RZ 3423 (# 93661) 06/17/02
Computer Science 6 pages

Research Report

Efficient Buffer Management for Scalable Media-on-Demand

Marcel Waldvogel*

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*Work started when the author was with Washington University in St. Louis, MO.

Wei Deng and Ramaprabhu Janakiraman

Applied Research Laboratory
Washington University in St. Louis
St. Louis, MO 63130-4899

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

IBM **Research**
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

Efficient Buffer Management for Scalable Media-on-Demand

Marcel Waldvogel
IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
mwl@zurich.ibm.com

Wei Deng
Applied Research Laboratory
Washington University in St. Louis
St. Louis, MO 63130-4899
USA
wdeng@arl.wustl.edu

Ramaprabhu Janakiraman
Applied Research Laboratory
Washington University in St. Louis
St. Louis, MO 63130-4899
USA
rama@arl.wustl.edu

***Abstract*— Widespread availability of high-speed networks and fast, cheap computation have rendered high-quality Media-on-Demand (MoD) feasible. Research on scalable MoD has resulted in many efficient schemes that involve segmentation and asynchronous broadcast of media data, requiring clients to buffer and reorder out-of-order segments efficiently for serial playout.**

In such schemes, buffer space requirements run to several hundred megabytes and hence require efficient buffer management techniques involving both primary memory and secondary storage: while disk sizes have increased exponentially, access speeds have not kept pace at all.

The conversion of out-of-order arrival to in-order playout suggests the use of external memory priority queues, but their content-agnostic nature prevents them from performing well under MoD loads. In this paper, we propose and evaluate a series of simple heuristic schemes which, in simulation studies and in combination with our scalable MoD scheme, achieve significant improvements in storage performance over existing schemes.

I. INTRODUCTION

The widespread availability of broadband connectivity to the end-user has opened up possibilities of high quality Media-on-demand (MoD) delivery to the home. Earlier work [1] on requests in video rentals suggests that an 80:20 rule might hold here: 80% of requests are for the top 20 movies. Recent research [2], [3], [4], [5], [6], [7] has thrown up a variety of scalable MoD schemes that attempt to conserve server bandwidth by segmenting popular MoD data and periodically broadcasting it to interested clients.

One downside of such periodic-broadcast schemes is that they require the client to re-order segments and buffer them till their proper playout time. These schemes therefore require clients to have buffer capacities of several hundred megabytes, necessitating use of a hard disk to store the bulk of the buffered data. While disk space usage is no longer an issue due to exponentially increasing disk capacities, disk load is: disk throughputs have not kept pace with capacities.

In this paper, we discuss schemes for efficient client-side management of buffers for scalable MoD schemes. Specifically, we address the problem of minimizing the time spent in disk I/O by the client during the course of accessing media data.

The rest of this paper is organized as follows: In § III we provide a brief introduction to the scalable MoD scheme that we consider. We then provide a justification for efficient buffer management in § IV. We introduce our schemes in § V and evaluate their performance in § VI. Finally, we briefly discuss server-side support for more efficient buffer management by clients.

II. RELATED WORK

Our buffering problem can be formulated as an external memory sorting problem involving random insertions and minimum element deletion, for which a heap- or priority-queue-based algorithm naturally suggests itself. One high-performance variant is a radix-heap-based structure, which stores frames in multi-level buckets according to an optimal radix. According to a comparative analysis performed by [8], radix heaps are the best existing scheme when the keys are integers in a pre-defined range. However, radix-bucket-based schemes neglect to take linear access into account. Widely seperated frames map to the same bucket and end up getting written to and read from disk together, resulting in suboptimum performance. We have used a modified version of the algorithm presented in [8] and in [9] for performance comparison.

