

Enhancing Patching Performance Through Double Patching^{*}

Ying Cai¹

Wallapak Tavanapong¹

Kien A. Hua²

¹ Department of Computer Science
Iowa State University
Ames, IA 50011, U.S.A

² Computer Science Program, SEECs
University of Central Florida
Orlando, FL 32816, U.S.A

Abstract

Patching is an efficient bandwidth-sharing technique for video-on-demand systems. Its performance, however, has limitation: as the time distance to the last regular multicast enlarges, the patching cost for new requests increases and eventually, a new regular multicast must be scheduled to balance the cost. In this paper, we address this problem by proposing a new technique called Double Patching. Our research is based on the observation that a patching stream can be shared by the video requests arriving in the next w_p time units if it delivers an additional $2 \cdot w_p$ time units of video data. With these additional data, the patching cost for these requests can be significantly reduced. Our performance study shows that the new technique can dramatically improve, in many workloads double, the performance of the original Patching. While the performance gain is significant, the new technique inherits the same simplicity from the original Patching. In particular, it does not impose any additional requirement on client download bandwidth - the same as the original Patching, the new scheme allows a client to receive data from no more than two video streams at any one time.

1. Introduction

Traditionally, query results are typically very small in size, and communication bandwidth is usually not a primary concern in designing information systems. This situation is changing with video data becoming more popular. Transmitting a video clip, as the result of a query, requires substantial bandwidth. The number of users that can receive services simultaneously, therefore, depends on the communication capability of the media servers. To achieve better system throughput, we can employ multicast to al-

low clients to share video streams. A technique, called Batching, was introduced in [1]. This approach delays service requests for a particular video, and waits for more to arrive within a batching interval. All these requests can be served together using a single multicast. This strategy significantly reduces the demand on server bandwidth. Each request, however, must wait until the end of the batching period. If we make this period short, the benefit of multicast diminishes. On the other hand, if users are made to wait too long, many of them may renege, again affecting the multicast efficiency.

To address the aforementioned dilemma, clients can join an on-going *regular multicast*, and cache the multicast data in their local disk. To serve these clients, the server initiates a *patching stream* to multicast only the leading portion of the video. When these clients finish playing back the patching stream (i.e., reaching the *skew point*), they continue to play back the remainder of the video using the data already buffered in the local disk. We called this scheme *Patching* in [2]. The benefits of patching are twofold. First, multicasts become more efficient since they can expand dynamically to accommodate future services. Second, since service requests do not have to wait for the batching process, *true* video on demand (VoD), i.e., no service delay, can be achieved.

The duration of the patching streams is the most important determinant of patching performance. They should be short since a patching stream typically serves only few, often one, clients. However, patching streams, too short, would incur many regular multicasts degrading the system performance to that of conventional batching. To determine the optimal duration for each patching stream, optimized schedulers were proposed in [3] and [4]. A different way to reduce patching costs is to enable clients to share patching data. Although various solutions have been proposed, they require either complicate channel management at both server and client sides or significant client receiving bandwidth. For instance, the stream merging algorithms [5] [6] require server to dynamically create merg-

^{*}This research is funded in part by US National Science Foundation under grant ANI-0088026. Contact Email: kienhua@cs.ucf.edu

ing trees and inform clients of which channels to download data. In addition, determining the optimal merging schedule for n video streams has a complexity of $O(n^3)$ and requires accurate prediction on the arrival time of video requests [5]. Sharing patching data is also realized in [7] and [8] by having the clients to receive data from three or more streams simultaneously. This scheme requires clients with substantial communication bandwidth making it unsuitable for many applications.

In this paper, we introduce a new technique that has the same implementation cost as the standard patching scheme while at any one time, the number of video streams a client needs to download is no more than two. Our solution is based on the following observation: a patching stream can be shared by the video requests arriving within the next w_p time units if it delivers an extra $2 \cdot w_p$ time units of data. In specific, given a client which is t time units late for a regular multicast, we can schedule a *long patching stream* to deliver $t + 2 \cdot w_p$ time units of data instead of just t time units as in standard patching. With these extra $2 \cdot w_p$ time units of video data, the cost to serve a request arriving in the next Δt time units, where $0 \leq \Delta t \leq w_p$, is just a *short patching stream* delivering only the first Δt time units data of the video. Under standard patching, the cost would be $t + \Delta t$ time units of the video data. The client can receive the entire video as follows. It first uses its two download channels to receive the video data from the short and long patching streams, respectively. After the short patching stream is exhausted, the same download channel is then switched to receive data from the regular multicast stream. Thus, two channels are used to download three segments from three different video streams. We prove in this paper that the three video segments make up of the entire video for the client. We call this strategy *Double Patching*, alluding to the fact that the patching cost can be dramatically reduced by using two patching streams to patch up a client time distance to a regular multicast.

