. If the difference between the last access time of the object and the current time is greater than the threshold value, the LPPB-R algorithm sets the reference count of the object on new value which refers that this object is not popular. In this mechanism, when the proxy cache finds this object which fits above situation for the first time, we change that reference count value to 2. When that object meets the similar situation again for the second time, we change that reference count value to 1 which is the minimum value of an reference count value. We use these two steps for the operation, because we want to give the last chance to the objects which were popular. Figure 2 show the whole mechanism for avoiding the cache pollution phenomenon. We put the period by which the cache checks the LRU list to 10000 requests and put the threshold value to 1000000 requests.

4 Performance evaluation

In this section, we present the result of extensive trace driven simulations that we have conducted to evaluate the performance of LPPB-R. We design our proxy cache simulator for doing the performance evaluation. This simulator illustrates the behavior of an proxy cache. We have some assumption to simulate the behavior of a proxy cache effectively. The size of an proxy cache is in the range from 0.01MByte to 500MByte. We assume that the proxy cache is located alone between user's clients and web servers and that there are not any ICP packet and any other cache interactions. We also assume that there are not any problems in the network, such as congestions and overflow buffers.

4.1 Traces used

In our trace-driven simulations we use traces from NLANR [1]. We have run our simulations with traces from the *pb* proxy server and the *bo2* proxy server of NLANR since September, 2000. Both proxy cache servers have 100MB/s traffic bandwidth.

We show some of the characteristics of these traces in Table 1. Note, these characteristics are the results when the cache size is infinite. But, since our simulations assume limited cache storage and the replacement events can occur, the ratios like hit rate , byte hit rate and reduced latency cannot be higher than the infinite cache ratios. And we also analyze the traffics by the requested object size, then we find that almost the requests are in the range from 256 byte to 2048 byte. And we also find that the hit rate in the pb trace is lower than in bo2 trace, but the byte hit rate in the pb trace is higher than in bo2 trace. This means that pb trace has more large objects that are requested frequently than bo2 trace.

4.2 Performance metrics and algorithms

We consider three aspects of web caching benefits: hit rate, byte hit rate and reduced latency. By *hit rate*, we mean the number of requests that hit in the proxy cache as a percentage of total requests. Higher the hit rate is, more requests the proxy cache can treat and less requests the original server must deal with. By *byte hit rate*, we mean the number of bytes that hit in the proxy cache as the percentage of the total number of bytes requested. Higher byte hit rate is , more network traffic decreases in the server side. By *reduced latency*, we mean the response time which is reduced by the cache. More latency is reduced, less time is needed for client to get the response for an request. And more latency is reduced, less overhead the network has.

We compare the performance of LPPB-R with LRU, LFU, LOG2SIZE [?] and Size Adjust LRU [2]. LRU and

Traces	PB server	BO2 server
Measuring days	2000.10.03-04	2000.09.14-20
Requests size	18595227993B	25607415375B
Objects size	10016257877B	15363371177B
Requests number	2366457	2215404
Objects number	1183791	969892
Hit Rate	49.976 %	56.221 %
Byte Hit Rate	46.135 %	40.004 %
Reduced Latency	5887966sec	7106974sec

Table 1. Traces used in our simulation (Cache size = infinity)

LFU are the representative algorithms of the traditional replacement algorithm. Both two algorithms use only one simple queue to manage the objects. LOG2SIZE algorithm is applied as the representative of the key-based replacement algorithm. This algorithm uses the object size as the primary key and the last access time as the secondary key. And this algorithm manages the object by the multi queue according to the object size. To compare between LPPB-R and other function-based replacement algorithms, we implement the Size Adjust LRU algorithm, which is one of the newest and most efficient algorithms. This uses the object size and the last access time as the parameters which is used to calculate the utilization. And this uses a multi queue according to the object size.

