SRB: Shared Running Buffers in Proxy to Exploit Memory Locality of Multiple Streaming Media Sessions

Songqing Chen[1], Bo Shen[2], Yong Yan, Sujoy Basu[2], and Xiaodong Zhang[1]
[1]Department of Computer Science
The College of William and Mary
Williamsburg, VA 23187, USA
{sqchen,zhang}@cs.wm.edu

[2]Mobile and Media System Lab
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{boshen,basus}@hpl.hp.com

Abstract

With the falling price of the memory, an increasing number of multimedia servers and proxies are now equipped with a large DRAM memory space. Caching media objects in the memory of a proxy helps to reduce network traffic, disk I/O bandwidth requirement, and data delivery latency. The running buffer approach and its alternatives are representative techniques to cache streaming data in the memory. However, there are two limits in the existing techniques. First, although multiple running buffers for the same media object co-exist in a given processing period, data sharing among the multiple buffers is not considered. Second, user access patterns are not insightfully considered in the buffer management. In this paper, we propose two techniques based on shared running buffers (SRB) in the proxy to address these limits. Considering user access patterns and characteristics of the requested media objects, our techniques adaptively allocate memory buffers to fully utilize the currently buffered data of streaming sessions, with the aim to reduce both the server load and the network traffic. Experimentally comparing with several existing techniques, we show that the proposed techniques have achieved significant performance improvement by effectively using the shared running buffers.

1. Introduction

The basic infrastructure of a Web content delivery network is a server-proxy-client system. In this system, the server delivers the content to the client through a proxy. The proxy can choose to cache the object so that subsequent requests to the same object can be served directly from the proxy without contacting the server. Proxy caching strategies have therefore been the focus of many studies. Much work has been done in caching the static Web contents to reduce network load and end-to-end latency.

The delivery of streaming media content presents several challenges: (1) The size of a streaming media object is usually orders of magnitudes larger than traditional text-based Web contents. For example, a two hour MPEG video requires more than 1 GB of disk space, while text-based Web objects are of the magnitude of 10 KB; (2) The demand of continuous and timely delivery of a streaming media object is more rigorous than that of the text-based Web pages. Therefore a lot of resources have to be reserved for delivering the streaming media data to a client. In practice, even a relatively small number of clients can overload a media server, creating bottlenecks by demanding high disk bandwidth on the server and high network bandwidth to the clients.

To address these challenges, researchers have proposed different methods to cache streaming media objects. Partial caching (see e.g. [3], [7], [10], [11], [13], [14]) is a technical method to cache either a prefix or segments of a media object in the disk storage of the proxy.

Researchers have also paid attention to dynamically caching streaming data in the proxy memory to reduce the access latency. The fixed-sized running buffer caching [1] and the interval caching [4], [5] are two representative memory-based running buffer based caching techniques. The basic idea of the running buffer based caching technique is as follows. When a request arrives, a fixed-sized buffer of a predetermined length is allocated in the main memory to cache the media data fetched by the proxy, hoping that subsquent requests could reuse the data in the memory instead of obtaining it from other sources (the disk, the origin server or other caches). In contrast, the interval caching technique uses a different approach. It considers two immediately followed requests as a request pair, and incrementally orders the arrival intervals of all request pairs (the arrival interval of a request pair is defined as the difference in their arrival times). In the memory allocation, the interval caching gives a preference to smaller intervals, ex-

Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)
1063-6927/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY

pecting more requests can be served for a given amount of memory. Figure 1 illustrates the basic ideas of the running buffer caching and the interval caching techniques.

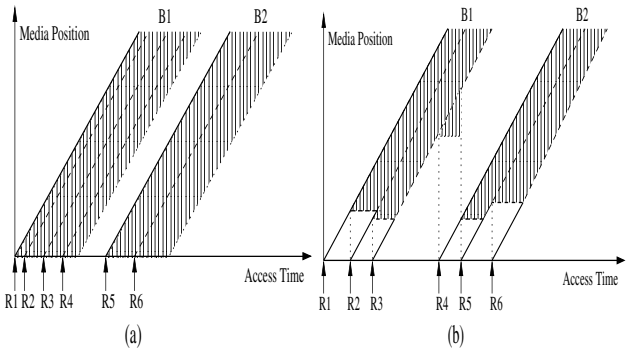

Figure 1. Running Buffer (a) and Interval Caching (b)

In Figure 1, the *Media Position* indicates a time position in a streaming media where the media object is delivered to the client. The solid slope represents a delivery session. In Figure 1(a), a fixed-sized buffer $B1$ is allocated upon the arrival of the request $R1$. Subsequently, requests R2, R3, R4 are served by the data cached in this buffer since they arrive in time. The request R5 does not arrive in time, so a new buffer $B2$ of the same length is allocated, which benefits the request R6.

