The Potential Costs and Benefits of Long-term Prefetching for Content Distribution

Arun Venkataramani Praveen Yalagandula Ravi Kokku Sadia Sharif Mike Dahlin

The University of Texas at Austin

Abstract

This paper examines the costs and potential benefits of long-term prefetching for content distribution. In contrast with traditional short-term prefetching, in which caches use recent access history to predict and prefetch objects likely to be referenced in the near future, long-term prefetching uses long-term steady-state object access rates and update frequencies to identify objects to replicate to content distribution locations. Compared to demand caching, long-term prefetching increases network bandwidth and disk space costs but may benefit a system by improving hit rates. Using analytic models and trace-based simulations, we examine several algorithms for selecting objects for long-term prefetching. We find that although the web's Zipf-like object popularities makes it challenging to prefetch enough objects to significantly improve hit rates, systems can achieve significant benefits at modest costs by focusing their attention on long-lived objects.

1 Introduction

As network bandwidths improve, network load will become a less important issue relative to client perceived quality of service. Web proxy caching reduces client perceived latency by storing recently referenced content closer to the users. However cache hit rates have not improved much in the past few years. Even if cache space is unlimited, uncacheable data, consistency misses for cached objects, and compulsory misses for new content cannot be avoided by a passive caching approach.

Prefetching attempts to overcome these limitations of passive caching by pro-actively fetching content without waiting for user requests. Traditional short-term prefetching uses recent client access history to predict and prefetch objects likely to be referenced in the near future. This approach is an appropriate strategy for clients and can significantly improve hit rates [4,5,6,14].

We focus on a technique more appropriate for large proxies and content distribution networks(CDNs) – long-term prefetching. Rather than basing prefetching decisions on the recent history of a client, long term prefetching seeks to increase hit rates by using global object access patterns to identify a collection of valuable objects to replicate to caches and content distribution servers.

In March 2001 an 80GB disk drive costs about $250, which makes it possible to store an enormous collection of data at a large content distribution site. However, maintaining a collection of hundreds of gigabytes or several terabytes of useful web data incurs not just a space cost but also a bandwidth cost: as object in the collection change, the system must fetch new versions of existing objects and must fetch newly created objects that meet the collection selection criteria. Given the large number of objects that must be cached to significantly improve hit rates due to Zipf-like access patterns of the web [5], it appears challenging to significantly improve web hit rates by maintaining such a collection. In particular it seems that bandwidth will be the primary constraint on long term prefetching.

In this paper, we present a model for understanding steady-state cache behavior in a bandwidth-constrained long-term prefetching environment. We then present and evaluate a long term prefetching policy based on both object request rates and lifetimes. Our hypothesis is that by prefetching objects that have reasonably high probabilities of access before they are updated-those that are both long-lived and popular, we can significantly improve hit rates for moderate bandwidth costs. We define the *prefetch quotient* of an object as the probability that it would be accessed before it is updated. We place a threshold on the prefetch quotients of objects and select them for prefetching if they cross the threshold.

The key contribution of our work is a threshold algorithm for long term prefetching that balances object access frequency and object update frequency and that only fetches objects whose probability of being accessed before being updated exceeds a specified threshold. Using synthetic and real proxy trace based simulations we establish that our algorithm provides significant hit rate improvements at moderate storage and bandwidth costs. For example for a modest-size cache that receives 10 demand requests per second, long-term prefetching can improve steady state hit rates for cacheable data from about 62% (for an infinite demand-only cache) to above 75% while increasing the bandwidth demands of the system by less than a factor of 2. More generally, we quantify the trade-offs involved in choosing a reasonable prefetch threshold for a given object access rate. Based on our trace based simulation, we conclude that the key challenge to deploying such algorithms is developing good predictors of global access rates; although we leave development of such predictors as future work, we provide initial evidence that simple predictors may work well.

The rest of the paper is organized as follows. Section 2 provides some background information about web traffic characteristics and also discusses a possible deployment scenario for long-term prefetching. Section 3 presents the model and algorithms that lay the theoretical groundwork for long-term prefetching. Section 4 discusses the methodology we used to evaluate long-term prefetching. As described later, both synthetically generated and real proxy traces were used in the simulations. Section 5 discusses the results of our simulations and provides further insights about how long-term prefetching works. Section 6 discusses some related work. Section 7 provides a summary of our conclusions.

2 Background

This section describes five key parameters of web workloads that determine the effectiveness of caching and prefetching: models of prefetching, locality of access, object size, object updates/new object creation, and the availability of spare bandwidth for prefetching. A possible deployment scenario is also discussed.

2.1 Prefetching models

We categorize prefetching schemes into two groups: short-term and long-term. In the short-term model, a cache's recent requests are observed and likely near-term future requests are predicted. Based on these predictions, the objects are prefetched. Considerable research has been performed on this type of model [16, 19, 29], most of which are based on variations of a Prediction-by-Partial-Matching (PPM) strategy [14].

In the long-term model of prefetching on which we focus, we assume that a cache or content distribution site maintains a collection of replicated objects based on global access pattern statistics such as object popularity and update rates. We envision a hierarchical structure for content distribution with lower level caches (proxy caches) primarily focusing on servicing client requests and the higher level caches (content distribution servers) on effective content distribution using long-term prefetching. Proxy caches can use short term prefetching to improve hit rates further. Content servers maintain a collection of popular objects and update these objects as they change. New objects are added to the collection based on server assistance and user access.

The content distribution system requires four components:

1. *Statistics tracking.* Our selection algorithm uses as input: (i) estimates of object lifetimes and (ii) estimates of access frequency to objects. Maintaining these estimates is a key challenge to deploying a long-term prefetching based system, and we do not address this problem in detail.

   If content servers are trusted by the content distribution system, they may be able to provide good estimates. Otherwise, the system itself must gather access probability reports from clients or caches and track object update rates. For example, a distributed federation of caches and content distribution nodes could gather local object access distributions and report these statistics to a central aggregation site which would distribute the aggregate statistics to the caches and nodes. There is some evidence that relatively short windows of time can provide good access estimates [23].

2. *Selection criteria.* Based on the statistics, the selection criteria module determines which objects should be included in the replica's collection. The rest of this paper discusses this issue in detail.

3. *Data and update distribution.* The system must distribute objects and updates to objects to caches that include the objects in their collection of replicated objects. We model a push-based system in which updates to replicas are sent immediately to caches that have "subscribed" to the object in question. We leave the details of constructing such a system as future work.

4. *Request redirection.* In order to enable clients to transparently access a nearby copy of a replicated object, an effective redirection scheme is needed. A number of experimental [18, 35] and commercial systems [1, 2] address this issue.

2.2 Popularity distributions

A key parameter for understanding long-term prefetching is the distribution of requests across objects. Several studies [3, 13, 20, 33] have found that the relative distribution with which Web pages are accessed follows a Zipf-like distribution. Zipf's law states that the relative probability of a request for the *i*'th most popular object page is inversely proportional to *i*. Cunha et al. [13] found that the request probability for a Web cache trace, when fitted with a curve of the form $1/i^\alpha$, yields a curve with exponent of $\alpha = 0.982$ which we will use as a default parameter. Other researchers have reached similar conclusions [5].

According to this model, given an universe of N Web pages, the relative probability of *i*th most popular page is

$$p_i = \frac{C}{i^\alpha}, \text{ where } C = \frac{1}{\sum_{k=0}^{N}\left(\frac{1}{k^\alpha}\right)} \quad (1)$$

For our synthetic workload, we will use this model of accesses with $N = 10^9$, $\alpha = .982$, and $C = 0.0389$.

