Objective-Optimal Algorithms for Long-Term Web Prefetching

Bin Wu and Ajay D. Kshemkalyani, *Senior Member*, *IEEE*

Abstract—Web prefetching is based on Web caching and attempts to reduce user-perceived latency. Unlike on-demand caching, Web prefetching fetches objects and stores them in advance, hoping that the prefetched objects are likely to be accessed in the near future and such accesses would be satisfied from the caches rather than by retrieving the objects from the Web server. This paper reviews the popular prefetching algorithms based on *Popularity*, *Good Fetch*, *APL characteristic*, and *Lifetime*, and then makes the following contributions: 1) The paper proposes a family of linear-time prefetching algorithms, *Objective-Greedy* prefetching, wherein each algorithm greedily prefetches those Web objects that most significantly improve the performance as per the targeted metric. 2) The *Hit rate-Greedy* and *Bandwidth-Greedy* algorithms are shown to be optimal for their respective objective metrics. A linear-time optimal prefetching algorithm that maximizes the *H/B* metric as the performance measure is proposed. 3) The paper shows the results of a performance analysis via simulations, comparing the proposed algorithms with the existing algorithms in terms of the respective objectives—the hit rate, bandwidth, and the *H/B* metrics. The proposed prefetching algorithms are seen to provide better objective-based performance than any existing algorithms. Further, *H/B-Greedy* performs almost as well as *H/B-Optimal*.

Index Terms—Web server, World Wide Web, Web caching, Web prefetching, content distribution, Web object, hit rate, bandwidth, optimal object selection, randomized algorithm.

✦

1 INTRODUCTION

$\mathbf{W}$EB caches are widely used in the current Internet environment to reduce the user-perceived latency of object requests. One example is a proxy server that intercepts the requests from the clients and serves the clients with the requested objects if it has the objects stored in it; if the proxy server does not have the requested objects, it then fetches those objects from the Web server and caches them and serves the clients from its cache. Another example is local caching that is implemented in Web browsers. In the simplest cases, these caching techniques may store the most recently accessed objects in the cache and generally use an LRU replacement algorithm that does not take into account the object size and object download cost. Cao and Irani [5] developed a Greedy Dual-Size algorithm that is a general-ization of the LRU replacement algorithm to deal with variable object sizes and download times and it was shown to achieve better performance than most other Web cache evicting algorithms in terms of hit rate, latency reduction, and network traffic reduction. However, all these techniques use on-demand caching or short-term caching and the objects to be cached are determined by the recent request patterns of the clients. Long-term prefetching, on the other hand, is a mechanism that allows clients to subscribe to Web objects to increase the cache hit rates and thus reduce user latency. The Web servers or Web object hosts proactively "push" fresh copies of the subscribed objects into Web caches or proxy servers whenever such objects are

updated. This makes the user hit rates for these objects always 1. The selection of the prefetched objects is based on the long-term (statistical) characteristics of the Web objects, such as their (average) access frequencies, update intervals, and sizes, rather than the short-term (recent) access patterns at individual caches [20]. The long-term characteristics are obtained and maintained by collaboration among content distribution servers. The statistics may be collected and published within a specific domain, such as a news Website and its subscribed customers. A prefetching mechanism may be applied in this domain to increase the performance. Intuitively, to increase the hit rate, we want to prefetch those objects that are accessed most frequently; to minimize the bandwidth consumption, we want to choose those objects with longer update intervals. We assume unlimited cache sizes for both on-demand and prefetching cases in this paper.

This paper first reviews the popular prefetching algo-rithms based on *Popularity* [17], *Good Fetch* [20], *APL characteristic* [13], and *Lifetime* [13]. Their performance can be measured using the different criteria discussed. The paper then makes the following contributions:

1. The paper proposes a family of prefetching algo-rithms, *Objective-Greedy* prefetching, that are direc-ted to improve the performance in terms of the various objectives that each algorithm is aimed at —hit rate, bandwidth, or *H/B* metric. Each of the *Objective-Greedy* prefetching algorithms, *Hit rate-Greedy*, *Bandwidth-Greedy*, and *H/B-Greedy* has linear-time complexity $O(n)$ and is easy to implement. The *H/B-Greedy* prefetching aims to improve the *H/B* metric, which combines the effect of increasing hit rate (H) and limiting the extra bandwidth (B) consumed by prefetching. This criterion was first

● *The authors are with the Department of Computer Science (MC 152), 851 South Morgan Street, University of Illinois at Chicago, Chicago, IL 60607-7053. E-mail: {bwu, ajayk}@cs.uic.edu.*

proposed by Jiang et al. [13] as a performance metric in terms of both hit rate and bandwidth usage. Jiang et al. [13] compared different algorithms, i.e., *Prefetch by Popularity* [17], *Good Fetch* [20], *APL characteristic* [13], and *Lifetime* [13], using this criterion, but did not give any algorithm to target this criterion. We hypothesize that our *H/B-Greedy* prefetching achieves close-to-optimal *H/B*-performance and this is justified by our simulation results. The *Hit rate-Greedy* and *Bandwidth-Greedy*, as presented in Sections 4.3 and 4.4, are actually optimal in terms of *Hit rate* and *Bandwidth*, respectively, as the performance metric.

2. By studying and solving the *maximum weighted average problem with preselected items*, we propose an *H/B-Optimal* prefetching algorithm that maximizes the *H/B* metric. This randomized algorithm also has a time complexity of $O(n)$, but with a larger constant factor than that of *H/B-Greedy*. We use *H/B-Optimal* prefetching to obtain an upper bound for the *H/B* metric and this upper bound can be used to evaluate the performance of other prefetching algorithms in terms of this objective metric.

3. The paper shows the results of a simulation analysis comparing the performance of all the above algorithms in terms of the hit rate, bandwidth, and *H/B* metrics.

- Each of the proposed prefetching algorithms is seen to provide better performance than any existing algorithms based on the respective prefetching objective. In particular, the best trade-off between increasing hit rate and reducing extra bandwidth usage is obtained when the *H/B* metric is used.
- Our simulation results show that the proposed *H/B-Greedy* algorithm offers reasonable improvement of the *H/B* metric over the best known algorithm and it is a good approximation to the optimal solution. The *H/B-Greedy* algorithm has two significant implementation-related advantages over the *H/B-Optimal*. 1) Even though they both have time complexity of $O(n)$, the constant factor for *H/B-Greedy* is much smaller than that of *H/B-Optimal*. 2) In dynamic situations where object characteristics change with time, *H/B-Greedy* adapts to the changes more conveniently than *H/B-Optimal*.

The performance comparison based on the simulations is summarized in Table 4 in Section 6.

Section 2 provides an overview of the Web object access model and other characteristics. It also reviews the known prefetching algorithms that are based on different object selection criteria. Section 3 gives a simple theoretical analysis of the steady state hit rate and bandwidth consumption, as well as a description of the *H/B* metric and the metrics derived from it. The discussion in this section is based on [13], [20]. Section 4 proposes the *Objective-Greedy* prefetching algorithms based on the objective metrics to be improved. Section 5 gives a detailed formulation of our *H/B-Optimal* prefetching algorithm.

TABLE 1
Notations for Web Prefetching Analysis

Notation	Meaning
a	Total access rate
p_i	Access frequency of object i
l_i	Lifetime of object i
s_i	Size of object i
h_i	Hit rate of object i
b_i	Bandwidth of object i
Hit_{pref}	Overall hit rate with prefetching
Hit_{demand}	Overall hit rate without prefetching
BW_{pref}	Overall bandwidth with prefetching
BW_{demand}	Overall bandwidth without prefetching
S	Total set of web objects
S'	Set of prefetched objects

Section 6 presents the simulation results comparing the proposed algorithms with other known algorithms. Section 7 gives the concluding remarks.

2 PRELIMINARIES

The summary of notations for the web prefetching analysis is listed in Table 1.

2.1 Web Object Properties

To determine which objects to prefetch, we need to have some information about the characteristics of Web objects, such as their access frequencies, sizes, and lifetimes. Researchers have found that the object access model roughly follows Zipf-like distributions [2], which state that p_i, the access frequency of the ith popular object within a system, can be expressed as $\frac{k}{i}$, where k is a constant and $k = \frac{1}{\sum_i (\frac{1}{i})}$.

Using Zipf-like distributions to model the Web page requests, Glassman [12] found that they fit $\frac{k}{i}$ quite well, based on his investigation of a collection of 100,000 HTTP requests. A better approximation provided by Bestavros et al. [1] generalizes the Web objects' access frequency p_i as following a form of Zipf-like distribution:

$$p_i = k/i^{\alpha}, \quad where \quad k = \frac{1}{\sum_i (\frac{1}{i^{\alpha}})}. \tag{1}$$

The value of α varies in different traces. Bestavros et al. [1] recommend a value of 0.986 and Nishikawa et al. [18] suggest $\alpha = 0.75$, based on their 2,000,000 requests access log.

Another characteristic that affects the hit rate and network bandwidth consumption is the Web object's lifetime. Web objects are generally dynamic since they are updated from time to time. An access to a cached object that is obsolete would lead to a miss and, in turn, require downloading the updated version from the Web server [10], [15]. An object's lifetime is described as the average time interval between consecutive updates to the object. A prefetching algorithm should take into account each object's lifetime in the sense that objects with a longer lifetime are better candidates to be prefetched in order to minimize the extra bandwidth consumption.

Crovella and Bestavros [7] have shown that the sizes of static Web objects follow a Pareto distribution characterized

by a heavy tail. Crovella and Barford [8] showed that the actual sizes of dynamic Web objects follow a mixed distribution of heavy-tailed Pareto and lognormal distribution. Breslau et al. [2] showed that the distribution of object size has no apparent correlation with the distribution of access frequency and lifetime. Using this observation, Jiang et al. [13] simply assumed all objects to be of constant size in their experiment. However, a more reasonable approach assumes a random distribution of object sizes and is used in this paper.

2.2 Existing Prefetching Algorithms

Prefetch by Popularity. Markatos and Chironaki [17] suggested a "Top Ten" criterion for prefetching Web objects, which keeps in cache the 10 most popular objects from each Web server. Each server keeps records of accesses to all objects it holds and the top ten popular objects are pushed into each cache whenever they are updated. Thus, those top-ten objects are kept "fresh" in all caches. A slight variance of the "Top Ten" approach is to prefetch the m most popular objects from the entire system. Since popular objects account for more requests than less popular ones, *Prefetch by Popularity* is expected to achieve the highest hit rate [13].

Prefetch by Lifetime. Prefetching objects leads to extra bandwidth consumption since, in order to keep a prefetched object "fresh" in the cache, it is downloaded from the Web server whenever the object is updated. Starting from the point of view of bandwidth consumption, it is natural to choose those objects that are less frequently updated. *Prefetch by Lifetime* [13], as its name indicates, selects m objects that have the longest lifetime and, thus, intends to minimize the extra bandwidth consumption.