In Figure 1(b), upon the arrival of $R2$, an interval is formed between $R1$ and $R2$, and a buffer of the interval size is allocated to cache data read by $R1$ from now on. So the request R2 only needs to read the first part of data from the proxy/server while receiving the rest data from the buffer. The situation changes until the arrival of request $R5$, where we assume the interval between $R4$ and $R5$ is smaller than that between $R3$ and $R4$. Since the interval between $R4$ and $R5$ is smaller than that between $R3$ and $R4$, the buffer allocated for the interval between $R3$ and $R4$ is deallocated, and the space is re-allocated to the new interval between $R4$ and $R5$.

However, the running buffer caching does not take consideration of user access patterns, resulting in the inefficient usage of the memory resource. For example, in Figure 1(a), the size of buffer $B1$ is bigger than the amount needed to serve the requests of R_1 through R_4, the size of buffer $B2$ is bigger than the amount needed to serve the request R_5 and the request R_6. The interval caching approach does consider the user access pattern in the buffer allocation. However, it shares another limit with the running buffer caching: data sharing among different buffers has not been considered. For example, $B2$ in Figure 1(b) does not need to run to the end of the media if the data cached in buffer $B1$ are shared by the later requests.

In this paper, we first propose a new memory-based caching algorithm for streaming media objects, called Shared Running Buffers (SRB). In this algorithm, the memory space on the proxy is allocated adaptively based on the user access patterns and the requested media objects themselves. By caching streaming sessions in running buffers, this algorithm dynamically caches media objects in the memory while delivering the data to the client so that the bottleneck of the disk and/or network I/O is relieved. More importantly, the data cached in different running buffers are fully shared, which is different from any of the previous work [1], [4], [5]. This approach is especially useful when requests to streaming objects are highly temporally localized. The algorithm also efficiently reclaims the idle memory space and does near-optimal buffer replacement at runtime when requests are terminated.

By further looking into the patching approaches that were heavily studied in the VOD (Video-On-Demand) environment [6], [8], we found that patching algorithms, such as the greedy patching, the grace patching and the optimal patching [9], take advantage of the client-side storage resource to buffer data in multiple channels without any delay. The greedy patching always patches to the existing complete stream while the grace patching restarts a new complete stream at some appropriate points in time. Furthermore, the optimal patching [12] considers how to reuse the limited storage on the client side to receive as many data as possible while listening to as many channels as possible. Motivated by this, we further propose an efficient media delivering algorithm: Patching SRB (PSRB), which further improves the performance of the media data delivery without the necessity of caching.

Finally, we evaluated our algorithms through a comprehensive set of simulations based on synthetic workloads and a real access trace of an enterprise media server. The simulation results show that the performance of our algorithms is superior to previous solutions.

The rest of the paper is organized as follows. Section 2 describes the memory-based SRB algorithms we proposed. Simulation based performance evaluation results are presented in Section 3. We then make concluding remarks in Section 4.

2. Shared Running Buffers (SRB) Media Caching Algorithm

It has also been shown that two current memory caching approaches of the media objects: the running buffer caching and the interval caching, do not make effective use of the limited memory resource.

IEEE COMPUTER SOCIETY

Motivated by the limits of the current memory buffering approaches, we design a Shared Running Buffer (SRB) based caching algorithm for streaming media with the aim to maximize the memory utilization. In this section, with the introduction of several new concepts, we first describe our basic SRB media caching algorithm in detail. Then, we present an extension to the SRB: Patching SRB (PSRB).

2.1. SRB Related Concepts

The algorithm first considers buffer allocation in a time span T starting from the first request. We denote R_i^j as the j-th request to media object i, and T_i^j as the arrival time of this request. Assume that there are n request arrivals within the time span T and R_i^n is the last request arrived in T. For the convenience of representation without losing precision, T_i^1 is normalized to 0 and T_i^j (where $1 < j \leq n$) is a time relative to T_i^1. Based on the above, the following concepts are defined to capture the characteristics of the user request pattern.

1. *Interval Series*: An interval is defined as the difference in time between two consecutive request arrivals. We denote I_i^j as the j-th interval for object i. An *Interval Series* consists of a group of intervals. Within the time T, if $n = 1$, the interval I_i^1 is defined as ∞; otherwise, it is defined as:

$$I_i^j = \begin{cases} T_i^{j+1} - T_i^j, & \text{if } 1 \leq j < n \\ T - T_i^n, & \text{if } j = n\,. \end{cases} \quad (1)$$

   Since I_i^n represents the time interval between the last request arrival and the end of the investigating time period, it is called as the *Waiting Time*.