The remainder of this paper is organized as follows. We describe Standard Patching in Section 2 and the details of Double Patching in Section 3. In Section 4, we discuss the performance optimization. We present our performance study in Section 5. In Section 6, we give our concluding remarks.

2. Standard Patching

In Standard Patching, when a free channel becomes available, a new video stream is scheduled to deliver a requested video. The new video stream can be a *regular stream* or a *patching stream*. A regular stream multicasts the video in its entirety while a patching stream transmits

only the first t time units of the video, where t is the temporal distance to the starting time of the last regular multicast of the same video.

To support Standard Patching, each client station needs to have three threads of control: two data loaders and a video player. Two scenarios can occur:

- If a client is served by a new regular stream, only one data loader is necessary to receive the video data. The data packets arriving at the client are piped directly to the video player for rendering onto the screen.
- In the case that a patching stream is scheduled, both of the client's loaders must be used to receive data from the patching stream and the latest regular stream simultaneously. In this case, the video player starts playing back the video data from the patching stream first while the data packets from the regular stream are temporarily cached in the client buffer. When the patching stream ends, the video player switches to play back the data in the disk buffer as the remaining data packets continue to arrive in the regular stream.

In Standard Patching, a patching stream is scheduled for a new request if the temporal distance between its arrival time and the starting time of the last regular stream of the same video is less than a certain threshold called *patching window* [3]; otherwise, a new regular stream is initiated, and all subsequent requests arriving within the next patching window will be served by patching streams. A detailed analysis on how to determine the optimal patching window to minimize server bandwidth requirement can be found in [9].

3. Double Patching

Standard Patching is very economic in delivering patching data. Each patch is just long enough to cover the missing portion of the video. In this section, we present a better patching technique called Double Patching. In this new scheme, a patching stream may transmit more than just the regular patching data (i.e., *long patching stream*). At first sight, this seems to be a waste of system resources. This small "waste", however, can significantly reduce the patching costs of the subsequent services.

As a motivating example, let us consider a regular stream of video v starting at time 0. At time t , a patching stream, S_a , is initiated for client A . One time unit later, another patching stream, S_b , is scheduled for client B . If we use $v[t_1, t_2]$ to denote the video segment from time t_1 to time t_2 assuming the beginning of the video as time zero, then the patches required by clients A and B are $v[0, t]$ and

$v[0, t + 1]$, respectively. In total, the server delivers $2 \cdot t + 1$ time units of data specifically for these two clients. Alternatively, if we make S_a deliver a long patching stream $v[0, t + 2]$, a patch with two extra time units of data, then client B can share stream S_a with client A as follows:

- Stream S_b needs to transmit only the first time unit of the video, $v[0, 0]$. As client B receives this segment using its loader L_1 , the client immediately plays back the video. At the same time, it downloads $v[1, t + 2]$ in stream S_a using loader L_2 .
- When stream S_b ends, client B starts to play back the second video segment $v[1, t + 2]$, as L_1 switches to download the remaining data, $v[t + 3, |v|]$ (where $|v|$ is the length of the video), being transmitted in the regular stream.
- Finally, when the playback of $v[1, t + 2]$ is completed, client B continues to play back the third video segment $v[t + 3, |v|]$.

This strategy is referred to as Double Patching in this paper. We note that the three video segments, $v[0, 0]$, $v[1, t + 2]$, and $v[t + 3, |v|]$, make up the complete video for client B . In total, the server delivers $(t + 2) + 1$ time units of data specifically for these two clients. Therefore, the saving in comparison with Standard Patching is $\frac{(2 \cdot t + 1) - (t + 3)}{2 \cdot t + 1} = \frac{t - 2}{2 \cdot t + 1}$. When the time skew t is large, this saving is approximately 50%. We note that Double Patching requires only two data loaders as in Standard Patching. Thus, it improves the standard technique without incurring additional system costs.

