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

Arun Venkataramani Praveen Yalagandula Ravindranath Kokku Sadia Sharif Mike Dahlin

Technical Report TR-01-13
Department of Computer Sciences
University of Texas at Austin
TAY 2.124, Austin, TX 78712-1188, USA

{arun,ypraveen,rkoku,dahlin}@cs, sharif@ece}.utexas.edu

June 30, 2001

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

In spite of advances in web proxy caching techniques in the past few years, proxy cache hit rates have not improved much. Even with unlimited cache space, passive caching suffers from uncacheable data, consistency misses for cached data and compulsory misses for new data. Prefetching attempts to overcome these limitations of passive caching by proactively fetching content without waiting for client requests. Traditional short-term prefetching at clients uses recent access history to predict and prefetch objects likely to be referenced in the near future and can considerably improve hit rates [8, 9, 18, 21, 32].

In this paper we examine a technique more appropriate for large proxies and content distribution networks (CDNs), namely *long-term prefetching*. Rather than basing prefetching decisions on the recent history of individual clients, 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.

As hardware costs fall, more aggressive prefetching becomes attractive making it possible to store an enormous collection of data at a large content distribution site. For example, in March 2001 an 80GB disk drive cost about \$250 [1]. 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 objects in the collection change, the system must fetch their new versions. It must also fetch newly created objects that meet its selection criteria. Due to the Web’s Zipf-like access patterns, a large number of objects must be actively prefetched to improve hit rates significantly [8]. Maintaining such a collection appears to be challenging. In particular, bandwidth expenditure will be the primary constraint in a long term prefetching strategy. For example, in May 2001 a 1.5 Mbps T1 connection cost about \$1000 per month [2].

In this paper, we present a model for understanding steady-state cache behavior in a bandwidth-constrained prefetching environment. Our hypothesis is that by prefetching objects that are both long-lived and popular, we can significantly improve hit rates for moderate bandwidth costs. 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 patterns. Although we leave development of such predictors as future work, we provide initial evidence that even simple predictors may work well.

The rest of the paper is organized as follows. Section 2 provides some background information about prefetching and our prefetching model. Section 3 presents the algorithms that we consider for long-term prefetching. Section 4 discusses the methodology we use to evaluate long-term prefetching. Section 5 discusses the results of our simulations and provides insights about how long-term prefetching works. Section 6 discusses related work. Section 7 summarizes our conclusions.

2 Background

This section describes five key parameters of web workloads that determine the effectiveness of caching and prefetching: prefetching models, object popularities, object sizes, object update patterns and lifetimes, and the availability of spare bandwidth for prefetching.

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 [18, 21, 32], most of which are based on variations of a Prediction-by-Partial-Matching (PPM) strategy [16].

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 [25].
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 [20, 38] and commercial systems [3, 4] address this issue.

2.2 Popularity distributions

A key parameter for understanding long-term prefetching is the distribution of requests across objects. Several studies [6, 15, 22, 36] 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. [15] 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 [8].

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 [14] 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. [8] suggests that there is little or no correlation between object sizes

and their popularity. However, an earlier study by Crovella et al. [15] 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. [17] 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. [8] 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 [9]. 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 [17]. 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. [9], 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 $\frac{1}{\text{threshold}}$ 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 [18], and Chandra et al. [12] 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 [14]. Furthermore, techniques such as multicast, digital fountain [10], delta-encoding [30], 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 are removed from the set of fresh objects in the cache when servers update cached objects, invalidating the cached copy.¹

¹For simplicity, we describe a system in which servers invalidate clients’ cached objects when they are updated [13, 26, 29, 37]. 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.

Figure 1: Equilibrium in bandwidth-constrained cache.

Figure 2: Equilibrium in bandwidth-constrained cache.

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 [19, 23], 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 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 .

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

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

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

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.

$lifetime_i * requestRate$ 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 [18], 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 [17], 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.

3.3 Constant-factor Optimality

In this section we show that the hit rate obtained by prefetching the set of objects computed by the Threshold algorithm is within a constant factor (2X) of the optimal achievable hit rate. We show this result by expressing the object selection problem in the context of prefetching as equivalent to the 0-1 Knapsack problem. We then show that the Threshold algorithm is equivalent to the *value-density* heuristic for the 0-1 Knapsack problem, which is known to be within a 2X factor of optimal solution, but for a small technicality.

