A Hybrid Caching Strategy for Streaming Media Files

Jussara M. Almeida[✤a], Derek L. Eager[b], Mary K. Vernon[a]

[a]Department of Computer Science, University of Wisconsin-Madison
1210 West Dayton Street, Madison, Wisconsin, 53706
USA

[b]Department of Computer Science, University of Saskatchewan
57 Campus Drive, Saskatoon, SK S7N 5A9
Canada

ABSTRACT

A recently proposed streaming media file caching algorithm, called Resource Based Caching (RBC), considers the impact of both file size and required delivery bandwidth in making cache insertion and replacement decisions. Previous comparisons between RBC and the least-frequency-used (LFU) policy conclude that RBC provides a better byte hit ratio in the cache. This paper revisits this policy comparison over a much broader region of the system design space than previously considered. The results provide more complete insight into the behavior of RBC, and support new conclusions about the relative performance of RBC and LFU. A new policy, Pooled RBC, is proposed. Pooled RBC includes three improvements to the original RBC policy and has significantly better performance than RBC. Finally, a new *hybrid LFU/interval caching* strategy is proposed. The new hybrid policy is significantly simpler to implement than RBC and performs as well or better than both Pooled RBC and LFU.

Keywords: streaming media, caching policies, proxy servers, performance evaluation, client workload

1. INTRODUCTION

The design of efficient caching policies for streaming media files is a challenging open research problem for several reasons. First, video files are typically large. A single file may require tens of megabytes to tens of gigabytes of storage, depending on the quality and duration of the video. Thus, most of the cached content must be stored on disks, and both the disk cache and main memory buffering must be carefully managed. Second, real time media file transfers require a significant amount of disk and network i/o bandwidth, sustained over long periods of time. This implies that effective caching algorithms should avoid using too much disk bandwidth to store new content in the cache during periods of high client interest in the current cache content. Third, clients may only be interested in receiving parts of files, as they request portions of the videos that especially interest them. This implies that partial file caching may be important. Finally, not much data is available about client behavior in video-on-demand (VOD) systems, and client behavior may change as streaming media systems become more powerful and new applications emerge. Thus, media caching algorithms should be designed to be robust for evolving client workloads, and should be evaluated over a wide range of current and hypothetical future system workloads.

Several media caching algorithms have been proposed and evaluated in prior work. Early papers[6,8,10,15,18,19] propose and analyze algorithms for caching intervals of video file data in main memory, so as to satisfy multiple client requests that arrive close in time. Tewari et al. [31,32] define a new disk-based caching policy called the resource-based caching (RBC) algorithm, which considers disk bandwidth as well as disk storage capacity, and caches a mixture of intervals and full files that have the greatest *caching gain*. They compare their RBC algorithm against the least-frequently-used (LFU) cache replacement algorithm that caches the videos with highest request frequencies, and conclude that RBC provides a better *byte hit ratio* in the cache. Rejaie et al. define a frequency-based caching architecture and policy for layered video streams [23,24], and explicitly leave the task of defining the RBC algorithm for layered video streams for future work[23]. In complementary work, several papers have proposed prefix caching [28] and other selective partial caching[17,34] so that variations in media display rate at the client end can be accommodated in the context of fixed-rate or smoothed variable-rate streaming from the server.

[✤] This work was partially supported by the National Science Foundation under Grant CCR 9975044, and by the Natural Sciences and Engineering Research Council of Canada under Grant OGP-0000264. Jussara Almeida is also partially supported by CNPq/Brazil. Authors may be contacted at: jussara@cs.wisc.edu, eager@cs.usask.ca, vernon@cs.wisc.edu.

Motivated by the narrowness of the system design space considered in the previous evaluations of the RBC and LFU caching algorithms, this paper revisits the comparison of these caching algorithms over a much broader region of the system design space than previously considered. In particular, the paper makes the following contributions to understanding streaming media caching strategies:

- A more complete definition of the RBC policy is provided, based on input from the policy inventors[30].
- A small number of key system and workload parameters are identified that can be varied to compare policies over a wide range of system configurations and client workloads.
- New results are provided that give more complete insight into the behavior and performance of the RBC policy.
- New conclusions are reached about the relative performance of RBC and LFU, partly due to the larger system design space considered in this paper.
- Four improvements to the RBC policy are proposed. The new policy with these improvements, *Pooled RBC***,** is evaluated and Pooled RBC outperforms RBC over the system and workload parameter space examined in this paper.
- A new *hybrid LFU/interval caching (LFU/IC****)*** policy is proposed. A key result of this paper is that LFU/IC is significantly simpler to implement than the RBC and Pooled RBC policies, and performs as well or better than both Pooled RBC and ordinary LFU over the entire system design space examined in this paper.

Based on insights derived during the policy evaluations, we conjecture that the generic class of hybrid LFU/IC policies is quite promising for both current and future VOD systems and client workloads.

Parallel work[1] considers a proxy cache that is distributed among *N* servers in which each video file is partitioned into *N* blocks for load balancing purposes. They use a set of client request traces[2] to compare several hierarchical file/block-level caching policies, including: LFU, FIFO, and LRU-k, under the assumption that the disks always have enough bandwidth to update and deliver the cache content for any of these caching policies. LRU-k caches a file whenever it is accessed and, if space is needed, selects for eviction the file with the largest k-distance, i.e., the largest difference between the current time and the time at which the kth most recent access was made to the file. For many of the configurations and traces they study, the LFU and LRU-k policies have similar performance that is superior to the other policies; in a few cases, LRU-3 is shown to have better (up to 8% higher) block hit rate than their LFU algorithm. The LFU/IC policy proposed in this paper is different than their LFU policy. First, the LFU/IC policy we envision estimates file access frequency before deciding to cache a file. Second, the LFU/IC policy caches intervals as well as files or popular file segments. However, the k-distance measure could be a useful practical method of estimating file (or partial file) access frequency in the LFU/IC policy. We comment on this further in section 2.1. Experimental comparison of the LFU/IC and LRU-k policies is left for future work.

The policy comparisons in this paper are performed using simulation with a synthetic workload controlled by a set of workload parameters that are designed to include workload characteristics reported in the literature[2,3,5,7,14]. The synthetic workload technique ensures that a known (and broad) region of the system design space is evaluated.

The rest of the paper is organized as follows. Section 2 provides definitions of the LFU, interval caching, and RBC policies. Section 3 defines the system model and evaluation methodology used in the performance evaluations presented in this paper. Section 4 shows results that compare the performance of RBC and LFU, and provides insights into the behavior of the RBC policy. Section 5 defines the new Pooled RBC and hybrid LFU/IC policies, and provides performance results comparing the new policies against RBC and LFU. Concluding remarks are provided in Section 6.

2. BACKGROUND

2.1. The LFU Caching Policy

The LFU caching policy simply caches the files or partial files that are estimated to have the highest access frequency at the current time. Two key open questions in implementing the policy are: (1) for what data granularity should access frequency be measured, and (2) for a given data granularity, how should current access frequency be estimated in an actual system?

For highest performance, if partial file caching is supported, access frequency should be measured at the granularity of the data requests that are issued by the clients. Similarly, access frequencies for different layers of a layered video stream should be measured separately, as suggested in previous works [23,24]. A key factor in assessing the appropriate granularity for a given system is to achieve a good balance between the precision and the simplicity of obtaining and maintaining the frequency measures.

