Performance Enhancement Using Intra-server Caching in a Continuous Media Server

Chutimet Srinilta

EECS Department
Syracuse University
Syracuse, NY 13244

Alok Choudhary

ECE Department
Northwestern University
Evanston, IL 60208

Abstract

Continuity of stream playback is the crucial constraint in designing a continuous media server. From a distributed memory architectural model developed earlier, we found that there were many points where the stream capacity of the server could be improved. The stream capacity was usually limited by the storage bottlenecks. Serving streams from memory cache eliminates disk accesses and data transfers between nodes which, in turn, helps relieve those bottlenecks. However, the capacity of the server ultimately depends on client access pattern. Client request assignment has an impact on cache hit ratio as well as workload distribution. It is also the major factor reflecting the server performance. In some cases where lower playback quality at some points in time is acceptable, delays can be added to improve the missed packet rate. This paper proposes various techniques to enhance the server performance and shows their reflections under different circumstances. Some techniques work well together. The best combination improves the capacity of the server by approximately 30%.

1 Introduction

Continuous media data such as audio, video and images, are found to be useful in a wide range of applications in many areas including education, medicine, sports, entertainment and space research. The key operation in such applications is the playback operation.

In general, a multimedia information system is composed of three components which are clients, communication network and multimedia server. The main responsibility of such server is to supply multimedia data to the clients according to their pace and time desired.

Continuous media data cardinally differ from tradition text data in characteristics. As a consequence, their storage, retrieval and transportation have to be handled differently. Among all the operating constraints, continuity is the most important. The concept of *interval caching* [1] where the temporal locality between streams accessing the same object is taken into account is the backbone of the proposed caching mechanism. In addition to the cache, this paper discusses other techniques to help improve the number of concurrent streams served while maintaining low number of missed packets.

The rest of the paper is organized as follows. Section 2 concisely describes the architecture of the server model and its data storage, retrieval and buffering strategies. Techniques to improve server performance are discussed in section 3. Section 4 outlines the performance metrics and experiment parameters. Section 5 presents and discusses the results. Section 6 concludes the paper.

2 The Server Model

2.1 Server Configuration

Our server model [3] consists of a group of computing nodes connected by an interconnection network. These nodes are independent. Each node is equipped with a processing unit, main memory and secondary storage device. These memory and storage device are considered private. They are not to be shared between any two nodes. Communication between the nodes is achieved by passing message through the interconnection network. This architecture is commonly known as a shared-nothing architecture.

The server model is independent of the architectural implementation. It can be implemented on a distributed memory parallel machine or a collection of personal computers or workstations interconnected by high speed links.

Base on the functionalities, the server nodes can be categorized into three logical types: manager node, dispenser node and storage node. The server nodes

can be configured in many ways. *Two-level* configuration (figure 1) where each physical node serves as one of the three node types is used in the experiments. One node performs as a manager node. Each of the rest is either a dispenser node or a storage node. The lines between any two nodes represent data transfer. The arrows indicate the transfer direction. The solid lines represents periodic activities, whereas the dotted lines represent one-time activities. The numbers labeled on the lines denote a sequence of activities during a typical stream playback operation.

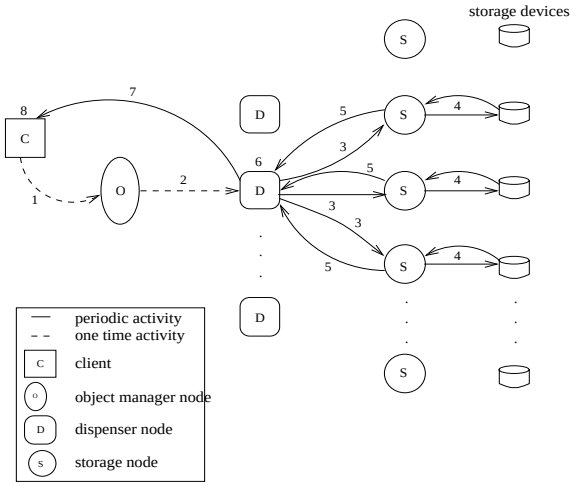

Figure 1: Logical view of the *two-level* configuration

Activities during a typical stream playback operation are briefly illustrated as follows. A client request enters the server at a manager node. If the available resources are sufficient for a stream playback, that client request is directed to one of the dispenser nodes. That dispenser node then periodically releases sub-requests to storage nodes (or to its cache) to obtain the data. After the arrival of the data, the dispenser node delivers them to the client according to the playback rate requirement of the stream.

