# Resource-based Caching for Web Servers*

**Renu Tewari, Harrick M. Vin, Asit Dan† and Dinkar Sitaram†[†]**

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
E-mail: {tewari, vin}@cs.utexas.edu
Phone : (512) 471-9732, Fax : (512) 471-8885

† IBM Research Division
T.J. Watson Research Center
Hawthorne, NY 10532
E-mail: {asit}@watson.ibm.com
Phone : (914) 784-7953

## ABSTRACT

The WWW employs a hierarchical data dissemination architecture in which hyper-media objects stored at a remote server are served to clients across the internet, and cached on disks at intermediate proxy servers. One of the objectives of web caching algorithms is to maximize the data transferred from the proxy servers or cache hierarchies. Current web caching algorithms are designed only for text and image data. Recent studies predict that within the next five years more than half the objects stored at web servers will contain continuous media data. To support these trends, the next generation proxy cache algorithms will need to handle multiple data types, each with different cache resource usage, for a cache limited by both bandwidth and space.

In this paper, we present a resource-based caching (RBC) algorithm that manages the heterogeneous requirements of multiple data types. The RBC algorithm (i) characterizes each object by its resource requirement and a caching gain, (ii) dynamically selects the granularity of the entity to be cached that minimally uses the limited cache resource (i.e., bandwidth or space), and (iii) if required, replaces the cached entities based on their cache resource usage and caching gain. We have performed extensive simulations to evaluate our caching algorithm and present simulation results that show that RBC outperforms other known caching algorithms.

**Keywords:** web caching, multimedia caching, bandwidth constrained cache, proxy server cache

## 1. INTRODUCTION

### 1.1. Motivation

The World Wide Web (WWW) has emerged as the most widely used tool for the access and dissemination of commercial, educational, news, and entertainment information on the internet. The accesses across the web are characterized by a simple request-response paradigm. These accesses are non-uniform with frequently changing access patterns, and result in server *hot-spots* (e.g., the Mars Pathfinder web sites recorded more than six million hits a day). Sudden increases in popularity cause server overload, network congestion, and an increase in the response time observed by the client. Caching is a technique used for dynamically adapting to changes in server popularity. Web objects are cached at client sites, at proxy servers shared by multiple clients,[1] or in cache hierarchies (e.g., SQUID,[2] Harvest[3] etc.). Caching closer to the client reduces the number of accesses from the remote server, thereby lowering network traffic, server load, and client observed response times.

Since web objects are relatively large compared to file blocks, they are cached on disk instead of memory. Disk caches differ from memory caches in that they are much larger in size but have much lower bandwidth. For the current web traffic, consisting of mostly text and image data, a disk cache is equivalent to a large but slow memory cache. Thus, the algorithms currently used for web caching are extensions of traditional memory caching algorithms.

It is conjectured, that by the year 2005, more than 50% of the information available over the internet will contain continuous media (CM) data.[4] Currently continuous media data is accessed similar to text and images using the download-and-play mode. With a download-and-play mode of access, data is transferred completely to the client site before display. Due to the large sizes of CM objects, this results in large space usage and wait time at the client. Streaming mode of access addresses this problem by enabling the client to initiate display of data with only a small start-up latency, without waiting for the entire object

*This research was supported in part by an IBM Faculty Development Award, Intel, the National Science Foundation (Research Initiation Award CCR-9409666 and CAREER award CCR-9624757), NASA, Mitsubishi Electric Research Laboratories (MERL), and Sun Microsystems Inc.

†the author is currently at Novell Corporation, Bangalore, India, e-mail: sdinakar@novell.com.

to be downloaded (e.g., Vosaic, RealAudio, Macromedia, Progressive Networks, VDO, Vivo, VXtreme, Xing).[5] To support the real-time requirements for streaming, web caches need to reserve cache bandwidth for each continuous media object. Thus, continuous media objects stored in the cache occupy both space and bandwidth. Since disks have limited bandwidth (disk bandwidths are around 2-7 MB/sec compared to memory bandwidths of 80-100 MB/sec), only a limited number of continuous media objects can be guaranteed real-time delivery. Current caching algorithms do not consider the space and bandwidth occupancy of the different types of objects. The main focus of this work is to develop cache replacement algorithms that handle different data types (continuous and non-continuous) for disk based caching in the world wide web.

### 1.2. Related Work on Caching Algorithms

To review the work on caching algorithms, we first discuss the traditional memory caching of fixed size pages. Next, we describe previous work on caching continuous media objects in memory. Finally, we describe proposed extensions of memory caching algorithms for disk caching to accommodate variable sized web documents (text and images only) and motivate the need for new algorithms to handle multiple data types in the WWW.

Traditionally, to reduce disk accesses and improve program performance, a subset of pages most likely to be accessed again are cached in memory.[6–10] To maximize the number of accesses served from cache, when the cache capacity is exceeded, the caching algorithm replaces pages to be accessed furthest in time. To estimate the time of the next access to a page, these algorithms use heuristics based on the recency or frequency of access. Recency-based algorithms exploit the *locality of reference* inherent in programs. LRU (Least Recently Used) is the most commonly used recency based algorithm. Frequency-based algorithms are suited for skewed access patterns in which a large fraction of the accesses go to a disproportionately small set of *hot* objects.[8] LFU (Least Frequently Used), which maintains a reference count for each cached page, is the most commonly used algorithm.[11] Frequency and recency based algorithms form the two ends of a spectrum of caching algorithms. Several algorithms, such as LRU-k and LRFU, that try to balance both recency and frequency have been proposed.[12,13] In addition to these algorithms, several other algorithms that utilize the knowledge of user access patterns have been developed.[11,14,10]

Caching for CM data has been studied in the context of memory caches.[15–18] Since CM data have periodic and sequential accesses, the buffer allocated to a block can be reassigned immediately after use. However, if another user is accessing the same data, within a short time interval, the block can be retained in the buffer for the later user. Examples of caching algorithms that are optimized for CM accesses include *interval caching*[16,17] and *distance caching*.[18] These algorithms exploit the sequentiality of CM accesses and cache *intervals* formed by pairs of consecutive accesses to the same CM object. The two requests that form such a consecutive pair are called the *writer* (or preceding request) and the *reader* (or following request). For instance, in Figure 1, in the interval $b_1$ consisting of the request pair $[S_1, S_2]$, $S_1$ is the writer and $S_2$ the reader. Whereas $S_1$ accesses data blocks from the server and retains them in the cache buffers, $S_2$ reads the blocks from the cache and frees the buffers.

