CLIENT-SIDE CACHE MANAGEMENT FOR SCALABLE AND INTERACTIVE VIDEO STREAMING

Kamal K. Nayfeh and Nabil J. Sarhan

Electrical and Computer Engineering
Wayne State University
Detroit, Michigan 48202
knayfeh,nabil@wayne.edu

1. ABSTRACT

The design of interactive Near Video-on-Demand (NVOD) systems is highly complicated when scalable stream merging is used. We propose an intelligent client-side cache management policy for these systems, allowing and exploiting cache discontinuity. This policy maximizes the percentage of interactive requests serviced from the client's own cache without requiring any resources from the server. The policy caches data from all streams that are being listened to by the client. As the cache becomes full, it purges data according to a purging algorithm. We present three purging algorithms: purge oldest data, purge the furthest data from the customer's playback point, and purge adaptively. Moreover, we experiment with another important decision, which is whether pausing users should continue to listen to streams when the cache becomes full. We evaluate the effectiveness of the proposed cache management policy and purging algorithms under realistic and complex workload through extensive simulations. We analyze many metrics, including waiting and blocking metrics, aggregate delay, cache hit rate, and cache fragmentation, considering a variety of system parameters.

Index Terms— Cache management, interactive requests, stream merging, Near Video-On-Demand (NVOD), multicast, realistic workload, video streaming.

2. INTRODUCTION

The high network and server requirements limit the scalability of media streaming systems. The most common approaches used to address the scalability challenge are Content Distribution Networks [1] and Peer-to-Peer [2]. While the first approach requires maintaining a huge number of geographically distributed servers [3], the second still relies heavily on central servers [4]. Both of these approaches mitigate the scalability problem but do not eliminate it because the fundamental problem is due to unicast delivery [4]. As multicast is highly effective in delivering high-usage content, there has been a growing interest in enabling native multicast to support high-quality on-demand video distribution, IPTV, and other major applications [4]. Moreover, multicast is enabled in a variety of fully owned networks, including Cable TV and university networks. This paper considers the multicast-based approach. Stream merging techniques [5, 6, 7] utilize the multicast facility by combining streams to reduce the cost per request.

The design of interactive Video-on-Demand (VOD) systems supporting stream merging is highly challenging. Although most prior studies assumed simple workload, characterization studies [8, 9, 10] show that the workload contains many interactive requests and the customer behavior varies with video length. Further design complications happen in practical systems, which cannot service all requests immediately. This model of operation is referred to as Near VOD (NVOD) and is the general case of True VOD (TVOD). Study [11] is the only study of interactive NVOD with stream merging, but deals only with the server side.

This paper is the first that considers the problem of client-side cache management in interactive NVOD system employing stream merging. Cache management was studied either at the server side of VOD systems [12, 13, 14] or at the client side in non-scalable system with no stream merging support [13]. We propose a novel cache management policy, which allows and exploits cache discontinuity. Existing systems even those with no multicast support, such as YouTube, do not allow for such discontinuity. Any interactive request to outside the local cache causes the client to clear the entire cache. The policy caches data from all streams types. As the cache becomes full, the policy purges data according to a purging algorithm. We present three purging algorithms: *Purge Oldest*, *Purge Furthest*, and *Adaptive Purge*. Purge Oldest removes the oldest data in the cache, whereas Purge Furthest clears the furthest data from the client's playback point. In contrast, Adaptive Purge tries to avoid purging any data that includes the customer's playback point or the playback point of any stream that is being listened to by the client. We experiment with another important decision, which is whether pausing users should continue to listen to streams when the cache becomes full.

We evaluate the effectiveness of the proposed cache management policy and purging algorithms under realistic and complex workload through extensive simulations. We analyze many metrics, including waiting and blocking metrics, aggregate delay, cache hit rate, and cache fragmentation. The aggregate delay is the average delay experienced by a client due to both waiting and blocking, incorporating the impact of increased customer frustration with every subsequent blocking during the same session. To quantify cache fragmentation, we introduce two metrics: *average number of segments* and *average gap length between consecutive segments*. We consider the impact of cache size, purge block size, and server capacity.

The main contributions of this paper can be summarized as follows. (1) We propose a novel client-side cache management policy that maximizes the percentage of interactive requests from the local cache without requiring additional resources from the server. The proposed policy can be easily generalized to work with any VOD system, not just when stream merging is utilized. (2) We introduce three purging algorithms. (3) We extensively evaluate the effectiveness of the proposed policy in terms of various metrics under realistic workload, considering several important parameters.

The rest of this paper is organized as follows. Section 3 discusses the background information and related work. Section 4 presents the proposed policy. Section 5 discusses the performance evaluation methodology. Finally, Section 6 presents and analyses the main results.