2.2 Object Storage

Objects are physically stored in storage devices, usually magnetic disks, attached to storage nodes. Striping technique is employed to obtain higher bandwidth and better load distribution. Each object is striped and stored across a group of storage nodes called *striping group*. The maximum achievable bandwidth for a stream is close to the bandwidth of all the disks in the group combined. The I/O load for each stream is evenly distributed over a striping group containing it.

Different objects can be stored in striping groups of different sizes. For each object, a storage parameter, *stripe factor*, defines number of storage nodes in a striping group across which the object is striped. Constraint here is that total bandwidth offered by a striping group has to be at least equal to the playback bandwidth requirement of the object that it contains. Objects can have different stripe factors.

An object can be viewed as a sequence of equal size data units. Each data unit is called *striping unit*. One striping unit may represent several physical data blocks. Our strategy is to store successive data units of an object in logically consecutive storage nodes in a round-robin fashion.

2.3 Object Retrieval

The retrieval operation has to be carried out in a periodic manner according to the real-time deadlines defined by stream playback rate. A retrieval technique called *batch retrieval* [4] is implemented in our server model. In brief, subsets of a striping group take turns retrieving the data for each playback stream. The total bandwidth from storage nodes in a subset has to satisfy the playback bandwidth of the object.

At a dispenser node, there is a fixed size buffer associated with each playback stream. One half of the buffer holds data received from the storage node(s) and the other half holds data to be delivered to the client. A dispenser node manages the buffer in such a way that the refilling and the consuming rates are equal. Thus, the two halves can be used interchangeably.

3 Performance Enhancement

3.1 Client Request Assignment

The performance of the server is sensitive to the client request pattern. The frequency of request to each object is not equal; moreover, it changes over time and cannot be precisely predicted. A manager node has choices in directing each client request to one of the dispenser nodes. The assigning goal is to balance workload among the dispenser nodes while utilizing the resources as effectively as possible.

Two assigning schemes, *RR* and *SOSD*, are studied. In the former scheme, *RR*, client requests are given to dispenser nodes in a round-robin fashion. The assignment is independent of the client request pattern. In the latter scheme, *SOSD*, client requests are distributed in such a way that requests for same object are directed to same dispenser node.

In *RR* scheme, each dispenser node serves approximately the same number of streams. On the other hand, when *SOSD* scheme is applied, a dispenser node serving popular objects may handle more number of streams. This may lead to load imbalance under skew

client request pattern. However, there is an advantage when dispenser node cache is present. *SOSD* scheme benefits the cache hits since requests have a tendency to be satisfied from the cache.

3.2 Multi-pool Interval Caching

The ideas for the proposed caching scheme arose from the very nature of stream playbacks. The stream playback operation needs stripe units in sequence. Multiple playback streams of the same object require the same sequence repeatedly. Moreover, successive stream playbacks always follow one another with fixed intervals in between. Thus, caching only the portion representing these intervals could avoid disk accesses of all streams accessing that object except the first one whose disk accesses are unavoidable anyway.

Figure 2 illustrates the situation. Suppose there are two stream requests for an object. Let the second stream request (S_2) arrive when the first stream request (S_1) is fetching stripe unit number 4. When S_1 is fetching stripe unit number 6, S_2 needs stripe unit number 3. Since their playback rates are equal, these two streams are always three stripe units apart.

