Caching in Video CDNs: Building Strong Lines of Defense

Kianoosh Mokhtarian and Hans-Arno Jacobsen
Middleware Systems Research Group
Department of Electrical and Computer Engineering, University of Toronto, Canada
kianoosh@msrg.utoronto.ca, jacobsen@eecg.toronto.edu

Abstract

Planet-scale video Content Delivery Networks (CDNs) deliver a significant fraction of the entire Internet traffic. Effective caching at the edge is vital for the feasibility of these CDNs, which can otherwise incur significant monetary costs and resource overloads in the Internet.

We analyze the challenges and requirements for video caching on these CDNs which cannot be addressed by standard solutions. We develop multiple algorithms for caching in these CDNs: (i) An LRU-based baseline solution to address the requirements, (ii) an intelligent ingress-efficient algorithm, (iii) an offline cache aware of future requests (greedy) to estimate the maximum caching efficiency we can expect from any online algorithm, and (iv) an optimal offline cache (for limited scales). We use anonymized actual data from a large-scale, global CDN to evaluate the algorithms and draw conclusions on their suitability for different settings.

Categories and Subject Descriptors C.2 [*Computer Systems Organization*]: Computer-Communication Networks

General Terms Performance

Keywords Content delivery networks, video caching

1. Introduction

A significant fraction of today's Internet traffic consists of video streams served by major providers such as YouTube, Netflix and Amazon. This massive traffic is usually delivered to users through a Content Delivery Network (CDN)—a network of geographically distributed cache servers. For example, YouTube alone is estimated to serve 15–35% of the entire Internet traffic during peak hours across different continents [23].

Delivering such substantial (and growing) volume of traffic can incur significant monetary costs for CDN operators and ISPs, if it is not properly handled at the edge. Unlike the delivery of small-size, latency-sensitive content such as Web search and email, the main goals for delivering bulky video traffic at such scale are to *avoid high traffic handling costs and overload on bottlenecks*; in terms of latency, it is often just enough if the server-to-user RTT is maintained within a reasonable bound compared to the required initial buffering of the video. Video CDNs try to serve as much traffic as possible at the edge, to make the service economically feasible and prevent such huge volume from harming the rest of the Internet. However, the success in achieving these goals depends largely on how effective the servers of the CDN can *cache content locally* and serve as a line of defense against the massive traffic.

In small CDNs consisting of a handful of server locations, it is feasible (and more reasonable) to manage content centrally, i.e., deciding how to host the files over the servers and update them dynamically with the demand [2, 6, 25]. This is not the case in large CDNs which host an extensive and dynamic set of files with transient demand patterns. These CDNs include a large number of servers all around the globe [12, 16] and host an increasingly large catalog of videos, e.g., over 100,000 hours of video are uploaded to YouTube every day [1]. Therefore, content management at file level is offloaded to the individual cache servers based on the request traffic they each receive, rather than being managed centrally, where this traffic is mapped to the servers primarily based on costs, constraints and delay bounds. Once a proper mapping is established, the efficiency of a CDN is primarily a matter of how well the individual servers can manage their cached contents: To serve as many of the incoming requests as possible. In other words, each server is expected to maintain a dynamic collection of popular content while both incurring as little cache-fill traffic as possible and redirecting minimal requests to other (less preferred) CDN servers for that user IP. However, determining when to bring in new content upon cache misses or to redirect requests, keeping both ingress (cache-fill) and redirected traffic low,

EuroSys 2014, April 13–16, 2014, Amsterdam, Netherlands

ACM 978-1-4503-2704-6/14/04. . . $15.00.
http://dx.doi.org/10.1145/2592798.2592817

and being able to manage the tradeoff between the two based on the server's configuration, place new challenges before these servers which cannot be addressed by existing solutions in the literature (Section 2).

In this paper, we take a closer look at the operations of individual cache servers in a video CDN for managing their contents and minimizing the ingress and redirected traffic. This is critical for a planet-scale CDN given the several-Tbps volume of traffic: *Pushing the efficiency of caches by every 1% saves significant traffic*[1].

The contributions of our work are as follows.

1. We identify a number of key challenges for video caching in a planet scale CDN and analyze a baseline solution built on LRU policies.
2. We develop a cache management algorithm that carefully adjusts ingress and redirected traffic and increases cache efficiency by over 12% particularly for ingress-constrained servers.
3. We formulate the *offline* caching problem where the cache is aware of future requests, and we solve it using a greedy algorithm as well as linear programming relaxation (for small scales). These algorithms enable sound analysis of online caching algorithms by providing an estimate of the maximum caching efficiency we can expect from any online algorithm, i.e., with perfect prediction of access patterns.
4. We use actual data from a large-scale, global video CDN to evaluate the algorithms based on traces from several servers across different continents.

This paper is organized as follows. Section 2 discusses the system model and the requirements specific to our video CDN caching problem. We review related work in Section 3. A detailed description of the video caching problem is given in Section 4. Sections 5 through 8 present the four caching algorithms described above, followed by detailed experimental results in Section 9. Section 10 presents concluding remarks and the related open problems for future work.

2. System Model and Practical Challenges

We describe our target CDN model and the related challenges for video caching in this section.

The primary goals of a CDN delivering a significant fraction of the entire Internet traffic is to reduce cost and overload on link/server bottlenecks. For example, to serve a given user IP, usually only a few server locations are preferred, such as a remote server in the user's ISP [12, 16] or one behind a settlement-free peering connection between the CDN network and the user's ISP. Therefore, mapping users to servers based on metrics unaware of such considerations, such as through a content hash [9, 15], is not feasible. Moreover, given the thousands of server locations and billions of video files which are dynamically updated on the servers, it is not reasonable to make (logically) centralized decisions for placing the files on the CDN servers. We also do not require that availability and per-location popularity of the individual files, which have no strong correlation with the global popularity [28], are tracked in a centralized directory for mapping requests to servers[2].

Rather, in our considered CDN, user IPs are mapped to servers primarily based on cost, constraints and delay bounds. Then, the servers manage their own cache contents based on the request traffic they receive—similar to the non-cooperative pull-based model in the literature [18, 19]. In this network of cache servers, instead of bringing in and caching all requested video files, a server may simply *redirect* a small fraction of requests (for unpopular videos) to other servers possibly having the file or willing to fetch and cache it. This way, each server will eventually host the files that are most popular among the users it serves. Note that it is not cost effective for the server to act as a proxy instead, since that will just use up two servers' resources rather than simply redirecting the user to the other server. Compared to the complex/nonscalable methods for arranging files on the servers, repeatedly exchanging server cache indices, or centrally tracking per-server files (all to enforce that requests get the files always at their first point of landing), offloading content management to the individual servers and allowing a small fraction of requests to be redirected would simply suffice. This enables the valuable advantages of *simplicity* and *easy scalability* which are vital for a planet-wide CDN.

If redirecting a request, the destination server is selected similarly to the way the initial server for each request is selected: according to a mapping of worldwide user networks (IP prefixes) to server locations[3]. For instance, one can employ a secondary map which defines the destination of *redirected* requests from each user network. One example of an alternative (secondary) server location is a higher level, larger serving site in a cache hierarchy, which captures redirects of its downstream servers besides possibly serving some user networks of its own. Alternatively, the location to redirect to can be one which also peers with the user network(s) that the initial location serves. In all cases, note that selecting the server location to redirect to, similar to select-

[1] While one might presume the cheap cost of adding disk storage as a perfect solution to this problem, the Zipfian pattern observed for video accesses shows that just a few percent of higher cache efficiency requires up to a multi-fold increase in disk size, even assuming the data does not grow.

[2] Note that this is different from bucketizing the large space of file IDs (e.g., using *hash-mod*) and taking the bucket IDs into account for mapping. The latter is a feasible (and recommended) practice for dividing the file ID space over co-located servers to balance load and minimize co-located duplicates. The elements of this coarse grained set are aggregated file ID groups comprising files of diverse popularities, which cannot serve as *atomic* units stored in their entirety on one server and removed from another.

