
MULTI-POOL CACHING IN CONTINUOUS MEDIA SERVER

CHUTIMET SRINILTA
Department of Computer Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, Thailand
E-mail: chutinet@eee.kmitl.ac.th

ALOK CHOUDHARY
Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208, USA
E-mail: choudhary@ece.nwu.edu

Efficient storage and retrieval strategies are essential to ensure a smooth and meaningful flow of information during a playback of continuous media object. Disk I/O is usually a potential bottleneck limiting capacity of a server. The number of concurrent sessions supported is usually limited by the bandwidth of a storage system. User request distribution has a direct effect on I/O load. Hence, one way to reduce number of I/O access is to reuse data that have already been fetched. Many possibilities exist for an efficient caching. A multi-pool interval cache stores data equivalent to intervals between successive playback sessions in object pools. Cache space is allocated to each object based on its access frequency. Therefore, multi-pool caching is dynamic. Cache access efficiency is controlled by two caching parameters. These parameters can be adjusted to suit different user request distributions and object profiles.

1 Introduction

A continuous media object can be viewed as a sequence of equal size media units. One media unit may represent several physical data blocks. Each media unit is called a file unit. A delivery of an object is called a session. One of the techniques to avoid multiple retrievals for user requests that request the same object is to group user requests and send multicast data [3, 7, 9]. With this technique, some users may experience long startup delays. Another technique is to reuse the data. Sheu et al. [8] proposed a chaining technique where data are cached at client stations in the chaining tree. These stations help serve user requests; thus, traffic to and from server is reduced. Work done relevant to media caching was included in [1, 4, 5, 6].

It is a characteristic of continuous media object that a sequence of data is delivered in response to a playback request. Multiple playback sessions of the same object require the same sequence of data. This characteristic supports caching because it is certain that the cached data are used in the future when one or more users initiate the playback requests of that same object. The concept of interval caching proposed by Den et al. [2] arose from this characteristic. Interval caching also takes into account the fact that successive playback sessions of the same object
always follow one another with fixed intervals in between. Figure 1 illustrates the situation. Suppose there are two user requests for object A. Let the second user request \( A_2 \) arrive at the server when the first user request \( A_1 \) is reading a stripe unit number 4. When \( A_1 \) is at a stripe unit number 6, \( A_2 \) is reading a stripe unit number 3. Since these two sessions are playing the same object, their pages are the same. Therefore, they are always three stripe units apart.

Interval caching keeps only the data in the interval between successive sessions in the cache. It is simple and efficient for playback operation. From Figure 1, if there are no other user requests for object A following \( A_2 \), stripe units are no longer needed after they are read by \( A_2 \). Hence, keeping just four stripe units in the cache is sufficient for these two sessions to share the data. By caching only the intervals, cache space is utilized more effectively. Moreover, the cache adapts itself dynamically to user request distribution as request interval is related to access frequency of object. This same concept is applied in the proposed caching scheme.

2 The Server Model

The server model is implemented on Intel Paragon parallel computer. It is composed of 16 compute nodes arranged in a pattern of 4x4 matrix. All server nodes are independent but connected. Communication between nodes is achieved by message passing.

Server nodes are categorized into three categories: manager node, dispatcher node and storage node. There are 1 manager node, 4 dispatcher nodes and 11 storage nodes in the model. Each dispatcher node has a multi-pool interval cache. Objects are striped and stored across all storage nodes. Manager node receives requests for object playbacks from users. It assigns each user request to one of the dispatcher nodes. The goal of the assignment is to maximize data sharing in the cache: user requests for the same object are assigned to the same dispatcher node. In case that the object is requested for the first time, the user request is directed to a dispatcher node handling the least number of sessions. Dispatcher node periodically sends sub-requests to groups of storage nodes to obtain object data. These data are stored in the cache and are delivered to each user according to a playback rate of the object.

The cache is implemented at dispatcher nodes in order to reduce number of sub-requests sent to storage nodes which consequently reduces both workload at storage nodes and network congestion within the server.

3 Cache Organization

A multi-pool interval cache can be viewed as a collection of equal size memory chunks. Each chunk is equivalent to one stripe unit. The cache organization is illustrated in Figure 2.

