Optimal and Efficient Merging Schedules for Video-on-Demand Servers

Derek Eager
Dept. of Computer Science
University of Saskatchewan
eager@cs.usask.ca

Mary Vernon
Computer Sciences Dept.
University of Wisconsin
vernon@cs.wisc.edu

John Zahorjan
Dept. of Computer Sci. & Eng.
University of Washington
zahorjan@cs.washington.edu

1 INTRODUCTION

The simplest video-on-demand (VOD) delivery policy is to allocate a new media delivery stream to each client request when it arrives. This policy has the desirable properties of "immediate service" (there is minimal latency between the client request and the start of playback, assuming that sufficient server bandwidth is available to start the new stream), of placing minimal demands on client capabilities (the client receive bandwidth required is the media playback rate, and no client local storage is required[1]), and of being simple to implement. However, the policy is untenable because it requires server bandwidth that scales linearly with the number of clients that must be supported simultaneously, which is too expensive for many applications.

This focus of this paper is on how to reduce the server bandwidth required through the design of efficient server delivery policies. The solution we arrive at preserves the properties of immediate service and simplicity of implementation, while decreasing server bandwidth to the logarithm of the number of simultaneously active clients. To achieve this, though, requires clients with receive bandwidth twice the media playback rate, and some amount of client local storage.

There has been considerable previous work on how to reduce server bandwidth in VOD systems. The work we present is inspired by the results in [5], which show that hierarchically merging data delivery streams can achieve nearly the minimum possible server bandwidth when clients have receive bandwidth twice the media playback rate and some local storage. The hierarchical stream merging approach is in turn inspired by patching [3,9], dynamic skyscraper [4], and piggybacking [8] approaches. Patching provides the "stream merging" mechanism shown in Figure 1. Under stream merging, a later arriving client joins the multicast stream delivering the media data to some earlier client, buffering the data it receives until it is needed for playback. Additionally, a patching stream (shown as a dotted line in Figure 1) is allocated to the new arrival to allow it to begin playback immediately. The patching stream terminates when it reaches the data already buffered by the client from the earlier stream, at which point the client completes playback using the buffered data plus the data acquired by continuing to listen to the earlier stream.

Dynamic skyscraper and piggybacking provide the notion of performing merges repeatedly, leading to a binary tree merging structure as shown in Figure 2.

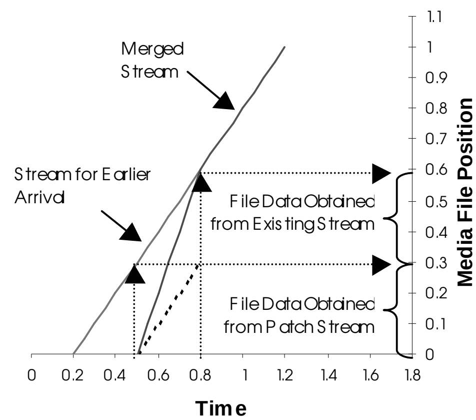

Figure 1: Stream Merge Operation
(The client arriving at time 0.5 is allocated a patch stream until time 0.8, from which it receives media data in the range 0.0 to 0.3. It also listens to the existing multicast stream, obtaining data in the range 0.3 to 0.6. At time 0.8 the merge is complete, the patch terminates, and the client listens to the existing stream only.)

The key question we address in this paper is how to create an efficient merge tree in this new environment, in which client merging is provided by the mechanism illustrated in Figure 1 and clients can snoop on any earlier multicast stream. That is, policies are needed to determine which clients to merge with what others, and in what order, to create a merge tree that minimizes the total server bandwidth required to deliver the media data to those clients.

2 OPTIMAL STREAM MERGING

The set of decisions made by a stream merging policy for any particular set of client requests describes a merge tree. The cost of a merge tree, measured in the total amount of data the server must send to satisfy that set of clients, is simply the sum of the projections of the line segments it contains onto the X-axis. For example, the cost of the tree in Figure 2 is 1.9.

[1] Throughout this paper we ignore the initial buffering required to help smooth the irregular delivery times in shared networks, as this is policy independent.