3. BACKGROUND INFORMATION

3.1. Stream Merging

The main stream merging techniques are Patching and Earliest Reachable Merge Target (ERMT). They classify customers into groups of requests for the same streams. Patching [7] (and references within) joins new requests into the latest multicast stream for the same video. The server sends the missing portion to the new requests using a stream, called a *patch*. When the client finishes playing the patch stream, the client continues playing from the already cached data. It is more cost effective to start a full stream after some time. ERMT [5] is a near optimal stream merging technique. A new client or a newly merged group of clients listens to the closest stream it can merge with if no later arrivals can preemptively catch them. The target stream can later get extended to satisfy the needs of the new client (only after the merging stream finishes and merges with the target), and this extension can affect its own merge target.

3.2. Support for Interactive Requests

Study [11] explains the different methods by which the interactive requests are serviced. Since pause and resume requests do not require any additional resources, they are serviced immediately. Upon a pause request, the customer continues to listen to the stream and buffer data if it is a B-Stream until the buffer is full. The customer's jump requests are serviced by one of the following methods. (1) The customer utilizes the cache, which may include past and future data. (2) The server merges the request with an already existing full-length multicast (B-Stream) or multicast patch stream (P-Stream) for the same video. For a successful merge, the target stream playback point should be within a certain tolerance, called *playback point deviation tolerance*, of the target playback point. (3) The server utilizes an available unicast interactive stream (I-Stream) using the Split and Merge (SAM) protocol [12]. In particular, the client is split from the multicast stream and is temporarily assigned an I-Stream to perform the interaction. Once the interaction is complete, the client is merged into an on-going stream.

If an interaction cannot be serviced, the request is placed in the blocking queue. Once a customer's request blocks, the customer stops listening to all streams. The server services blocked requests by merging them with other B-Streams or by using I-Streams.

4. PROPOSED SOLUTION

4.1. Considered System

The system consists of a NVOD server, streaming clients, and a network connecting them. The major components of the NVOD server are waiting queues (with one queue for each video), a blocking queue, a queuing manager, a dynamic I-Stream allocation module, a SAM technique, a stream merging protocol, a waiting scheduling policy, a blocking scheduling policy, and streams. The streams are divided into B-Streams, P-Streams, and I-Streams. B-Streams and P-Streams are used to service waiting customers and can be shared with other requests for the same video.

The customer starts a streaming session by issuing a request to playback a certain video. The server decides whether to service the request based on the availability of server channels. A channel is the set of required server resources for delivering a video stream. The number of such channels is called *server capacity*. If adequate channels are available, the request is delivered using a stream merging technique (Patching or ERMT); otherwise, the request is placed in the waiting queue for that video. The server applies a waiting scheduling policy to determine which waiting queue to service when streams become available. The most popular policies are *First Come First Serve* (FCFS) [15], *Maximum Queue Length* (MQL) [15], and *Minimum Cost First* (MCF) [16] and references within. FCFS selects the queue with oldest request and is thus the fairest, whereas MQL tries to maximize the number of requests that can be serviced with one channel by selecting the queue with the largest number of requests. In contrast, MCF selects the queue with the minimum cost in terms of stream length.

During a streaming session, the customer issues interactive requests to pause the video, jump forward, or backward by a certain amount relative to the current playback position. Interactive requests are serviced in the manner discussed in Subsection 3.2. The server applies a blocking scheduling policy to service blocked customers when server channels become available. The waiting and blocking scheduling policies are not necessarily the same. Waiting and blocked customers defect when the waiting/blocking time exceeds their tolerance.

4.2. Cache Management

The main contribution of this study is to propose an intelligent client-side cache management policy to maximize the cache utilization for servicing interactive requests by allowing and exploiting cache discontinuity. Gaps in the cached continuous data are allowed (e.g. minutes 1-2 and 3-4 of the video can be in the cache without 2-3). The client's cache consists of one or more segments of data. Each segment is a contiguous set of video data. The data kept in the cache can be future data for stream merging techniques and past data the customer watched already. All stream types including full-length B-Streams, Patch Streams (P-Streams), and Interactive Streams (I-Streams) are used to fill the cache. Both future and past data are utilized to service interactive requests. When two segments overlap, they are merged to form one segment.

A cache hit happens when an interactive request is serviced from the cache. Hence, the *cache hit rate* is defined as the probability that the interactive request is serviced from the cache. To assess the cache discontinuity, we introduce two new metrics: the *average number of segments* and *the average gap length*. The first is the customer's average number of segments during a streaming session, whereas the second is the average length (in seconds) between each two consecutive segments during a customer's session.