4.3 Implementation issues on LPPB-R

There are two consideration to implement the LPPB-R algorithm. One is whether we can use the practical policy rather than the ideal theory without any loss in the performance. The other is how the β value set to get the ideal performance in all metrics.

First, we compare the performance of them, the ideal theory and the practical policy. According to these results, we find that both of them get the similar performance in all metrics. Consequently, we use the practical policy to implement LPPB-R algorithm without any loss in the performance .

Next, we measure the performance of the all metrics with variable β values to both of two traces to select the β value for good LPPB-R. According to these results, we find the following facts: If we set the β value to the range from 0.5 to 0.7, LPPB-R achieves the best hit rate. On the other hand, closer to zero the β value is, better the performance of the cache is in the byte hit rate and the reduced latency. These mean that there are some trade off between the metrics for using the β value. For example, if we set the β value to 0.7 to achieve the best hit rate, the performance in the byte hit rate and the reduced latency decreases. So, when we set the β value, we must consider the balance of the performances in all metrics. We set the β value to 0.5 in simulation for bo2 traces, and to 0.3 for pb traces.

Last, because of the difference of mechanisms for calculating the popularity value, we divide the LPPB-R algorithm into LPPB-R 1 and LPPB-R 2. LPPB-R 1 algorithm uses the first simple way to get the popularity value. And LPPB-R 2 algorithm uses the second way which uses the β value.

4.4 Performance Measurement

4.4.1 Hit rate

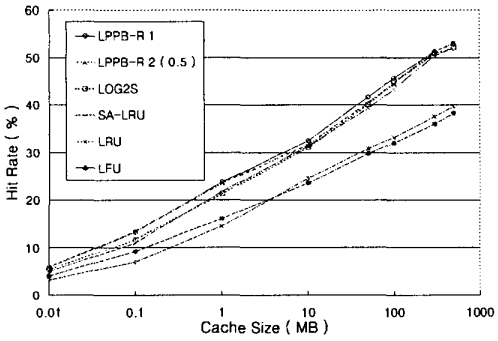
The results from Figure 3 show that clearly, LPPB-R achieves the best hit rate among all algorithms across traces and cache sizes. Both of the LPPB-R 1 and the LPPB-R 2 (0.5)have the best hit rate. In the result of bo2 traces, if the cache size is 500MByte hit rate of the LPPB-R is about 53%. And LPPB-R has more hit rate by the value from 2% to 4% than Size Adjust LRU independently of the cache size. This result presents that the popularity value makes up for the weak point, which is the weakening temporal locality, in the Size Adjust LRU.

The traditional algorithms like LRU and LFU have the smallest hit rate among all algorithms. This result is quite natural since the traditional algorithms don't consider the characteristics of the Web traffics. In the result of bo2 traces, if the cache size is 500MByte, LFU has less hit rate by 12% than LPPB-R algorithm. On the other hands, LOG2SIZE that is the one of the key-based algorithm has high hit rate. This show the characteristic of the key based algorithm which use the object size as the primary key. In other words, if we use key based algorithms, hit rate is very high, but other metrics, like byte hit rate and reduced latency, are low. This fact is shown in the following section.

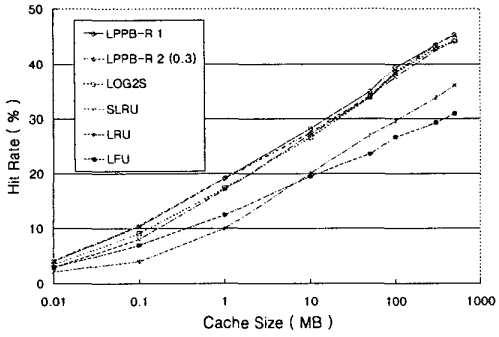
4.4.2 Byte hit rate

LPPB-R achieves better byte hit rate than Size Adjust LRU, not generally but partially. But, LPPB-R achieves better byte hit rate than LOG2SIZE which is key based algorithm or LFU in general. Namely, if the cache size is smaller than 40MByte LPPB-R achieves the best byte hit rate among all algorithms, but if the cache size is larger than 40MByte LRU achieves the best byte hit rate among all algorithms. This result is found in both of bo2 traces and pb traces. The reasonable reason for this result is that the traces have the large objects, which are requested frequently.