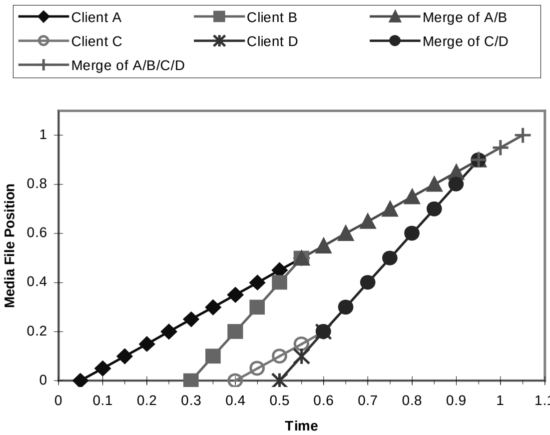

Figure 2: Example Merge Tree

The optimal merge tree for a given set 1…M of client request times can be computed using a dynamic program [1, 6]. The dynamic program exploits the fact that the optimal tree for just the client requests in the range (i…j) is formed by merging the optimal trees over just the ranges (i…k-1) and (k…j), for some i<k≤j. The cost of the merged tree is the sum of the costs of the sub-trees, minus some "fix-up" representing the savings obtained by the additional merge operation. The tree shown in Figure 2 is optimal for the given client request times.

The dynamic program used to compute optimal merge trees for a known set of arrival times can be adapted to compute the minimum cost tree to complete the set of active streams at each arrival instant in a real system. For example, when client C arrives in Figure 1, the dynamic program would dictate that C should merge with B at time 0.5, and then the merged stream for B and C should merge with A at time 0.75. When client D arrives, the program computes the merges shown in the figure; note that these merges can be implemented if client B has continued to snoop on client A's stream during time 0.4 to 0.5. (We make use of this idea in the next section.)

A significant problem with using the dynamic program at each request arrival instant is that it requires time $O(M^3)$ and space $O(M^2)$. Given that this approach is a heuristic in any case (since the merge trees computed are optimal only if no further arrivals take place), we are motivated to look for simpler ways to determine merge trees that are likely to be "good" but not necessarily optimal.

3 EARLY MERGING

Examination of optimal merge schedules largely supports the intuitive hypothesis that, in building the merge tree, one should merge the two neighboring streams that can be merged at the earliest point in time, followed by the next such pair, and so on. It can be shown, in fact, that this policy yields the optimal tree when any set of three or fewer client requests is considered. However, for four or more requests, there are occasional exceptions. Figure 2 is an example. Repeatedly following this heuristic of "early merging" merges the arrivals at time 0.3 and 0.4, then that merged stream with the arrival at time 0.5, and finally that merged stream with the arrival at time 0.05. The cost of this tree is 1.95, versus the optimal cost of 1.9. Nonetheless, this heuristic is attractive as it is easy to implement and correctly captures an overwhelming fraction of the merges that occur in the optimal trees.[2]

In order to implement the early merging heuristic, we follow two principles. First, a client (or group of clients) should decide which earlier stream to snoop on when it arrives or when it merges with another stream, since listening to the earlier stream as soon as possible will minimize the total server load required for the merge to take place. Second, the client(s) should snoop on the chosen earlier stream until either the merge is successful or another (group of) client(s) preemptively merges with their patch stream. This latter principle enables selection of near-optimal merge pairs at the latest possible time with respect to client arrivals.

A key point is that, regardless of which earlier stream is chosen as the merge target, the decision can be undone by future client requests – a new request arriving shortly after this one will merge with it, "resetting" the time at which those streams can begin merging with earlier streams, and so perhaps altering the decision of which such stream is the appropriate target. Note that resetting implies that the group that is caught must "throw away" whatever data it has accumulated by listening to an earlier stream up to the merge point. While this may seem wasteful, it is not, because that data must be retransmitted in any case to satisfy the clients that caught up, since they have not accumulated this later data.

In the remainder of this section we describe three variants of the early merging family that differ in which earlier stream is chosen for an arriving client, or a newly merged group, to snoop on. These variants differ in how aggressive they are in finding the earliest possible merge (i.e., in the complexity of computing the stream to listen to).