3.1 Server Design

Without loss of generality, we assume that the server has only one video. As in Standard Patching, the server bandwidth is multiplexed into a set of logical channels, each capable of sustaining a video stream transmitting data at the playback rate. These channels are used to deliver three types of video streams: regular streams, long patching streams, and short patching streams. Depending on the type of the stream currently riding on a given channel, it is referred to as a *regular channel*, an *long patching channel*, or a *short patching channel*. When a channel is dispatched, it is given a workload, specified as $v[t]$, where v and t denote the unique ID of the video file and the desired playback duration, respectively. For instance, $v[3]$ indicates that the free channel should deliver only the first three time units of video v . After the transmission is completed, the channel again becomes *free*, and is available for the next service.

The stream scheduling policy is as follows. The very first request for a given video is serviced using a regu-

lar stream. For all requests arriving within the next w_m time units, either a long patching stream or a short patching stream is scheduled. w_m is called a *multicast window*, and is the minimal scheduling distance between any two consecutive regular streams. A regular stream and its following long and short patching streams initiated before the next regular stream are said to form a *multicast group*. After a regular stream, short patching streams are scheduled for requests arriving within the next w_p time units. These consecutive short patching streams form a *patching group*. The threshold w_p is called *patching window*, and is set to be no larger than w_m . We note that this patching window is different from the patching window in Standard Patching. If $w_p = w_m$, Double Patching becomes Standard Patching. In general, w_m is many times greater than w_p . After w_p time units, a long patching stream is scheduled for the next service request. This stream “leads” the next patching group which consists of short patching streams scheduled for requests arriving within the next w_p time units. This pattern is repeated for each of the subsequent patching groups until a new request whose time skew to the regular stream of the current multicast group is greater than w_m , in which case a regular stream is initiated to start a new multicast group. The scheduling pattern for one multicast group is illustrated in Figure 1. In the next section, we will determine the values of w_m and w_p such that the mean server bandwidth required to support true on-demand services is minimized.

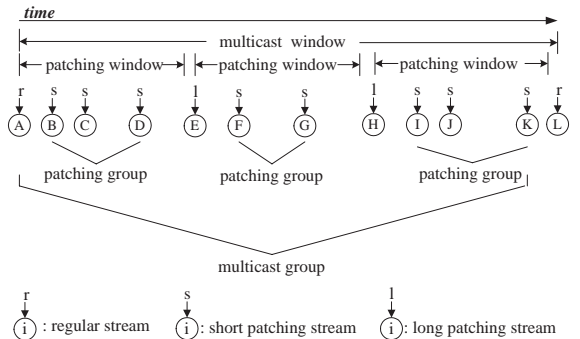


Figure 1. Pattern of stream initiation

A client makes a service request by sending a request token ($ClientID$) to the server, where $ClientID$ is its unique network address. We present the algorithm for the server software in Figure 2. This procedure is executed when a channel becomes free. The server notifies the client by sending a service token, (ID_{sp} , ID_{lp} , ID_r), where ID_{sp} , ID_{lp} , and ID_r are the IDs of the short patching channel, the long patching channel, and the regular channel, respectively. This token informs the client the channels it should use to receive data for its requested video. We note in this procedure that if the free channel

is to deliver a long patching stream, its workload is set to $v[\text{skew} + 2 \cdot w_p]$, where skew is the current time distance to the beginning of the last regular stream.

Algorithm: *Server Main Routine*

LastRegular: the last regular channel ID
LastLPatching: the last long patching channel ID
skew(c): the time skew to the start time of the current stream on channel c

- Initialize service token as ($ID_{sp} = \text{null}$, $ID_{lp} = \text{null}$, $ID_r = \text{null}$);
- Dispatch a free channel, say *FreeChannel*;
- If *LastRegular* is *null* or $\text{skew}(\text{LastRegular}) > w_m$, then
 - Set $ID_r = \text{FreeChannel}$ and its workload to be $v[|v|]$;
 - Set *LastRegular* = *FreeChannel*.
- Otherwise,
 - If $\text{skew}(\text{LastLPatching}) > w_p$, then
 - * Set $ID_r = \text{LastRegular}$;
 - * Set $ID_{lp} = \text{FreeChannel}$ and its workload to be $v[\text{skew}(\text{LastLPatching}) + 2 \cdot w_p]$;
 - * Set *LastLPatching* = *FreeChannel*.
 - Otherwise,
 - * Set $ID_r = \text{LastRegular}$;
 - * Set $ID_{lp} = \text{LastLPatching}$;
 - * Set $ID_{sp} = \text{FreeChannel}$ and its workload to be $v[\text{skew}(\text{LastLPatching})]$.
- Send token (ID_{sp} , ID_{lp} , ID_r) to clients
- Activate *FreeChannel*.