Buffer management could also be treated as a cache replacement problem, with the additional property that there are at most two accesses to each block, one random (during arrival), the other serial (during playout). The optimum cache replacement strategy of writing the frame with the most distant playout to disk, while minimizing

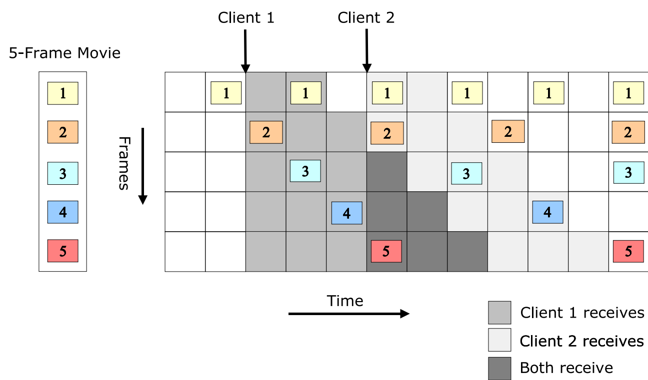

Fig. 1. Basic Transmission Pattern

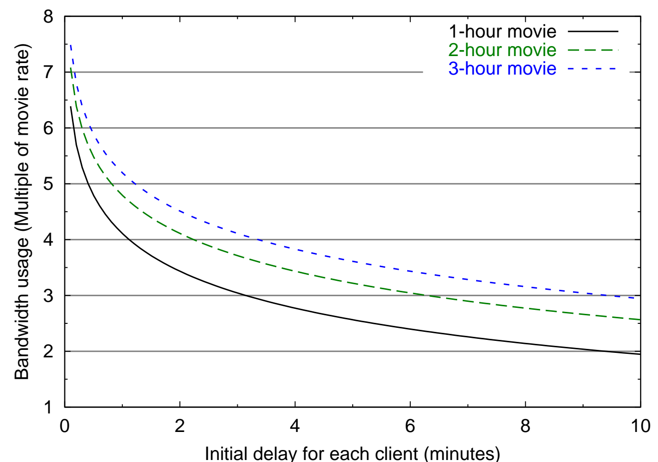

Fig. 2. Bandwidth vs. Delay

the number of disk accesses, ends up using one seek for every frame read or written, incurring a high overall disk I/O cost.

III. Scalable MoD Transmission

Consider the broadcast of a popular movie of n frames. Assume the frames are to be broadcast to satisfy the on-demand requirements of multiple clients with different join times. Now, a client with a join time of t and a wait-time of w will require frame f at time t_f no later than playout time $t+w+f-1$, i.e., $t \leq t_f < t+w+f$. Thus each client has a window of $w+f$ instants in which to receive frame f. In the absence of client feedback, i.e., in a proactive system, on-demand delivery for each client is ensured by broadcasting frame f at least once every $w+f$ instants.

This is formalized as algorithm IDEAL (Algorithm 1) below. The schedule generated by algorithm IDEAL (with $w=0$) is plotted in Fig. III, showing the frames transmitted during each instant and the receive windows for two clients joining at instants 1 and 4. In this example, we assume a *transmit* system call that schedules frame f for transmission at instant t using a transmission queue.

Algorithm 1 Algorithm IDEAL

```
for all frames f_j do
  λ ← j + w;
  for (t ← λ; t ≤ t_max; t+ = λ) do
    transmit (t, f_j);
  end for
end for
```

It can be proved [7] that the average bandwidth for the entire movie is:

$$B = \sum_{f=1}^{n} \frac{1}{w+f} \approx \ln \frac{n+w}{w}, \qquad (1)$$

where B is normalized to the playout bandwidth of the movie. In other words,

$$\text{Bandwidth (in frames/instant)} \approx \ln \frac{\text{Movie length}}{\text{Initial delay}} .$$

In practical terms, serving a 2-hour 300 kbps Real Media or MPEG-4 movie with a 5-minute initial delay requires a server and client bandwidth of ≈ 1 Mbps. Thus, the system begins to be advantageous as soon as the number of clients exceeds 3. Fig. III shows the scaled bandwidth usage (relative to the bit rate of the movie) as a function of the initial delay (relative to the length of the movie).

