A chunk-based caching algorithm for streaming video

Dohy Hong, Danny de Vleeschauwer, François Baccelli

A chunk-based caching algorithm for streaming video

Dohy Hong
N2Nsoft
Paris, France
dohy.hong@n2nsoft.com

Danny De Vleeschauwer
Bell Labs & SMACS
Alcatel-Lucent Bell & UGent
Antwerp & Gent, Belgium
danny.de_vleeschauwer@alcatel-lucent.com

Francois Baccelli
Computer Science Department
École Normale Supérieure
Paris, France
francois.baccelli@ens.fr

Abstract—**It is customary nowadays that large web objects are cached somewhere close to the user. This saves traffic upstream of the cache and offers the users a better responsiveness. Caching algorithms typically rank the objects in some way and cache the top-ranked objects. In this paper we study a scenario in which a requested video is (instantaneously) streamed to the user and in which the video library is highly dynamic: new videos are frequently introduced, get popular, get consumed and fade away. Caching streaming videos differs from caching traditional web objects as the former are consumed as their information trickles in, while the latter have to be downloaded (almost) completely before they can be consumed. We develop a caching algorithm specifically for streaming video taking into account the dynamicity of the library. First we make sure that its ranking algorithm can follow the dynamicity of the library (better than traditional algorithms can). Second we segment each video in chunks and propose a new algorithm to rank these chunks. We compare the performance of caching based on this new ranking algorithm with traditional caching algorithms and show that chunking is most beneficial.**

Index Terms—**caching, popularity, streaming.**

I. Introduction

Packet-based networks, in particular the Internet, are more and more used to distribute video. Channel-based networks (e.g., terrestrial, satellite, cable networks) over which traditionally video was delivered can rely on some form of broadcast, i.e., a number of channels are offered side by side over a shared medium and users can tune in to one of these channels. While this service is also offered over a packet-based network via multicast trees (a service, which is often referred to as "linear programming IPTV"), packet-based networks more easily support personalized on-demand video services: a video library is offered via a catalogue (hosted on a server) and a separate flow is set up for each requesting user.

In such an on-demand system the amount of video traffic on the distribution network is in principle proportional to the number of users, while in linear programming IPTV it was proportional to the number of channels. Since there are many more users than channels, this leads to a prohibitive increase of the capacity requirement. Fortunately, most videos in the library are of interest to more than one user and although the users do not consume a particular video at exactly the same time any more, that video is still consumed over a limited time span [1]. So, if it can be identified which are the most popular videos at any moment in time, these could be cached close to the user; transporting them only once to this cache (and as many times as needed to the users served by that cache), would save a considerable amount of transport capacity on the distribution network.

Based on a similar reasoning, caches are deployed over the Internet to offer users a faster response when downloading large web objects. Since the fair share (e.g., offered by the Transport Control Protocol (TCP)) on the path from the cache to the user is larger than the fair share end-to-end, the web object could be downloaded and consumed faster. Such a set of strategically deployed caches (referred to as a content distribution network) are well understood nowadays: all large web objects (for which a content owner pays) are cached and so these objects are served with less waiting time for the user. Note that the primary aim is the increased responsiveness, while the decrease in transport capacity is a secondary consequence.

Caching for streaming differs in a number of aspects from caching large web objects. First, as indicated above, the prime aim of caching for streaming is different: it envisions decreasing the required transport capacity on the distribution network as much as possible. Second, the dynamicity of the video library has a different character: videos are introduced, get popular and fade away in popularity. As a result a video object is consumed over a relatively short time span (while traditional web objects are requested more or less uniformly over prolonged periods). Moreover, video objects are usually much larger than traditional web objects. For these reasons it is very important to store the right content at the right time in the caches in video streaming. However, the biggest difference is that a video stream is consumed very soon after it has been requested. In contrast to traditional web objects (including downloaded video and to a lesser extent progressively downloaded video), which need to be downloaded completely before they are consumed, a video stream is consumed as the information trickles in. For that reason segmenting the video in chunks will turn out to be beneficial.

Part of the research leading to these results has received funding from the European Union's Seventh Framework Programme ([FP7/2007-2013]) under grant agreement n° 248775.

The rest of this paper is organized as follows. The next section describes some related work. In Section III the principles upon which our new caching algorithm is based are highlighted. Section IV explains the simulator we used and compares the performance of our new algorithm with traditional algorithms. Finally, in Section V we draw the main conclusions.

II. Related work

Caching algorithms have been used to decide what to store in web proxies [5][9], in CPU caches [8] and in surrogate servers in a video on-demand application [3][6]. In all of these cases the purpose of caching is to reduce the response time of a request for data. Deploying caches between the data server and the data client(s) alleviates the processor of the data server and reduces the load on a capacity-constrained link somewhere between the data server and the data client. In this paper we mainly concentrate on reducing the load on the network.

