Caching on the Changing Web

Ying Xing

May 17, 2001

Abstract

Web caching is widely used to reduce access latency and network traffic. Achieving good performance in web caching requires consideration of many factors, such as object sizes, retrieval latencies, the objects' autonomous updates, etc. While object sizes and retrieval latencies are usually considered in the studying of caching algorithms, the validation costs due to the object updates tend to be ignored by most of the works. However, when there are many objects in the cache whose access rates are relatively slow compare to their change rates, the performance of caching will be greatly impaired. In this work, we present two algorithms to improve the performance of caching in the above case. The first algorithm considers both the object access frequency and object update frequency using a single benefit-cost function. The second algorithm uses multiple queues to separate frequently updated objects and infrequently updated objects. By discounting the frequently updated and infrequently accessed objects in the cache, the algorithms can effectively improve the performance of caching.

1 Introduction

Caching of web objects is widely used to reduce access latency and network traffic. Web caching problems have different properties than traditional memory caching because web objects are more complicated than memory pages. Web objects have different sizes, retrieval times, and are updated autonomously at their sources. All these factors affect the performance of caching. Many studies have examined how to incorporate object size and retrieval time into cache replacement policies [3, 9, 6], while few works consider the data consistency problem in replacement algorithms. In this paper, we examine the performance of caching with different object access rates and update patterns, and give a more complete picture of caching on autonomously changing data.

When data are updated autonomously outside the cache, validation costs must be paid to maintain data consistency. If data access rates are much faster than data update rates, validation costs can be, for simplicity, ignored in cache replacement policies. However, if the above condition does not hold, the effectiveness of caching will be greatly impaired. In this case, considering data validation cost as a factor in cache replacement policies may be beneficial. Obviously, if we need to evict one object choosing from two objects with the save size, retrieval time and estimated access frequency, we want to replace the one updated more frequently. In these work, we present two algorithms implementing this idea and discuss their performance.

The rest of the report is organized as follows. Section 2 introduce the data consistency model our work based on. Section 3 presents our caching algorithms. Section 4 and 5 introduce the experimental setup and discuss the results. Related work is discussed in section 6. Section 7 and 8 introduce our future work and summarize the contribution of the present work.

2 Data Consistency Model

There are two data consistency models for autonomously updated data. The first one is strong data consistency model, where no stale data can be returned. Each cached copy must be validated with the server upon each request. This is very inefficient because of the unbounded Internet delay. Thus, most web caches use the weak data consistency model, where stale data might be returned occasionally. In the weak data consistency model, each cached object is assigned a Time-To-Live (TTL) value when it is first cached. Before the time of $now+TTL$, the cached copy can be returned upon request without validating it with the server. After that time, the cached copy is considered expired. If an expired cached object is requested, a GET If-Modified-Since request is sent to the server. The server will return a new object if the object has been updated since the provided time.

Our work is based on the weak consistency model and uses the adaptive TTL approach to set the TTL for each web object. The adaptive TTL approach is widely used in web caching schemes. In this approach, if the expiration time of an object is not provided by the web server, its TTL is set as a constant times the object's *age*, where *age* is defined as the amount of time since the object's last modification. The adaptive TTL approach is based on the observation that *young* objects tend be modified more frequently than *old* objects. Thus TTL can also be used as an indication of an object's change rate.

3 Algorithms

3.1 Benefit-Cost Model

One basic goal of caching is to minimize the overall request response time. If the benefit of object i is defined as the latency reduction, gained by caching object i, and the cost of caching i is defined as the space i occupied, then the above goal is the same as maximizing the overall benefits with fixed cost. This optimization problem is NP-hard. A suboptimal solution can be achieved by replacing the object with the lowest benefit over cost value each time [10].

When validation cost is considered, the benefit-cost function of object i can be defined as

$$V_i = \frac{B_i}{C_i} = \frac{r_i \cdot l_i - (v_i - u_i) \cdot c_i - u_i \cdot l_i}{s_i} \tag{1}$$

with the following parameters:

- r_i - estimated access frequency of object i
- l_i - average retrieval time of object i
- v_i - average validation frequency of object i
- u_i - average observed update frequency of object i
- c_i - average connection time to validate object i when it is not updated
- s_i - size of object i

This function can be understood as follows, When a time interval T is considered, object i is expected to be referenced $r_i \cdot T$ times in interval T. If object i is not cached, then the total response

time of requests for object i is $r_i \cdot l_i$. If object i is kept in the cache, then the response time of a request for i is 0 unless i is found to be expired. When i is found to be expired, a validation request is send to the server. If i is not updated, the average response time of the validation cost is the average connection time c_i, otherwise, the average response time is the object's average retrieval time l_i. During interval T, on average, object i is validated $v_i \cdot T$ times and observed to be updated $u_i \cdot T$ times. Thus the total response time of request for object i during interval T is $(v_i - u_i) \cdot c_i + u_i \cdot l_i$ when i is cached. So the latency reduction gained by caching i is just $r_i \cdot l_i - (v_i - u_i) \cdot c_i - u_i \cdot l_i$. That is the benefit of caching object i: B_i in equation 1.

3.2 TTL Integrated Algorithm

Although the above benefit-cost function is well formed, some parameters used such as r_i, u_i can only be very roughly estimated, In this section, we introduce the TTL Integrated Algorithm which uses an approximated form of equation 1 and achieve good performance by tuning an "effect parameter".

The new benefit-cost function is derived from equation 1 by the following approximation:

- The current TTL_i is used to approximate the average TTL_i, thus v_i can be approximated by $\frac{1}{TTL_i}$.
- Assume the observed age of an object is roughly proportional to its average lifetime, where lifetime is defined as the amount of time between the object's two successive modification. Then based on the adaptive TTL approach, u_i can be approximated by a constant times $\frac{1}{TTL_i}$.
- c_i is approximated by a constant times l_i.

The approximated benefit-cost function is then:

$$\hat{V_i} = \frac{\hat{B_i}}{C_i} = (r_i - \frac{C}{TTL_i}) \cdot \frac{l_i}{s_i} \tag{2}$$

where C is a constant.

By having a constant C adjusting the balance between r_i and $\frac{1}{TTL_i}$, r_i does not need to be an absolute frequency anymore. It can be any relative value that can be used to compare the estimated access frequency of two objects. For example, r_i can be the access count estimated from some LFU algorithm, or it can even be expressed by sequence number using the LRU algorithm. The algorithm that is used to estimate r_i is called the base algorithm.