IV. The Buffer Management Problem

As it stands, Algorithm IDEAL results in too spiky a bandwidth usage to be implemented in practice. We have developed schemes [7] that schedule frame transmission in a way that preserves its bandwidth optimality without violating peak bandwidth constraints. These schemes work by "fuzzifying" the schedules, redistributing bandwidth more equitably over time by moving frames away from bandwidth "spikes". This is formalized as algorithm BASIC (Algorithm 2). The crux of it is the FINDNEIGHBOR function, which finds an alternate neighboring time slot for frames that algorithm IDEAL originally schedules in relatively 'crowded' time slots.

We do not discuss further the ways in which FIND-NEIGHBOUR may be implemented, but refer the interested reader instead to [7]. For this work , it is sufficient to assume that for a movie of n frames, any client with a join time of t_s and an initial wait period of w would receive at least one instance of any given frame f at some instant t_f before playout, i.e. in the time interval $[t_s,\, t_s+w+f)$.

It is clear that the scheme relies heavily on buffering and reordering of out-of-order frames by the client. It can

Algorithm 2 Algorithm BASIC

```
B_est ← B_act ← 0;
for all frames f_j do
  λ ← j + w;
  B_est += 1/λ;
  for (t ← λ; t ≤ t_max; t += δ_t, B_act[t]++) do
    δ_t ← FINDNEIGHBOR;
    transmit (t + δ_t, f_j);
  end for
end for
```

be shown [7] that the client buffer size as a function of time t relative to t_s is

$$B(t) = \begin{cases} t \ln \frac{n+w}{w} & 1 \le t \le w, \\ t \ln \frac{n+w}{t} & w \le t \le n+w \end{cases} \quad . \tag{2}$$

The peak buffer size is $\frac{n+w}{e}$ of the entire movie size, at a fraction $\frac{n+w}{e}$ (e being Euler's constant) of the entire playout time into movie. For a two-hour MPEG-2 movie with a transfer rate of 4 Mb/s and a 30 s initial delay, this would translate to a peak buffer requirement of approximately 700 MB.

Thus the typical client is forced to distribute its buffer between a small fast cache in primary memory and a large, slow hard disk that holds the bulk of the buffered frames. As frames arrive over the network, existing frames are displaced from the cache and written to disk until playout time. This naturally leads us to the question of doing this efficiently from a disk I/O performance perspective. Why is this important? Let us look at some of the reasons:

- For set-top boxes, more efficient buffer management obviates the need for higher-performance hardware, leading to lower costs.
- Clients could be commodity PCs running multiple concurrent tasks: both memory and storage are shared resources that should be used optimally.
- Some proxies or transcoding devices located, e.g., at cable head-ends, receive, buffer, and reorder frames (among other things) before streaming them serially to constrained end-systems such as diskless set-top boxes. Better buffer management contributes to greater scalability.

V. Efficient Buffer Management Schemes

A. Disk Metrics

We use a relatively simple metric to estimate disk performance: Consider a disk with these parameters:

RANDOM SEEK TIME, S_r: This is the average seek time for unrelated read/writes.

SEQUENTIAL SEEK TIME, S_s: This is the seek time for adjacent blocks of the same read/write.

BLOCK SIZE, B: Disk space is always allocated in blocks. All disk I/O is in multiples of B bytes.

TRANSFER RATE, T: This is the rate at which data can be read/written. If we assume that the disk allocator writes data in one write over multiple adjacent blocks, the time taken for a read/write of b bytes is given by

$$t(b) \approx S_r + \frac{S_s b}{B} + \frac{b}{T} \quad .$$

Effectively, the time to transfer n frames in one read/write between memory and disk is of the following form:

$$t(n) \approx C_1 + C_2 \times n, \tag{3}$$

where C_1 and C_2 are constant for a given disk.

This model might seem simplistic in these times of intelligent caching disk controllers, but considering the massive amounts of data involved, our analysis shows that this simple model provides a close approximation. For typical disk and transmission parameters, $C_1 \approx 10$ ms and $C_2 \approx 2$ ms. We have arrived at these parameters by experimenting with simulations of the *Seagate Barracuda* disk, as obtained from DiskSim [10]. Moreover, we are working on refining these measures, possibly through more complex disk models.