2.3 Object sizes

Studies by Barford and Crovella [12] show that web object sizes exhibit a distribution that is a hybrid of a log-normal and a heavy tailed Pareto distribution. The average size of a web object has been shown to be around 13KB. Work by Breslau et al. [5] suggests that there is little or no correlation between object sizes and their popularity. However, an earlier study by Crovella et al. [13] claims an inverse relationship between object sizes and popularities, *i.e.* users *prefer* small documents. They show a weak Zipf correlation between popularity and size with a zipf parameter -0.33. However we did not observe an appreciable correlation between object sizes and popularity in the Squid traces we analyzed. Hence, for our simulations we do not assume any correlation. It must, however, be emphasized that if the inverse correlation were to be assumed, a prefetching strategy based on maintaining popular objects will perform better with respect to both bandwidth and cache size.

2.4 Update patterns and lifetimes

Web objects have two sources of change - (i) updates to objects that are already present, (ii) introduction of new objects. The work by Douglis et al. [15] shows that (mean) lifetimes of web objects are distributed with an overall mean of about 1.8 months for html files and 3.8 months for image files. Though they analyzed lifetimes for objects in varying popularity classes, little correlation is observed between lifetime and popularity. The work by Breslau et al. [5] further strengthens the case for lack of strong correlation between lifetime, popularity and size of objects.

The lifetime distribution for a single object over time is found to be exponential in [6]. They consider the Internet as an exponentially growing universe of objects with each object changing at time intervals determined by an exponential distribution. Their analysis shows that the age distribution of an exponentially growing population of objects with (identical) exponential age distributions remains exponential with the parameter given by the sum of the population growth and object update rate constants. They then show with respect to the cost of maintaining a collection of fresh popular objects that the introduction of new objects on the Internet is equivalent to changing objects in a static universe of objects with a different rate parameter.

In our simulations we use the data for lifetime distribution presented in [15]. We assume no correlation with popularity or size. Our criterion for selecting an object is based on its current popularity and mean lifetime and is independent of its past and of other objects. We therefore describe our algorithm in terms of the bandwidth cost to update a fixed collection of objects as they are updated. But following analysis done by Brewington et al. [6], our algorithm and analysis also apply to the case of maintaining a changing collection of objects that meet the system's replication selection criteria. We explain this assumption in greater detail in section 4.

2.5 Spare prefetch resources

Prefetching increases system resource demands in order to improve response time. This increase arises because not all objects prefetched end up being used. Resources consumed by prefetching include server CPU cycles, server disk I/O's, and network bandwidth. Therefore, a key issue in understanding prefetching is to determine an appropriate balance between increased resource consumption and improved response time.

Unfortunately, system resources and response time are not directly comparable quantities, and the appropriate balance depends on the value of improved response time and on the amount of "spare" system resources that can be consumed by prefetching without interfering with demand requests. For example, if a system has ample spare bandwidth, it may be justified to, say, quadruple bandwidth demands to improve response time by, say, 20%, but in other circumstances such a trade-off would be unwise.

Often, prefetch algorithms explicitly calculate the probability that a candidate for prefetching will be used. For such algorithms, it is natural to specify a *prefetch threshold* and to prefetch objects whose probability of use exceeds the prefetch threshold. This approach limits the excess resources consumed by prefetching to a factor of at most $\frac{1}{threshold}$ times more resources than a demand system. Note that the total amount of resource expansion may remain significantly below this upper bound because systems may not prefetch all objects and because some objects may attain a higher useful prefetch fractions than enforced by this threshold.

Although determining an appropriate prefetch threshold for a system is challenging, several factors support the position that aggressive prefetching can be justified even if it "wastes" system resources.

- *User time is valuable.* If bandwidth is cheap and human waiting time expensive, prefetching can be justified even if it significantly increases bandwidth demands and only modestly improves response times. For example, Duchamp argues for a prefetch threshold of 0.25 in his hyperlink prefetching system [16], and Chandra et al. [9] argue that thresholds as low as 0.01 may be justified given current WAN network transfer costs and human waiting time values.
- *Technology trends favor increased prefetching in the future.* The prices of computing, storage, and communications fall rapidly over time, while the value of human waiting time remains approximately constant.
- *Prefetch requests may be less expensive to serve than demand requests for the same amount of data.* Servers may schedule prefetch requests to be handled in the background, and networks may benefit from the reduced burstiness of prefetch traffic [12]. Furthermore, techniques such as multicast, digital fountain [7], delta-encoding [27], and satellite links appear well suited to long-term prefetching for content distribution and allow data transmission at a much lower cost than traditional network transmission.

Overall, we conclude that if aggressive prefetching is shown to significantly improve response time, the infrastructure can and will be built to accommodate it.

3 Model and algorithms

In contrast with traditional caching, where space is the primary limiting factor, for long-term prefetching bandwidth is likely to be the primary limiting factor. In this section, we first describe an equilibrium model useful for understanding the dynamics of bandwidth-constrained long-term prefetching. We then describe our algorithms.

3.1 Bandwidth equilibrium

A long-term prefetching system attempts to maintain a collection of object replicas as these objects change and new objects are introduced into the system.

Figure 1 illustrates the forces that drive the collection of fresh objects stored in a cache towards equilibrium. New objects are inserted into the cache by demand requests that miss in the cache and by prefetches. Objects

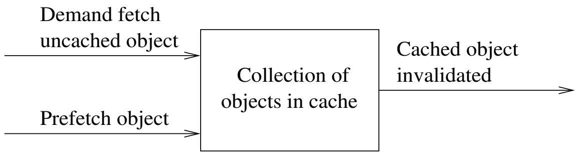


Figure 1: Equilibrium in bandwidth-constrained cache.

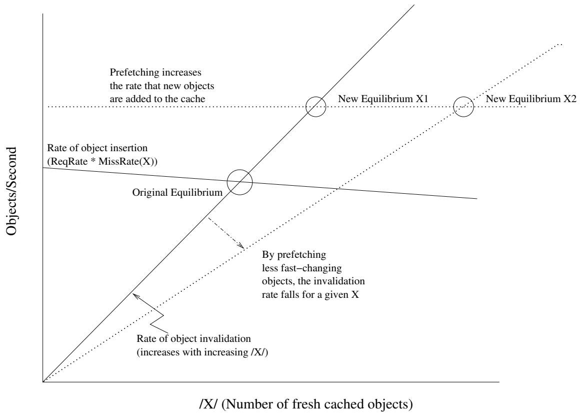


Figure 2: Equilibrium in bandwidth-constrained cache.

are removed from the set of fresh objects in the cache when servers update cached objects, invalidating the cached copy.[1]

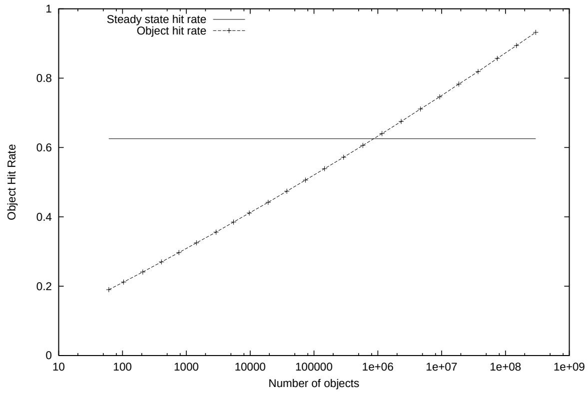
The solid lines in Figure 2 illustrates how this equilibrium is attained for a demand-only cache with no prefetching. Let X be the set of fresh objects in the cache at a given moment, and let $|X|$ denote the number of objects in this set. If the number of requests per second being sent to the cache is $ReqRate$, then the rate of object insertion into this set is $ReqRate \cdot MissRate(X)$. For a given request rate, the miss rate typically falls slowly as $|X|$ increases [17, 21], and the rate of insertion falls with it. At the same time the rate of invalidations (or expirations) of cached objects increases as $|X|$ increases. As the figure illustrates, these factors combine to yield an equilibrium collection of objects that can be maintained in the cache.

