AViC: A Cache for Adaptive Bitrate Video

Zahaib Akhtar
University of Southern California

Yaguang Li
University of Southern California

Ramesh Govindan
University of Southern California

Emir Halepovic
AT&T Labs – Research

Shuai Hao
AT&T Labs – Research

Yan Liu
University of Southern California

Subhabrata Sen
AT&T Labs – Research

ABSTRACT

Video dominates Internet traffic today. Users retrieve on-demand video from Content Delivery Networks (CDNs) which cache video chunks at front-ends. In this paper, we describe AViC, a caching algorithm that leverages properties of video delivery, such as request predictability and the presence of highly unpopular chunks. AViC's eviction policy exploits request predictability to estimate a chunk's future request time and evict the chunk with the furthest future request time. Its admission control policy uses a classifier to predict *singletons* — chunks evicted before a second reference. Using real world CDN traces from a commercial video service, we show that AViC outperforms a range of algorithm including LRU, GDSF, AdaptSize and LHD. In particular LRU requires up to $3.5\times$ the cache size to match AViC's performance. Further, AViC has low time complexity and has memory complexity comparable to GDSF.

CCS CONCEPTS

• **Information systems** → **Multimedia streaming**; • **Theory of computation** → **Caching and paging algorithms**.

KEYWORDS

HTTP Adaptive Bitrate Video, Caching, Content Delivery Networks

ACM Reference Format:
Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. 2019. AViC: A Cache for Adaptive Bitrate Video. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359989.3365423

1 INTRODUCTION

Video forms the majority of the Internet traffic today and is likely to continue its rapid growth in the near future [8, 10]; by one estimate [10], 82% of all IP traffic in 2022 will be video.

Video delivery background. Regardless of the type of network or device used, most video delivery in the Internet employs *adaptive streaming*. This technique divides a video into *chunks* each of which has a fixed playback duration. Each chunk is then encoded at different qualities (or *bitrates*) that trade-off reduced perceptual quality for lower network bandwidth. Thus, the term chunk denotes a segment of video of fixed playback duration encoded at a specific bitrate. Video content providers (also called *publishers* [11]) select parameters for the chunk duration and the bitrates to maximize user experience over a variety of devices and network conditions.

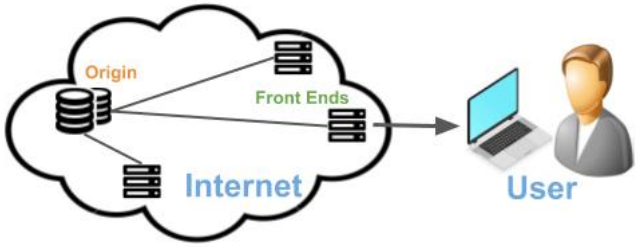


Figure 1: Serving video over the Internet

The video player software on a client device requests successive chunks from a server, stores retrieved chunks in a playback buffer, and displays chunks on the device. The player runs an adaptive bitrate (ABR) algorithm [12, 28, 35, 43] to determine the bitrate at which to download a chunk: this decision, usually based on measurements of network state (*e.g.,* available bandwidth), attempts to deliver the highest quality video to users while minimizing stalls.

The role of CDNs. Video content publishers use CDNs to deliver video-on-demand[1] to end-users [11] (Figure 1). CDNs consist of a small number of *origin* servers that store video content and a large number of *front-end* servers positioned topologically closer to users. When a client device requests a chunk of video, the CDN redirects it to the nearest front-end server, which, in turn, contacts the origin server for the requested bitrate of the chunk. The front-end server then *caches* the chunk to serve subsequent requests. This architecture enables the CDN to situate content closer to the end user and reduces response latency. Equally important, front-end servers can serve a significant fraction of requests from the cache, thereby shielding origin servers and reducing the CDN's wide-area bandwidth usage.

Caching algorithms. Given a cache of a fixed size, caching algorithms strive to retain those *objects* requested often. An object is the unit of cached information; examples include memory pages, web pages or video chunks. Caching algorithms achieve this using a *caching policy* based on recency (when the object was last accessed),

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365423

[1] In this paper, we focus on video-on-demand; we leave caching policies for live video to future work.

frequency (the rate of accesses to the object), object size, or a combination of these. Earlier caching policies focused on *eviction* — which object to remove from the cache when a new object arrives. Recent work [19] has shown the importance of *admission control* — deciding whether to admit an object into the cache — especially for workloads where object sizes vary.

Caching algorithms differ in *complexity* (the processing time needed to make a caching decision) and *performance* (measured by the cache's *hit ratio*, the fraction of requests served from the cache). In general, higher complexity results in higher performance. The choice of algorithm design is usually constrained by the request arrival rate. For instance, operating systems need to make page replacement decisions at sub-millisecond timescales, so they often use simple algorithms like Least Recently Used (LRU) eviction. Web proxies see lower request arrival rates, so they can use slightly more complex algorithms [21, 22, 44] that take recency, frequency and size into account in making caching decisions.

Why video caching is different. When a user starts a video *session*, they can begin watching from any location in the video, but each session generally makes forward progress; once a session requests chunk $k, k < N$, of an N-chunk video it is highly likely to request chunk $k + 1$ soon in the future. In this paper, we argue that *caching algorithms that leverage the properties of video workloads are likely to be more effective than existing caching algorithms*. Earlier algorithms such as LRU (Least Recently Used) and GDSF(Greedy-Dual Size Frequency) [22], or more recent ones like LHD [15] and AdaptSize [19], are designed under the Independent Reference Model (IRM) [23], which assumes that every request is independent of others. This assumption does not hold for video. Other work has leveraged this predictability to pre-fetch chunks from the CDN origin [34, 39, 42]. However, for the pre-fetching to be effective, the CDN edge cache must know what bitrate the client will request the chunk at; it is not, in general, possible for an edge cache to know this without client coordination. Moreover, an erroneous pre-fetch decision can waste bandwidth and can pollute the cache.

In this paper, we explore the design of a video caching algorithm for CDNs that leverages properties of today's HTTP adaptive video streaming paradigm to achieve better performance than state-of-the-art caching techniques.

Contributions. Our first contribution is to understand which properties of the delivery paradigm can help improve caching. Using insights from a large video delivery service, we find that chunk requests are roughly periodic, with average inter-arrival time corresponding to the chunk playback duration. We also find the existence of a significant number of *singletons*: chunks cached upon the first request but never referenced thereafter. Singletons reduce the efficacy of caching by occupying cache space that more popular objects could use.

Our second contribution is the design of AViC[2] which leverages these two observations and combines an eviction strategy with admission control. Eviction uses a simple heuristic to estimate a chunk's next request time and evicts the chunk with the farthest estimate. Admission control uses a data driven approach to train a binary classifier that predicts whether a chunk is a singleton.

[2] AViC stands for **A**daptive bitrate **Vi**deo **C**ache

However, naive implementations of eviction and admission control can significantly increase the computational and memory complexity of the caching algorithm. When a chunk arrives, AViC needs to update its reference estimates for every other chunk in the video, which can be computationally expensive. Moreover, for effective admission control, AViC may need to maintain metadata about videos and chunks long after their eviction. Our third contribution is a suite of optimizations that reduce computational complexity to well below the chunk inter-arrival time, and the memory complexity to that of competing state-of-the-art algorithms.

Using request traces from a large video delivery service, we show that AViC outperforms existing caching algorithms. It improves upon state of the art algorithms such as LHD and AdaptSize by up to 15% on byte and object hit ratios. These savings allow AViC to reduce wide-area traffic by more than 39% over competing algorithms, and to reduce requests to origin servers by more than 18%. Many CDNs use LRU, and AViC improves upon LRU significantly: an LRU cache requires up to $3.5\times$ the size of an AViC cache to match its hit ratio. This finding is important because CDNs have thousands of front-ends and they partition their caches at each front-end across video content providers, so requiring a smaller cache for achieving the same hit ratio enables the CDN to serve more customers with the same front-end infrastructure.

2 PROPERTIES OF VIDEO DELIVERY

We analyzed a large trace of video requests to understand properties of video delivery that we could exploit to design a caching algorithm for video.

Dataset. We used a dataset containing CDN HTTP GET request logs from a popular video service in the US. This service provides access to a large library of on-demand videos. Clients access content from the service either using set-top boxes connected to a residential broadband network or through mobile applications for Android and iOS platforms which may connect through a 3G/4G cellular network or WiFi.