To predict access frequencies for a given data granularity, the simplest cases are such that simple reference counts can be used, or future access frequencies are known from previous access frequencies. However, in the more common case, access frequencies change over time in unpredictable ways. In this case, precisely how to estimate current access frequencies is an open question. One proposed approach is to use reference counts with an aging mechanism such as a periodic right shift of one bit to form an exponential decaying average usage count [27]. Another proposed approach is to use the reference count, perhaps weighted by the fraction of the file that is accessed in each request, over the most recent interval of (unspecified) length t [23,24]. A third approach [32] is to estimate the mean time between requests (i.e., the inverse of the access frequency) each time a new request arrives, using a weighted average of the most recent inter-request time and the mean inter-request time computed when the previous request arrived. We note that, for any integer k, the ratio of k to the time since the k^{th} most recent client request can also be considered an estimate of file or partial file access rate. In this case, the distinction between the LFU policy and the LRU-k policy is that the LFU policy only stores a file in the cache when this measure of access frequency is among the highest access frequencies for the files and/or partial files that fit in the cache. LRU-k stores a file (or partial file) in the cache whenever it is accessed.

Each policy evaluated in this paper (i.e., LFU, RBC, Pooled RBC, and LFU/IC) uses file access frequency estimates for deciding which data should be stored in the cache. Since methods for estimating access frequency are an open and active research topic, we will compare the performance of these policies primarily under the assumption that each policy has perfect knowledge of access frequencies. We have verified that the relative policy performance results obtained in this paper are the same if each policy uses the particular method of estimating access frequency used in the original performance comparison of RBC and LFU [32], which is the third approach outlined above. It thus seems likely that the relative policy performance will be insensitive to improved practical methods for estimating access frequency that may be developed in the future.

2.2. The Interval Caching Policy

Interval caching policies[6,7,8,10,15,18] have been proposed and analyzed for caching partial video streams in main memory. In these policies, if two client requests for file i arrive close together in time, then the data delivered to the first client may be stored in the memory cache until it is delivered to the second client. More generally the data may be stored in the memory cache until it has been delivered to several future clients that have requested the file close together in time. If the data is deleted when the second client receives it, the cached data is called an *interval* of data (equal to the amount of data delivered between the first two client requests). If the data is deleted when a later client receives it, the cached data is called a *run* (equal to the sum of several intervals between consecutive client requests). The policy in these previous papers caches the set of smallest intervals or runs that can fit in the main memory cache.

2.3. The RBC Caching Policy

The Resource Based Caching (RBC) algorithm is a disk caching policy that characterizes each file object by its space and bandwidth requirements and a measure called *caching gain*. RBC tries to efficiently utilize the limited resource, either disk space or disk bandwidth. Like LFU, RBC handles other files as well as streaming media files. However, to simplify the description of the policy, we assume a workload of only streaming media files. The development of the RBC policy was guided by two sometimes incompatible goals: (1) keep disk space utilization and disk bandwidth utilization approximately equal, and (2) replace the entity in the cache that is estimated to be needed again farthest in the future. As we illustrate in Section 4, the value of the first goal is somewhat unclear; a more intuitive goal is simply to maximize the disk bandwidth that is used to deliver content to clients. The second goal leads to optimal policy performance if the estimate of which data is needed farthest in the future is accurate and the system is not bandwidth-constrained.

In the remainder of this section, we describe the RBC policy as defined in previous literature[32], followed by some clarifications of the policy that are omitted in the previous literature. Along the way, we propose two changes to the policy that do not affect the performance results obtained in this paper, but which are perhaps more intuitive than the original definition of the policy.

2.3.1. RBC as Defined in the Previous Work

RBC uses the parameters defined in Table 1. When a client request for file i arrives and file i is not fully cached, the RBC policy follows two steps. In step 1, the system uses the criteria in Table 2a to select the granularity of file i (i.e., interval, run, or full file) that should be considered for placement in the cache. In step 2, the system decides whether the entity selected in step 1 should be cached, possibly removing other cached entities, as described in Table 3.

RBC keeps two ranked lists of cached entities: one is ranked by *space goodness*, the other by *bandwidth goodness*. The entity (i,x) selected in Step 1 will only be cached if space equal to $S_{i,x}$ *and* bandwidth equal to $B_{i,x}$ can be allocated to it. Note that for full files, this implies that *bandwidth equal to the estimated average number of clients that will actively view the file is* *allocated to the file for the duration of its lifetime in the cache*. The requisite space and bandwidth can be allocated from unallocated space and bandwidth, respectively, or by replacing cached entities with lower space goodness and/or lower bandwidth goodness, following the rules given in Table 3. An interval or run is deleted from the cache when the last client to join the interval or run finishes receiving the file.

The criteria used in step 1 of the RBC algorithm are designed to keep bandwidth and space utilization equal when choosing the granularity of the entity to cache. Given this goal, however, it is not clear why in the case when U_{space} is greater than $U_{bw} + \varepsilon$, RBC selects the entity with minimum $S_{i,x}$ rather than the entity with maximum space goodness. We have experimented with a simpler and more intuitive rule for step 1, given in Table 2b, namely always select the entity with the largest space goodness, regardless of the values of U_{bw} and U_{space}. The RBC policy evaluated in section 4 uses the rules for step 1 as defined in Table 2a; the Pooled RBC policy evaluated in section 5 uses the simplified rule in Table 2b. Further experiments have determined that, perhaps surprisingly, the change in rules for step 1 does not affect the performance of RBC.

Table 1: Parameters Used in RBC

Notation	Description
$R_{i,file}$	Estimated average number of clients that request file *i* during the delivery time for file *i*
$R_{i,x}$	Current number of clients receiving but not writing entity *x* of file *i* ($x \in$ {interval, run}).
$W_{i,x}$	Current number of clients receiving and writing entity *x* of file *i* to disk ($x \in$ {interval, run ,file})
R_i	Bit rate for receiving file *i*.
$B_{i,x} = (R_{i,x} + W_{i,x}) \times r_i$	Bandwidth requirement of entity *x* of file *i* ($x \in$ {interval, run , file})
$S_{i,x}$	Size, in bytes, of entity *x* of file *i* ($x \in$ {interval, run , file})
$g_{i,x} = R_{i,x} \times r_i$	**Caching gain** for entity *x* of file *i* ($x \in$ {interval, run , file})
$G_{space}(i,x) = g_{i,x}/S_{i,x}$	**Space goodness measure** for entity *x* of file *i*
$G_{bw}(i,x) = g_{i,x}/B_{i,x}$	**Bandwidth goodness measure** for entity *x* of file *i*
U_{space}	Current disk space utilization
U_{bw}	Current disk bandwidth utilization

Table 2: RBC Step 1 - Select Granularity of Entity (x)
a) Original RBC Algorithm

(1) *If* $U_{bw} > U_{space} + \varepsilon$ *:*

Choose entity x with the lowest $\dfrac{W_{i,x}}{R_{i,x} + W_{i,x}}$

(2) *If* $U_{space} > U_{bw} + \varepsilon$ *:*

Choose entity x with minimum $S_{i,x}$

(3) $U_{space} - \varepsilon < U_{bw} < U_{space} + \varepsilon$ *:*

Choose entity x with largest $\dfrac{R_{i,x} / (R_{i,x} + W_{i,x})}{S_{i,x}}$

(b) Proposed Simplified Step 1

(1) *Choose entity x with maximal* $G_{space}(i,x)$

Table 3: RBC Step 2 - Caching Decision for Entity x

(1) *If enough unallocated space and unallocated bandwidth:*
Cache entity x

(2) *If enough unallocated space but bandwidth constrained:*
Use bandwidth goodness list to select candidates for eviction

(3) *If enough unallocated bandwidth but space constrained:*
Use space goodness list to select candidates for eviction

(4) *If both bandwidth and space constrained:*
Let S_{free} *and* B_{free} *be the unallocated space and unallocated bandwidth, respectively. Walk on both lists: at each step, remove entity from bandwidth goodness list if* $S_{free}/S_{i,x} > B_{free}/B_{i,x}$ *or from space goodness list otherwise.*

2.3.2. Clarifications of the RBC Policy

The previous literature[29,32] omits some key details of the RBC policy. Fortunately, one of the policy inventors has given us the missing information needed to perform our policy comparisons[30]. In this section, we list some of the most important details for future reference.