The dotted lines in Figure 2 illustrate how prefetching can change this equilibrium. First, as the horizontal line illustrates, prefetching increases the rate at which new objects are added to X. If the collection of objects prefetched have similar lifetimes to the collection of objects fetched on demand, then invalidation rates will behave in a similar fashion, and a new equilibrium with a larger set X will be attained as shown by the point labeled *New Equilibrium* X_1.

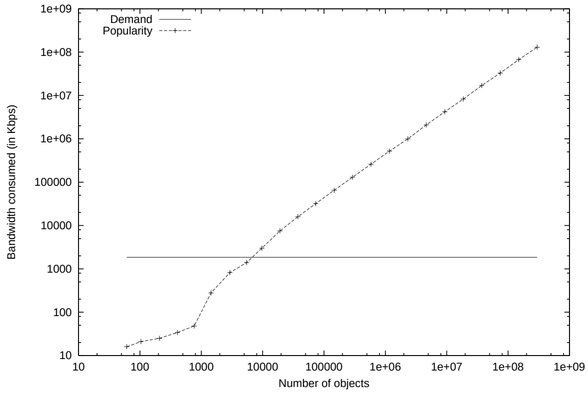
A prefetching system, however, has another degree of freedom: it can choose what objects to prefetch. If a prefetching system chooses to prefetch relatively long-lived objects, its invalidation rate for a given number of prefetched objects $|X|$ may be smaller than the invalidation rate for the same number of demand fetched objects. This change has the effect of shifting the invalidation rate line down, and yields a new equilibrium, *New Equilibrium* X_2, with $|X_2| > |X_1|$.

A potential disadvantage of preferentially prefetching long-lived objects is that the system may thereby reduce the number of frequently-referenced objects it prefetches. In particular, although $|X_2| > |X_1|$, if the

[1] For simplicity, we describe a system in which servers invalidate clients' cached objects when they are updated [11, 24, 26, 34]. Client-polling consistency would yield essentially the same model: in that case, objects that expire are removed from the set of objects that may be accessed without contacting the server.



(a) Hit rate



(b) Bandwidth

Figure 3: (a) Hit rate and (b) bandwidth consumed by prefetching as k, the number of prefetched objects, is varied for the Popularity algorithm.

objects in X_1 are more popular than the objects in X_2, the hit rate for equilibrium X_1 may exceed the hit rate for equilibrium X_2.

3.2 Prefetching algorithms

In this paper, we consider two prefetching algorithms. The first attempts to maintain an equilibrium cache contents containing the most popular objects in the system. The second attempts to balance object popularity (which represents the benefit of keeping an object) against object update rate (which represents the cost of keeping an object.)

3.2.1 Popularity

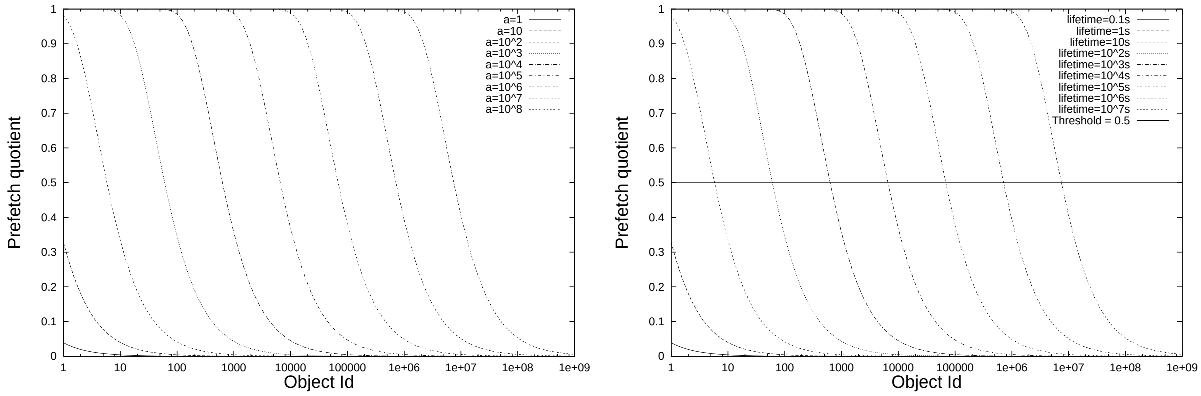
The Popularity algorithm identifies the k most popular objects in the universe and maintains copies of them in the cache. Whenever any one of these objects is updated (or a new object joins the set of the most popular k objects), the system fetches the new object into the cache immediately.

Figure 3 shows the hit rate achieved and bandwidth consumed by the Popularity algorithm when it prefetches the k most popular objects. Although this approach can achieve high hit rates, its bandwidth costs are high. In particular, to improve the hit rate by 10% compared to the steady state hit rate of a demand cache that receives 10 requests per second, the Popularity algorithm must consume at least 1000 times more bandwidth than such a demand cache.

Because of these high bandwidth demands, we do not consider the Popularity algorithm further in this paper.

3.2.2 Threshold

The Threshold algorithm balances object access frequency and object update frequency and only fetches objects whose probability of being accessed before being updated exceeds a specified threshold. In particular, for object i, given the object's lifetime $lifetime_i$, the probability that a request will access that object P_i, and the total request rate of demand requests to the cache $requestRate$ and assuming that object references and updates are independent, the probability that a cache will access an object that it prefetches before that object dies is



(a) Threshold map

(b) Example

Figure 4: Prefetch threshold v. object popularity for lines of constant $objectLifetime \cdot cacheRequestRate$. (a) shows the threshold map and (b) illustrates the set of objects that will be prefetched for a threshold of 0.5 and a request rate of 10 request/second.

$$P_{goodFetch} = 1 - (1 - P_i)^{lifetime_i \cdot requestRate} \tag{2}$$

$lifetime_i * requestRate$ is the total number of requests to the cache expected during the object lifetime, and $(1 - P_i)^{lifetime_i \cdot requestRate}$ is the probability that none of these requests access object i.

The Threshold algorithm prefetches the collection of objects whose $P_{goodFetch}$ exceeds some threshold. Note that this definition of threshold is similar to that used by several short-term prefetching algorithms [16], and it provides a natural way to limit the bandwidth wasted by prefetching. For each object prefetched, we will consume at most $\frac{1}{threshold}$ times more bandwidth accessing that object than a demand system. Note that the total amount of wasted bandwidth may remain significantly below this value because the system does not prefetch all objects and because some objects will attain a higher useful prefetch fraction than enforced by this threshold.

To aid in understanding this algorithm as it applies to web caching, Figure 4-a shows a ”contour map” that relates these thresholds to a Zipf-like popularity distribution. We sort objects by popularity and the x-axis represents the object index i in this sorted order. Assuming accesses to objects follows a Zipf-like distribution $P_i \approx \frac{C}{i^\alpha}$, P_i falls slowly with increasing i. Each line in the figure represents a line of constant $objectLifetime \cdot cacheRequestRate$, and the y-axis represents $P_{goodFetch}$ threshold values. $ObjectLifetime$ depends on the particular object chosen and $cacheRequestRate$ depends on the demand fetch rate to the cache under consideration. Given a proposed threshold for a particular cache, one can determine what sets of objects will qualify for prefetching based on their popularities and update rates. For example, Figure 4-b illustrates the set of objects that will be prefetched for a threshold of 0.5 and a request rate of 10 request/second. Objects among the top-1000 are worth prefetching if they will live at least 10^3 seconds (about 17 minutes); in contrast objects among the top-1,000,000 are worth preserving if they will live at least 10^6 seconds (about 12 days). Given that a significant fraction of objects live for 30 days or longer [15], a busy cache such as this may be justified in prefetching a large collection of objects.

Overall, it appears that long-term bandwidth-constrained prefetching will be most attractive for relatively large, busy caches. Reducing the request rate to a cache by an order of magnitude reduces the set of objects eligible to be prefetched to that cache for a given threshold by approximately an order of magnitude.