Figure 1 shows an example of a customer's cache for a system employing Patching. The cache consists of two segments, which are currently being filled from two different streams. The customer is listening to two streams at time i: one patch stream (Stream1) delivering the current data, and a full-length stream (Stream2) delivering future data. The customer's playback point is the same as that of Stream1. At time $i+1$, both segments have cached data. The two segments overlap at time $i+2$, and thus get merged into one bigger segment encompassing the data in both. At the same time, Stream1 fin-

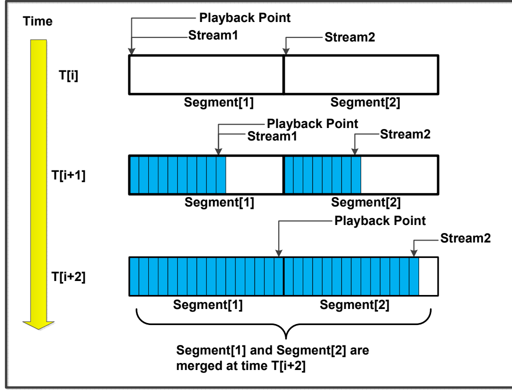


Fig. 1: Clarification of Cache Structure

ishes delivering all its data. Any interactive request changes the customer's playback point, and thus it becomes different from the stream(s) that it is listening to, if it is not already so.

```
//Handle all paused customers
HandlePause(Customer) {
  //Is customer listening to I_Stream?
  if (Customer.Stream == I_STREAM) {
    //Stop listening to I-Stream and stop caching
    StopListeningToIStream(Customer);
    StopCaching(Customer); }
  //Is cache greater than max allowed size?
  if (CacheSize ≥ MAX_CACHE_SIZE) {
    //Should we continue to listen to streams?
    if (C2L == Yes) {
      KeepListeningToStream(Customer);
      KeepCaching(Customer); }
    else {
      //Stop caching and stop listening to stream
      StopCaching(Customer);
      StopListeningToStream(Customer); } }
  AdjustCache(Customer); } //HandlePause
HandlePlay(Customer) { //Handle all playing customers
    AdjustCache(Customer); } //HandlePlay
AdjustCache(Customer) {
  //Is customer listening to first stream?
  if (Customer.Stream1 == Yes) {
    //Advance the cache data in segment 1
    AdvanceCache(Customer, segment1); } //if
  //Is customer listening to second stream?
  if (Customer.Stream2 == Yes) {
    //Advance the cache data in segment 2
    AdvanceCache(Customer, segment2); } //if
  for (s = 0; s < NumSegments; s++){
    if (Segment[s]overlapsSegment[s+1]) {
      Merge(Segment[s], Segment[s+1]); } }
  PurgeAlg(Customer); } //AdjustCache
```

Fig. 2: Simplified Algorithm for Cache Management

Figure 2 illustrates the cache management policy for customers in the Pause and Play states. Since Jump requests are either serviced immediately or cause the customer to block, the caching algorithm does not deal with jump requests and has no Jump state. Jump requests, however, cause a shift in the customer's playback point within the cache.