Using equation 2 and a base algorithm estimating r_i, the TTL integrated algorithm computes a benefit-cost value for each object. The object with the lowest benefit-cost value is always chosen to be replaced.

To achieve desired performance, constant C must be tuned to adjust the effect of validation cost in the benefit-cost function. How to adjust C and it's effect is discussed in Section 5.

3.3 TTL Based Multi-Queue Algorithms

Since the benefit-cost value for each object can only be very roughly estimated, the comparison of the benefit-cost value between two object is often inaccurate. In this section, we separate the objects into 4 classes and compare the benefit-cost between classes instead of between objects.

By comparing the access frequency and the change frequency, web objects can be classified into the following groups:

- frequently accessed and infrequently changed (FAIC) objects
- infrequently accessed and infrequently changed (IAIC) objects
- frequently accessed and frequently changed (FAFC) objects
- infrequently accessed and frequently changed (IAFC) objects

The benefit value of these classes are compared in Table 1. Obviously, it's beneficial to keep FAIC

	FAIC	IAIC	FAFC	IAFC
FAIC	-	$>$	$>$	$>$
IAIC	$<$	-	?	$>$
FAFC	$<$	?	-	$>$
IAFC	$<$	$<$	$<$	-

Table 1: Benefit comparison between classes.

objects in the cache and replace IAFC objects before other objects. The comparison between IAIC objects and FAFC objects depends on the specific objects that are compared.

First, using the objects' TTLs as an indication of their change frequency, we can separate all the cached objects into two priority queues. Objects with TTL less than a threshold are maintained in a queue called the *short TTL* queue. Objects with TTL larger than the threshold are maintained in the other queue called the *long TTL* queue. The threshold is also termed the *boundary TTL*. Many caching algorithms, such as LRU and LFU, can be used to estimate the objects' relative access frequency. We use one of these algorithms to sort the above priority queues by the objects' estimated relative access frequency, such that the frequently accessed objects are at the top of the queues and infrequently accessed objects are at the bottom of the queues.

When an object needs to be evicted from the queue, a decision must be made to choose the bottom object from some queue. Below, we discuss three approaches for selecting the queues:

***Short TTL* Queue First:** The bottom object in the *short TTL* queue is always chosen for replacement first. This is a straight forward approach based on the fact that IAFC objects are at the bottom of the *short TTL* queue. However, by doing so, the FAIC objects that are on the top of the *short TTL* queue are always evicted form the cache before IAIC objects that are at the bottom of *long TTL* queue. One way to avoid this is to set a minimum queue length for the *short TTL* queue. The top sub-queue with that minimum queue length in the *short TTL* queue can be called *steady* queue. Objects inside *steady* won't be replaced. When the boundary TTL is set appropriately such that the benefit values of the objects at the bottom of the *steady* queue is roughly equal to the benefit values of the objects at the bottom of the *long TTL* queue, objects in the cache can be replaced in a rough order from objects with small benefit to objects with large benefit.

End Compare: In this approach, the benefit value of the of the two bottom objects are compared directly. The object with lower benefit is evicted from the cache. The performance of this approach depend on how well the benefit value can be estimated.

Performance Feedback: The performance of a caching algorithm can be measured by the cache's latency reduction ratio [10]. For a set of requests, R, the cache's latency reduction ratio is

defined as

$$\frac{\sum(l_i - s_i)}{\sum l_i}$$

where l_i is object i's retrieval time and s_i is the real response time of the request for object i. The summation is over all the requests. Many studies [1, 3, 6] and our experimental results show that a cache's hit ratio and latency reduction ratio is $o(cache\ size)$. If we define $\sum(l_i - s_i)$ on set R as the total benefit of the cache, the marginal benefit of the cache can be defined as $\frac{\triangle total\ benefit}{\triangle cache\ size}$. Because the latency reduction ratio is $o(cache\ size)$, the marginal benefit of a cache decreases as cache size increases in general.

If we have two caches with different marginal benefit and we can increase the size of one of the caches a little bit, then increase the size of the cache with larger marginal benefit results in a larger total benefits of the two caches than increase the size of the cache with smaller marginal benefit. If the total size of the two caches is fixed, the total benefits of the two caches is maximized when the marginal benefits of the two caches are the same.

The two queues in our algorithm can also be viewed as two caches. Each time we need to evict an object from a cache, we want to chose the cache with a smaller marginal benefit. The real marginal benefit of a cache is hard to measure. So we just use the total benefit divided by the size of the cache as an approximation of the marginal benefit. The total benefit is proportional to the latency reduction ratio. Thus, each time when an object need to be evicted from the cache, we choose the object in the queue with the smaller latency reduction ratio divided the size of the cache. We call this approach *Performance Feedback* approach.

The above idea of using two virtual queues can also be extended to use multi-queues. The more the queues, the harder it is to choose the boundary TTLs. In our experience, two or three queues are enough.

4 Experimental Setup

The performance of the above caching algorithms are tested on a statistical generative object update model and a statistical generative request model. These models are described below,

4.1 Web Object Model

There are 5 parameters of the web objects that are useful in our caching algorithms. They are object size, retrieval latency, connection latency and lifetime. All these parameters should be random variables. However, since we only want to study the impact of object change on the performance of caching algorithms, except for lifetime, all the other parameters are held constant across all the objects. In [8], the average ratio of connection latency over retrieval latency is 0.36 and 0.12 respectively in two traces. In our experiment, we use 0.2 which is roughly the average of the above two ratio as the connection latency over retrieval latency ratio for all the objects.

The study in [5] shows that the events of a web object's modifications usually happens as a Poisson process. Thus the lifetime of a web object can be generated from an exponential distribution. It's not very clear what distribution best characterize the distribution of the means of the lifetimes. Gamma distribution are used in [5] and [2]. We test the performance of LRU on five different distributions. The means of those distributions are set to be the same (30 days). Some of these distributions are far from reality, but it is useful to compare the performance of caching with different object update pattern. These five distributions are referred to as

- Single Point distribution - All the objects have the same lifetime mean (30 days).
- Fast_Slow distribution - Half of the lifetime mean is set to be 1 day, and the other half is set to be 59 days.
- Uniform distribution - The lifetime means are uniformly distributed between 0 and 60.
- Gamma1 distribution - This is a gamma distribution with 30 as its mean and a relatively small variance. Most of the lifetime mean generated by this model are near to 30 days.
- Gamma2 distribution - This is a gamma distribution with 30 as its mean and a relatively large variance. Using this model, there will be both a lot of fast changing objects and a lot of slowly changing objects generated.