4 Methodology

In the previous section we introduced the Threshold algorithm that computes a collection of objects for which the probability of access before it gets updated exceeds a certain threshold. We claimed that this algorithm balances object access frequency and update frequency thereby resulting in minimal wasted prefetch bandwidth and still gives attractive improvements in hit rate. In order to verify this, we perform a cost-benefit analysis of the Threshold algorithm in terms of improvement in hit rate, bandwidth consumption and cache size through two sets of simulation experiments - (i) based on a synthetically generated set of 1 billion objects, and (ii) a proxy trace based simulator implementing a prediction based version of the Threshold algorithm. The synthetically generated workload experiments fundamentally give a proof of concept for the performance of the Threshold algorithm. A key benefit of using a synthetic workload is that it allows us to model global object popularities, including objects that have not been accessed in the trace of a particular cache. On the other hand the proxy trace based experiments analyze the performance of the implementation of an adaptive, prediction based version of the Threshold algorithm on a smaller but realistic workload. This workload automatically exhibits temporal locality between accesses to the same object, their size distribution and models burstiness in request traffic as well. It is directly comparable to the performance of real web proxies and hence serves as a sanity check.

4.1 Simulation methodology

For simplicity, we assume the demand request arrival rate follow a Poisson distribution (We discuss the effect of this assumption later in the paper). For calculation of steady state hit rates, we assume that the request arrival rate has a mean of a requests per second. We define the lifetime of an object as the time between two modifications to that object. The lifetimes of an object are known to follow an exponential distribution [6]. In our model, we assume that the average lifetime of an object i to be l_i. As previously explained, the probability of an access to an object with popularity i follows a zipf-like distribution. We assume the objects are indexed with respect to their popularities. Hence the probability of a request is for an object with index i is $p_i = C * (1/i^{\alpha})$.

Steady state demand hit rate

In this section we present a closed form expression for calculating the steady state hit rate of a demand cache with an infinite cache size. An analytical expression is necessary as the simulation based study is not plausible for calculating the steady state parameters because of the huge number of objects and hence the long time requirements for cache warm up. We define $P_{A_i}(t)$ as the probability for accessing an object i at t time units before present time and $P_{B_i}(t)$ as the probability of no updates done to the object i in t time. Now, the probability of a hit on a request to a demand cache is

$$P_{hit_d}(i) \quad = \quad \sum_i p_i \int_0^{\infty} P_{A_i}(t) P_B(t) dt, \tag{3}$$

Suppose $P_{(a,t)}(k)$ is the probability of k accesses occurring in t time given an access arrival rate of a. With the assumption of request arrivals following Poisson distribution, the probability of k arrivals occurring in t seconds is $P_{(a,t)}(k) = e^{-at}.\frac{(at)^k}{k!}$. The probability of object i being accessed on a request is p_i. The probability of no accesses to object i in this time t is

$$P(0 \text{ accesses to object } i \text{ in } t \text{ time}) \quad = \quad \sum_{k=0}^{\infty} P(k \text{ requests in } t) P(\text{none of these } k \text{ requests is for } i)$$

$$
\begin{aligned}
&= \sum_{k=0}^{\infty} \left(e^{-at} \frac{(at)^k}{k!} \right) (1-p_i)^k, \\
&= e^{-at} \sum_{k=0}^{\infty} \frac{(at(1-p_i))^k}{k!} \\
&= e^{-at} e^{at(1-p_i)} \\
&= e^{-(ap_i)t}
\end{aligned}
$$

The above equation implies that the inter access rates to an object *i* follow an exponential distribution with a mean time of $(1/ap_i)$. Hence, the probability of an access to an object *i* occurring *t* time units after an access to the same object is $(ap_i)e^{-(ap_i)t}$, which is also the distribution for $P_{A_i}(t)$. Hence,

$$P_{A_i}(t) = (ap_i)e^{-(ap_i)t} \tag{4}$$

Given that the lifetimes of an object *i* are exponentially distributed with an average l_i, the probability of no updates to that object happen in time *t* is

$$P_{B_i}(t) = e^{-t/l} \tag{5}$$

From Equations 3, 4 and 5, the probability of hit on an access will be

$$
\begin{aligned}
P_{hit_d}(i) &= \sum_i p_i \int_0^{\infty} \left((ap_i)e^{-(ap_i)t} \right) \left(e^{-t/l} \right) dt \\
&= \sum_i p_i (ap_i) \left(\left. \frac{e^{-(ap_i+1/l)t}}{-(ap_i+1/l)t} \right|_0^{\infty} \right) \\
&= \sum_i p_i \left(\frac{ap_i l}{ap_i l + 1} \right) \qquad (6)
\end{aligned}
$$

The fraction $\frac{ap_i l}{ap_i l+1}$ represents the hit rate among accesses to the object *i*. Stated otherwise, this denotes the probability of object *i* being fresh when it is being accessed. We also call this as *freshness factor* of object *i* denoted as $ff(i)$.

Steady state prefetch hit rate

In the Threshold algorithm we propose for prefetching objects, an object is always kept fresh by prefetching it, when any change occurs, if its $P_{goodFetch}$ as calculated in Equation 2 is above chosen threshold value *T*. For an object i, if $P_{goodFetch}(i)$ is more than the chosen threshold value T, then we will have a hit on that object for all accesses. For other objects the hit rate remains same as calculated in previous which will be $ff(i)$. Hence, the steady state hit rate in a prefetch based scheme with threshold value *T* is

$$P_{hit_p}(i,T) = \sum_i p_i h_i, \text{ where} \tag{7}$$

$$h_i = \begin{cases} 1 & \text{if } P_{goodFetch}(i) > \text{T} \\ \frac{ap_i l}{ap_i l+1} & \text{otherwise} \end{cases}$$

Steady state cache sizes

We estimate the steady state demand and prefetch cache sizes in this section. The estimation of these sizes for a billion object real web workload mean a long run of simulation. Assuming a billion object web with 13KB average size, a pessimistic steady state demand cache size is 13TB. But the objects get updated and hence we do not need to keep all objects in cache.

In steady state, the probability of an object *i* being fresh in the cache is given by $\int_0^\infty P_{C_i}(t)P_{B_i}(t)dt$, where $P_{C_i}(t)$ is the probability of last access to object i is t time units before now, and P_{B_i} is as defined in the previous section. The probability $P_{C_i}(t)$ will be same as P_{A_i} defined in the previous section as the probability of getting access to an object *i* at time *t* follows an exponential distribution, which is memoryless. Hence the probability of an object i being fresh in the cache at some random time instance is also the freshness factor of i, $ff(i)$. Given the freshness factors of objects, the estimated steady state cache size is $\sum_i s_i ff(i)$. Hence, the estimated demand cache size in steady state, $CSize_{ss_d}$, is

$$CSize_{ss_d} = \sum_i s_i \frac{ap_i l}{ap_i l + 1} \tag{8}$$

When using the Threshold algorithm for prefetching with threshold value T, the estimated prefetch cache size is calculated as

$$CSize_{ss_p} = \sum_i s_i * f_i, where \tag{9}$$

$$f_i = \begin{cases} 1 & \text{if } P_{goodPrefetch}(i) > \text{T} \\ \frac{ap_i l}{ap_i l+1} & \text{otherwise} \end{cases}$$