- Runs versus consecutive intervals:

  When a new client request could create either a larger run or a new (consecutive) interval in the cache, we learned that the original policy [29,32] does not consider the possibility of a consecutive interval in Step 1. We thus implemented that assumption in the results provided in this paper. We also implemented a version of RBC that considers both the interval and the run in Step 1, and did not observe any significant change in performance for the workloads in this paper.

- What happens when a new request to a cached file arrives and all of the file's pre-allocated bandwidth is already in use?

  In this case, RBC [29,32] allocates a new stream from the cache only if there is enough unallocated disk bandwidth or if enough entities with lower bandwidth goodness can be evicted from the cache in order to allocate a new stream for the file. We tried omitting the step that evicts cached entities, which resulted in very slightly better system performance. The results provided in section 4 are for the original RBC policy, which includes the evictions. The Pooled RBC policy, defined in section 5.1, does not evict entities when bandwidth is needed for a cached full file.

- What happens when the policy decides to cache a full file and there are already intervals of the file in the cache?

  The original computes the additional resources needed to cache the full file, and if the requisite space and bandwidth can be allocated the full file is cached. In this case, the intervals are still kept in the cache as long as the full file is cached, but if the full file is later evicted from the cache, the intervals will also be evicted. For the results in this paper, we use the same algorithm for deciding whether to cache the full file. However, if the full file is later evicted, but the intervals are still active, we allow the intervals to be retained in the cache (with their original space goodness, bandwidth goodness and bandwidth allocation). We experimented with both alternatives and found no noticeable difference in performance.

We point out that the RBC policy is significantly more complex than the LFU policy, in terms of implementation as well as running time. For RBC, each cached entity must have an entry on each goodness list. On each caching decision, possibly both goodness lists must be searched. The relative performance of LFU and RBC should be evaluated in the context of these relative implementation and runtime complexities.

2.4. Disk Technology Trends

An important factor in relative caching policy performance is the *ratio* of effective proxy server streaming bandwidth to storage space, which changes with each new generation of disks. For example, a Seagate Barracuda model ST15150W disk, which is just two generations earlier than today's disks, has a capacity of 4GB and a peak transfer rate that varies from 4.8 to 7.32MB/s, depending on which track stores the data. In contrast, today's Ultrastar 72ZX has roughly 18 times the capacity of the Barracuda (i.e., 73.4 GB), and roughly 5 times more bandwidth than the Barracuda (i.e., peak transfer rate that varies from 22–37 MB/s) [20]. These two disks illustrate how rapidly disk technology is evolving, and that disk capacity increases by 60% each year while transfer rate increases by only 40% a year [22]. These multiplicative factors create an increasing gap between disk bandwidth and disk capacity. The implications of these trends for evaluating relative proxy caching policy performance are discussed further in Section 3.

3. CACHING POLICY COMPARISONS: SYSTEM MODEL AND EVALUATION METHODOLOGY

The *byte hit ratio* for a given streaming media caching policy is defined as the fraction of the total bytes delivered that are delivered from the cache. Note that this ratio may be smaller than the fraction of delivered bytes that are found in the cache, due to the finite disk bandwidth. We use byte hit ratio as the primary metric for comparing caching policies, as it provides a direct measure of the savings in remote network and server bandwidth. We also provide disk space and bandwidth utilization, as well as graphs of the entities that are cached, to provide further insights into the behavior of the policies.

Throughout the remainder of the paper, we refer to the objects that clients request as files, although any given object might be a partial file that is accessed more frequently than other portions of the same file. We also assume that the proxy server is able to use all of its effective disk bandwidth for streaming data to clients. That is, the server has sufficient network i/o bandwidth for streaming data from its memory and disks. If, instead, network i/o is the bottleneck, the system parameters defined below could be modified to use network i/o bandwidth instead of disk streaming bandwidth.

Many system and workload parameters affect the byte hit ratio for each of the caching policies. These parameters include the file size, delivery rate, and request rate for each of the files that are requested by clients, the number of files, the number of disks on the proxy, the disk storage space and bandwidth, and the method used to place the cached files on the disks. Previous studies of caching policies have fixed most of the system and workload parameters (e.g., particular file sizes, streaming rates, and popularity ranking), and then independently varied a small number of the parameters (such as cache size, proxy bandwidth, and client request arrival rate). Furthermore, the parameters have been varied over very narrow ranges of possible values. How the policies compare for other proxy systems and workloads is then unknown.

Fortunately, as explained below, it is neither necessary nor desirable to vary each of the system and workload parameters independently over the range of possible values in order to understand the relative performance of caching policies for the range of proxy systems and workloads that can occur in practice. Instead, we define a large region of the system and workload parameter space that can be explored by varying the five parameters that are given in Table 4 and defined further below. Moreover, as explained below, we have run additional simulation experiments to check that the policy comparisons are similar for other important regions of the system and workload parameter space.

Table 4: Principal System Model Parameters

Parameter	Description
n	Number of files accessed
θ	Parameter for the Zipf-like distribution used to model the file access probabilities
N	Client request rate, in units of average number of requests per average file duration T
C	Cache size, expressed as a fraction of the total amount of file data accessed
B	Disk bandwidth, expressed as a fraction of the bandwidth needed to deliver the cached files under LFU

3.1. System Assumptions

To define a broad system and workload parameter space that can be explored efficiently, we make the following assumptions, augmented with additional experiments as described below.

(1) As in previous studies[9,11,12,18,32], we assume a fixed number of files, ***n*,** each access is independent, and fixed access frequencies that are skewed as defined by a Zipf-like distribution with parameter θ (i.e., p_i = P(access file i) = $C/i^{(1-\theta)}$).

(2) In most of the experiments, we assume all files have the same size and delivery rate (r), and that each client requests the entire file, and thus the delivery time, ***T*,** is the same for all files.

(3) As in many previous studies of streaming media caching policies[11,12,15,32], we assume in most of the simulation experiments that client request arrivals are Poisson, with request rate λ.

(4) We assume that files are placed on the disks in a way that balances the load on the disks [19,26].

Zipf-like distributions of file access frequencies have been observed in several prior studies of workloads for videos as well as Web content[3,5,9,14]. As discussed in Section 2.1, practical methods for estimating each file's access frequency dynamically in an actual system are an open research problem. Similarly, to our knowledge, it is unknown how file access frequencies vary over time in streaming media systems. However, results in section 4 will show that for workloads in which LFU outperforms RBC, the RBC policy caches mostly full files. In this case, the assumptions of fixed access frequencies and perfect knowledge of file access frequencies benefit both LFU and RBC equally. Thus, more general assumptions of access frequencies and estimators for file request frequencies are not expected to change the conclusions in this paper.