4.2 Request Model

Breslan et. al. have shown that requests in real web traces follow a Zipf-like distribution [1]. The requests in our experiment are generated independently from a Zipf-like distribution. The inter-arrival time between requests is exponentially distributed. The experiments we do with $1000,000$ objects and $1000,000$ requests show similar results with experiments with $100,000$ objects and $100,000$ requests. Since the experiments with the later parameter setting requires less memory are much faster, we use the latter parameters setting for the experiments described here.

4.3 Compared Algorithms

Since the size, retrieval latency and connection latency parameters are the same for all the objects, we just use LRU as the base algorithm for all the algorithms presented in Section 3. These algorithms are:

TTL Integrated LRU (TTL-I-LRU): This is the TTL Integrated algorithm with LRU as its base algorithm. r_i used here is the sequence number of the requests, which means the least recently requested object has the smallest estimated access frequency.

***Short TTL* Queue First Multi-Queue LRU (SQF-MQ-LRU):** This is a TTL Based Multi-queue algorithm with the *Short TTL* Queue First approach. The LRU algorithm is used to sort each priority queue.

End Compare Multi-Queue LRU (EC-MQ-LRU): This is also a TTL Based Multi-queue algorithm with the LRU algorithm sorting the priority queues. The End Compare approach is used to select the object to be evicted. The benefit of the object is estimated by

$$\frac{ttl}{queue\ size}$$

where *ttl* is the amount of time since the moment of the comparison until the object expires. Since the objects compared are the bottom objects of the queues, and the queues are sorted using the LRU algorithm, the larger the *queue size*, the longer the objects has been in the cache. Thus the object is less likely to be referenced again in the near future. Since *ttl* is how much time remains until this object may need to be validated again, the lower the *ttl*, the less benefit we get from keeping the object. To make the *queue size*s reflect the access frequency of the bottom object of the queue, a minimum queue length need to be maintained for each queue.

Performance Feedback Multi-Queue LRU (PF-MQ-LRU): This is the third TTL Based Multi-queue algorithm with the LRU algorithm sorting the priority queues. The Performance Feedback approach is used to select the object to be evicted. A minimum queue length must be maintained for each queue for the latency reduction ratio to be meaningful.

4.4 Performance Metrics

We use the latency reduction ratio as the primary performance measurement since response time is usually what the users care the most.

We also compare the hit ratio of the caches since it reflects the network traffic reduction. A request for an expired object in the cache is counted as a cache miss in computing hit ratio.

5 Experimental Results

5.1 The impact of object changing on Caching

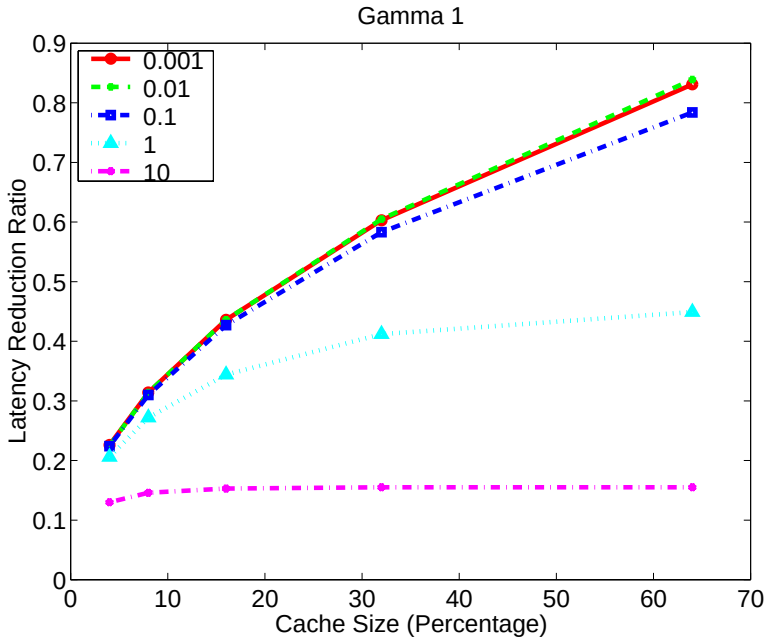
In the experiment, we first compare the performance of the LRU algorithm using different request inter-arrival time. This is because an object's change rate is only meaningful compare to the object's request rate, so we can fix the update pattern of the web objects and only change the request rate. Figures 1 and 2 show the latency reduction and hit ratios with Gamma1 and Gamma2 object lifetime mean distributions. It is not surprising that as the request rate decreases, The performance is serious degraded. The latency reduction ratios are always higher than the hit ratios because the expired data can still contribute to the latency reduction.

An interesting result is that when the performance of caching is tested on different lifetime mean distributions with the same distribution mean, the performance varies greatly. Figure 3 compares the latency reduction and hit ratio of the LRU algorithm with the five different lifetime mean distributions described in Section 4. The results show that the performance of caching is related more to the percentage of fast changing objects than to the shape of the distributions. As the percentage of fast changing objects increases, the performance decreases. It implies that those objects with small reference over change frequency values impair the performance the most. The motivation of the algorithms presented in Section 3 is simply to give these objects small benefit value and evict them from the cache quickly.

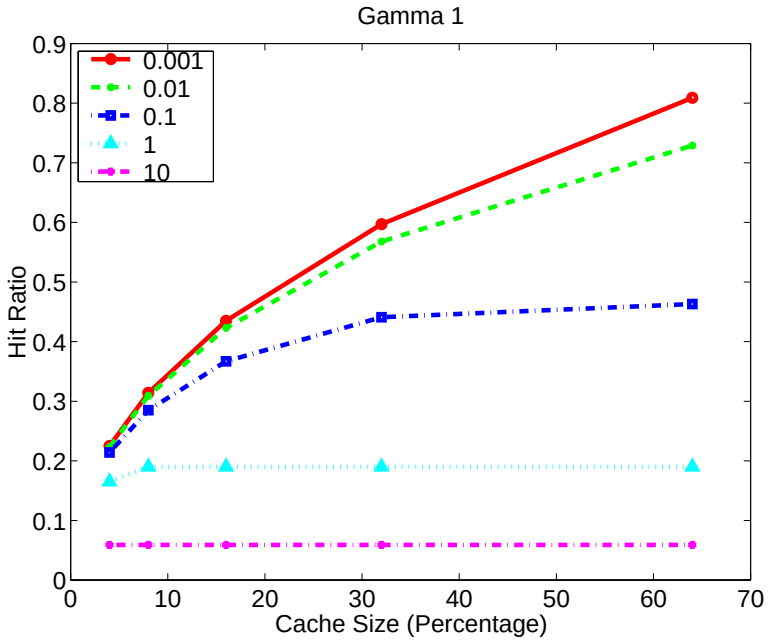
Before discussing the performance of those algorithms, we first have a look at in what situations it is suitable to apply those algorithms.