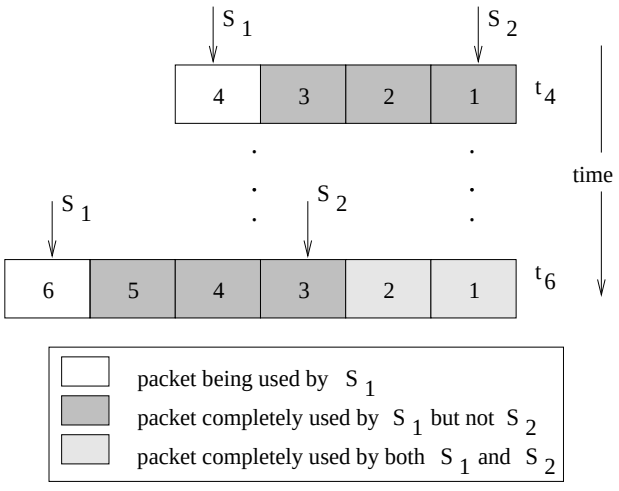

Figure 2: Fixed interval between successive playback sessions

From the same example, if there is no other requests for this object following S_2, stripe units are no longer needed after they are used by S_2. Hence, keeping just the last four stripe units in the cache is sufficient for these two streams. Since the interval between streams directly depends on object access frequency, the interval caching scheme adapts itself dynamically to the client request pattern.

Cache Organization

The cache is composed of a set of cache entries, a number of object pools and a free pool. Each cache entry has a pointer to one object pool. An object pool is a circular linked list of stripe units being used by active streams. One object pool holds stripe units of single object. This makes cache lookup simple. For each stream, only one pool needs to be searched. In addition to a pointer to an object pool, each cache entry contains information about objects such as object id and pool status. The free pool contains empty stripe units which are available for any streams. Adding a new stripe unit into an object pool is done by obtaining one empty stripe unit from the free pool, filling it with the data and attaching it to the pool.

The first stream accessing an object is called an *owner* of the object pool. During the normal playback operation, only the *owner* adds stripe units into the object pool, other streams will just read from there. The *owner*'s sub-request causes a cache miss; thus, a stripe unit will be fetched directly from the disk. This eliminates unnecessary cache lookups.

The *caching window* defines the length of the object pool. The object pool is considered to be either *open* or *closed*. Suppose the value for the *caching window* is *CW*. An *open* object pool is the pool that contains less than the first *CW* stripe units of an object. The pool remains open and new stripe units are added to it until the number of stripe units in the pool reaches *CW*. At that point, the pool is examined if some stripe units can be freed—returned to free pool. Such units are stripe units with the most number of references. The reason for this is that these stripe units will no longer be used after the pool is closed. As in figure 2, if the *caching window* is 6, the pool is closing at time t_6. Stripe units 1 and 2 (which was used twice each) will be returned to free pool and then the object pool which is now holding 4 stripe units is marked *closed*. The content of a *closed* object pool is equivalent to the interval between the *owner* and the last stream accessing the pool at the closing time. The closed pool will not be accessed by the any new streams.

As the playback proceeds, it can be viewed as a *caching window* is shifting toward the end of the object one stripe unit at a time. Considering only streams accessing the same object, every stream starting within one *caching window* interval from the beginning of the *owner* stream is guaranteed to have nearly 100% cache hit at all times.

Large *caching window* has a tendency to capture more number of streams, but it requires more space. Hence, *caching window* should be set to the value just

long enough that if an object has not been accessed by any other stream during the first *caching window* interval, it is justified not to store that object in the cache.

Adding a new stripe unit into an open object pool is done by obtaining one empty stripe unit from free pool, filling it with the data and attaching it to the object pool. For the closed pool, adding a new stripe unit is as simple as replacing the content of the oldest stripe unit in the pool. No stripe unit is taken from free pool once an object pool is closed.

When the free pool is empty, it means that the cache is full. New object requests arriving after that will have to go directly to the disk. Since the cache is providing almost 100% cache hit for the current active streams, it may be better not to replace their contents with that of the new streams. However, if new object is more popular, replacement may be considered.

At this stage, cache is implemented at dispenser nodes because doing so not only eliminates disk accesses, but also reduces the traffic in the interconnection network. Objects will not be removed from the cache until their playback operations are finished.

3.3 Delaying the Deadlines

A server has to maintain a sequence of deadlines during the stream playback operation. A dispenser node is supposed to send out certain amount of data to clients periodically. However, there are times when there are not enough data in the buffer. The deadlines are then missed. As a consequence, the clients encounter data losses. The reason for this is that the requested data have not yet arrived from the storage nodes. From the experiments with fixed deadlines, we found that once a deadline was missed, the following deadlines tended to be missed too. This led to an idea of postponing the deadlines.

Our strategy is to postpone the deadlines whenever there is a miss. This helps a dispenser node catch up with the deadlines from that point on. Assuming that packet number n (P_n) is scheduled to be delivered to client at time t_n, figure 3 shows how the delivery would be if the deadline was missed at time t_2. Delaying by x periods means that the following packets are scheduled for the delivery x periods behind their original scheduled time.