We first introduce the Knapsack problem which is as follows: Given a set of items $S = \{1, \dots, n\}$, where item i has size s_i and value v_i and a knapsack with capacity C , find the subset $S' \subseteq S$ that maximizes the value of $\sum_{i \in S'} v_i$ given that $\sum_{i \in S'} s_i \leq C$.

The value-density heuristic for the Knapsack problem is as follows: Let $d_1, d_2, \dots, d_n$ denote the items in the order of non-increasing value-density, *i.e.* $\frac{v_i}{s_i}$. Return the set of objects as $S' = \{d_1, \dots, d_k\}$, where k is the largest integer such that $\sum_{1 \leq i \leq k} d_i \leq C$. The following lemma expresses the optimality of the value-density heuristic to within a constant factor provided no single object has greater value than the combined value of the set of objects computed by the value-density heuristic.

Lemma 1: Let P_{opt} be the value of the optimal selection of objects and P denote the set computed by the value-density heuristic. Then, $P_{opt} \leq 2(\max(P, \max_{i \in [1, n]}(v_i)))$

Theorem 1: The byte hit rate achieved by the Threshold algorithm is within a 2X factor of the optimal.

Proof Outline: We model the object selection problem in the context of prefetching as an instance of the 0 – 1 Knapsack problem. Given a set of n objects, with a popularity distribution P_i , a lifetime distribution $l f_i$, $1 \leq i \leq n$, a size distribution $s z_i$, a total bandwidth constraint B and an infinite demand cache, the prefetching problem is to select a suitable subset of objects S to keep updated locally in order to maximize hit rate. The contribution of object i to the object hit rate is P_i . However, the contribution to the hit rate because of prefetching object i , given the existence of an infinite demand cache, is the probability that object i is accessed at least once during its lifetime, divided by l_i which is given by $\frac{P_{goodFetch}(i)}{l_i}$. The contribution to the byte hit rate is therefore proportional to $\frac{P_{goodFetch}(i) * s z_i}{l_i}$. We understand this expression as the *value* of object i . The "size" or the bandwidth cost of keeping the local copy of object i updated is $\frac{s z_i}{l f_i}$. Therefore, the *value-density* of object i is given by $P_{goodFetch}(i)$, which is exactly the criterion that the threshold algorithm uses to select objects. Thus the threshold algorithm is equivalent to the value-density heuristic for the 0-1 Knapsack problem and hence achieves a byte hit rate that is within a 2X factor of the optimal. $\square$

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 Analytic model

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 [9]. In our model, we assume that the average lifetime of an object i to be l_i . Each object i is assumed to have a fixed size s_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 that the previous access to object i was t time units before the present time and $P_{B_i}(t)$ as the probability that no updates were done to object i since its last access t time units in the past. Now, the probability of a hit on a request to a demand cache is

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

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 $P_{(a,t)}(k) = e^{-at} \cdot \frac{(at)^k}{k!}$. 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

$$\begin{aligned} P(0 \text{ accesses to object } i \text{ in } t \text{ time}) &= \sum_{k=0}^{\infty} P(k \text{ requests in } t) P(\text{none of these } k \text{ requests is for } i) \\ &= \sum_{k=0}^{\infty} \left(e^{-at} \frac{(at)^k}{k!} \right) (1 - p_i)^k, \\ &= e^{-at} \sum_{k=0}^{\infty} \frac{(at(1 - p_i))^k}{k!} \\ &= e^{-at} e^{at(1 - p_i)} \\ &= e^{-(ap_i)t} \end{aligned}$$

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,

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

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

$$P_{B_i}(t) = e^{-t/l_i} \quad (5)$$

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

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

The fraction $\frac{ap_i l_i}{ap_i l_i + 1}$ 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

$$\begin{aligned} P_{hit_p}(T) &= \sum_i p_i h_i, \text{ where} \\ h_i &= \begin{cases} 1 & \text{if } P_{goodFetch}(i) > T \\ \frac{ap_i l_i}{ap_i l_i + 1} & \text{otherwise} \end{cases} \end{aligned} \quad (7)$$

Steady state cache sizes

The steady state cache size is defined as the total size of fresh objects in a demand cache of infinite size at steady state when the cache is in equilibrium, *i.e.* the rate at new incoming objects are brought into the cache equals the rate at which objects get invalidated. 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. However, the steady state cache size as defined above, is likely to be lesser if invalidation of objects were to be taken into consideration.