2. *Average Request Arrival Interval (ARAI)*: The *ARAI* is defined as $\sum_{k=1}^{n-1} I_i^k / (n-1)$ when $n > 1$. ARAI does not exist when $n = 1$ since it indicates only one request arrival within time frame T and thus we set it as ∞.

For the buffer management, three buffer states and three timing concepts are defined respectively as follows.

1. *Construction State & Start-Time*: When an initial buffer is allocated upon the arrival of a request, the buffer is filled while the request is being served, expecting that the data cached in the buffer could serve the closely followed requests for the same object. The size of the buffer may be adjusted to cache less or more data before it is frozen. Before the freezing happens, the buffer is in the *Construction State*.

   Thus, the *Start-Time* of a buffer B_i^j, the j-th buffer allocated for object i, is defined as the arrival time of the last request before the buffer is frozen. We use S_i^j to denote the *Start-time* of the buffer B_i^j. The requests arriving in a buffer's *Construction State* are called as the *resident requests* of this buffer and the buffer is called as the *resident buffer* of these requests.

2. *Running State & Running-Distance*: After the buffer freezes its size it will serve as a running window of a streaming session and moves along with the streaming session. Therefore, the state of the buffer is called the *Running State*.

   The *Running-Distance* of a buffer is defined as the distance in terms of time between a running buffer and its preceding running buffer. We use D_i^j to denote the *Running-Distance* of B_i^j. Note that for the first buffer allocated to an object i, D_i^1 is equal to the length of object i: L_i. Here, we assume a complete viewing scenario initially. Since we are encouraging the sharing among the buffers, the buffer B_i^j needs only to run to the end point of B_i^{j-1}. Mathematically, we have:

$$D_i^j = \begin{cases} L_i, & \text{if } j = 1 \\ S_i^j - S_i^{j-1}, & \text{if } j > 1. \end{cases} \quad (2)$$

3. *Idle State & End-Time*: When the running window reaches the end of the streaming session, the buffer enters the *Idle State*, which is a transient state that allows the buffer to be reclaimed.

   The *End-Time* of a buffer is defined as the time when a buffer enters *idle state* and is ready to be reclaimed. The *End-Time* of the buffer B_i^j, denoted as E_i^j is defined as:

$$E_i^j = \begin{cases} S_i^j + L_i, & \text{if } j = 1 \\ T_i^{latest} + D_i^j, & \text{if } j > 1. \end{cases} \quad (3)$$

   T_i^{latest} denotes the arrival time of the most recent request to the object i. Here, the T_i^{latest} dynamically changes with the coming of new requests and so does the E_i^j. The detailed updating procedure of E_i^j is described in the following section.

2.2. SRB Algorithm

For an incoming request to the object i, the SRB algorithm works as follows: (1) If the latest running buffer of the object i is caching the prefix of the object i, the request will be served directly from all the existing running buffers of the object. (2) Otherwise, (a) If there is enough memory, a new running buffer of a predetermined size T is allocated. The request will be served from the new running buffer and all existing running buffers of the object i. (b) If there is no enough memory, the SRB buffer replacement algorithm (see 2.2.3) is called to either re-allocate an existing running buffer to the request or serve this request without caching. (3) Update the *End-Time*s of all existing buffers

IEEE COMPUTER SOCIETY

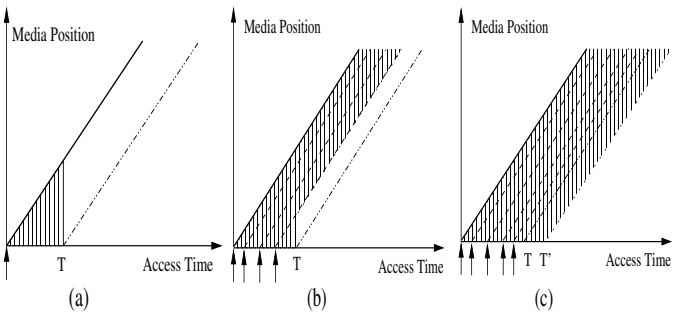

Figure 2. SRB Memory Allocation: the Initial Buffer Freezes its Size

of the object i based on Equation 3. During the process of the SRB algorithm, parts of a running buffer could be dynamically reclaimed as described in Section 2.2.2 due to the request termination and the buffer is dynamically managed based on the user access pattern through a lifecycle of three states as described in Section 2.2.1.