[3] The detailed schemes for mapping users to servers is beyond the scope of this paper. As discussed in Section 1, this mapping is primarily based on traffic constraints and costs, that is, the (estimated) demand of user networks, server capacities, constraints of the involved paths (e.g., some given peering capacity), delay bounds and so on.

ing the initial server for the request, is independent of the individual files requested.

We focus on the operations of the individual cache servers, the key building block for creating an efficient CDN. Historically, caching techniques have been employed in a wide range of computing applications such as Web proxies, databases, operating systems and CPU caches. The common operation of these caches is as follows: Upon a request for an object not in the cache, fetch it from the backend and store it based on a cache replacement algorithm. Some well known algorithms are to evict the object that is Least-Recently Used (LRU), Least-Frequently Used (LFU), or even Most-Recently Used (MRU) [20].

In the CDN we study, there are new challenges that cannot be addressed by standard caching solutions. These challenges are as follows.

Cache fill and redirect. Offloading content placement decisions to individual servers enables a flexible, horizontally scalable CDN, where a server may fetch and cache the requested file upon a cache miss, or simply redirect the request. There is an inherent tradeoff between ingress and redirected traffic: Limiting ingress will start to increase redirects and the other way around. It is therefore not only about the cache replacement problem anymore: It is about deciding between cache fill and redirection for each cache-miss, keeping both quantities low, and being able to manage the tradeoff between the two.

Different cache-fill and redirection preferences. The underlying connections across a worldwide CDN may include CDN-owned or leased links. Moreover, the connection of the CDN network to the rest of the Internet may be through peering or transit connections with different traffic handling costs for ISPs and CDNs. Therefore, the CDN servers can have uplinks with diverse constraints for cache-filling data. In addition, an important parameter for the willingness of a server to cache-fill is the utilization of its egress (serving) capacity. For a server at which the current contents suffice to serve as many of the requests as can fully utilize the egress capacity, there is no point to bring in new content upon cache misses. This is because still the same volume of requests will be served and redirected, hence wasted (and possibly harmful) ingress. Furthermore, sometimes the server's ingress traffic and the consequent disk writes can overload the disks and harm the read operations for cache-hit requests. We have observed that in this case, for every extra write-block operation we lose 1.2–1.3 reads. This becomes particularly a problem for servers that have smaller disks or are serving a more diverse request profile, both resulting in more cache misses and potential cache fill.

Thus, the CDN servers can have *different ingress capabilities* for cache-filling data. While some servers may not be constrained and be able to cache-fill normally, some others would prefer to redirect away a good fraction of cache misses rather than cache-filling them, specially if there are appropriate alternative locations not as constrained. That is, different *operating points* in the tradeoff between cache-fill and redirection percentage, while all yielding "the same byte hit rate", can translate into diverse consequences depending on the server. Therefore, the individual servers' *ability and willingness to cache-fill* (i.e., the CDN's preference at that server) is an important factor that needs to be properly taken into account in caching decisions.

Diverse intra-file popularities. The access patterns of the different parts of a video file are usually different. The first segments of the video often receive the highest number of hits compared to the rest [11]. It is thus not efficient to cache-fill or evict video files in their entirety. The consequent partial caching introduces new challenges for determining whether to cache-fill or redirect a request where the requested file has some parts present and some missing in the cache.

3. Related Work

Caching content on a network of servers has been considered in various forms in the literature. Cooperative Web caching techniques have been studied extensively, a comprehensive survey of which can be found in [21]. For the specific case of content delivery networks, different methods for caching and serving content to users have been proposed [19]. For example, the cache servers may exchange digests of the content they hold [22], report their (frequently updated) contents to a centralized directory [10], or be selected to serve users (partially) based on content hashes [8, 9, 15]. These techniques can optimize metrics such as the first-byte latency, but they are not suitable for our considered *scale* and *request mapping concerns*, as discussed in detail in Section 2.

The content placement problem across a CDN has become an interesting topic of research studied in [2, 4, 6, 25] among others. These approaches depend on global knowledge of the popularity and availability of files across the CDN server locations. Techniques for mapping users to CDN locations are investigated in [14, 26, 27]. Some of the prior approaches assume the requested files are always available (or are cache-filled) at the selected server [14, 27]. Wendell et al. present a general content-independent mapping algorithm based on specified policies [26]—an orthogonal work to ours which can benefit from the CDN model we consider.

The cache management problem has been analyzed extensively in the literature [20]. Belady's algorithm for offline caching evicts the object requested farthest in the future as the optimal cache replacement solution [5]. For online caching, LRU is known as the most popular replacement policy used in Web caches, for its simplicity and effectiveness given the temporal locality of access patterns [3, 7]. Variants of LRU, such as the Greedy Dual Size (GDS) [7] and GDS-Popularity [13] algorithms have been proposed to make it more sensitive to factors such as object size variability. We

deal with fixed-size chunks as described in the next section; also when bringing in a chunk range as a whole, the size is not a concern as it is the byte hit rate that matters. Other LRU variants try to incorporate access frequency information such as the LRU-K [17] and LNC-W3 [24] algorithms. Our workload demonstrates a long, heavy tail in the access frequency curve of the files. Putting the popular, frequently requested files aside which automatically stay on the caches, the files on the borderline of caching, i.e., those brought into or evicted from the cache, fall on this tail and usually have very few accesses in their lifetime in the cache. More importantly, earlier works address the classic problem of cache replacement, whereas in our case, it is about deciding between cache replacement and redirection and being able to manage the consequent tradeoff between ingress and redirected traffic based on the server's preference.

4. The Video CDN Caching Problem

We formally define our video caching problems in this section. Let R denote a request arriving at the server, which may be received from a user or from another (downstream) server for a cache fill. The request contains video ID $R.v$ and byte range $[R.b_0, R.b_1]$. We also define a timestamp $R.t$ set to the request's arrival. The server may serve the request or redirect it to another server (HTTP 302) as described. Although the server may store files partially, it needs to either fully serve or fully redirect a requested byte range, i.e., clients do not download a single byte range from multiple servers, though they may request different ranges at their own choice from different servers.

To simplify the support for partial caching, we can divide the disk and the files into small chunks of fixed size K bytes (e.g., 2 MB). Doing so eliminates the inefficiencies of allocating/de-allocating disk blocks to segments of arbitrary sizes. Rather, we deal with units of data uniquely identified with a video ID v and chunk number c. The chunk range for request R can be denoted as $[R.c_0, R.c_1]$ = $[\lfloor R.b_0/K \rfloor, \lceil R.b_1/K \rceil]$.

4.1 Ingress-vs-redirect Tradeoff

To incorporate the factors discussed in Section 2, namely the tradeoff between ingress and redirect ratios as well as the server's preference between the two, we define a cost C_F for every cache-filled byte and a cost C_R for every redirected byte ($C_F, C_R > 0$). We also denote their ratio by $\alpha_{F2R} = C_F/C_R$. It is in fact this ratio that eventually matters rather than C_F and C_R which are normalized as $C_F + C_R = 2$ (see Eq. (4)). Note that C_F and C_R are not actual monetary costs to be quantified. They are simply modeling variables whose relative value to each other, $\alpha_{F2R} = C_F/C_R$, is the controlling parameter that captures the CDN's preference at that server and defines the server's operating point (e.g., see Figure 5).

At some server locations, it is preferred that the servers ingress less traffic while a controlled increase in redirections is acceptable and harmless. An example of this case is the server locations that have saturated egress most of the time. The servers at these locations will serve the same amount of traffic and redirect the same whether cache-filling normally ($\alpha_{F2R} = 1$) or conservatively (e.g., $\alpha_{F2R} = 2$), hence wasted ingress. Furthermore, the server may be disk-constrained, where too much ingress and the consequent disk writes negatively impact regular disk reads as discussed earlier. Another example for conservative ingress is a location whose cache-fill traffic traverses the CDN's backbone network possibly shared with other services, and can overload it if ingressing too much. On the other hand, the alternative location to which it normally redirects (without traversing CDN backbone) is not as constrained, such as a location with a larger set of racks and disks, i.e., a deeper cache and less need for ingress, or a location closer to its cache-fill origin. In all the above cases, $\alpha_{F2R} > 1$ indicates that the server should limit its ingress: Fetch new content only when the new file is *sufficiently more popular* than the least popular file in the cache. The different operating points of a server based on the value of α_{F2R} are analyzed in Section 9.