Steady State Bandwidth

We present the steady state bandwidth requirements for both demand based access methods and the Threshold algorithm based scheme. The estimated steady state bandwidth in the case of just demand fetches is

$$BW_{ss_d} = \sum_i s_i(1 - ff(i))ap_i \tag{10}$$

For Threshold algorithm based prefetch strategy,

$$BW_{ss_p} = \sum_i s_i * h_i, where \tag{11}$$

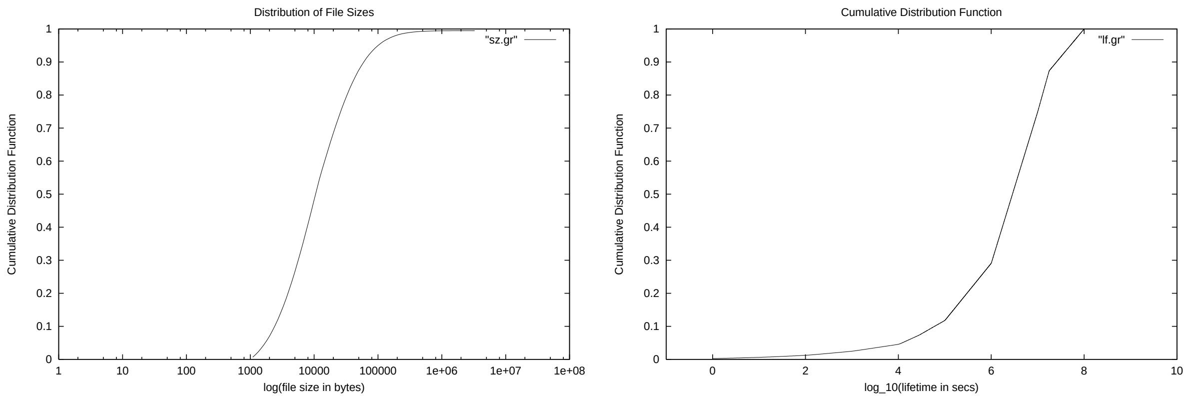
$$h_i = \begin{cases} 1 & \text{if } P_{goodFetch}(i) > \text{T} \\ ap_i(1 - ff(i)) & \text{otherwise} \end{cases}$$

The proofs for these derivations are presented in Appendix A.

4.2 Simulation parameters

The synthetic workload simulator assumes a set of 1 billion objects that exhibit a Zipf like popularity distribution with the Zipf parameter $\alpha = -0.982$. The sizes of the objects are assumed to follow a *log-normal* +

pareto like distribution as explained in [4]. We assume that there is no correlation between object sizes and their popularities. The distribution of object lifetimes was taken from [15] again assuming no correlation with popularity or size. The analysis in [5] support these assumptions of the non existence of any observable correlation between the popularity, lifetime and size of objects. Figures 4.2 show the cumulative distribution functions for object sizes and lifetimes.



(a) Object sizes (b) Object lifetimes

Figure 5: Cumulative distribution

All data is considered to be cacheable. This assumption is justified since we are concerned only with the improvement in hit rate because of prefetching and uncacheable data affects both demand and long term prefetch caching alike. We also speculate that efforts such as active caches [8], active names [31], and ICAP [30] to support execution of server program at caches and Content Distribution Nodes. Also the efforts to improve cache consistency semantics [34, 11, 24, 26] will enable caching of much currently uncacheable data. To obtain the steady state hit rate we need to simulate a trace consisting of several billion records. Since this takes an inordinate amount of time, we resort to the expression derived in the previous section to compute the steady state hit rate of an infinite-size demand cache. The bandwidth consumption for the prefetching strategy is computed by summing $size_i/lifetime_i$ over all prefetched objects with ($P_{goodFetch} > Threshold$).

The above simulation methodology has several limitations. It ignores burstiness of the request traffic and approximates it by a fixed average arrival rate. This will directly affect the number of consistency misses as seen by a demand cache. Burstiness of request traffic can also hurt prefetching at a proxy since there may not be any bandwidth available for prefetching at the peak points of demand traffic. (On the other hand prefetching can also help burstiness by smoothing out demand.) Ignoring temporal locality in the synthetically generated trace underestimates the hit rate seen by the demand cache.

The above methodology models a scenario where the universe of objects being accessed at the cache is fixed and their popularities known *a priori*. However, the Internet is a dynamic set of objects with new objects being created continuously. But, our model and the prefetching strategy extends to a dynamic universe of objects by assuming the existence of an oracle to perform the statistics gathering as described in Section 2.1. Such an oracle continuously maintains the information about changing set of objects, their popularities and lifetime distributions. The prefetching strategy obtains a snapshot of the Internet from the oracle and decides whether or not to prefetch an object solely on the basis of its current popularity and mean lifetime, independent of other objects and independent of its history. Thus, given good statistical data, it makes no difference if the collection of objects in the prefetch set change over time. As noted in Section 2.1, developing such a realistic statistics predictor module is a subject left as future work. Though such an oracle is unrealistic, we show in the next subsection that the prefetching strategy yields to an adaptive/learning implementation that shows attractive performance in practice.

4.3 Proxy trace simulation

The trace based simulator uses a 12 day trace logged by the Squid proxy cache. The trace consists of about 10 million records and accesses to 4.2 million unique objects. We simulate an LRU based demand cache and a prefetch cache running an adaptive version of the Threshold algorithm. Query URLs (with a "?" in them) are considered as both uncacheable and unprefetchable.

The sizes of the objects are used as obtained from the trace. However, lifetimes are generated synthetically, since the traces do not contain object update information. The object lifetime distribution was obtained from the data in [15]. This distribution shows a mean lifetime of about 1.8 months and a median of 12.8 days for HTML files, and a mean of 3.8 months and a median of 63.9 days for image files. Since our prefetch strategy is sensitive to object lifetimes, this distribution could directly and significantly affect our results. Hence we also perform a sensitivity analysis of the performance of the Threshold algorithm with respect to median object lifetime by shifting the probability distribution curve of the lifetimes by several orders of magnitude along the lifetime axis.

The adaptive version of the Threshold algorithm assumes no oracular information about the popularities of the objects. The simulator maintains a running count of the current popularities of the objects as they are accessed. At any access the simulator computes the prefetch quotient of the object as defined in 2 and checks if it crosses the threshold, in which case the access is considered to be a prefetch hit. The demand cache runs in parallel as before and independent of the prefetch cache. We do not assume any cooperation between the demand and the prefetch cache. We call this policy of maintaining running object popularities as *predictor 1*.

The assumption that an object would have *already* been prefetched depending on its current popularity, even if it is accessed for the first time, is unrealistic if caches rely on only local history information, since the universe of objects representing the Internet is dynamic. (It is possible in a scenario where the server aggressively pushes new objects with popularity predictions.) However, for local prefetching it is reasonable to expect an object to be seen at least once before it can be predicted for prefetching. Hence, we also simulate another adaptive version of the Threshold algorithm, called *predictor 2* wherein an object's current popularity is used to compute its prefetch quotient only if it has been seen at least once before. With this modification prefetching will save only consistency misses and not compulsory misses for prefetched objects. Predictor 1 is a simplistic predictor, but it simulates a small amount of global access frequency knowledge by increasing the count of object i *before* a reference rather than after a reference. Thus, new objects can appear on the system prefetch list (albeit with low initial access frequencies. The fundamental difference between predictors 1 and 2 is that the former counts a request just before it is seen and the latter, just after.

5 Results

5.1 Synthetic simulations

Figure 6(a) plots the hit rates obtained by our prefetching policy for various prefetch thresholds with increasing request arrival rate. Note that demand request arrival rate is a key parameter describing the scale of the content distribution node under consideration. As noted in Figure 4 an increase in demand arrival rate increases the set of objects that meet a given threshold. The steady state hit rates for demand cache serve as a baseline for comparing the hit rates obtained by our prefetch policy. The graphs clearly show that we get improved hit rates by the threshold policy. For example, at an arrival rate of 1 req/sec, for threshold 0.5, we get an overall hit rate of 55% as compared to a hit rate of 50% obtained by an infinite demand cache in steady state. For an arrival rate of 10 req/sec, for threshold 0.1, we get an overall hit rate of 75% as against a hit rate of 63% obtained by the demand cache. In summary, the graphs show that significant improvements in hit rates are possible with long-term threshold prefetching.