2.2.1. SRB Buffer Lifecycle Management Initially, a running buffer is allocated with a predetermined size of T. Starting from the *Construction State*, it then adjusts its size by going through a three-state lifecycle management process as described in the following.

- Case 1: the buffer is in the *Construction State*. The proxy makes a decision at the end of T as follows.
  - If $ARAI = \infty$, which indicates that there is only one request arrival so far, the initial buffer enters the *Idle State* (case 3) immediately. For this request, the proxy will serve as a bypass server, i.e., the content is passed to the client without caching. This scheme gives preference to more frequently requested objects in the memory allocation. Figure 2(a) illustrates this situation. The shadow indicates the allocated initial buffer, which is reclaimed at T.
  - If $I_n > ARAI$ (I_n is the waiting time), the initial buffer is shrunk to the extent that the most recent request can be served from the buffer. Subsequently, the buffer enters the *Running State* (case 2). This running buffer will serve as a shifting window and run to the end. Figure 2(b) illustrates an example. Part of the initial buffer is reclaimed at the end of T. This scheme performs well for periodically arrived request groups.
  - If $I_n \leq ARAI$, the initial buffer maintains the construction state and continues to grow to the length of T', where $T' = T - I_n + ARAI$, expecting that a new request arrives very soon. At T', the $ARAI^{'}$ and $I_n^{'}$ are recalculated and the above procedure repeats. Eventually, when the request to the object becomes less frequent, the buffer will freeze its size and enter the *Running State* (case 2). In the extreme case, the full length of the media object is cached in the buffer. In this case, the buffer also freezes and enters the running state (a static running). For most frequently accessed objects, this scheme ensures that the requests to these objects are served from the proxy directly. Figure 2(c) illustrates this situation. The initial buffer has been extended beyond the size of T for the first time.

    The buffer expansion is bounded by the available memory in the proxy. When the available memory is exhausted, the buffer freezes its size and enters the running state regardless of future request arrivals.
- Case 2: the buffer is in the *Running State*. After a buffer enters the running state, it has run away from the beginning of the media object and subsequently arrived requests can not be served completely from the running buffer. In this case, a new buffer of an initial size T is allocated and subsequent requests are served from the new buffer as well as its preceding running buffers.

  In addition, the *End-Time* of the new running buffer needs to be determined and the *End-Time*s of its preceding running buffers E_i^{j-1}, ... E_i^2 need to be modified according to the arrival time of the latest request, as shown in Equation 3.

  Figure 3(a) illustrates the maximal data sharing in the SRB algorithm. The requests R_i^n to R_i^{k+1} are served simultaneously from B_i^1 and B_i^2. Late requests could be served from all existing preceding running buffers . **Note that except for the first buffer, the other buffers do not have to run to the end of the object.** When the buffer runs to its *End-Time*, it enters the *Idle State* (case 3).
- Case 3: the buffer is in the *Idle State*. When a buffer enters the *Idle State*, it is ready for reclamation.

In the above algorithm, the time span T (which is the initial buffer size) is determined based on the object length. Typically, a *Scale* factor (say, 1/2 to 1/32) of the origin length is used. To prevent a extremely large or small buffer size, the buffer size is bounded by a upper bound: *High-Bound* and a lower bound: *Low-Bound*. It can be adjusted by the streaming rate to allow the initial buffer to cache a reasonable length (e.g., 1 minute to 10 minutes) of media data.

The algorithm requires the client be able to listen to multiple channels at the same time: once a request is posted, it should be able to receive data from all the ongoing running buffers of that object simultaneously.

IEEE COMPUTER SOCIETY

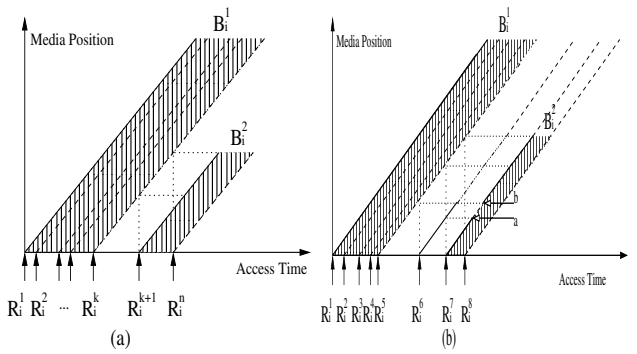

Figure 3. Data Sharing among Buffers in SRB Algorithm (a) and Example of PSRB Algorithm (b)

2.2.2. SRB Buffer Dynamic Reclamation The memory reclamation of a running buffer is triggered by two different types of session terminations: the complete session termination and the premature session termination.