3.1 Earliest Reachable Merge Target (ERMT)

In this variant of early merging, a new client or newly merged group of clients snoops on the closest stream that it can merge with if no later arrivals preemptively catch them. For example, in Figure 2, client B will listen to the stream for client A, client C will listen to the stream for client B, and client D will also listen to the stream that was initiated for client B. D snoops on B's stream because D cannot merge with C (since C will merge with B at time 0.5), D can catch the merged stream for B and C, and this is the earliest reachable merge target for D if no later arrivals preemptively merge with D.

One way to compute the stream to snoop on is to "simulate" the future evolution of the system, given the current merge target for each active stream and the rule for determining new merge targets, and assuming no future client arrivals. A more efficient, incremental maintenance of the merge tree is also possible [6]. These approaches are not described here due to space constraints.

3.2 Simple Reachable Merge Target (SRMT)

The requirement of ERMT that all merge targets be the earliest possible complicates the calculation of which stream to snoop on. A simpler approach is to determine the closest reachable merge

[2] Note that if client D arrives at time 0.44 instead of 0.5, the heuristic leads to the optimal merge tree with similar structure to the tree shown in the figure.

target if each currently active stream terminates at its current target merge point (or at the end of the file if it has no current target merge point). For example, if client D arrives at time 0.49 in Figure 2, D will snoop on the stream for client A, since D cannot reach client B's stream before its target merge point at time 0.55.

The SRMT is easily computed. For M currently active streams numbered 1..M in order of earliest client arrival time, let $D_{j,i}$, $1 \le j < i \le M$, be the distance between streams j and i (i.e., position of j minus the progress of i). Let T(j) be the known target stream for each stream j<i. Stream i is assigned merge target k for which $D_{ki} < D_{T(k),k}$ and k is as large as possible, k<i.

SRMT overlooks some merges that ERMT finds. (For example, if client D arrives at time 0.49 under ERMT, client D will snoop on client B's stream.) This happens because SRMT ignores the fact that a new merged stream is created when two streams merge. This simplifies the calculation of the stream to snoop on, but results in some merge operations taking longer than necessary.

3.3 Closest Target (CT)

This scheme simply chooses the closest earlier stream still in the system as the next merge target. In Figure 2, if client D arrives at time 0.49, D would simply snoop on the stream initiated for C.

The merge targets computed by CT are not necessarily reachable, even if no further arrivals occur. The reason is that the target stream may itself merge with its target before it can be reached by the later stream. When this happens, the later stream must select a new merge target, again using the CT algorithm.

4 PERFORMANCE RESULTS

This section uses simulation to explore the performance of the early merging policies defined in Section 3. The first question we address is the extent to which the early merging characteristic is essential for good performance. We do this by comparing our policies to two hierarchical merging policies that do not have early merging as a goal, and to optimal stream merging with *a priori* knowledge of all client request arrival times.

Further experiments are carried out with ERMT as a representative of the early merging family. We assess its average server bandwidth requirement as a function of request arrival rate and available client buffer space. Then, we provide performance comparisons with the previously proposed dynamic skyscraper [4] and optimized grace patching [2,7] techniques, considering server bandwidth requirements as well as average client waiting times and balking frequencies. We omit comparisons with piggybacking because clients only receive on a single multicast stream in a piggybacking system, which limits the attainable performance gains.

Additional results, and details regarding our experimental methodology, are given in [6].

4.1 Comparison with Optimal Stream Merging

Figure 3 compares the early merging policies with two "static pairing" hierarchical merging policies. The Y-axis values are the percent difference in average server bandwidth compared to the optimal, offline stream merging schedule. The client request rate *N* is expressed as the average number of requests that arrive per unit of time (which is defined as the playback time for the file). In this figure, we assume that clients have enough local storage to buffer any data that is received ahead of schedule.

The static pairing policies are Static Tree (ST) and Hierarchical Even-Odd (HEO). ST merges streams according to a complete binary tree, based solely on the client arrival number. HEO [8] sets the merge target of a stream to the next youngest still existing stream, so long as that stream currently has no merge target.. (If it does, no merge is scheduled.) In both cases, only merges that can complete before the target terminates are scheduled.