In steady state, the probability of an object i being fresh in the cache is given by $\int_0^\infty P_{A_i}(t) P_{B_i}(t) dt$, where $P_{A_i}(t)$ and P_{B_i} are as defined earlier. Hence the probability of an object i being fresh in the cache at some random time instance is just the freshness factor of i , $ff(i)$ as derived before. 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

$$CSize_{ss_d} = \sum_i s_i \frac{ap_i l}{ap_i l + 1} \quad (8)$$

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

$$CSize_{ss_p} = \sum_i s_i * f_i, \text{ where} \quad (9)$$

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

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 consumed by just demand fetches is

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

For Threshold algorithm based prefetch strategy, the steady state bandwidth consumed by both prefetch and demand fetches is

$$BW_{ss_p} = \sum_i s_i * h_i, \text{ where} \quad (11)$$

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

The proofs for these derivations are presented in Appendix A.

4.2 Analytic model 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 [7]. We assume that there is no correlation between object sizes and their popularities. The distribution of object lifetimes was taken from [17] again assuming no correlation with popularity or size. The analysis in [8] support these assumptions of the non existence of any observable correlation between the popularity, lifetime and size of objects. Figure 5 shows the cumulative distribution functions for object sizes and lifetimes.

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 [11], active names [34], and ICAP [33] to support execution of server program at caches and Content Distribution Nodes. Also the efforts to improve cache consistency semantics [37, 13, 26, 29] will enable caching of much currently

Figure 5: Cumulative distribution

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 [5] at NCSA, Urbana Champaign between Feb 27 and Mar 10 2001. 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 implementing the Threshold algorithm using two simple predictors for assessing object popularities. Query URLs (with a "?" in them) are considered as both uncacheable and un prefetchable.

The sizes of the objects are used as logged in the trace. However, as in the analytic model, lifetimes are generated synthetically from the distribution given in [17], because the traces do not contain object update information. 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.

We use a simple statistical model to estimate the popularities of the objects. The simulator maintains a running count of the number of accesses to each object and computes object popularity by dividing this number by the total number of accesses seen. At any access the simulator computes the $P_{goodFetch}$ of the object as defined in Equation 2 and checks if it exceeds the threshold, in which case the access is considered to be a prefetch hit. We call this object popularity predictor as *Predictor1*. *Predictor1* is aggressive, as it uses the current access to update the corresponding object’s popularity before computing its $P_{goodFetch}$. This predictor can potentially save compulsory misses to an object since it knows an object’s popularity even before the first access to it. Such a predictor is feasible in a scenario where popularity information is aggressively pushed out by the server, or is widely shared across several cooperating caches. We also simulate a more conservative predictor known as *Predictor-2* that can not prevent compulsory misses. An object has to be seen at least once before its popularity can be used to consider it for prefetching by this predictor. For comparison purposes, we also simulate an oracle *EverFresh* that gets rid of *all* consistency misses, *i.e.*, an access is a hit if the corresponding object has been seen before. The hit rate so obtained is the maximum attainable by a cache based on local prefetching alone. Even the oracle cannot prevent compulsory misses or achieve hits to uncacheable objects. For all of the experiments, the cache was allowed to warm up for 8 days. The long period of warm up will prevent inflated estimates of popularities corresponding to accesses in the beginning of the trace.

As part of future work we intend to measure the performance of more aggressive predictors that gather popularity information from several cooperating caches. In such a scenario, it might be possible to considerably reduce compulsory as well as consistency misses.

5 Results

5.1 Analytic Evaluation

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 and 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 show that the threshold policy improves hit rates. For example, at an arrival rate of 1 req/sec, for threshold 0.5, we get an overall hit rate of 55% 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% compared to a hit rate of 62% obtained by the demand cache.

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.

(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

As seen from the graphs, the choice of a threshold value affects the observed hit rate and the overhead. A high threshold implies a better chance of use of the prefetched object but a decreased hit rate since fewer objects will qualify for prefetching. At the same time it implies a reduced bandwidth and prefetch cache size overhead.

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

Figure 6(d) helps us in analyzing 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 0.1 would correspond to a 2TB prefetch cache size. Given today's disk costs, it would cost around \$6400 to add a 2TB disk. From Figure 6(a), a threshold of 0.1 at an arrival rate of 10 req/sec would provide us with a 75% hit rate.