The assumption of uniform size, and delivery rate for all files greatly simplifies the exploration of the design space and makes our results easier to reproduce in future studies. More importantly, we provide results in Section 4 to show that the relative performance of the policies is the same for the assumption of multiple file sizes and delivery rates as it is for uniform file size and delivery rate equal to the average values of the respective quantities in the more general workload.

The Poisson arrival assumption allows our new results to be directly compared with the previous results[32]. Furthermore, in a previous study, Poisson arrivals were observed for workloads on six different Web servers [3], and we have recently found that the arrivals to two streaming media servers that serve course content at different major U.S. universities, measured over periods during which the request rate is relatively stable, are approximately Poisson. On the other hand, since inter-request times at other Web servers have been observed[4,21,33] to have a heavier tail than the exponential, we performed additional experiments with a heavy-tailed distribution of inter-arrival times (i.e., Pareto with parameters α =1.5, k = 1.0[4]). We found that the conditions under which RBC performs better or worse than LFU remain the same for these heavy tailed inter-arrival times as for the Poisson arrivals. We did not experiment with explicit temporal locality, but prior work has shown that the simple model of independent requests with a Zipf-like distribution of file popularity is enough to capture the temporal behavior observed at web proxies [5], especially for large caches [14,16]. Furthermore, we conjecture that it affects performance similarly to the heavy-tailed distribution of inter-request time (i.e., more of the file intervals that might be cached in the RBC policy are small).

3.2. System and Workload Parameters

The system assumptions defined above reduce the number of parameters that need to be specified to include: the number of files accessed (n), the parameter of the Zipf distribution of file access frequencies (θ), the request arrival rate, the cache size, the uniform file duration (T), the uniform file streaming rate (r), and the total proxy disk bandwidth. In this section, we eliminate the average file duration and average file streaming rate parameters by expressing three of the other parameters in units that account for the effects of file duration and streaming rate.

Specifically, the cache size ***C*** is expressed as a fraction of the total file data, which captures the impact of file duration and streaming rate on cache space. For given values of B and N (defined below), The performance of a given caching policy will remain the same if the uniform file size is doubled and the cache size is also doubled (given that each file represents the unit of data that is requested by the clients). Thus, it is not necessary to vary absolute cache size, file duration, and file streaming rate independently to capture the impact of cache size on policy performance. Instead, the parameter C can be varied between zero and one.

Similarly, the total client request arrival rate, ***N***, is expressed as the average number of client requests that arrive during the average time it takes to stream a file to a single client. For example, N =30 if client requests arrive at the rate of sixty per hour and the uniform file duration is thirty minutes, or if client requests arrive at the rate of twenty per hour and the uniform file duration is ninety minutes. In each of these cases, an average of thirty concurrent streams are needed to deliver the files to the clients. Thus, N can be varied to capture the impact of request arrival rate and file duration on total bandwidth needed to deliver all requested files (measured in units of the streaming rate). Interesting values of N for evaluating caching policy performance range at least from one hundred to several thousand. Lower and higher values of N may also be of interest for particular servers.

Finally, the total disk bandwidth, ***B***, is expressed as a fraction of $r \times S^*$, where ***S**** is the average number of active streams that are needed to deliver the data that is cached by the LFU policy. Given that $\lfloor n \times C \rfloor$ is the index of the least popular file that fits in the LFU cache, S^* is computed from the input parameters as follows:

$$S^* = N \times \sum_{i=1}^{\lfloor n \times C \rfloor} p_i . \tag{1}$$

where p_i is the fraction of client requests that access file i. Given the particular disks that are used for the cache and the uniform file delivery rate, ***r***, the number of concurrent streams each disk can support can be computed using the calculations by Ozden *et al.*[19]. The results of such calculations are shown in Table 5 for three recent generations of disks. The number of concurrent streams per disk times the number of disks divided by S^* is the value of B. Alternatively, since the load is balanced across the disks, the ratio of the number of concurrent streams per disk to the number of concurrent streams needed to deliver all of the content on the disk is also equal to B. For example, if the Seagate disk caches three two hour MPEG-1 (1.5 Mb/s) movies, and the total request rate for the three movies is ten client requests per hour, thirty streams are needed to deliver the data cached on the disk and B = 25/20=1.25. Similarly, if the Ultrastar disk caches eighty-seven thirty minute MPEG-2 (4 Mb/s) videos and the total request rate for the shows is equal to 120 requests per hour, 60 concurrent streams are needed to deliver the cached content and B = 42/60 = 0.7. Note that depending on the duration and delivery rate of the files stored on the disk and the total request rate for the files cached on the disk, the value of B can be less than or greater than 1.0.

We have verified with simulation experiments that caching policy performance is determined by the values of B, C, and N, rather than on the particular values of the more basic system parameters that are used to calculate these values. If B = 1.0, we expect that a proxy that implements the LFU policy has enough bandwidth to be able to serve most requests for the cached files from the cache, and will have bandwidth utilization close to 100%. Thus, we will call systems where B = 1.0 *bandwidth balanced* systems. Similarly, in systems where B < 1.0, we expect that a significant number of client requests that hit in the LFU proxy cache can not be delivered from the cache due to limited disk bandwidth. Such systems will be called *bandwidth deficient* systems. When B is greater than 1.0, in the LFU caching system, bandwidth utilization can be expected to be approximately equal to 1/B, and thus these systems will be called *bandwidth abundant* systems. For the experiments in this paper, we will consider values of B equal to 0.75, 1.0, and 2.0 as representative of these three types of systems.

The results presented in this paper are for experiments that hold θ constant and vary n. The frequency of rentals for various movies in a particular week in video stores follows a Zipf(0.271) distribution[9]. We choose the more conservative value of θ = 0 because the higher skew could favor the interval caching capabilities of RBC as compared with LFU. However, additional experiments have shown that the relative performance was the same for θ = 0.271.

In summary, to explore relative caching policy performance over the entire design space defined in section 3.1, we only need to vary ***N, C, B*** and ***n***. In the Section 4 we will present results that show how RBC and LFU perform as a function of these parameters.

Table 5: Disk Bandwidth for Particular Disk Models

Disk	Disk Space (GB)	File Delivery Rate	Disk Space (hours of video)	Disk Bandwidth (# of concurrent streams)
Seagate ST15150W	4	1.5 Mb/s	6.36	25
Ultrastar 72ZX	73.4	1.5 Mb/s	116.76	108
Ultrastar 72ZX	73.4	4 Mb/s	43.8	42

4. PERFORMANCE COMPARISON: RBC AND FREQUENCY-BASED CACHING

To evaluate the performance of the caching policies considered in this paper, we have implemented a simulator in C that can simulate any of the policies on a synthetic workload that is specified by the parameters n, θ, N, C, and B. The simulator can also simulate synthetic workloads that are specified by the more primitive system parameters given in section 3. The 95% confidence intervals for all results presented in this paper are within ± 2% of the reported values.