When the cache becomes full, data is purged according to a *purging algorithm*. To avoid fragmenting the cache, the data is either purged from the segment tail (i.e., the oldest data in the segment) or head (i.e., the most recent data). The data is purged in the unit of *Purge Block*, which we define as the minimum amount of data that can be purged at one time. We present three purging algorithms *Purge Oldest*, *Purge Furthest*, and *Adaptive Purge*. Purge Oldest purges the tail of the oldest segment in the customer's cache, whereas Purge Furthest purges the furthest data from the customer's playback point. The furthest data could be the segment's tail or head. In contrast, Adaptive Purge tries to avoid purging data that includes the customer's playback point or the playback point of any stream that is currently being listened to by the client.

```
AdaptivePurge(Customer) {
  int Cond1, Cond2, Cond3;
  CacheSize = CalcCacheSize(Customer);
  if ((CacheSize − MAX_CACHE_SIZE) ≥ 0) {
    for (s = 0; s < NumSegments; s++){
      //Is customer's Playback Point inside segment tail or head
      if (CustPlay ≤ Seg[s].start) and (CustPlay ≥ Seg[s].end)
        Cond1 = BOTH;
      else if (CustPlay ≤ Seg[s].start)
        Cond1 = TAIL;
      else if (CustPlay ≥ Seg[s].end)
        Cond1 = HEAD;
      //Is First Stream Playback Point inside segment tail or head
      if (Strm1Play ≤ Seg[s].start) and (Strm1Play ≥ Seg[s].end)
        Cond2 = BOTH;
      else if (Strm1Play ≤ Seg[s].start)
        Cond2 = TAIL;
      else if (Strm1Play ≥ Seg[s].end)
        Cond2 = HEAD;
      //Is Second Stream Play Point inside segment tail or head
      if (Strm2Play ≤ Seg[s].start) and (Strm2Play ≥ Seg[s].end)
        Cond3 = BOTH;
      else if (Strm2Play ≤ Seg[s].start)
        Cond3 = TAIL;
      else if (Strm2Play ≥ Seg[s].end)
        Cond3 = HEAD; } }
  CalcPurgeData(Customer);
} //AdaptivePurge
CalcPurgeData(Customer) {
  for (s = 0; s < NumSegments; s++){
    //Find which segment to purge
    PurgeSeg = GetOldestSegment(Customer)
    //Determine how much to purge
    Extra = CacheSize − MAX_CACHE_SIZE;
    PurgeData = PURGE_BLK * ceil(Extra/PURGE_BLK);
    //Purge the segment tail
    Purge(Customer,TAIL, PurgeData, PurgeSeg); } //for
  //Calculate the new cache size
  CacheSize = CalcCacheSize(Customer);
  //Is cache size within limit
  if ((CacheSize − MAX_CACHE_SIZE) ≤ 0) return;
  for (s = 0; s < NumSegments; s++){
    //Find the furthest segment from the customer playback
    //And whether to purge the segment head or tail
    PurgeSeg = GetFurthestSegment(Customer, HeadTail)
    //Determine how much data to purge
    Extra = PURGE_BLK * ceil(Extra/PURGE_BLK);
    //Purge the segment
    DoPurge(Customer,HeadTail, Extra, PurgeSeg);
  //Calculate the new cache size
  CacheSize = CalcCacheSize(Customer);
  //Is cache size within limit
  if ((CacheSize − MAX_CACHE_SIZE) ≤ 0) return; } }
DoPurge(Customer, HeadTail, Extra, s) {
  //Purge the segment head or tail?
  if (HeadTail == TAIL) and (Cond1 == TAIL)
  and (Cond2 == TAIL) and (Cond3 == TAIL) {
    //Purge from the segment tail
    if (Seg[s].Size ≥ Extra)
      Seg[s].start = Seg[s].start + Extra
    else
      Remove(Seg[s]); } //if
  if (HeadTail == HEAD)and(Cond1 == HEAD)
  and (Cond2 == HEAD) and (Cond3 == HEAD) {
    //Purge from the segment head
    if (Seg[s].Size ≥ Extra)
      Seg[s].start = Seg[s].end − Extra
    else
      Remove(Seg[s]); }} //DoPurge
```

Fig. 3: Simplified Algorithm for Adaptive Purging Algorithm

Figure 3 illustrates the operation of the Adaptive Purge algorithm. It starts by evaluating three conditions for every segment in the customer's cache. Condition 1, Condition 2, and Condition 3 are whether the playback point of the customer, the playback point of the first stream the customer is listening to (if any), or the playback point of the second stream the customer is listening to (if any), respectively, allow the purging of segment tail (TAIL), head (HEAD), or both (BOTH). A condition allows for purging a segment portion (tail, head, or both) if the corresponding playback point does not fall inside that segment portion. If the customer's playback or the playback point of any stream that is being listened to by the client is within the purged data, the segment is split into two segments: one segment is the playback point purged, and the other is the rest of the segment data.

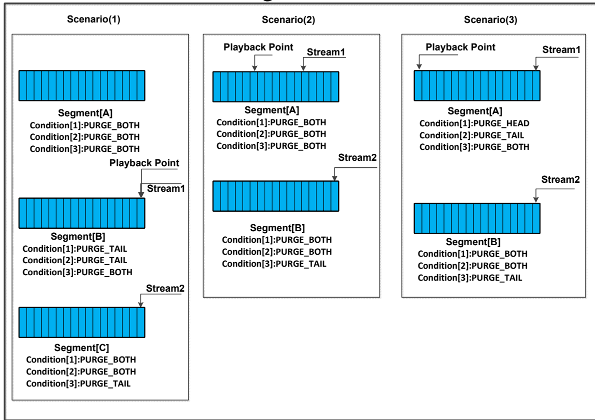


Fig. 4: Illustration of Adaptive Purge Algorithm

Figure 4 illustrates how the values of three conditions are computed in three scenarios. In the first scenario, Segment A does not include the client's playback point or the playback point of any stream that is being listened to by the client. Hence, the three conditions are set to BOTH. Segment B has the customer playback point and Stream1 playback point at the head of the segment. Hence, both Condition 1 and Condition 2 are set to TAIL. Condition 3 is set to BOTH because Stream2 playback point is not within Segment B. The algorithm loops through all segments starting from oldest. If all the aforementioned conditions allow for purging of the segment tail, that tail is purged; otherwise, the algorithm purges the furthest data from the customer's playback point. If the purged data is enough to keep the cache size smaller than the maximum size, the algorithm stops; otherwise, it moves to the next segment.