Figure 2. Algorithm for Video Server

3.2 Client Design

A client uses two data loaders, L_1 and L_2 , to receive data and write them to a local buffer. Another thread, *VideoPlayer*, fetches the corresponding data packets from the buffer, reassembles the video frames, and renders them onto the screen. As discussed in the server design, a client sends its own network address as part of the token for a service request. It then waits for the service token (ID_{sp} , ID_{lp} , ID_r). When it arrives, the client examines the values of ID_{sp} and ID_{lp} and acts accordingly as follows:

1. $ID_{sp} = \text{null}$ and $ID_{lp} = \text{null}$: This combination indicates that the server is about to start a new regular stream on channel ID_r . The client needs only use one loader, L_1 , to receive the entire video.
2. $ID_{sp} \neq \text{null}$ and $ID_{lp} = \text{null}$: This combination indicates that the server is about to start a short patching stream on channel ID_{sp} . This scenario happens when the client time skew to the last regular stream is within w_p time units. In this case, the client must activate both loaders L_1 and L_2 in order to download

data from the short patching channel ID_{sp} and the regular channel ID_r simultaneously.

3. $ID_{sp} = \text{null}$ and $ID_{lp} \neq \text{null}$: This combination indicates that the server is about to start a new long patching stream. In this case, the client needs to activate both loaders L_1 and L_2 to download data from the long patching channel ID_{lp} and the regular channel ID_r simultaneously.
4. $ID_{sp} \neq \text{null}$ and $ID_{lp} \neq \text{null}$: This combination indicates that the server is about to start a short patching stream on channel ID_{sp} . This case happens when the client's time skew to the last regular stream is greater than w_p time units, while to the last long patching stream is less than w_p time units. Under this circumstance, the client uses loader L_1 to download data first from the short patching channel ID_{sp} , and then from the regular channel ID_r . In parallel with these activities, L_2 is used to receive data on the long patching channel ID_{lp} . The data stream arriving on channel ID_{lp} is used to bridge the gap between the two streams coming on channels ID_{sp} and ID_r , respectively. These three data segments make up the entire video.

The client's *VideoPlayer* can start as soon as L_1 begins downloading data. For the first three cases, it is trivial to see that the playback is continuous and complete since the client is served by at most two data streams, and it has two loaders to keep up with the incoming data. The fourth case is less obvious; and we discuss it as follows.

Let us say a long patching stream for a video v started t time units after the last initiation of a regular stream. According to the server routine (Figure 2), this long patching stream transmits the video segment $v[0, t + 2 \cdot w_p]$. If a client arrives p time units after this stream begins, then the short patching stream initiated for this client, according to the server routine, delivers the video segment $v[0, p]$. This p time units of data is received by loader L_1 and directly piped to the *VideoPlayer* for display. At the same time, L_2 receives the video segment $v[p + 1, t + 2 \cdot w_p]$ from the long patching stream. The corresponding data packets are cached in the client's local disk. After p time units, *VideoPlayer* starts consuming data in this disk buffer while L_1 switches to download the video segment $v[t + 2 \cdot p + 1, |v|]$ on the regular channel. Since the server routine ensures that $p \leq w_p$, the three downloaded segments, $v[0, p]$, $v[p + 1, t + 2 \cdot w_p]$, and $v[t + 2 \cdot p + 1, |v|]$, are sufficient to make up the entire video.

4 Performance Optimization

In the above discussion, an important issue remains unsolved: what should the sizes of the multicast window w_m and the patching window w_p be? We say the values of w_m and w_p are optimal if they result in minimal requirement on the server bandwidth in providing true on-demand services, i.e., no service delay. To determine their optimal values, we derive a mathematical formula to capture the relationship between their sizes and the required server bandwidth. We recall that a regular stream and the following streams initiated before the next regular stream are said to form a *multicast group*. The mean server bandwidth requirement can then be computed as $\frac{D}{\tau}$, where D is the mean total amount of data transmitted by a multicast group and τ is the average time duration τ between two consecutive multicast groups. Note that D is equal to the summation of D_r , D_{lp} , and D_{sp} , where D_r , D_{lp} , and D_{sp} denote the mean total amount of data delivered by the regular stream, the long patching streams, and the short patching streams, respectively, in one multicast group. They can be calculated as follows.