Firstly, if the request rates are fast enough for almost all of the objects, then it is not necessary to use those algorithms because the performance of caching is influenced little and can only be improved little. Secondly, if the request rates are too slow for almost all of the objects, the performance can hardly be improved. Thus, only when there are both a lot of objects with small access over change frequency values, and a lot of object with large access over change frequency values, our algorithms are most useful. This situation is also close to what is seen in reality.

Next, we study the performance of those caching algorithms in Section 3 under the Gamma2 distribution which generates many fast changing and slow changing objects. The request inter-arrival time mean used in the experiments is 0.1 minute.

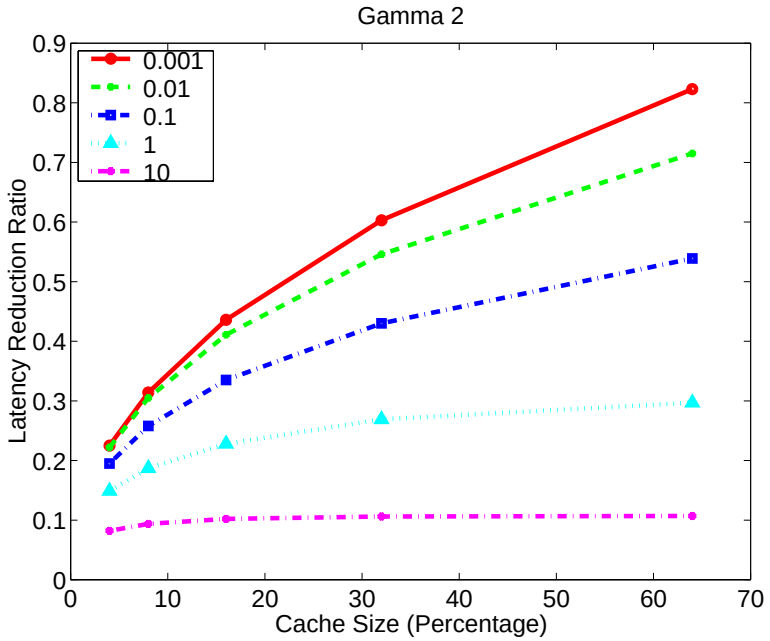


(a)

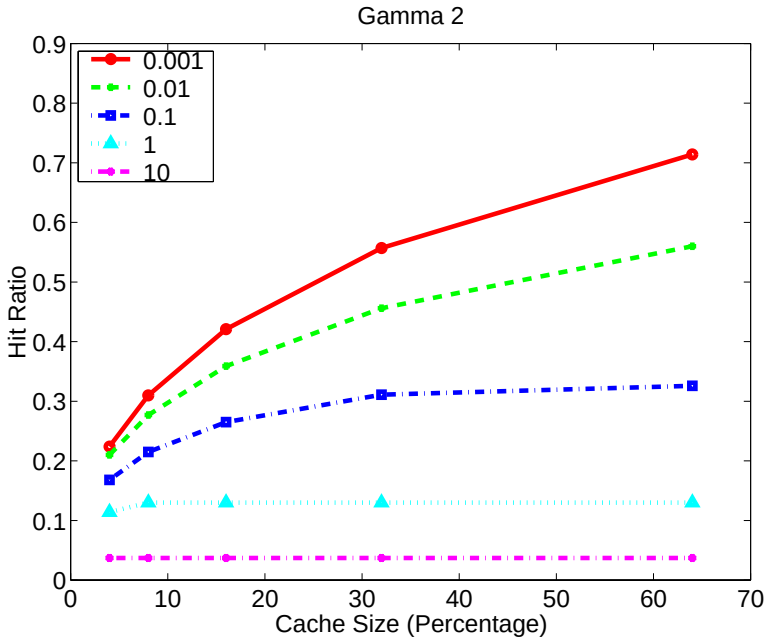


(b)

Figure 1: Performance of LRU algorithm on Gamma1 object lifetime mean distribution with different request inter-arrival rate. The mean of the Gamma2 distribution is 30 days. The mean of the request inter-arrival time are 0.001, 0.01, 0.1, 1, 10 minutes respectively.

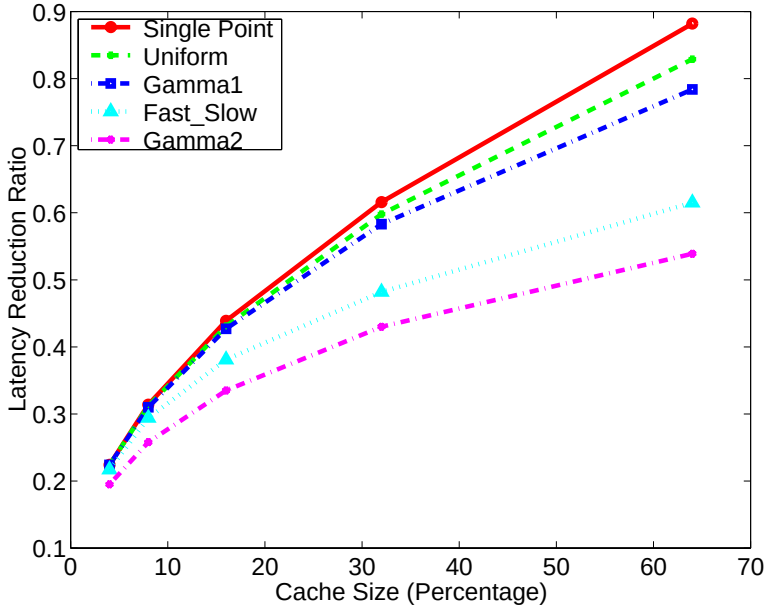


(a)