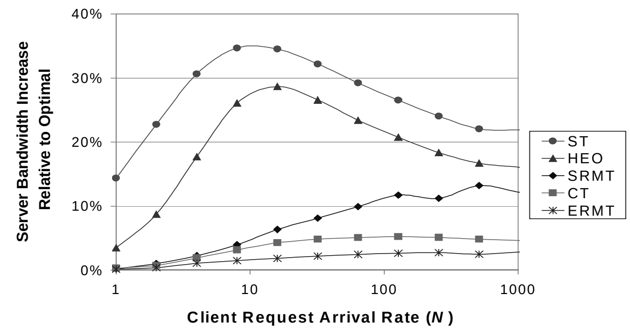

Figure 3: Performance Relative to Optimal Stream Merging

The key observations from this figure are:

- The early merging policies significantly outperform the policies that do not include the early merging characteristic.
- All three variants of early merging have close to optimal bandwidth requirements, leaving little room for improvement.
- It appears to be more important to merge with the closest streams (as in CT and ERMT) than to never listen to unreachable streams (as in SRMT).

4.2 Server Bandwidth Requirements

Figure 4 shows the effect of client arrival rate and limited client buffer storage on the average server bandwidth required per client by early merging (as represented by the ERMT variant), for delivery of a single file. Bandwidth requirements are expressed in units of the playback rate. The results are obtained from a simulation in which merges that would exceed the local storage capacity of any client in the merging stream are not scheduled.

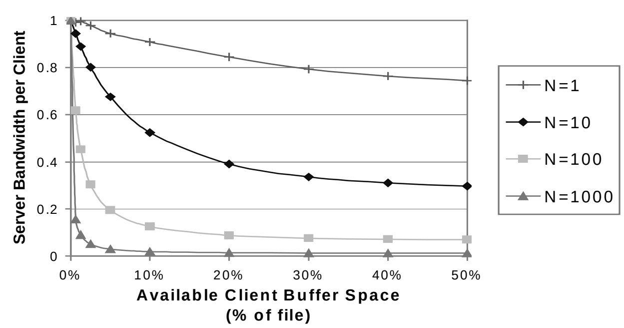

Figure 4: Bandwidth Requirements of Early Merging

As illustrated in the figure, per client server bandwidth decreases with increasing arrival rate, and total server bandwidth grows only logarithmically in the client request rate (rather than linearly as with unicast delivery). Buffer sizes on the order of 10% of the file size are sufficient to achieve much of the performance gains.

4.3 Comparison with Previous Techniques

This section assumes clients have enough local storage to buffer data as needed. As in [5], Figure 5 shows that the previously proposed dynamic skyscraper [4] and optimized grace patching [2,7] techniques require substantially greater average server bandwidth to deliver a popular file than does early merging.