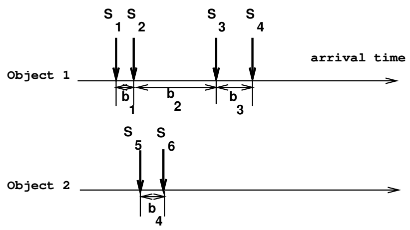



**Figure 1.** Interval Caching for Continuous Media

Observe that all of these algorithms for memory caches assume that the system is limited by disk performance. However, in the context of the WWW, where a request involves retrieving objects from remote servers, the network becomes a bottleneck. Such an environment differs from memory caching in that: (1) the cached object have varying sizes (instead of fixed size pages), (2) the cached data could be shared among clients, and (3) objects belong to different data types. Current disk-based web caching algorithms are adaptations of traditional memory caching.[19–21] For example, commonly used access attributes are: (1) time of last reference among a set of objects (Squid Cache),[2] (2) the time-of-day of access,[22] (3) the size of the object (or its logarithm),[21] (4) the number of accesses to an object over a time interval as the primary key and the time for last access as the secondary key (e.g., the HyperG),[23] (5) the logarithm of object sizes as the primary key and the time for last access as the secondary key (LRU-MIN)[20]), (6) time of last access per object group where the logarithm of object sizes is used to group objects,[24] and (7) the estimate of the average page loading delay and retrieval time.[25,26]

Since most of the data accessed on the web today contains text and static images, the above algorithms seem adequate. As streaming of CM data gains popularity, web caches will need to reserve cache bandwidth for each object. Since disks have limited bandwidth, memory caching algorithms for CM objects cannot be directly used for web caches. For example, the interval caching policy incurs the overhead of writing a block thereby reducing the effective bandwidth to 50%, while the frequency-based caching of entire objects is restricted to caching a few objects due to the large size of CM objects. Given that disk caches are limited by both space and bandwidth, in order to maximize the hit ratio, web cache management algorithms have to efficiently manage both these resources.

In summary, to overcome the dynamically occurring overloads at web servers, objects are cached on disks at intermediate proxy servers. Since disks are limited by both bandwidth and space, cache management algorithms will be required to efficiently utilize both cache space and bandwidth. None of the existing caching algorithms achieve all of these objectives.

### 1.3. Research Contributions of This Paper

In this paper, we present a resource-based caching (RBC) algorithm for web proxies and servers. The resource-based caching algorithm characterizes each object by the bandwidth and space requirement pair and a caching gain. The cache is modeled as a two-constraint knapsack. The algorithm describes heuristics to convert the two-constraint knapsack model to multiple single constraint models. The algorithm dynamically selects the granularity of the entity to be cached that minimally uses the limited cache resource (i.e., bandwidth or space). The formulation of the algorithm enables it to be generalized for handling different data types and cache performance objectives (hit ratio, byte hit ratio etc.). We present extensive simulation results to evaluate our caching algorithm, and show that RBC outperforms other known caching algorithms.

The rest of this paper is organized as follows. In Section 2, we describe our resource-based caching algorithm. Results of our simulations are presented in Section 3, and finally, Section 4 summarizes our results.

## 2. RESOURCE-BASED CACHING (RBC)

Consider the hierarchical information delivery architecture of the WWW, with a remote server storing continuous (i.e., audio and video) and non-continuous (e.g., textual and numeric data, imagery, etc.) media objects. The resource requirement of each object $O_i$, $i \in [1..N]$ is defined as the tuple $\langle s_i, b_i \rangle$, where $s_i$ and $b_i$, respectively, denote the size and access bandwidth. Let an intermediate server maintain a cache of size $S$ and bandwidth $B$. Each atomic unit of an object in cache is called an entity. The granularity of the entity (e.g., entire file, block, group of blocks) is not fixed but depends on the type of object. CM entities are further classified into *dynamic* or *static* entities. For dynamic entities (e.g., intervals), the contents of the entity changes with time, while for static entities (e.g, entire objects) it remains the same. Let $\mathcal{E} = \{E_1, E_2, ..., E_n\}$ denote the set of entities in the cache, and let $\widehat{s}_i$ and $\widehat{b}_i$, respectively denote the cache space and bandwidth occupied by entity $E_i$. Each CM object requires bandwidth reservation, while non-CM objects are served on a best-effort basis. In order to avoid starvation of non-CM objects, a fraction of the cache bandwidth is pre-allocated for the entire class of non-CM objects (the remaining cache bandwidth is shared by both CM and non-CM objects).

In what follows, we describe the main components of the RBC algorithm, namely: defining the characteristics of an entity, selecting the desired entity to cache, and determining which, if any, entity to replace.

### 2.1. Entity Characteristics

The characteristics of an entity are defined by its: (i) granularity, (ii) resource requirement, and (iii) caching gain.

#### 2.1.1. Granularity of Cached Entity

Depending on its type, an object can be cached partially (e.g., block, group of blocks) or in its entirety.

- *Continuous media objects:* Due to the sequential and periodic nature of access, a request for a CM object can be served from the cache by maintaining a set of consecutive blocks forming an interval between two successive requests to the same object. An interval consists of a single reader and writer. In general, if multiple users access the same object concurrently, a set of adjacent intervals could be grouped together to form *runs*, with a single writer and multiple readers. This introduces a range of possible entities that can be cached; from an interval to single-writer multiple-reader runs, to the entire object. Figure 2 shows the runs $r_1$ to $r_4$, that are formed for a set of users accessing the same data object. The run $r_1$ is a simple interval of writer $S_1$ and reader $S_2$, while run $r_4$ consists with writer $S_1$ and readers $S_2, S_3, S_4, S_5$. Since run caching not only amortizes the write overhead over multiple readers but also uses less space than the entire object, it is useful when both space and bandwidth of the cache are equally limited.

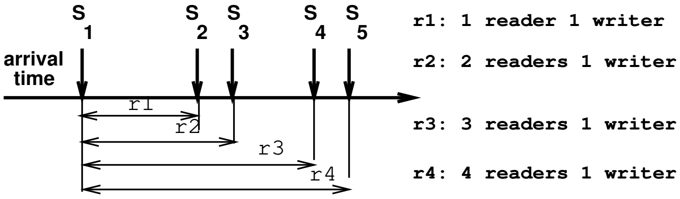



**Figure 2.** Run Caching for Continuous Media

- *Non-continuous media objects:* Non-CM objects in the WWW are normally accessed, and hence, cached in their entirety[‡].

### 2.1.2. Estimating Resource Requirement

The resource requirement of an entity is given by the cache space and bandwidth occupied by it.

- *Continuous media objects:* If an entity $E_i$ consists of the entire object $O_j$, then its cache space occupancy equals the object size, i.e., $\widehat{s}_i = s_j$. On the other hand, if $E_i$ is a dynamic entity (interval or run) of duration $t_i$, with an access bandwidth $b_i$, then $\widehat{s}_i = \min\{s_j, t_i * b_i\}$.

  The bandwidth occupancy of an entity is the cumulative access bandwidth required by its concurrent readers and writers. An interval has exactly one reader and one writer; a run, on the other hand, has multiple readers and one writer. If the entity is an entire object, then it could have multiple concurrent readers and at most one writer (if the object is being concurrently written to cache). Thus, for entity $E_i$ with access bandwidth $b_i$ per user, $r_i$ concurrent readers and $w_i$ ($w_i = 0$ *or* $1$) writers, the bandwidth occupancy is $\widehat{b}_i = (r_i + w_i) * b_i$.

- *Non-continuous media objects:* Since non-CM objects are cached in their entirety, if entity $E_i$, stores object $O_j$, then its space occupancy is given by $\widehat{s}_i = s_j$. Given that non-CM objects are served on a best-effort basis, for all non-CM objects the entity's bandwidth occupancy $\widehat{b}_i = 0$.

### 2.1.3. Caching Gain

There are many metrics for measuring the overall performance improvement by caching. Two such metrics are: (i) *the Hit Ratio (HR)* which denotes the ratio of number of accesses from cache to the total number of accesses, and (ii) the *Byte Hit Ratio (BHR)* which equals the ratio of the total number of bytes accessed from cache to the total number of bytes accessed. The caching gain of an entity depends on the metric used to measure cache performance.