In the complete session termination, a session terminates only when the delivery of the whole media object is completed, which only happens when the buffer is in the *Running State*. In this case, assume that R_i^1 is the first request being served by a running buffer. When R_i^1 reaches the end of the media object, the following two scenarios happen for the *resident buffer* of R_i^1;

1. If the resident buffer is the only running buffer for the media object, the resident buffer enters the idle state. In this state, the buffer maintains its content until all the resident requests reach the end of the session. On that time, the buffer is released.
2. If the resident buffer is not the only running buffer, that is, there are succeeding running buffers, the buffer enters the idle state and maintains its content until its *End-Time*. Note that the *End-Time* may be updated by succeeding running buffers.

The premature session termination is much more complicated. In the premature session termination, the request arriving later may terminate earlier, which can happen when a buffer is in the *Construction State* or the *Running State*. Considering a group of consecutive requests R_i^1 to R_i^n that are being served by a running buffer, the session for one of the requests, say R_i^j, where $1 < j < n$, terminates before everyone else. The situation is handled as follows.

1. If R_i^j is served from the middle of its resident buffer, that is, there are preceding and succeeding requests served from the same running buffer, the resident buffer maintains its current state and the request R_i^j gets deleted from all its associated running buffers.
2. If R_i^j is served from the head of its resident buffer, the request is deleted from all of its associated running buffers. The resident buffer enters the idle state for a time period of I. During this time period, the content within the buffer is moved from R_i^{j+1} to the head. At the end of the time period I, the buffer space from the tail to the last served request is released and the buffer enters the running state again.
3. If R_i^j is served at the tail of a running buffer, two scenarios are further considered.
   - After deleting the R_i^j from the request list of its resident buffer, if the request list is not empty, then do nothing. Alternatively, the algorithm can choose to shrink the buffer to the extent that R_i^{j-1} can still be served from the buffer (assuming R_i^{j-1} is a resident request of the same buffer). In this case, the *End-Time*s of the succeeding running buffers need to be adjusted.
   - If R_i^j is at the tail of the last running buffer, the buffer will be shrunk to the extent that R_i^{j-1} is the last request served from the buffer. R_i^j is deleted from the request list.

2.2.3. SRB Buffer Replacement Policy The replacement policy is important in the sense that the available memory is still scarce compared to the size of video objects. So to efficiently use the limited resources is critical to achieve the best performance gain. In this section, we propose popularity based replacement policies for the SRB media caching algorithm. The basic idea is described as follows:

- When a request arrives while there is no available memory, all the objects that have on-going streams in the memory are ordered according to their popularities calculated in a certain past time period. If the object being demanded has a higher popularity than the least popular object in the memory, then the latest running buffer of the least popular object will be deallocated, and the space is re-allocated to the new request. Those requests without running buffers do not buffer their data at all. In this case, theoretically, they are assumed to have no memory consumption.

We have precisely analyzed our popularity based replacement policies by both the modeling and the simulation in the reference [2], which is omitted due to page limits.

2.3. Patching SRB (PSRB) Media Delivering Algorithm

Since the proxy has a finite amount of memory space, it is possible that the proxy serves as a bypass server to transiently cache concurrent sessions. The SRB algorithm pro-

IEEE COMPUTER SOCIETY

hibits the sharing of such sessions, which may lead to excessive server access when there are intensive request arrivals to many different objects. To solve this problem, the SRB algorithm is extended to a Patching SRB (PSRB) algorithm which enables the sharing of such bypass sessions. It is important to note that PSRB scheme makes the memory-based SRB algorithm work with the memoryless patching algorithm.

Figure 3(b) illustrates a PSRB scenario. The first running buffer B_i^1 has been formed for requests R_i^1 to R_i^5. No buffer is running for R_i^6 since it does not have a close neighboring request. However, a patching session has been started to retrieve the absent prefix in B_i^1 from the content server. At this time, request R_i^6 is served from both the patching session and B_i^1 until the missing prefix is patched. Then, R_i^6 is served from B_i^1 only (the solid line for R_i^6 ends).