The dataset is from a single CDN over a period of 5 weeks and has 585 M requests for 620 TB of aggregate video data transferred, where 120 M requests are from clients in 3G/4G cellular network and 465 M from residential broadband.

Each entry in the dataset describes a single HTTP GET request for a video chunk. It records the request arrival time, a unique ID for the video, the chunk number within the video, the bitrate, a unique session ID, and the size of the response returned for the request. It does not contain any form of user-identifiable or personal information.

In analyzing this dataset, we uncovered three facets of video delivery pertinent to the design of caching algorithms: (a) Large variability in object (chunk) sizes; (b) Predictability of chunk inter-arrivals; and (c) The prevalence of *singletons* (chunks requested exactly once). The following paragraphs quantify these observations.

Chunk size variability. Recall from §1 that publishers encode chunks at multiple bitrates, so clients can adapt download quality using ABR algorithms [12, 28, 35, 40, 43]. The video service we study uses 7 encoding bitrates. Figure 2 shows the distribution of sizes of requested chunks separately for the cellular and residential clients and combined for the full set of clients. Cellular downloads

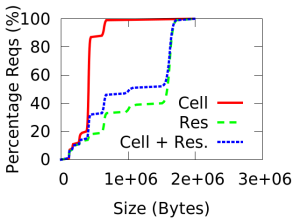


Figure 2: Chunk size distribution for cellular and residential clients

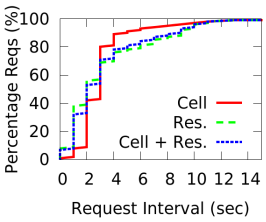


Figure 3: Intra-session request interval of client players

see a range of chunk sizes, from 150 KB to about 800 KB, whereas residential customers see chunk sizes up to 1800 KB, 12 times the smallest chunk size.

Size variability is an important factor in cache efficacy. The earliest caching algorithms (for page replacement or file buffer management in OSs) cached fixed size objects. Optimal caching algorithms (like Belady's MIN [16]) exist for this setting. However, when objects vary in size, especially by an order of magnitude as is the case in our dataset, caching decisions must take object size into account when making admission control and eviction decisions. This is because admitting a large object in the cache sacrifices the opportunity cost of instead caching smaller objects with potentially higher aggregate hit ratio.

Finally, an algorithm that caches objects of varying sizes must perform well both by *object hit ratio*, the fraction of requested objects served from the cache, and *byte hit ratio*, the fraction of requested bytes served from the cache. This is especially important for caching video: an algorithm that sacrifices byte hit ratio for high object hit ratio results in lower bandwidth savings over the wide area network, while one that sacrifices object hit ratio for higher byte hit ratio can result in a higher fraction of requests reaching the origin server.

Request arrival predictability. As discussed in §1, client players gradually download successive chunks in a video. These players use playback buffers which limit advanced buffering to a few tens of seconds of video. Once the playback buffer is full, a player makes another request only when space becomes available in the buffer. This suggests that request arrivals should be roughly periodic: if most clients watch videos at the normal playback rate (instead of, for example, fast forwarding or rewinding), the average request inter-arrival time should be roughly the duration of a chunk. More precisely, it should be slightly smaller than the chunk duration, because the player anticipates buffer availability and issues the next request before the playback completes.

Figure 3 shows the CDF of request intervals *within a single session*, separately for cellular and residential clients as well as the combined dataset (Cell + Res). Almost 89% of cellular requests and 79% of residential requests arrive within 4 seconds, the chunk duration used by this video service. Furthermore, the tail of all distributions stretches out beyond 10 seconds. We conjecture (but cannot conclusively verify, because we lack player-side instrumentation) that these larger intervals may be due to stalls, paused playbacks or players releasing more than one chunk from the buffer before re-filling.

The mean inter-arrival times for the cellular devices is 3.01 s, while for the full workload is 3.17 s, and the standard deviations are respectively 1.89 s and 2.85 s. While there is some variability in these inter-arrival times, the fact that the mean inter-arrival time matches our intuition about player behavior (see above) is encouraging, and suggests that future chunk request times might be predictable.

Such predictability is important for caching algorithms: intuitively, caching algorithms strive to evict objects likely to be referenced farthest in the future [16]. For most workloads, estimates of future reference times are difficult to obtain, but that may not be the case for video, as our data suggests.

The prevalence of singletons. An unusual aspect, at least in our data set, is the prevalence of *singletons*: chunks requested exactly once over a long period of time. Figure 4 shows the CDF of the number of references to a chunk, within an interval of 12 hours for different types for clients. For cellular clients over this 12 hour period, 51% of requests belonged to chunks not referenced again within that time interval. For residential clients 36% of requests over a 12 hour period belonged to chunks which were not referenced again. We also experimented with a longer interval of 24 hours (graph omitted for brevity). Over a 24 hour interval, 43% of cellular and 29% of residential requests belonged to chunks which were only referenced once.

A singleton can *pollute* a cache, by excluding more popular objects from the cache, so a well-designed caching algorithm must strive to exclude singletons. Size variability exacerbates the problem: large singletons can exclude several smaller objects from the cache, thereby adversely affecting both object and byte hit ratios.

Singletons can occur because the corresponding chunk belongs to a highly unpopular video (as with web pages, video popularity is heavy-tailed [14, 24]). Consider Figure 5, which shows the CDF of the number of user sessions for videos which contributed singletons for different types of clients. Among cellular clients 46% of singletons belong to videos with a single session whereas for residential clients only 38% of singletons belong to videos with a single session.

Among residential clients, up to 18% of singletons are from videos which saw at least 10 sessions over a 12 hr period. To understand how videos with more than one session can result in singletons, consider the following scenario. Suppose user A streams a video v and is shortly followed by another user B who also streams the same video. User A's network is stable and well provisioned to allow viewing at a high bitrate whereas user B's network is unstable and can only support a lower bitrate. Due to user A's session, the high quality bitrate gets cached at the server, however, requests from user B will result in cache misses because it is requesting a lower bitrate not used by user A, so user A's chunk is a singleton.

To quantify this intuition, Figure 6 shows the distribution of requests which used a particular bitrate, for cellular and residential clients and for the combined dataset. Almost 65% of the cellular requests accessed bitrate 3, but few requests accessed bitrates 1, 5 and 6, likely resulting in singletons. Across the residential trace, bitrate 6 is the most preferred bitrate, accounting for 60% of all requests, but other bitrates account for smaller fractions of requests. This disparity suggests that caching algorithms must make eviction and admission control decisions at the granularity of chunk bitrates.

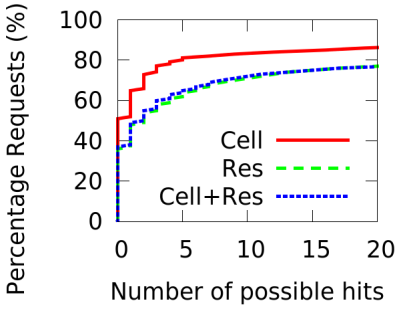


Figure 4: Caching potential of requests

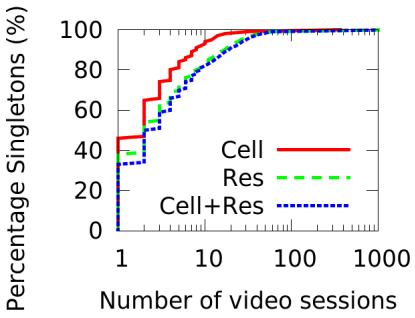


Figure 5: Popularity of videos which contributed singletons

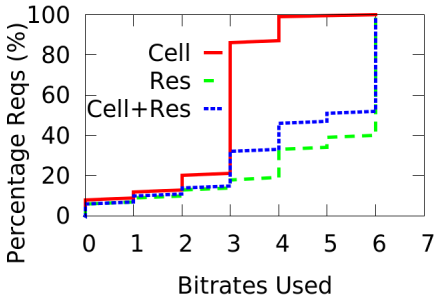


Figure 6: Bitrate usage across different types of clients.

3 AVIC DESIGN

In this section, we describe how we leverage the video delivery properties discussed in §2 to design an effective caching algorithm, called AViC, for video caching in CDNs. AViC has two components: eviction and admission control.

3.1 Overview and Challenges

Eviction. AViC exploits the *predictability of request arrivals* within a session to estimate the arrival time of the next request for a given chunk (its *request estimate*). This enables us to design an eviction strategy that evicts the chunk whose request estimate is farthest in the future. The intuition for why such an algorithm would perform well comes from the design of MIN [16], a cache replacement algorithm that replaces objects farthest in the future. MIN is optimal for fixed size objects; we know of no optimality result for caching algorithms with variable sized objects (as in our setting) [18].