Note that significant improvements are achievable across a broad range of CDN scales. Although lower arrival rates reduce the collection of objects that meet the prefetch threshold criteria, lower arrival rates also reduce the steady state hit rates achieved by a demand cache.

Figure 6(b) plots the total number of objects (from our simulated universe of one billion objects) that qualify to be prefetched at various threshold values with increasing arrival rates. Figures 6(c) and 6(d) plot the amount of prefetch bandwidth and prefetch cache needed to maintain the prefetched objects.

As seen from the graphs, the choice of a thereshold value affects the hit rate that we obtain and the overhead that we incur. A high threshold implies a better chance of use of the prefetched object, but a decreased hit rate since we do not prefetch many objects. At the same time it implies a reduced bandwidth and prefetch cache size overhead, again, since we do not prefetch many objects. This conclusion is well supported by the graph in Figure 4.

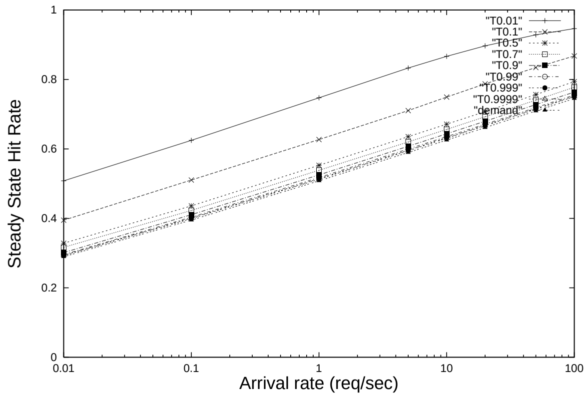
Figure 6(c) suggests that, for as the arrival rate increases, even low thresholds would incur a nominal bandwidth overhead as compared to a demand bandwidth. For example, for an arrival rate of 10 req/sec, a threshold of 0.1 would cost us a bandwidth of 1.6Mbps which is reasonable compared to the demand bandwidth of 800Kbps. From figure 6(a), this would give us a hit rate of 75%.

Figure 6(d) helps us in analysing a typical cache size budget needed at a real world proxy given its request rate. For example, a 10req/sec request rate, with a threshold of 0.01 would correspond to a 10TB prefetch cache size. Given today's disk costs, it would cost around $32000 to add a 10TB disk. From figure 6(a), a threshold of 0.01 at an arrival rate of 10 req/sec would provide us with an impressive 87% hit rate.

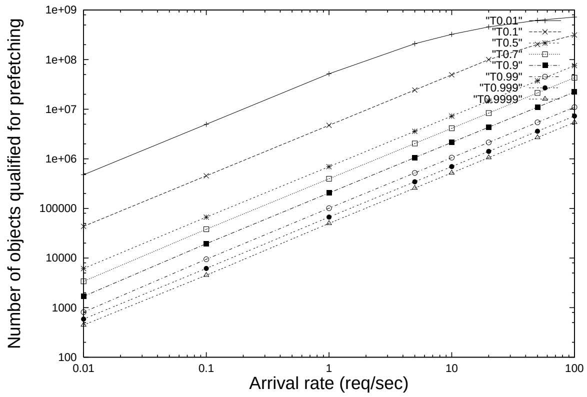
5.2 Trace based simulations

Figure 7 shows the hit rates and byte hit rates with increasing demand cache size for a demand based cache and a "demand+prefetch" based cache implementing our policy. The hit rate curves for the demand cache flatten after a certain size mainly because we used a limited trace size of 12 days.

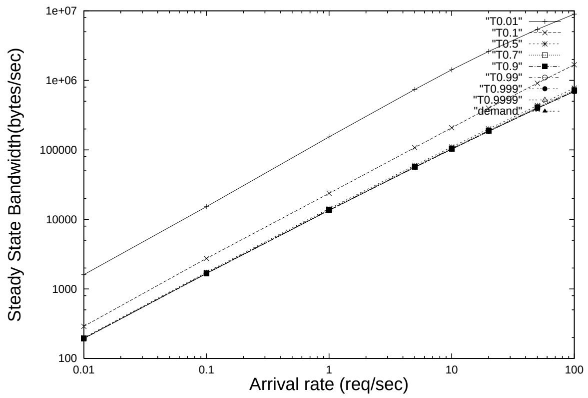
We use two simple statistical modules in our simulations namely predictor 1 and predictor 2. Predictor 1 assumes that an object accessed presently is considered to be prefetched if it qualifies for prefetch. Predictor 2 limits the prefetchability by assuming that an object is considered to be prefetchable only if atleast one access has already been seen for the object. Under our model, we can only avoid consistency misses and cannot reduce compulsory misses. A simple analyis of the trace showed that out of the 10.9 million requests that the proxy received, 1.08 million of them were consistency check messages (TCP_REFRESH_HIT, TCP_REFRESH_MISS, TCP_REFRESH_FAIL_HIT). Since consistency messages account for 10% of the requests, using a prefetch policy that avoids all consistency messages would give a direct 10% increase in hit rate which is a notable improvement for real world proxies.



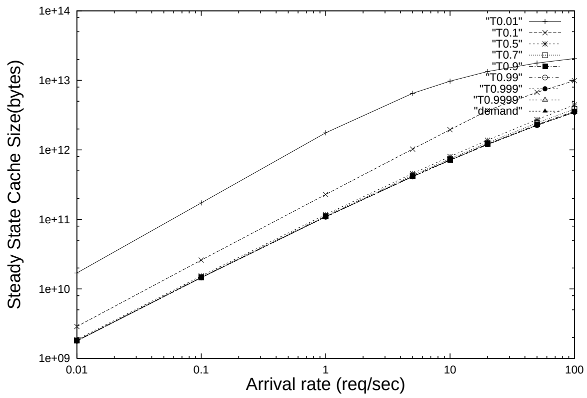
(a) Steady state hit rate



(b) Number of objects prefetched

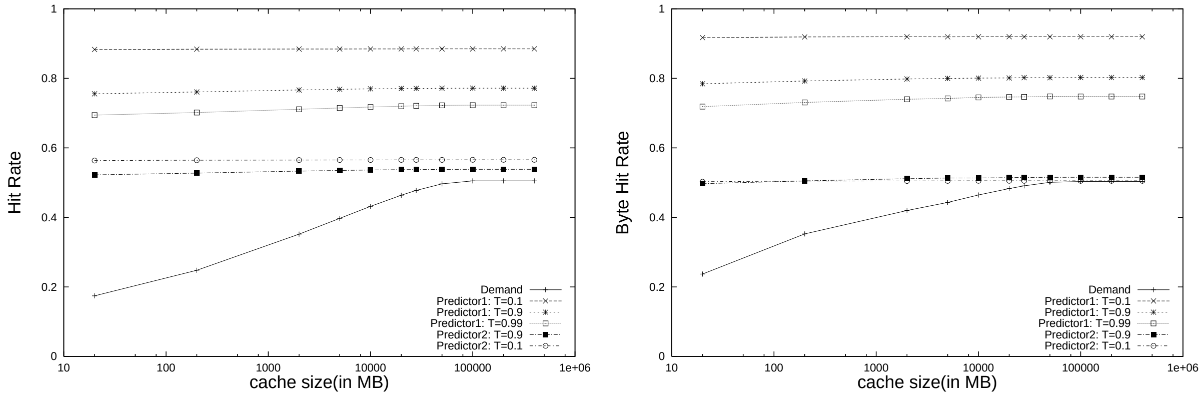


(c) Steady state bandwidth consumed



(d) Steady state cache size

Figure 6: Prefetching popular long lived objects. Effect of increasing request arrival rates for various thresholds



(a) Hit Rate (b) Byte Hit Rate

Figure 7: Effect of increasing demand cache size

Figure 8 plots hit rate achieved and Figure 9 plots bandwidth used and prefetch cache size required by our prefetching policy with varying threshold, when run over the 12 day squid proxy trace. The graphs clearly suggest that with a modest bandwidth cost, we can achieve significant hit rates.