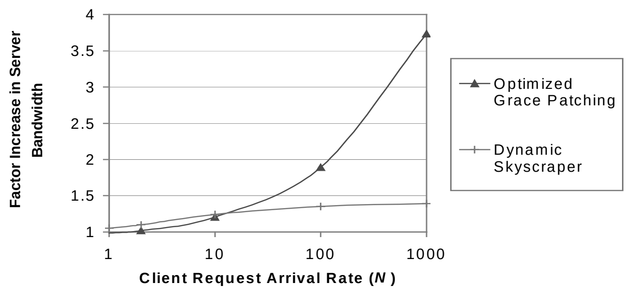

Figure 5: Previous Techniques Relative to Early Merging

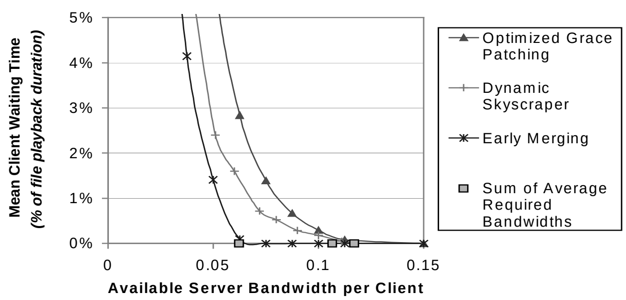

Figure 6: Mean Client Waiting Times

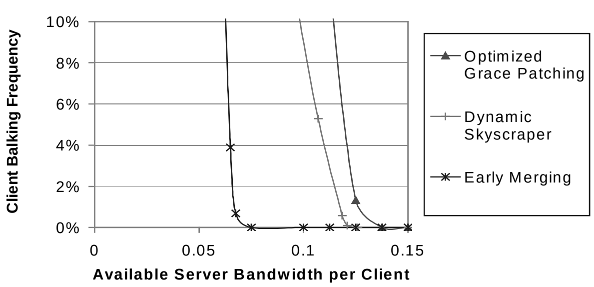

Figure 7: Client Balking Frequencies

Figures 6 and 7 consider systems with fixed available server bandwidths, supporting requests for 20 equal-sized "hot" files with a total request rate of N=2000. File request frequencies are given by the Zipf(0) distribution. Figure 6 shows average client waiting time, as a function of available server bandwidth, for each of the techniques. Figure 7 shows balking frequencies when clients don't wait, but rather give up if immediate service is not possible. Note that waiting times increase rapidly when too little server bandwidth is available, and that, for each technique, the sum of the average server bandwidths required for immediate delivery of each file defines an appropriate system operating point. As illustrated in both figures, early merging provides dramatically better performance than that provided by the previous techniques.

5 CONCLUSIONS

This paper has proposed a family of "early merging" VOD delivery policies that hierarchically merge delivery streams for a given file using the heuristic of performing the earliest merge first. Results show that early merging, unlike other hierarchical merging policies, achieves performance close to optimal offline hierarchical merging (in which all client request arrival instants are known in advance). Furthermore, a very simple heuristic for determining a target stream to snoop on (i.e., the closest earlier stream still in the system), is sufficient to achieve nearly all of the performance gain. Results also show that early merging greatly outperforms the previously proposed dynamic skyscraper and optimized patching techniques, with respect to both server bandwidth required for immediate service and average client waiting time or balking frequency for a fixed available server bandwidth. Results in [5] show that not much further improvement in performance is possible.

ACKNOWLEDGEMENTS

This work was supported in part by the NSF (Grants CCR-9704503 and CCR-9975044) and NSERC (Grant OGP-0000264).

REFERENCES

[1] C. C. Aggarwal, J. L. Wolf, and P.S. Yu, "On Optimal Piggyback Merging Policies for Video-On-Demand Systems", *Proc. ACM SIGMETRICS Conf.*, Philadelphia, PA, May 1996.

[2] Y. Cai, K. A. Hua, and K. Vu, "Optimizing Patching Performance", *Proc. MMCN'99*, San Jose, CA, Jan. 1999.

[3] S. W. Carter and D. D. E. Long, "Improving Video-on-Demand Server Efficiency Through Stream Tapping", *Proc. ICCCN'97*, Las Vegas, NV, Sept. 1997.

[4] D. L. Eager and M. K. Vernon, "Dynamic Skyscraper Broadcasts for Video-on-Demand", *Proc. MIS'98*, Istanbul, Turkey, Sept. 1998.

[5] D. L. Eager, M. K. Vernon, and J. Zahorjan, "Minimizing Bandwidth Requirements for On-Demand Data Delivery", *Proc. MIS'99*, Indian Wells, CA, Oct. 1999.

[6] D. L. Eager, M. K. Vernon, and J. Zahorjan, "Optimal and Efficient Merging Schedules for Video-on-Demand Servers", CSE TR# 99-08-01, U. of Washington, Seattle, Aug. 1999.

[7] L. Gao and D. Towsley, "Supplying Instantaneous Video-on-Demand Services Using Controlled Multicast", *Proc. IEEE ICMCS'99*, Florence, Italy, June 1999.

[8] L. Golubchik, J. C. S. Lui, and R. Muntz, "Reducing I/O Demand in Video-On-Demand Storage Servers", *Proc. ACM SIGMETRICS Conf.*, Ottawa, Canada, May 1995.

[9] K. A. Hua, Y. Cai, and S. Sheu, "Patching: A multicast technique for true video-on-demand services", *Proc. ACM MULTIMEDIA'98*, Bristol, U.K., Sept. 1998.