- *Continuous media entities:* To maximize BHR, the caching gain $g_i$ of a CM entity $E_i$ is defined as the total bandwidth saved (i.e., the bytes transferred per second). For an entity with access bandwidth $b_i$ and an estimated number of concurrent readers $\widehat{r}_i$, the gain is given by

$$g_i = \widehat{r}_i b_i \tag{1}$$

  For a dynamic entity (interval or a run), $\widehat{r}_i$ is known at the time the run or interval is formed, and is equal to the number of concurrent readers. For a static entity (i.e., the entire object), on the other hand, the estimated number of concurrent readers is computed from the interarrival time between requests (also referred to as the time-to-reaccess or TTR). Thus, if a static entity has access probability $p_i$ and $\lambda$ is the request arrival rate in the system, then the interarrival time between requests is $\frac{1}{\lambda p_i}$. Assuming that the entity consists of the entire object, each user will be serviced from the cache for a duration of $\widehat{s}_i / b_i$. The estimated number of concurrent readers is given by the ratio of the time duration of the object to the interarrival time between requests. Hence,

$$\widehat{r}_i = \frac{\widehat{s}_i / b_i}{1/(\lambda p_i)} = \lambda p_i \widehat{s}_i / b_i \tag{2}$$

  To maximize HR, the gain of a CM entity is defined as the number of requests per second, which is the inverse of the interarrival time between requests. Hence,

$$g_i = \widehat{r}_i b_i / \widehat{s}_i = \lambda p_i \tag{3}$$

[‡]HTTP/1.0 protocol does not support random block accesses.

- *Non-continuous media entities:* For a non-CM entity $E_i$, the caching gain for maximizing BHR is defined as the bytes transferred per second, i.e., the ratio of the total number of bytes transferred to the interarrival time between requests. The gain is given by

$$g_i = \frac{\widehat{s_i}}{1/\lambda p_i} = \lambda p_i \widehat{s_i} \tag{4}$$

where $p_i$ is the access probability and $1/\lambda p_i$ is the interarrival time between requests. For maximizing HR, the gain of a non-CM entity is defined as the number of requests per second, i.e.,

$$g_i = \lambda p_i \tag{5}$$

Observe that the value of the caching gain depends on the estimate of the time-to-reaccess or the access probability of an object. Since $TTR = \frac{1}{\lambda p_i}$, determining either is sufficient. The long term access probability could be known apriori without knowing the exact access sequence (e.g., VOD applications with slow changing access patterns), or estimated by maintaining access statistics. Some techniques used to determine the access probability or TTR are outlined below.

- *Reference Count (REFCNT)* — In this technique, the reference count is used to estimate the access probability (e.g., LFU, FBR use reference count[11]). However, the reference count measure does not capture the recency of objects.
- *Time of Access (REFTIME)* — In this technique, the history of access times is maintained which is used to estimate the interarrival time.[13] The LRU technique uses only the time of the last access, while the LRU-k uses the time of the $k^{th}$-to-last access. A typical value selected for k is 2 (i.e., LRU-2). One of the drawbacks of this technique is that it requires the history of the last k accesses to be maintained per object, which imposes significant overhead for large values of k. Also, selecting a large value of k to favor frequency of access ignores the recency of the previous k-1 accesses. A small k value favors recency of access ignoring the long term frequency.

To overcome the drawbacks of the above mentioned techniques, we designed the WLRU-n technique in which the mean time-to-reaccess (MTTR) is computed as the weighted sum of the interarrival time between the previous $i^{th}$ and the $i-1^{th}$ access. Let the access times of the last $n$ accesses be $t_n, t_{n-1}...t_1$, where $t_i$ is time of the last $i^{th}$ access (i.e, $t_1$ is the most recent access time and $t_0$ is the current time). Let the weighting function, $W(i)$, be decreasing, $W(i) < W(i-1)$. Then $MTTR = \Sigma^{i \geq 0}(t_i - t_{i+1}) * W(i)$. Observer that for $W(i) = \alpha W(i-1), \alpha \leq 1$ and $W(0) = 1 - \alpha$, the history information of the previous access times need not be maintained. Thus MTTR at time $t_0$ is, $MTTR(t_0) = (1-\alpha)(t_0 - t_1) + \alpha MTTR(t_1)$. The averaging factor $\alpha$ can be tuned to bias for or against recency.

Having defined the set of entities that can be cached and their characteristics, we now present details of the RBC algorithm which consists of: (i) selecting the granularity of the cached entity, and (ii) determining which, if any, of the cached entities to replace.

## 2.2. Selection of Cached Entity Granularity

When a request arrives for an object not in cache, the RBC algorithm selects the granularity of the entity to cache. This selection depends on the state of the cache and the entity characteristics.

The *state of the cache* is defined as a pair $(\mathcal{U}_s, \mathcal{U}_b)$, where $\mathcal{U}_s$ and $\mathcal{U}_b$ denote the utilization of the cache space and bandwidth, respectively[§]. Specifically,

$$\mathcal{U}_s = \frac{1}{S} \sum_{E_i \in \mathcal{E}} \widehat{s}_i; \qquad \mathcal{U}_b = \frac{1}{B} \sum_{E_i \in \mathcal{E}} \widehat{b}_i; \quad 0 \leq \mathcal{U}_s, \mathcal{U}_b \leq 1.$$

To ensure that both the cache space and bandwidth are effectively utilized, RBC selects the granularity of the entity (entire object, interval, etc.,) so as to minimally use the resource that is currently over-utilized. This prevents either space or bandwidth to be saturated while the other remains under-utilized. Observe that if the state of the cache is represented as a point within a unit square where the x- and y-axis denote the space and the bandwidth utilization, respectively (see Figure 3), then all the points close to the diagonal capture the state where $\mathcal{U}_s \approx \mathcal{U}_b$. The selection methodology is described below:

[§]The cache bandwidth $B$ in the equation represents the total bandwidth available for CM objects.

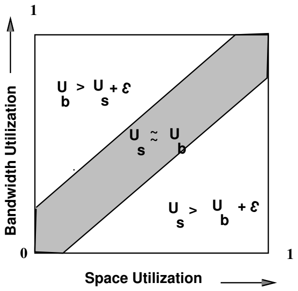



**Figure 3.** Cache State

- If the bandwidth utilization of the cache is greater than the space utilization (within an approximation bound), i.e., $\mathcal{U}_b > \mathcal{U}_s + \epsilon$, then to minimize the wasted bandwidth, the entity with the lowest write bandwidth overhead is selected. The write overhead for entity $E_i$ is $\frac{w_i}{r_i + w_i}$, where $r_i$ is the number current readers and $w_i$ is the number of writers. Note that for an interval, due to the continuous discarding of blocks by the reader, the write overhead is 50%.

- If the space utilization of the cache is greater than the bandwidth utilization, i.e., $\mathcal{U}_s > \mathcal{U}_b + \epsilon$, then to minimize the space usage, the entity with the lowest space usage $\widehat{s_i}$, is selected.

- If the bandwidth and space utilization of the cache are approximately equal, i.e., $\mathcal{U}_s - \epsilon < \mathcal{U}_b < \mathcal{U}_s + \epsilon$, then both the space and wasted bandwidth overheads are minimized. The entity selected has the largest effective bandwidth to space overheard ratio. The effective bandwidth is the bandwidth that is available for reads. The entity with the largest ratio has the lowest write bandwidth overhead per unit space used. The ratio is given by $\frac{1 - \frac{w_i}{r_i + w_i}}{\widehat{s_i}}$.