During playout of the movie, a number of frames are written to disk and read back again. Our goal is to minimize the total I/O time spent over these frames, as estimated by (3). In the following sections, we present our algorithms.

B. Most Distant Playout (MDP) Replacement

MDP is similar to the optimum cache replacement algorithm: the principle is to replace frames that would be required farthest in the future. With sequentially accessed media data, the highest-numbered frame in the cache is the ideal candidate for replacement.

In MDP, instead of replacing just the last frame in the cache, a number κ of frames in cache are written out to disk as a single chunk. Preemptively writing out a batch of likely-to-be replaced frames amortizes seek time over κ frames, instead of using up one seek for each frame. When the earliest frame in the sequence is to be played out, the entire chunk is read back into the cache again.

MDP is suboptimum in terms of the number of total accesses to disk; some frames cycle more than once between memory and disk. But as access time for reasonable frame sizes is much lower then seek time, MDP, by effectively trading in more reads/writes for fewer seeks, is able to achieve a reduction in overall disk I/O time.

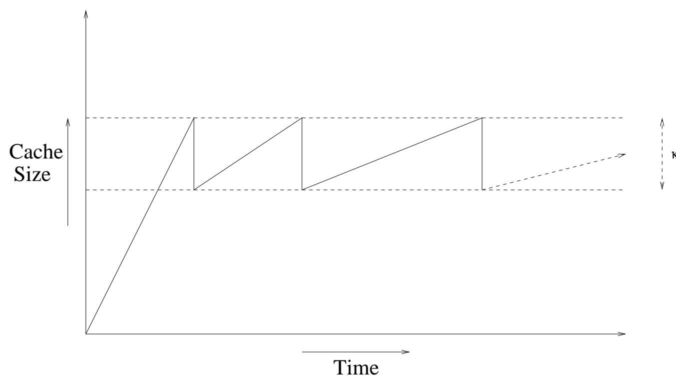

Fig. 3. Buffer growth with batched writes (Schematic)

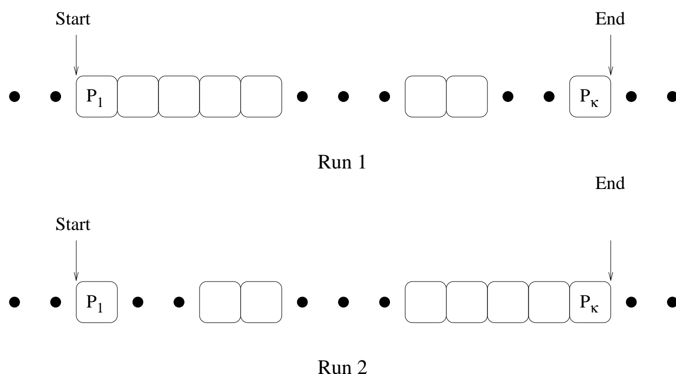

Fig. 4. Two compact runs. Squares represent frames in memory, dots their absence.

C. Most Compact Sequence (MCS) Replacement

Another strategy is to take advantage of linear access of frames, and write out the κ-long run of frames that has the most compact playout schedule of the frames currently in memory, in order to reduce the rate of blocks cycling between disk and memory. For a run of κ frames with playout times $P_1, P_2, \cdots P_\kappa$, we define its *sparseness* (as opposed to compactness) by

$$S = \sum_{i=1}^{\kappa}(P_i - P_1)$$

We choose the rightmost such sequence with minimum S. In the best case, this is a stretch of κ continuous frames.

Sparseness can be thought of as a cumulative measure of wasted buffer occupancy: the earliest frame in a sparse run wastefully drags along much later frames from disk to memory. Replacing compact runs lowers the risk of later frames in the run getting replaced again before playout.

An advantage of this definition is that it selects Run-1 over Run-2 in Fig. 4, although both have the same 'density.' A run with more frames near the head is a better candidate for replacement because fewer frames (if at all) get cycled back to disk, while other frames necessary for playout are being fetched.