Caching algorithms (also often referred to as cache replacement algorithms) take decisions based on observations. Most algorithms use recency and/or frequency of requests for an object or a combination thereof to rank the objects. The ones that are ranked the highest (and which are implicitly assumed to be requested the most frequently over the next period) are cached. The ranking is updated regularly (e.g., at each request time or at constant intervals). Often this ranking (or estimated request rate) is used in a cost function that trades off the transport capacity saved with the storage cost incurred by caching the object [3].

Least-recently used (LRU) [7], which is solely based on recency, suffers from attaching too much importance to unpopular objects. Even an unpopular object jumps to the first rank upon a request for it and subsequently takes a long time to be evicted from the cache, where it occupies a place that could have been used by another object more fruitfully. Least-frequently used (LFU) [9], which is solely based on frequency, is sluggish as it attaches as much importance to ancient as to recent history. Often an aging factor is used to improve its performance. In [2][6] it is proved that by tuning the aging factor the algorithm can be made to behave like LRU or like LFU and that often there is an optimal aging factor in between these two extremes for which the hit ratio is maximal. More complex algorithms to determine the ranking (or equivalently the expected request rate) have been studied: based on a combination of LRU and LFU [8] or based on a neural network [4].

The main difference between the algorithm developed in this paper and the algorithms discussed above is that our algorithm takes the time structure of video objects to be cached into account. Indeed, the fact that streaming video is consumed as it trickles in opens up new possibilities for improving the caching performance: if a set of users are consuming a streaming video, each being at a different point in time, it is known which parts of the video will still be visited and which parts are superfluous. A method based on a similar premise is discussed in [10], but concentrates on the initial portions of the video, while our method encompasses the complete video.

III. Caching algorithm

Figure 1 shows the set up that we consider in this paper. A video library is offered to a set of users from an origin server. A cache, which serves a subset of the users, is deployed on the edge of the distribution network. The aim of the cache is to reduce the traffic on the distribution network.

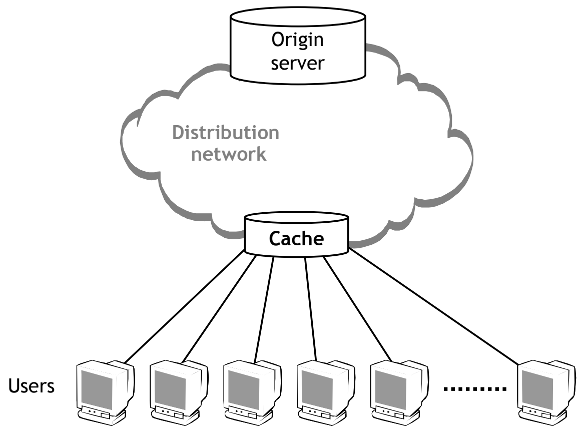


Figure 1. Cache network configuration

The video library considered in this paper is highly dynamic. New videos are introduced frequently. Just after the introduction time of a new video it is most popular, and the more time elapses since its introduction time, the less frequently it is asked. This popularity evolution is typical for a catch-up television service as studied in this paper (see [1]). Each video has its own popularity decay curve that follows either an exponential or a power law. Similar consumption patterns have been observed for other video services, e.g., for YouTube in [4]. In this paper all videos are assumed to be encoded at the same bit rate and to have the same length (of 120 minutes).

The cache is not large enough to store all videos. Therefore a selection needs to be made of which videos to cache at any moment in time. For that purpose the caching algorithm ranks the videos and caches the ones that are ranked in the first L positions.

The caching algorithm that we develop in this paper is based on two principles: 1) scoring videos based on requests for them and 2) segmenting the videos in chunks, which we combine with the knowledge that chunk m+1 of a given video will be requested with a high probability in the near future if chunk m of that video is currently streamed to some user. To be able to separate the benefits both principles bring, we first consider a simplified version of our caching algorithm without chunking, which makes decisions by ranking videos in their entirety. For the full version of our caching algorithm all videos are segmented in chunks of equal duration. In this case the decision which chunk to cache is based on the ranking of the chunks. Both versions of the algorithms are described in more detail in the next paragraphs.

A. *Scoring*

The simplified version of our algorithm is based on keeping a score S_k for each video k (k=1, 2, …, K). Each time video k is requested, it gets a bonus, i.e., its score is increased by an amount A, while the score of all other videos is decreased by 1. When a new video is requested for the first time its score is initialized to a value B. Moreover, the score is kept within the interval [-C,C], by truncating it to these bounds if adding a value A or decreasing it by 1 would move it outside this interval. The simplified version of our algorithm re-ranks at each request time the videos based on these scores S_k and the first L ranked videos are cached. Hence, at these request instants one of the following three events can occur

1. The requested video resided already in the cache (an event that is referred to as a "hit"). In this case the caching algorithm updates the ranking, but no videos are evicted from the cache. Moreover, the video is served from the cache. Note that no flow is needed on the distribution network.