Alternatively, at some server locations, ingress and redirection make no difference and there is no gain in favoring one over another. The most common example of this case is a remote downstream server located in the same network as the user [12, 16], from which any alternative location to redirect to or cache-fill from has exactly the same network cost and delay as from the user. $\alpha_{F2R} = 1$ expresses this configuration for our algorithms. Finally, $\alpha_{F2R} < 1$ (e.g., 0.5–0.75) indicates non-constrained/cheap ingress, such as an underutilized server with spare uplink capacity.

Note that in this work, our focus is not on detailed optimization of α_{F2R} values across the CDN caches. Rather, we focus on the key building block for CDN caching, which is an optimized underlying cache algorithm that complies to the desired fill-to-redirect ratio (α_{F2R}) for each server. Given such a cache with defined behavior, the CDN will have the means for further (global) optimization. While this is an orthogonal problem to the present work described in Section 10, we also note that this is not an over-complicated task in our current considered CDN: The common preference for servers is $\alpha_{F2R} = 1$, while a fraction of servers are constrained as explained above and can readily benefit from $\alpha_{F2R} > 1$—a default value of 2. Relieving these servers through optimized reduction and adjustment of ingress and redirected traffic is one of the main motivations for our present work.

4.2 Cache Efficiency

Given C_F and C_R, the cost for serving request R by fetching its full byte range equals $(R.c_1 - R.c_0 + 1) \times K \times C_F$. The cost of redirecting this request to be served from an alternative server equals $(R.b_1 - R.b_0 + 1) \times C_R$ for the

CDN and the cost of serving it directly from the cache is 0; there is indeed the cost of space and power for handling and serving each request, which is small and nearly the same for the three decisions, thus normalized as zero in the model. Note the different use of $R.b$ and $R.c$ in the above costs since a chunk is fetched and stored in full even if requested partially. We can quantify the total cost of a cache server as follows.

$$\begin{aligned}\text{Total cost} \quad = \quad & \text{num ingress bytes} \times C_F + \\ & \text{num redirected bytes} \times C_R. \end{aligned} \tag{1}$$

We note that a simple *cache hit rate* value can no longer represent the efficiency of a cache, since the cache may simply redirect every request that has missing chunks and obtain a hit rate of 100%. The redirection percentage alone is not a sufficient metric either: Imagine cache-filling all misses. For the special case of equal cache-fill and redirection cost ($\alpha_{F2R} = 1$), we can define the efficiency of a cache server as the ratio of requested bytes that were served directly from the cache, not cache-filled or redirected—imagine Eq. (2) with $C_F = C_R = 1$. For the general case with arbitrary α_{F2R}, we can define cache efficiency as follows based on the server's cache hits, fills and redirections with their corresponding costs.

$$\begin{aligned}\text{Cache efficiency} = 1 - & \frac{\text{Bytes served by cache-filling}}{\text{Total requested bytes}} \times C_F \\ & - \frac{\text{Redirected bytes}}{\text{Total requested bytes}} \times C_R \end{aligned} \tag{2}$$

$$C_F + C_R = 2. \tag{3}$$

Clearly, maximizing the cache efficiency metric defined in Eq. (2) is equivalent to minimizing the cache's total cost in Eq. (1). Moreover, because it is only the relative value of C_F and C_R to each other (α_{F2R}) that matters for caching decisions, we can simply enforce $C_F + C_R = 2$ in Eq. (3) to normalize. The constant 2 comes from the case with $\alpha_{F2R} = 1$ where cache efficiency simply equals the fraction of requested bytes that were served directly from the cache ($C_F = C_R = 1$ in Eq. (2)). The cache efficiency metric defined in Eq. (2) takes a value in $[-1, 1]$[4]. The value of C_F and C_R used in our algorithms can be obtained from α_{F2R} and Eq. (3) as follows.

$$C_F = \frac{2\alpha_{F2R}}{\alpha_{F2R} + 1}; \;\; C_R = \frac{2}{\alpha_{F2R} + 1}. \tag{4}$$

4.3 Problem Definition

Our video caching problems can be defined as follows.

[4] While a negative cache efficiency is not intuitive, we can imagine a server that is missing all requested files and is cache-filling them all—no cache hit and no redirection. This server would be performing more poorly when ingress is costlier than redirect (a negative cache efficiency) compared to when $C_F = C_R = 1$ (zero cache efficiency). We do not normalize the efficiency to $[0, 1]$ to better highlight the differences.

LRU-based Video Cache

HandleRequest(R)
// The case for warmup phase (disk not full) not shown.
1. t =*VideoPopularityTracker.LastAccessTime*($R.v$)
2. *VideoPopularityTracker.Update*($R.v$, t_{now})
3. **if** $t ==$ NULL **or** $t_{now} - t >$ *DiskCache.CacheAge*()
4. **return** REDIRECT
5. S = *DiskCache.MissingChunks*($[R.c_0, R.c_1]$)
6. *DiskCache.EvictOldest*($S.Size()$)
7. *DiskCache.Fill*(S)
8. **return** SERVE

Figure 1. Video cache with LRU-based popularity tracking and replacement.

Problem 1 (Online Cache)**.** *Given the sequence of past requests $R_1, \ldots, R_{i-1}$, a disk size and the current contents of the disk, make one of the following decisions for request R_i such that cache efficiency (Eq. (2)) over all requests is maximized: (1) serve the request and cache fill any missing chunks, or (2) redirect the request. For (1), also determine the chunks to be evicted from disk.*

Problem 2 (Offline Cache)**.** *Given the full sequence of requests $R_1, \ldots, R_n$ and a disk size, make one of the decisions as in Problem 1 for each R_i such that cache efficiency over all requests is maximized.*

The next four section describe our proposed video caching algorithms.

5. xLRU Cache for Video CDNs

LRU is the most popular replacement policy used in Web caches due to temporal locality, which is based on scoring objects by their last access time [3, 7]. The least recently used object is treated as the least popular one to be evicted to make room for new data. For the video caching problem, there is an additional option of not serving a request if the file is not popular enough to be cached at all.

Therefore, a video cache based on two LRU queues can operate as follows (Figure 1). First, a disk cache stores partial video files as chunks with an LRU replacement policy. To minimize ingress traffic and avoid too many redirects at the same time, it is best for the server to keep only the most popular videos (from the server's perspective). Hence, a video popularity tracker on top of the disk cache tracks the popularity of each video file as its last access time—how recently a chunk of the file was requested. The popularity tracking algorithm shares similarities with the LRU-2 algorithm [17]: If there is no previous request for the file, i.e., first time seeing a request for the file, the video fails the popularity test and is redirected. The same result will be returned if the age of the last request for the video is older than the age of the oldest chunk on disk, also known as the *cache age*.

Otherwise, the request is sent to the disk cache for serving, including the cache-fill of any missing chunks.

The disk cache and the popularity tracker can both be implemented using the same data structure, which consists of a linked list maintaining access times in sorted order, and a hash map that maps keys to list entries. These keys are video IDs in the popularity tracker, and video ID plus chunk number in the disk cache. This enables $O(1)$ lookup of access time, retrieval of cache age, removal of the oldest entries, and insertion of entries at list head. Note that insertion of a video ID with an arbitrary access times smaller than list head is not possible. Historic data that will not be useful anymore according to the cache age is regularly cleaned up.