Given falling hardware costs, and the potential to use system-level techniques to decrease cost of handling prefetch requests, even more aggressive prefetching such as $T=0.01$ might be contemplated in the future. From the graphs, at the cost of 10TB disk and 10Mbps bandwidth, we could achieve 87% hit rate which effectively means a decrease in missrate from 45% to 13%.

5.2 Trace based simulations

A simple analysis of the trace showed that out of the 10.9 million requests that the proxy received, approximately 10% were consistency check messages. This implies that even an ideal prefetching strategy that prevents all of the consistency misses cannot give an improvement of more than 10% in hit rate over that of an infinite demand cache, unless, its statistics tracing spans multiple caches, or it allows servers to supply popularity estimates when objects are created.

In our experiments we assume a demand cache of size 28GB. We allow the cache to be warmed for 8 days and then gather performance measurements over the remaining 4 days.

Figures 7(a), (b) and (c) show the change in hit rate, overall bandwidth consumption (demand+prefetch) and overall cache size (demand+prefetch) with varying thresholds. In figure 7(a), the *EverFresh* algorithm gives a hit rate of about 61%. At a threshold of 0.1 *Predictor2* almost matches this "optimal" achievable hit rate. *Predictor1*, as expected, gives much higher hit rates since it can avoid compulsory misses as well. Figure 7(c) shows that for a threshold of 0.1, the bandwidth blow-up as compared to the demand bandwidth is less than 2X. *Predictor1* shows lesser bandwidth costs because of the *smoothing* out of prefetched bytes over a larger duration (a typical prefetched object's lifetime) as opposed to the post-warming period of 4 days for *Demand* and *Predictor2*. Figure 7(b) shows that the increase in cache size due to prefetching is nominal as well. Thus, we conclude that we can obtain significant improvements in hit rate at modest bandwidth and cache space costs.

Figures 7 (d) and (e) show the attainable hit rate with respect to the bandwidth and cache space costs respectively. These graphs have also been generated from the data obtained from experiments performed by varying the threshold and have been included for ease of reference.

To test the sensitivity of our results to our assumptions about lifetimes, we vary the mean life time of objects and study its effect. Figures 7(f)- 7(h) show our results. 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 taken from [17]. This graph varies the mean lifetime of the objects across several orders of magnitude, with each unit representing a change in the average lifetimes by a factor of 10. The graphs show that when life times are small, we get less hit rate improvement but at the same time use less

(a) Object Hit rate

(b) Overall cache size

(c) Total Bandwidth

(d) Hit rate vs. Total Bandwidth

(e) Hit rate vs. Total cache size

(f) Object Hit rate sensitivity

(g) Bandwidth sensitivity

(h) Total cache size sensitivity

Figure 7: (a)-(e) Effects of varying the Threshold value on hit rate and bandwidth; (f)-(h) Sensitivity analysis of results

prefetch cache size and bandwidth. This result is expected because our algorithm does not select short lived objects. At bigger lifetime values, we achieve higher hit rates at a reduced 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 objects, 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 lifetimes 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 medium-sized trace. But given the results we obtained, increasing the trace length would only benefit our results rather than hurt them.

6 Related Work

The idea of prefetching in the web has been widely studied by many researchers recently. Most of the research has concentrated on short-term prefetching based on recent access patterns of clients. Duchamp [18] provides a survey of various research contributions relevant to short-term prefetching in the web: software systems, algorithms, simulations and prototypes [31], and papers that establish bounds [27]. Duchamp [18] proposes to prefetch hyperlinks of web pages based on aggregate access patterns of clients as observed by the server. This approach gives priority to popular URLs and hence is similar to our popularity algorithm. Our (threshold based) approach is different from the above approaches in that we consider long-term prefetching suitable to CDNs [3] and busy proxies by prefetching objects that are both popular and long lived so that we get long-term benefits.

Gwertzman et al. [24] discuss push caching which makes use of a server's global knowledge of usage patterns and network topology to distribute data to cooperating servers. A number of research efforts support multicast delivery for web content distribution. Li et al. [28] investigate multicast invalidation and delivery of popular, frequently updated objects to web cache proxies. These techniques are complementary to our work and can be used to realize the distribution of updates that we assume in our CDN model.

Implementations of cooperating caches are becoming increasingly commonplace. Sharing of popularity information across caches enables not only better estimates of popularity but can also prevent some compulsory misses. Venkataramani et al. [35] develop 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 long-term prefetching to a collection of cooperating caches.

