# BIG Cache Abstraction for Cache Networks

Eman Ramadan, Arvind Narayanan, Zhi-Li Zhang
Department of Computer Science & Engineering
University of Minnesota, USA

Runhui Li, Gong Zhang
Huawei Future Network Theory Lab
Hong Kong

***Abstract*—In this paper, we advocate the notion of "BIG" cache as an innovative abstraction for effectively utilizing the distributed storage and processing capacities of all servers in a cache network. The "BIG" cache abstraction is proposed to partly address the problem of (cascade) thrashing in a hierarchical network of cache servers, where it has been known that cache resources at intermediate servers are poorly utilized, especially under classical cache replacement policies such as LRU. We lay out the advantages of "BIG" cache abstraction and make a strong case both from a theoretical standpoint as well as through simulation analysis. We also develop the *dCLIMB* cache algorithm to minimize the overheads of moving objects across distributed cache boundaries and present a simple yet effective heuristic for addressing the cache allotment problem in the design of "BIG" cache abstraction.**

## I. Introduction

A key premise of emerging *information-centric* network (ICN) architectures is that storage is an integral part of the network substrate. Namely, many – if not all – routers will be equipped with the capability to store/cache data objects on-the-fly. For brevity, we refer to routers with storage as *cache servers*, and a network of cache servers as a *cache network*. Given a cache network, how to effectively utilize the (*distributed*) storage capability of cache servers to help deliver information in a scalable and efficient manner is one of many major design questions in ICNs and content distribution networks (CDNs) in general.

In conventional web content delivery, (reactive) object caching takes advantage of the *temporal* locality of reference associated with a typical user browsing behavior. Caching frequent accessed objects thus reduces web delivery latency and bandwidth consumption. In large-scale streaming video delivery, server load, bandwidth usage, and energy consumption are often times more important design and operation considerations than merely latency. This is due to many factors: the size of video objects, the scale of the content delivery system and lack of temporal locality of reference (within a short time period). As no single server or even a single data center has all the processing and network bandwidth to serve all user demands, large online content service providers such as Google/YouTube and Netflix have resorted to employ one or multiple CDNs to help deliver content to users in a scalable and timely manner. These CDNs are often organized in a hierarchical structure with multiple tiers of geographically dispersed content servers [1], [2], [3], [4], where the lowest tier functions as edge servers (closest to users) for content delivery and origin servers lie at the top of the hierarchy.

Unlike existing content delivery systems which resort to (often exogenous, coarser-grained) complex mechanisms, *e.g.*, DNS anycasting & redirections, IP anycasting, or HTTP redirections [1], [2], [3], [4], for content request (re-)routing and load balancing, ICNs enable *fine-grained* and *in-network* content request routing and caching based on *content name* directly. This allows an *edge* server in ICN to directly route requests through a sequence of intermediate cache servers (see Fig. 1) towards the *origin* server of a content provider. Such ability of *caching-along-the-path* (or more generally *in-network caching*) is touted as one of many advantages of a content-centric network with built-in network caches [5].

However, due to the problem of *thrashing*, the effectiveness of "caching-along-the-path" (*i.e.*, object replication) has been questioned in some studies (see, *e.g.*, [6] and discussion in § II-B.) This has led the authors in [6] to argue that content should only be cached at edge servers. With explosive growth in various types of content or information and insatiable demands for them, it is unlikely that relying solely on the storage and processing capacities of edge cache servers alone would be sufficient to meet the scalability and performance of future ICNs. *It is therefore imperative to fully utilize the storage and processing capacities of all cache servers in a cache network.* In order to develop more effective and efficient (*distributed*) algorithms to achieve such a goal, we argue that new abstractions and better theories are called for.

In this paper, we advocate the notion of *one "BIG" cache* as an innovative abstraction for effectively utilizing the distributed storage and processing capacities of all cache servers in a cache network (see § II for model assumptions and notations): Consider a collection of objects served by an *origin* server $C_o$ of a content provider in a cache network. Given an *edge* server $C_e$ where requests for this collection of content from a (local) populace of users closer to the edge server are first received and serviced, let $C_2, \ldots, C_H$ be a sequence of intermediate cache servers along the path from the edge server to the origin server. Under the proposed "BIG" cache abstraction, each of the intermediate cache servers allots a portion of its cache capacity, $C_h^e$, $0 \leq C_h^e \leq C_h$, $h = 2, \ldots, H$, to serve user requests received by the edge server $C_e = C_1^e$ for content objects offered by the origin server $C_o$; but instead of treating them as *separate* caches, we view them *collectively* as if the cache pieces were "glued" together to form one "BIG" (*virtual*) cache of capacity $C^e = \sum_{h=1}^{H} C_h^e$ (see Fig. 3 and § III for more details.)