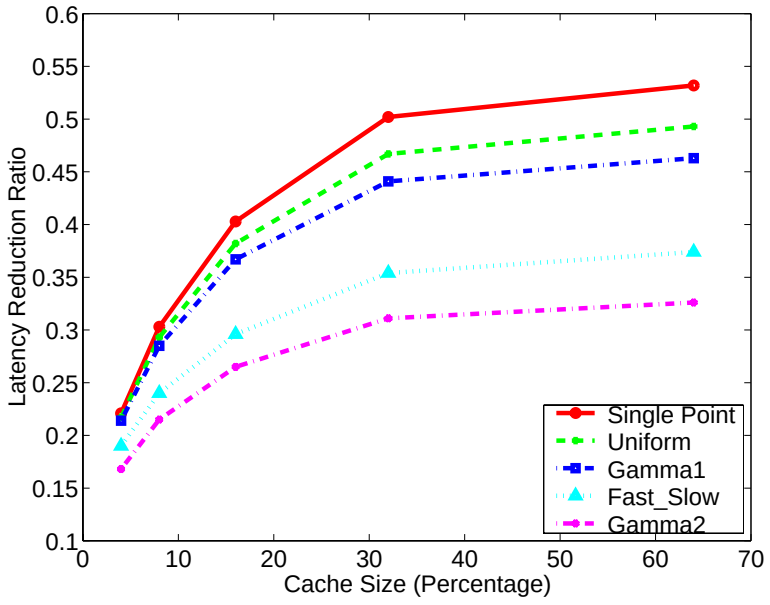


(b)

Figure 2: Performance of LRU algorithm on Gamma1 object lifetime mean distribution with different request inter-arrival rate. The mean of the Gamma2 distribution is 30 days. The mean of the request inter-arrival time are 0.001, 0.01, 0.1, 1, 10 minutes respectively.



(a)



(b)

Figure 3: Performance of LRU algorithm on different object lifetime mean distributions. The request inter-arrival mean is 0.1 minute.

5.2 Parameter Tuning

The algorithms in Section 3 all have some parameters which need to be tuned in order to achieve optimal performance.

5.2.1 TTL Integrated LRU algorithm

The parameter for the TTL-I-LRU algorithm is the validation cost balance coefficient C. The experimental results show that as C increases from 0, the latency reduction ratio and hit ratio of TTL-I-LRU first increases then decreases. This is because, when C is 0, TTL-I-LRU algorithms is reduced to the LRU algorithm. When C increases, the objects that change frequently get less benefit value, thus they are evicted from the cache faster. So the performance get better. If C is too large, the objects in the cache are mainly ordered by their TTL value. Those frequently referenced objects can't get enough value to stay in the cache. So the performance degrades. As long as C is within a certain range, the performance of the TTL-I-LRU is always better than LRU.

The experimental results also show that the best value of C is approximately proportional to the size of the cache. This is because, when the size of the cache increases, it takes longer for an object to be evicted. Thus increase C helps to evict those frequently changed objects faster.

The best C is not exactly proportional to the size of the cache, but the correct range for this parameter is very large. So C can just be roughly set to be a constant times the size of the cache, eg. Figure 4 shows the performance of TTL-I-LRU with two different choice of the constant.

5.2.2 TTL Based Multi-Queue Algorithms

There are two parameters in the TTL Based Multi-Queue algorithms: the number of queues, and the boundary TTLs. In our experiment, we test both two and three queues with different boundary TTLs.

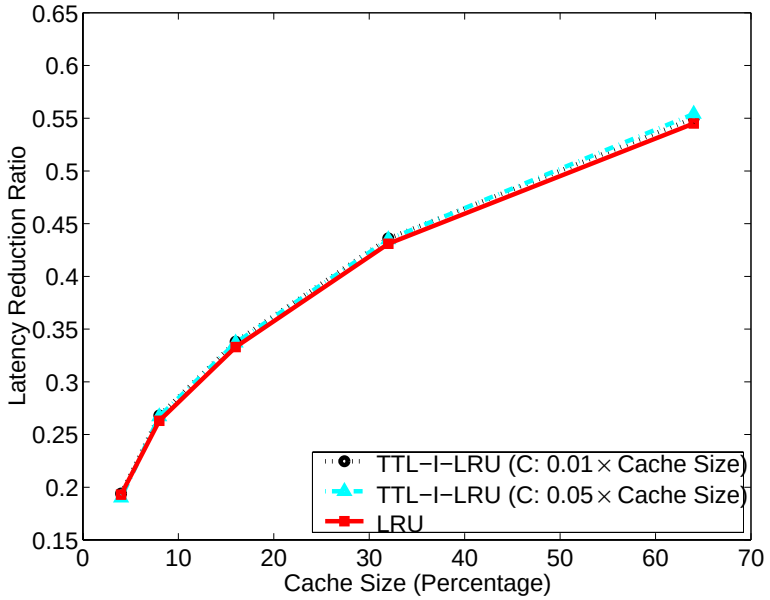
Figures 5, 6 and 7 show the influence of the choice of the boundary TTLs on the performance of SQF-MQ-LRU, EC-MQ-LRU, and PF-MQ-LRU with 2 queues respectively. These algorithms can also be called SQF-2Q-LRU, EC-2Q-LRU and PF-2Q-LRU respectively. The performance of SQF-MQ-LRU is very sensitive to the choice of the boundary TTLs. This is because it always evicts objects from one queue first. Thus, only when the boundary can be chosen correctly, can SF-MQ-LRU evict objects in the roughly correct order. See the discussion in Section 3.

The performance of EC-MQ-LRU and PF-MQ-LRU is not very sensitive to the choice of the boundary TTLs. This is because objects in both queue have opportunities to be evicted, and good choice can be made dynamically. The benefit values of the bottom objects are compared in EC-MQ-LRU. The average performance of the queues are compared in PF-MQ-LRU. Both of the algorithms can achieve good performance with different boundary TTLs.

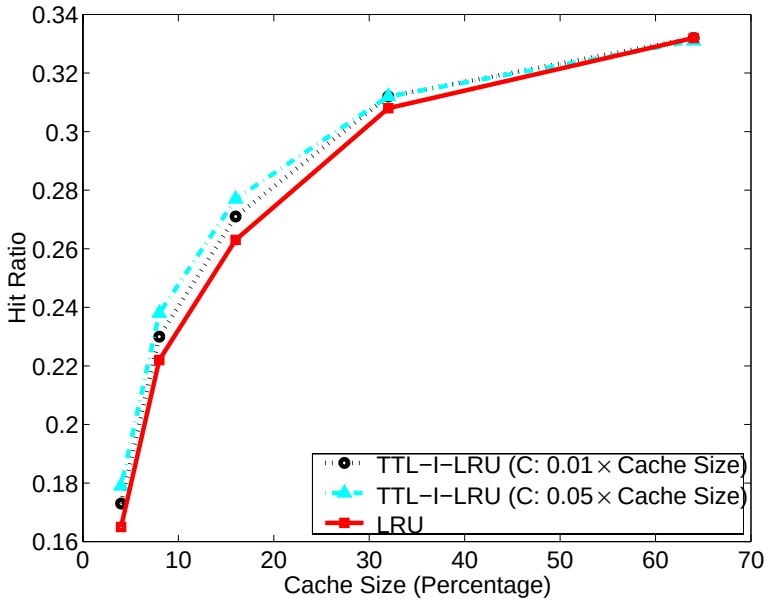
5.3 Performance Comparison

Figure 8 shows the the performance of different algorithms using their best or relatively good parameters. The improvement of TTL-I-LRU is not as significant as the TTL based MQ algorithms. The reason may be that using sequence numbers in LRU as an indication of reference frequency is not a good choice. TTL based MQ algorithms use the object's relative position in the queue as the indication of reference frequency.