Shifting deadlines improves the missed packet rate. It also degrades the quality of stream playback. We can see from figure 3 that, within the same amount of time, less number of packets are delivered when deadlines are delayed. For example, with a half-period delay factor, the playback is delayed by one period every two misses. A larger delay factor results in poorer display at the client, but it does give a dispenser node a longer time to get back on track. However, it ultimately depends on the client quality of service criterion whether the tradeoffs are justified.

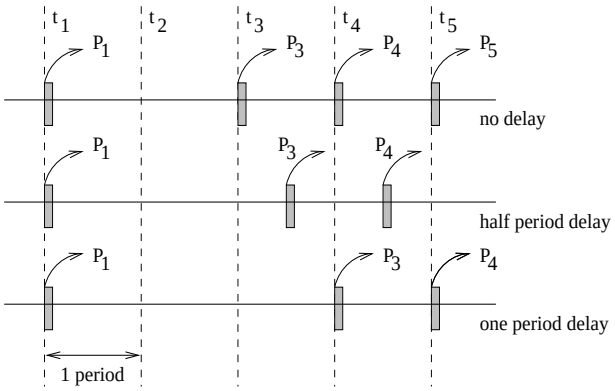

Figure 3: Deadlines delay

4 Experiments

4.1 Experimental Model

The server model was implemented on the Intel Paragon parallel computer. The experimental model consisted of 16 computing nodes. They were configured as 1 manager node, 4 dispenser nodes and 11 storage nodes. The size of each stream buffer at dispenser nodes was 2MB. Cache can be implemented at dispenser nodes. Cache size was set to 20MB and the caching window was fixed at 10 stripe units. Size of a packet to be sent to clients was 64KB.

The server model stored 100 unique objects. Each objects was striped across all the storage nodes. All objects belonged to a single media type, video, with a constant bandwidth requirement of 1.5 Mbits/s (MPEG-1 standard) or 4 Mbits/s (MPEG-2 standard). Experiments were conducted under two client request patterns shown in figure 4. Both patterns conform to Zipfian distribution with different parameters. According to [2], the low skew pattern represents the actual rate at video stores.

4.2 Assumptions

All objects were stored at the storage nodes in compressed digital form. Every object had same storage, retrieval and caching parameters. Each computing node was assumed to have enough disk capacity and main memory for all the playback streams that the server was serving. The communication network between the server model and the clients was assumed to be reliable and fast enough to handle the bandwidth required for object playback. Disk accesses were simulated. Lastly, the decompression of the data was

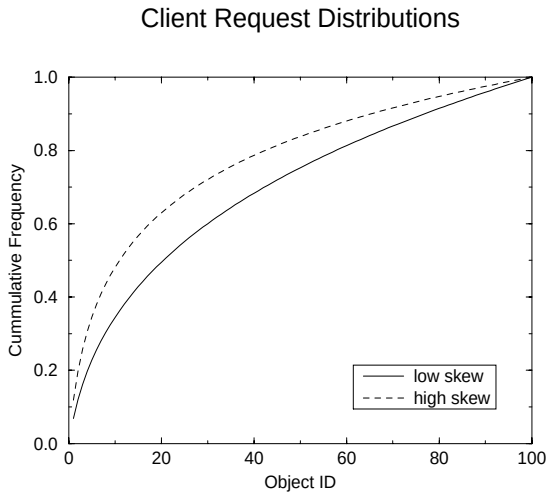

Figure 4: Client request patterns

carried out at the clients' sites. The communication and decompression issues were not addressed in our experiments.

4.3 Parameters and Metrics

Two stripe sizes (512KB and 256KB), three delay factors (no delay, half-period delay and one-period delay), two client request assigning schemes (*RR* and *SOSD*) and two client request patterns (high skew and low skew) were experimented in every combination.

Performance metrics are percentage of cache requests, percentage of missed packets and percentage of delayed deadlines. The percentage of cache requests is the ratio of the number of sub-requests that is satisfied from the cache to the total number of sub-requests. The percentage of missed packets is the ratio of the number of packets that is not ready at time of delivery to the total number of packets scheduled to be delivered. The percentage of delayed deadlines is the ratio of the number of deadlines that is deferred to the total number of deadlines if no delay is added. The percentage of missed packets reflects how well the server performs and the percentage of delayed deadlines shows how much the playback quality is reduced because of the additional delays.