Good Fetch. Venkataramani et al. [20] proposed a *Good Fetch* criterion that balances the access frequency and update frequency (or lifetime) of Web objects. In the *Good Fetch* algorithm, the objects that have the highest probability of being accessed during their average lifetime are selected for prefetching. Assuming the overall object access rate to be a, the frequency of access to object i to be p_i, and the average lifetime of this object to be l_i, the probability that object i is accessed during its lifetime can be expressed as

$$P_{goodfetch} = 1 - (1 - p_i)^{al_i}. \qquad (2)$$

The *Good Fetch* algorithm prefetches a collection of objects whose $P_{goodfetch}$ exceeds a certain threshold. The intuition behind this criterion is that objects with relatively higher access frequencies and longer update intervals are more likely to be prefetched and this algorithm tends to balance the hit rate and bandwidth in that it increases the hit rate with a moderate increase of bandwidth usage. Venkataramani et al. [20] argued that this algorithm is optimal to within a constant factor of approximation. However, it could behave inefficiently under some specific access-update patterns (see Section 4.1).

APL Algorithm. Jiang et al. [13] provided another approach for choosing prefetched objects. Again by assuming a, p_i, and l_i as in *Good Fetch*, they used ap_il_i as the criterion. Those objects whose ap_il_i value exceeds a given threshold will be selected for prefetching. The ap_il_i value of

an object i represents the expected number of accesses to this object during its lifetime. The higher this value, the greater the chances (and, possibly, the greater the times) this object is accessed during its lifetime. Thus, prefetching such objects seems to have a better effect on improving the overall hit rate. This algorithm also intends to balance the hit rate and bandwidth consumption.

3 OBJECTIVE METRICS

3.1 Steady State Hit Rate

The access pattern of an object i is assumed to follow the Poisson distribution with the average access rate being ap_i, and the update interval is assumed to follow the exponential distribution with the average interval being l_i [20]. As per the analysis in [20], we define $P_{A_i}(t)$ as the probability that the last access occurs within time t in the past and $P_{B_i}(t)$ as the probability that no update occurs within time t in the past. Then, the following hold:

$$P_{A_i}(t) = 1 - e^{-ap_it}, \qquad (3)$$

$$P_{B_i(t)} = e^{-t/l_i}. \qquad (4)$$

For an object i that is not prefetched, a current access is a *hit* if the last access occurred after the last update and the probability of an access being a *hit* is given as follows:

$$P_{hit}(i) = \int_0^\infty P_{A_i}(t)dP_{B_i}(t) = \frac{ap_il_i}{ap_il_i + 1}. \qquad (5)$$

This probability is the hit rate of an object under on-demand caching and is also named the *freshness factor* of object i, or *f(i)*. For the prefetched objects, the hit rate is 1. So, the hit rate of object i is expressed as:

$$h_i = \begin{cases} \frac{ap_il_i}{ap_il_i+1}, & i \ is \ not \ prefetched \\ 1, & i \ is \ prefetched. \end{cases} \qquad (6)$$

The overall hit rate resulting from a prefetching mechanism is thus:

$$Hit_{pref} = \sum_i p_ih_i. \qquad (7)$$

Note that the overall on-demand hit rate is $\sum_i p_if(i)$.

3.2 Steady State Bandwidth Consumption

Let s_i be the size of object i. If object i is not prefetched, then only when an access results in a cache miss would this object be retrieved from the Web server. Thus, the bandwidth for this object is $ap_i(1 - f(i))s_i$. If object i is prefetched, then this object is downloaded from its Web server to the cache each time it is updated in the server and the bandwidth is $\frac{s_i}{l_i}$. So, we have the bandwidth consumption for object i [20]:

$$b_i = \begin{cases} ap_i(1 - f(i))s_i, & i \ is \ not \ prefetched \\ \frac{s_i}{l_i}, & i \ is \ prefetched. \end{cases} \qquad (8)$$

The total bandwidth resulting from prefetching is:

$$BW_{pref} = \sum_i b_i. \qquad (9)$$

3.3 H/B Model

Venkataramani et al. [20] did the performance evaluation of their *Good Fetch* algorithm based on computational simulations as well as trace-based experiments. The effects of prefetching were demonstrated with hit rate improvement and increase of bandwidth under various threshold values. A smaller threshold value for $P_{goodfetch}$ results in a higher hit rate and more bandwidth consumption because more objects are prefetched. According to how important the user-perceived latency is relative to the bandwidth consumption, different threshold values may be adopted in this algorithm. For example, if the waiting time is a critical requirement and there is adequate available bandwidth and hardware resources, a smaller threshold value could be chosen to achieve a higher hit rate.

A balanced measure of the prefetching algorithms was proposed by Jiang et al. [13]. It is called the *H/B* metric and is defined as:

$$H/B = \frac{Hit_{pref}/Hit_{demand}}{BW_{pref}/BW_{demand}}. \quad (10)$$

Here, Hit_{pref} and Hit_{demand} are the overall hit rate, with and without prefetching, respectively; BW_{pref} and BW_{demand} are the total bandwidth with and without prefetching.

The *H/B* metric expresses the ratio of hit rate improvement over the bandwidth increase. It is a quantitative evaluation of the hit rate improvement a prefetching algorithm can bring relative to excessive bandwidth consumption. In addition, a more generalized form, H^k/B [13], can be used to give relative importance to hit rate or bandwidth by varying the value of k.

$$H^k/B = \frac{(Hit_{pref}/Hit_{demand})^k}{BW_{pref}/BW_{demand}}. \quad (11)$$

With this form of the *H/B* metric, $k > 1$ indicates an environment with abundant available bandwidth and hardware resources and the improvement of hit rate is more favored than the economy on bandwidth. When the network bandwidth is limited, a smaller k (possibly $k < 1$) is used instead.

Jiang et al. [13] used the *H/B* metric and its H^k/B form to evaluate the performance of *Good Fetch* [20] and *Popularity* [17], as well as the *Lifetime* and *APL* algorithms that they proposed. However, Jiang et al. did not propose any algorithm that directly targeted either the *H/B* or the H^k/B metric.

4 OBJECTIVE-GREEDY PREFETCHING

In this section, we first discuss some drawbacks of existing algorithms (Section 4.1). In Sections 4.2, 4.3, and 4.4, we provide a detailed explanation and theoretical analysis of our *Objective-Greedy* prefetching algorithms. This family of algorithms is greedy because each algorithm always chooses to prefetch those objects that would most significantly improve the performance metric it is aimed at. In Section 4.2, we formulate the theory for and then derive the *H/B-Greedy* algorithm that greedily improves the performance as measured by the *H/B* metric. In Sections 4.3 and 4.4, the *Hit Rate-Greedy* and the

Bandwidth-Greedy prefetching algorithms, which are shown to be special cases of the H^k/B-*Greedy* prefetching with k set to infinity and to 0, respectively, are proposed.

4.1 Problems with Existing Algorithms

The existing prefetching algorithms (*Good Fetch* [20] and *APL* [13]) stemmed from intuition that balances the hit rate and bandwidth. However, they are actually far from being ideal in choosing the prefetched objects to achieve the optimal performance with regard to hit rate or bandwidth.

- For an object i that is not prefetched, for example, if the accesses and updates to this object are alternated (i.e., there is exactly one access between every two consecutive updates to object i), the on-demand hit rate is 0 and the bandwidth consumption of this object is $\frac{s_i}{l_i}$. If we prefetch object i, the hit rate of this object is improved to 1 and the bandwidth is unchanged, which indicates that object i is an ideal object to prefetch. However, this "perfect" candidate is not always favored by *Good Fetch* or *APL* since $P_{goodfetch}$ or apl of this object is not always high compared to other objects. Note that this special access-update pattern cannot be captured by the long-term characteristics p and l of the system model.

- The *APL* algorithm, for example, prefers those objects with a larger product of lifetime and access frequency. Consider two objects, A and B, of the same size; the access rate for A is 30 accesses/day and that for B is three accesses/day, the average lifetime for A is one day and that for B is 15 days. Which object should *APL* choose to prefetch? This algorithm would more likely choose B rather than A, but a careful analysis shows that, to achieve a higher hit rate, we should choose A, for less bandwidth, we should choose B, and, for a higher value of the *H/B* metric, we need to know the characteristics of all other objects before making a decision!

  Analysis: Suppose $p_A = 30c$, $p_B = 3c$, where c is a constant. We have

$$(APL)_A = a \cdot 30c \cdot 1 = 30ac,$$
$$(APL)_B = a \cdot 3c \cdot 15 = 45ac.$$

  B is more preferred by *APL* since $(APL)_B > (APL)_A$. The overall hit rate improvements, respectively, by prefetching objects A and B are:

$$(\Delta H)_A = p_A(1 - f(A)) = \frac{30c}{30ac + 1},$$
$$(\Delta H)_B = p_B(1 - f(B)) = \frac{3c}{45ac + 1}.$$

  As $(\Delta H)_A > (\Delta H)_B$, prefetching A achieves higher hit rate improvement than prefetching B.

  The extra bandwidth consumptions imposed by prefetching objects A and B are:

$$(\Delta BW)_A = \frac{s_A}{l_A}(1 - f(A)) = \frac{s_A}{30ac + 1},$$
$$(\Delta BW)_B = \frac{s_B}{l_B}(1 - f(B)) = \frac{s_B}{15(45ac + 1)}.$$

As $s_A = s_B$, we have $(\Delta BW)_A > (\Delta BW)_B$ and prefetching B would cause less bandwidth consumption than prefetching A.

The problems mentioned here are primarily due to the fact that the existing prefetching algorithms simply choose to prefetch an object based on its individual characteristics with no consideration for issues having broader impact, such as the following: 1) How much effect would prefetching a specific object have on the overall performance? 2) Between two objects with different access-update characteristics, which one, if prefetched, would have a greater influence on the total hit rate and bandwidth?

Another problem worth noting is the actual sizes of objects. The existing algorithms simplified the object sizes to be constant. However, a random distribution may reflect a better approximation of object sizes in the Internet.

4.2 H/B-Greedy Prefetching

Note that a prefetching algorithm selects a subset of objects from the cache. When an object i is prefetched, the hit rate is increased from $\frac{ap_i l_i}{ap_i l_i + 1}$ to 1, which is $\frac{1}{f(i)}$ times that of on-demand caching; the bandwidth for object i is increased from $\frac{s_i}{l_i + \frac{1}{ap_i}}$ to $\frac{s_i}{l_i}$, which is also $\frac{1}{f(i)}$ times the bandwidth for object i under on-demand caching. Prefetching an object leads to the same relative increase in its hit rate and bandwidth consumption.

Recall the *H/B* metric (10) that measures the balanced performance of a prefetching algorithm. We observe that this measure uses the hit rate and bandwidth of on-demand caching as a baseline for comparison and, as they are constants, the *H/B* metric is equivalent to:

$$\left(\frac{H}{B}\right)_{pref} = \frac{Hit_{pref}}{BW_{pref}} = \frac{\sum_i p_i h_i}{\sum_i b_i},$$ (12)

where h_i and b_i are the hit rate and bandwidth of object i, respectively, as described in Section 3.