Figures 3a-c show the performance of RBC and LFU for three different values of arrival rate N and n=100 files. There are two main conclusions to be drawn from these Figures. First, the relative performance of RBC and LFU is largely insensitive to the arrival rate♣ Figure 4a shows that the relative policy performance is also largely insensitive to the number of files, except in the extreme case where only a small number of files fit in the cache (i.e., C = 10% and n < 40). Relative policy performance is even less sensitive to n for larger values of C (not shown). Thus, it appears that the primary parameters that affect the relative performance of the RBC and LFU policies are B and C. Second, the LFU policy outperforms RBC when disk bandwidth is a limited resource (B = 0.75 or B = 1.0); otherwise RBC slightly outperforms LFU (B = 2.0). Previous work that compares RBC and LFU [32] considers a small set of workloads and cache parameters for which B is greater than 1.0, and concludes that RBC has higher byte hit ratio than LFU. Thus, it appears that our results agree qualitatively with these previous results[32], although we find that when B > 1.0 RBC outperforms LFU by a smaller amount.

To determine the sensitivity of our conclusions to the assumptions of uniform file duration and uniform file streaming rate, we ran experiments with a heterogeneous workload in which files are partitioned into 3 different classes defined by the file length. A separate Zipf distribution is used to model accesses to files of each class. Each class has 100 files and each file has an equal probability of having a playback rate equal to either 1.5 or 4.5 Mbit/s. Class 1 contains 5-minute clips, class 2

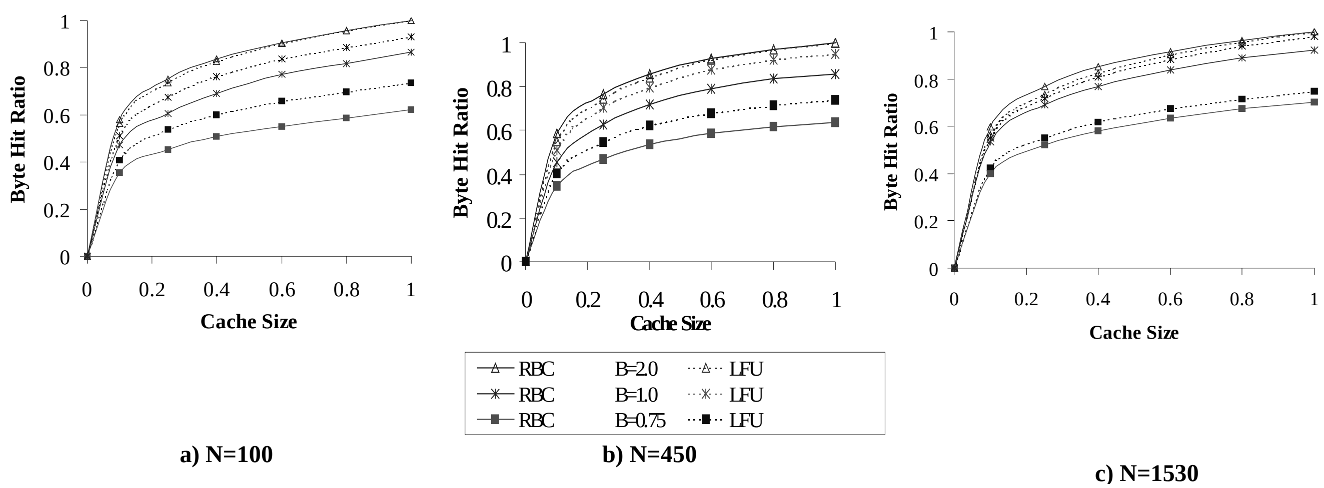
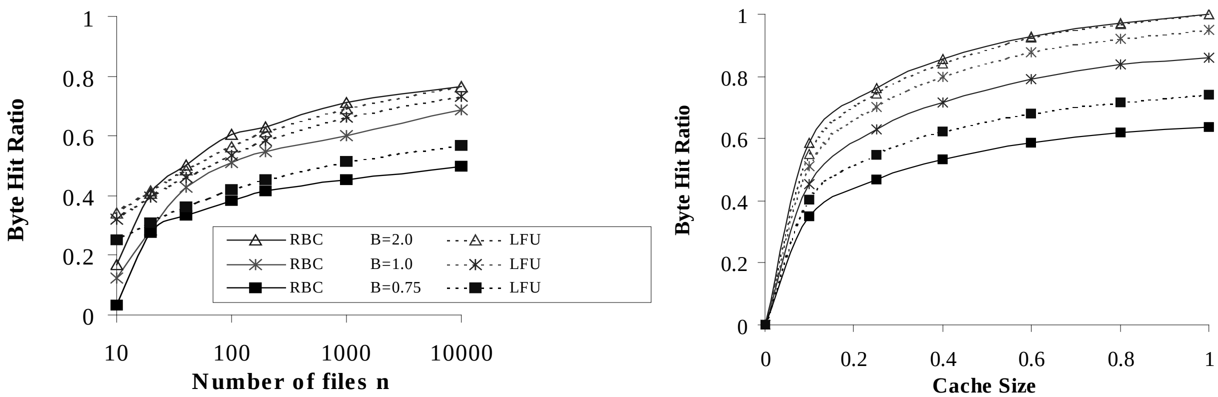


Figure 3: Performance Comparison of RBC and LFU (n = 100)

♣ Generally, the performance of RBC relative to that of LFU improves as the arrival rate increases, which agrees with results in [32], but the same relative ordering of the policies is observed for the various values of N.

contains 30-minute videos and class 3 100-minute videos. The access frequencies for each of the three classes are: 0.6, 0.3 and 0.1, respectively. As shown in Figure 4b, with client request arrival rate equal to 15 requests per minute (N = 330), the relative performance of the RBC policies is as would be expected for n = 300 and N = 330 and uniform file durations and delivery rates. That is, the results in Figure 4b are intermediate between the results in Figures 3a and 3b.

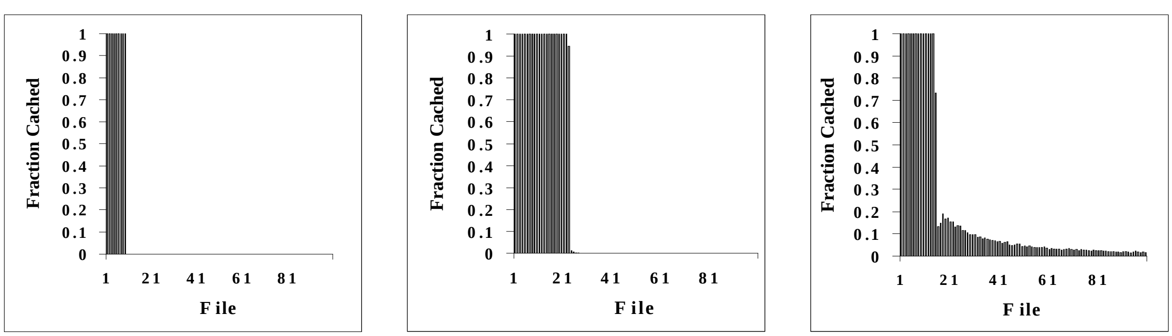


a) Varying number of files n (N=450, C=0.1)

b) Variable file size and delivery rate (N=330, n=300)