5 Results and Discussion

We are presenting only the selected results from the experiments where all streams had playback rate of 1.5Mbits/s. The server model could support less number of concurrent streams (100-400 streams) when the playback rate was 4 Mbits/s. However, the trends of the graphs appear the same.

The cache of size 20MB can store almost 2 minutes of the object. Each object pool can hold object up to 28 seconds and 14 seconds when stripe size is equal to 512KB and 256KB, respectively. Hence, stream requests of the same object arriving the dispenser node with less than those periods apart will be served from the cache. Since the length of object pools is fixed in terms of stripe units, the cache holds more objects when stripe size is small.

Percentages of cache requests for the configuration when no delay was added are shown in figure 5.

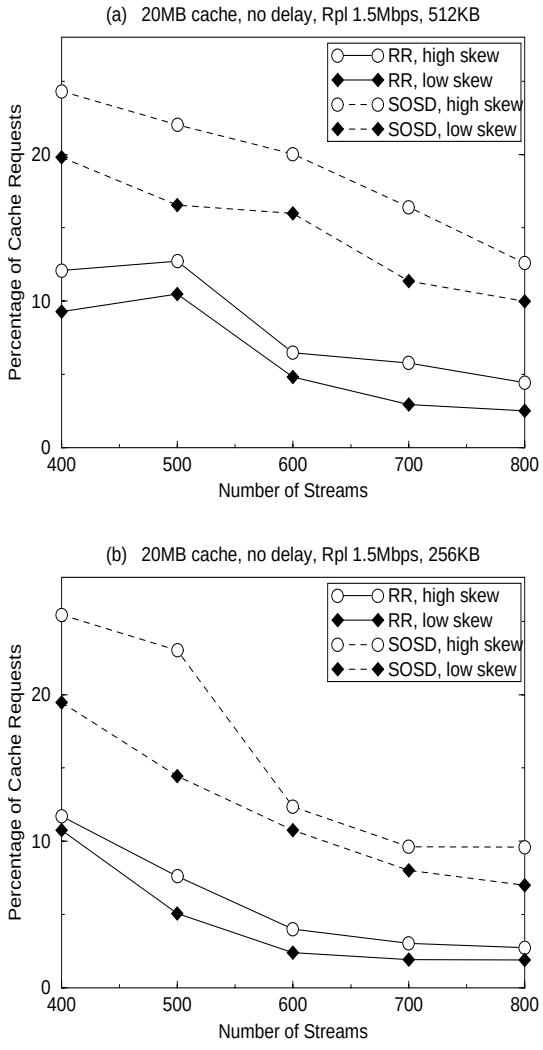

Figure 5: Percentage of cache requests

More sub-requests were captured in the cache when *SOSD* assigning scheme was applied. Also, under high skew request pattern, more sub-requests were found in the cache. This was because the cache was serv-

ing more number of streams. Percentages of cache requests decreased as number of concurrent streams increased. The reason for this was that the very first stream requests had filled up the cache; and they were to remain there until the playback finished. Thus, the stream requests arriving after that were forced to be served with the data from storage nodes. In general, the percentages of cache requests were slightly lower when stripe units were small.