Consider the *H/B* ratio under on-demand caching:

$$\left(\frac{H}{B}\right)_{demand} = \frac{Hit_{demand}}{BW_{demand}} = \frac{\sum_i p_i f(i)}{\sum_i \frac{s_i}{l_i} f(i)}.$$ (13)

A prefetching algorithm chooses an appropriate subset of objects from the entire collection and, for each of these prefetched objects, say object i, we simply change the corresponding *f(i)* term to 1 in (13) to obtain $(H/B)_{pref}$. Our *H/B-Greedy* algorithm aims to select a group of objects to be prefetched such that the *H/B* metric would achieve a better value than that obtained by the existing algorithms. Since the object characteristics, such as access frequencies, lifetimes, and object sizes, are all known to the algorithm, it is possible that, given a number m, we could select m objects to prefetch such that $(H/B)_{pref}$ reaches the maximum possible value. This optimization problem can be formalized as finding a subset S' of size m from the entire collection of objects, S, such that $(H/B)_{pref}$ is maximized:

$$S' = argmax_{S' \subset S, |S'|=m} \left[\left(\frac{H}{B}\right)_{pref} \right]$$

$$= argmax_{S' \subset S, |S'|=m} \left[\frac{\sum_{i \in S} p_i f(i) + \sum_{j \in S'} p_j(1 - f(j))}{\sum_{i \in S} \frac{s_i}{l_i f(i)} + \sum_{j \in S'} \frac{s_j}{l_j}(1 - f(j))} \right].$$ (14)

This is a variation of the *maximum weighted average (MWA) problem* proposed by Eppstein and Hirschberg [11]. The *maximum weighted average* problem is best explained by the example of course selection for maximizing the GPA: "Given an academic record of a student's curriculum with a total of n courses, select a subset of k courses (each with different score and credit hours) from the record that would obtain the highest weighted average (GPA) among all subsets of k courses." Eppstein and Hirschberg gave an algorithm to solve this problem with an expected time complexity of $O(n)$. If we introduce the constraint that, among the k courses, c ($c < k$) specific courses must be included (i.e., preselected), we now have an MWA problem of choosing $(k - c)$ courses such that the weighted average of the k courses is maximized. Similarly, our optimal object selection problem has to deal with the on-demand hit rate Hit_{demand} and bandwidth BW_{demand}, where Hit_{demand} and BW_{demand} correspond to the sum of weighted scores and sum of weights of the preselected courses, respectively. This makes our problem equivalent to the course selection problem *with preselected courses*. In Section 5, we modify Eppstein and Hirschberg's algorithm and present a linear time algorithm that solves this optimal object selection problem, described in (14).

In this section, we introduce a greedy algorithm, the *H/B-Greedy* prefetching algorithm, as a first solution toward designing an optimal solution to the problem in (14). It aims to improve *H/B* by greedily choosing objects based on their individual characteristics. The *H/B-Greedy* algorithm attempts to select those objects that, if prefetched, would have the most benefit for *H/B*. For example, suppose, initially, no object is prefetched and the *H/B* value is expressed as $(\frac{H}{B})_{demand}$; now, if we prefetch object j, the *H/B* value will be updated to:

$$\frac{\sum_{i \in S} p_i f(i) + p_j(1 - f(j))}{\sum_{i \in S} \frac{s_i}{l_i} f(i) + \frac{s_j}{l_j}(1 - f(j))} =$$

$$\left(\frac{H}{B}\right)_{demand} \times \frac{\left(1 + \frac{p_j(1 - f(j))}{\sum_{i \in S} p_i f(i)}\right)}{\left(1 + \frac{\frac{s_j}{l_j}(1 - f(j))}{\sum_{i \in S} \frac{s_i}{l_i} f(i)}\right)} = \left(\frac{H}{B}\right)_{demand} \times incr(j).$$ (15)

Here, *incr(j)* is the factor that indicates the amount by which *H/B* can be increased if object j is prefetched. We call *incr(j)* the *increase factor* of object j.

Our *H/B-Greedy* prefetching algorithm (Algorithm 1) thus uses *increase factor* as a selection criterion and chooses to prefetch those objects that have the greatest *increase factors*. We hypothesize that this algorithm is an approximation to the optimal solution because, in real-life prefetching, due to the bandwidth constraints, the number of prefetched objects m is usually small. Furthermore, the *increase factors* of

individual objects are generally very close to 1 due to the large number of objects in the system. Thus, previous selection of prefetched objects is expected to have little effect on subsequent selections in terms of improving H/B. This hypothesis is verified by our simulations described in Section 6.

Procedure **H/B-Greedy** in Algorithm 1 takes the set of all objects in the Web servers—S, the number of objects to be prefetched—m, and the total access rate—a, as the inputs. Each object i is represented by a tuple $\langle p_i, l_i, s_i \rangle$, where $p_i, l_i,$ and s_i denote the access frequency, the lifetime, and size of the object, respectively.

Algorithm 1 H/B-Greedy prefetching
1: *procedure* **H/B-Greedy**(S, m, a)
2:   **Inputs:**
3:     a set of objects of type $\langle p_i, l_i, s_i \rangle$: S
4:     number of objects to be prefetched: n
5:     the total access rate: a
6:
7:   **for** each object $i \in S$ **do**
8:     compute the freshness factor: $f(i) = \frac{ap_i l_i}{ap_i l_i + 1}$
9:   **end for**
10: compute the on-demand overall hit rate
      $Hit_{demand} = \sum_{i \in S} p_i f(i)$
11: compute the on-demand overall bandwidth
      $BW_{demand} = \sum_{i \in S} \frac{s_i}{l_i} f(i)$
12: **for** each object $i \in S$ **do**
13:     compute the increase factor $incr(i)$ as defined in (15)
14: **end for**
15: select m objects with the largest *increase factors* using
      randomized selection
16: mark the selected objects *"prefetched"*
17:
18: $Hit_{pref} = Hit_{demand}$
19: $BW_{pref} = BW_{demand}$
20: **for** each object j that is prefetched **do**
21:     $Hit_{pref} = Hit_{pref} + p_j(1 - f(j))$
22:     $BW_{pref} = BW_{pref} + \frac{s_j}{l_j}(1 - f(j))$
23: **end for**
24: **return** $\frac{Hit_{pref}}{BW_{pref}}$

Analysis: The computation of freshness factor $f(i)$ and increase factor $incr(i)$ each take constant time. Hence, the for-loops in lines 7-9 and lines 12-14 each take $O(|S|)$ time. The computation in line 10 and line 11 also takes $O(|S|)$ time each. In line 15, we use randomized selection to select m objects with the greatest *increase factors*. The expected time for this selection operation is $O(|S|)$. Selecting the mth largest value using randomized selection takes expected time of $O(|S|)$ [6]. The $m - 1$ largest values get moved by the partitioning process to one side of the selected value (the mth largest value) in the array after selection completes; thus, these m values are explicitly identified. Line 16 and the loop in lines 20-23 each take $O(m)$ time ($m < |S|$). Thus, the total expected time complexity of the *H/B-Greedy* algorithm is $O(|S|)$.

The space requirement for this algorithm includes the space for storing $f(i)$, $incr(i)$, and the characteristic triple $\langle p_i, l_i, s_i \rangle$ for each object, which totals up to $O(|S|)$.

4.3 Hit Rate-Greedy Prefetching

Sometimes it is desirable to maximize the overall hit rate given the number of objects to prefetch, m. Jiang et al. [13] claimed that *Prefetch by Popularity* achieves the highest possible hit rate. However, a special form of our *Objective-Greedy* algorithms would actually obtain a higher hit rate than *Prefetch by Popularity*.

Observe that the contribution to the overall hit rate by prefetching object i is:

$$H\_contr(i) = p_i(1 - f(i)) = \frac{p_i}{ap_i l_i + 1}. \qquad (16)$$

Thus, if we choose to prefetch those objects with the largest hit rate contributions, the resulting overall hit rate must be maximized. We call this algorithm *Hit Rate-Greedy* (or *H-Greedy*) prefetching and it is obtained from our *Objective-Greedy* principle when we try to optimize the overall hit rate as the objective metric. *H-Greedy* prefetching is an extreme case of the H^k/B-*Greedy* prefetching algorithm: As we care only about the hit rate and the bandwidth is of no importance, we let k in the H^k/B metric go to infinity and this metric becomes the hit rate metric:

$$\lim_{k \to +\infty} \left[\frac{(Hit_{pref}/Hit_{demand})^k}{BW_{pref}/BW_{demand}} \right]^{\frac{1}{k}} = Hit_{pref}/Hit_{demand}.$$

We claim *H-Greedy* prefetching is *H-Optimal* since

$$Hit_{pref} = Hit_{demand} + \sum_{i \in S'} H\_contr(i),$$

where S' denotes the set of prefetched objects.

4.4 Bandwidth-Greedy Prefetching

Another optimization problem in prefetching is to minimize the excessive bandwidth consumption, given the number of objects to prefetch. Intuition may suggest that *Prefetch by Lifetime* has the least bandwidth usage [13]. However, by applying our *Objective-Greedy* principle with bandwidth as the objective, we get an algorithm that results in even less bandwidth consumption.

Using analogous reasoning to that for the *Hit Rate-Greedy* algorithm, the extra bandwidth contributed by prefetching object i is:

$$B\_contr(i) = \frac{s_i}{l_i}(1 - f(i)) = \frac{s_i}{ap_i l_i^2 + l_i}. \qquad (17)$$

Hence, if we prefetch those objects with the least *B_contr(i)* values, we could finally expect the minimum excessive bandwidth given the number of prefetched objects, m. We call this algorithm the *Bandwidth-Greedy* (or *B-Greedy*) prefetching and it is another extreme case of the H^k/B-*Greedy* prefetching algorithm: If we care only about the bandwidth usage and not the hit rate, we let the exponent k tend to limit zero and the H^k/B metric becomes the bandwidth metric. *B-Greedy* is also *B-Optimal* since

$$BW_{pref} = BW_{demand} + \sum_{i \in S'} B\_contr(i),$$

where S' denotes the set of prefetched objects.

TABLE 2
Analogy between $H/B\text{-}Optimal$ Prefetching and the Maximum Average Course Selection Problem,
Enhanced to Deal with Preselected Courses

$H/B\text{-}Optimal$ Prefetching	Meaning	Course Selection	Meaning
i, j	Object	i, j	Course
$p_i(1 - f(i))$	Hit rate contribution by prefetching i	v_i	Weighted score of i
$\frac{s_i}{l_i}(1 - f(i))$	Bandwidth contribution by prefetching i	w_i	Weight of i
$\sum_{i \in S} p_i f(i)$	On-demand hit rate	v_0	Sum of pre-selected scores (weighted)
$\sum_{i \in S} \frac{s_i}{l_i} f(i)$	On-demand bandwidth	w_0	Sum of pre-selected weights
S	Entire set of objects	S	Entire set of courses

A simulation-based performance comparison of these two algorithms and all the other algorithms is shown in Section 6.3.