2. The requested video did not reside in the cache and the caching algorithm decides to cache the video, because this request makes that this video gets upgraded to a rank in the first L positions. The server copies the video into the cache, i.e., the information trickles in at the video rate, thus consuming one flow over the distribution network. This also makes that the video that resided at rank L (i.e., the last rank) just before the current request, is gradually being evicted from the cache (in fact it is being overwritten by the new video). As the new video information trickles in (and is copied into the cache, overwriting the video at position L), it is forwarded towards the requesting user.

3. The requested video did not reside in the cache and the caching algorithm decides that the requested video does not need to be cached this time. The ranking is updated, and (since the algorithm did not cache the video) the requested video has a rank larger than L. The video is served from the origin server, thus consuming one flow over the distribution network.

In this paper the value S_k is maintained for every video ever seen, but in practice videos that would have a too low value S_k (indicating that they would not have been visited for a long time) would be deleted from the list. If there is a tie between two videos (i.e., if they have equal S_k values) the video with the lowest k value takes precedence.

B. *Number of guaranteed hits*

The full version of our caching algorithm also maintains the values S_k in just the same way as described above. Moreover each video is segmented in M chunks and each chunk inherits the score S_k from the video it belongs to. For each chunk m of video k a value $N_{k,m}$ is maintained that accumulates the number of guaranteed hits this chunk will have, knowing which videos are currently watched by the users (and assuming that no user aborts watching a video). Figure 2 illustrates that the value $N_{k,m}$ indicates how many times that particular chunk m of video k, will be consumed in the near future (given the current user behavior). This counter $N_{k.m}$ is maintained as follows:

1. The values $N_{k,m}$ are increased by 1 for all values of the index m, each time video object k is requested by a user.

2. The value of $N_{k,m}$ is decreased by 1 after a user watching video object k has consumed chunk m.

3. Remark that if a user aborts viewing video object k before the end of the video (or uses other trick-play commands like "rewind" or "fast rewind"), the values $N_{k.m}$ need to be updated accordingly: e.g., in case a user aborts watching a video for each m larger than or equal to the current chunk the user was watching before he or she aborted, the $N_{k,m}$ value needs to be decreased by 1. However, note that "abort", "rewind" or "fast rewind" events do not occur in our simulations.

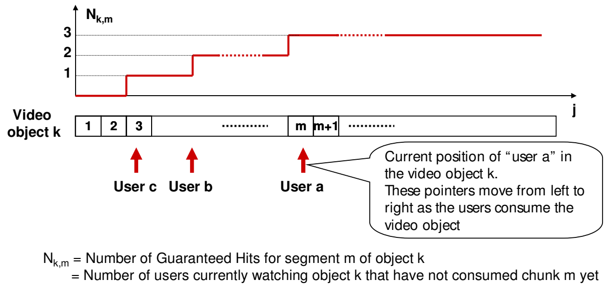

$N_{k,m}$
3
2
1
j
Video object k
1
2
3
m
m+1
User c
User b
User a
Current position of "user a" in the video object k.
These pointers move from left to right as the users consume the video object
$N_{k,m}$ = Number of Guaranteed Hits for segment m of object k
= Number of users currently watching object k that have not consumed chunk m yet

Figure 2. Maintaining $N_{k,m}$ for video k

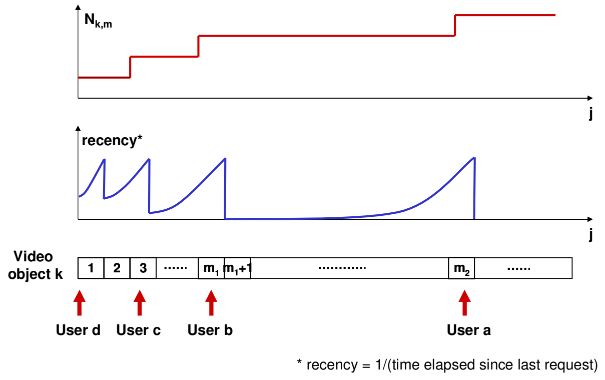


Figure 3. Why our algorithm performs better than LRU

For the rest this chunk-based caching algorithm (i.e., the full version of our caching algorithm) operates in a similar way as the simplified version of our algorithm: at each request time for a chunk, one of the three types of events described in the previous paragraph occurs (i.e., a cache hit, a cache miss combined with a cache update or a cache miss without a cache update). The only difference is that the ranking is now based on comparing the values $N_{k,m}$. (the higher the value $N_{k,m}$ the higher the rank of the chunk (k,m)) while the values S_k are merely used as tie-breakers. If after ranking chunks based on both $N_{k,m}$ and S_k there is still a tie, chunks are ranked based on their chunk number.