We introduce and analyze a new option, called *Continue-to-Listen* (C2L), specifying whether the pausing user continues to listen to all multicast streams and to purge data according to the purging algorithm or stops listening to all streams when the cache gets full.

5. PERFORMANCE EVALUATION METHODOLOGY

We developed a simulator for an interactive NVOD system, including both the clients with caches and the server. The system supports various stream merging and scheduling techniques. The simulator stops after a steady state analysis with 95% confidence interval is reached. We experimented with a wide range of system parameters, but only the main results are shown for space limitation.

Table 1 summarizes the default parameters used. We characterize the waiting and blocking tolerance of customers as Poisson with a mean of 30 seconds. We examine the server at different loads by varying server capacities from 196 Gbps to 320 Gbps. Only 10% of the server capacity is reserved for I-Streams [11]. We use the FCFS scheduling policy for waiting and blocking customers to avoid any issues of unfairness. The playback deviation point tolerance is kept at 10 seconds. The default client side cache size dedicated to the video streaming application is kept at 50 MB. We use a small default value to demonstrate the behavior in highly constrained scenarios (e.g., highly limited available RAM in some mobile devices).

We utilize the results of [8], which characterizes the workload of a media streaming server using actual access logs. This workload determines the distributions of all requests, including interactive requests, for various videos. It shows a strong relationship between the video file duration and interactive operations and also shows a correlation between consecutive interactive operations. The results of that study are consistent with another study [9], but the latter is less detailed and did not allow for the generation of a full synthetic workload. The average arrival rate (λ) is 30 requests per minutes. We study 100 videos of varying lengths and request rates. The workload indicates that 91% of the requested files have average bit-rates of $300 - 350$ Kbps. The average bitrates of the remaining files are in the range of $200 - 300$ Kbps. To be more reflective of recent video bitrates, we use 2.1 Mbps for the first group and 2.45 Mbps for the latter.

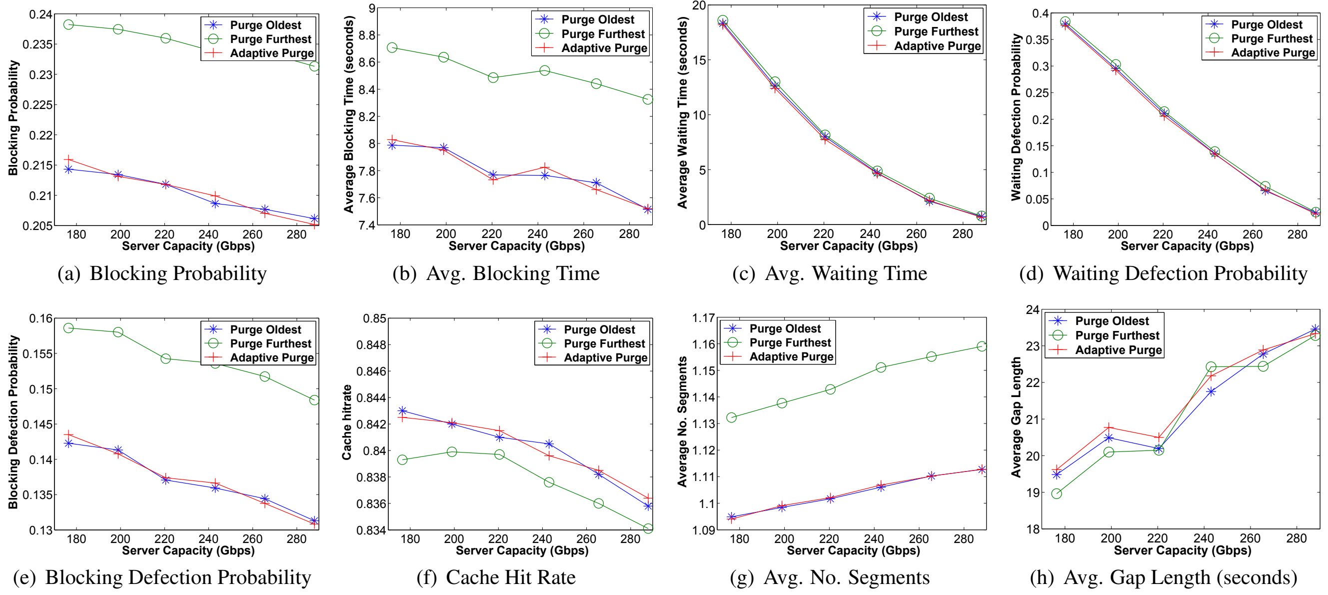
Table 1: Default Parameters Values

Parameter	Default Value(s)
Arrival Rate	30 Requests / minute
Number of Videos	100
Waiting Tolerance Model	Poisson with mean = 30 sec.
Blocking Tolerance Model	Poisson with mean = 30 sec.
Server Capacity	196 - 320 Gbps
Video Bit-rate	$2.1 - 2.45$ Mbps
I-Streams	10% of server capacity
Cache Size	50 MByte
Playback Point Deviation Tolerance	10 seconds