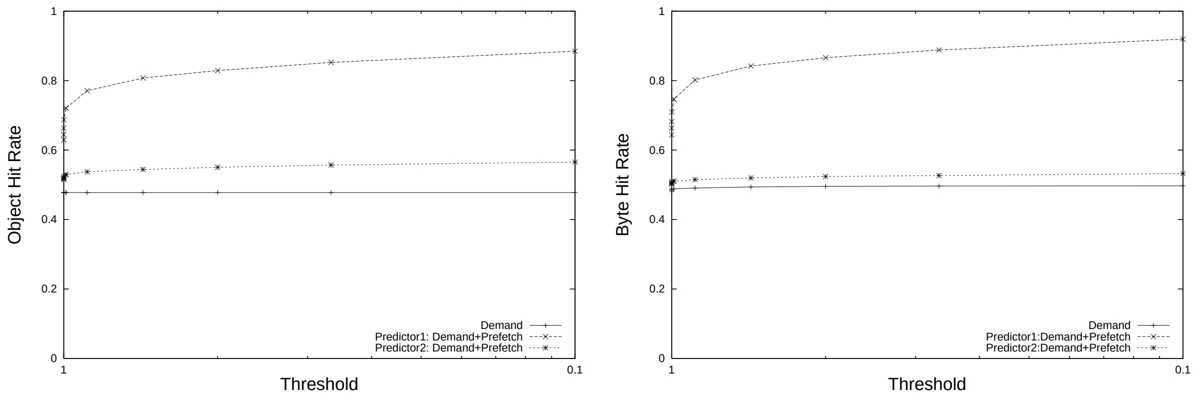
To test the sensitivity of our results to our assumption of lifetimes, we varied the mean life time of objects and studied its effect. Figure 10 shows our observations. The x-axis shows a shift factor s which denotes the horizontal displacement along the lifetime axis (on a logscale) of the probability density function corresponding to the CDF shown in 4.2. This varies the mean lifetime of the objects across several orders of magnitude. The graphs show that when life times are very low, we get less hit rate improvement but at the same time use less prefetch cache size and bandwidth. This is because our algorithm does not select short lived objects. To keep the chosen objects always fresh, our bandwidth consumption increases because the selected objects have low life times now. At higher lifetime values, we achieve higher hit rates at a less cost of bandwidth and prefetch cache size. The hit rate graphs at high threshold values flatten as the number of unique objects are limited, and almost all the objects would have already qualified for prefetch. This observation holds for a universe of a fixed set of objects; if we are already caching all the ojects, then increase in life times or arrival rates would not alter hit rates. But the bandwidth required to keep the objects refreshed reduces proportionally as the life times increase.

In summary, trace based simulation results show that our prefetch algorithm indeed provides significant hit rate improvements. One limitation of our trace based study is that we chose a limited trace. But given the results we obtained, increasing the trace length would only benefit our results rather than hurt them.

6 Related Work

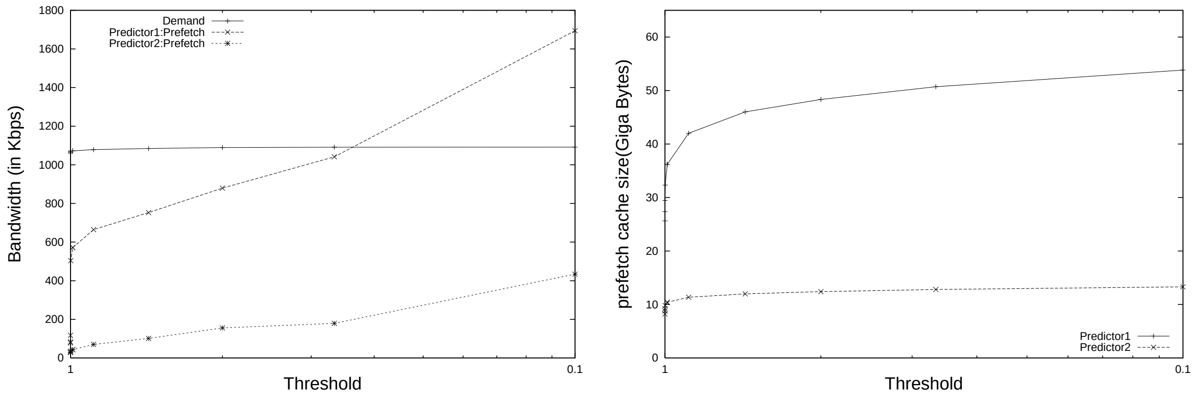
Latency reduction using passive caching is limited due to the presence of uncacheable data, creation of new objects and modifications to existing ones. Proposed techniques for improving the performance of passive caching seek to push or pull content to the edges at the cost of increases in bandwidth and storage consumption. These include predictive prefetching algorithms, server initiated pushing, and both manual an automatic content replication and distribution systems.

Short-term prefetching uses predictive algorithms at the server, proxy, or browser level to determine which objects to prefetch. Padmanabhan et al. [28] describes a server-hint architecture in which the server accumulates data from previous client accesses and provides hints to new clients based on it.Cohen et al. [10]



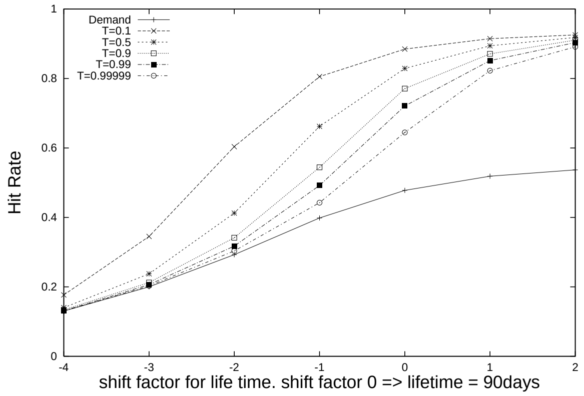
(a) Hit Rate (b) Byte Hit Rate

Figure 8: Prefetching popular long lived objects. Effect of varying threshold on hit rates

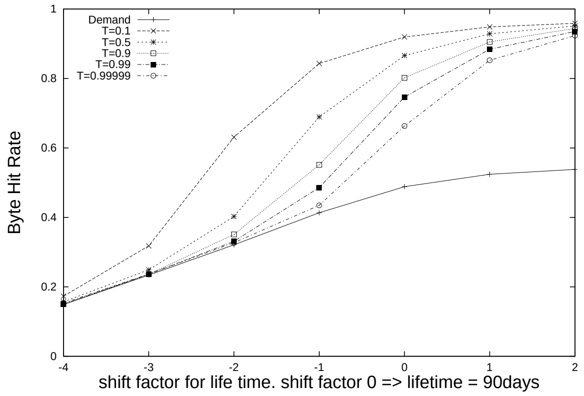


(a) Bandwidth usage (b) Prefetch cache size

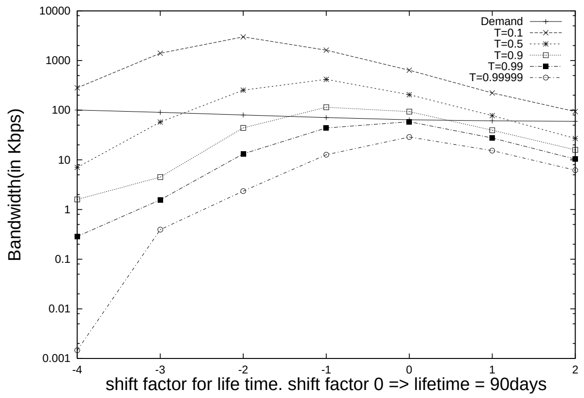
Figure 9: Prefetching popular long lived objects. Effect of varying threshold on prefetch bandwidth and cache size