Figure 4: Impact of System Parameters on the Relative Performance of RBC and LFU

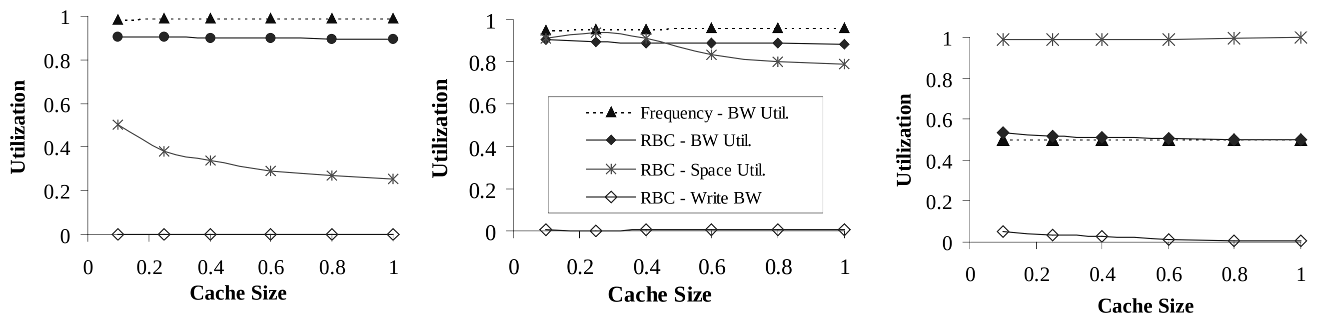
To better understand why RBC does not perform as well as LFU when B = 0.75 and B = 1.0, the average fraction of each file cached by RBC was measured, and is given in Figure 5 for C = 0.25, N = 450 and n = 100. When $B \leq 1.0$ (Figures 5a and 5b), RBC fills the cache with full files, as does LFU. However, as explained in section 2, RBC only caches a file if there is enough unallocated disk bandwidth. As a consequence, when B < 1.0, RBC caches fewer files than LFU and fewer files than the cache can hold. Moreover, RBC does not allow unused bandwidth that is currently allocated to one file to be used to serve a request to another file. Statistical fluctuations in the actual number of concurrent requests for each file implies that allocated bandwidth is often not fully utilized, as will be shown in Figure 6. When more bandwidth is available (Figure 5c), RBC performs better by choosing to cache intervals and runs instead of some of the least popular files[32]. However, it is important to note that RBC still caches the most popular files and only caches intervals for the less popular files. Thus, for B > 1.0, the gap between the RBC and LFU policies is larger for small cache sizes. As C increases, more files are fully cached, and interval caching becomes less effective because short intervals occur more rarely for the less popular files. From additional experiments that vary B from 1 to 2 for a cache size C = 0.1, we found that RBC yields a higher byte hit ratio than LFU only if $B \geq 1.2$, but no further improvements in relative performance are achieved as B increases beyond 1.4. This is explained by the results in Figure 5, since there are insufficient numbers of cacheable intervals to use the extra bandwidth. A higher arrival rate implies that there are more cacheable intervals, and there is less variation in the number of concurrent requests for each hot file, reducing the amount of unused allocated bandwidth when B = 0.75 or B = 1.0. This explains why the performance of RBC relative to LFU improves as arrival rate, N, increases, as was shown in Figure 3.



a) B = 0.75 **b) B= 1.0** **c) B = 2.0**

Figure 5: Average Fraction of Each File Cached by RBC (N = 450, n = 100, C=0.25)

Figure 6 shows the fraction of the available disk bandwidth that is used by RBC and LFU to deliver data to clients. It also shows the space utilization and the disk bandwidth used for writing intervals into the cache for RBC. A key observation is that although RBC tries to equally utilize disk bandwidth and space, the two utilizations are quite different when $B \neq 1.0$. When $B \leq 1.0$, RBC allocates all bandwidth to delivering the full files stored in the cache, but uses only about 90% of the allocated bandwidth to deliver the data, due to the frequent discrepancy between number of active requests per file and amount of bandwidth allocated to each file. When $B = 2.0$, RBC caches a significant number of intervals, which is reflected in a greater use of bandwidth for writing into the cache, but the intervals are a small fraction of the total data delivered to clients because they are for the less popular files. For $B \leq 1.0$, LFU uses nearly 100% of the disk bandwidth to deliver data from the cache. However, LFU can only take advantage of a small amount of excess bandwidth (when $B > 1$), for use when there are more than the average number of concurrent requests for each file.



a) B = 0.75 **b) B = 1.0** **c) B = 2.0**

Figure 6: Space and Bandwidth Utilization for RBC and LFU (N =450, n = 100)

5. IMPROVING RBC AND LFU

5.1. Pooled RBC

We propose a new policy, called **Pooled RBC**, that allows full files to share their bandwidth allocation. Whenever a new full file is added to the cache, its bandwidth allocation is added to the *bandwidth pool*. When a request for a cached file arrives, a new stream from the bandwidth pool (if possible) is assigned to that request. When it finishes delivering the file, the bandwidth is returned to the pool and can be assigned to a new request, even if that request is for a different cached file. Pooled RBC uses the simplified step 1 (section 2.2) to select the granularity of the entity to cache. If a request for a cached file can't get a stream from the bandwidth pool, Pooled RBC forwards the request to the remote server rather trying to evict entities from the cache to free up bandwidth. Otherwise, Pooled RBC operates just like RBC, which in this paper includes the improvement that allows intervals for a file to remain in the cache when a file is evicted (as explained in Section 2.3.2).

Figure 7 compares the performance of Pooled RBC and LFU and should be compared with Figure 3b. When $B = 0.75$ or $B = 1.0$, Pooled RBC performs better than RBC and has a byte hit ratio much closer to the LFU policy. When $B = 2.0$, Pooled RBC performs the same as RBC, because the partitioned bandwidth allocations in RBC do not hurt policy performance when there is excess disk bandwidth. We conclude that Pooled RBC is superior to the original RBC.

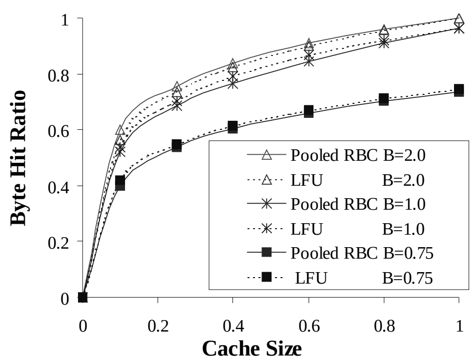


Figure 7: Performance of Pooled RBC and LFU (N=450, n = 100)