We consider the following **performance metrics**: *average waiting time*, *waiting defection probability*, *average blocking time*, *blocking defection probability*, *blocking probability*, and *aggregate delay*. The waiting defection probability is defined as the probability that the customer leaves the server because the waiting time exceeded the customer's tolerance. Likewise, the blocking defection probability is defined as the probability that the customer leaves the server without finishing watching the video because the customer stayed in the blocking queue longer than its tolerance. The average waiting time is the average time the customer stays in the waiting queue. Similarly, the average blocking time is the average time the interactive request stays in the blocking queue. The blocking probability is defined as the likelihood of an interactive request to block. The aggregate delay is the average delay experienced by a client due to both waiting and blocking. To reflect the increased customer frustration, the weight is increased for every subsequent blocking during the same session [11].

6. RESULT PRESENTATION AND ANALYSIS

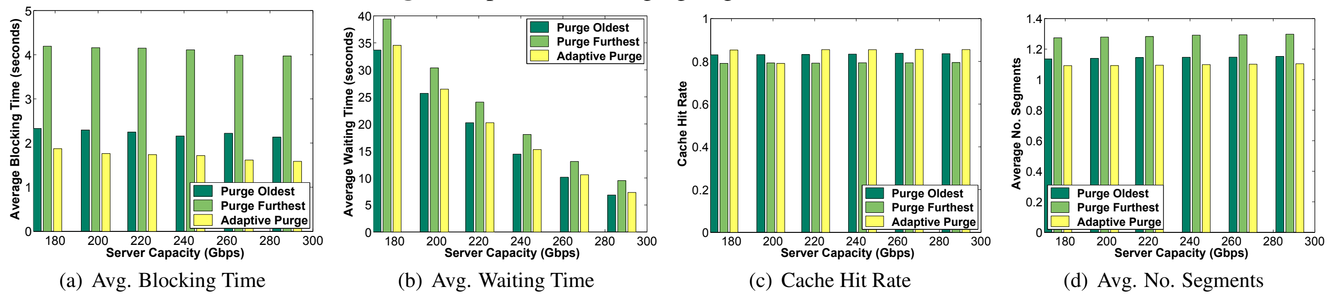
Figure 5 illustrates the effect of the purging algorithm on the system performance when C2L is not considered. The Adaptive Purge and Purge Oldest algorithms significantly outperform Purge Furthest in terms of blocking metrics. Since the Adaptive Purge defaults to the purging the oldest data first, the performance of the Adaptive Purge is close to the Purge Oldest. The purging algorithm affects the cache of already



(a) Blocking Probability (b) Avg. Blocking Time (c) Avg. Waiting Time (d) Waiting Defection Probability

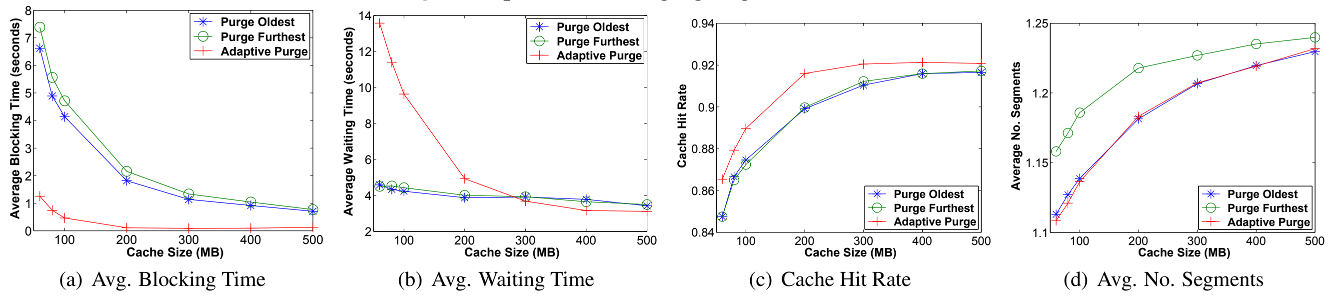
(e) Blocking Defection Probability (f) Cache Hit Rate (g) Avg. No. Segments (h) Avg. Gap Length (seconds)