The cache is composed of a free pool, a number of object pools and a set of cache entries. The free pool contains a number of stripe units. These stripe units are available to any session. Each cache entry has a pointer to an object pool. An object pool is a circular linked list of stripe units being accessed by sessions. One object pool holds stripe units of one object. In addition to a pointer to object pool, each cache entry contains object information such as object identification number and number of stripe units contained in the object pool.

4 Cache Management

A caching window (CW) represents the allowable number of stripe units in the interval between any two successive sessions accessing the same object. While it is efficient when frequently accessed data are cached, it is even more efficient when those data occupy little cache space. The goal is to keep as little amount of data as possible while serving as many sessions as possible. There is a bound on the caching window. The value for caching window should be given based on frequency of access of object and total cache size. Different objects may have different caching windows. Caching window simply is the length of an object pool (in terms of stripe units)
When one object is requested by many users, these users are to share the same object pool only if their requests enter the server within an interval of the caching window assigned for the object. Each user request sharing the object pool is classified as either an initial or a duplicate request, depending upon the order in which it arrives at the server. The initial request is the first user request that requests the object, the duplicate requests are the subsequent ones. Every object pool has only one initial request, but may have more than one duplicate request. Initial request writes data to object pool and duplicate request reads data from it.

In Figure 3, let \( A \) be a user request for the \( n \)th playback session of an object. User requests reach the server at times indicated by shaded boxes on a time line. Suppose object \( A \) is granted a caching window of size three stripe units. Hence, the object pool contains at most three stripe units at all times. In the figure, the numbers in object pools identify stripe units that are stored in the pools. \( t_{mn} \) represents the time when the \( n \)th stripe unit is placed in object pool \( M \).

An object pool is created when an initial request arrives at the server. After a caching window is determined, \( CW \) stripe units are taken from the free pool to form a new object pool. In case that there is not enough stripe units left in the free pool, the new object pool is not setup.

An object pool is either open or closed at any given time. At the beginning, the object pool is open. Every stripe unit in the pool is empty. The content of these empty stripe units are filled one by one at times indicated by \( t_{mn} \)'s in Figure 3. After every empty stripe unit in the object pool is filled, just before the initial request writes the content of the next stripe unit (at time \( t_{14} \) in Figure 3), the object pool is examined to see if any stripe units can be returned to the free pool. Such stripe units are those stripe units with the most number of access. The reason for this is that those stripe units will not be accessed after the object pool is closed. In Figure 3, \( A_1 \), \( A_2 \) and \( A_3 \) are all finished with stripe units number 1 and 2 by the time \( t_{14} \). These stripe units are no longer used by either \( A_1 \), \( A_2 \) nor \( A_3 \) after \( t_{14} \). Stripe unit number 1 is made available and returned to the free pool. The new stripe unit (stripe unit number 4) replaces stripe unit number 2. Object pool 1 is now closed. From then on, object pool 1 contains only two stripe units. A closed object pool holds only stripe units equivalent to the gap between the initial session and the last session reading from the object pool at the closing time. After the object pool is closed, a newly fetched stripe unit replaces the oldest one. It can be viewed as the caching window is shifting toward the end of the object one stripe unit at a time.

If another user request for that same object arrives after the object pool is closed and there is enough cache space available, a new object pool is allocated. The new object pool will capture a new set of user requests that arrive within the time when the object pool is being filled up.
In Figure 3, all user requests are for object A. $A_1$ is an initial request of object pool 1. $A_2$ and $A_3$ are duplicate requests of object pool 1 because stripe unit number 1 is found in object pool 1 at times when $A_2$ and $A_3$ enter the server. Object pool 1 is closed at time $t_{D_1}$. $A_4$ arrives the server after that. $A_4$ does not find stripe unit number 1 in object pool 1; therefore, it cannot retrieve data from object pool 1. A new object pool, object pool 2, is created and $A_4$ becomes its initial request. $A_5$, $A_6$, and $A_7$ are duplicate requests of object pool 2. Both object pools contain data of object A.