7 Conclusion

In this paper, we focussed on a technique called long-term prefetching, which is beneficial for web proxies and content distribution networks. In this prefetching model, we have explored an algorithm for object selection based on the popularity and lifetime of objects. We evaluated the performance of a long-term prefetching algorithm whose aggressiveness can be tuned by varying a prefetch threshold. For example, our analytical evaluation showed that, for a cache that receives 10 demand requests per second, our prefetching algorithm can improve hit rates by up to 13 percent, while increasing the bandwidth requirements by just over a factor of 2. Our trace-based results show that using simple history based predictors one can eliminate

a significant fraction of consistency misses. More sophisticated popularity predictors that share information between cooperating caches can result in up to a 30% improvement in hit rate at a bandwidth blow-up of less than a factor of two.

The work in this study brings up three key issues which make interesting topics of future research.

1. Exploring the performance of a two level prefetcher composed of a long-term prefetching module that is useful at the entry level while accessing web pages, and a short-term prefetcher that can prefetch hyperlinks there onwards.
2. Exploring mechanisms to estimate the long term effects of prefetching on the underlying network, and the increased opportunities it offers for traffic shaping. resulting from the smoothening of demand network traffic.
3. Extending the threshold algorithm to a set of cooperating caches that can share aggregated popularity information.

An in depth understanding of these issues will form the basis for the design of efficient content distribution systems in the future.

References

- [1] <http://cs.utexas.edu/users/dahlin/techreports/data/diskprices/data>.
- [2] <http://www.netdoor.com/info/tier3pricing.html>.
- [3] <http://www.akamai.com>.
- [4] <http://www.digitalisland.com>.
- [5] <http://www.squid-cache.org>.
- [6] 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.
- [7] P. Barford and M. Crovella. Generating representative workloads for network and server performance evaluation, 1998.
- [8] 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.
- [9] B. E. Brewington and G. Cybenko. How dynamic is the web? *WWW9 / Computer Networks*, 33(1-6):257–276, 2000.
- [10] 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.
- [11] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. Technical Report CS-TR-1998-1363, 1998.

- [12] 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.
- [13] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. 1998.
- [14] M. Crovella and P. Barford. The network effects of prefetching. In *Proceedings of IEEE Infocom*, 1998.
- [15] 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.
- [16] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. pages 257–266, 1993.
- [17] F. Douglass, 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.
- [18] D. Duchamp. Prefetching Hyperlinks. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, October 1999.
- [19] B. Duska, D. Marwood, and M. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *USITS97*, Dec 1997.
- [20] 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.
- [21] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance, 1999.
- [22] S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.
- [23] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *USITS97*, Dec 1997.
- [24] J. Gwertzman and M. Seltzer. An analysis of geographical push-caching, 1997.
- [25] G Karakostas and D. Serpanos. Practical LFU implementation for Web Caching . Technical Report TR-622-00, Department of Computer Science, Princeton University, 2000.
- [26] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. 1998.
- [27] T. M. Kroege, D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [28] Dan Li and David R. Cheriton. Scalable web caching of frequently updated objects using reliable multi-cast. In *Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99)*, OCT 1999.
- [29] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of ICDCS 1997*, May 1997.

- [30] 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.
- [31] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of the SIGCOMM '96 conference*, 1996.
- [32] T. Palpanas. Web prefetching using partial match prediction, 1998.
- [33] The ICAP Protocol Group. Icap the internet content adaptation protocol. Technical Report draft-opes-icap-00.txt, IETF, December 2000.
- [34] 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.
- [35] Arun Venkataramani, Mike Dahlin, and Phoebe Weidmann. Bandwidth constrained placement in a WAN. In *ACM Principles of Distributed Computing. To appear in Aug 2001*.
- [36] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of the ACM SIGCOMM'96 conference*, 1996.
- [37] 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.
- [38] 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

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

$$P(k \text{ accesses to } i \text{ in } T \text{ time}) = e^{-\lambda} \frac{\lambda^k}{k!}, \text{ where } \lambda = ap_iT \quad (13)$$

$$P(j \text{ hits in } k \text{ accesses to } i) = C_j^k (p'_i)^j (1 - p'_i)^{k-j}, \text{ where} \quad (14)$$

$p'_i = \text{probability of hit on an access to } i$

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

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