Differently form LRU, LFU cannot achieve the good byte hit rate because of the cache pollution phenomenon. And LOG2Size which use the object size as the primary key also has low byte hit rate since this algorithm is proper

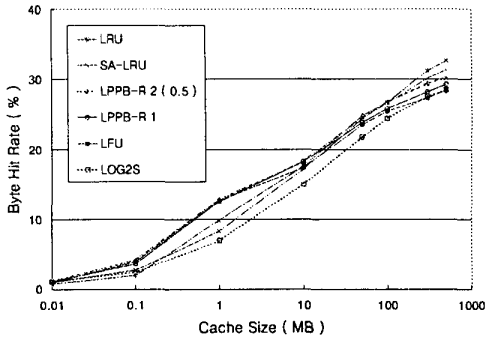


(a) BO2 Traces

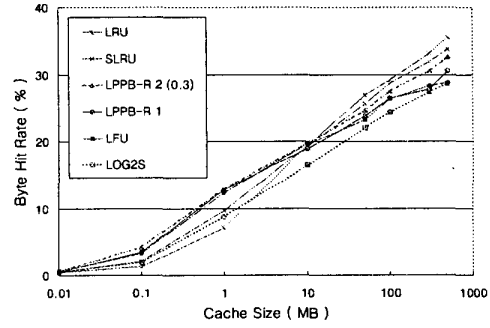


(b) PB Traces

Figure 3. Hit Rates of many algorithms for each trace



(a) BO2 Traces



(b) PB Traces

Figure 4. Byte Hit Rate of many algorithms for each trace

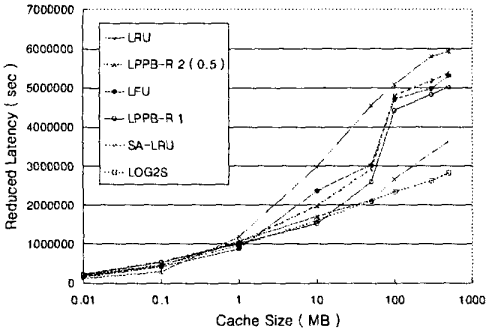
to store the small objects. But, Size Adjust LRU follows the LRU characteristic because this algorithm use the temporal locality which is the feature of LRU. The difference of the byte hit rate between Size Adjust LRU and LRU is 1%.

LPPB-R uses the reference count rather than the temporal locality. In other words, this uses the feature of LFU rather than LRU. Though LPPB-R has that feature, LPPB-R doesn't follow the LFU characteristic but achieves the reasonable byte hit rate. Especially, if the β value is near zero, LPPB-R 2 achieves high byte hit rate. The byte hit rate of LPPB-R 2 with 0.1 is similar to that of Size adjust LRU. So if we want to achieve the high byte hit rate rather than the high hit rate, we set the β value to the near value from zero and get the high byte hit rate. And LPPB-R achieve the better byte hit rate than other algorithms, in the small-sized cache, like the main memory cache for hottest objects. Figure 4 show the result of byte hit rate.

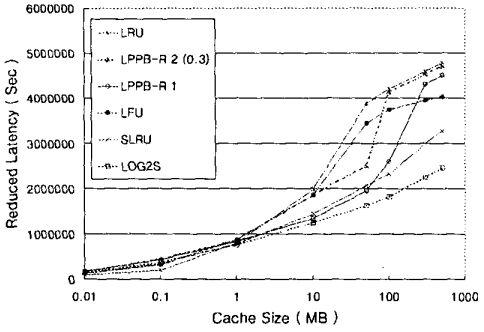
4.4.3 Reduced Latency

Another major concern for proxies is to reduce the latency of HTTP requests through caching, as numerous studies have shown that the waiting time has become the primary concern of Web users. To measure the performance of this metric, we assume that the average time to transfer a byte is constant for each server but different between each server.