As we will argue in this paper, "BIG" cache abstraction affords several advantages. By treating the cache servers *collectively* as one "BIG" cache with storage capacity distributed across multiple smaller constituent cache pieces, objects may move between the boundaries of the constituent cache pieces as their access rates increase or decrease; however, eviction

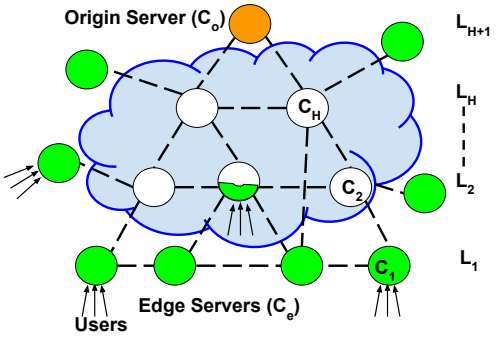


Fig. 1. Network Model

only happens when an object is removed from one cache piece *but not* placed in another. As a result, it enables us to fully utilize the allotted cache space at all (intermediate) cache servers along the path from the edge server to the origin server, yielding much higher overall hit rates. The higher overall hit rates also significantly decrease the need to fetch content from the origin server, thereby reducing its load as well as the network bandwidth demand. By minimizing or eliminating the problem of thrashing (which produces wasteful disk operations that consume a lot of energy), we also create significant energy savings at the cache servers.

"BIG" cache abstraction allows any existing cache replacement strategy such as LRU, FIFO, $k$-HIT, $k$-LRU to be applied *as a single consistent strategy* to the entire (virtual) cache. In other words, the cache replacement strategy now becomes orthogonal to the problem of content/object allocation across multiple distributed caches. From a technical standpoint, we are afforded an added benefit in that it is more amendable to conduct theoretical performance analysis of one "BIG" (virtual) cache comprising of a set of distributed caches that is governed by a single consistent cache replacement policy, as will be discussed in § III. "BIG" cache abstraction also enables us to develop more effective cache replacement algorithms that are designed through a (logically) *centralized* perspective while yet can be *implemented in a distributed manner with little overheads*. As an example, in § IV we present *dCLIMB*, with the goal to minimize object movement overheads across boundaries of distributed caches while attaining higher utilization of these distributed caches for better hit rate and overall system performance. In § V, we discuss the problem of the cache allotment in the design of "BIG" cache abstraction, illustrate how this can be formulated as an optimization problem, and outline a simple and effective heuristic to tackle this problem in practice. The evaluation of "BIG" cache abstraction is presented in § VI where we demonstrate its efficacy over existing methods in a hierarchical tree caching structure. The paper is concluded in § VII.

## II. Assumptions and Motivation

### A. Network Model

Considering a network of cache servers $C_s \in \mathcal{C} = \mathcal{E} \cup \mathcal{H}$ and a single *origin* content server $C_o$ offering a collection of content objects, $\mathcal{O} = \{O_1, \ldots, O_N\}$. Each $C_e \in \mathcal{E} \subseteq \mathcal{C}$, is an *edge* server deployed to service requests for content objects offered by $C_o$ from users of one user populace located close to $C_e$. For each edge server $C_e$, a request for an object $O_i$ from one of its user populace is first routed to $C_e$; the request is serviced directly by $C_e$ if it has $O_i$ cached in its cache; otherwise it routes the request along a sequence of intermediate cache servers, $C_h \in \mathcal{H} \subseteq \mathcal{C}$, towards the origin server $C_o$. If one of the intermediate cache servers has $O_i$ cached in its cache, the request is serviced by returning the cached object along the path back to $C_e$, which is then delivered to the user. If none of them have the object cached, the request is routed to and serviced by the origin server. We consider a general graph topology as illustrated in Fig. 1, each edge server $C_e = C_1$ is connected to the origin server through a path which can traverse any number of *intermediate* cache servers, $C_h$, $2 \leq h \leq H$, and $C_o = C_{H+1}$, where $H$ is the maximum number of hops from any edge server to the origin server. We refer to the cache server on the $h^{th}$ hop from an edge server as a layer-$(h+1)$ cache server, where $L1$ is the edge server. We do not need to impose $\mathcal{E} \cap \mathcal{H} = \emptyset$; in other words, an intermediate cache server with respect to one edge server can itself be an edge server servicing another user populace as shown in Fig. 1.

For each edge server $C_e \in \mathcal{E}$ servicing one user populace, we assume that user requests for content objects in $\mathcal{O}$ follow the standard *independent reference model* (IRM): user requests are independent of each other, and the request arrival process for object $O_i$ is governed by a Poisson process with rate $\lambda_i^e$. Let $\lambda^e = \sum_i \lambda_i^e$ be the total object request rate for $\mathcal{O}$ at edge server $C_e$. Then $a_i^e = \lambda_i^e / \lambda^e$ denotes the access probability of object $i$. In our simulation studies, we will often assume $\{a^e\}$ is Zipf-distributed. We further assume that user requests across different user populaces are independent of each other, although request arrival processes for the same object $O_i$ may be governed by Poisson processes with different request rates, and the access probabilities $\{a^e\}$'s may vary across different populaces (*e.g.*, one object may be popular among one populace, but not among another populace.)