We enable dynamic adjustment between cache fill and redirection according to α_{F2R}, resulting in what we call **xLRU Cache**, by enhancing the popularity test as follows. In Figure 1 Line 3, the popularity of a video is modeled with an approximate Inter-Arrival Time (IAT) of the requests for it, measured as $IAT_{R.v} = t_{now} - t$. Similarly, the popularity of the least popular chunk on disk, which roughly estimates the least popular video, is modeled by $IAT_0 = CacheAge$. However, if the cost of cache fill is, for instance, twice that of redirection ($\alpha_{F2R} = 2$), we expect a video to be twice as popular as the cache age, i.e., requested with a period at most half the cache age, in order to qualify for cache fill. Therefore, the redirection criteria in xLRU cache is designed as follows.

$$(t_{now} - t) \times \alpha_{F2R} > DiskCache.CacheAge(). \quad (5)$$

This condition, if true, indicates that the requested video is not considered popular enough for the given fill-vs-redirect preference (α_{F2R}) and should be redirected—in Line 3 of the pseudocode, the second term of the "or" operator.

6. Cafe Cache for Video CDNs

We develop a new algorithm for video caching that is Chunk-Aware and Fill-Efficient, hence called **Cafe Cache**[5]. In a nutshell, Cafe Cache estimates the joint cost of the current request as well as the expected future ones, when deciding to serve or redirect a request. This enables an accurate compliance of the resultant ingress and redirected traffic with the desired level defined by α_{F2R}. Moreover, in Cafe Cache, the popularity of a video is tracked based on its individual chunks. Thus, it takes into account the intra-file diversity of chunk access patterns, and it can estimate a popularity for some chunks not previously seen, as it is necessary for scoring some requested chunk ranges. In addition, Cafe Cache tracks popularity as a gradually updated inter-arrival time, which prevents unpopular videos from staying for long (polluting) the cache.

[5] The highlight of Cafe's chunk-awareness is in its computation of redirection/serving utilities for making cache admission decisions. This is not to be mistaken by chunk-level caching (i.e., not bringing in whole files unnecessarily) which is done by both Cafe and xLRU.

In the following, we first present the high level operation of Cafe Cache's request admission algorithm, followed by a description of the underlying details: Inter-arrival times, their implications for ordering the videos and chunks, and the consequent data structures employed by Cafe Cache.

Unlike xLRU Cache where popularity tracking and request admission is done at file level while disk cache is managed separately at chunk level, these two tasks are aggregated in Cafe Cache. Given request R, Cafe Cache computes the *expected cost* for serving and for redirecting the request based on the individual chunks enclosed in R, and serves/redirects the request accordingly: whichever incurs a smaller cost. Let $\mathbf{S}$ denote the set of requested chunks ($[R.c_0, R.c_1]$), $\mathbf{S}' \subseteq \mathbf{S}$ the set of requested chunks missing in the cache, and $\mathbf{S}''$ the set of old chunks to be evicted should the missing chunks be brought into the cache ($|\mathbf{S}'| = |\mathbf{S}''|$). The serving of request R incurs a twofold cost: (i) the cost of cache-filling any missing chunks ($\mathbf{S}'$), and (ii) the expected cost of redirecting/cache-filling some requests in the future—those for the chunks being evicted ($\mathbf{S}''$). Alternatively, if the requested chunks ($\mathbf{S}$) are considered not popular enough and get redirected, we incur the cost of redirecting the request right now and possibly in the future. This can be formalized as follows.

$$\begin{aligned} \mathrm{E}[\mathrm{Cost}_{\mathrm{serve}}(\mathbf{S})] &= |\mathbf{S}'| \times C_F + \\ &\quad \sum_{x \in \mathbf{S}''} \frac{T}{IAT_x} \times \min\{C_F, C_R\} \quad (6) \\ \mathrm{E}[\mathrm{Cost}_{\mathrm{redirect}}(\mathbf{S})] &= |\mathbf{S}| \times C_R + \\ &\quad \sum_{x \in \mathbf{S}'} \frac{T}{IAT_x} \times \min\{C_F, C_R\}, \quad (7) \end{aligned}$$

where IAT_x is the estimated inter-arrival time for chunk x (described shortly), and T indicates how far in the future we look into. That is, we estimate the number of requests for a chunk in the near future through an imaginary window of time during which we expect the inter-arrival times to be valid—a measure of popularity dynamics and cache churn. The cache age itself is a natural choice for the length of this window, T, which has yielded highest efficiencies in our experiments. Moreover, in Eqs. (6) and (7) we have multiplied the expected number of future requests by $\min\{C_F, C_R\}$. This is because we cannot be certain at the moment whether we will cache fill or redirect those chunks—most likely whichever incurs a lower cost, hence the *min* operator. C_F and C_R in Eqs. (6) and (7) are given based on Eq. (4).

Inter-arrival times in Cafe Cache are tracked as exponentially weighted moving average (EWMA) values. This enables Cafe to have IATs responsive to the dynamics of access patterns yet resistant to transient access changes. For each chunk x, the server tracks the previous IAT value, dt_x, and the last access time, t_x. On a new request for x at time

t, these values are updated as follows.

$$dt_x \leftarrow \gamma\,(t - t_x) + (1-\gamma)\,dt_x$$
$$t_x \leftarrow t$$

Then, the IAT of x at any time t' can be obtained as:

$$IAT_x(t') = \gamma\,(t' - t_x) + (1-\gamma)\,dt_x. \tag{8}$$

Cafe Cache needs to maintain the chunks in a data structure ordered by IAT values, similarly to the xLRU Cache, though with an added flexibility which is discussed shortly. In this data structure, a chunk gradually moves down and approaches eviction unless if a new request arrives and moves it up. Upon such request at time t, the chunk is (re-)inserted in the data structure with key $key_x(t)$. In xLRU Cache, a value of $key_x(t) = t_x$ where t_x is the last access time for x would simply satisfy the ordering requirement: Chunk x is placed lower than chunk y iff $IAT_x(t) > IAT_y(t)$ for any t. Recall that in xLRU, $IAT_x(t) = t - t_x$.

In Cafe Cache, we use a *virtual* timestamp defined as follows as the chunk's key for insertion at time t.

$$key_x(t) = t - IAT_x(t) = t - \gamma\,(t - t_x) - (1-\gamma)\,dt_x. \tag{9}$$

However, unlike xLRU where $key_x(t) = t_x$ is a time-invariant value, it is not in Cafe Cache. This means that if $key_x(t) < key_y(t)$ at insertion time t, we need to make sure the same order holds at a later lookup time t'. This is guaranteed through the following theorem.

Theorem 1. *By keying data items x and y with any arbitrary, fixed timestamp T_0 as $key_x(T_0) = T_0 - IAT_x(T_0)$ and $key_y(T_0) = T_0 - IAT_y(T_0)$, for any timestamp t we have $key_x(t) < key_y(t)$ iff $key_x(T_0) < key_y(T_0)$.*

The proof is not too complex and is therefore omitted; it is based on the linear form of Eq. (9).

Based on these keys which can order the chunks with respect to their popularities (i.e., IAT values), Cafe Cache maintains chunks in a data structure that enables the following operations: Insert a chunk with the aforementioned virtual timestamp $key_x(T_0)$; look up the IAT of a chunk; and retrieve/remove the least popular chunks—entries with smallest keys. Note that in Cafe Cache, the chunks are not always inserted with a key higher than all existing keys, unlike the case for xLRU Cache where $key_x(t) = t_x$. Rather, a chunk gradually moves up this set according to its EWMA-ed IAT value. Therefore, as a data structure that enables such insertions, we employ a binary tree maintaining the chunks in ascending order of their keys, as well as a hash map to enable fast lookup of IAT values by chunk ID. In other words, we replace the linked list in xLRU Cache with a binary tree set. This enables the desired flexibility in insertions, with an insertion/deletion time of $O(\log N)$ and lookup/retrieval of least popular chunks in $O(1)$.

Finally, we have devised a further optimization for the efficiency of Cafe Cache: We would like to have an IAT estimate for chunks that are not ever seen before, but belong to a video from which some chunks exist in the cache. Thus, we separately maintain the set of chunks cached from each file, indexed by file ID. The IAT of an unvisited chunk from video file v is estimated as the largest recorded IAT among the existing chunks of v.