Fig. 5: Impact of the Purging Algorithm without C2L



(a) Avg. Blocking Time (b) Avg. Waiting Time (c) Cache Hit Rate (d) Avg. No. Segments

Fig. 6: Impact of the Purging Algorithm with C2L



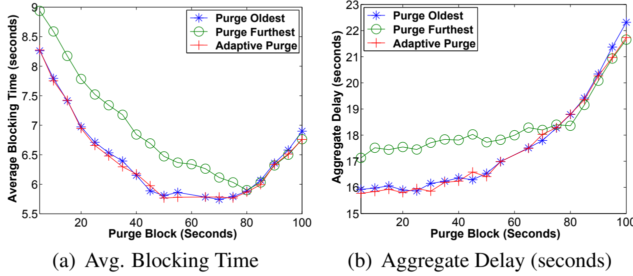
(a) Avg. Blocking Time (b) Avg. Waiting Time (c) Cache Hit Rate (d) Avg. No. Segments

Fig. 7: Impact of Cache Size [Adaptive Purge with C2L. Purge Oldest and Purge Furthest without C2L]

admitted customers. Hence, there is no significant difference between the purging algorithms in terms of waiting metrics.

The cache hit rate decreases with server capacity. Since the system tries first to merge the interactive request with another multicast (B-Stream or P-Stream) for customers not listening to any stream and there is a higher likelihood the system would find a multicast stream at higher server capacity, the merge percentage for interactive requests increases with server capacity. Hence, the cache hit rate decreases. The cache fragmentation also increases with the server capacity as shown by the average number of segments and the average gap length. As the server capacity increases, a higher percentage of interactive requests are serviced by merging with other streams. The playback points of these streams are more likely to be outside the client cache. Since the client caches data from all streams, the probability of caching more segments increases. The cache hit rate is higher for Adaptive Purge and Purge Oldest than Purge Furthest at all server capacities because Purge Furthest is more likely to purge future data that will be needed in the future. Purge Oldest and Adaptive Purge, however, avoid purging future data. Although not shown, the Aggregate Delay improves with server capacity because it incorporates both waiting and blocking times and both improve with increased server capacity.

Figure 6 illustrates the performance of the purging algorithms when C2L is employed. Adaptive Purge outperforms Purge Oldest and Purge Furthest in terms of the average blocking time. Both Adaptive Purge and Purge Oldest outper-



(a) Avg. Blocking Time (b) Aggregate Delay (seconds)

Fig. 8: Impact of Purge Block Size

form Purge Furthest in terms of average waiting time. Both the cache hit rate and the average number of segments do not change significantly with the server capacity for all purging algorithms.

Figure 7 illustrates the impact of the cache size on the waiting and blocking metrics. In the figure, Adaptive Purge uses C2L, whereas Purge Oldest and Purge Furthest do not. Both waiting and blocking metrics improve with the cache size for all purging algorithms. The cache hit rate improves with cache size until it reaches a certain limit beyond which no significant improvement can be achieved due to diminishing returns. Hence, the blocking metrics improve. The waiting metrics improve because the system can serve longer patches, which can fit in the larger cache. Hence, the relative percentage of P-Streams to B-Streams increases with cache size. Consequently, the system utilizes the smaller more efficient P-Streams than the more expensive B-Streams. We notice that Adaptive Purge with C2L significantly outperforms the other two algorithms in terms of blocking metrics because when the pausing customer continues to listen to all streams, there are more future data in the cache. The future data becomes very useful when the customer resumes playback, thereby improving the cache hit rate as shown in Figure 7(c). At a smaller cache size, Purge Oldest and Purge Furthest outperform Adaptive Purge in the waiting metrics. At a larger cache size, however, Adaptive Purge outperforms the other two algorithms.

Finally, Figure 8 illustrates the impact of the purge block size on the system performance. The blocking metrics keep improving with purge block size until the size reaches a certain value (about 50 seconds), afterwhich it starts to worsen. This behavior is explained by the fact that the average number of cache segments per customer keeps decreasing until it reaches the block size of 50 seconds, afterwhich it starts to increase. The aggregate delay does not change much until a certain purge block size (about 60 seconds), afterwhich it starts to increase with purge block size.