5   H/B-OPTIMAL PREFETCHING

We now propose an optimal prefetching approach that achieves the highest H/B value given the number of objects to prefetch. The goal is to select a subset of m objects to prefetch such that the objective $(H/B)_{pref}$ is maximized. The objective $(H/B)_{pref}$ is defined in (12).

5.1   Designing an Optimal Solution

As mentioned in Section 4.2, this optimization problem is equivalent to the *maximum weighted average problem* modified to deal with preselected items. Eppstein and Hirschberg [11] studied a simpler version, which is the *maximum weighted average problem without preselected items*. They used an example of course selection: From a total of n courses, each with weight w_i and weighted score v_i, select m courses that maximize the weighted average. They proposed a linear time solution to this course selection problem. We now require the course selection problem to also model preselected courses besides the m courses to be chosen. Thus, the *H/B-Optimal* prefetching problem is analogous to the weighted average problem, however, with preselected courses that must be included besides those m courses.

The analogy between the *H/B-Optimal* prefetching problem and the course selection problem, now modified to model the preselected courses, is given in Table 2.

To be consistent with Eppstein and Hirschberg's notations in the problem of choosing optimal subsets, we rephrase the selection problem as removing k objects from the entire set of n objects, where $k = n - m$. In [11], they defined a characteristic function for course i:

$$r_i(x) = v_i - w_i x. \qquad (18)$$

$r_i(x)$ represents the weighted difference between the score of course i and an arbitrary value, x. If a subset of courses average to x, the sum of $r_i(x)$ over the subset is 0.

The above formulation of $r_i(x)$ did not address preselection, which we have to take into account in our specification. To specify such a characteristic for the case with preselected items, we redefine $r_i(x)$ as

$$r_i(x) = \left(v_i + \frac{v_0}{n - k}\right) - \left(w_i + \frac{w_0}{n - k}\right)x. \qquad (19)$$

For simplicity, we use the terms v_i, w_i, v_0, w_0 borrowed from the context of course selection. Note that v_0 and w_0 represent on-demand hit rate and bandwidth, respectively, in the context of Web prefetching. The terms $\frac{v_0}{n-k}$ and $\frac{w_0}{n-k}$ are introduced to account for the effect of preselection, under a specific case of choosing $n - k$ items. Their significance will be shown shortly.

We now define a heuristic function $F(x)$ as

$$F(x) = max_{|S'|=n-k, S' \subset S}\left[\sum_{i \in S'} r_i(x)\right]. \qquad (20)$$

By definition, $F(x)$ is the sum of the $n - k$ greatest $r_i(x)$ values. We have the following properties related to $F(x)$:

Lemma 1. *Suppose A^* is the maximum weighted average after choosing $n - k$ items, with preselected items whose summed value is v_0 and summed weight is w_0. Then, for any subset $S' \subset S$ and $|S'| = n - k$,*

$$(1) \sum_{i \in S'} r_i(A^*) \leq 0,$$

$$(2) \sum_{i \in S'} r_i(A^*) = 0 \text{ iff } S' \text{ is the optimal subset.}$$

Proof. Let $S' \subset S$, and $|S'| = n - k$. Then,

$$\sum_{i \in S'} r_i(A^*) = \sum_{i \in S'}\left[\left(v_i + \frac{v_0}{n - k}\right) - \left(w_i + \frac{w_0}{n - k}\right)A^*\right]$$

$$= \left(w_0 + \sum_{i \in S'} w_i\right)\left(\frac{v_0 + \sum_{i \in S'} v_i}{w_0 + \sum_{i \in S'} w_i} - A^*\right).$$

If S' is not an optimal subset, $\frac{v_0 + \sum_{i \in S'} v_i}{w_0 + \sum_{i \in S'} w_i} < A^*$, thus $\sum_{i \in S'} r_i(A^*) < 0$. If S' is an optimal subset, then $\frac{v_0 + \sum_{i \in S'} v_i}{w_0 + \sum_{i \in S'} w_i} = A^*$ and we have $\sum_{i \in S'} r_i(A^*) = 0$.   □

Lemma 1 indicates that the optimal subset contains those items that have the $n - k$ largest values of $r_i(A^*)$. To find the optimal subset, we can plot the curve for each $r_i(x)$ and choose the $n - k$ objects with the largest $r_i(A^*)$ values.

Example. See Fig. 1. In this example, $n = 6$ and $k = 2$. Each line represents the characteristic function $r_i(x)$ for some
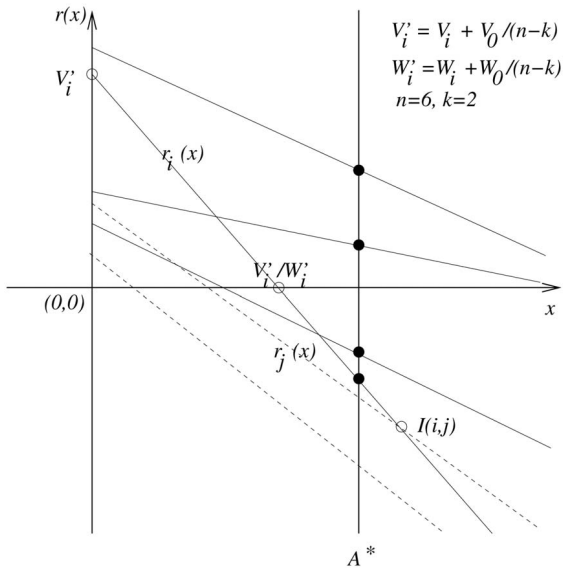
Fig. 1. Choosing the optimal subset if A^* is known.

object, i. $I(i, j)$ denotes the intersection of $r_i(x)$ and $r_j(x)$. If the optimal average A^* is known, then the optimal subset consists of four objects whose intersections with line $x = A^*$ are marked by solid dots.

We cannot directly obtain the optimal subset by plotting the curve for each $r_i(x)$ and choosing the $n - k$ objects with the largest $r_i(A^*)$ since the value of A^* is unknown. However, we can use the following lemma to limit the range of A^* to a sufficiently small interval so that the total ordering of $r_i(x)$ values within this small interval can be determined.

Lemma 2. *1)* $F(x) > 0$ *iff* $x < A^*$, *2)* $F(x) < 0$ *iff* $x > A^*$, *3)* $F(x) = 0$ *iff* $x = A^*$.

Proof. Part 1. Let $x < A^*$. By definition,

$$F(x) = max_{|S'|=n-k, S' \subset S} \left(\sum_{i \in S'} r_i(x) \right)$$

and there exists subset S_x such that $F(x) = \sum_{i \in S_x} r_i(x)$. Let S^* be the optimal subset to our selection problem. Then:

$$F(x) = \sum_{i \in S_x} r_i(x)$$
$$\geq \sum_{i \in S^*} r_i(x) = \sum_{i \in S^*} \left[\left(v_i + \frac{v_0}{n-k} \right) - \left(w_i + \frac{w_0}{n-k} \right) x \right]$$
$$> \sum_{i \in S^*} \left[\left(v_i + \frac{v_0}{n-k} \right) - \left(w_i + \frac{w_0}{n-k} \right) A^* \right] = F(A^*) = 0.$$

To prove the other direction, assume $F(x) > 0$. Thus, $F(x) = \sum_{i \in S_x} r_i(x) > 0$.
According to Lemma 1, $\sum_{i \in S_x} r_i(A^*) \leq 0$.
We have:

$$\sum_{i \in S_x} \left[\left(v_i + \frac{v_0}{n-k} \right) - \left(w_i + \frac{w_0}{n-k} \right) x \right]$$
$$> \sum_{i \in S_x} \left[\left(v_i + \frac{v_0}{n-k} \right) - \left(w_i + \frac{w_0}{n-k} \right) A^* \right].$$

Thus, $x < A^*$.
Part 2. Let $x > A^*$, then $r_i(x) < r_i(A^*)$.
Using Lemma 1, we have:

$$F(x) = \sum_{i \in S_x} r_i(x) < \sum_{i \in S_x} r_i(A^*) \leq 0.$$

Now, let $F(x) < 0$. We have:

$$F(x) = \sum_{i \in S_x} r_i(x) \geq \sum_{i \in S^*} r_i(x).$$

Thus, $\sum_{i \in S^*} r_i(x) < 0$ and, hence,

$$\sum_{i \in S^*} r_i(x) < \sum_{i \in S^*} r_i(A^*).$$

This is expanded as:

$$\sum_{i \in S^*} \left[\left(v_i + \frac{v_0}{n-k} \right) - \left(w_i + \frac{w_0}{n-k} \right) x \right]$$
$$< \sum_{i \in S^*} \left[\left(v_i + \frac{v_0}{n-k} \right) - \left(w_i + \frac{w_0}{n-k} \right) A^* \right],$$

which simplifies to $x > A^*$.
Part 3. Follows trivially. □

Lemma 2 provides a powerful method for searching for or approaching the maximum average using a binary-like search: We can narrow the range of A^* down to (x_l, x_r) if we know $F(x_l) > 0$ and $F(x_r) < 0$. The computation of $F(x)$ can be performed within $O(n)$ time using a randomized selection procedure.

Since $r_i(x)$ is a linear function, by computing the intersection of two curves and comparing their slopes, we can determine the ordering of these two curves within any interval that does not contain their intersection. If we use Lemma 2 to narrow down the range of A^* to an interval that contains no intersection, then the total ordering of $r_i(x)$ is determined within that interval and so is the ordering of $r_i(A^*)$ among all objects i. However, this method requires $O(n^2)$ time, as the computation of intersections needs $O(n^2)$ time; the binary traversal needs $O(\lg(n))$ comparisons and each comparison needs $O(n)$ time, which is too much overhead. In addition, to find an optimal subset, we do not need the total ordering among the entire set. We extend the randomized linear time selection algorithm of Eppstein and Hirschberg to now account for preselection. This gives the *H/B-Optimal* prefetching algorithm, which has the expected linear time complexity.

5.2 H/B-Optimal Prefetching Algorithm

The structure of the *H/B Optimal* prefetching algorithm is based on the Eppstein-Hirschberg algorithm. The difference is that we use our new definition of the object's characteristic function, $r_i(x)$, to account for the preselection imposed by the on-demand hit rate and bandwidth in the context of the Web prefetching problem. In a sense, this transformation is a linear-time reduction.

The algorithm maintains two properties for each object i: its weighted value $v_i = H\_contr(i)$ as given in (16) and weight $w_i = B\_contr(i)$ as given in (17). The value and weight represent the object's additional contribution to the overall hit rate and bandwidth if the object is prefetched.