An interesting result in TTL based MQ algorithms is that the best performance of all the algorithms are almost the same. Only EC-3Q-LRU performs a little better than the other algorithms in hit ratio. This maybe because all these algorithms are based on the same basic idea, and when

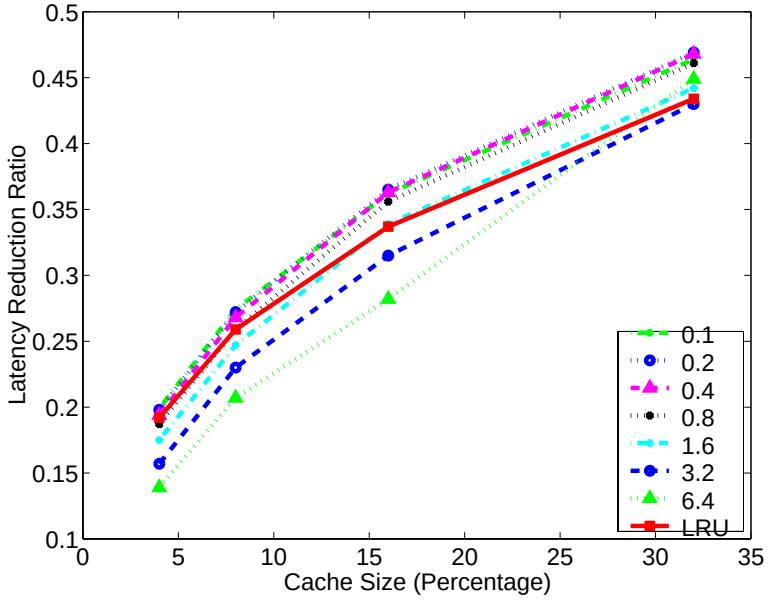


(a)

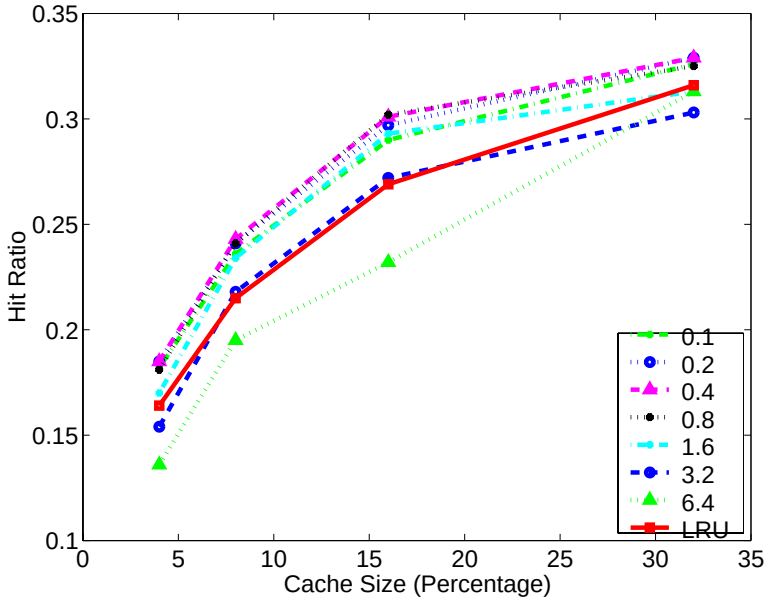


(b)

Figure 4: Performance of TTL-I-LRU algorithm

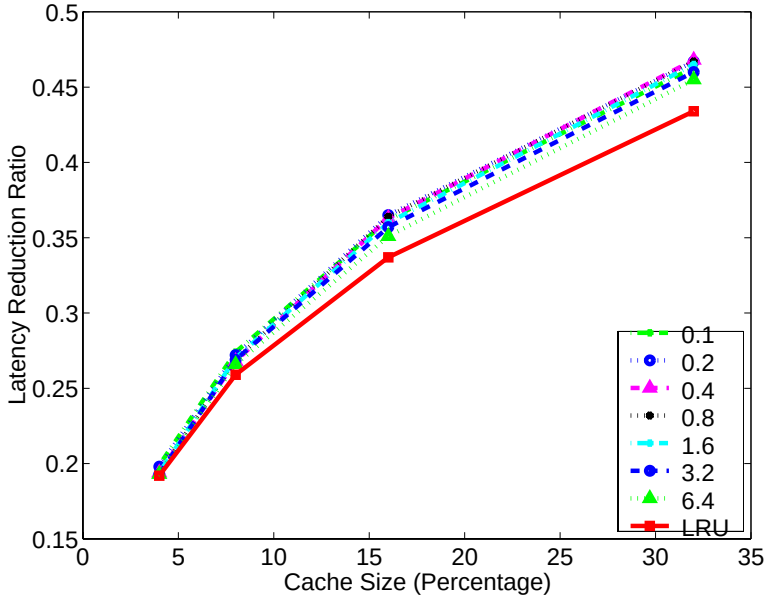


(a)