### 2.3. Cached Entity Replacement Algorithm

Having determined the entity to cache, RBC determines which, if any, of the entities present in the cache should be replaced. To identify the candidates for removal, RBC orders the accessed entity and those present in the cache by *goodness*, which depends on (i) the caching gain $g_i$, (ii) the resource usage $\langle \widehat{s_i}$ and $\widehat{b_i} \rangle$ and, (iii) the cache state $(\mathcal{U}_s, \mathcal{U}_b)$. All the entities in the cache are sorted in ascending order of their goodness values. The selection of entities for removal proceeds from the one with the worst goodness value till sufficient resources have been released to accommodate the new entity. To resolve a tie among entities with equal goodness values, a *best-fit* selection is made based on the cache resource used by each entity.

The computation of the goodness value depends on whether the cache, with respect to the entity $E$, is: (i) *unconstrained*, (ii) *space constrained*, (iii) *bandwidth constrained*, or (iv) *bandwidth and space constrained*. For simplicity of description, we assume that the cache performance objective is to maximize the byte-hit-ratio. Similar arguments and analysis can be carried out for maximizing hit ratio and other performance objectives.

1. **Unconstrained**: The cache is said to be unconstrained if the cache space and bandwidth available exceed the requirements of the entity to be cached. In this scenario, the entity is placed in cache and the state of the cache is appropriately updated.

2. **Space Constrained**: The cache is said to be space constrained if the available bandwidth is sufficient to accommodate entity $E$ (i.e., $\widehat{b} \leq (1 - \mathcal{U}_b) * B$) but the available space is not (i.e., $\widehat{s} > (1 - \mathcal{U}_s) * S$). In this case, each entity $E_i$ is assigned a *goodness* value $\mathcal{G}_i$, which represents the gain per unit space used, or the *gain density*. Thus, for caching gain $g_i$ and space used $\widehat{s_i}$ the goodness is given by
$$\mathcal{G}_i = g_i / \widehat{s_i} \tag{6}$$
Recall that for the objective of maximizing BHR, the gain $g_i = \widehat{r_i} * b_i$, which for entire objects reduces to $g_i = \lambda p_i \widehat{s_i}$ (Equation 1, 2). Thus, the goodness equation reduces to $\mathcal{G}_i = \lambda p_i = 1/TTR$. The goodness value maps to the inverse of the time-to-reaccess the object (similar substitutions can be made for intervals). Observe that this goodness value is identical to the metric used for traditional space constrained processor caches that replace the data block that has the largest time-to-reaccess. The cache replacement algorithm in the space constrained case is described in detail in Figure 4(a).

**space _constrained_policy** (new entity $E_i$)
$\forall$ entities $E_k$ in cache
  if ($\mathcal{G}_k < \mathcal{G}_i$)
    removelist = removelist '+' $E_k$
  order cachelist in ascending $\mathcal{G}$ values; $i < k, \mathcal{G}_i \leq \mathcal{G}_k$
while ($space\_free < \widehat{s_i}$)
    $space\_free = space\_free + \widehat{s_k}$;
    mark $E_k$ for deletion; k++
if ($space\_free \geq \widehat{s_i}$)
  second pass release any extra space marked
  delete marked entities
  return success
else return failure

(a) Space Constrained

**bw _space_constrained_policy** (new entity $E_i$)
$\forall$ entities $E_k$ in cache
  if ($g_k/\widehat{b_k} < g_i/\widehat{b_i}$) bwlist = bwlist '+' $E_k$
  if ($g_k/\widehat{s_k} < g_i/\widehat{s_i}$) splist = splist '+' $E_k$
  order splist and bwlist by increasing goodness value
while ($bw\_free < \widehat{b_i}$ & $space\_free < \widehat{s_i}$ )
    if (empty splist & bwlist )
      break;
    if ( $bw\_free/\widehat{b_i} < space\_free/\widehat{s_i}$ )
      select k from bwlist; remove k from lists
    else
        select k from splist; remove k from lists
    mark k for deletion; update bw_free and space_free
if ($bw\_free \geq \widehat{b_i}$ & $space\_free \geq \widehat{s_i}$)
  second pass release any extra bandwidth or space marked
  delete marked entries
  return success
else return failure

(b) Bandwidth and Space Constrained

**Figure 4.** Cache Replacement Policy

3. **Bandwidth constrained**: The cache is said to be bandwidth constrained if the available space is sufficient to accommodate the entity $E$ (i.e., $\widehat{s} \leq (1 - \mathcal{U}_s) * S$) but the available bandwidth is not (i.e., $\widehat{b} > (1 - \mathcal{U}_b) * B$). We define *goodness* value of a continuous media entities $E_i$, in this case to be the gain per unit cache bandwidth used, that is

$$\mathcal{G}_i = g_i/\widehat{b_i}$$

Recall that to maximize BHR, $g_i = r_i b_i$ and $\widehat{b_i} = (r_i + w_i) * b_i$, thus $\mathcal{G}_i = \frac{r_i}{r_i + w_i}$. Thus for maximizing BHR, the goodness ordering maps to the ratio of total bandwidth saved to the cache bandwidth used. A ratio of 1, indicates that an entity uses all the cache bandwidth for reading from the cache and no bandwidth is wasted in writing to the cache. This is the case when the object is not being concurrently written to the cache. The goodness value measures to what degree the write overhead is amortized over the gain from cache reads.

4. **Bandwidth and Space Constrained**: The cache is said to be bandwidth and space constrained if neither the available space nor the available bandwidth is sufficient to accommodate entity $E_i$ (i.e., $\widehat{s_i} > (1 - \mathcal{U}_s) * S$ and $\widehat{b_i} > (1 - \mathcal{U}_b) * B$). In this case the cached entities are ordered into two lists. An *s-list*, which orders entities by their gain per unit space (i.e, $\mathcal{G}_i = g_i/\widehat{s_i}$), and a *b-list* that orders by the gain per unit bandwidth (i.e, $\mathcal{G}_i = g_i/\widehat{b_i}$). The algorithm removes entities from either of the two lists until enough resources are released for the new entity. Let $S_{free}$ and $B_{free}$, respectively, represent the currently available space and bandwidth. RBC selects an entity at the top of the s-list if $S_{free}/\widehat{s_i} < B_{free}/\widehat{b_i}$, otherwise the one at the top of the b-list is selected. After that entity is marked for possible removal, the values of $S_{free}$ and $B_{free}$ are re-computed, and the list selection process is repeated. In this procedure, the entities selected for removal are those that have the worst goodness value in either the bandwidth constrained case or the space constrained case. The cache replacement algorithm in the dual constrained case is described in Figure 4(b).

### 2.3.1. Properties of the Entity Replacement Algorithm

The entity replacement algorithm has the following properties:

**Property** 1. *When multiple entities ordered by $\mathcal{G}_i$ values are removed from cache, their combined goodness is less than that of the entities remaining in the cache.*

**Proof:** Let entities $E_i, E_j, E_k$ be ordered by goodness. Let $bs_i, bs_j, bs_k$ be the constrained resource usage, respectively. Hence, $\frac{g_i}{\widehat{bs_i}} < \frac{g_j}{\widehat{bs_j}} < \frac{g_k}{\widehat{bs_k}}$. Now, if entities $i$ and $j$ are removed from cache, and entity $k$ remains in cache, their combined goodness is the ratio of the combined gain to resource occupied. Hence, by definition and the above inequality,

$$\frac{g_i + g_j}{\widehat{bs_i} + \widehat{bs_j}} < \frac{g_k}{\widehat{bs_k}}$$

■

**Corollary:** If a new entity is added to the cache, its goodness value is greater than the combined goodness value of all the entities it replaces from the cache.