Function **H/B-Optimal** tries to find those $n - k$ objects with the largest $r(A^*)$ values. The inputs to this function are:

1. S, the entire set of objects,
2. k, the number of objects to be excluded from the selection,
3. h_{demand}, the on-demand hit rate, and
4. b_{demand}, the on-demand bandwidth.

Algorithm 2 H/B-optimal Prefetching (based on the maximal weighted average algorithm)
1: *function* **H/B-Optimal**$(S, k, h_{demand}, b_{demand})$
2: **for** $i \in S$ **do**
3:      $v_i = v_i + \frac{h_{demand}}{|S|-k}$, $w_i = w_i + \frac{b_{demand}}{|S|-k}$
4: **end for**
5: **if** $k = 0$ **then**
6:      **return** $\frac{h_{demand}}{b_{demand}}$
7: **end if**
8: $left = 0$, $right = +\infty$
9: **while** $|S| > 1$ **do**
10:     randomly choose i from S
11:     **for** $j \in S$ **do**
12:         switch(**compare**(*i,j,left,right*))
13:             case *EQUAL*: $E = E \cup \{j\}$, break;
14:             case *LESS*: $Y = Y \cup \{j\}$, break;
15:             case *LARGER*: $X = X \cup \{j\}$, break;
16:             case *UNKNOWN*: $Z = Z \cup \{j\}$, break;
17:     **end for**
18:     **repeat**
19:         **if** $|Z| > 0$ **then**
20:             $A = $ median(i, Z)
21:             $FA = \mathbf{F}(A, S, k)$
22:             **if** $FA = 0$ **then**
23:                 **return** A
24:             **else if** $FA > 0$ **then**
25:                 $left = A$
26:             **else**
27:                 $right = A$
28:             **end if**
29:             **for** $j \in Z$ **do**
30:                 switch(**compare**(*i,j,left,right*))
31:                     case *LESS*: $Y = Y \cup \{j\}$, $Z = Z - \{j\}$, break;
32:                     case *LARGER*: $X = X \cup \{j\}, Z = Z - \{j\}$, break;
33:                 **end for**
34:         **end if**
35:         **if** $|X| + |E| \geq |S| - k$ **then**
36:             remove $min(|E|, |X| + |E| - (|S| - k))$ members of E from S
37:             remove Y from S
38:             $k = k - (num\ \ of\ \ removed\ \ objects)$
39:         **else if** $|Y| + |E| \geq k$ **then**
40:             mark (as "prefetched") and combine

$min(|E|, |Y| + |E| - k)$ members of E and all members of X into a single object c
41:             $X = c$
42:             $S = S - \{combined\ \ members\} + \{c\}$
43:         **end if**
44:     **until** $|Z| \leq |S|/32$
45: **end while** {Now $|S| = 1$ and the only object in S is referred to as c}

46: **return** v_c/w_c

1: *function* **compare**$(i, j, left, right)$
2: **if** $w_i = w_j$ **then**
3:      $\delta = v_j - v_i$, $intersection = -\infty$
4: **else**
5:      $\delta = w_i - w_j$, $intersection = \frac{v_i - v_j}{w_i - w_j}$
6: **end if**
7: **if** $\delta = 0$ **then**
8:      **return** *EQUAL*
9: **else if** ($intersection \leq left$ and $\delta > 0$) or ($intersection \geq right$ and $\delta < 0$) **then**
10:     **return** *LARGER*
11: **else if** ($intersection \leq left$ and $\delta < 0$) or ($intersection \geq right$ and $\delta > 0$) **then**
12:     **return** *LESS*
13: **else**
14:     **return** *UNKNOWN*
15: **end if**
16:
17: *function* **median**(i, Z)
18: **for** $j \in Z$ **do**
19:     $inter(j) = \frac{v_i - v_j}{w_i - w_j}$
20: **end for**
21: **return** median value in array $inter$
22:
23: *function* **F**(A, S, k)
24: **for** $i \in S$ **do**
25:     $r_i(A) = v_i - Aw_i$
26: **end for**
27: **return** $\sum (largest\ \ (|S| - k)\ \ r_i(A)\ \ values)$

The **for**-loop in lines 2-4 adjusts the values of v_i and w_i for each object, i, to address the issue of preselection, as described in (19). The notion "value" and "weight" in the following explanation will therefore denote the adjusted meanings.

In the outer loop (lines 9-45), we randomly select an object i from the current set S and compare $r_i(A^*)$ value with $r_j(A^*), \forall j \in S$. As A^* is unknown, the comparison is performed indirectly, as implemented in function **compare**, which is called to compute the ordering between two linear functions $r_i(x)$ and $r_j(x)$ within the range *(left, right)*. If the intersection of $r_i(x)$ and $r_j(x)$ lies out of this range, we can decide that one of the lines goes above the other over the entire range by comparing their slopes. Since this range always contains A^*, the ordering between $r_i(A^*)$ and $r_j(A^*)$ is determined. However, if the intersection lies within this range, the ordering between $r_i(A^*)$ and $r_j(A^*)$ is undetermined.

The function **H/B-Optimal** maintains four sets:

- X, contains those objects whose $r(A^*)$ are known to be greater than that of object i;
- Y, contains those objects whose $r(A^*)$ are known to be less than that of object i;
- E, contains all the objects in S that have the same value and weight as object i;
- Z, contains those objects for which we currently do not know whether its $r(A^*)$ is greater than, less than, or equal to that of object i.

The range of A^* is initially set to $(0, \infty)$, which is apparently true in our context. Then, after calling function **compare** to compute the ordering between i and j within the current range, we insert object j into X, Y, E, or Z according to the result returned by **compare** (lines 11-17).

For any object j in Z, the abscissa of the intersection of $r_j(x)$ and $r_i(x)$ lies in the current A^*-range, *(left, right)*. In the inner loop (lines 18-44), we select the median value of these x-coordinates, A. By Lemma 2, if $F(A) > 0$, it is known that $A < A^*$ and, if $F(A) < 0$, then $A > A^*$. Thus, we narrow the range of A^* down by roughly a half and can expect to move some objects from Z into X or Y in each iteration. After the size of Z is adequately small, say $(|S|/32)$, we may have enough elements in X or Y to perform the removing or *combination* operation that reduces the size of the remaining problem.

1. If $|X| + |E| > |S| - k$: All members in Y and some members in E can be ranked within the k objects with the least $r(A^*)$ values. These objects can be removed from our selection and the remaining problem has a smaller size.
2. If $|Y| + |E| > k$: All members in X and some in E can be ranked as within the $|S| - k$ objects with the greatest $r(A^*)$ values and they must be selected.

Once we determine that some objects must be selected, we *combine* them into a single member of S by adding their values and weights, as the value and weight of the *combined* object.

Suppose the number of those selected object is p, then the remaining selection domain reduces to $|S| - p + 1$ and the number of objects to be selected reduces to $|S| - p + 1 - k$.

Finally, when there is only one member remaining in S, this is the *combination* of $n - k$ objects with the highest $r(A^*)$ values. These objects constitute the optimal subset $S^*(|S^*| = n - k)$ and the return value

$$\frac{v_c}{w_c} = \frac{v_0 + \sum_{i \in S^*} v_i}{w_0 + \sum_{i \in S^*} w_i}$$

is the optimal $(H/B)_{pref}$ we are seeking.

In the implementation of the prefetching operation, for each object that is to be *combined* as mentioned above, we mark it as "*prefetched*" before it is removed from the selection domain S.

Complexity: The expected time complexity of **H/B-Optimal** is $O(|S|)$; see the Appendix.

5.3 H/B-Optimal versus H/B-Greedy Prefetching

H/B-Greedy prefetching is useful even though we have found the *H/B-Optimal* prefetching algorithm. The following are the reasons:

1. Our simulation experiments (Section 6) confirm the hypothesis (Section 4.2) that *H/B-Greedy* behaves almost as well as *H/B-Optimal* prefetching and *H/B-Greedy* is easier to implement. Note that, although both *H/B-Optimal* and *H/B-Greedy* have a time complexity of $O(n)$, the *H/B-Optimal* prefetching has a much larger constant factor.
2. *H/B-Greedy* prefetching is especially convenient in a dynamic environment where the characteristics of Web objects could change from time to time. When such a change occurs, the content distribution server implementing *H/B-Greedy* can simply accommodate this change as follows: Suppose the characteristics of object i has changed:

   - Object i was not prefetched and *incr(i)* has increased. Select the minimum *increase factor* among those prefetched objects, *incr(j)*. If *incr(i) > incr(j)*, then substitute object j with object i in the prefetched set.
   - Object i was prefetched and *incr(i)* has decreased. If, now, *incr(i)* is the minimum in the prefetched set, then select the maximum *increase factor* among those objects that are not prefetched, *incr(j)*. If *incr(i) < incr(j)*, substitute object i with object j in the prefetched set.
   - Otherwise, do nothing.

   The above operations are easy to implement with little overhead. However, in a content distribution server that implements *H/B-Optimal* prefetching, such a change requires the entire algorithm to execute again, with considerable overhead.

6 SIMULATIONS AND RESULTS

6.1 Evaluation of H/B-Greedy and H/B-Optimal Prefetching Algorithms

We ran simulation experiments on the following six prefetching algorithms: *Popularity* [17], *Good Fetch* [20], *APL* [13], *Lifetime* [13], and *H/B-Greedy* and *H/B-Optimal*, which are proposed in this paper. The simulation model made the following assumptions:

- Object access frequency p_i follows Zipf's distribution with $\alpha = 0.75$ [13].
- Object size s_i is randomly distributed between 1 and 1M bytes. Unlike the previous experiments, which assumed a fixed object size, this assumption aims to capture more realistic data on the performance.
- Object lifetime l_i is randomly distributed between 1 and 100,000 seconds.
- The total access rate a is 0.01/second.