When R_i^7 and R_i^8 arrive and form the second running buffer B_i^2, they are served from B_i^1 and B_i^2 as described in the SRB algorithm. In addition, they are also served from the patching session initiated for R_i^6. Note that the patching session for R_i^6 is transient, or we can think of it as a running buffer session with zero buffer size. As evident from the figure, the filling of B_i^2 does not cause server traffic between position a and b (no solid line between a and b) since B_i^2 is filled from the patching session for R_i^6. Sharing the patching session further reduces the the number of server accesses for R_i^7 and R_i^8. In general, the PSRB algorithm is a combination of the SRB algorithm with the optimal patching algorithm proposed in [12]. By using more client-side storage, PSRB tries to maximize the data sharing among concurrent sessions in order to minimize the server-to-proxy traffic.

3. Performance Evaluation

To evaluate the performance of the proposed algorithms and to compare them with prior solutions, we have implemented an event-driven simulator to model a proxy's memory caching behavior. Both synthetic workloads and a real workload extracted from enterprise media server logs are used. However, in the following context, only the performance results based on the real workload are presented. Others are omitted due to page limits. Interested readers can refer to [2].

The real workload, named as REAL, is extracted from HP Corporate Media Solutions, covering the period from April 1 to April 10, 2001. There are a total of 403 objects, and the unique object size accounts to 20G. There are 9000 requests, which run for 916427 seconds, roughly 10 days. Our analysis shows that 83% requests only view the objects for less than 10 minutes and 56% requests only view the objects for less than 10%. Only about 10% requests view the whole objects.

3.1. Evaluation Metrics

Since the *object hit ratio* or *hit ratio* is not suitable for evaluating the caching performance of the streaming media, we use the *server traffic reduction* (shown as "bandwidth reduction" in the figures) to evaluate the performance of the proposed caching algorithms. If the algorithms are employed on a server, this parameter indicates disk I/O traffic reduction.

Using SRB or PSRB algorithms, a client needs to listen to multiple channels for the maximal sharing of the cached data in the proxy's memory. We measure the traffic between the proxy and the client in terms of *average client channel requirement*. This is an averaged number of channels the clients are listening to during the sessions. Since the clients are listening to earlier on-going sessions, storage is needed at the client to buffer the data before its presentation. We use the *average client storage requirement* in percentage of the full size of the media object to indicate the storage requirement on the client side.

If a session terminates before it reaches the end of the requested media object, it is possible that the client has already downloaded future part of the media stream which is no longer needed. To characterize this wasted delivery from proxy to the client, we record *average client waste* during the simulation. It is the percentage of wasted bytes versus the averaged total prefetched data.

The effectiveness of the algorithms is studied by simulating different *scale* factors for the allocation of the initial buffer size and varying memory cache capacities. The streaming rate is assumed to be constant for simplicity. The simulations are conducted on a HP workstation x4000, with 1 GHz CPU and 1 GB memory, running Linux Redhat 7.1.

For each simulation, we compare a set of seven algorithms in three groups. The first group contains buffering schemes which include the running buffer caching and the interval caching. The second group contains patching algorithms, specifically the greedy patching, the grace patching and the optimal patching. The third group contains the two shared running buffer algorithms proposed in this paper.

3.2. Performance Results

First, we evaluate the caching performance with respect to the initial buffer size. With a fixed memory capacity of 1GB, the initial buffer size varies from 1 to 1/32 of the length of media objects. For each *scale* factor, the initial buffer of different sizes is allocated if the length of the media object is different. The *server traffic reduction*, the *average client channel requirement* and the *average client storage requirement* are recorded in the simulation. The results are plotted in Figure 4 and Figure 5.

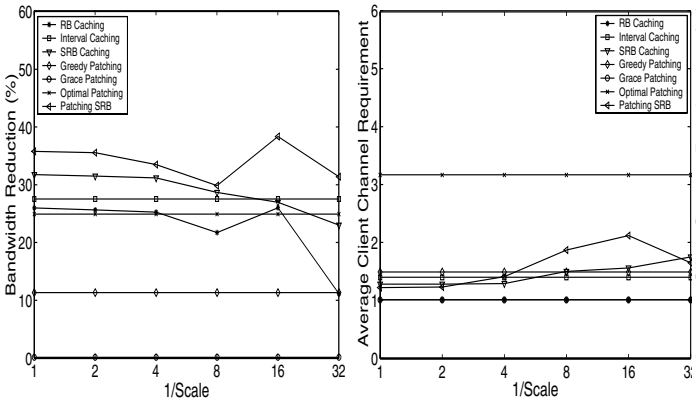

Figure 4. REAL: (a) Bandwidth Reduction and (b) Average Client Channel Requirement with 1GB Memory