**Property** 2. *The best-fit choice to resolve a tie among entities with equal goodness values ensures that, for a given amount of resource required (space or bandwidth), the caching gain is maximized.*

**Proof:** If two entities, $E_i$ and $E_j$, have goodness values, $\mathcal{G}_i = \mathcal{G}_j$ then $\frac{g_i}{\widehat{bs_i}} = \frac{g_j}{\widehat{bs_j}}$. Let the resource required be $BS_{req}$. If $\widehat{bs_i} > BS_{req}$ and $\widehat{bs_j} > BS_{req}$, then by best fit the entity selected for deletion has resource usage $bs_k = min(\widehat{bs_i}, \widehat{bs_j})$; the other entity with the gain of $max(g_i, g_j)$ remains. The best-fit selection can also be applied over groups of entities, if individual $\widehat{bs_i}$ values are smaller than the resource required. The combined entity will have the same goodness value as the individual ones; i.e.

$$\frac{g_i + g_j}{\widehat{bs_i} + \widehat{bs_j}} = \mathcal{G}_i = \mathcal{G}_j$$

and hence can be treated as a single entity. ■

### 2.4. Discussion

The problem of minimizing the load on the remote server and network using a cache with limited size and bandwidth, can be mapped to a two constraint knapsack problem. The RBC algorithm reduces this problem to two single constraint knapsack problems. For a single constraint cache (e.g., limited space), the caching problem reduces to the 0/1 knapsack problem. The 0/1 knapsack problem consists of filling a knapsack of fixed size S, with objects $x_i$ each with a gain $g_i$ and a size $s_i$, such that the total gain is maximized. The objective is to maximize $x_i g_i$ under the constraint that $x_i s_i < S$, where $x_i \in 0, 1$ depending on whether the object is selected or not. The 0/1 knapsack problem is known to be NP-complete. However, if the gain values for each object are identical, then the 0/1 knapsack has an optimal solution which consists of ordering the objects in ascending order of size.[27]

For a fractional knapsack problem (objects are not whole but can be fractional), the optimal solution is the greedy ordering by gain density, $g_i/s_i$. For the 0/1 knapsack problem with varying values of $g_i$, selecting entities based on the greedy ordering of gain density is a simple approximation. Various approximate algorithms of the 0/1 knapsack problem with $\epsilon$ approximation factor exist, but they are all off-line algorithms.[27] The RBC algorithm uses the gain density approximation in ordering the entities for the different cache constraints. This approximation is optimal if each entity has identical caching gain but varying resource requirement.

## 3. EXPERIMENTAL EVALUATION

In order to compare the resource based caching algorithm with other known algorithms, we have a developed a prototype system and an event-driven simulation model. The prototype is built on a cluster of AIX workstations, with one node being the proxy cache server and the remaining running the client code. The caching module at the cache server interacts with the local file-system and the remote server. The caching module maintains (i) multiple priority queues to implement the caching policy for each data type, (ii) data structures to store access information for each data type, and (iii) a hash list to map ⟨filename, logical block number⟩ to cached blocks. In on-going work, the caching module is integrated with the local multimedia file-system. To experiment with multiple configurations and understand the behavior of the caching algorithm, we have developed and experimented with an event driven simulation model, the results of which are described in the following sections. The workload used for the simulation model was derived after analyzing different web-server access logs.

## 3.1. Selecting Workloads

To determine the workload parameters, we analyzed the data gathered from NLANR/Squid cache, NCSA, NASA and Digital web-server logs.[28,29] The workload is characterized by defining: (i) the distribution of requests among different objects types, (ii) the request arrival pattern or access distribution, and (iii) the object size distribution.

- *Access distribution across data types:* Table 1 shows the access distribution across different data types gathered from different web-server access logs.[28,29] The current statistics indicate a predominance of text and image data accesses. Given that web-server access-traces are mostly for text/image data, we use synthetic traces and access distributions across data types in our simulations.

| Data Type | % Hits | % Bytes |
|---|---|---|
| Image | 67 | 46 |
| Text+html | 15 | 10.6 |
| Video | 1 | 2 |
| Audio | 0 | 0 |

NLANR Squid Cache

| Data Type | % Hits | % Bytes |
|---|---|---|
| Image | 48 | 36 |
| Text+html | 51 | 50 |
| Video | 0.1 | 6 |
| Audio | 0.2 | 3 |

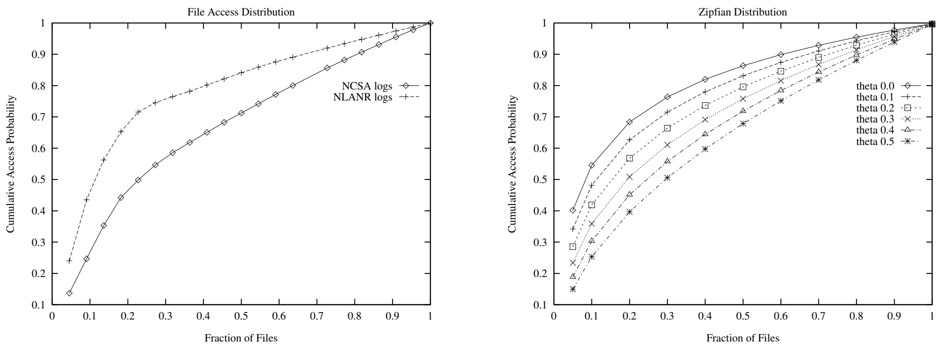
NCSA logs

| Data Type | % Hits | % Bytes |
|---|---|---|
| Image | 63 | 48 |
| Text+html | 30 | 18 |
| Video | 1 | 30 |
| Audio | 0.2 | 1 |

NASA logs

**Table 1.** Web-server Request Distribution Data

- *Determining Access Skew:* We analyzed the web-server access logs to determine the file access skew. Figure 5(a) depicts the file access skew graph for two web servers, NCSA and NLANR/Squid. It can be seen that about 70% of the accesses are restricted to 20% of the files, that is a 70-20 skew. Skews of the order of 80-20 and 90-10 are not uncommon.[28] It has been shown that the Zipf distribution can be used to accurately model such access skews.[21] In our workload, we selected the file access skew to be derived from a Zipfian distribution for different values of the skew parameter $\theta$. A Zipf distribution yields high access skews at low values of $\theta$, and tends to become more uniform at larger values of $\theta$. Figure 5(b) shows the cumulative access probability of objects as a function of the fraction of objects accessed for a Zipf distribution with different values of $\theta$. It demonstrates that, for a cumulative access probability of 75%, the fraction of objects accessed ranges from 26% to 60% as the value of $\theta$ increases from 0 to 0.4, which matches that observed in Figure 5(a).