7. Optimal Cache (Offline)

Even with perfect popularity prediction and caching, the efficiency of a cache in a dynamic CDN can be enhanced up to a certain point. Having an estimate of this maximum is critical for understanding and improving caching algorithms. In other words, this estimates how much of the inefficiency to blame on the caching algorithms and how much on the nature of the data. We design offline caching algorithms that assume the knowledge of the complete sequence of requests. We first formalize this problem as an Integer Programming (IP) problem and try to find the maximum possible cache efficiency through Linear Programming (LP) relaxation of the IP problem. While a computationally complex solution applicable to limited scales, this algorithm provides insights on where our greedy offline algorithm stands.

Let t ($1 \le t \le T$) where T is the size of the request sequence denote discretized time such that $t = i$ refers to when the i-th request of the sequence (R_i) arrives. Also let J denote the total number of unique video chunks present in the sequence, and j ($1 \le j \le J$) the j-th unique {video ID, chunk number}. The requests can be represented with a $J \times T$ matrix $\{m_{j,t}\}$ ($m_{j,t} \in \{0,1\}$) where $m_{j,t} = 1$ iff request R_t includes the j-th unique chunk. Similarly, we define the binary matrix $\{x_{j,t}\}$ to hold the result: $x_{j,t} = 1$ iff the j-th unique chunk should be in the cache at time t. A secondary result variable is defined as $\{a_t\}$ where $a_t \in \{0,1\}$ ($1 \le t \le T$) indicates whether R_t should be served ($a_t = 1$) or redirected ($a_t = 0$). The optimal video caching problem can be defined as follows.

$$\min \quad \sum_{j=1}^{J}\sum_{t=1}^{T} |x_{j,t} - x_{j,t-1}|/2 \times C_F + \sum_{t=1}^{T}(1-a_t) \times C_R \times |R_t|_c \tag{10a}$$

$$\text{s.t.} \quad x_{j,t} \in \{0,1\} \quad (\forall j,t) \tag{10b}$$

$$a_t \in \{0,1\} \quad (\forall t) \tag{10c}$$

$$x_{j,t} \ge a_t \quad (\forall j,t \text{ s.t. } m_{j,t} = 1) \tag{10d}$$

$$x_{j,t} \le x_{j,t-1} \quad (\forall j,t \text{ s.t. } m_{j,t} = 0) \tag{10e}$$

$$\sum_{j=1}^{J} x_{j,t} \le D_c \ (\forall t), \tag{10f}$$

where $x_{j,0}$ is defined as 0, $|R_t|_c$ denotes the size of request R in number of chunks, and D_c is the total disk size in chunks. Eq. (10a) states the total cost which is to be minimized.

The number of chunk fills is counted as $|x_{j,t} - x_{j,t-1}|/2$ since each fill comes with an eviction, both triggering a 1 in $|x_{j,t} - x_{j,t-1}|$; we can safely assume the cache is initially filled with garbage. Constraint (10d) ensures that if a request is admitted ($a_t = 1$), all its chunks are present or are brought into the cache. Constraint (10e) ensures no useless cache fill. Although this will be taken care of by the objective function, constraint (10e) helps speeding up the computations.

To get a pure linear formulation that can be fed to software libraries, we introduce new variables $y_{j,t} = |x_{j,t} - x_{j,t-1}|$ and rewrite the objective function (10a) as follows.

$$\min \sum_{j=1}^{J}\sum_{t=1}^{T} y_{j,t}/2 \times C_F + \sum_{t=1}^{T}(1-a_t)\times C_R \times |R_t|_c. \quad (11)$$

Moreover, the following new constraints are introduced that ensure the consistency of $y_{j,t}$. Note that the last constraint (12c) is only to speed up the computations.

$$y_{j,t} \geq x_{k,t} - x_{k,t-1} \quad (12a)$$
$$y_{j,t} \geq x_{k,t-1} - x_{k,t} \quad (12b)$$
$$y_{j,t} \leq 1. \quad (12c)$$

Although there exist libraries for (heuristically) optimizing this IP problem, our primary goal with the above formulation is finding a guaranteed, theoretical lower bound on the achievable cost—equivalently, an upper bound on cache efficiency (Section 4.2). We therefore solve an LP-relaxed version of the optimal caching problem by loosening constraints (10b) and (10c) to allow non-integer values in $[0, 1]$. This provides a further lower bound on cost, below which is not possible to reach by any caching algorithm. We assess the tightness of this bound empirically by comparing Optimal Cache and Psychic Cache in Section 9. Further theoretical analysis is an interesting problem left as future work in a more theory oriented study.

8. Psychic Cache (Offline)

Given the large size of the data that we use in our experiments, we need an offline cache with computation and memory requirements independent of the data scale. We design Psychic Cache which is particularly efficient in speed and memory. Psychic Cache handles a request by looking into the next N requests for every chunk and it can operate as fast as our online caches, xLRU and Cafe, on the long sequence of requests.

Psychic cache does not track any past requests for the chunks. It maintains a list $\mathbf{L}_x$ of timestamps for chunk x indicating its future requests. Also, $|\mathbf{L}_x|$ is bounded by a given N for efficiency, where $N = 10$ has proven sufficient in our experiments—no gain with higher values.

Given request R, Psychic Cache computes the expected cost of serving or redirecting the request similarly to Cafe Cache, except for estimating the cost of potential future redirections/cache-fills. Instead of using an inter-arrival time computed from past requests for a chunk (Eqs. (6) and (7)), Psychic Cache computes this value directly from the future requests themselves: it captures each request coming at time t through an inter-arrival value of $1/(t - t_{now})$, which results in a fast computable combination of how far in the future and how frequent the chunk is requested.

$$\mathrm{E}[\mathrm{Cost}_{\mathrm{serve}}(\mathbf{S})] = |\mathbf{S}'| \times C_F + \sum_{x\in\mathbf{S}''}\sum_{t\in\mathbf{L}_x} \frac{T}{t - t_{now}} \times \min\{C_F, C_R\} \quad (13)$$

$$\mathrm{E}[\mathrm{Cost}_{\mathrm{redirect}}(\mathbf{S})] = |\mathbf{S}| \times C_R + \sum_{x\in\mathbf{S}'}\sum_{t\in\mathbf{L}_x} \frac{T}{t - t_{now}} \times \min\{C_F, C_R\}, \quad (14)$$

where $\mathbf{S}$ denotes the set of requested chunks, $\mathbf{S}' \subseteq \mathbf{S}$ the missing ones, and $\mathbf{S}''$ ($|\mathbf{S}''| = |\mathbf{S}'|$) the chunks to be potentially evicted—those requested farthest in the future. Similar to Cafe Cache, $\min\{C_F, C_R\}$ is considered as the cost of a future redirection/fill (Eqs. (6) and (7)). Also similarly, the value of T, which represents for how long we rely on these estimates, is set to the cache age. Note that cache age is computed differently in Psychic Cache since there is no history of past requests: It is tracked separately as the average time that the evicted chunks have stayed in the cache.

9. Experimental Results

In this section, we present our experiments on real data from a large-scale CDN. The data includes anonymized video request logs of six selected servers around the world: One in Africa, Asia, Australia, Europe, and North and South America. These logs belong to a one month period in 2013. We replay the logs of each server to the different algorithms and measure the resultant ingress traffic, redirection ratio and the overall cache efficiency as described in Section 4.2. When reporting the average cache efficiency for an experiment, the average over the second half of the month is taken to exclude the initial cache warmup phase and ensure steadiness.

First, we evaluate the efficiency of Psychic Cache compared to Optimal Cache in a limited scale. Then, we analyze the performance of xLRU, Cafe and Psychic Caches for different setups. These setups include different server disk sizes and different fill-to-redirect configurations. As elaborated in Sections 2 and 3, we are not aware of any previous work applicable to our video caching problem that we can include in our experiments.

9.1 Psychic: An Estimator of Maximum Expected Efficiency

Recall that Psychic Cache which looks into future requests is developed to estimate how well a cache would do assuming perfect prediction of access patterns, i.e., video popularities and temporal trends. On the other hand, Optimal Cache

incorporates the entire request sequence in an Integer Programming (IP) formulation in which the achieved cost represents a theoretical minimum for any possible caching algorithm. This minimum is further lower-bounded through LP-relaxation of the IP problem, as described in Section 7.