D. Optimal Block Sizes

The rationale for writing out blocks of κ frames instead of single frames is to trade an increase in the number of (cheaper) disk transfers for a decrease of (costly) seeks. The optimum value of κ depends on the relative costs of seeks vs. read/writes as well as the cache size.

Increasing κ will reduce seeks and buy shorter disk I/O time upto a point. Thereafter, increased access times predominate, and the parameters move away from the optimum. Figures 5 and 6 plot κ as a fraction of the cache size C, against I/O time as a fraction of $n+w$, the playout time. The I/O time is calculated based on the disk parameters discussed above. Fig. 5 shows the results for MDP and Fig. 6 for MCS. It is seen that $\kappa \approx 2 \ldots 3\%$ of the cache size provides the best performance for both MDP and MCS. All I/O times are given relative to the actual playout time. As all of the blocks written to disk are of the same size, disk allocation management becomes trivial. We have also experimented with a variable κ, but feel that the performance gains accrued might not be worth the increased complexity in disk management.

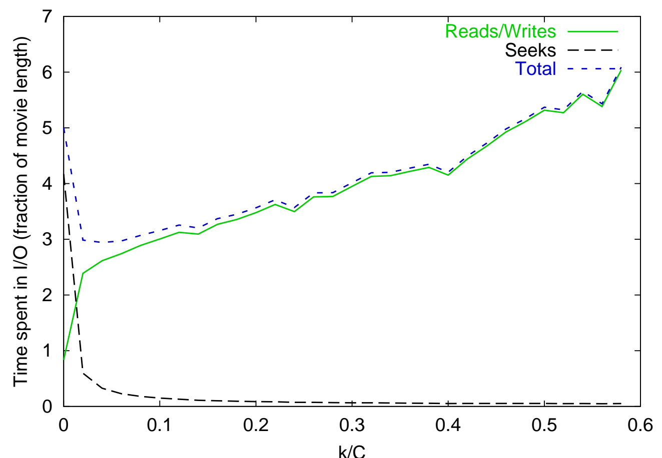

Fig. 5. MDP: $\frac{\kappa}{C}$ vs. I/O time

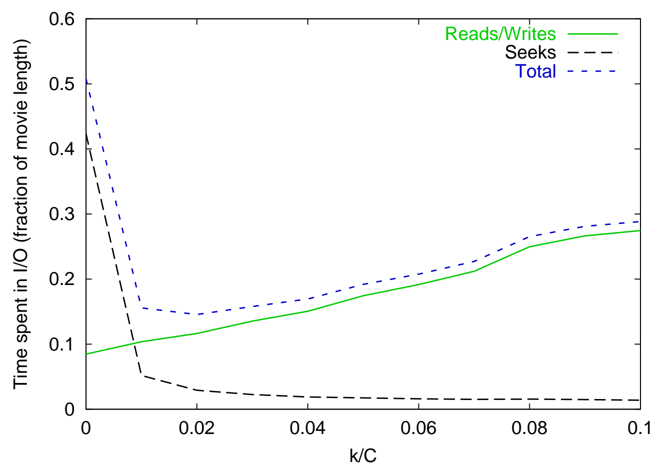

Fig. 6. MCS: $\frac{\kappa}{C}$ vs. I/O time

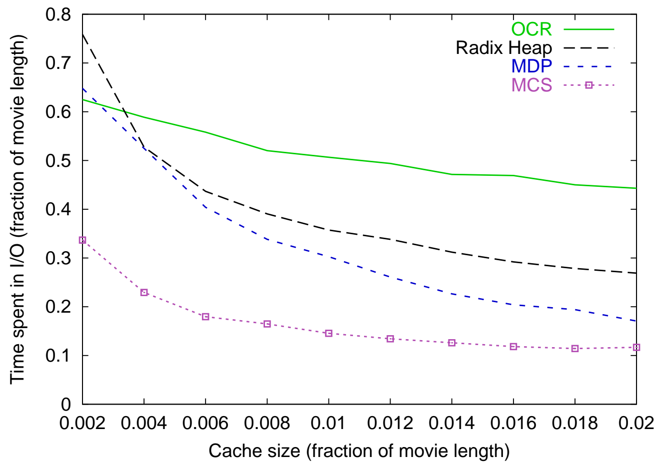

Fig. 7. Cache size vs. I/O time

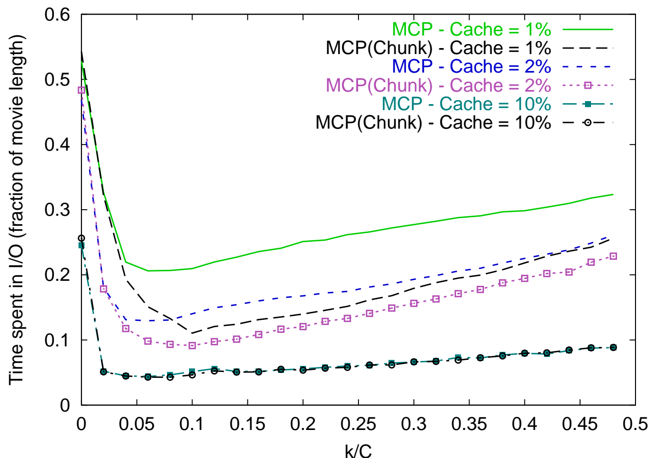

Fig. 8. Relative performance of server-side chunking.

VI. Performance Comparison

Fig. 7 summarizes the performance of MDP and MCS replacement schemes in comparison with the radix-heap-based algorithm and the optimum cache replacement algorithm. It is clearly seen that in the feasible operating regions of Cache size = 0.2 % to 2 % (≈ 6 to 60 MB for typical MPEG-2 movies), our schemes outperform existing schemes by more than a factor of two in disk I/O.

In our implementation of the radix-heap algorithm presented in [9], we have improved its efficiency somewhat by modifying it to take advantage of the entire memory available and write out radix buckets to disk only when this memory is used up: The performance of radix-tries as described by the authors [8] would have been much worse. Array heaps [8], another promising data structure, did not perform better in practice than the radix-heap algorithm in our tests.