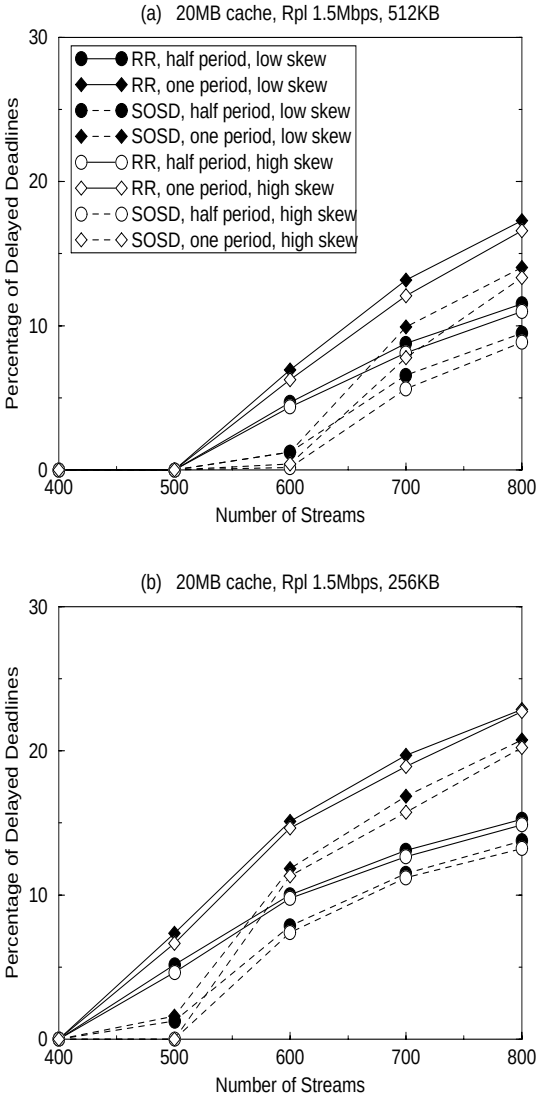

Figure 6: Percentage of delayed deadlines

Deadlines were delayed when a deadline was missed. More delays means less amount of data were delivered. Thus, the playback quality is poorer. Percentages of delayed deadlines are shown in figure 6.

A large delay factor resulted in greater number of shifted deadlines. This was normal because the the deadlines were postponed by a larger margin. The percentages were lower for configurations with larger stripe size (figure 6a).

The request assigning scheme *SOSD* performed better in all cases. This was because requests for same object were served by the same dispenser node. The dispenser nodes were able to deliver streams from the cache. This reduced the chance of missing deadlines.

High skew request pattern resulted in lower number of delayed deadlines. However, the trends of percentages of delayed deadlines under both requests pattern were similar.

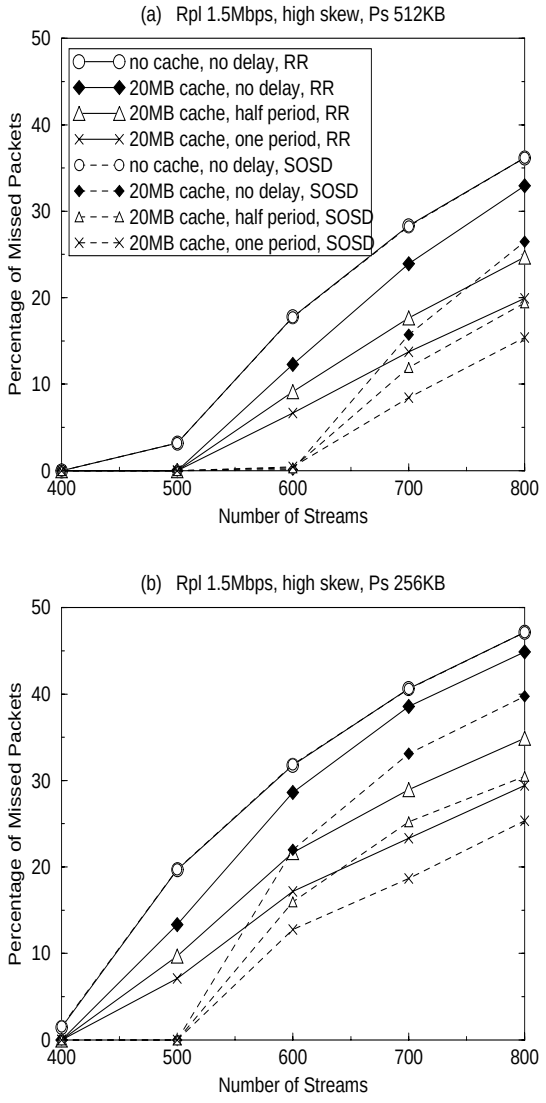

Figure 7: Percentage of missed packets (high skew)

A packet is missed if a stripe unit does not arrive at the dispenser node before the time when it is scheduled to be delivered to the client. Misses occur because storage nodes take long time fetching the data and/or the congestion in the interconnection network. Percentage of missed packets under high and low skew access patterns are shown in figures 7 and 8, respectively.