We ran four sets of simulations in which the total number of objects was set to 10^3, 10^4, 10^5, and 10^6, respectively. In each set of simulations, the six prefetching algorithms were experimented with. For each simulation
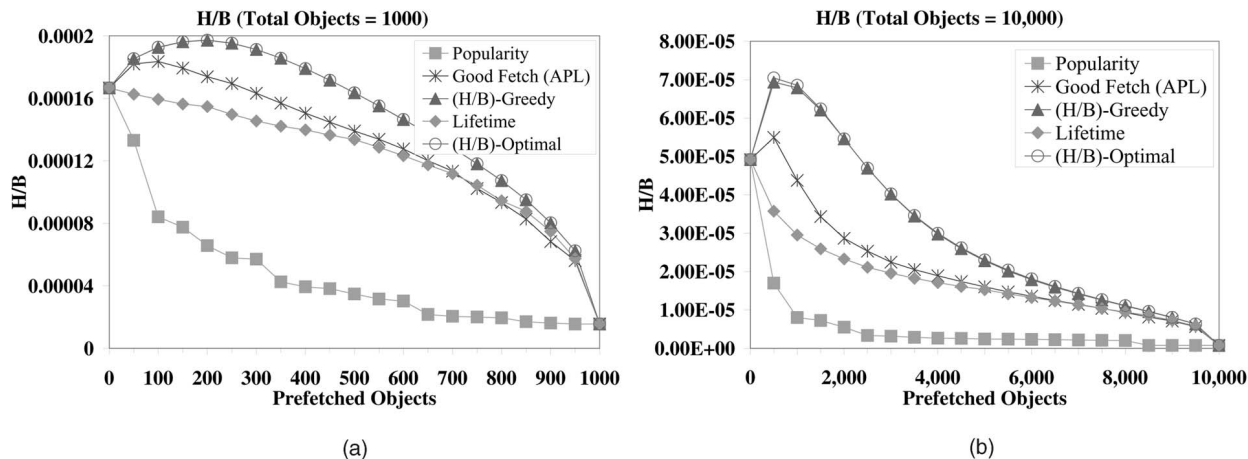
Fig. 2. *H/B* for a total of (a) 1,000 objects, (b) 10,000 objects.
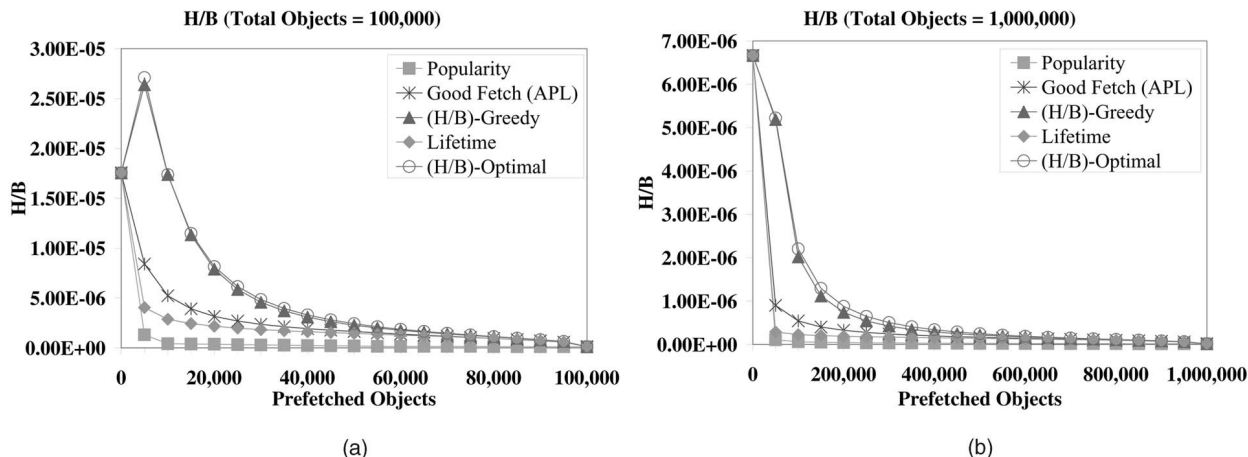
Fig. 3. *H/B* for a total of (a) 100,000 objects, (b) 1,000,000 objects.

experiment, the number of prefetched objects was varied incrementally and we measured the resulting hit rates, bandwidth consumption, and the *H/B* metrics. We use the equivalent expression $(H/B)_{pref}$, as defined in (12), for the *H/B* metric in our experiments, since Hit_{demand} (given by (7) when no object is prefetched) and BW_{demand} (given by (9) when no object is prefetched) are constants in a given simulation set. For each setting of parameter values, we ran three runs. The values varied by less than 1 percent and, so, we report only the mean values. Figs. 2 and 3 show the *H/B* values for a total of 10^3, 10^4, 10^5, and 10^6 objects. The figures for hit rate and bandwidth for a total of 10^6 objects are given in Fig. 4. (The figures for hit rate and bandwidth for 10^3, 10^4, and 10^5 total objects showed a similar trend.) Note that the values when *Prefetched Objects = 0* correspond to the metrics under on-demand caching (assuming unlimited cache sizes). In particular, $(H/B)_{demand}$ is the value as computed using (13), which also gives the definitions of Hit_{demand} and BW_{demand}.

A little surprisingly, it is found that *Good Fetch* [20] has almost the same performance as *APL* [13]—they appeared to choose the same group of objects. Hence, the simulation results in the figures show the same curves for these two algorithms and we group them as one algorithm (*Good*

Fetch/APL) in the figures (and later in the performance summary in Table 4). The almost identical behavior of *Good Fetch* and *APL* can be explained as follows: The selection criteria of these two algorithms, $1 - (1 - p)^{al}$ and apl, have high conformity, that is, sorting objects with these two criteria, respectively, we get almost the same order. In most of the cases, these two algorithms choose the same subsets of objects and are almost equivalent to each other.

It is seen from the simulations that *H/B-Greedy* prefetching beats *Prefetch by Popularity*, *Good Fetch*, *Prefetch by APL*, and *Prefetch by Lifetime* in terms of *H/B* for any number of prefetched objects, as expected. This is because the object selection in *H/B-Greedy* prefetching is always guided by the objective *H/B*. The gain by *H/B-Greedy* over *Good Fetch (APL)* prefetching is more significant when the number of prefetched objects is relatively small compared to the total number of objects. For example, the greatest relative gains and their occurring positions are approximately:

1. Nineteen percent improvement in *H/B* at the point where about 40 percent of objects are prefetched when the total number of objects is 10^3.
2. Ninety percent improvement at about 20 percent prefetched when the total number is 10^4.
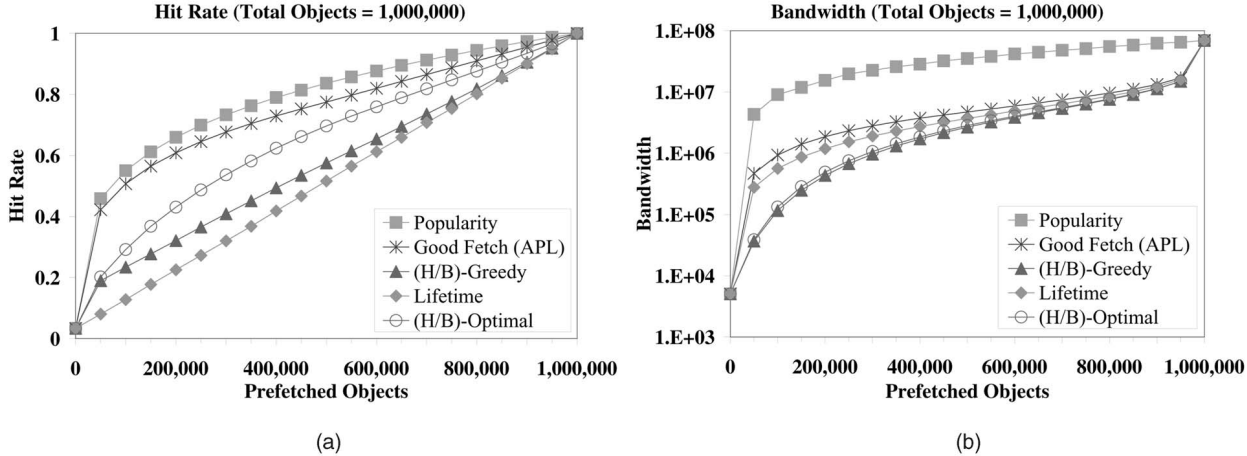
Fig. 4. Performance of 10^6 objects: (a) hit rate and (b) bandwidth.

3. Two hundred thirty-three percent improvement at about 10 percent prefetched when the total number is 10^5.
4. Four hundred seventy-five percent improvement at about 5 percent prefetched when the total number is 10^6.

The curves for *Good Fetch (APL)*, *H/B-Greedy*, and *H/B-Optimal* initially ascend to some highest values as the number of prefetched objects increases and then gradually descend and converge with other curves. The reason for this scenario is that, when the prefetched number is relatively small, the algorithms that use balanced metrics (*Good Fetch (APL)*, *H/B-Greedy*, and *H/B-Optimal*) have better chances to choose those objects that contribute the largest goodness for the balanced metric. Specifically, *Good Fetch (APL)*, *H/B-Greedy*, and *H/B-Optimal* have better chances of choosing those objects that have the largest values of $P_{goodfetch}(APL)$, the largest values of *increase factor*, and the best contribution to improving H/B, respectively. As the number of prefetched objects grows, we have to include those objects that have less contribution. When the number of prefetched objects exceeds some threshold, objects that have a negative effect on the balanced metric begin getting added, making the curves go down as the prefetched number continues to grow. In the meanwhile, the sets of prefetched objects obtained by different algorithms also begin to have a greater number of members in common. Thus, the difference in the performance of various algorithms diminishes as the number of prefetched objects increases.

Figs. 2 and 3 also show that *H/B-Greedy* is a good approximation to *H/B-Optimal*. When the total number is 10^3, the difference in performance is almost indistinguishable—the H/B values differ at most by 0.08 percent. They differ by at most 1.6 percent in the case of a total of 10^4 objects, at most 7 percent in the case of a total of 10^5 objects, and at most 19.2 percent in the case of a total of 10^6 objects. *H/B-Greedy* is seen to perform much closer to the *H/B-Optimal* than any other algorithm in all cases.

From Fig. 4, observe that the hit rates of the algorithms differ by a small factor from each other, roughly by a factor between 1 and 10 for most of the range. Although *Popularity* has the highest hit rate, it requires several orders of

magnitude more bandwidth than the other algorithms. (All algorithms except *Popularity* take into account bandwidth usage.) We also studied the impact of varying the total number of objects from 10^3 to 10^6, on hit rate as well as bandwidth. We observed that the relative performance of the algorithms in terms of hit rate as well as bandwidth stays the same and the graphs are very similar to Fig. 4.

6.2 Impact of Access Pattern, Access Frequency, and Lifetime

Thus far, we have seen the performance of all the algorithms for the simulation parameter settings: $\alpha = 0.75$ [13] for the Zipf's distribution of the object access frequency, maximum object lifetime $l_{max} = 100,000$ seconds, and total access rate $a = 0.01/second$. We also tested the algorithms for various values of these parameters, as shown in Table 3. The notable observations about the impact on H, B, and H/B are also briefly summarized in the table. Our *observation* is that the relative performance of all the algorithms for all the metrics (hit rate, bandwidth, and H/B) remains the same as what we have presented in Section 6.1, over these ranges of the parameters. The detailed results and graphs can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm.

6.3 Evaluation of the *Hit Rate-Greedy* and *Bandwidth-Greedy* Algorithms

In this part, we set up our simulations to investigate the performance of our *Hit Rate-Greedy* and *Bandwidth-Greedy* prefetching algorithms in terms of hit rate and bandwidth, respectively. To show the optimality with regard to their corresponding performance metrics, we compare them with all the algorithms we have studied, including *H/B-Greedy* and *H/B-Optimal*.