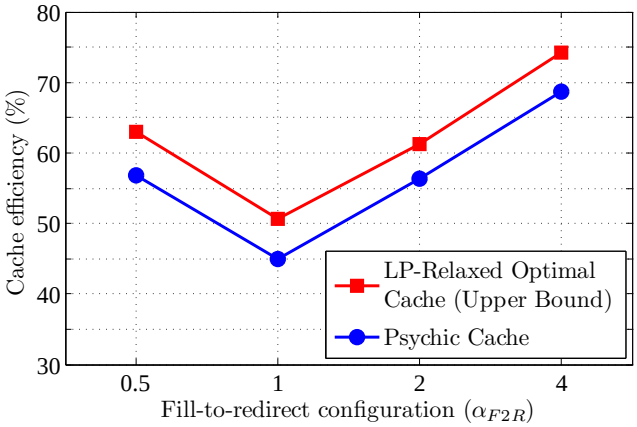
Due to memory and computational intensity of Optimal Cache, our experiments with this cache are conducted on a limited sample of the data. While this limited experiment is not as conclusive as our comprehensive experiments with the other caches, it is primarily to provide an idea of where the heuristic Psychic Cache stands as an indicator of the maximum expected efficiency in the next experiments. The data for this experiment is limited as follows. We use the traces of a two day period, which we down-sample to contain the requests for a representative subset of 100 distinct files—selected uniformly from the list of files sorted by their hit count during the two days. We also cap the file size to 20 MB for this experiment. We select the disk size such that it can store 5% of all requested chunks in the down-sampled data.

We run Optimal and Psychic Caches on this data, and measure their achieved efficiency. We do not include xLRU and Cafe Caches in this experiment since they operate according to a history of past requests, hence unable to produce reliable results in only a two day period; they are evaluated in detail shortly. Psychic and Optimal cache, on the other hand, do not require any history, and their first-hour outcome is as good as the rest.

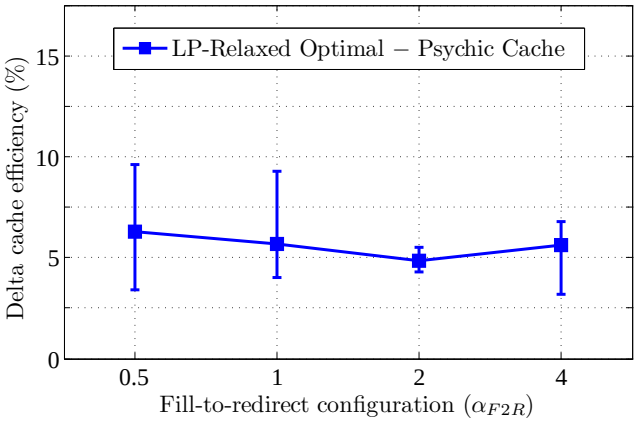
Figure 2(a) illustrates the efficiency of Psychic Cache compared to the LP-relaxed upper bound obtained by Optimal Cache. We also plot the average, minimum and maximum delta efficiency between Psychic and Optimal across all six servers through the error bars in Figure 2(b). This figure shows that the cache efficiency achieved by Psychic Cache is on average within 5–6% of the LP-relaxed bound. Note that an exact optimal solution is also within a gap of this theoretical bound as it is obtained through LP relaxation, a nonzero gap as we have observed, though theoretical analysis of the tightness of this gap is left for a future study. In the remainder of this section, we use Psychic Cache, our best known offline algorithm for actual scale, as an indicator of the highest cache efficiency expected from other (history-based) algorithms—one that is equally efficient in speed and memory with constant-size data maintained per video chunk, but with psychic prediction of future accesses even for files never seen before.

9.2 xLRU, Cafe and Psychic Performance

First, we take a close look in Figure 3 at the instantaneous performance of the caches: The redirection ratio, the ingress to egress percentage (i.e., the fraction of served traffic that incurred cache-fill) denoted in the figures by "Ingress %", and the overall cache efficiency as defined in Section 4.2. This figure corresponds to our selected server in Europe, given a disk size of 1 TB and $\alpha_{F2R} = 2$. Also, chunk size is 2 MB and $\gamma = 0.25$ (Eq. (8)) in this and other experiments. In



(a) Cache efficiencies averaged over the 6 servers.

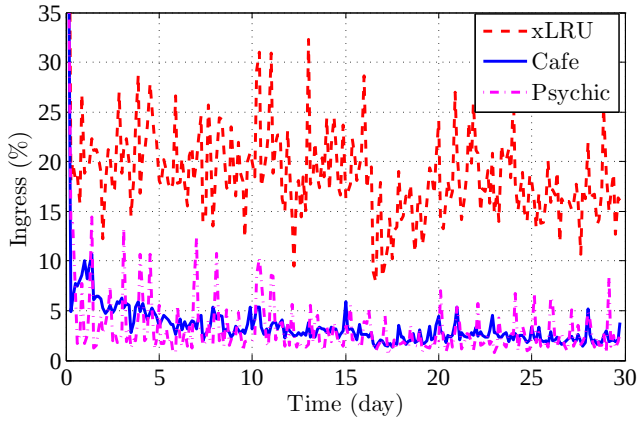


(b) Average, minimum and maximum of delta cache efficiency: LP-Relaxed Optimal minus Psychic Cache in Figure 2(a).

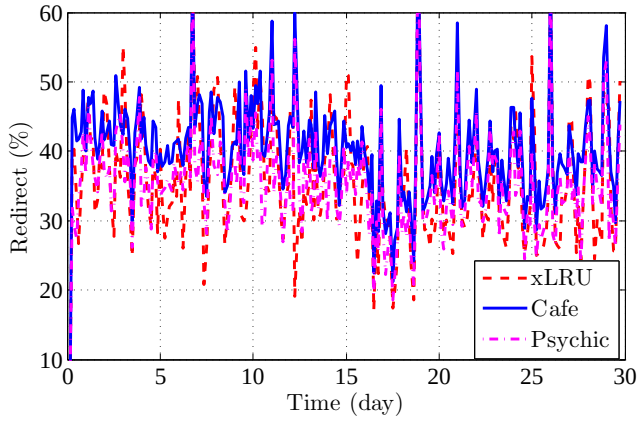
Figure 2. Performance of Psychic Cache compared to (LP-relaxed) Optimal Cache.

Figure 3, we can observe a diurnal pattern in both ingress and redirection for all caches, with their peak values occurring at busy hours. Overall, while the three caches incur comparable redirection rates with Cafe Cache being slightly higher, there is a significant drop of the incoming cache-fill traffic from xLRU to Cafe and Psychic, even though xLRU tries to admit only videos that are sufficiently more popular than the current contents (Eq. (5)). In other words, Psychic and Cafe caches perform more accurately in approving cache-fill for the right content. Thus, Cafe Cache achieves an average 10.1% increase in cache efficiency compared to xLRU, and the offline algorithm in Psychic achieves a 12.7% increase—respectively 0.101 and 0.127 in Eq. (2).

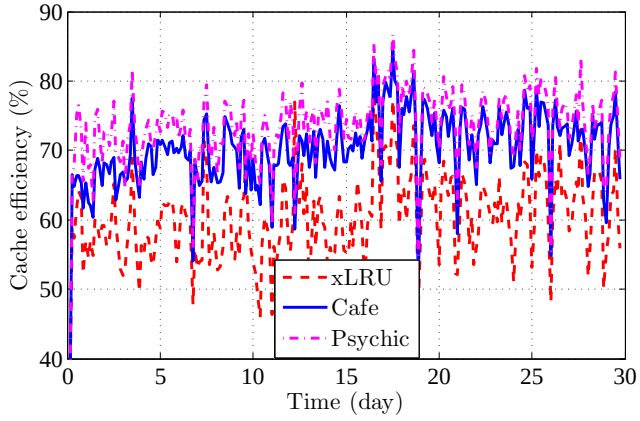
Next, we analyze the efficiency of caches for different α_{F2R} configurations. Figure 4 plots the average efficiency of the caches for this experiments on the European server given 1 TB of disk. According to the figure, when ingress is not so costly, the performance of Cafe and xLRU Caches are comparable. For example, for $\alpha_{F2R} = 1$, Cafe achieves a cache efficiency of 61% which is about 2% higher than



(a) Ingress to egress percentage.