(a) Access Log Distributions (b) Zipf Distribution

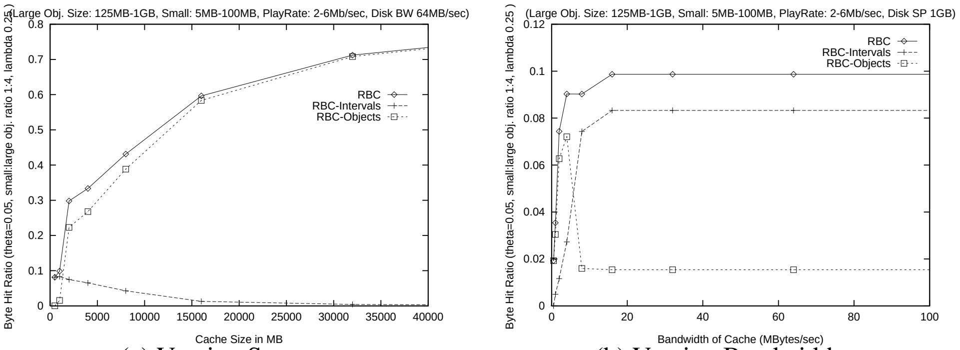
**Figure 5.** File Access Distribution

- *Object Sizes:* The NCSA/NASA servers consist of text and images objects ranging from 3 to 64KB and video and audio objects ranging from 100KB to 15MB. Although these figures show a relatively small-sized CM data files, they are more due to the limitations of current network bandwidth and viewing preferences. However, it can be seen that the object sizes of text/images differ by an order of magnitude from audio and video. Therefore when selecting object sizes, in our workload, the short CM objects (e.g., audio size is around 1MB) are set to be orders of magnitude smaller that the large objects (e.g., video range is 100MB), but are an order of magnitude larger than the text and image objects (100 KB). Each class of objects has a range of uniformly distributed sizes.

In what follows, we analyze the behavior of the resource based caching algorithm and compare it with several known algorithms for varying workload parameters. Note, that the space and bandwidth values assumed for cache are not representative of a single disk but that of a logical disk cache that could be mapped to a disk array.

## 3.2. Analysis of the RBC Algorithm

We analyze the behavior of the RBC algorithm for the objective of maximizing BHR, with varying cache space and bandwidth capacity. For this analysis, the workload is restricted to CM objects of varying size (short audio clips to large video streams) and bandwidth requirements; the granularity is restricted to entire objects and intervals. The algorithm uses the WLRU-n technique for estimating the time-to-reaccess for different objects which is used in the goodness computation based on the resource requirement and cache state. Figures 6(a,b) illustrate how the RBC algorithm selects the granularity of the cached entity as the cache resources vary. Figure 6(a) shows that, when the cache size is small (less than 2GB) for a fixed bandwidth of 64MB/sec, the RBC algorithm selects more intervals than entire objects. As the cache size increases and the cache bandwidth becomes the bottleneck, more of the cache is filled with entire objects. In fact, when the cache size is large ($>$ 20GB) only entire objects are selected for caching. Figure 6(b), on the other hand, shows that, for a cache size of 1GB, when the cache bandwidth is low (less than 5MB/sec), a large proportion of the cache is filled with entire objects so as to amortize the writing overhead over a large number of reads. As the cache bandwidth increases, however, the cache size becomes the constraint and the number of intervals selected increases.




(a) Varying Space (b) Varying Bandwidth

**Figure 6.** Analysis of the RBC algorithm

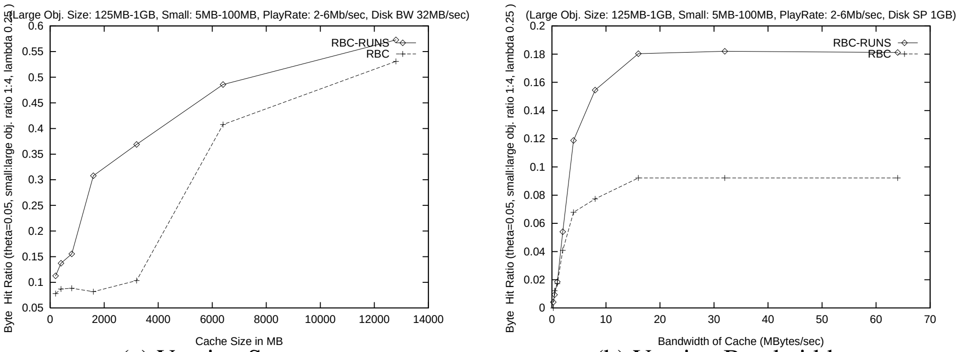
### 3.2.1. Extending Intervals to Runs

As outlined in Section 2.1.1, when multiple users are accessing the same object, a set of adjacent intervals could be grouped together to form *runs* and considered for caching. When the bandwidth and space of the cache is limited, it is profitable to group intervals together to amortize the write overhead over multiple readers and use less space. Figures 7(a,b) compares the hit ratios yielded by two versions of the RBC caching algorithm: one that considers only intervals and the other that considers runs (or groups of intervals). When the bandwidth is limited (32MB/sec), the RBC algorithm which caches runs of intervals yields around 25% higher hit ratios (see Figure 7(a)). On the other hand, in Figure 7(b) when the cache size is fixed (1GB), RBC with run caching improves the hit ratio by around 70%. This is due to the improved bandwidth usage for a much lower space usage compared to entire objects or intervals. For the following experimental results, by default, RBC caching includes runs unless otherwise stated.

## 3.3. Comparison of RBC and other Algorithms

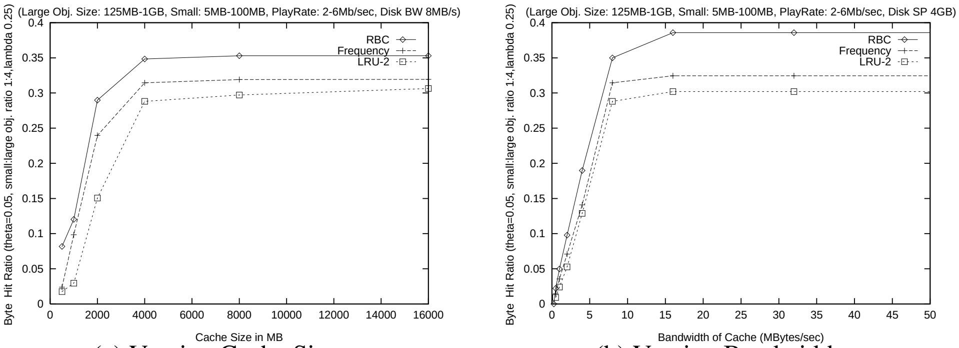
To evaluate the performance of RBC, we compare its BHR values with other traditional caching algorithms. In this comparison the workload is restricted to CM objects of varying size and bandwidth requirements; the granularity consists of entire objects, intervals and runs. The RBC algorithm is compared with (i) frequency-based caching of entire objects using WLRU-n, and (ii) recency-based caching using LRU-2.