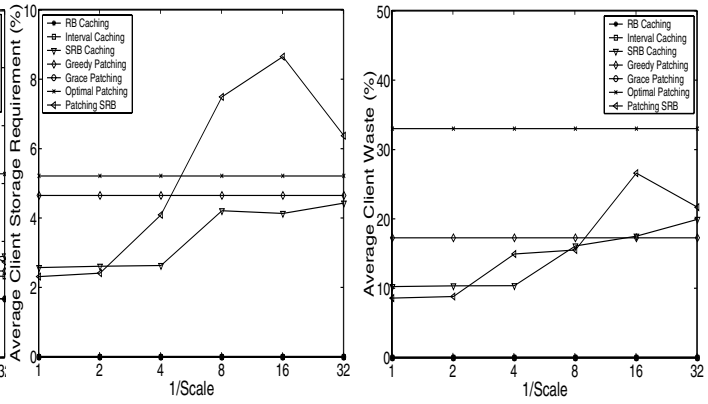

Figure 5. REAL: (a) Average Client Storage Requirement(%) and (b) Client Waste(%) with 1GB Memory

Figure 4(a) shows the server traffic reduction achieved by each algorithm. Notice that PSRB achieves the best traffic reduction. As expected, the performance of the three patching algorithms does not depends on the *scale* factor for allocating the initial buffer. Neither does that of the interval caching since the interval caching allocates buffers based on access intervals. The changes of *scale* factor have a significant impact on the performance of the proposed SRB and PSRB algorithms, especially when the *scale* factor is 1/8, 1/16 or 1/32. This is due to the burst nature of the accesses logged in the workload and the trade-off between the number of running buffers and the sizes of running buffers. A larger buffer implies that more requests can be served from the proxy buffer. However, a larger buffer also indicates that less memory space is left for other requests. This in turn leads to a larger number of server accesses since there is no available memory. On the other hand, a smaller buffer may serve a smaller number of requests but it leaves more memory space for the system to allocate for other requests. Despite of these performance fluctuations, we can still see that PSRB and SRB achieve higher traffic reduction rates from Figure 4. The result concludes that PSRB uses 60% of the client channel to achieve 5% higher traffic reduction than the optimal patching as shown in Figure 4(b).

Figure 5(a) shows the average storage requirement on the client. PSRB allows the session sharing even when memory space is not available. It is therefore expected that PSRB achieves the highest rate of server traffic rate reduction. In the mean time, it also requires the largest client side storage. On the other hand, SRB achieves about 4% less traffic reduction averagely, but the requirement on client channel and storage is significantly lower.

Figure 5(b) shows that the client side wastes for PSRB

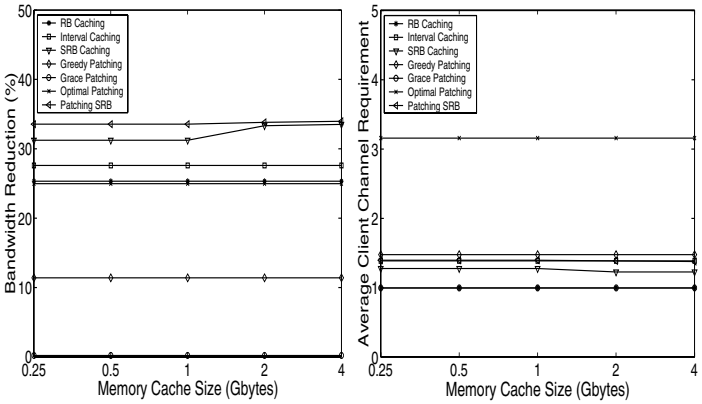

Figure 6. REAL: (a) Bandwidth Reduction and (b) Average Client Channel Requirement with the Scale of 1/4

and SRB are about 33% and 15% respectively, compared to 0% for the interval caching. Since the wasted bytes are not counted as hits, this leads to the lowered traffic reduction rate for PSRB and SRB. From another perspective, even with the waste, PSRB and SRB still can achieve better performance than other techniques.

Setting the initial buffer size as 1/4 of the requested media object, we again evaluate the caching performance with the increasing amount of the available proxy memory. Figure 6 and Figure 7 show the server traffic reduction rate and the client side statistics respectively.