(b) Redirected traffic.



(c) Cache efficiency.

Figure 3. Ingress, redirection, and overall cache efficiency over the 1-month period. Best viewed in color.

xLRU. On the contrary, when the server is constrained for ingress, the efficiency of Cafe Cache approaches that of Psychic, as it can reduce the ingress equally effectively and hold a well selected set of popular files, hence incurring only a small increase in redirections as depicted earlier in Figure 3. For example, for $\alpha_{F2R} = 2$, Cafe Cache achieves

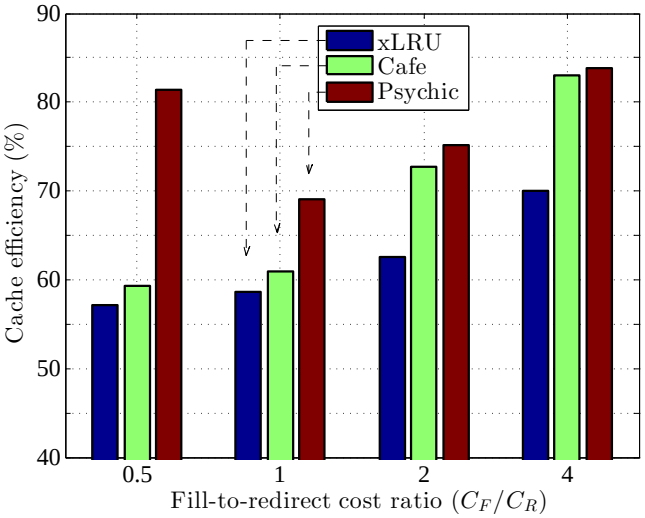


Figure 4. Efficiency of the algorithms for different ingress-to-redirect configuration. Each group of 3 bars represents xLRU, Cafe and Psychic from left to right.

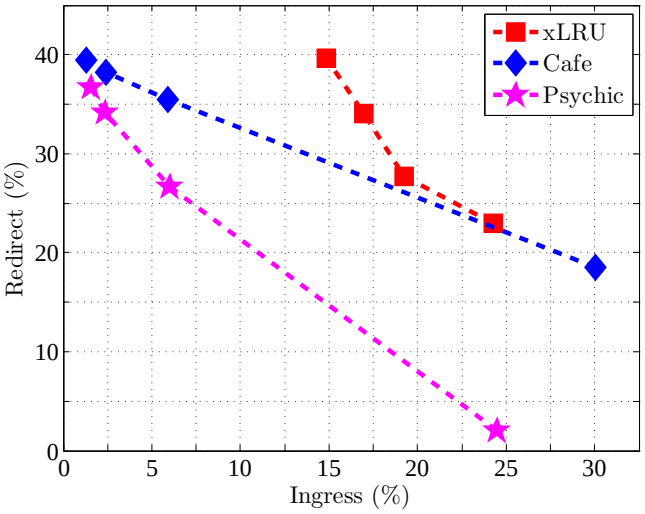


Figure 5. Different operating points of each algorithm in the tradeoff between cache fill and redirection, governed by α_{F2R}. The four operating points from left to right are obtained by setting α_{F2R} to 4, 2, 1 and 0.5, respectively.

an efficiency of 73%, close to the 75% of Psychic and 11% higher than the 62% of xLRU. From a cost-wise perspective, compared to xLRU, Cafe reduces the inefficiency (which translates into cost) from 38% to 27%, which is a relative 29% reduction. We also observe a considerable gap between xLRU/Cafe and Psychic Cache for $\alpha_{F2R} = 0.5$, which is because xLRU and Cafe will (intentionally) not bring in a file of which no previous request is ever seen, while Psychic does, based on future requests.

To further understand the tradeoff between cache fill and redirection and the way α_{F2R} can be used to define the operating point of each server in this tradeoff, we illustrate

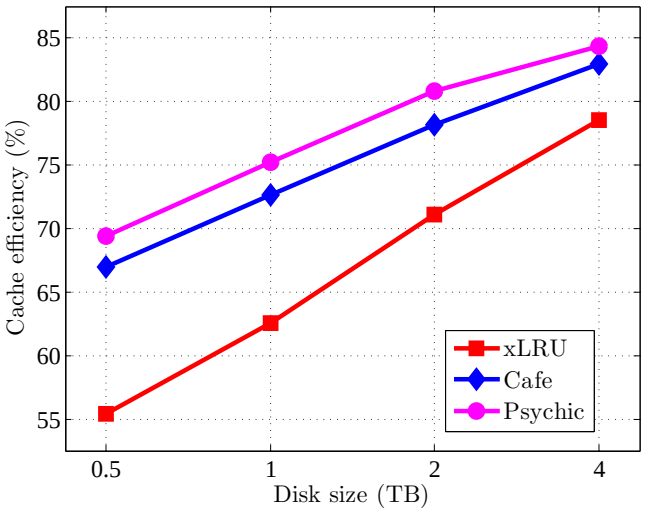


Figure 6. Efficiency of the algorithms given different disk capacities.

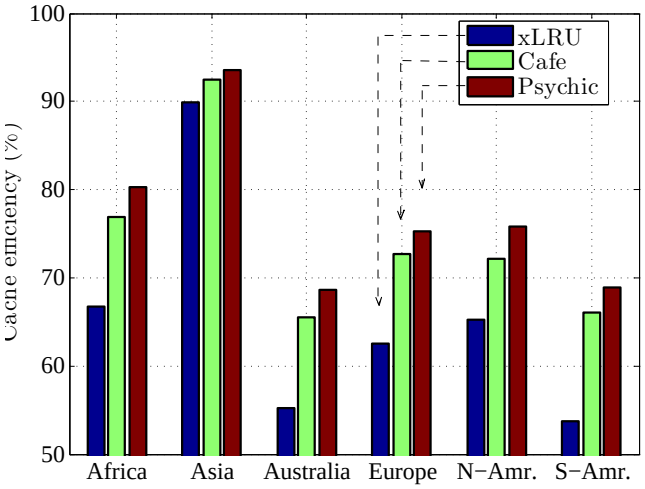


Figure 7. Efficiency of the algorithms on traces from six servers around the world. Each group of 3 bars represents xLRU, Cafe and Psychic from left to right.

these points in Figure 5. This figure shows the ingress to egress percentage for the algorithms on the horizontal axis, and the redirection ratio on the vertical axis, for different α_{F2R} values and a disk size of 1 TB on the European server. Data points from left to right correspond to $\alpha_{F2R} = 4$, 2, 1 and 0.5. The figure shows that as ingress becomes more and more costly (going from right to left), all caches tend to hold onto the data stored on their disks, and redirect more requests instead. However, the ingress percentage can only be reduced to 15% by xLRU even for $\alpha_{F2R} = 4$, while Cafe and Psychic Caches closely comply with the given costs and shrink the ingress to only a few percent. On the other hand, for cheap ingress which is represented by the rightmost data points, both xLRU and Psychic suffer from high redirections as observed and discussed in the previous experiment.

We also evaluate the performance of the caches for different disk sizes, which is plotted in Figure 6—experimented for the European server with $\alpha_{F2R} = 2$. As expected, efficiencies increase by providing more disk. However, xLRU results in an increasing inefficiency as disk size becomes limited, while Cafe maintains its small distance with the offline algorithm. This is because holding on to the right content on the disk and being accurate in approving cache fill for new content, in which Psychic and Cafe caches are better than xLRU, is more critical when the disk is more limited, i.e., small cache age. In this experiment which models an ingress-constrained server ($\alpha_{F2R} = 2$), to achieve the same efficiency xLRU requires 2 to 3 times larger disk space than Cafe Cache. In the non-ingress-constrained case ($\alpha_{F2R} = 1$), xLRU requires only up to 33% larger disk.