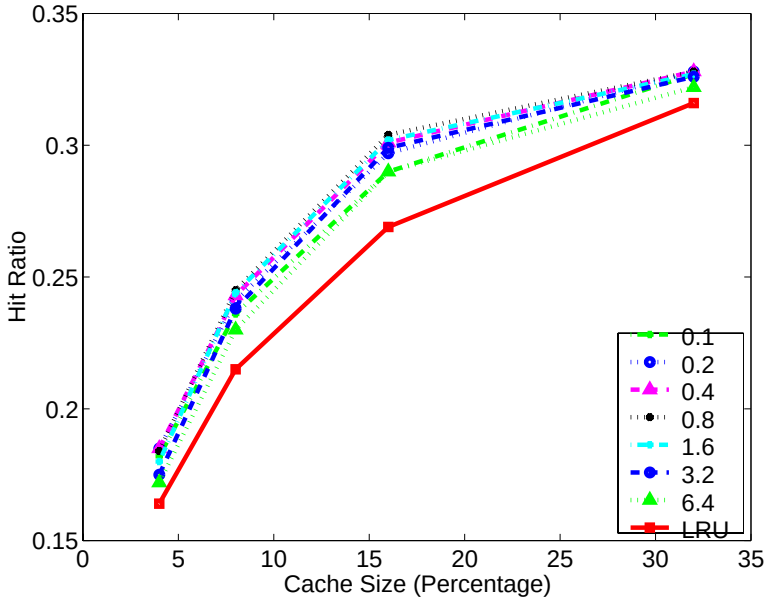


(b)

Figure 5: Performance of SQF-2Q-LRU algorithm with different TTL boundaries

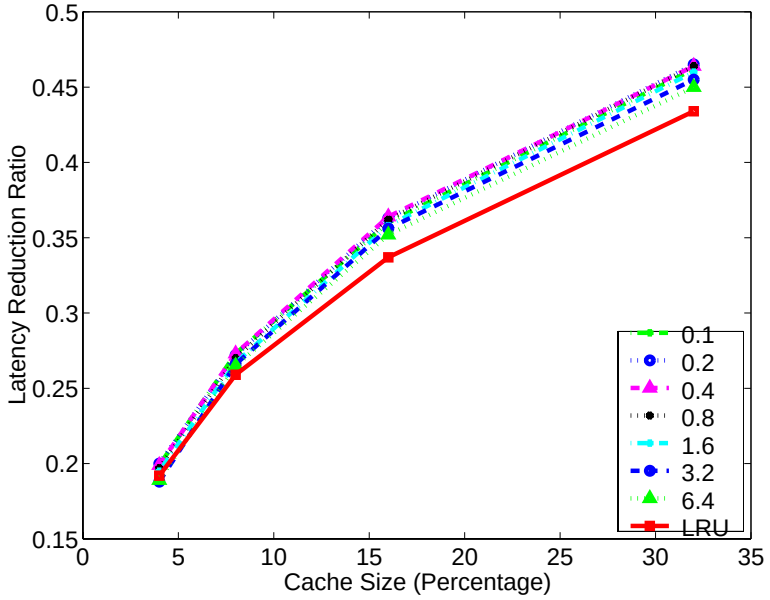


(a)

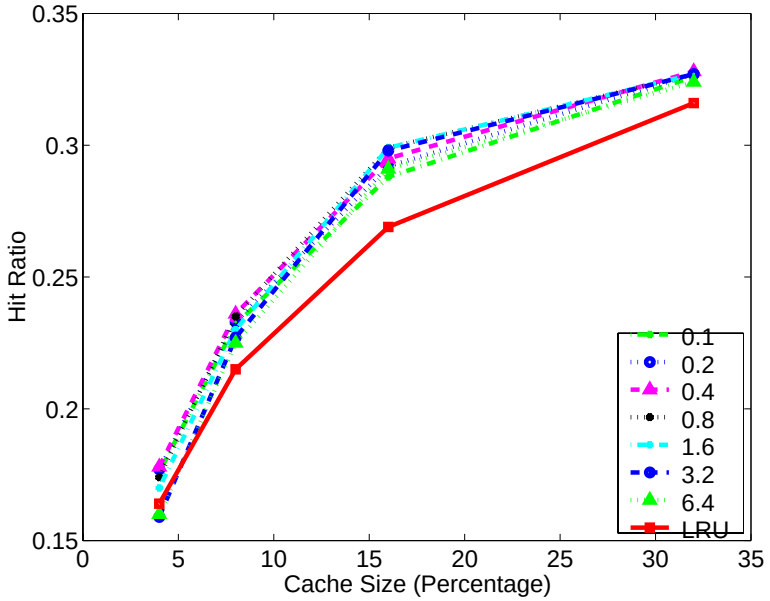


(b)

Figure 6: Performance of EC-2Q-LRU algorithm with different TTL boundaries

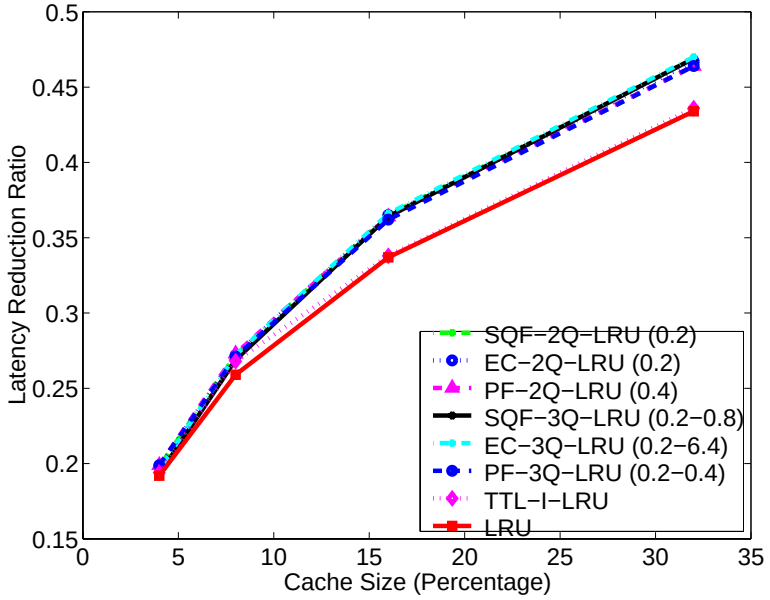


(a)

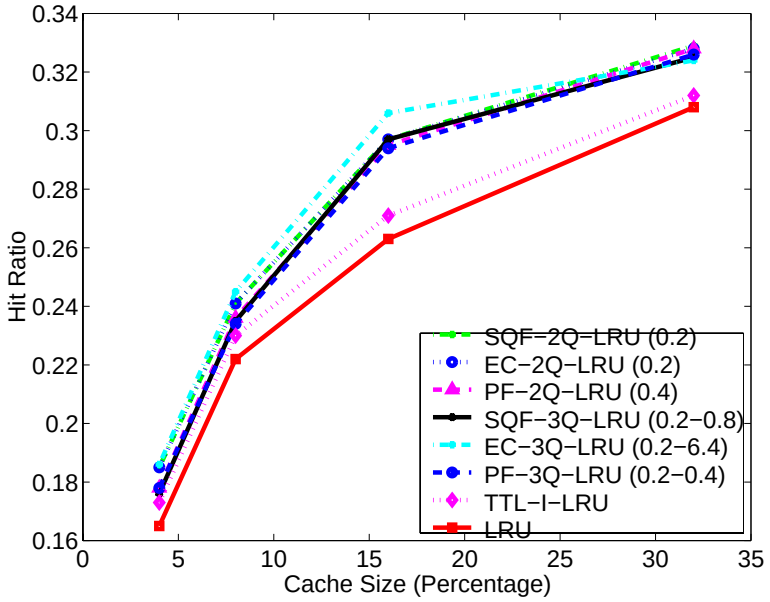


(b)