Figure 6(a) shows that distances between the traffic reduction curves of PSRB, SRB and the interval caching become much smaller in general. This reinforces the observation that PSRB and SRB have more waste due to the partial viewing nature. In addition, the grace patching achieving al-

IEEE COMPUTER SOCIETY

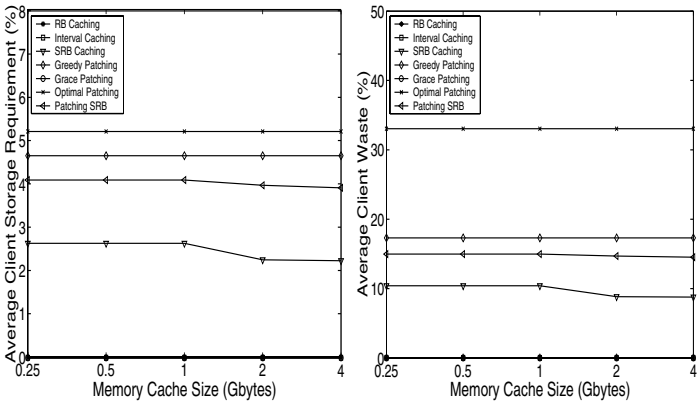

Figure 7. REAL: (a) Average Client Storage Requirement(%) and (b) Client Waste(%) with the Scale of 1/4

most no traffic reduction shows its incapability in dealing with the partial viewing situation.

Figure 6 and Figure 7 also show a flat gain when the memory capacity increases. It seems to indicate that the memory capacity is a minor factor. Once again, the burst nature of the request arrival may play a role here. In addition, the volume of the burst may also be low which leads to the fact that a limited amount of memory space suffices the sharing of sessions.

4. Conclusion

In this paper, we propose two new algorithms for efficiently delivering streaming media objects. Shared Running Buffers (SRB) caching algorithm is proposed to dynamically cache media objects in the proxy memory during the delivery. Patching SRB (PSRB) algorithm is proposed to further enhance the memory utilization in the proxy. Our algorithms can adaptively allocate the memory buffer by considering the user access pattern, and enable the data being fully shared among different buffers. Extensive simulations are conducted to evaluate these algorithms. The simulation results demonstrate the efficiency achieved by the proposed algorithms. Both algorithms require the client capable of listening to multiple channels at the same time. Compared with previous solutions which also require multiple client channels, the proposed algorithms achieve higher server traffic reduction rate with less or similar load on the link between the proxy and the client.

Acknowledgments We appreciate the constructive comments from the anonymous referees. We also thank Mitch Trott and Susie Wee of HP Laboratories for their comments and suggestions on this work.

References

[1] E. Bommaiah, K. Guo, M. Hofmann, and S. Paul. Design and implementation of a caching system for streaming media over the internet. In *Proceedings of IEEE Real Time Technology and Applications Symposium*, May 2000.

[2] S. Chen, B. Shen, S. Basu, and Y. Yan. Srb:the shared running buffer based proxy caching of streaming sessions. In *Hewlett-Packard Laboratories Tech. Report*, 2003.

[3] S. Chen, B. Shen, S. Wee, and X. Zhang. Adaptive and lazy segmentation based proxy caching for streaming media delivery. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 2003.

[4] A. Dan and D. Sitaram. Buffer management policy for an on-demand video server. In *IBM Research Report 19347*, 1993.

[5] A. Dan and D. Sitaram. A generalized interval caching policy for mixed interactive and long video workloads. In *Proceedings of Multimedia Computing and Networking*, Jan. 1996.

[6] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proceedings of ACM Multimedia*, Oct. 1994.

[7] D. L. Eager, M. C. Ferris, and M. K. Vernon. Optimized regional caching for on-demand data delivery. In *Proceedings of Multimedia Computing and Networking*, Jan. 1999.

[8] L. Gao and D. Towsley. Supplying instantaneous video-on-demand services using controlled multicast. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, June, 1999.

[9] K. A. Hua, Y. Cai, and S. Sheu. Patching : A multicast technique for true video-on-demand services. In *Proceedings of ACM Multimedia*, Sept. 1998.

[10] S. Lee, W. Ma, and B. Shen. An interactive video delivery and caching system using video summarization. In *Computer Communications*, volume 25, pages 424–435, March 2002.

[11] R. Rejaie, M. Handley, H. Yu, and D. Estrin. Proxy caching mechanism for multimedia playback streams in the internet. In *Proceedings of Web Caching Workshop*, March 1999.

[12] S. Sen, L. Gao, J. Rexford, and D. Towsley. Optimal patching schemes for efficient multimedia streaming. In *Proceedings of ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 1999.

[13] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proceedings of IEEE INFOCOM*, March 1999.

[14] K. Wu, P. S. Yu, and J. Wolf. Segment-based proxy caching of multimedia streams. In *Proceedings of WWW*, Sept. 2001.

IEEE COMPUTER SOCIETY