A. Computational Complexity

We note that both MDP and MCS are quite easy to implement compared with heap-based schemes, as they use relatively simple structures like lists and arrays. A simple array based implementation of MDP with a cache of size of C frames takes $O(C)$ time to insert a frame and $O(1)$ time to delete κ frames from cache. Note that these operations are in addition to the actual Disk I/O operations which, we assume, happen in the background. Implementing the cache as a balanced binary tree would result in $O(\log C)$ time for inserting and deleting a single frame. A simple array-based implementation of MCS takes $O(C)$ time to insert a frame and $O(C)$ time to replace a run of κ frames. As memory constraints will typically limit C to not more than a few hundred frames, this will be extremely inexpensive compared with the other operations involved.

VII. Server support for Client Caching

The success of MCS depends on finding enough compact runs to write to disk during each replacement. This can be exploited at the server side by altering schedules such that frames whose playout times are close together tend to be transmitted during nearby instants, a technique we refer to as “chunking.”

Specifically, we implement chunking by defining a chunk size H and a time frame $T(f)$ for each frame f, ensuring that no two frames in a chunk are scheduled more than $T(f)$ instants from each other. The time-frame function $T(f)$ is typically a linearly increasing function of frame size and is of the form $\delta_a * (w + f) + \delta_d * w$ where δ_a and δ_d are small constants. More details may be found in our detailed paper on our server-side scheduling scheme [7]. The chunk size, too, is an important parameter: the server's flexibility in scheduling frames over time decreases with increasing chunk size. Empirically, we observe that using chunk sizes that correspond to the client-side cache replacement size κ produces optimal results with no significant impact on the server bandwidth usage. This makes intuitive sense, since the best case for MCS replacement is a chunk of κ consecutive frames.