However, AViC must surmount several challenges in designing eviction policy: it must estimate request estimates for chunks for inactive sessions (those for which no chunk exists in the cache), must determine when to update request estimates and how to keep these estimates up-to-date, and also obtain request estimates at the granularity of chunk bitrates given *chunk size variability*. §3.2 describes how AViC addresses these challenges.

Admission control. Based on the *prevalence of singletons*, AViC's admission control component aims to prevent singletons from polluting the cache. This goal translates into a simple objective: determining whether a chunk is a singleton or not. This is a binary classification problem, and AViC uses trace data to train a machine learning classifier. The key challenge in the design of the classifier is the choice of classification approach, and feature selection. In particular, feature selection must account for *chunk size variability*. §3.3 describes AViC's admission control component.

Efficiency. AViC must make caching decisions faster than request inter-arrival times, and its memory footprint must be comparable to competing caching algorithms. Naive implementations of eviction can result in high processing times especially for updating request estimates. Similarly, both eviction and admission control require maintaining meta-data for videos whose chunks are no longer in the cache, which, without careful design, can inflate AViC's memory footprint. §3.4 describes how AViC achieves its efficiency goals.

3.2 Eviction Policy

AViC evicts the chunk with the farthest next estimated request time. To achieve this, AViC's eviction policy consists of three key design elements. First, it maintains chunk meta-data (*e.g.,* request estimate, chunk ID) in sorted order by using a max-heap data structure keyed on the request estimate. It inserts this meta-data into the max-heap when the chunk is first requested and then maintains it as long as the chunk exists in the cache. Second, it keeps these estimates fresh by detecting, and correcting stale estimates. Finally, since multiple bitrates are available for each chunk, AViC takes this into account when updating request estimates in the max-heap. We now describe each of these design elements in detail.

Maintaining sorted next request estimates. To calculate next estimates for chunks, the eviction policy relies on three key operations: *Create*, *Estimate* and *Update*. AViC invokes *Create* on each cache miss, and *Estimate* and *Update* on both cache hits and misses.

***Create*.** This operation instantiates meta-data associated with a video including cached chunks of the video and ongoing sessions. If the requested chunk belongs to a video v that has no other chunk currently cached, we create a new *video record* R_v for v. R_v stores information about ongoing video sessions by maintaining a map of session records. Each session record contains a sessionID (a 32 byte string), the last chunk requested by the session and the timestamp at which the chunk was requested. In addition to session records, R_v also contains v's mean session inter-arrival time.

***Estimate*.** This operation calculates the next request estimate for a chunk upon receipt of a request for the chunk. Algorithm 1 shows the pseudocode of *Estimate*, suppose that a request arrives for the n-th chunk of video v on some session i. Let S_v denote the set of all ongoing sessions for v. *Estimate* finds that session from S_v which will next request the n-th chunk. Denote this session by j and further let the last chunk requested by j be the m-th chunk. Then *Estimate* calculates the next request estimate r_n for the n-th chunk as follows.

Where $m < n$, t is the current time (at which the n-th chunk request arrived), and d is the chunk duration (§2):

$$r_n = t + \big(n - m\big)d \tag{1}$$

This leverages the predictability of request arrivals, and estimates the next arrival for n to be $n - m$ chunk duration in the future.

However, *Estimate* must also consider the case when the search through S_v yields no existing session likely to request chunk n in

Algorithm 1: Pseudocode of Estimate operation

```
1  function Estimate (v, n, t)
   Input: v : the parent video of requested chunk
   Input: n : the index of requested chunk
   Input: t : the current unix timestamp
   Output: r_n : the next estimated request for chunk
2  r_n ← ∞
3  S_v ← v.sessions
4  for j ∈ S_v do
5      m ← j.lastChunk
6      if m < n then
           /* let d be duration of a chunk                    */
7          est ← t + (n − m) × d
8          if est < r_n then
9              r_n ← est
10         end
11     end
12 end
13 if r_n = ∞ then
       /* the case when no other session will request n
          in future i.e. m ≥ n for all j in S_v               */
14     r_n ← t + v.interArrival + n × d
15 end
16 return r_n
```

the future. This happens when $m \geq n$ for all ongoing sessions. In this case, the request for n must come from a new session (in all our analyses, we assume rewinds and fast-forwards are rare). If the average session inter-arrival time for v is I, then:

$$r_n = t + I + nd \tag{2}$$

Thus, the estimate r_n in this case is offset by the time it would take for a new session for v to arrive.

***Update*.** A single chunk request for any chunk in a video can cause the next estimates of all other chunks in the video to change. Hence all these chunks need to have their next request estimates re-computed. The *Update* operation achieves this. When a request arrives for a chunk of video v, and regardless of whether the request results in a cache hit or miss, *Update* scans the max-heap and extracts the metadata of all chunks of video v, then calls *Estimate* for each chunk, obtains the updated request estimate, and updates the metadata associated with the corresponding chunk.

Keeping request estimates fresh. AViC invokes *Estimate* for a cached chunk only when a request arrives for it. Videos whose sessions were active in the recent past, but have since become idle, can have stale request estimates for their chunks. Furthermore, these chunks may be situated deep in the max-heap by virtue of the estimates they had acquired while popular and thus are unlikely to be evicted. This results in *cache poisoning* which can adversely impact cache efficacy.

To address this, AViC maintains video records in an LRU list. The LRU list allows it to quickly find out which videos have not had a recent session. On each request of a chunk, AViC picks the least recently used video and performs *Update* on its chunks as if the requested chunk belonged to this video (in addition to updating chunks belonging to the video of the requested chunk). This ensures that stale videos will eventually have their estimates recomputed.

Request estimates for different bitrates. So far we have described the design of AViC implicitly assuming a single bitrate per video. However, publishers use multiple bitrates for their videos [11]. This implies that, when a chunk request for the m-th chunk arrives, AViC must estimate, for each chunk index i in the video, a request estimate for every bitrate b corresponding to that estimate. Specifically, consider a service that uses 3 bitrates, b_1, b_2 and b_3. Let's say that a session requests chunk 1, and chunk 3, encoded at b_2 and chunk 5, encoded at b_3 are in the cache. Should AViC update request arrivals for both of these using Equation 1?

One possible approach for future request estimation with multiple bitrates is to preempt the actions of a client player and prioritize the bitrate it is likely to request [39]. However, this approach requires intricate knowledge of client player's adaptive bitrate (ABR) algorithm. Given the range of ABR algorithms [12, 28, 30, 35, 40, 43] and a large user base comprising of range of heterogeneous devices [11] this can be challenging.

AViC adopts a data driven approach instead. Figure 6 shows that some bitrates in our dataset are more popular than others. AViC uses this relative popularity of the bitrates to weigh the estimates, based on the idea that less popular bitrates will have a request estimate farther in the future than more popular ones. It uses a simple heuristic, inflating the request estimate by the relative popularity. For example, suppose a video uses just a standard definition (SD) and a high definition (HD) bitrate. Further, assume that clients are 4 times as likely to prefer HD bitrates than SD. Then to calculate the request estimate for an SD chunk, AViC would compute it by inflating the estimate for the HD bitrate by 4 times.

More precisely, say a video uses k bitrates $b_1 \ldots b_k$. Let $w_j, j \in 1..k$ be the ratio of the number of accesses to chunks with bitrate b_j to chunks with bitrate b_i where b_i is the *most frequently accessed bitrate*. Then, AViC adapts Equation 1 as follows to compute the next request time for chunk n encoded using bitrate b_j:

$$r_{n_{b_j}} = t + \frac{(n-m)d}{w_j} \tag{3}$$

It adapts Equation 2 similarly, and we omit this for brevity. To compute w_j, AViC maintains a counter of accesses for each bitrate b_j for each video.

3.3 Admission Control

To address the prevalence of singletons, admission control simply seeks to detect whether a chunk is likely to be a singleton. If so, it prevents that chunk from entering the cache. Admitting singletons not only takes up cache space for more useful chunks but can also evict chunks that contribute to higher hitrates.

We model singleton detection as a supervised binary classification problem and train a Gradient Boosted Decision Tree (GBDT) [31] classifier. This classifier outputs a probability estimate for a future reference of a chunk; we apply a threshold τ on this probability [31] to admit the chunk.