While the results reported so far correspond to one server, our experiments on the data from the other five servers around the world have demonstrated closely similar patterns. Figure 7 plots one of these results: Cache efficiency of the different algorithms, on a 1 TB disk with $\alpha_{F2R} = 2$. The same trend between the algorithms is observed across all servers. In this figure, the different levels of efficiency from server to server indicate different request profiles observed by these servers, i.e., request volume and diversity compared to the same 1 TB disk size given to all. For example, the selected server in Asia is serving more limited requests compared to the South American one, hence higher efficiencies in Figure 7—similar to the trend in Figure 6. Moreover, we notice a wider gap between xLRU and the other two algorithms for busier servers such as the one in South America, which confirms Psychic's effectiveness in particular for heavy-load, disk-constrained servers.

Summary. Our experiments with different server traces and setups show that xLRU and Cafe Caches achieve comparable efficiencies (Cafe up to 2% higher) for cases where ingress is not so costly: $\alpha_{F2R} \leq 1$. For $\alpha_{F2R} > 1$ which is the common case of servers with constrained ingress, Cafe performs with over 12% higher cache efficiencies, close to that of Psychic Cache that is aware of future requests. Our experiments also show that Cafe Cache performs with an increasingly higher efficiency than xLRU as the disk size becomes limited compared to the incoming request profile.

10. Conclusions

Efficient caching is crucial to the feasibility of a planet-scale video CDN. We have studied the video caching problem in such CDNs and analyzed its particular requirements in detail. We developed a simple LRU-based solution to meet these requirements (xLRU Cache), as well as a more intelligent solution specially for ingress-constrained servers (Cafe Cache). Moreover, we studied the offline caching problem assuming knowledge of future requests. We de-

signed a greedy algorithm based on this knowledge (Psychic Cache) which heuristically estimates the maximum cache efficiency we can expect from online algorithms, and an LP-relaxed optimal algorithm for limited scales (Optimal Cache). Using real data from a large-scale global CDN, we evaluated the different algorithms and showed that while xLRU can be good enough where cache ingress is not expensive, Cafe Cache performs with over 12% higher cache efficiency for the common case of servers with ingress constraints, achieving an efficiency relatively close to the offline Psychic algorithm.

There are a number of promising research directions that we would like to explore as part of our future work. A number of these problems include CDN-wide optimality with Cafe Cache, further optimality analysis for offline caching, proactive caching for spare ingress, and policy-friendly, cooperative cache management.

CDN-wide optimality with Cafe Cache. This work focuses on an optimized caching algorithm based on the given ingress-to-redirect preference α_{F2R}. As described in Section 4.1, one of the main targets of our work is to relieve ingress for constrained cache servers with proper alternative locations; see the same section for assignment of α_{F2R} values in our target CDN. Nevertheless, Cafe Cache with defined behavior through α_{F2R} (Figure 5) can as well be used as the underlying building block to adjust traffic between any group of constrained/non-constrained servers, which can be done through finer tuning of α_{F2R} for correlated servers. Capturing different inter-server correlations and global optimization of a planet-scale CDN based on Cafe Cache is an interesting problem for future research. We are currently working on CDN-wide experiments with Cafe Cache to further analyze this problem. Furthermore, dynamic adjustment of α_{F2R}, although not recommended in a wide range due to the resultant cache pollution and cache churn, can be considered in a small range through a control loop for better responsiveness to dynamics.

Optimal cache. We have not comprehensively addressed the problem of optimal offline caching in this paper. An exact optimal solution for actual scale, whether the proposed IP formulation or a customized algorithm, and/or analysis of the tightness of the LP-relaxed version can be a beneficial future study.

Proactive caching. We have primarily targeted the problem of constrained ingress capabilities which exists for many servers. For cheap/non-constrained ingress, on the other hand, we still observe a gap between the efficiency of our caches and the estimated maximum. Although the leading cause of this gap is just requests for files never seen before (Section 9), we are investigating how to take best advantage of under-utilized ingress whenever possible, such as proactive caching during early morning hours.

Policy-friendly, cooperative cache management. It is not a (readily) efficient practice for our target CDN to simply distribute the content catalog over its server locations. As discussed in Sections 2 and 3, this is due to the specific considerations for delivering the substantial video traffic, the size of the network and content corpus, and the observation that there are usually only a few server locations preferred for a given user, e.g., the CDN's remote server rack placed inside the user's ISP [12, 16], and clearly not one in another ISP network even if nearby. Scalable, cooperative/coordinated content management while taking into account such important considerations of a large CDN is a practical and beneficial future work.

Acknowledgments

The authors would like to thank Prof. Fabian E. Bustamante and the anonymous reviewers of EuroSys'14 for enabling important improvements to our paper.

This work was supported by an ORF grant under the Connected Vehicles for Smart Transportation project.

References

[1] YouTube statistics. `http://www.youtube.com/yt/press/statistics.html`.

[2] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: automated data placement for geo-distributed cloud services. In *Proc. of NSDI'10*.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proc. of PDIS'96*.

[4] S. Bakiras and T. Loukopoulos. Combining replica placement and caching techniques in content distribution networks. *Computer Communications*, 28(9):1062–1073, June 2005.

[5] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, June 1966.

[6] S. Borst, V. Gupta, and A. Walid. Distributed caching algorithms for content distribution networks. In *Proc. of IEEE Infocom'10*.

[7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. of USENIX USITS'97*.

[8] D. Karger et al. Web caching with consistent hashing. *Computer Networks*, 31(11-16):1203–1213, 1999.

[9] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *Proc. of ACM/USENIX NSDI'04*.

[10] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: An approach to building large Internet caches. In *Proc. of Workshop on Hot Topics in Operating Systems*, pages 93–98, April 1997.

[11] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube traffic characterization: a view from the edge. In *Proc. of ACM IMC'07*.

[12] Google, Inc. Google Global Cache program. `https://peering.google.com/about/ggc.html`, 2013.

[13] S. Jin and A. Bestavros. Popularity-aware greedy dual-size Web proxy caching algorithms. In *Proc. of ICDCS'00*.

[14] S. Narayana, J. Jiang, J. Rexford, and M. Chiang. To coordinate or not to coordinate? Wide-area traffic management for data centers. Technical Report, Princeton University, 2012.

[15] J. Ni and D. Tsang. Large scale cooperative caching and application-level multicast in multimedia content delivery networks. *IEEE Communications*, 43(5):98–105, May 2005.

[16] E. Nygren, R. Sitaraman, and J. Sun. The Akamai network: A platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, July 2010.

[17] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of ACM SIGMOD'93*.

[18] G. Pallis and A. Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1):101–106, January 2006.

[19] A. Pathan and R. Buyya. A taxonomy and survey of content delivery networks. Technical Report, University of Melbourne, 2007.

[20] S. Podlipnig and L. Boszormenyi. A survey of Web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, December 2003.

[21] P. Rodriguez, C. Spanner, and E. Biersack. Analysis of Web caching architectures: hierarchical and distributed caching. *IEEE/ACM Transactions on Networking*, 9(4):404–418, August 2001.

[22] A. Rousskov and D. Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22-3):2155–2168, November 1998.

[23] Sandvine Inc. Global Internet phenomena report, 2h 2013, November 2013.

[24] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of Web documents. In *Proc. of WWW'97*.

[25] X. Tang and J. Xu. QoS-aware replica placement for content distribution. *IEEE Transactions on Parallel and Distributed Systems*, 16(10):921–932, October 2005.

[26] P. Wendell, J. Jiang, M. Freedman, and J. Rexford. DONAR: decentralized server selection for cloud services. In *Proc. of ACM SIGCOMM'10*.

[27] H. Xu and B. Li. Joint request mapping and response routing for geo-distributed cloud services. In *Proc. of IEEE Infocom'13*.

[28] M. Zink, K. Su, Y. Gu, and J. Kurose. Watch global, cache local: YouTube network traffic at a campus network - measurements and implications. In *Proc. of ACM/IEEE/SPIE MMCN'08*.