7. CONCLUSION

The main results can be summarized as follows. (1) Using our proposed cache management policy, up to 92% of all interactive requests are serviced from the client's own cache without requiring additional server resources. (2) The overwhelming majority of jump backward and resume interactive requests are serviced from the cache. The jump forward interactive requests cache hit rate is significantly lower than the other two interactive requests because some requested data is future data that is never cached. (3) Determining which data to purge when the cache becomes full has a significant impact on the cache hit rate and blocking metrics. Purging the oldest data improves cache hit rate and blocking metrics. To further enhance the performance, we experimented with an adaptive purging algorithm. Interestingly, examining the additional conditions does not provide considerable improvements over purging the oldest data in specific cases. (4) Choosing the cache purge block size has a significant impact on the blocking and waiting metrics. The optimal block size is system dependent. (5) Increasing the cache size improves both waiting and blocking metrics until the cache size reaches a certain limit, beyond which, no significant improvement can be achieved due to diminishing returns. (6) Another important decision is whether pausing customers continue to listen (C2L) to streams when the cache becomes full. Enabling C2L is highly desirable, especially for sufficiently large caches.

8. REFERENCES

[1] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan, "Analyzing the potential benefits of cdn augmentation strategies for internet video workloads," in *Proc. of on Internet Measurement Conf.*, New York, NY, USA, 2013, IMC '13, pp. 43–56, ACM.

[2] Zhijie Shen, Jun Luo, Roger Zimmermann, and Athanasios Vasilakos, "Peer-to-peer media streaming: Insights and new developments," in *Proc. of the IEEE 99(12)*, 2011, pp. 2089–2109.

[3] George Pallis and Athena Vakali, "Insight and perspectives for content delivery networks," *Communications of the ACM*, vol. 49, pp. 101–106, January 2006.

[4] Vaneet Aggarwal, Robert Caldebank, Vijay Gopalakrishnan, Rittwik Jana, K. Ramakrishnan, and Fang Yu, "The effectiveness of intelligent scheduling for multicast video-on-demand," in *Proc. of ACM Multimedia*, 2009, pp. 421–430.

[5] Derek L. Eager, Mary K. Vernon, and John Zahorjan, "Bandwidth skimming: A technique for cost-effective Video-on-Demand," in *Proc. of Multimedia Computing and Networking Conf. (MMCN)*, Jan. 2000, pp. 206–215.

[6] Marcus Rocha, Marcelo Maia, Italo Cunha, Jussara Almeida, and Sergio Campos, "Scalable media streaming to interactive users," in *Proc. of ACM Multimedia*, Nov. 2005, pp. 966–975.

[7] Sung Soo Moon, Kyung Tae Kim, Seong Woo Lee, Hee Yong Youn, Ohyoung Song, and Ohyoung Song, "An efficient vod scheme combining fast broadcasting with patching.," in *Proc. of ISPA*, 2011, pp. 189–194.

[8] Cristiano Costa, Italo Cunha, Alex Borges, Claudiney Ramos, Marcus Rocha, Jussara Almeida, and Berthier Ribeiro-Neto, "Analyzing client interactivity in streaming media," in *Proc. of The World Wide Web Conf.*, May 2004, pp. 534–543.

[9] Joonho Choi, A. Reaz, and B. Mukherjee, "A survey of user behavior in vod service and bandwidth-saving multicast streaming schemes," *Communications Surveys Tutorials, IEEE*, vol. 14, no. 1, pp. 156–169, First 2012.

[10] Florin Dobrian, Asad Awan, Dilip Joseph, Aditya Ganjam, Jibin Zhan, Vyas Sekar, Ion Stoica, and Hui Zhang, "Understanding the impact of video quality on user engagement," *Commun. ACM*, vol. 56, no. 3, pp. 91–99, Mar. 2013.

[11] Kamal Nayfeh and Nabil Sarhan, "Design and analysis of scalable and interactive near video-on-demand systems," in *Proc. of Multimedia and Expo Conf. (ICME)*, July 2013, pp. 1–6.

[12] Wanjium Liao and Victor O. K. Li, "The split and merge (SAM) protocol for interactive video-on-demand systems," in *Proc. of the IEEE Computer and Communications Societies Conf.*, Apr. 1997, pp. 1349–1356.

[13] Emmanuel L. Abram Profeta and Kang G. Shin, "Providing unrestricted VCR functions in multicast video-on-demand servers," in *Proc. of IEEE Int'l on Multimedia Computing and Systems Conf. (ICMCS)*, July 1998, pp. 66–75.

[14] Shahid Akhtar, Andre Beck, and Ivica Rimac, "Hifi: A hierarchical filtering algorithm for caching of online video," in *Proc. of ACM on Multimedia Conf.*, New York, NY, USA, 2015, MM '15, pp. 421–430, ACM.

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

[16] Nabil J. Sarhan, Mohammad A. Alsmirat, and Musab Al-Hadrusi, "Waiting-time prediction in scalable on-demand video streaming," *ACM Transactions on Multimedia Computing, Communications, and Applications (ACM TOMCCAP)*, vol. 6, no. 2, pp. 1–24, March 2010.