Figure 3 illustrates that traditional caching algorithms operating on chunks do not take into account the time structure of the video (i.e., the fact that chunk (m+1) necessarily follows chunk m) and therefore often make wrong caching decisions. This figure shows a snapshot of video k currently being consumed by four users. The chunk (m_1+1) has a low recency value, as "user a" consumed this chunk a long time ago and "user b" did not reach it yet, so that a traditional caching algorithm (e.g., LRU) operating on chunks would currently not cache it. It can be seen that this chunk is about to be visited by "user b" and two additional users in the near future, such that it merits to be cached, which a caching algorithm based on $N_{k,m}$ would do. This qualitative reasoning will be further quantified in Section IV.

As stated before the most efficient caching algorithm is the one that, at any moment in time, caches those chunks that are most likely to be consumed in the near future. It is clear that in order to determine this likelihood a suitable combination of consumption of videos in the past and currently ongoing videos should be made. Our algorithm is an example of such a combination, where the knowledge related to the currently consumed videos gets priority over the historical video consumption. Other combinations are possible too.

IV. SIMULATIONS

This section briefly describes the simulator and then uses this simulator to assess the performance of the caching algorithm described in Section III.

A. Simulator

We have developed a time-driven (slotted) simulator to assess the performance of this caching algorithm. The time slot is chosen small enough so that it has no impact.

We study a catch-up television service as an example in this paper. We took inspiration for the process with which new videos are introduced in the library and how requests for existing videos are generated in the observation of a real-life system analyzed in [1]. In the simulator videos are introduced into the video library at time instants generated by a Poisson process with arrival rate 10/day. So, on average ten new videos are introduced in the library per day.

The aging process of the videos is as follows. After the introduction time of video k a user can request video k to be streamed to him or her. These requests for video k are generated according to a non-stationary Poisson process with arrival rate (which is referred to as the "demand curve for video k" in this paper) that is specific to video k. This demand curve for video k is randomly drawn at the introduction time of video k in the simulator. Overall the demand curve exposes an exponential decay, but there may be some moments where the content popularity gets a boost. In particular we model a demand curve as $\rho(t)\cdot\exp(-t/\tau)$ ($t>0$ and 0 otherwise), with $\rho(t)$ a piece-wise constant function. The decay time τ is uniformly drawn between 1 day and 3 days and the initial request rate $\rho(0)$ is uniformly drawn in the interval [43/day,129/day]. Ninety percent of the videos have a constant $\rho(t)$, and hence, have a pure exponentially decaying request rate. The remaining ten percent of the videos are (randomly selected to be) upgraded to become "popular" videos. For those the (randomly selected) decay time τ is divided by 2 and the initial request rate $\rho(0)$ is multiplied by 10. Moreover, for these "popular" movies, the demand curve is boosted after a $t=(j\cdot 7)$ days to a value $10\cdot\rho(0)/(5\cdot j)$ (with j=1, 2, …). In particular, for these popular videos the function $\rho(t)$ is a piece-wise constant function specified as: $\rho(t)=10\cdot\rho(0)$ for $0<t<7$ and $\rho(t)=10\cdot\rho(0)/(5\cdot j)\cdot\exp(j\cdot 7/(\tau/2))$ for $j\cdot 7<t<(j+1)\cdot 7$ (with j=1, 2, …). This behavior we introduced because we noticed that in the catch-up television service studied in [1] the previous episode of the series gets a boost when the next one is aired (see also [6]). Figure 4 shows two typical demand curves, one for a "popular" video and one for an un"popular" video.

Once the demand curve associated with video k is determined, requests for video k are generated by a non-stationary Poisson process with an arrival rate equal to the demand curve.

This is just an example of a (realistic) way to generate requests. The simulator can work with requests generated in any other way, in particular with a list of requests observed in an actually deployed video-on-demand system. This list of requests (generated internally or read from a list) drives the simulator.