Figure 7: Performance of PF-2Q-LRU algorithm with different TTL boundaries



(a)



(b)

Figure 8: Performance comparison of different algorithms.

the boundary TTLs are chosen correctly, objects from different class can all of be evicted in a relative correct order. Since using 2 queues works well enough to distinguish the four kinds of objects defined in section 3, it seems that using 2 queues is usually good enough. It is not very necessary to use 3 queues since the boundaries are harder to choose. Although the best performance of TTL Based algorithms are very close to each other, we still prefer the End Compare approach and Performance Feedback approach to the *Short TTL* Queue first approach. This is because the *Short TTL* Queue First approach is too sensitive to the choice of boundary TTLs.

6 Related Work

While much research has been done on web cache replacement algorithms, few works consider the objects' updates in the replacement policies. Those replacement policies that explicitly consider the objects' update rates are introduced below,

Shim et. al. [10] integrate validation cost into a cache replacement policy using a profit function defined for each object. Their profit function has a similar form to Equation 1. Absolute request frequencies and change frequencies are estimated from records in the history. The performance of their algorithm is compared with LRU, and LRU-MIN. Since object size and retrieval latency are not major consideration in LRU and LRU-MIN, it is not very clear whether their algorithms improves on LRU and LRU-MIN because they consider validation cost, or only because they take size and latency into account. The trace they use is relatively small.

The GD-lifetime algorithm [7] is another way to integrate object update information with estimated request frequency. They use the "lifetime" which refer to the time that object can remain valid in the cache, as the base value for GD-Size algorithm. Their results only shows a small positive effect on the performance.

Chen et. al. [4] study the lifetime behavior of web objects. They classify web objects into four categories: highly mutable objects, stable documents, short life documents and others. The short life documents are those object that are only accessed in a few days. They claim that keeping highly mutable and short life time objects in the cache does not help to increase the cache hit ratio. So they design a two-state TTL algorithm to evict highly mutable and short life objects faster. The cache is divided into equal areas. When an object is cached the first time, it is put into the first part of the cache. After a short time, if the object is still valid, it is moved to the second part of the cache and is assigned a longer TTL. They claim a 2.8% hit ratio improvement on average vs. two other TTL consistency algorithms.

7 Future Work

The current experiment is conducted using a statistical generative request and object update model. The results are very encouraging. Our next step is to test the performance of the above algorithms on the real web access traces. Various workloads will be of interesting.

We also want to study how to decide the boundary of the queues in TTL based Multi-Queue algorithms automatically by analyzing the objects TTL distribution in the cache.

The current caching scheme does not actively refresh objects after they expire. For those frequently accessed and frequently changed data, active refreshing might be useful in improving the performance. Thus, we want to study how to use the limited bandwidth to achieve the best performance improvement.

8 Conclusion

Web object's are usually updated autonomously. When a weak data consistency policy is used, web caches must pay validation costs for the requests of expired data. If the object request rate is relatively slow there will be a fair amount of expired objects in the cache, and the the performance of caching can be significantly degraded. In this work, we present two different approaches to integrate the validation cost into web caching algorithms. By evicting infrequently accessed but frequently changed objects before other objects, these algorithms can achieve better performance than algorithms that do not consider validation cost. The experiments in this work are based on the generative request and object update model. In our future work, we will test the performance of those algorithms on real web access traces.

References

[1] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, *Web caching and Zipf-like distributions: Evidence and implications*, Proceedings of the INFOCOM '99 conference, March 1999, `http://www.cs.wisc.edu/~cao/papers/zipf-like.ps.gz`.

[2] Brian E. Brewington and George Cybenko, *How dynamic is the web?*, Proceedings of the 9th International WWW Conference, May 2000, `http://www9.org/w9cdrom/264/264.html`.

[3] Pei Cao and Sandy Irani, *Cost-aware WWW proxy caching algorithms*, Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97) (Monterey, CA), December 1997, `http://www.cs.wisc.edu/~cao/papers/gd-size.ps.Z`.

[4] X. Chen and P. Mohapatra, *Lifetime behavior and its impact on web caching*, July 1999, `http://citeseer.nj.nec.com/chen99lifetime.html`.

[5] Junghoo Cho and Hector Garcia-Molina, *Synchronizing a database to improve freshnessr*, Proc. of ACM SIGMOD, 2000, `http:://citeseer.nj.nec.com/cho00synchronizing.html`.

[6] Sohudonohudong Jin and Azer Bestavros, *GreedyDual* Web caching algorithms: Exploiting the two sources of temporal locality in Web request streams*, Proceedings of the 5th International Web Caching and Content Delivery Workshop, May 2000, `http://www.terena.nl/conf/wcw/Proceedings/S2/S2-2.pdf`.

[7] Balachander Krishnamurthy and Craig Wills, *Proxy cache coherency and replacement – towards a more complete picture*, Proceedings of the ICDCS conference, June 1999, `http://www.research.att.com/~bala/papers/ccrcp.ps.gz`.

[8] Balachander Krishnamurthy and Craig E. Wills, *Piggyback server invalidation for proxy cache coherency*, Computer Networks and ISDN Systems **30** (1998), no. 1-7, 185–193, `http://www.elsevier.nl/cas/tree/store/comnet/sub/1998/30/1-7/1844.pdf`.

[9] Luigi Rizzo and Lorenzo Vicisano, *Replacement policies for a proxy cache*, Tech. Report RN/98/13, UCL-CS, 1998, `http://www.iet.unipi.it/~luigi/lrv98.ps.gz`.

[10] Junho Shim, Peter Scheuermann, and Radek Vingralek, *Proxy cache design: Algorithms, implementation and performance*, IEEE Transactions on Knowledge and Data Engineering (1999), `http://www.ece.nwu.edu/~shimjh/publication/tkde98.ps`.