We chose GBDT for several reasons. First, it is fast: training a single model takes less than 10 minutes on a dataset comprising of 100 M requests. Second, the size of the resultant model is small,

Feature	Description
Day	The day of the week
Time	Time of day in 24 hours format
Chunk size	Size of chunk in bytes
Chunk index	Index of chunk in video
Chunk bitrate	Bitrate of requested chunk
Total video sessions	Global No. of video sessions
Video sessions 24 hrs	24hr average no. of video sessions
Video interarrival	mean interarrival of video sessions
Video last interval	Time since start of last session

Table 1: Features used by admission control classifier.

with an average size of 175.86 KB. Third, GBDTs (and most decision tree based classifiers) are generally more interpretable than various deep learning techniques. Finally, they are quick to query and scale well with requests [17].

A rule-based alternative for admission control would analyze the video workload and determine admission control rules. For instance, Figure 6 showed that cellular clients seldom used higher bitrates for videos. Using this information it may be possible to design an effective admission control which denies admission to high bitrate chunks. However, it is not clear whether such a rule based system can be easily designed manually. More importantly, it would require laborious manual analysis when the workload changes or when a publisher changes bitrate choices.

The design of the classifier is non-trivial for two reasons. First, the definition of a singleton presupposes a *time horizon*, the time over which a requested chunk is never requested again. Choosing the time horizon is our first challenge. The second challenge is the choice of features.

Choosing the time horizon. In a cache, an object o is a singleton if (a) o enters into the cache upon first access, and (b) the caching algorithm evicts o before its next request. The time between first access and eviction is the time horizon. Time horizons may depend upon cache size. An object o may be a singleton with respect to a 32 GB cache but may not be a singleton with respect to a 64 GB cache (the latter holds more objects, so the next request for o may arrive before the caching algorithm evicts o). This observation motivates AViC's approach of *training different classifiers for different cache sizes*. This additional training does not impose a burden on large CDNs with significant compute resources and GBDTs require minimal resources to begin with.

But to train the classifier, we need to estimate the time horizon for a given cache size. We chose a simple estimator: the time to completely replace the contents of a cache using a FIFO policy. For each cache size, we can estimate the time horizon cheaply by simulating a short trace through a FIFO cache of the corresponding size.

After estimating the time horizon T, we generate training samples from our traces by marking as a singleton any chunk request whose next request is more than time T in the future.

Feature selection. The second challenge in classifier design is feature selection. Intuitively, several features of a chunk are likely to predict whether it is a singleton or not. For example, if chunks arrive infrequently, or video sessions arrive infrequently, then a chunk may likely be a singleton. If the chunk, or its corresponding video, are less popular, singletons might arise (§2). Finally, the size of a chunk can potentially be a signal indicating singleton status: as Figure 6 suggests, some bitrates are more likely to be singletons. For this reason, our GBDT classifier uses these features (Table 1). In §4 we analyze the importance of each feature for classification.

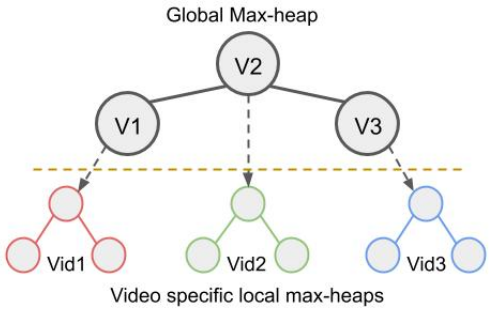


Figure 7: Hierarchical arrangement of max-heaps

Other details. In our evaluations, we train a model for each day (24-hour period) using request logs from the previous 2 weeks (3 week windows only provided marginal benefits). This re-training accounts for changes in the video workload. We can do this quickly because GBDT trains fast.

3.4 Performance Optimizations

AViC must make caching decisions faster than requests arrive, and must have a memory footprint comparable to existing caching algorithms. This section describes optimizations that speed up *Estimate* and *Update* operations, and reduce metadata requirements for admission control.

Optimizing *Estimate*. To calculate the request estimate for a chunk, *Estimate* must scan all ongoing sessions of a video to find the session likely to request the chunk earliest. However, searching though all sessions is unnecessary since some sessions may already have gone past the particular chunk. For example consider the scenario where a video v contains 30 chunks. The video has three ongoing sessions A, B and C. Assume that A last requested chunk 3, B requested 8 and C last requested chunk 27. When a request for chunk 9 arrives from B the only session that is likely to request chunk 9 again in the future is A (assuming A is unlikely to fast forward, and C unlikely to rewind the video).

To narrow the search space of candidate sessions, AViC uses a nested hashmap (hashmap of hashmap) data structure to store session records. The outer hashmap uses a bucket of chunk indices as keys. Each bucket then maps to another hashmap which maps a chunk index in the bucket to a list of sessions which last requested the chunk. If we consider a bucket size of 5 for the outer hashmap, then in the example above, video v has a total of 6 buckets (30 chunks divided by 5) and each of these buckets maps to another hashmap with a maximum size of 5 (5 chunks in each bucket). So, A gets hashed to bucket 0 in the outer hashmap and then in the nested hashmap the chunk last requested by A (chunk 3) stores A's session record. Similarly, B maps to bucket 1 in the outer hashmap and C to bucket 5. When a request for chunk 9 arrives from B, AViC starts searching from bucket 1 and moves backwards, searching till *any* other session that is going to request chunk 9 in the future is found. This results in a constant ($O\left(1\right)$) time search complexity, because

after indexing in to the outer hashmap, the search space is restricted to the size of bucket. AViC uses a bucket size of 25 chunks, which requires a total of 432 buckets for a 3-hour video with 4-second chunks.

Optimizing *Update*. This operation recalculates the request estimates for all chunks of a given video. This can be expensive since a video can have thousands of chunks.

Observe that *Update* does not need to update every chunk. Instead, for a given video, it only needs to update that chunk with the farthest next estimated request. As long as this chunk's request estimate is accurate, AViC can make eviction decisions. However, searching this local farthest chunk is expensive using a single max-heap.

To optimize this lookup, AViC organizes the cache as a hierarchy of local max-heaps and a global heap (Figure 7). There is one local heap for each video, which contains its chunks sorted according to the request estimates. The global max-heap has pointers to the root nodes of all the local video specific heaps. Then, finding the farthest chunk in a video is a single heap lookup.

With this design, to evict a chunk AViC first consults the global heap to find the video which holds the chunk with the farthest next reference. Let c_f denote this chunk and assume that video v holds c_f. AViC then removes the meta data associated with c_f from v's local heap. After removing the chunk, if v's local heap is non-empty, AViC updates v's node in the global heap with the new root of the local heap. It can then evict c_f. These steps require three heap operations.

Optimizing *Update* also reduces the amount of work AViC needs to do to avoid stale estimates (§3.2). On each chunk request, AViC only updates the request estimate for the farthest chunk of the least recently used video.

Reducing metadata storage. AViC's eviction and admission control policies use historical information associated with videos. AViC's eviction algorithm orders chunks by their next request estimates (§3.2). To compute these estimates, Equation 2 needs the inter-arrival time of sessions for a video. Likewise, for admission control (§3.3), AViC assembles a set of features on each chunk request, then uses these to query the GBDT model. These features are video-specific (Table 1). As such, AViC might need to maintain state for every video, which can result in a high memory footprint.

To reduce memory footprint, observe that AViC may not need to maintain accurate state for every video. For instance, consider a video v which has inter-arrival of I between successive sessions and further suppose that I is significantly large, *e.g.,* $>$ 24 hours. In this case, if AViC caches chunks of v, it would likely evict them much earlier than I — the interval after which another session for v arrives. In other words, chunks of v will be singletons. For such unpopular videos, maintaining accurate estimates of inter-arrivals does not contribute towards improving the performance of AViC.

Based on this insight, instead of maintaining state for all videos, AViC maintains historical state for the N most recently requested videos. It does this by maintaining video records in two separate LRU lists. An *Active* LRU list which keeps track of the videos currently in cache and an *Inactive* LRU list which keeps track of videos requested recently but are no longer active. When AViC evicts a video v from the Active list it moves it to the Inactive list but preserves its state. If a session for v arrives while it is in the Inactive list, AViC moves v back to the Active list. If no further sessions arrive while a video is in the Inactive list, AViC eventually evicts the video and removes its state. AViC uses a value of $N = 5000$ which allows it to achieve high performance while keeping AViC's memory footprint comparable to or within the memory footprint of other algorithms. We evaluate AViC's sensitivity to N in §4.

4 EVALUATION

This section compares AViC to several other caching algorithms and quantifies the benefits of our design choices.