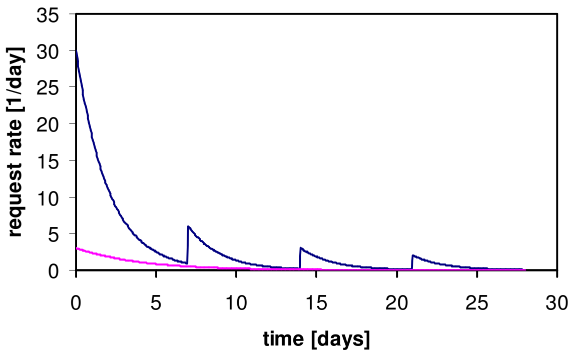


Figure 4. Two typical demand curves

In each slot the simulator takes the following actions. First the requests for videos are generated as described above. If there are requests for video k in this slot the values of S_k and $N_{k,m}$ are updated as described in Section III. All new requests are added to the list of ongoing requests. Then, for all ongoing requests it is checked whether chunk (k,m) ends, in which case its corresponding value $N_{k,m}$ is updated as described in Section III. Subsequently it is tested for each ongoing request whether or not the requested information (i.e., that part of the chunk that needs to be served in the current slot) resides in the cache or needs to be retrieved from the origin server, and in the latter case whether or not it needs to be cached (based on the decision made by the caching algorithm). If the requested information resides in the cache the counter H that maintains the number of hits is increased. Additionally a counter N is maintained that keeps the total number of times information was requested. The average hit ratio which is reported at the end of the simulation is defined as H/N and indicates the percentage with which the average traffic volume on the distribution network can be decreased by deploying a cache.

Since new videos are introduced at a rate of 10 videos per day and each video is consumed over the time span of a few days, we intuitively expect that a cache size of a few tens of videos will suffice to obtain a high hit ratio [6]. Therefore we study caches that range in size from 1 to 100 videos.

Finally, in all simulations, we set the parameters A, B and C as follows: A=2·L, C=60·L and B is set equal to A at the beginning of the simulation and later follows the average of S_k in particular B=max{A,average{S_k}} (where the average is only calculated based on videos that reside in the cache). This is just one, albeit a robust way, to choose these parameters. In all tests we have performed we have observed that the choices of the parameters A, B and C have a second order impact on the cache hit ratio. In particular, the major conclusion that we draw in this paper, i.e., that a caching algorithm can benefit from the knowledge that chunks in a video are consumed one after the other, is insensitive to the choice of the parameters A, B and C. We leave the fine tuning of those parameters for future work.

B. Performance assessment

In this section we assess the performance of the proposed caching algorithm.

First we compare the simplified version of our caching algorithm, which does not segment the video in chunks, with traditional algorithms, which also operate on entire videos. Figure 5 illustrates that our scoring algorithm (labeled "CC-120") outperforms LRU (labeled "LRU-120") considerably for small cache sizes. Figure 6 shows that for small cache sizes the performance of our algorithm increases by more than 100% with respect to LRU. The reason is that our algorithm unlike LRU does not promote a video to the first rank each time it is requested. For LFU (labeled "LFU-120") the window size was optimally tuned: 12h yielded the best performance and the optimum was broad. It can be seen in Figure 6 that our algorithm gains about 10% with respect to LFU. Finally, Figure 5 also shows the optimal hit ratio (labeled "HR-120"). To obtain this optimal hit ratio the videos are ranked in each slot based on the (theoretical) request rate with which requests were generated and the L top ranked videos are cached. In fact this is the same procedure as in our algorithm except that in the optimal algorithm the ranking is not based on S_k but on the theoretical request rates. These theoretical request rates are in principle not known to the caching algorithm, but if they were, this would be the best performance that a caching algorithm working on entire videos can reach. It can be seen that the simplified version of our caching algorithm, which is solely based on scoring videos, almost reaches the optimal performance in case entire videos are cached.

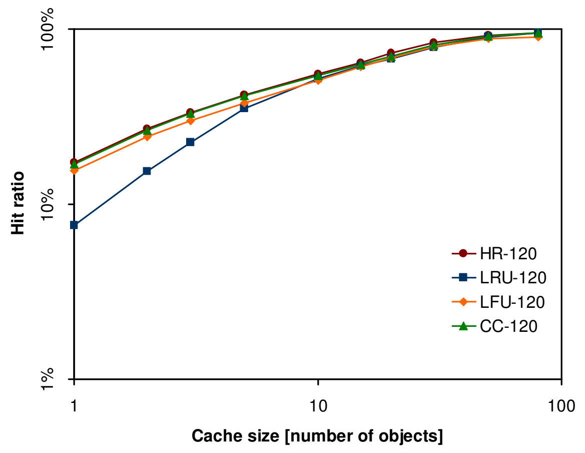


Figure 5. Comparison of our simplified algorithm (CC) with LRU, LFU and the optimal algorithm operating on entire videos