Figure 8(a) and 8(b) show the variation in the byte hit ratios for the different cache algorithms with varying cache size (with fixed disk bandwidth 8MB/sec) and varying cache bandwidth (with fixed space 4GB), respectively. They demonstrates that, once the cache bandwidth (space) saturates, the byte hit-ratio reaches a plateau, and is unaffected by further increase in cache size (bandwidth). Figure 8(a) shows that for a fixed cache bandwidth, when the space increases, the bandwidth becomes




(a) Varying Space (b) Varying Bandwidth

**Figure 7.** Effect on run caching on the performance of RBC algorithm




(a) Varying Cache Size (b) Varying Bandwidth

**Figure 8.** Comparison of caching algorithms - a snapshot

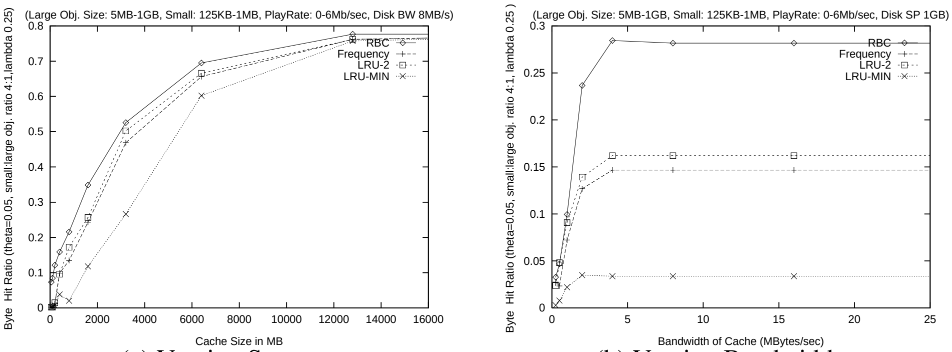
a constraint. The RBC algorithm forms an envelope over both recency and frequency caching algorithms, with around 20% higher hit ratio. Figure 8(b) shows that for a fixed cache size, when the bandwidth increases, the cache space becomes the constraint, and the hit ratios saturate; the RBC algorithm remains an envelope of either policies (atleast 29% higher than all the other policies).

Note that, for the objective of maximizing BHR in the space constrained case, the goodness value maps to inverse of TTR, which is identical to the ordering used by all algorithms. The only difference being the computation of TTR. RBC differs from them however, by choosing the granularity of the cached CM entities based on the space utilization of the cache, selecting either intervals, runs or entire objects. In the bandwidth constrained case, RBC selects for replacement entities that have a large write overhead while other algorithms use the TTR ordering. RBC also distinguishes between objects that are being written to the cache and the ones that are already in the cache. Consequently, RBC has much higher hit ratios compared to the other policies when both space and bandwidth are limited.

## 3.4. Mixed Continuous and Non-continuous Media Objects

In the previous experiments the workload was restricted to CM objects that had both space and bandwidth requirements. Non-CM objects on the other hand, do not require any retrieval rate guarantees, and hence do not reserve any bandwidth. Figures 9(a,b) compare the byte hit ratios yielded by the LRU-MIN, LRU-2, frequency-based and the RBC algorithm for mixed CM and non-CM objects. LRU-MIN was selected as a representative web caching algorithm that considers the time of last access and the size of the objects. Since interval caching is not suited for non-CM objects it was not used for the comparison. For this experiment, the sizes of non-CM objects were assumed to range from 100KB to 1 MB, while the sizes of CM objects ranged from 5MB to 1 GB. The data rate requirement for CM objects was assumed to vary in the range 2-6Mb/s. The ratio of accesses is 80% for non-CM objects and 20% for CM objects.

As Figure 9(a) demonstrates, RBC caching algorithm yields higher byte hit ratios (10-50%) than the LRU-2 and frequency-based algorithm. For the space constrained case shown in figure 9(b) for a cache size of 1GB, RBC yields a byte hit ratios that are 6 times higher than LRU-MIN and 80% higher than frequency or recency based caching. Note that while the number of



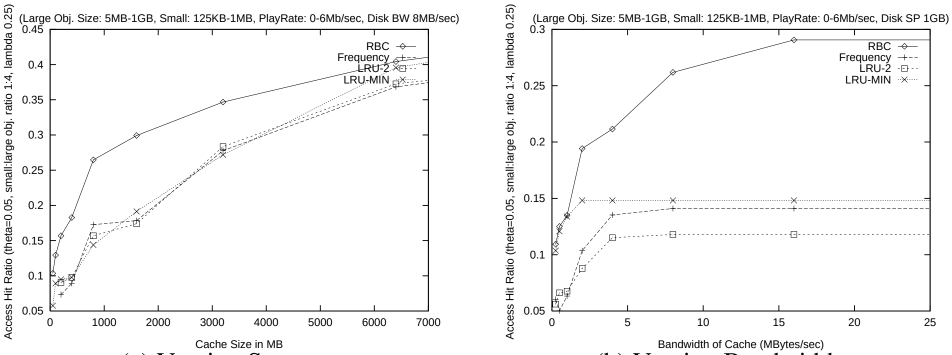

(a) Varying Space (b) Varying Bandwidth

**Figure 9.** Mixing non-CM and CM data

references to CM objects are much smaller, they contribute much more to the byte hit ratio due to their size. However, since LRU-MIN deletes all objects larger than the one to be inserted in LRU order, it has the poorest performance. Since the RBC algorithm considers the cache resource available, it performs better than LRU-2 or frequency-based algorithms.

The mixing of non-CM objects with CM objects does not affect the behavior of the algorithms when bandwidth is not a constraint. When bandwidth is a constraint, however, the performance of other algorithms is much poorer compared to RBC. Since other algorithms do not consider the bandwidth occupied by objects, they may unnecessarily replace non-CM objects which do not release any reserved bandwidth.

### 3.5. Hit Ratio as Performance Objective



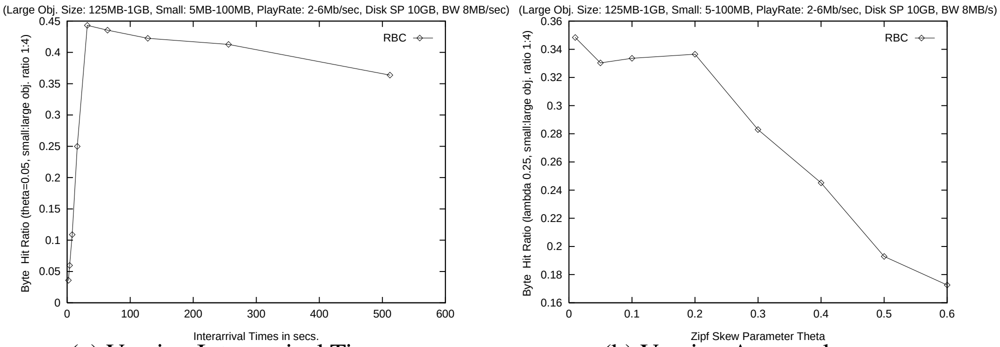

(a) Varying Space (b) Varying Bandwidth

**Figure 10.** Performance Objective as Access Hit Ratio

In the previous experiments the performance objective was chosen to be the byte hit ratio. It was shown in Section 2.1.3, that for BHR the ordering in the space constrained scenario mapped to the inverse of the time-to-reaccess. Thus although a large sized object uses larger cache space, it has a proportionately larger byte hit ratio. When the cache objective is to maximize hit ratio (based on the number of accesses) the goodness value of entities in the RBC algorithm map to the access rate per unit resource used. Figure 10 compares the hit ratio of RBC, frequency caching, LRU-2 and LRU-MIN. RBC has much higher hit rate values compared to the algorithms (36%). This is due to the fact that, unlike BHR, a large sized object occupies more cache space but does not contribute proportionately to the hit ratio. None of the other algorithms considered the space and bandwidth usage of the entities to be cached. Figure 10(a) shows the bandwidth constrained case (BW limited to 8MB/sec) for varying space where the RBC algorithm performs 36% better than LRU-2 and the frequency-based algorithm. LRU-MIN shows a better performance compared to LRU-2 and frequency-based caching since it deletes the larger size objects first. Note that a large size object occupies larger cache space without any proportional contribution to the cache hit ratio, and hence replacing in the larger size first order improves performance. Figure 10(b) depicts the space constrained scenario (space limited to 1GB) for varying bandwidth. The RBC algorithm achieves a 90% higher hit ratio compared to the other algorithms.