4.1 Methodology

Implementation. We have implemented AViC in Go [3]. This implementation takes as input a request trace and a cache size, processes the requests in the trace, and outputs the byte and object hit ratios.

Dataset. We have a dataset containing 5 weeks worth of a subset of requests to a large video provider (§2). We use weeks 4 and 5 from our dataset to compare AViC with other alternatives, and the weeks 2, 3 to train the admission control classifier[3]. In weeks 4 and 5, the data contains 117 M requests with a total of 124 TB of aggregate requests. Of the 117 M requests, 24 M correspond to cellular clients and the remaining 93 M correspond to residential clients. We evaluate AViC and other alternatives separately for cellular and residential client requests, and also collectively over the 117 M requests.

Metrics. We compare performance using both byte hit ratio and object hit ratio (§2). Both metrics are important for video: a high byte hit ratio reduces backbone traffic from CDN front ends to origin servers and a high object hit ratio reduces the number of requests to CDN origin. Further, by showing both metrics together, our goal is also to show that AViC doesn't compromise one metric while achieving high performance on the other (as some competing algorithms do).

Baseline algorithms. We compare AViC against four algorithms: LRU, GDSF [22], AdaptSize [19] and LHD [15]. We pick LRU since it is one of the most widely deployed cache algorithms. GDSF combines frequency and size with recency to improve over LRU. AdaptSize combines admission control with LRU. Its admission control uses an adaptive size threshold to prevent admission of large objects into the cache. LHD uses *hit density* to maximize those objects in the cache which contribute most to the hitrate given their size. Together, these four algorithms cover a range of designs. Finally, to present results with reference to an approximate upper bound for the achievable hit ratios, we also show hit ratios of an *Oracle* which applies MIN [16] to our variable-sized workload. As discussed in §2, MIN is not optimal for variable-sized objects and we have not changed MIN's eviction strategy to take size in to account. To the best of our knowledge, no provable offline optimal algorithm is yet known for variable sized objects [18]. For our workload, therefore, MIN only provides a loose upper bound.

To evaluate these algorithms, we have used author-provided implementations for AdaptSize and LHD [1, 7], and have implemented LRU and GDSF.

[3] As discussed in §3 we also experimented with using the first three weeks to train the classifier, but it yielded marginal benefits.

Cache Size (GB)	LRU	GDSF	AdaptSize	LHD	AViC	Oracle
32	10.56	13.11	9.88	12.46	**18.05**	28.58
64	14.32	16.52	13.23	17.51	**23.14**	35.35
128	19.55	22.33	18.46	25.41	**29.98**	43.39
256	27.28	32.51	27.51	34.98	**38.75**	52.30
512	38.00	44.41	36.60	44.47	**48.17**	61.31
1024	50.22	54.58	41.28	52.23	**57.71**	69.58

Table 2: Byte hit ratio comparison for residential clients

Cache Size (GB)	LRU	GDSF	AdaptSize	LHD	AViC	Oracle
32	9.60	12.77	9.64	11.74	**15.75**	25.01
64	12.93	15.93	12.64	16.12	**20.01**	30.73
128	17.41	20.87	17.27	22.70	**25.75**	37.63
256	23.93	29.35	24.97	30.97	**33.41**	45.55
512	33.02	39.73	32.88	39.92	**42.38**	54.04
1024	43.71	49.64	37.82	48.41	**52.07**	62.48

Table 3: Object hit ratio comparison for residential clients

Cache Size (GB)	LRU	GDSF	AdaptSize	LHD	AViC	Oracle
32	16.27	18.87	14.44	18.46	**23.76**	38.08
64	22.30	26.93	21.32	27.39	**32.60**	46.18
128	31.02	37.60	29.75	37.40	**41.30**	54.50
256	42.03	46.53	39.85	45.64	**50.54**	62.02
512	53.34	55.62	46.34	55.26	**58.19**	67.28
1024	62.24	63.20	40.98	63.15	**64.24**	68.65

Table 4: Byte hit ratio comparison for cellular clients

4.2 Performance Comparison

Residential clients. Table 2 and Table 3 show the byte and object hit ratios over a range of caches sizes for residential clients. AViC consistently outperforms other algorithms both in byte and object hit ratios at different cache sizes. For instance, for a 128 GB cache, AViC improves over LRU by 24.0% and over GDSF by 17.6% in byte hit ratios. To put it another way, LRU needs $3.5\times$ the cache size to match the byte hit ratio performance of AViC. Finally, AViC is about 26.5% better than AdaptSize and about 10.5% better than LHD in byte hit ratio (with slightly lower gains on object hit ratio), both recently proposed algorithms, across the range of cache sizes we evaluate.

Cellular clients. Table 4 and Table 5 show the byte and object hit ratios for the cellular clients. Cellular clients tend to prefer slightly lower bitrates than residential clients and therefore exercise the cache differently. As with the previous results, AViC outperforms the competing algorithms, but by smaller margins. For a 128 GB cache AViC outperforms LRU by 18.9%, GDSF by 6.8%, AdaptSize and LHD by 21.2% and 7.2% for byte hit ratios. For cellular clients, an LRU cache requires approximately $2\times$ the cache size used by AViC to match its performance.

All clients combined. Table 6 and Table 7 show the byte and object hit ratios over a range of different cache sizes for our full trace dataset (cellular and residential client combined). AViC is able to consistently outperform all competing algorithms over the entire trace as well: up to 28.1% better than LRU and AdaptSize, and up to 18.6% better than LHD and GDSF. At larger cache sizes, the performance difference narrows: with a 1 TB cache, all algorithms perform comparably, and approach the Oracle.

Cache Size (GB)	LRU	GDSF	AdaptSize	LHD	AViC	Oracle
32	15.85	18.45	14.33	18.21	**23.01**	36.96
64	21.68	26.47	21.01	27.03	**31.61**	44.74
128	30.14	36.99	29.41	36.85	**39.47**	52.67
256	40.78	45.71	39.25	44.85	**47.81**	59.88
512	51.58	54.30	45.41	53.95	**55.58**	65.15
1024	60.09	61.47	40.77	61.47	**61.93**	66.53

Table 5: Object hit ratio comparison for cellular clients

Cache Size (GB)	LRU	GDSF	AdaptSize	LHD	AViC	Oracle
32	9.97	12.37	8.92	10.86	**16.94**	27.73
64	13.53	15.60	11.60	14.70	**21.72**	34.27
128	18.35	20.35	16.34	20.39	**28.16**	42.05
256	25.58	29.10	24.64	29.43	**36.78**	50.85
512	35.64	41.45	35.47	39.40	**46.47**	59.97
1024	47.77	52.23	44.84	48.44	**56.26**	68.50

Table 6: Byte hit ratio comparison for all clients.

Cache Size (GB)	LRU	GDSF	AdaptSize	LHD	AViC	Oracle
32	8.82	12.28	9.32	11.01	**14.37**	24.12
64	12.01	15.38	12.21	14.79	**18.27**	29.74
128	16.09	19.84	16.80	20.41	**23.68**	36.62
256	22.03	27.54	23.71	29.04	**31.33**	44.75
512	30.46	38.52	33.19	39.07	**40.90**	53.89
1024	41.13	48.99	42.63	48.51	**51.06**	62.89

Table 7: Object hit ratio comparison for all clients.

Explaining performance differences. We now try to understand what causes AViC to perform better than some of the competing algorithms.

GDSF and LHD. Unlike AViC, GDSF and LHD prefer to evict larger chunks over smaller chunks. This strategy can help achieve higher object hit ratios at the expense of lower byte hit ratios. When object sizes follow a normal distribution, this strategy may work well. However, the size distributions for video delivery may be significantly skewed, because video chunk sizes depend on factors such as the type of bitrates provided by the content provider, the network the clients are on, and the ABR algorithms used by client player. As Figure 6 shows, 60% of residential client requests were for the highest bitrate chunks. As such, penalizing the size of the chunks can put GDSF and LHD at a disadvantage because they are likely to prefer higher bitrate chunks for eviction — the same chunks which can potentially deliver higher byte and object hit ratios if cached.