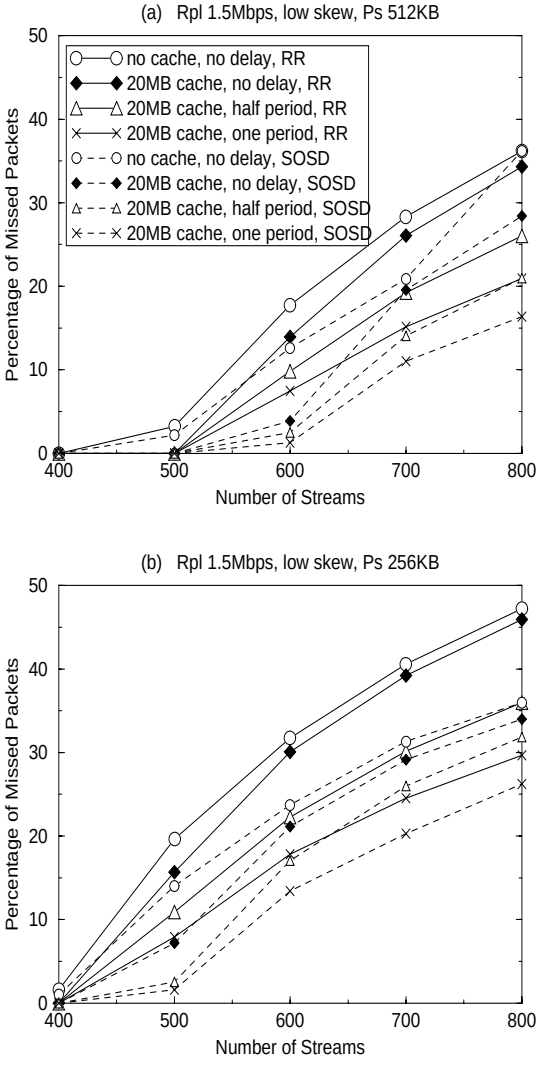

Figure 8: Percentage of missed packets (low skew)

More packets were missed as the server served greater number of concurrent streams. Configurations with smaller stripe size (figures 7b and 8b) were more sensitive to the number of streams, especially under high skew request pattern (figures 7b).

A larger delay factor gave lower number of missed packets. *SOSD* assigning scheme resulted in lower number of missed packets. Its effect was more noticeable under high skew request pattern. *SOSD* scheme was very effective when the cache was present.

The model with cache performed better. In most cases, it could handle up to 500 streams without any missed packet while the model without cache could support only up to 400 streams. Moreover, when delays were added, the number of missed packets was less. This happened because the dispenser nodes had more time to obtain the data. Request pattern did not show the effect much with the *RR* assigning scheme. This was because the way the requests were assigned was independent of request pattern.

The acceptable operating region should have lower than 30% of missed packets. Within this range, the model with no cache could support up to 600-700 concurrent streams while the one with cache and delays could handle more than 800 streams. Even though the configurations with one-period delay yielded better missed packets rate, they produced large number of delayed deadlines. For better stream quality, with delayed deadlines less than 10%, it is better to use half-period delay.

6 Conclusions

The ultimate goal of the multimedia server design is to maximize the number of streams served while maintaining acceptable stream quality. We introduced several techniques to improve the performance of the server. Multi-pool interval caching makes cache lookup simple. Unnecessary lookups of the fresh object references can be avoided. It is also adaptive. Adding delays is effective, but adding too much will hurt the stream quality. *SOSD* assigning scheme performs well when cache is present. The combined effect of *SOSD* scheme, deadline delays and cache improves the capacity of the server by approximately 30%.

References

[1] A. Dan, D. Dias, R. Mukherjee, D. Sitaram, and R. Tewari. Buffering and caching in large scale video servers. In *Proceedings of COMPCON*, 1995.

[2] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Second Annual ACM Multimedia Conference and Exposition*, 1994.

[3] D. Jadav, C. Srinilta, and A. Choudhary. I/O scheduling tradeoffs in a high performance media-on-demand server. *Proceedings of the 2nd Intl. Conference on High Performance Computing*, December 1995.

[4] C. Srinilta, D. Jadav, and A. Choudhary. Design and evaluation of data storage and retrieval strategies in a distributed memory continuous media server. *International Parallel Processing Symposium*, 1997.