5 Object Pool Allocation

Consider only sessions of the same object, all sessions starting within one caching window interval from the beginning of the initial session are handled by the same object pool. Therefore, a large caching window has a tendency to serve more user requests. However, it requires more cache space. A caching window should be set to the value just long enough so that if an object has not been accessed by any other session during the first caching window interval, it is justified not to store it in the cache.

As stated earlier, a caching window should be bounded to a value based on several factors including frequency of access of an object and total cache size. Consider a database of $M$ objects. The manager node reschedules user requests. It releases them to dispatcher nodes at rate $a_i$ user requests per second. Let a user choose to view object $i$ with a probability $p_i$, $0 \leq p_i \leq 1$, where $\sum_{i=1}^{M} p_i = 1$. These probabilities reflect access probability of each object. Caching window can be determined from the following equation.

$$CW_i = \text{rounded}(C \times p_i) \times \varepsilon$$

where,

- $CW_i$ is caching window of an object pool containing object $i$,
- $C$ is total cache size in terms of number of stripe units, and
- $\varepsilon$ is expansion factor.

The value for a caching window is rounded to the closest integer. An expansion factor is a caching parameter used to extend object pool length. A short object pool is not very effective, especially when object's playback rate is high because object pool duration is short. The object pool is closed in a short time after it is created. As a result, it can capture only a small number of user requests. However, long object pools are not always very efficient. If a pool is long but is accessed by a small number of user requests, it may be better to obtain the data directly from the disk($\varepsilon$), saving cache space for more popular objects. Note that object pool replacement can be considered. However, the replacement process does add load on the server nodes. Moreover, each active object pool is currently providing some benefits. We choose objects to be stored in the cache rather than store every object and replace them later.

6 Minimum Access Probability

A multi-pool caching scheme is more effective when the difference in arrival times of user requests to the cached object is small. In other words, the scheme works well when access frequency of the cached object is high. Therefore, the cache should select to store only objects that have a tendency to be requested frequently. A threshold for the minimum access probability of a tentatively cached object is defined. Any object whose probability of access is lower than this threshold value is not cached. This threshold value depends on a number of parameters that are discussed later in the section.

6.1 Frequency of Access

The frequency of user requests for a particular object assigned to a particular dispatcher node ($f_{dim}$) depends on three factors: request arrival rate, object's access probability, and request assignment scheme. Let $a_i$ be an arrival rate at the manager node of user requests for object $i$, and $p_{im}$ be a probability that the manager node directs user request for object $i$ to dispatcher node $m$. $f_{dim}$ can thus be obtained as below.

$$f_{dim} = a_i \times p_{im}$$

$$f_{dim} = a_i \times a_j \times p_{im}$$

$$f_{dim} = a_j \times p_{im}$$

In our model, $p_{im}$ is equal to 1 because user requests for the same object are always assigned to the same dispatcher node. Thus, the access frequency depends only on the request arrival rate and the probability of access of the object. Equation 2 then becomes

$$f_{dim} = a_i \times p_{i}.$$
6.2 Object Pool Duration

A duration of an object pool is governed by object pool length, size of each stripe unit, and playback rate of object. Duration of object pool \( n \) containing object \( i \) \( (d_n) \) is determined from the following equation.

\[
d_n = \frac{CW_i \times P_i}{R_{pl_i}}
\]  

(4)

6.3 The Bound

Content of an object pool is overwritten by newly fetched stripe units after the pool is closed. Thus, it is reasonable to cache object only when it is requested at least twice over a time interval equivalent to one pool duration. This draws a lower bound on object's access probability.

\[
f_{d_{in}} \times d_n \geq 2
\]

\[
f_{d_{in}} \geq \frac{2}{d_n}
\]

(5)

Equation 5 can be generalized to

\[
f_{d_{in}} \geq \frac{eff}{d_n}
\]

(6)

where,

\( \text{eff} \) is a degree of efficiency.

A degree of efficiency is another caching parameter. It indicates a minimum allowable access efficiency of each object pool. It defines object selection criterion that determines whether an object should be stored in the cache. In other words, degree of efficiency (eff) is a minimum number of user requests expected to access each object pool. If object's access probability does not provide that the object is to be requested more than eff times in one pool duration, the object is not cached. This criterion must be exercised especially when object pool replacement is not considered. In case that the value of eff is too high, not many objects are cached. Thus, the I/O load does not reduce much. On the other hand, if the value is too low, unpopular objects are cached leaving inadequate available cache space for popular objects that are requested later.