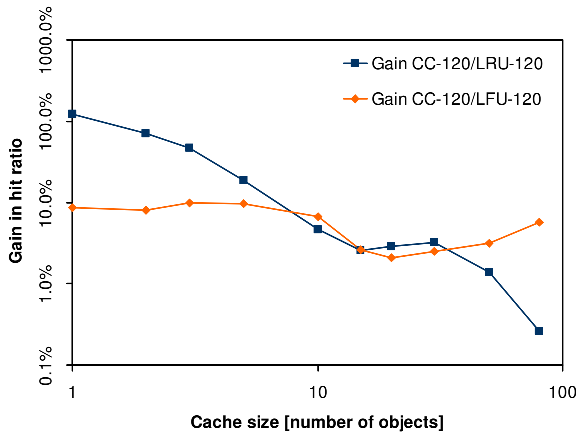


Figure 6. Performance gain of our simplified algorithm (CC) with respect to LRU and LFU operating on entire videos

Second, we consider the full version of our caching algorithm. We segment the video in chunks of 1 minute each and compare our full version with LRU and LFU (labeled "LFU-1"), which also operate on chunks. Figure 7 and 8 illustrate that our algorithm (labeled "CC-1") outperforms LRU (labeled "LRU-1") and LFU (labeled "LFU-1") where the latter again used an optimal window of 12h, even more than in the case when all algorithms work on entire videos, especially for small cache sizes. Apparently taking into account the time structure of the video, as our algorithm does presents more valuable information on the chunks that will be retrieved next than the knowledge related to how long ago or how many times these chunks were consumed in the past. Part of this may be due to the fact that in our model a video is always completely consumed whenever it is started. However, our algorithm can be extended for cases in which there is a non-negligible probability that a user abandons watching a video. Also in this case the algorithm will rank the chunks to be cached based on combining past behavior (similar to S_k) and ongoing behavior (i.e., the fact that for currently consumed videos there is a high likelihood that the next parts will be consumed too, similar to $N_{k,m}$). If the probability of abandoning the video is small enough, the latter will have to be weighted more than the former. Also in this case of chunk-based algorithms we consider the performance of a caching algorithm where the S_k values are not estimated but are taken equal to the true request rates (labeled "HR-1"). In particular, such an algorithm operates almost exactly the same as the full version of our caching algorithm: it ranks the chunks based on $N_{k,m}$ as well, but instead of using S_k as tie-breaker, it uses the theoretical request rates for this purpose. Remember that this strategy was optimal in case entire videos were cached, but this is no longer the case for chunk-based caching. Nevertheless it can be seen that our algorithm almost reached the performance of this ideal algorithm which shows that we do not lose a lot of performance by estimating rather than knowing S_k.

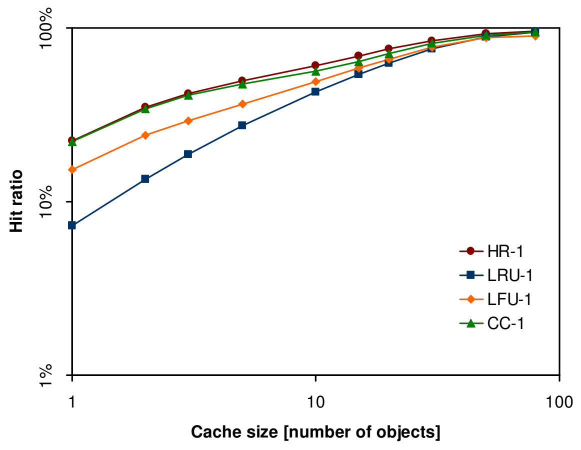


Figure 7. Comparison of our full algorithm (CC) with LRU, LFU and the ideal algorithm operating on chunks of 1 minute

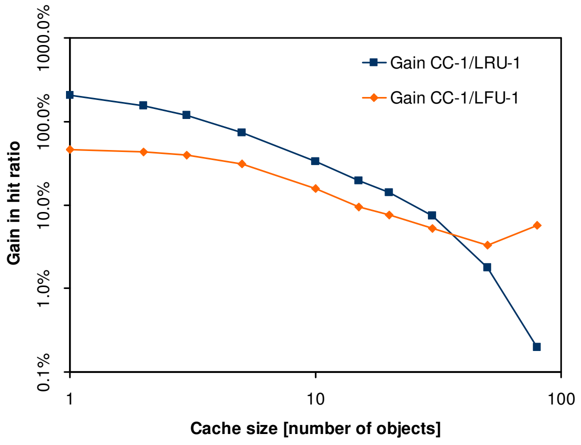


Figure 8. Comparison of our full algorithm (CC) with LRU and LFU operating on chunks of 1 minute

Figure 9 illustrates the gain of chunking. It compares the hit ratio of Figure 5 with the one of Figure 7. For the three considered algorithms (LRU, LFU and CC) there is a performance gain. For small cache sizes the gain is largest for the full version of our algorithm. For large cache sizes chunking does not improve the performance of our algorithm a lot, but notice that the hit ratio of the simplified version of our algorithm already approaches 100% for such cache sizes as shown in Figure 5.