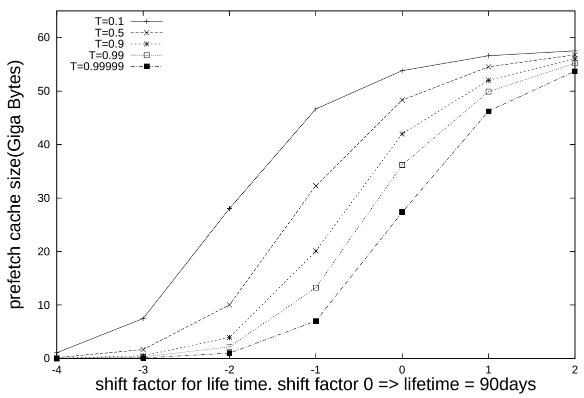
(a) Hit Rate



(b) Byte Hit Rate



(c) Bandwidth usage



(d) Prefetch cache size

Figure 10: Prefetching popular long lived objects. Effect of varying mean life time of objects

examines a variety of techniques for grouping resources that are likely to be accessed together, in order to generate hints for prefetching applications and cache validation.Fan et al. [19] describes a proxy-initiated prefetching technique which uses Prediction by Partial Matching, where the next request is predicted based on past accesses by the user. Embedded URLs provide good candidates for prefetching as demonstrated by Duchamp [16].

Gwertzman et al. [22] discusses the use of a server's global knowledge of usage patterns and network topology to distribute data to cooperating servers. Because push caching is server initiated it gives origin servers greater control over content dissemination. Servers can decide whether to retain pushed content based on object popularity.

Manual mirroring of web sites has been used to distribute server load. Content Distribution Networks (CDN) take this one step further by trying to dynamically replicate popular content. Commercial CDNs replicate web content provided by paying customers to servers near the edges of the network. For example Akamai [1] serves embedded content, like images, from its set of content distribution servers, while the html text is fetched from the origin server. A number of research efforts support multicast delivery for web content distribution. Li et al. [25] investigate multicast invalidation and delivery of popular, frequently updated objects to web cache proxies. Their protocol, MMO, groups objects into volumes, each of which maps to one IP multicast group. They show that, by forming volumes of the appropriate size, the benefit from reliable multicast outweighs the cost of delivering extraneous data as well as the overhead of multicast reliability. Zhang et al. [35] proposes an architecture for a multicast-based mesh of caches for data dissemination. Popular content propagates itself to more caches closer to clients, while less popular content is stored only at a few caches near the origin server.Byers et al. [7] describes a prototype system for reliable delivery of bulk data over the internet which uses IP multicast, and erasure codes.

Venkataramani [32] develops a provably near optimal algorithm for placing objects in a set of cooperating caches that are constrained by update bandwidth. Given the network distances between the cooperating caches and the predictions of access rates from each cache to a set of objects, the placement algorithms determine where to place each object in order to minimize the average access cost. The algorithm could be used to extend the longterm prefetching to cooperating collection of caches.

7 Conclusion

Advances in storage and networks have dramatically reduced the cost of object replication. This paper examines the costs and benefits of replicating collections of popular long-lived objects to improve web performance. We first develop a steady-state model for bandwidth-constrained web caches. We then evaluate the performance of a long-term prefetching algorithm whose aggressiveness can be tuned by varying a prefetch threshold. Overall, we find that this approach can significantly improve steady state hit rates at modest bandwidth costs.

References

[1] `http://www.akamai.com`.

[2] `http://www.digitalisland.com`.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing Reference Locality in the WWW. In *Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[4] P. Barford and M. Crovella. Generating representative workloads for network and server performance evaluation, 1998.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom*, 1999.

[6] B. E. Brewington and G. Cybenko. How dynamic is the web? *WWW9 / Computer Networks*, 33(1-6):257–276, 2000.

[7] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[8] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. Technical Report CS-TR-1998-1363, 1998.

[9] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource management for scalable disconnected access to web services. In *WWW10*, May 2001.

[10] E. Cohen, B. Krishnamurthy, and J. Rexford. Efficient Algorithms for Predicting Requests to Web Servers. In *INFOCOMM 99*.

[11] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. 1998.

[12] M. Crovella and P. Barford. The network effects of prefetching. In *Proceedings of IEEE Infocom*, 1998.

[13] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-based Traces. Technical Report TR-95-010, Boston University, CS Dept, Boston, MA 02215, April 1995.

[14] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. pages 257–266, 1993.

[15] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. C. Mogul. Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[16] D. Duchamp. Prefetching Hyperlinks. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[17] B. Duska, D. Marwood, and M. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *USITS97*, Dec 1997.

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the 1998 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 254–265, August 1998.

[19] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance, 1999.

[20] S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.

[21] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *USITS97*, Dec 1997.

[22] J. Gwertzman and M. Seltzer. An analysis of geographical push-caching, 1997.

[23] G Karakostas and D. Serpanos. Practical LFU implementation for Web Caching . Technical Report TR-622-00, Department of Computer Science, Princeton University, 2000.

[24] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. 1998.

[25] Dan Li and David R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99)*, OCT 1999.

[26] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of ICDCS 1997*, May 1997.

[27] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. Technical report, Digital/Compaq Western Research Lab, Dec 1997.

[28] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of the SIGCOMM '96 conference*, 1996.

[29] T. Palpanas. Web prefetching using partial match prediction, 1998.

[30] The ICAP Protocol Group. Icap the internet content adaptation protocol. Technical Report draft-opes-icap-00.txt, IETF, December 2000.

[31] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active names: Flexible location and transport of wide-area resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[32] Arun Venkataramani and Mike Dahlin. Bandwidth constrained placement in a WAN, Jan 2001.

[33] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Procedings of the ACM SIGCOMM'96 conference*, 1996.

[34] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of ICDCS 1998*, May 1998.

[35] L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the NLANR Web Cache Workshop*, June 1997. http://ircache.nlanr.net/Cache/Workshop97/.

A Steady state bandwidth

Consider a time period *T* over which accesses appear at a rate of *a* arrivals per unit time. The number of accesses in time T has a Poisson distribution with mean aT. Let the relative access probability of an object *i* is p_i. From the discussion in Section 4.1, we can infer that the number of accesses to an object *i* in time period *T* also follows a Poisson distribution with mean ap_iT.

The estimated bandwidth is

$$BW_{ss_d} \quad = \quad \frac{1}{T}\sum_i \sum_{k=0}^{\infty} P(k \text{ accesses to } i \text{ in } T \text{ time}) \sum_{j=0}^{k} P(j \text{ hits in those } k \text{ accesses})(k-j)s_i \quad (12)$$

$$
\begin{aligned}
P(k \text{ accesses to } i \text{ in } T \text{ time}) &= e^{-\lambda}\frac{\lambda^k}{k!}, \text{ where } \lambda = ap_iT && (13)\\
P(j \text{ hits in } k \text{ accesses to } i) &= C_j^k(p_i')^j(1-p_i')^{k-j}, \text{ where} && (14)\\
& \quad p_i' = \text{probability of hit on an access to } i
\end{aligned}
$$

The probability of hit on an access to object i is same the freshness factor calculated in Section 4.1. From above equations,

$$
\begin{aligned}
BW_{ss_d} &= \frac{1}{T}\sum_i \sum_{k=0}^{\infty} e^{-\lambda}\frac{\lambda^k}{k!}\sum_{j=0}^{k} C_j^k(p_i')^j(1-p_i')^{k-j}(k-j)s_i \\
&= \frac{1}{T}\sum_i \sum_{k=0}^{\infty} e^{-\lambda}\frac{\lambda^k}{k!}(k(1-p_i')s_i) \\
&= \frac{1}{T}\sum_i (1-p_i')(s_i)\sum_{k=0}^{\infty} ke^{-\lambda}\frac{\lambda^k}{k!} \\
&= \sum_i s_i(1-p_i')ap_i. && (15)
\end{aligned}
$$