Because chunking causes each client to receive adjacent frames closely together in time, most iterations of MCS will find clusters of frames that are to be played out near each other. Thus, altering server schedules to support MCS cache replacement at the client results in further gains in the performance of the MCS algorithm. Fig. 8 illustrates the relative performance with and without chunking for various cache and chunk sizes. As can be seen, this effect is most pronounced for small caches $(1 - 2\%)$. For a cache able to hold 10% of the movie, the effect becomes effectively nil. Therefore, Fig. 8 shows the two 10% lines directly on top of each other.

VIII. CONCLUSIONS

Recent scalable MoD systems place a heavy load on the client in terms of buffer space. When designing cheaper end-systems or more scalable proxies, efficient buffer management becomes critical. We have proposed and evaluated some simple schemes drawing upon these principles:

BATCHED I/O: Cache replacement in blocks of multiple frames amortizes seek times over these frames. Empirically, we found replacing $2 \ldots 3\%$ of the cache to be optimal.

USING ACCESS PATTERNS: The knowledge of processing continuous media requiring linear playout access allows the efficient writing of compact sequences to reduce cycling of data between memory and disk.

SERVER-SIDE SUPPORT FOR CLIENT-SIDE CACHING: Chunking helps servers support highly inexpensive clients with minute amounts of main memory and very slow external storage, at a small trade-off in additional bandwidth.

Working in concert, these techniques achieve significant performance gains compared with known algorithms described in the literature. For example, when the cache size is 1% of the movie length, using MCS replacement with chunking reduces disk I/O usage to less than 10% of the time, compared to about 40% of the time with radix heaps. In other common cases, our results, although less impressive, improve performance by at least a factor of two.

We are currently working on improving our heuristics and providing theoretical upper bounds. Additionally, we aim to find out more about optimum cache replacement strategies when the server transmission schedule is known to the clients.

REFERENCES

[1] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin, "Scheduling policies for an on-demand video server with batching," in *Proceedings ACM Multimedia '94*, Oct. 1994, pp. 391–398.

[2] Kien A. Hua and Simon Sheu, "Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems," in *Proceedings of SIGCOMM '97*, Sept. 1997, pp. 89–100.

[3] Li-Shen Juhn and Li-Meng Tseng, "Harmonic broadcasting for video-on-demand service," *IEEE Transactions on Broadcasting*, vol. 43, no. 3, pp. 268–271, Sept. 1997.

[4] Jehan-François Pâris, Steven W. Carter, and Darrel D. E. Long, "Efficient broadcasting protocols for video on demand," in *Proceedings 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, July 1998, pp. 127–132.

[5] Subhabrata Sen, Lixin Gao, and Donald F. Towsley, "Frame-based periodic broadcast and fundamental resource tradeoffs," Tech. Rep. 99-78, University of Massachusetts, Amherst, 1999.

[6] Derek L. Eager, Mary K. Vernon, and John Zahorjan, "Minimizing bandwidth requirements for on-demand data delivery," in *Proceedings of Multimedia Information Systems Conference (MIS '99)*, Oct. 1999.

[7] Ramaprabhu Janakiraman, Marcel Waldvogel, and Lihao Xu, "Fuzzycast: Efficient video-on-demand over multicast," in *Proceedings of Infocom 2002*, New York, NY, USA, June 2002.

[8] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer, "An experimental study of priority queues in external memory," in *Proceedings of the Workshop on Algorithm Engineering*, Berlin, 1999, vol. 1668 of *Lecture Notes in Computer Science*, pp. 345–358, Springer Verlag.

[9] G. S. Brodal and J. Katajainen, "Worst-case efficient external-memory priority queues," in *Proceedings of the Scandinavian Workshop on Algorithms Theory*, Berlin, 1998, vol. 1432 of *Lecture Notes in Computer Science*, pp. 107–118, Springer Verlag.

[10] Gregory Robert Ganger, *System-Oriented Evaluation of I/O Subsystem Performance*, Ph.D. thesis, University of Michigan, Ann Arbor, 1995, Also available as technical report CSE-TR-243-95.