In this simulation, a total of 10^6 objects was assumed; object sizes were randomly distributed between 1 and 1M bytes; object lifetimes were randomly distributed between 1 and 100,000 seconds and the total access rate was set to 10.0/second. The results are shown in Fig. 5. *H-Greedy* has a slightly higher hit rate than *Popularity* even though the curves appear close. The bandwidth differences among *H/B-Greedy*, *Lifetime*, *H/B-Optimal*, and *B-Greedy* are very small compared to the other algorithms. Hence, we

TABLE 3
Impact of Varying Access Pattern (Changing α), Lifetime, and Access Rate

| Parameter	Impact on H, B, H/B for all the algorithms
$\alpha = 0.75$ [12] was varied to $\alpha = 0.64$ [2], $\alpha = 0.986$ [1]. (for 10^5 total # of objects)	$\underline{H}$: larger α implies higher on-demand hit rate. With large α, a few top objects account for more accesses than with small α; & those objects with higher access freq. usually have a higher *freshness factor* (Eq. (5)). Thus the overall on-demand hit rate increases with α. The relative performance of algorithms in terms of hit rate, same as in Section 6.1. $\underline{B}$: BW usage (on-demand and for all prefetching algos) not affected. $\underline{H/B}$: Peaks for *H/B-Optimal*, *H/B-Greedy*, *Goodfetch (APL)* come early with larger α; likely due to the relatively bigger diffs in access rate among web objects with larger α.
$l_{max} = 10^5$ sec was varied to 50,000 sec, 1,000,000 sec. (for 10^5 total # of objects)	$\underline{H}$: on-demand hit rate is higher with longer object lifetimes; objects with longer lifetimes have a higher *freshness factor* (Eq. (5)). $\underline{B}$: All algorithms use more BW when lifetimes are shorter; shorter lifetimes imply higher update frequencies to refresh the prefetched copies. $\underline{H/B}$: For all algorithms, curves converge more quickly under shorter lifetimes, due to faster increase of bandwidth usage.
$a = 0.01/$ sec was varied to $a = 0.005/$ sec, $a = 1.0 /$ sec. (for 10^5 total # of objects)	$\underline{H}$: Significant impact. Higher a implies higher on-demand H and higher prefetching H, refer *freshness factor* (Eq. (5)). $\underline{B}$: on-demand BW significantly affected: increases with increasing a, refer Eqs. (8), (9). Shapes of BW curves for all prefetching algos unaffected by a. $\underline{H/B}$: *Popularity* performs worst; with higher a, its difference from others is magnified.
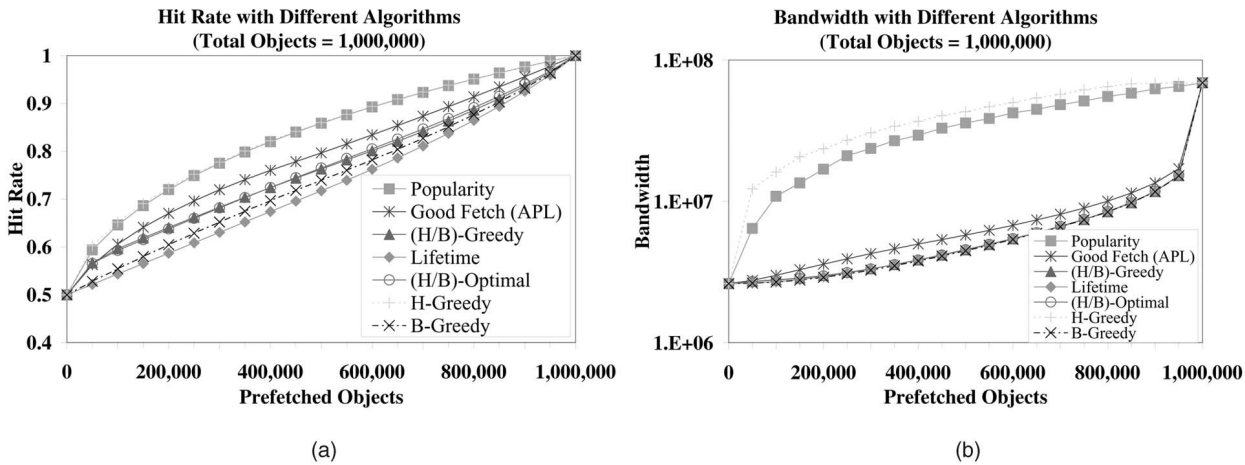
Fig. 5. Greedy algorithms: (a) Comparing the hit rate of *Hit Rate-Greedy*. (b) Comparing the bandwidth of *Bandwidth-Greedy*.

magnify the differences between *B-Greedy* and the remaining three algorithms in Fig. 6. The simulation results show that, as predicted, the *Hit Rate-Greedy* prefetching achieves the highest hit rate and *Bandwidth-Greedy* prefetching results in the lowest bandwidth usage among all the presented algorithms. Note that the hit rate increases are comparable among all the algorithms. However, the bandwidth requirements for *H-Greedy* and *Popularity* are greater by one to two orders of magnitude than those for the other algorithms, for most of the range of the number prefetched. This indicates that *H-Greedy* and *Popularity* should be avoided when seeking to decrease delay without an unreasonable bandwidth increase, such as when using a balanced metric like *H/B*. (Comparing Figs. 4 and 5, the differences in the on-demand hit rate and bandwidth are the result of using a different total access rate a (0.01/sec versus 10.0/sec). The bandwidth when all the objects are prefetched (right end of the graphs) is not affected by a. See (8) and (9).)

6.4 Performance Summary

Based on our simulations, the comparative performance among all the algorithms we studied is summarized in Table 4. The algorithms are ranked according to their performances in terms of H, B, and H/B, respectively. A *"1"* indicates the best performance among all listed algorithms and the larger the number, the worse the performance.

Among the eight algorithms listed in Table 4, *H-Greedy* (also *H-Optimal*) attains the highest hit rate, *B-Greedy* (also *B-Optimal*) consumes the least bandwidth, and *H/B-Optimal* achieves the highest H/B value in *all cases*, as expected. *H/B-Greedy* performs almost as well as *H/B-Optimal* for the H/B metric.

An interesting observation is that, for the various settings of the total number of objects, when the number of prefetched objects is small, *H/B-Greedy* prefetching beats *H/B-Optimal* in terms of hit rate; however, when this number exceeds some value in each case, *H/B-Optimal*
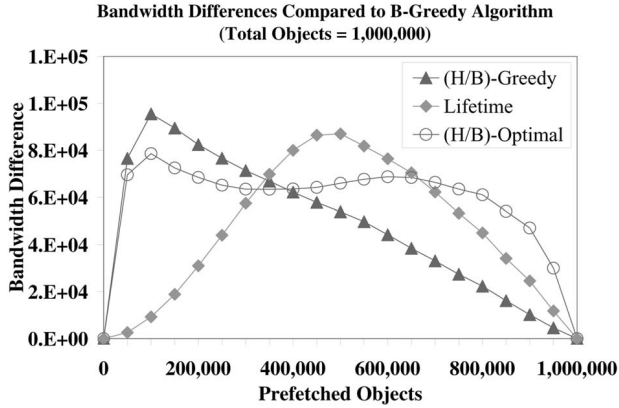
Fig. 6. Bandwidth for *H/B-Greedy*, *Lifetime*, and *H/B-Optimal*, relative to that for *B-Greedy*. (The curves show the difference in bandwidth between the listed algorithms and *B-Greedy*.)

achieves a higher hit rate than *H/B-Greedy* prefetching. Under such behavior, we specify the algorithm that obtains a higher hit rate on most of the occasions as being a better algorithm in the table rankings. Hence, we specify *H/B-Optimal* as being better than *H/B-Greedy* for the hit rate metric. The same rule is used for the other metrics, where necessary.

7 CONCLUSIONS

This paper surveyed several well-accepted Web prefetching algorithms—*Prefetch by Popularity* [17], *Prefetch by Lifetime* [13], *Prefetch by APL* [13], and *Good Fetch* [20]. It then proposed a family of algorithms intended to improve the respective performance metrics under consideration—the hit rate, the bandwidth, or the *H/B* ratio. These greedy algorithms all have linear time complexity and are easy to implement. *H/B-Greedy* prefetching achieves significant improvement of the *H/B* metric over any existing prefetching algorithms; *Hit Rate-Greedy* and *Bandwidth-Greedy* prefetching are optimal in terms of *Hit Rate* and *Bandwidth* as objective metrics, respectively.

As *H/B-Greedy* is not optimal in terms of *H/B* metric, we also proposed an expected linear time randomized algorithm *H/B-Optimal* prefetching that obtains the maximum *H/B* ratio given the number of objects to prefetch. Simulations confirmed our hypothesis that *H/B-Greedy* performs almost as well as *H/B-Optimal*. Compared to the *H/B-Optimal* prefetching algorithm, *H/B-Greedy* is easier to implement and more convenient to adjust to a dynamic environment where the object characteristics change from time to time.

Some directions for future work are as follows:

1. For the *H/B-Greedy* and *H/B-Optimal* prefetching algorithms, the simulations showed that, when the number of prefetched objects is equal to some value n_{max}, the *H/B* metric attains the globally maximum value. This n_{max} is helpful in determining how many objects to prefetch in order to maximize the efficiency of network resources. It is a challenge to determine the n_{max} value for *H/B-Greedy* and *H/B-Optimal*, given the total number of objects.

TABLE 4
Performance Comparison of Algorithms
in Terms of Various Metrics

Algorithm	Hit Rate	Band-width	H/B
Popularity [17]	2	6	5
Good Fetch [20] (*APL* [13])	3	5	3
Lifetime [13]	7	2	4
H/B-Greedy	5	3	2
H-Greedy	1	7	NA
B-Greedy	6	1	NA
H/B-Optimal	4	4	1

A lower value denotes better performance.

2. The principles of Web prefetching studied in this paper have potential applications in the fields of a) wireless networks where the power control and bandwidth control are of special importance [3], [4], [21] and b) P2P networks and networks requiring dynamic Web access, where the data availability and load balancing are more important [9], [19], [14]. The challenge is to adapt and extend the results of this paper to such networks.

APPENDIX

COMPLEXITY ANALYSIS OF THE H/B-OPTIMAL ALGORITHM

In the *H/B-Optimal* algorithm, our adaptation of the function $r_i(x)$ (19) to deal with preselected items corresponding to the on-demand hit rate and on-demand bandwidth can be seen to have $O(n)$ time complexity. This adaptation does not affect the remaining complexity analysis of the Maximum Weighted Average solution of Eppstein and Hirschberg [11]. We rephrase and elaborate on the analysis of [11] for clarity.

Lemma 3. *Let n denote the size of S at the beginning of the outer loop of **H/B-Optimal**. At the end of that iteration, the expected number of objects to be removed or combined is at least $49n/256$.*

Proof. Sort all objects in S based on their $r(A^*)$ values in ascending order. Let the rank of $r_i(A^*)$ be p. As object i is chosen randomly, p is uniformly distributed from 1 to n (see Fig. 7). We now have six cases.