A minimum access probability for a tentatively cached object is obtained from equations 3 and 6. The final equation is shown below.

\[
p_{i} \geq \frac{eff}{a_r \times d_n}
\]

(7)

Access probability of an object must satisfy equation 7 in order for the object to be cached.

7 Experiments

A number of test cases were conducted to demonstrate server behaviors. The size of a stripe unit was set to 512KB. Cache size at each dispatcher node was 20MB. User request distribution followed the distribution in Figure 4 which conformed to Zipfian distribution. There are 400 playback sessions at a steady state.

![User Request Distribution](image)

Figure 4. User request distribution

7.1 Assumptions

It was assumed that all objects belonged to a single media type: video. There were a total of 100 distinct objects in a video library. Each object had a constant playback rate of either 1.5Mbits/s (MPEG-1 standard) or 4Mbits/s (MPEG-2 standard). All objects had the same display length. They were stored at storage nodes in a compressed digital form. Compression issues were not addressed. Every object had the same storage and retrieval parameters.
Each server node was assumed to have enough disk capacity and main memory for all the playback sessions the server was serving. Disk access operations were simulated. Disk seek time and rotational latency were given using a random number generator. Disk transfer time varied according to the size of a stripe unit. Total disk service time was simulated by elapsing a system timer. Communication network between the server model and the user was assumed to be reliable and fast enough to handle the bandwidth required for object playback. Lastly, data decompression was carried out at users' sites. Communication from server to user and decompression issues were not covered in the study.

7.2 Performance Metrics

Performance metrics used in the experiment are discussed below.

- **Percentage of cache sub-requests**: This metric is a ratio of the number of sub-requests satisfied from the cache to the total number of sub-requests. This metric shows the workload taken by the cache.
- **Percentage of read access**: This metric is a proportion of read access in total cache access. This metric shows cache access efficiency.
- **Percentage of unused cache space**: This metric represents a portion of available cache space when the server is operating at a steady state. It reflects cache space utilization.

7.3 Experimental Results and Discussion

Experiments were conducted under five object profiles: 100:0, 70:30, 50:50, 30:70 and 0:100. The name of the profile indicates the mixed ratio of playback rates of objects stored at the server. For example, a 70:30 profile means that 70% of the objects the server possesses are MPEG-1 video and 30% are MPEG-2 video.

Object pool length was varied. The value of expansion factor increased from 1 to 8. Degree of efficiency was used in test cases where eff was set to 2 which means that every cached object are requested at least twice in a time interval equivalent to one pool duration.

Results are presented in Figures 5-7. Percentage of cache sub-requests is shown in Figure 5. Percentage of unused cached space is shown in Figure 6. Percentage of cache read access is shown in Figure 7. The top row in each figure (labeled a) presents results when degree of efficiency was not used. The bottom row in each figure (labeled b) presents results when degree of efficiency was set to 2.
Figure 6. Unused cache space

Figure 7. Cache access efficiency
In general, when degree of efficiency was not used, percentage of cache sub-requests decreased as expansion factor increased (Figure 5a). The percentage dropped 5-8% as expansion factor increased from 1 to 8.

When degree of efficiency was used, the trend of the graphs was different (Figure 5b). The graphs increased and then decreased. The peaks of the graphs were reached at a smaller expansion factor when object profile contained more of fast objects. For example, the highest percentages were reached at expansion factor of 3 in 0:100 and 30:70 profiles, where the highest percentages were reached at expansion factor of 6 in 100:0 and 70:30 profiles.

Expansion factor extended the length of object pools. Increasing expansion factor enabled each object pool to carry more stripe units. Consequently, object pool duration was longer and more sub-requests were addressed from the cache. Thus, when cache space is available (more than 20%) at a steady state, increasing expansion factor within that range increased number of sub-requests satisfied from the cache. However, when the cache was completely or nearly full at a steady state, increasing expansion factor further had a tradeoff. The unfavorable effect was that increasing expansion factor decreased the total number of object pools. The total number of cache reads might be less because fewer objects were cached although each object pool was able to address a greater number of sub-requests. This disadvantage of having fewer object pools began to override the advantage of long object pool at the value of expansion factor where the graphs started to convex down.