To demonstrate this more concretely, we consider a variant of GDSF algorithm. This variant modifies the cost function used by GDSF by halving the size penalty to highest bitrate chunks. Figure 8 shows the byte and object hit ratios comparison between the original GDSF algorithms and this modified version (GDSF-MOD). Notice that GDSF-MOD achieves higher byte hit ratios while achieving

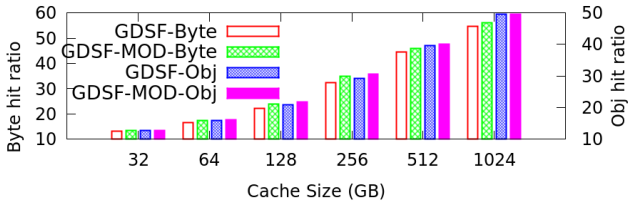


Figure 8: Performance comparison of a modified GDSF algorithm

higher or comparable object hit ratios. Even with this improvement, however, GDSF-MOD does not outperform AViC.

The penalty imposed on size also explains why the performance gap between AViC and baselines is smaller for cellular clients compared to residential clients. As Figure 6 shows, cellular clients tend to prefer bitrate 3 whose chunk size is $4.1\times$ smaller than bitrate 6. The adverse impact of chunk sizes impacts the cellular workload less, so competing algorithms fare slightly better for these clients.

AdaptSize. Admission control in AdaptSize uses a dynamic size threshold to decide whether to admit an object to the cache or not. It assigns a low probability of admission to objects which are large. This results in high object hit ratios for a Web workload whose size varies over a large range (1 B to 1 GB [19]).

However, our results for both residential and cellular clients show that AdaptSize is unable to outperform LRU, for two reasons. First, the size distribution of video chunks is in smaller range (150 KB to 1800 KB) than that of Web workloads. Second, unlike Web workloads which may have few large objects, in both the residential and the cellular traces, higher bitrates are more popular (Figure 6). 60% of residential clients prefer the highest bitrate whereas 78% of cellular clients prefer the 3rd and 4th bitrate. Because of these two factors, AdaptSize's admission control often prevents higher bitrate chunks from entering the cache, resulting in poor performance. Over the combined trace, AdaptSize narrowly outperforms LRU in object hit ratios (Table 7). A less skewed aggregate size distribution in the combined trace allows AdaptSize to find reasonable size thresholds to filter large chunks while still delivering high object hit ratios.

Unlike AdaptSize, AViC does not solely rely on the size of the chunk to decide admission. Instead it uses other factors such as the bitrate of the chunk, its position in the video (chunk index), and its popularity, which allows it to make better admission decisions.

Implications of results. While AViC's absolute gains over competing algorithms seem small, they are still significant. Many CDNs use LRU, and AViC's performance gains over LRU are significant. In particular, the fact that LRU requires 2-3.5$\times$ bigger cache, especially at small cache sizes, to achieve the same hit ratios as AViC is important. CDNs can have thousands of front end servers [20], and may serve hundreds of video publishers [11]. For each publisher, at each front end, a CDN might wish to virtualize the cache — partition a large cache into smaller caches dedicated to each publisher, to avoid cache interference between clients of different publishers. Our results show that AViC offers significant *cache consolidation* opportunities: to achieve today's hit ratios, a CDN need only provision half to a third size of caches in its front ends.

Figure 9(a) shows the hourly average CDN bandwidth savings and Figure 9(b) shows the average hourly requests prevented from reaching the origin. At smaller cache sizes (32 GB, 64 GB, 128 GB),

Cache Size (GB)	AViC	w/o adm ctrl	w/o BR weights	w/o stale avoidance	Oracle
32	23.76	22.66	20.19	13.52	38.08
64	32.60	30.48	27.37	19.00	46.18
128	41.30	39.40	36.49	26.55	54.50
256	50.54	48.93	46.20	36.83	62.02
512	58.19	57.74	56.19	48.96	67.28
1024	64.24	64.05	63.29	60.73	68.65

Table 8: Impact of AViC's components on performance

AViC on average saves up to 39.2% more bytes per hour than the closest competitor. Further, it saves up to 18.8% of requests per hour served by the CDN's origin servers. These savings in bytes and number of requests are significant. Video is the dominant component of Internet traffic, and wide-area bandwidth costs are significant [29], so even small reductions in wide-area traffic can be important. These savings can also benefit origin server provisioning.

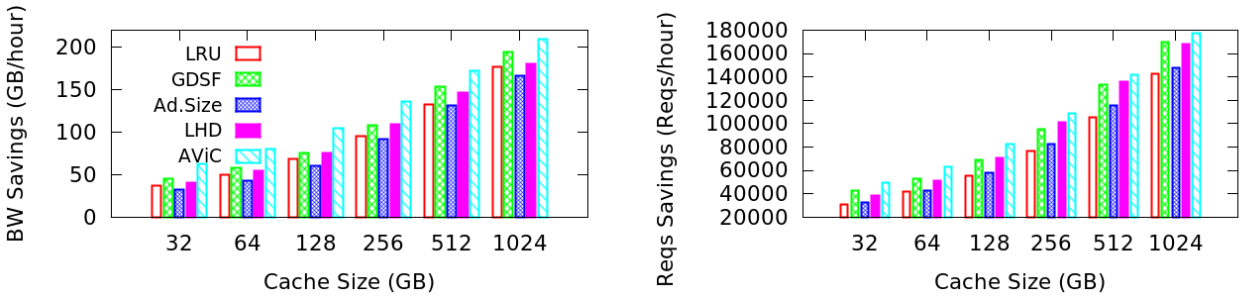
Finally, recent trends suggest a move towards higher-quality video streaming formats, such as 4K and Ultra-HD [9, 11]. This means that in the future content providers are likely to offer even higher bitrates. As such, the gap between AViC and algorithms which simply penalize object size, such as GDSF and AdaptSize is likely to be higher in the future.

Performance compared to Oracle. While AViC outperforms the competing algorithms, it exhibits a noticeable performance gap relative to Oracle. For example, at smaller cache sizes, AViC's hit ratios are between $60-65\%$ of Oracle's hit ratios. Three factors explain these performance differences.

AViC uses the mean inter-arrival time of sessions for videos for request estimates. However, inter-arrival times can take a few sessions to converge to a stable value, until which time estimates can be inaccurate. We found, using a 128 GB cache on the residential trace, that a significant fraction (3.42%) of non-singleton chunks, when they were evicted, had inaccurate estimates because not enough sessions had arrived to obtain a reliable mean inter-arrival time estimate. Even with good estimates for the mean inter-arrival time, AViC can have erroneous request time estimates for two reasons.

AViC assumes that seeks or pauses in video, or users reneging, are relatively infrequent. However, this assumption can be easily violated in the real world, invalidating future request estimates. AViC also does not account for client activity resulting from network throughput conditions. For instance, it does not know when a client is about to experience re-buffering, which can also induce error in request estimates. Finally, in the initial period of a session, client players tend to request chunks at smaller time intervals to quickly fill up the playback buffer. During this period, AViC can incorrectly estimate timestamps for future requests of a chunk, but this estimation error is upper bounded by the size of the playback buffer. Thus, if a client uses a 2 minute buffer, and (hypothetically) fills it up instantaneously at the beginning of the session, the error in a request estimate can be at most 2 minutes.

We computed the distribution of the estimation error for those chunks which had good estimates for the mean inter-arrival times. For a 128 GB cache on the residential trace, while the median error was only 4 s (about one chunk duration), the 99-th percentile error was 52.3 s. This long tail of the error distribution likely explains the difference between AViC and the Oracle.



(a) Per hour bandwidth savings comparison

(b) Per hour request savings comparison

Figure 9: Savings in bandwidth and requests to the CDN Origin.

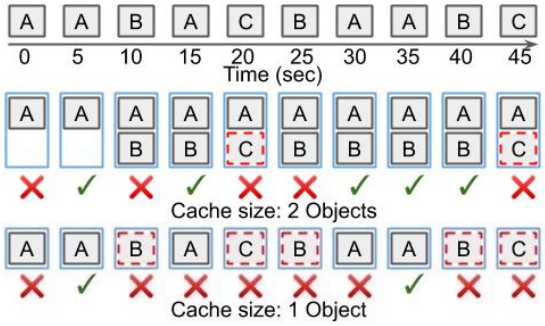


Figure 10: Singletons with a toy cache of size 2 and 1 object

4.3 Ablation Study

Each component of AViC contributes significantly to performance. Table 8 shows byte hit ratios when we remove one of the three main components of AViC: admission control, correct request estimation for bitrates, and stale estimate avoidance. These results are for cellular clients; the object hit ratio and results for residential and the full set of clients are qualitatively similar, and we omit these for space.

Especially at lower cache sizes, the mechanism to prevent stale estimates by updating the least recently used video contributes the most (over 10% gains at cache sizes up to 512 GB), since stale videos can potentially cause poisoning of the cache limiting its performance over the long run. Weighting requests estimates by bitrate popularity, and admission control, each improve performance by a few percentage points.