5.2. Hybrid Caching Strategies

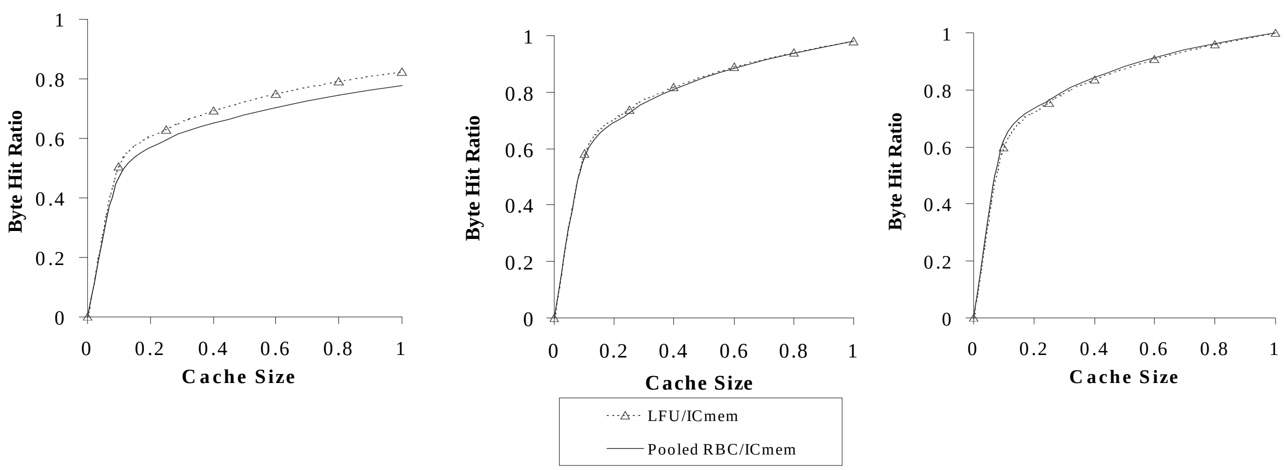
Pooled RBC slightly outperforms LFU if there is excess disk bandwidth, which is used to cache intervals. Since LFU is a much simpler policy to implement, we investigate two hybrid policies designed to improve LFU performance by storing intervals in a small, separate cache, either in main memory or in a small fraction of the available disk space.

5.2.1. Memory Interval Cache

In this section, we investigate how Pooled RBC and LFU perform if the system is augmented with a main memory cache managed by the simple policy that stores the smallest intervals[6]. By caching intervals in memory, we expect that the performance of LFU will improve. Interval caching in memory may also improve Pooled RBC by avoiding the disk writes required to cache intervals on the disk.

We compare two systems, named LFU/IC_{mem} and Pooled RBC/IC_{mem}, respectively. In each system, the primary policy, either LFU or Pooled RBC, is used to manage the disk cache, and simple interval caching is used for the memory cache. In both systems, when a request for a file in the disk cache arrives, it is served from disk if possible; otherwise, either policy tries to cache an interval in memory. If the requested file is not in the disk cache, then either system first checks whether an interval can be cached in memory; otherwise, Pooled RBC checks whether an entity can be cached on the disk, and LFU simply sends the request to the remote server. When an interval is removed from the memory cache, LFU does nothing while Pooled RBC checks whether the interval should be cached on the disk.

Figure 8 compares the performance of LFU/IC_{mem} and Pooled RBC/IC_{mem}, for memory cache size, ***M*,** equal to 0.25% of the total file data. We experimented with *M* up to 8% of the total file data, and the most significant improvement in the performance of each system was obtained for *M* equal to 0.25% of the total data. Memory interval caching improves the performance of both policies, especially when $B < 1.0$. In this case, some of the requests to the most popular files are served from memory when disk bandwidth is fully utilized. Because LFU stores more entire files than Pooled RBC, it uses the memory cache to serve requests to the most popular (cached) files more often than Pooled RBC. As a consequence, the memory cache has a higher byte hit ratio in the system with LFU disk cache. The memory interval cache also improves the performance of both policies when $B \geq 1.0$. Overall, the key conclusion is that the LFU policy performs as well or better than Pooled RBC for all values of *B* and *C*, when interval caching in memory is added to the policy.



a) B = 0.75 **b) B=1.0** **c) B = 2.0**

Figure 8: Performance of Hybrid PooledRBC/IC_{mem} and LFU/IC_{mem} (M=0.25%)

5.2.2. The Hybrid LFU/IC Disk Caching

This section evaluates the performance of LFU if a fraction *s* of the disk space (e.g. s = 5%) is reserved for caching intervals. LFU/IC_{disk} only caches intervals for the files that are *not* in the LFU disk cache. Figure 9 provides results for the LFU/IC_{disk} policy when the disk space *s* reserved for interval caching is equal to 5% of the total disk space. The performance of Pooled RBC and LFU is also given in the graphs. In this case, we find that the LFU/IC_{disk} system performs as well or better than Pooled RBC at all values of *C* and *B*. For $s > 5\%$, there is no significant improvement and the performance may actually degrade. For example, s = 10% hurts performance for cache size $C \geq 0.8$.

We have shown that the hybrid LFU/IC policy, with interval caching either on disk or in memory performs as well or better than Pooled RBC under all values of *C* and *B.* We also note that if a file becomes suddenly very popular, interval caching can be used to serve some requests from the proxy while enough information becomes available to decide to cache the entire file in the LFU disk cache, in an actual system with time varying file access frequencies.

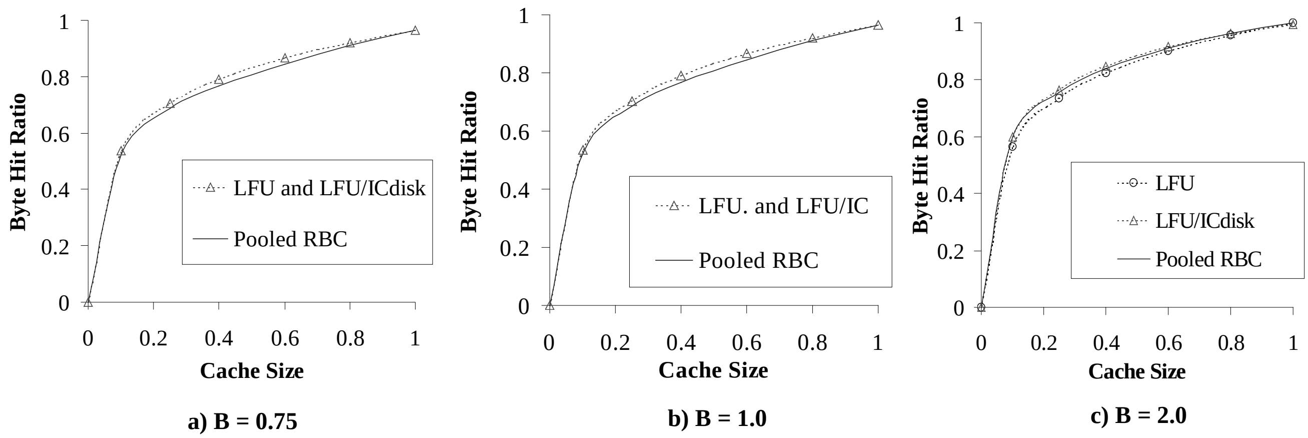


Figure 9: Performance of Hybrid LFU and Interval Caching on Disk - LFU/IC$_{disk}$ (*s* = 5%)

6. CONCLUSIONS

This paper has evaluated several policies for caching video files at proxy servers. In particular, a new comparison of Resource-Based Caching (RBC) and the least-frequently-used (LFU) cache policy was performed over a much broader region of the system design space, including the impact of disk technology trends. This comparison led to new conclusions about the relative performance of these policies. Specifically, RBC only outperforms LFU if the system has excess disk bandwidth and the cache size is small. In other scenarios, LFU performs at least as well as, if not significantly better, than RBC. We also proposed a new policy, Pooled RBC, which contains four improvements to RBC. One of these improvements, the sharing of allocated bandwidth among the cached files results in a significant improvement in policy performance for systems that are bandwidth deficient or bandwidth balanced. However, Pooled RBC and the original RBC share a complex implementation and high time complexity. A much simpler and at least equally effective approach is a hybrid LFU/IC policy with interval caching done in a separate small cache, either in main memory or on disk. Each of the hybrid LFU/IC policies performs as well or better than Pooled RBC for all values of the system and workload parameters examined in this paper.