In this metric, the traditional algorithms like LRU and LFU, which are simple and pair for all objects, achieve better reduced latency than other algorithms except our LPPB-R algorithm. But LOG2Size has the minimum of the reduced latency. Our LPPB-R almost achieves the best performance and especially has much better performance than Size Adjust LRU. In the bo2 traces, if the cache size is 500MByte, the difference of the reduced latency between LPPB-R and Size- Adjust LRU is 2 million seconds. The



(a) BO2 Traces



(b) PB Traces

Figure 5. Reduce Latency of many algorithms for each trace

effect of this value is equal to saving about one day for one thousand users. Though Size Adjust LRU follows the LRU property in byte hit rate, in reduced latency this doesn't follow that property. The reason of this result is the high priority of the size parameter. Consequently, Size Adjust LRU follows LOG2Size property which has the minimum of reduced latency. On the other hand, LPPB-R which has the LFU property gets better performance than LFU. Especially, if we set β value to the value which is near to zero, the reduced latency of LPPB-R 2 can be closer to that of LRU.

5 Conclusion

This paper introduces the new function-based replacement algorithm, LPPB-R for the web proxy cache and shows that it outperforms existing replacement algorithms in many performance aspects, including the hit rate and the reduced latency. LPPB-R algorithm uses the object popularity value and the object size to calculate the utilization value. Through trace-driven simulations, we show that LPPB-R algorithm outperforms other replacement algorithms and is practical. Especially, in the comparison between LPPB-R and Size Adjust LRU, we show that the popularity value covers the weak point of Size Adjust LRU in the proxy cache. In addition, we show that LPPB-R is easy to adjust the performance to needs of the proxy cache. This feature is possible by the characteristic of β value.

Consequently, we conclude that using the LPPB-R is better than other algorithms in the proxy cache which treats the traffics for the long-time period.

References

[1] National laboratory for applied network research. weekly access logs at nlanr's proxy caches. *Available via ftp:// ircache.nlanr.net/Traces/.*

[2] C. Aggarwal, H. L.Wolf, and P. S.Yu. Caching on the world wide web. *IEEE Transaction on Knowledge and data Engineering*, 11(1), January 1999.

[3] M. Arlitt and C. Williamson. Internet web servers: Workload characterization and implications. *IEEE/ACM Transactions on Networking*, 5(5):631–644, 1997.

[4] P. Barford, A. Bestarvros, A. Bradley, and M. Crovella. Changes in web client access patterns : characteristics and caching implications. *WWW journal*, 2(1), 1999.

[5] L. Breslau, P. Cao, L. Fa, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. *In Proceedings of Infocom'99*, April 1999.

[6] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. *In Proceedings of Usenix symposium on Internet Technology and Systems*, December 1997.

[7] J.C.Bolt and P.Hoschka. Performance engineering of the world-wide web: Application to dimensioning and cache design. *Proceedings of the 5th International WWW Conference*, May 1996.

[8] L.Rizzo and L.Vicisano. Replacement policies for a proxy cache. *Research Note RN/98/13, Department of Computer Science, University College London*, 1998.

[9] P.Scheuermann, J.Shim, and R.Vingralek. A case for delay-conscious caching of web documents. *Proceedings of the 6th International WWW Conference*, April 1997.

[10] R.P.Wooster and M.Abrams. Proxy caching that estimates page load delays. *Proceedings of the 6th International WWW Conference*, April 1997.

[11] S.Williams, M.Abrams, C.R.Standbridge, G.Abdulla, and E.A.Fox. Removal policies in network caches for world wide web documets. *In Proceedings of the ACM Sigcomm96*, August 1996.

[12] J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.