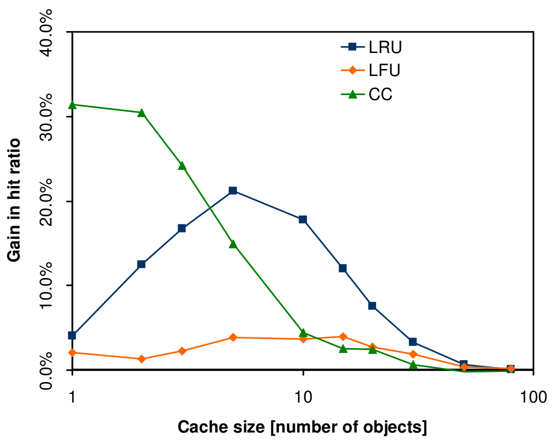
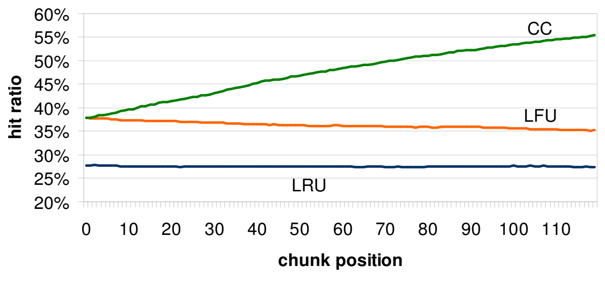


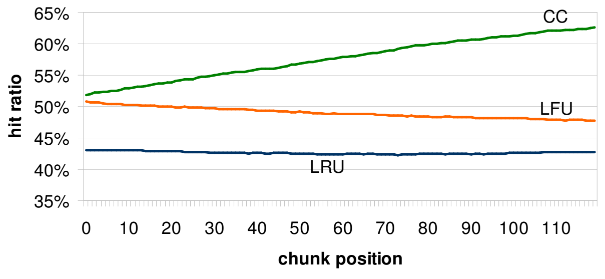
Figure 9. Performance gain of using chunks of 1 minute rather than caching the video as a whole.

To further explain why the full version of our caching algorithm has a better performance than traditional ones and to verify the arguments made when discussing Figure 3, we consider two additional statistics. The first one assesses the hit ratio per chunk number. The second one pertains to the nature of the chunks that are evicted from the cache.

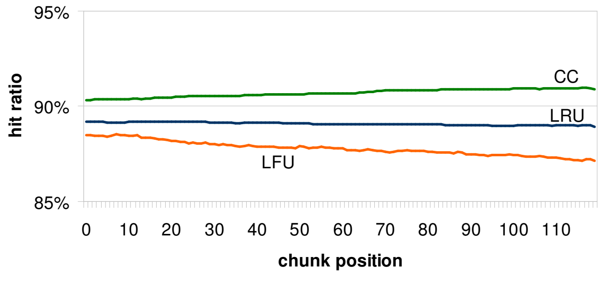
Figure 10 shows the hit ratio associated with chunk number m (m=0, 1, …, 119). Remember that we segmented each video in M=120 chunks. The hit ratio associated with chunk number m, is the fraction of times a chunk with chunk number m could be served from the cache. Remark that, since users do not abandon watching a video, chunks with chunk number m_1 are requested an equal number of times as chunks with chunk number m_2 as all chunks of a particular video are requested an equal amount of times. Hence, the overall hit ratio (shown in Figure 7) is the average over all possible values m of the hit ratio associated with chunk number m. It can be seen that, for small cache sizes (L=5), the full version of our algorithm has the tendency to cache chunks with a higher chunk number, i.e., it has the tendency to cache chunks towards the end of the video. This is a direct consequence of the fact that our algorithm anticipates better the immediate future. For larger cache sizes (L=10), the chunk distribution flattens a bit as a function of the chunk number and the difference between the full version of our caching algorithm and LRU and LFU is less pronounced. In the case of a large cache (L=50), all chunks, except the very unpopular ones, are in the cache and since each of them is visited once per request for a video, this results in a more or less flat distribution. Remark that for all cache sizes, our algorithm ("CC") achieves a better hit ratio associated with chunk number m than LRU and LFU for all values of the chunk number m. This illustrates why the full version of our algorithm outperforms traditional caching algorithms.



(L=5)



(L=10)



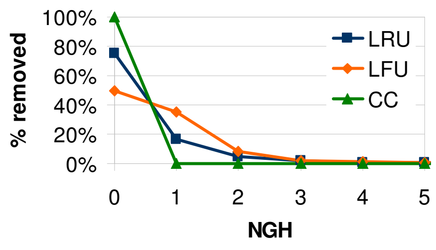
(L=50)

Figure 10. Hit ratio as a function of the chunk number for our full algorithm (CC), LRU and LFU.