Case 1. If p falls in area $A1$, then at least $k + \frac{n}{32}$ objects have their $r(A^*)$ less than that of object i. After all inner loops are completed, we would have already determined at least k objects whose $r(A^*)$ is less than $r_i(A^*)$. All objects whose $r(A^*)$ is determined to be greater than $r_i(A^*)$ can be combined. Considering that there are at most $n/32$ objects now in Z, the expected number of objects that could be combined in this iteration of the outer loop is at least:

$$\left[n - \frac{1}{2}\left(\left(k + \frac{n}{32} \right) + \left(n - \frac{n}{32} \right) \right) \right] - \frac{n}{32} = \frac{1}{2}\left(n - k - \frac{n}{16} \right).$$
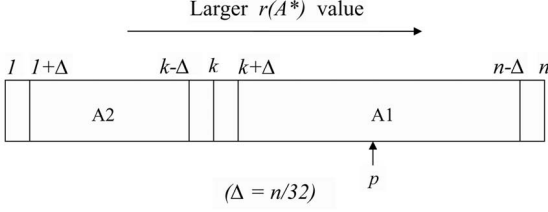
(21)

Fig. 7. The ranking of objects in S in terms of their $r(A^*)$ values.

Case 2. If p falls in area $A2$, then at least $n - (k - \frac{n}{32})$ objects have a higher $r(A^*)$ than object i. After all inner loops are completed, for at most $\frac{n}{32}$ objects (those in Z), we do not know the relative $r(A^*)$ ordering with respect to object i. Thus, we can determine at least $n - k$ objects whose $r(A^*)$ is greater than $r_i(A^*)$, which indicates that all objects whose $r(A^*)$ is less than i can be removed. As at most $\frac{n}{32}$ objects are in set Z, the number of objects that can be removed is $p - \frac{n}{32}$. As p is uniformly distributed in range $[1 + \frac{n}{32}, k - \frac{n}{32} - 1]$, the expected number of objects that can safely be removed is:

$$\frac{1}{2}\left[\left(1 + \frac{n}{32} - \frac{n}{32}\right) + \left(k - \frac{n}{32} - 1 - \frac{n}{32}\right)\right] = \frac{1}{2}\left(k - \frac{n}{16}\right). \quad (22)$$

Case 3. If p falls within the range $[1, 1 + \frac{n}{32})$, then the number of objects whose $r(A^*)$ is greater than that of object i is at least $n - \frac{n}{32}$ and the number of objects whose $r(A^*)$ is less than $r_i(A^*)$ is at most $\frac{n}{32}$. As there are at most $\frac{n}{32}$ objects in Z, it is not clear how many objects can be removed. So, we can only take the worst case: No objects can be removed.

Case 4. If p falls in the range $(n - \frac{n}{32}, n]$, then at most $\frac{n}{32}$ objects have their $r(A^*)$ greater than object i. $|X|$ cannot be determined, so, in the worst case, no objects can be combined.

Case 5. If p falls in the range $[k, k + \frac{n}{32})$, then the number of objects whose $r(A^*)$ is less than $r_i(A^*)$ is at least k and the number of objects whose $r(A^*)$ is greater than $r_i(A^*)$ is at least $n - k - \frac{n}{32}$ and at most $n - k$. In this case, due to the existence of Z, we are unable to know the fact "at least k objects whose $r(A^*)$ is less than $r_i(A^*)$" and, thus, we do not know whether $|Y| + |E| \geq k$ is true. So, we cannot combine any objects.

Case 6. If p falls in $[k - \frac{n}{32}, k)$, then the number of objects whose $r(A^*)$ is less than $r_i(A^*)$ is at least $k - \frac{n}{32}$ and at most k. The number of objects whose $r(A^*)$ is larger than $r_i(A^*)$ is at least $n - k$. However, since there exists Z whose size is at most $\frac{n}{32}$, we do not know this fact: $|X| + |E| < n - k$. Thus, we are unable to perform the removal operation.

Note that only Cases 1 and 2 can result in a reduced number of objects. For Case 1, the average number of objects to be combined is $\frac{1}{2}(n - k - \frac{n}{16})$ and the probability of occurrence is $(n - k - \frac{n}{16})/n$. For Case 2, the average number of objects to be removed is $\frac{1}{2}(k - \frac{n}{16})$ and the probability of occurrence is $(k - \frac{n}{16})/n$. Thus, the expected reduction in the number of objects is:

$$\frac{1}{2}\left(n - k - \frac{n}{16}\right)\left(n - k - \frac{n}{16}\right)/n + \frac{1}{2}\left(k - \frac{n}{16}\right)\left(k - \frac{n}{16}\right)/n$$
$$= \frac{1}{2n}\left[\left(n - k - \frac{n}{16}\right)^2 + \left(k - \frac{n}{16}\right)^2\right]. \quad (23)$$

This is minimized (worst case) when $k = \frac{n}{2}$ and, in the worst case, computes to $\frac{49}{256}n$. $\square$

Lemma 4. *Let n denote the size of S at the beginning of the outer loop of **H/B-Optimal**. In a single iteration of the outer loop, the expected time complexity is $O(n)$.*

Proof. The for-loop (lines 11-17) takes n time. As the exit condition for the inner loop is set to $|Z| \leq \frac{n}{32}$, the inner loop can be iterated at most five times within an outer loop. Initially, let $|Z| = n$.

Note that the call to **median**(i, Z) (line 20) takes time $O(|Z|)$, the call to $\mathbf{F}(A, S, k)$ (line 21) takes time $O(n)$. In addition, the call to **compare**$(i, j, left, right)$ (line 30) takes constant time, so the for-loop (lines 29-33) takes $O(|Z|)$ time. The "removing" or "combining" operations (lines 35-43) take time proportional to the number of objects removed or combined, which is expected to be $O(49n/256)$.

To summarize, a single iteration of the inner loop takes time $O(n)$ and so do five iterations. The total time for an outer loop is thus $O(n)$. $\square$

From Lemmas 3 and 4, we can derive the recurrence relation of the expected time complexity for **H/B-Optimal**:

$$T(n) = O(n) + T\left(\frac{207}{256}n\right). \quad (24)$$

In this equation, $O(n)$ is the expected operation time within one iteration of the outer loop and the expected number of objects left in S decreases to $\frac{207}{256}n$ after an outer loop iteration. This resolves to a linear expected time complexity.

ACKNOWLEDGMENTS

REFERENCES

[1] A. Bestavros, C.R. Cunha, and M.E. Crovella, "Characteristics of WWW Client-Based Traces," technical report, Boston Univ., July 1995.

[2] L. Breslau, P. Cao, L. Fan, G. Philips, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. IEEE Infocom,* pp. 126-134, 1999.

[3] G. Cao, L. Yin, and C.R. Das, "Cooperative Cache-Based Data Access in Ad Hoc Networks," *Computer,* vol. 37, no. 2, pp. 32-39, Feb. 2004.

[4] G. Cao, "Proactive Power-Aware Cache Management for Mobile Computing Systems," *IEEE Trans. Computers,* vol. 51, no. 6, pp. 608-621, June 2002.

[5] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *Proc. USENIX Symp. Internet Technologies and Systems,* pp. 193-206, 1997.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Selection in Expected Linear Time," *Introduction to Algorithms,* second ed., pp. 185-189, 2001.

[7] M.E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Trans. Networking,* vol. 5, no. 6, pp. 835-846, 1997.

[8] M. Crovella and P. Barford, "The Network Effects of Prefetching," *Proc. IEEE Infocom,* pp. 1232-1239, 1998.

[9] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P.J. Shenoy, "Adaptive Push-Pull: Disseminating Dynamic Web Data," *Proc. WWW Conf. 2001,* pp. 265-274, 2001.

[10] F. Douglis, A. Feldmann, B. Krishnamurthy, and J.C. Mogul, "Rate of Change and Other Metrics: A Live Study of the World Wide Web," *Proc. USENIX Symp. Internet Technologies and Systems,* pp. 147-158, 1997.

[11] D. Eppstein and D.S. Hirschberg, "Choosing Subsets with Maximum Weighted Average," *J. Algorithms,* vol. 24, no. 1, pp. 177-193, 1997.

[12] S. Glassman, "A Caching Relay for the World Wide Web," *Computer Networks & ISDN Systems,* vol. 27, no. 2, pp. 165-173, 1994.

[13] Y. Jiang, M.Y. Wu, and W. Shu, "Web Prefetching: Cost, Benefits and Performance," *Proc. 11th World Wide Web Conf. (WWW),* 2002.

[14] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin, "NPS: A Non-Interfering Deployable Web Prefetching System," *Proc. USENIX Symp. Internet Technologies and Systems,* 2003.

[15] T.M. Kroeger, D.E. Long, and J. Mogul, "Exploring the Bounds of Web Latency Reduction from Caching and Prefetching," *Proc. USENIX Symp. Internet Technologies and Systems,* pp. 13-22, 1997.

[16] C. Liu and P. Cao, "Maintaining Strong Cache Consistency in the World-Wide Web," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 12-21, 1997.

[17] E.P. Markatos and C.E. Chironaki, "A Top 10 Approach for Prefetching the Web," *Proc. INET '98: Internet Global Summit,* July 1998.

[18] N. Nishikawa, T. Hosokawa, Y. Mori, K. Yoshidab, and H. Tsujia, "Memory Based Architecture with Distributed WWW Caching Proxy," *Computer Networks,* vol. 30, nos. 1-7, pp. 205-214, 1998.

[19] S. Shah, K. Ramamritham, and P.J. Shenoy, "Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 7, pp. 799-812, July 2004.

[20] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin, "The Potential Costs and Benefits of Long-Term Prefetching," *Computer Comm.,* vol. 25, no. 4, pp. 367-375, 2002.

[21] L. Yin, G. Cao, C. Das, and A. Ashraf, "Power-Aware Prefetch in Mobile Environments," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 571-578, 2002.

Bin Wu received the BS and MS degrees from the Department of Energy Engineering at Zhejiang University, China, in 1992 and 1995, respectively. He received the MS degree in computer science from the University of Illinois at Chicago in 2002. He is now working toward the PhD degree in computer science at the University of Illinois at Chicago. His research intersts include distributed computing, peer-to-peer networks, networks protocols and algorithms, and Web-based healthcare systems.

Ajay D. Kshemkalyani received the PhD degree in computer and information science from The Ohio State University in 1991 and the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987. His research interests are in computer networks, distributed computing, algorithms, and concurrent systems. He has been an associate professor at the University of Illinois at Chicago since 2000, before which he spent several years at IBM Research Triangle Park working on various aspects of computer networks. He is a member of the ACM and a senior member of the IEEE. In 1999, he received the US National Science Foundation's CAREER Award. He is currently on the editorial board of the Elsevier journal, *Computer Networks*.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.