On-going research includes analyzing client request logs for various video on demand servers, and evaluating caching policies using workload traces from these request logs. These workload traces include an unknown and dynamically changing set of file access frequencies. This research includes investigating alternative practical methods of estimating file (or block) access frequency, and comparing the LFU/IC policy to the proposed LRU-k policy[1]. Another important topic for future research is developing effective dynamic caching policies for streaming media data that is delivered using recently defined multicast merging techniques [11,12,13].

References

1. S. Acharya and B. Smith, “MiddleMan: A Video Caching Proxy Server”, *Proc. 10th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV),* Chapel Hill, NC, June 2000.
2. S. Acharya, B. Smith and P. Parnes, “Characterizing User Access To Videos On The World Wide Web”, *Proc. SPIE/ACM Conf. on Multimedia Computing and Networking*, San Jose, CA, Jan. 2000, pp. 130-141.
3. M. Arlitt and C. Willliamson, “Web Server Workload Characterization: The Search for Invariants”, in *Proc ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems,* Philadelphia, May 1996, pp. 126-137.
4. P. Barford and M. Crovella, “Generating Representative Web Workloads for Network and Server Performance Evaluation”, *Proc. ACM Sigmetrics '98/Performance '98 Joint Int'l. Conf. on Measurement and Modeling of Computer Systems,* Madison, WI, June 1998, pp. 151-160.
5. L. Breslay, P. Cao, L. Fan, G. Phillips, S. Shenker, “Web Caching and Zipf-like Distributions: Evidence and Implications”, *Proc. IEEE Infocom*, New York City, NY, Mar. 1999, pp. 126-134.

6. 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, NY, Jan. 1993.
7. A. Dan and D. Sitaram, "A Generalized Interval Caching Policy for Mixed Interactive and Long Video Environments", *Proc. of Multimedia Computing and Networking Conference*, San Jose, CA, Jan. 1996.
8. A. Dan and D. Sitaram, "Multimedia Caching Strategies for Heterogeneous Application and Server Environments", *Multimedia Tools and Applications*, vol. 4, May 1997, pp. 279-312.
9. A. Dan and D. Sitaram and P. Shahabuddin, "Scheduling Policies for an On-Demand Video Server with Batching", *Proc. of ACM Multimedia*, San Francisco, CA, Oct. 1994, , pp. 15-21.
10. A. Dan, D. Dias, R. Mukherjee, D. Sitaram, and R. Tewari, "Buffering and Caching in Large Scale Multimedia Servers", *Proc. of IEEE COMPCON*, San Francisco, CA, Mar. 1995, pp. 217-224.
11. D. L. Eager, M. C. Ferris and M. K. Vernon, "Optimized Regional Caching for On-Demand Data Delivery", *Proc. IS&T/SPIE Conf. on Multimedia Computing and Networking*, San Jose, CA, Jan. 1999, pp. 301-316.
12. D. L. Eager, M. C. Ferris and M. K. Vernon, "Optimized Caching in Systems with Heterogeneous Client Populations", *Performance Evaluation, Special Issue on Internet Performance Modeling*, Vol. 42, No. 2/3, Sept. 2000, pp. 163-185.
13. D. L. Eager, M. K. Vernon and J. Zahorjan, "Bandwidth Skimming: A Technique for Cost-Effective Video-on-Demand", *Proc. Multimedia Computing and Networking (MMCN)*, San Jose, CA, Jan. 2000, pp. 206-215.
14. S. Jin and A. Bestavros, "Temporal Locality in Web Request Streams: Sources, Characteristics and Caching Implications", Technical Report BUCS-TR-1999-009, Computer Science Department, Boston University, Aug. 1999.
15. M. Kamath, K. Ramamritham and D. Towsley, "Continuous Media Sharing in Multimedia Database Systems", *Proc. International Conference on Database Systems for Advanced Applications*, Singapore, April 1995, pp. 79-86.
16. A. Mahanti, D. Eager and C. Williamson, "Temporal Locality and its Impact on Web Proxy Cache Performance", *Performance Evaluation, Special Issue on Internet Performance Modeling*, Vol. 42, No. 2/3, Sept. 2000, pp. 187-203.
17. Z. Miao and A. Ortega, "Proxy Caching for Efficient Video Services Over the Internet", *Proc. Packet Video Workshop*, Columbia University, NY, April 1999.
18. B. Ozden, R. Rastogi and A. Silberschatz, "Buffer Replacement Algorithms for Multimedia Storage Systems", *Proc. International Conf. on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996, pp. 172-180.
19. B. Ozden, R. Rastogi and A. Silberschatz, "Disk Striping in Video Server Environments", *Proc. International Conf. on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996, pp. 580-589.
20. D. Patterson, "Computers for the Post-PC Era", *Sun Computer Systems SE Symposium*, Burlingame, CA, Feb. 2000.
21. V. Paxson and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Assumption", *IEEE/ACM Transaction on Networking*, Vol. 3, No. 3, June 1995, pp. 226-244.
22. D. Patterson and K. Keeton, "Hardware Technology Trends and Database Opportunities", *Sigmod 98 Keynote Address*, Seattle, WA, June 1998
23. R. Rejaie, M. Handley, H. Yu and D. Estrin, "Proxy Caching Mechanism for Multimedia Playback Streams in the Internet", *Proc. Fourth International WWW Caching Workshop*, pp. 100-111, San Diego, Mar. 1999.
24. R. Rejaie, H. Yu, M. Handley and D. Estrin, "Multimedia Proxy Caching Mechanism for Quality Adaptive Streaming Applications in the Internet", *Proc. of IEEE Infocom*, Tel Aviv, Israel, Mar. 2000.
25. L. Rowe, private communication , Nov. 1999.
26. J. R. Santos, R. Muntz and B. Ribeiro-Neto, "Comparing Random Data Allocation and Data Striping in Multimedia Servers", *Proc. ACM Sigmetrics'2000 Conf. On Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000, pp. 44-55.
27. A. Silberschatz, J. Peterson, P. Galvin, *Operating System Concepts*, Addison Wesley 1992.
28. S. Sen, J. Rexford and D. Towsley, "Proxy Prefix Caching for Multimedia Streams", *Proc. IEEE Infocom'99*, New York, Mar. 1999.
29. R. Tewari, "Architectures and Algorithms for Scalable Wide-Area Information Systems", PhD Thesis, Computer Science Department, University of Texas at Austin, Aug. 1998.
30. R. Tewari, private communication, April 2000.
31. R. Tewari, H. Vin, A. Dan, and D. Sitaram, "Caching in Bandwidth and Space Constrained Hierarchical Hyper-media Servers", Technical Report CS-TR-96-30, Department of Computer Sciences, University of Texas at Austin, Jan. 1997
32. R. Tewari, H. Vin, A. Dan and D. Sitaram, "Resource-based Caching for Web Servers", in *Proc. of SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA, Jan. 1998, , pp. 191-204.
33. W. Willinger, M. Taqqu, R. Sherman and D. Wilson, "Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level", *Proc. ACM SIGCOMM*, Cambridge, MA, Aug. 1995, pp. 100-113.
34. Y. Wang, Z.-L. Zhang, D. Du and D. Su, "A network conscious approach to end-to-end video delivery over wide area networks using proxy servers", *Proc. IEEE Infocom*, San Francisco , CA, Mar. 1998, pp. 660-667.