First, the total amount of video data delivered by the regular stream, D_r is equal to $|v|$, where $|v|$ is the playback length of the video, since there is only one regular stream in each multicast group.

The value of D_{lp} , the total amount of video data delivered by the long patching streams, can be determined as follows. Within one multicast group, after the initiation of the regular stream, the mean interval of initiating a new long patching stream is $w_p + \frac{1}{\lambda}$ time units, where λ is the request arrival rate. This is due to the fact that a long patching stream is always scheduled for the first request arriving w_p time units after the initiation of the regular stream or the previous long patching stream in the same multicast group. Therefore, on average there are $\lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor$ long patching streams in one multicast group. Since a long patching stream delivers the first $t + 2 \cdot w_p$ time units of the video data if its time skew to the regular stream is t time units, we have

$$D_{lp} = \sum_{n=1}^{\lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor} [n \cdot (w_p + \frac{1}{\lambda}) + 2 \cdot w_p].$$

For the value of D_{sp} , the total amount of video data delivered by the short patching streams, we analyze as follows. We recall that the short patching streams in one multicast group can be organized into patching groups. On average, there are $\lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor + 1$ patching groups inside a multicast group. We note that except the last one, each

patching group delivers the same mean amount of video data as the patching window w_p is fixed. The amount of data delivered by such a patching group can be analyzed as follows. When a short patching stream is scheduled for a skew of t time units, it delivers a patch of t time units. Thus, if k short patching streams are initiated between t and $t + \Delta t$, the amount of data delivered by these streams can be approximated as $k \cdot t$, if Δt is small enough. If we use $P(k, T)$ to denote the probability of initiating exactly k short patching streams during a time interval of T , then the total amount of data delivered by the short patching streams initiated between t and $t + \Delta t$ can be computed as $\sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$. As a result, if we partition the patching window w_p into $\lfloor \frac{w_p}{\Delta t} \rfloor$ small time slices, then the mean amount of video data delivered by this group of short patching streams can be calculated as $\sum_{t=1}^{\lfloor \frac{w_p}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$. For the short patching streams in the last group, as they are started for the requests arriving in the last $w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + 1/\lambda)$ time units, the amount of data they deliver is equal to $\sum_{t=1}^{\lfloor \frac{w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + 1/\lambda)}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$ time units. Therefore, the total amount of data delivered by the short patching streams in one multicast group can be calculated as

$$D_{sp} = \lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor \cdot \sum_{t=1}^{\lfloor \frac{w_p}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t) + \sum_{t=1}^{\lfloor \frac{w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + \frac{1}{\lambda})}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t).$$

We assume that the request arrival process is Poisson with rate λ . The probability density function is $f_t = \lambda e^{-\lambda x}$, for $x \geq 0$, where t is the random variable representing the time interval of two successive requests. Under this assumption, $P(k, t) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$ and $\sum_{k=1}^{\infty} k \cdot P(k, t) = t \cdot \lambda$. If we make Δt equal to 1 second and keep the precision of w_m and w_p to second, then we have

$$D_{sp} = \lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor \cdot \sum_{t=1}^{w_p} t \cdot \lambda + \sum_{t=1}^{w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + \frac{1}{\lambda})} t \cdot \lambda.$$

Since the mean interval of initiating two consecutive regular streams is equal to $w_m + \frac{1}{\lambda}$, the mean server bandwidth requirement is

$$Bandwidth_{DoublePatching} = \frac{D_r + D_{lp} + D_{sp}}{w_m + \frac{1}{\lambda}} \cdot b, \quad (1)$$

where b is the playback rate of the video. The above formula can be used to determine the optimal sizes for the multicast window w_m and the patching window w_p .