### 3.6. Varying Workload Characteristics

The values of the performance objective, HR or BHR are also affected by the access skew and the request arrival rates. Figure 11(a) shows the effect of varying the request inter-arrival times on the byte hit ratio for RBC. It demonstrates that very small inter-arrival times cause the cache to be in a constant state of upheaval resulting in thrashing and hence, low hit ratios. Increasing the inter-arrival time decreases the number of concurrent accesses to an object, and hence, increases the average interval length. This results in a smaller number of intervals to be cached, and hence, reduces the byte hit ratio. Figure 11(b) shows the effect of varying the access skew. As the access skew becomes more uniform (with increasing $\theta$), the byte hit ratio falls as there are fewer concurrent accesses.




(a) Varying Inter-arrival Times (b) Varying Access skew

**Figure 11.** Effect of Workload

## 4. CONCLUDING REMARKS

The WWW employs a hierarchical data dissemination architecture in which hyper-media objects stored at a remote server are served to clients across the internet, and cached at intermediate proxy servers on-demand. To minimize the load imposed on the network and servers, caching is used to maximize the data transferred from the proxy server caches. These cache algorithms differ from processor caches in that they handle multiple data types, each with different cache resource usage, for a cache limited by both bandwidth and space. In this paper, we present a resource based caching (RBC) algorithm that manages the heterogeneous requirements of web caches. The RBC algorithm (i) dynamically selects the granularity of the entity to be cached that minimally uses the limited cache resource (i.e., bandwidth or space), and (ii) replaces the cached entities based on their cache resource usage and caching gain. Through extensive simulations, we demonstrated that the RBC algorithm achieves higher hit ratios as compared to several existing algorithms under widely varying workloads and cache configurations. The evaluation of the prototype system and integrating the caching module with the file-system for efficient delivery of hyper-media objects is the focus of some of our on-going research.

## REFERENCES


1. A. Luotonen and K. Altis, "World-wide web proxies," *Computer Networks and ISDN Systems* **27**(2), 1994.
2. D. Wessels, " Squid Internet Object Cache," in *online document available from http://squid.nlanr.net/Squid*, May 1996.
3. A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, " A Hierarchical Internet Object Cache," in *Proceedings of 1996 Usenix Technical Conference*, January 1996.
4. G. A. Gibson, J. Vitter, and J. Wilkes, "Storage and I/O Issues in Large-Scale Computing," *ACM Workshop on Strategic Directions in Computing Research, ACM Computing Surveys* , 1996. http://www.medg.lcs.mit.edu/doyle/sdcr.
5. VXtremeInc, "Server-less Versus Server-based Video Streaming," in *online document available from http://www.vxtreme.com/developers/wp970101.html*, 1997.
6. M. Blaze and R. Alfonso, "Dynamic Hierarchical Caching in Large-Scale Distributed File Systems," in *Proceedings of International Conference on Distributed Computing Systems*, June 1992.
7. M. D. Dahlin, R. Wang, T. E. Anderson, and D. Patterson, "Cooperative caching: Using remote client memory to improve file system performance1994," in *Proceedings of the Operating Systems Design and Implementation Symposium*, 1994.

8. A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," in *ACM SIGMETRICS*, pp. 143–152, May 1990.
9. M. Feeley and et. al., "Implementing global memory management in a workstation cluster," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
10. R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, " Informed Prefetching and Caching," in *Proceedings of the 15th Symposium on Operating System Principles*, pp. 79–95, December 1995.
11. H. Chou and D. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Proceedings of the 11th VLDB Conference* , 1985.
12. D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU Replacement Policy: A spectrum of block replacement policies ," in *Seoul National University Technical Report SNU-CE-AN-96-004*, March 1996.
13. E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-k page replacement algorithm for database disk buffering," in *Proceedings of International Conference on Management of Data*, May 1993.
14. P. Cao, E. W. Felten, and K. Li, " Application-controlled file caching policies," in *Proceedings of 1994 Usenix Summer Technical Conference*, pp. 171–182, June 1994.
15. A. Dan, D. Dias, R. Mukherjee, D. Sitaram, and R. Tewari, "Buffering and caching in large scale multimedia servers," in *Proceedings of IEEE COMPCON*, pp. 217–224, March 1995.
16. A. Dan and D. Sitaram, "Buffer Management Policy for an On-Demand Video Server," IBM Research Report RC 19347, T.J Watson Research Center, Yorktown Heights, New York, January 1993.
17. A. Dan and D. Sitaram, "A Generalized Interval Caching Policy for Mixed Interactive and Long Video Environments," in *IS&T SPIE Multimedia Computing and Networking Conference*, (San Jose, CA), January 1996.
18. B. Ozden, R. Rastogi, and A. Silberschatz, "Buffer replacement algorithms for multimedia storage systems," in *Proceedings of the International Conference on Multimedia Computing and Systems*, pp. 172–180, June 1996.
19. A. Bestavros and et. al., "Application-Level Document Caching in the Internet," in *Proceedings of Workshop on Services and Distributed Environments*, June 1995.
20. M. Abrams and et. al., "Caching proxies: Limitations and potentials," in *4th International World-Wide Web Conference*, pp. 119–133, December 1995.
21. S. Williams and et. al., "Removal policies in network caches for world-wide web documents," in *ACM SIGCOMM*, pp. 293–305, August 1996.
22. J. Pitkow and M. M. Recker, "A simple yet robust caching algorithm based on dynamic access patterns," in *Proceedings of 2nd International WWW Conference*, pp. 1039–1046, October 1994.
23. K. Andrews and et. al., "On second generation hypermedia systems," in *Proceedings of ED-MEDIA, World Conference on Educational Multimedia and Hypermedia*, June 1995.
24. S. Irani, "Page Replacement with Multi-size Pages and Applications to Web Caching," in *Proceedings of Symposium on Theory of Computing*, March 1996.
25. J. Bolot and P. Hoschka, " Performance Engineering of the World Wide Web," in *Proceedings of WWW Journal, available online at http://www.w3.org/pub/WWW/Journal/3/s3.bolot.html*, pp. 185–195, Summer 1996.
26. R. Wooster and M. Abrams, " Proxy Caching that Estimates Page Load Delays," in *Proceedings of Sixth International WWW Conference, available online at http://proceedings.www6conf.org/HyperNews/get/PAPER250.html*, April 1997.
27. C. Papadimitriou and K. Steiglitz, *Combinatorial Optimization, Algorithms and Complexity*, Prentice Hall, 1982.
28. M. F. Arlitt and C. L. Williamson, " Web Server Workload Characterization: The Search for Invariants," in *Proceedings of SIGMETRICS 96*, pp. 126–137, May 1996.
29. T. Kwan, R. E. McGrath, and D. A. Reed, "User Access Patterns to NCSA's World Wide Web Server," Dept. of Computer Science Research Report available online at http://www-pablo.cs.uiuc.edu/Papers/WWW.ps, University of Illinois at Urbana-Champaign, February 1995.