Table 8 also allows us to examine the merits of admission control in greater detail. The hitrate improvement due to admission control is relatively higher for smaller sized caches because the effectiveness of admission control is proportional to the percentage of requests which act as singletons in a given request workload. Smaller caches have a smaller time horizon (§3.3) causing a larger percentage of requests to act as singletons in a given workload. For larger cache sizes, a smaller fraction of requests act as singletons, so admission control shows lower gains.

To understand this better, consider Figure 10 which shows a simple request trace of three different objects A, B and C. Requests for object C arrive after 25 sec on average, whereas requests for objects B and A arrive every 15 sec and 8.75 sec (on average) respectively. The corresponding behaviours of AViC caches of size 2 and size 1 are also shown with check marks representing hits and cross marks representing cache misses. Notice that for the size 2 cache, requests for C act as singletons because it is evicted before a second request arrives for it, whereas for size 1 cache, requests for both objects B and C are singletons as admitting neither to the cache would result in any hits.

4.4 Time and Memory Complexity

Time complexity. To be practical, AViC must service requests faster than the rate at which they arrive (§3.4). Figure 11(a) plots two curves: the distribution of request inter arrival times and the distribution of AViC's request processing latency. To compute the latter, we use a single 2.4 Ghz core of an Intel Xeon server. Because of its careful data structure and request estimate design, AViC is able to process requests 1-2 orders of magnitude faster than request arrivals.

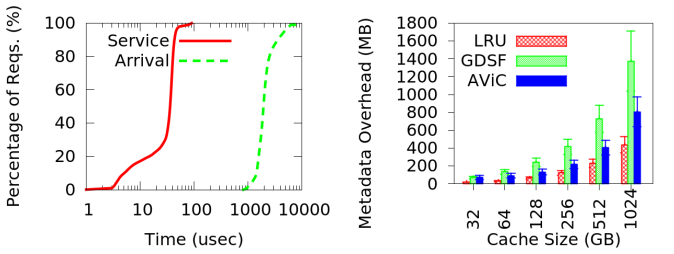
Memory complexity. To analyze the memory usage, we profile the amount of memory allocated to AViC by using Go's run-time profiler [5]. This is the preferred approach for profiling Go programs and provides accurate statistics on how much memory Go's runtime allocates to a process [4]. We periodically place calls using the run-time profiler to sample the memory size during a run of AViC Go.

Figure 11(b) shows the average size of the allocated memory for AViC in comparison to LRU and GDSF. The error bars show the standard deviation. As expected, LRU is the most efficient in terms of memory usage, whereas AViC uses less than $2\times$ the memory used by LRU. Notice that among the three algorithms, GDSF uses the most memory. This difference arises from the way Go's runtime allocates memory to dynamically sized data structures.

Both AViC and GDSF use heaps to maintain the metadata associated with cached objects. Heaps in Go use dynamic arrays. The size of these dynamic arrays are not fixed at compile time, instead the Go runtime initializes them to a default size. To grow the heap, the runtime resizes the array typically by a factor of $1.25 - 2\times$.

The difference in memory usage arises in the way GDSF uses heaps. By design, GDSF uses a single heap to track all chunks in the cache. The size of this single heap is much larger than the smaller heaps which result due to the hierarchical heap design used by AViC (§3). As a result, GDSF's single heap grows by larger amounts (in absolute terms) than AViC's heaps: the runtime individually resizes the latter's heaps.

Cost of Maintaining Session State. Recall from §3 that for each active video AViC maintains the set of ongoing session records. A



(a) Request inter arrivals and service latency

(b) Memory complexity compared to LRU and GDSF.

Figure 11: Savings in bandwidth and requests to the CDN Origin.

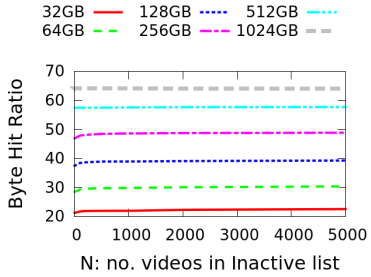


Figure 12: Byte hit ratio sensitivity to N

single session record uses 40 bytes: a 32 byte sessionID, a 4 byte integer to record the last chunk requested by the session, and a 4 byte integer to record the timestamp at which the last chunk was requested. As an upper bound, consider a server that is concurrently serving 10 M sessions. For this server, the cost of maintaining the session pool is 400 MB. However, we note that in practice the number of concurrent sessions video servers can handle, may be orders of magnitude lower. This is because video chunks can be large (typically between few hundreds of Kilobits to tens of Megabits) and a much smaller number of concurrent sessions can end up saturating the available bandwidth of a server.

Summary. Together, these results demonstrate that AViC (i) can handle a high load of requests without compromising performance and (ii) keeps memory overhead low compared to other algorithms. Since we use standalone implementations of LHD and AdaptSize, we could not use the same profiling approach to estimate their memory usage. We have left it to future work to instrument their implementations to obtain accurate memory usage for these algorithms, but expect that their memory footprint will be no lower than LRU.

4.5 Sensitivity Analysis

Global history. AViC uses an *Inactive video list* to maintain metadata associated with videos which do not have any chunks cached (§3). This metadata allows it to compute features needed for admission control as well as the inter-arrival times for videos needed for request estimates.

We now analyze how sensitive are AViC's results to N, the number of video records maintained in the Inactive video list. Figure 12 show the byte hit ratios (object hit ratio omitted for brevity) for different cache sizes as a function of N. Large cache sizes (512 GB, 1024 GB) are not sensitive to N. This is because large caches are big enough to already hold unpopular videos in the Active list of videos and hence do not benefit from the Inactive list. Smaller caches are sensitive up to around $N = 50$. AViC chooses to be conservative and uses a value of $N = 5000$ since the memory cost of maintaining 5000 video records is small; at this value, AViC is not affected by discarding history.

Admission control. Figure 13 shows the relative importance of different features to the performance of GBDT's classifier. The two most important features are the average number of sessions for a video over a 24 hour period and the inter arrival time of new sessions for a video. The bitrate of the requested chunk also plays an important role followed by other chunk level attributes such as the size of the chunk and the index of the chunk in the video. Notice that chunk index is a weaker predictor of singletons than the bitrate of the requested chunk. This is likely due to unpopular bitrates which can cause singletons to occur at any chunk index of a video. These relative importance results conform to intuition and suggest that the GBDT classifier learns request patterns well.

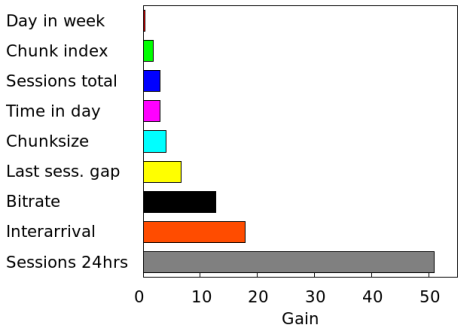


Figure 13: Importance of features to GBDT classification.

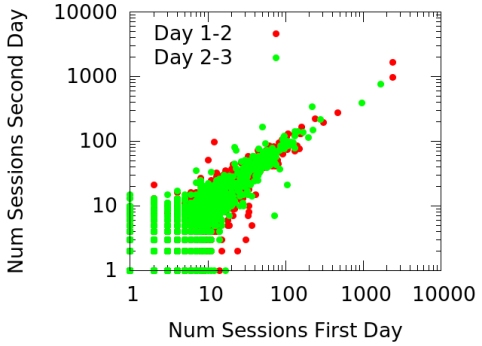


Figure 14: Correlation of video popularity over time.

To understand why the workload is amenable to learning, we analyze the correlation of popularity of videos over adjacent periods of 24 hours. Figure 14 shows this correlation over two sets of adjacent days. It shows that the videos which are popular tend to remain popular the next day and unpopular videos remain unpopular. We have also analyzed this correlation over 48 hour intervals and found similar results; we omit these for brevity.

5 DISCUSSION

Buffering strategy and user behavior. AViC's design assumes that client players limit advanced buffering to at most a few minutes. Today, most players use a 30 sec-240 sec playback buffer [2, 6]. We believe that this is unlikely to change due to: (i) bandwidth limitations and (ii) the risk of user session abandonment which can result in wasted bandwidth. We also argue that this assumption applies generally to a large range of client players because in designing AViC we have neither filtered out requests nor optimized for any particular type of player.