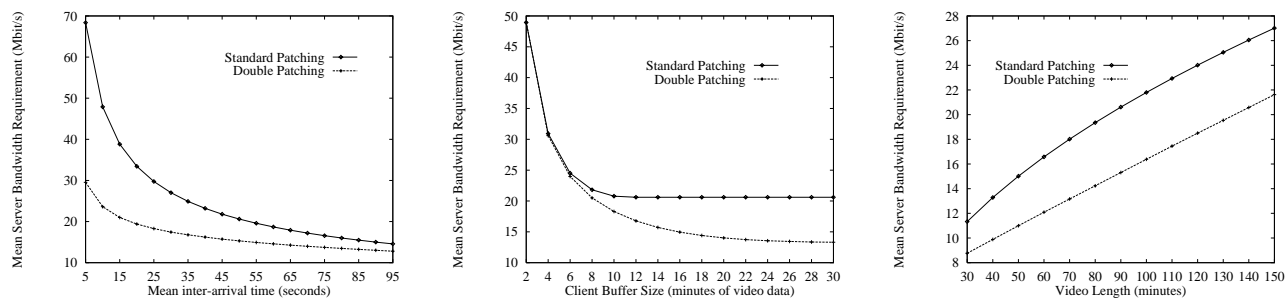


Figure 3. Performance Study Results

5. Performance Study

We compare the performance of Double Patching with that of Standard Patching. The mean bandwidth required to achieve true on-demand services is used as the performance metric. The mean server bandwidth required by Double Patching can be computed using the formulas presented in Section 4. Similarly, the same can be computed for Standard Patching using the formulas we derived in [3]. In our simulation, the video is assumed to be encoded using MPEG-1 with an average playback rate of 1.5 Mbits/sec. The effect of the *inter-request arrival time*, *client buffer size*, and *video length* on the mean server bandwidth requirement is showed in Figure 3. Because of space constrain, we omit the discussion of these figures.

6 Concluding Remarks

In this paper, we introduced a technique, called *Double Patching*, to address the performance limitation of our initial patching idea. Under Standard Patching, requests arriving within a patching window are able to share the same multicast using a patching stream. As time elapses, these patching streams need to “patch” more data, and therefore incur a higher communication cost. To optimize system performance, Standard Patching needs to multicast fairly frequently to keep the patching cost low. This limits data and bandwidth sharing to a small group of service requests arriving within a relatively small patching window. Double Patching overcomes this constraint by using long patching streams. Although they are slightly more expensive than a standard patching stream, they allow subsequent requests including those arriving very late to share a multicast using a short patching stream. The advantages of this strategy are twofold. First, short patching streams are significantly less expensive than the standard patching stream. Second, many more service requests can share a multicast. We note that these advantages are achieved without additional implementation cost in term of client download bandwidth

requirement - the same as in Standard Patching, the new technique requires a client to download data from no more than two channels simultaneously.

To maximize the performance potential of Double Patching, we developed a probabilistic model to analyze its performance. This analysis allowed us to design an optimal scheduler for the Double Patching approach. As we have shown in [2] that Standard Patching significantly outperforms conventional multicast (i.e., batching), we focus on comparing Double Patching with Standard Patching in this paper. Our simulation results indicate that the new method is significantly better. The performance improvement is more than double for some of the workloads.

References

- [1] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic batching policies for an on-demand video server. *Multimedia Systems*, 4(3):112–121, June 1996.
- [2] K. A. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. In *Proc. of ACM Multimedia*, pages 191–200, Bristol, U.K., September 1998.
- [3] Y. Cai, K. A. Hua, and K. Vu. Optimizing patching performance. In *Proc. of SPIE’s Conf. on Multimedia Computing and Networking (MMCN’99)*, pages 204–216, San Jose, CA, USA, January 1999.
- [4] L. Gao and D. Towsley. Supplying instantaneous video-on-demand services using controlled multicast. In *Proc. IEEE International Conference on Multimedia Computing and Systems*, pages 117–121, Florence, Italy, June 1999.
- [5] D. L. Eager, M. K. Vernon, and J. Zahorjan. Optimal and efficient merging schedules for video-on-demand servers. In *Proc. of ACM Multimedia’99*, pages 199–202, Orlando, FL, USA, October 1999.
- [6] A. Bar-Noy and R. E. Ladner. Competitive on-line stream merging algorithms for media-on-demand. In *Proc. of 12th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 364–373, 2001.
- [7] S. Sen, L. Gao, J. Rexford, and D. Towsley. Optimal patching schemes for efficient multimedia streaming. In *Proc. IEEE NOSS-DAV’99*, Basking Ridge, NJ, U.S.A., June 1999.
- [8] C. Griwodz, M. Liepert, M. Zink, and R. Steinmetz. Tune to lambda patching. In *2nd Workshop on Internet Server Performance (WISP’99)*, Atlanta, GA, U.S.A., May 1999.
- [9] Y. Cai and K. A. Hua. Sharing multicast videos using patching streams. *Multimedia Tools and Applications*, to appear.