The percentage of cache read access reflected the effectiveness of cache access. Each object pool should serve at least one duplicate object. Since there was always one user writing to the pool, the acceptable proportion of cache read access in total cache access should be at least 50%.

Cache access efficiency is determined from Figure 7. The access efficiency was higher when degree of efficiency was used. This was because the cache selected to store only frequently requested objects. Cache read access contributed 70-90% when degree of efficiency was set to 2 (Figure 7b) compared to 40-70% in Figure 7a.

The number of cache read access was dependent on the proportion of slow objects in object profile. This again was an effect of pool duration. Percentage of cache read access was higher when object profile was dominated by slow objects.

Expansion factor did not produce a significant effect on the percentage of cache read access. The percentage varied only a little when expansion factor increased from 1 to 8. This happened because increasing expansion factor affected not only the number of cache read access, but also the total number of cache access. Therefore, the ratio of both numbers did not vary greatly upon the changes of the expansion factor.

8 Conclusion

Multi-pool caching was effective under various object profiles. In the experiment, up to almost 40% of total sub-requests were addressed from the cache. The two caching parameters, expansion factor and degree of efficiency, had an effect on workload taken by the cache as well as cache access efficiency. Expansion factor helped the cache capture more sub-requests and degree of efficiency made cache access more effective. These parameters are to be adjusted to fit the system. Since cache access efficiency was independent of the expansion factor, the expansion factor should be set to the value within the range where percentage of cached sub-requests is at the maximum. Object profile possessed by the server is another factor. It should be taken into account in selecting the value of expansion factor as well. The results showed that with a proper setting, the cache was utilized efficiently and the server had a tendency to support more concurrent sessions.

References

IMMEDIATE AND ADVANCE RESOURCE RESERVATIONS ARCHITECTURE WITH QUALITY OF SERVICE ROUTING AGENTS

FARAG SALLABI, AHMED KARMOUCH

Multimedia Information and Mobile Agents Research Laboratory
School of Information Technology and Engineering, University of Ottawa
E-mail: sallabi@slg.eneq.uottawa.ca
karmouch@siit.uottawa.ca

We present an architecture for end-to-end resource reservations with user interactions. This architecture is based on domain agents, where in each network domain there is a domain agent. The domain agent is responsible of making immediate and advance resource reservations, and adapting the reserved resources according to user interactions. We also present an architecture for domain agents that facilitate the communications of domain agents with the receivers, the senders and the routers. It contains a QoS routing agent, to construct a QoS intra-domain path. The QoS routing agent uses a modification of Dijkstra's algorithm to find the narrowest-path with the minimum number of hops that accommodate the requested QoS. We also implemented a simulation model to monitor the rejection probability of the immediate and advance requests, and to monitor the utilization of each link in the simulated network topology.

1. Introduction

The present architecture of the Internet provides a very simple service, the best-effort service. Using this service, packets compete for the network resources, and traverse the network according to the service of the underlying routing protocol algorithm. Packets have no right to choose their paths and even ask for paths that satisfy their QoS needs. This service is not suitable for continuous, real-time traffic that has stringent quality of service requirements. Therefore, new service architectures have come into existence to provide solutions to the QoS problem. The most obvious solutions are to control the number of flows injected into the network and to enable applications reserve network resources according to their QoS demands.

The first architectures for resource reservations suggest that applications could ask for reservations that start immediately and continue for indefinite periods of time [13]. These architectures also face the problem of the scarce of network resources, where most resource reservation requests are rejected. To overcome this problem, new schemes have been proposed to enable applications reserve resources in advance [6,11,12,14]. In fact, these schemes does not explicitly solve the problem of scarcity, but it enable users choose the periods of time in which resources are available. Once the resources are reserved, they will be available to users at the start of their applications. Meanwhile, advance reservations give chance to service