AViC's design also does not account for user behavior. As discussed in §4, user activity such as seeks and pauses can cause inaccurate estimates for future request timestamps. While we do not characterize how frequently user paused or navigated a video in our traces, our results use real traces which capture realistic user behavior. On these traces, AViC obtains high hit-ratios even though it doesn't model seeks and pauses. Future work can explore whether, and to what extent, estimating seeks and pauses can improve hit-ratios.

Importance of smaller cache sizes. Our evaluations use small cache sizes as well as large ones. We believe it is important to evaluate video caching for smaller cache sizes because CDNs statically partition a cache across multiple video content providers (to avoid one provider's video chunks from polluting another's cache). In this setting, an algorithm that has high cache hit rates for smaller sizes helps CDNs provision edge caches better.

AViC's generalizability. AViC's design is particularly optimized for adaptive bitrate video and it is unlikely to perform well for other types of workloads. However, any workload where future request timestamps can be estimated can potentially benefit from AViC's approach.

6 RELATED WORK

Caching can improve the performance of databases, key-value stores, operating systems and web proxies, to name a few. So, caching algorithms have been extensively studied in the literature [15, 19, 22, 25–27, 33, 36, 38, 44]. Many of these designs combine recency and frequency using different techniques. For example, Facebook's photo caching algorithm [27] uses four LRU lists and moves objects from lower LRU lists to higher lists on hits and evicts objects from the lowest of the 4 lists. Similarly, LRFU [33] augments a single LRU list with a LFU (least frequently used) list hence allowing the eviction policy to leverage recent history as well as older history from the LFU list. These general purpose algorithms assume the Independent Reference Model (IRM). As we have discussed in this paper, the IRM model does not necessarily hold for video delivery.

Algorithms can also use the size of an object when making eviction decisions. For example, the Greedy Dual Size family of algorithms [22, 44] penalize objects based on their sizes. This design evicts large sized objects first. AdaptSize [19] uses admission control to prevent large sized objects from getting into the cache, which is useful when object sizes vary such that the admission of a single object can evict a large number of smaller objects. LHD [15] computes a hit density which is a function of the objects size. The algorithm maximizes the hitrate density of the cache. AViC does not use the size alone but combines it with other attributes of the chunk for its admission control decisions.

Other prior work has observed that caching efficacy can be improved by leveraging properties of the application. PacMan [13] coordinates caching and eviction decisions across caches in a cluster running parallel applications, to ensure memory locality for all parallel instances of a task. This ensures lower task completion times. AViC focuses on caching video chunks at a single CDN edge cache, and leverages predictability of video access to improve performance.

Prior work has exploited the dependency between successive video chunks requests to design pre-fetching strategies [34, 39, 42]. Pre-fetching allows the cache to maximize hit ratios, but, for video, suffers from two limitations. First, it relies on accurately predicting client bitrate adaptation behavior. This is difficult to do in practice because of heterogeneity in client devices [11] and variety in ABR algorithms [12, 28, 30, 40, 43]. Second, these schemes can pre-fetch video chunks that a client may never request (*e.g.,* if the session terminates or user seeks or pauses playback) wasting wide-area network bandwidth. AViC does not assume knowledge of the bitrate adaptation algorithm, and never pre-fetches chunks. Instead, it focuses on estimating *when* a cached chunk will be requested again and evicts the chunk with the farthest estimate.

Recent work has explored machine learning techniques to design caching algorithms. LFO [17] pre-processes request traces to generate the caching behavior of an offline Oracle and uses decision trees to learn the behavior. AViC also uses decision trees but to solve a simpler problem of classifying singletons. More broadly, recent work has explored training deep neural nets to make eviction [37, 41] and admission control (RL-Cache [32]) decisions. While these deep learning techniques can offer higher accuracy, decision trees are light weight and more interpretable which makes them better suited to AViC. Future work can explore whether deep-learning based admission control is significantly more effective than AViC's approach.

7 CONCLUSION

In this paper, we describe AViC, a CDN front-end caching algorithm specifically designed for adaptive bitrate video. AViC leverages three properties of video delivery: chunk size variability, predictability of request arrivals, and the prevalence of singletons. Its eviction policy uses predictability of request arrivals to estimate future chunk request times, and its admission control predicts singleton chunks. AViC incorporates performance optimizations to reduce time and memory complexity. Using request traces from a real-world video service, we show that AViC outperforms a range of algorithms including LRU, GDSF, AdaptSize and LHD by up to 50% in byte and object hit ratios. Future work can attempt to close the gap between AViC and Oracle, by improving request estimates during pauses, stalls, or early in the session when players may request chunks back-to-back.

Acknowledgements. We express gratitude to our shepherd, Ganesh Ananthanarayanan and the anonymous reviewers for their constructive feedback which greatly helped improve the paper. We also thank Greg Smith and Gwo-Ming Jan for their help. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1413978.

REFERENCES

[1] AdaptSize. https://github.com/dasebe/webcachesim.
[2] Dash Industry Forum: Dash.js. https://github.com/Dash-Industry-Forum/dash.js.
[3] Golang. https://golang.org/.
[4] Golang: Debugging performance issues in Go programs. https://github.com/golang/go/wiki/Performance.
[5] Golang: runtime package. https://golang.org/pkg/runtime/.
[6] JW Player. https://www.jwplayer.com/.
[7] LHD. https://github.com/CMU-CORGI/LHD.
[8] Sandvine: Global Internet phenomena report . https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/2h-2014-global-internet-phenomena-report.pdf.
[9] Cisco: It Came to Me in a Stream... . https://www.cisco.com/web/about/ac79/docs/sp/Online-Video-Consumption_Consumers.pdf, 2012.
[10] Cisco: Visual Networking Index: Global Mobile Data Traffic Forecast Update 2016-2021 . http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html, 2017.
[11] Z. Akhtar, Y. S. Nam, J. Chen, R. Govindan, E. Katz-Bassett, S. Rao, J. Zhan, and H. Zhang. Understanding video management planes. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, 2018.
[12] Z. Akhtar, Y. S. Nam, R. Govindan, S. Rao, J. Chen, E. Katz-Bassett, B. M. Ribeiro, J. Zhan, and H. Zhang. Oboe:Auto-tuning video ABR algorithms to network conditions. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, 2018.
[13] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.
[14] M. F. Arlitt and C. L. Williamson. Internet web servers: workload characterization and performance implications. 1997.
[15] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.
[16] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 1966.
[17] D. S. Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, 2018.
[18] D. S. Berger, N. Beckmann, and M. Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst.*, 2018.
[19] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, 2017.
[20] M. Calder, X. Fan, Z. Hu, E. Katz-Bassett, J. Heidemann, and R. Govindan. Mapping the Expansion of Google's Serving Infrastructure. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, 2013.
[21] P. Cao and S. Irani. Cost-aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, 1997.
[22] L. Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. 1998.
[23] E. G. Coffman and P. J. Denning. *Operating Systems Theory (Prentice-Hall series in automatic computation)*. Prentice Hall, 1973.
[24] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of www client-based traces. 1995.
[25] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A Highly Efficient Cache Admission Policy. volume 13, Nov. 2017.
[26] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, 2015.
[27] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.
[28] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2014.
[29] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4), Aug. 2013.
[30] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies*, CoNEXT, 2012.
[31] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems 30*, 2017.
[32] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman. Rl-cache: Learning-based cache admission for content delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, NetAI'19, 2019.
[33] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, 1999.
[34] J. Liu and B. Li. A qos-based joint scheduling and caching algorithm for multimedia objects. *World Wide Web*, 2004.
[35] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2017.
[36] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03.
[37] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, NetAI'18, 2018.
[38] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, 1993.
[39] S.-H. Shen and A. Akella. An information-aware qoe-centric mobile video cache. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13, 2013.
[40] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman. BOLA: Near-optimal Bitrate Adaptation for Online Videos. In *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM, 2016.
[41] K. Suksomboon, S. Tarnoi, Y. Ji, M. Koibuchi, K. Fukuda, S. Abe, M. Nakamura, M. Aoki, S. Urushidani, and S. Yamada. Popcache: Cache more or less based on content popularity for information-centric networking. 10 2013.
[42] O. Verscheure, C. Venkatramani, P. Frossard, and L. Amini. Joint server scheduling and proxy caching for video delivery. *Proceedings of WCW 2001*, 2001.
[43] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, London, United Kingdom, 2015.
[44] N. E. Young. The K-Server Dual and Loose Competitiveness for Paging. *CoRR*, cs.DS/0205044, 2002.