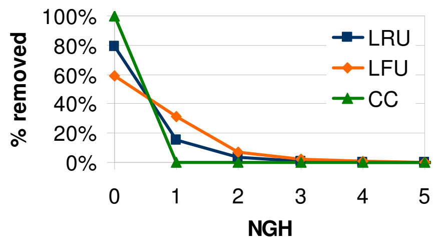
Figure 11 illustrates the nature of the chunks that are evicted from the cache. We make a distinction between the chunks based on their $N_{k,m}$ value. This figure shows that the full version of our algorithm (by construction) very rarely removes chunks for which the $N_{k,m}$ value is non-zero, while LRU and LFU do so. This corroborates the argument made when discussing Figure 3.

V. Conclusions

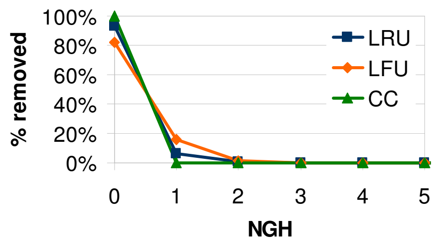
In this paper we proposed and evaluated a caching algorithm specifically targeted at streaming video. Each video is segmented in chunks and the algorithm makes caching decisions for these chunks. Traditional caching algorithms rank the chunks based on how often they were visited in the past and cache the highest ranked chunks (until the cache is full). Our algorithm also ranks the chunks and caches the top-ranked chunks, but it ranks them differently. The ranking is primarily based on the number of times the chunks will be visited in the near future. This value is determined in an efficient way by observing which chunks of which videos are currently being watched combined with the knowledge that when chunk m of video k is currently being consumed all chunks with a higher chunk number than m of that video k will be visited in the near future (under the assumption that the users do not abort watching a video). As tie-breaker for chunks that have the same value, our algorithm uses the historical consumption pattern (which is similarly, but better, measured than in traditional algorithm). We have proved via simulations that chunking combined with this new strategy of ranking the chunks, yields a considerable performance increase, especially for small cache sizes.



(L=5)



(L=10)



(L=50)

Figure 11. Percentage of removed chunks as a function of their number of guaranteed hits (NGH), i.e., $N_{k,m}$ value, for our full algorithm (CC), LRU and LFU.

References

[1] Z. Avramova, D. De Vleeschauwer, S. Wittevrongel, H. Bruneel, “Analysis and modeling of video popularity evolution in various online video content systems: power-law versus exponential decay”, Proceedings of the First International Conference on Evolving Internet (INTERNET09), pp. 95-100, Cannes, (France), August 23-29, 2009.

[2] Z. Avramova, D. De Vleeschauwer, S. Wittevrongel, H. Bruneel, “Performance Analysis of a Caching Algorithm for a Catch Up Television Service”, Multimedia System Journal, pp. 1-14, 2010.

[3] S. Borst, V. Gupta, A. Walid, “Self-organizing algorithms for cache cooperation in content distribution networks”, Bell Labs Technical Journal 14(3), 113–126 (2009).

[4] M. Cha, H. Kwak, H., P. Rodriguez, Y. Ahn, S. Moon, “I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system”, Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC07), San Diego (CA), October 24-26, 2007).

[5] J. Cobb, H. ElAarag, “Web proxy cache replacement scheme based on back-propagation neural network”, Journal of Systems and Software, Vol. 81, pp. 1539–1558, 2008.

[6] D. De Vleeschauwer, K. Laevens, “Performance of caching algorithms for IPTV on-demand services”, IEEE Transactions on Broadcasting, Vol. 55, No. 2, pp. 491-501, June 2009.

[7] G. Hasslinger, O. Hohlfeld, “Efficiency of caches for content distribution in the Internet”, Proceedings of the 22nd International Teletraffic Congress (ITC 22), Amsterdam (the Netherlands), September 7-9, 2010.

[8] N. Megiddo, D.S. Modha’ “Outperforming LRU with an adaptive replacement cache algorithm”, Computer, Vol. 37, No. 4, pp. 58–65, April 2004.

[9] L. Shi, Z. Gu, L. Wei, Y Shi, "An Applicative Study of Zipf’s Law on Web Cache", International Journal of Information Technology, Vol. 12, No. 4, pp. 49-58, 2006.

[10] K.-L. Wu, P.S. Yu, J.L. Wolf, “Segment-based proxy caching of multimedia streams”, Proceedings of the 10th international conference on World Wide Web, pp. 36–44, Hong Kong (Hong Kong), 2001.