In this paper, we will use the term *(content) object allocation* to refer to methods to decide whether to allocate cache and place objects *across a distributed set of cache servers* by utilizing their cache storage capacities. For example, "caching-along-the-path" (also known as "cache everywhere" [7]) strategy will always cache a copy of an object when it passes through a cache server; whereas the "caching at the edge server" strategy will only cache a copy of an object at an edge server – all intermediate servers will simply pass along the object without caching a copy. We will reserve the standard term *cache replacement* for methods to decide on what object to place, where to place it (*within a single cache*) and which object to evict if needed, upon receiving a request for a new or existing object. Given a content object allocation strategy and a cache replacement policy, consider a layer-$h$ cache server, $C_h$, $h = 1, \ldots, H$, and assume that it is on the path from an edge server $C_e$ to the origin server $C_o$. We abuse the notation by also using $C_h$ to denote the cache size of the cache server $C_h$. We denote the *hit probability* of object $O_i$ at cache server

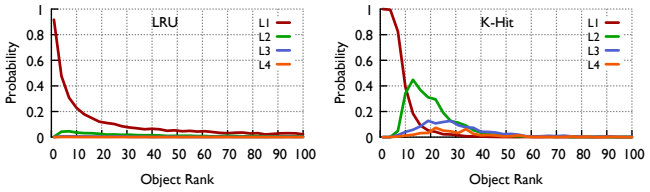



Fig. 2. Object hit probabilities with: $N = 100$, $C_h = 10$, $H = 4$, $R = 1M$, $k = 4$, $timer = 500$ $hits$, Zipf. distr. with $\alpha = 1.0$

$C_h$ by $\eta^e_{ih}$. In other words, $\eta^e_{ih}$ represents the probability that object $O_i$ is cached and serviced by $C_h$ for user requests for $O_i$ routed from edge server $C_e$. The overall hit probability of cache server $C_h$ for servicing requests routed from edge server $C_e$ is calculated by $\eta^e_h = \sum_i \eta^e_{ih}$. Summing over all edge servers, $\eta^o_h = \sum_e \eta^e_h (\leq C_h)$ measures how efficient the cache server $C_h$ is being utilized for servicing all content requests: if $\eta^e_h$ is low, then few objects are cached at $C_h$. Clearly, $1 - \sum_e \eta^e_i$ is the probability that object $O_i$ is being serviced by the origin server (and thus the overall *miss* probability of $O_i$), where $\eta^e_i = \sum_{h=1}^{H} \eta^e_{ih}$; and $\sum_e \sum_i \lambda^e_i (1 - \eta^e_i)$ is the overall rate of requests serviced by the origin server. The latter indicates the load of the origin server: higher the load at the origin server, less effective the cache network is in helping scale out the user content demands. If we also know the (average) network latency $\phi^e_h$ from a user populace serviced by an edge server $C_e$ to a cache server $C_h$, we can also derive expressions for computing the overall service latency for servicing requests for each object $O_i$ and the overall service latency for all objects.

### B. Problem of (Cascade) Thrashing

Thrashing occurs when objects are cached and then quickly evicted frequently, significantly reducing the cache efficiency as it leads to the problem of cache under-utilization. In a single cache, this is caused by a mismatch of the cache replacement policy and object access patterns. In a tandem network of cache servers, thrashing can create a cascading effect, where an evicted object causes a miss at an earlier cache, which then triggers another eviction at one or multiple subsequent cache servers. We illustrate this problem by considering the "caching-along-the-path" or *leave-copy-everywhere* (LCE) strategy together with LRU employed at each server: when a request for an object $O_i$ is routed from an edge server $C_e = C_1$ towards the origin server $C_o = C_{H+1}$, if $O_i$ is available at $C_h$, $1 \leq h \leq H + 1$, the request is serviced at $C_h$; as $O_i$ is returned from $C_h$ back to $C_1 = C_e$, each cache server $C_{h'}$, $1 \leq h' < h$, would always cache a copy of $O_i$, evicting the least recently used object if the cache is full.

Consider a simple tandem network of four cache servers, $C_e = L_1, \ldots, L_4$, with $L_5 = C_o$ the origin server offering a collection of 100 objects of unit size. The object access probabilities follow a Zipf distribution with $\alpha = 1.0$. All cache servers have a cache of size $C_i = 10$, $i = 1, \ldots, 4$. The left plot in Fig. 2 shows the object hit probability at the edge server ($L_1 = C_e$) as well as the three intermediate cache servers with a simulation run of $1M$ requests. We see that the object hit probability at the edge server ($L_1$) drops rapidly, with the hit probability for the $5^{th}$ most popular object only $40\%$; whereas the object hit probabilities at the three intermediate servers are uniformly low for all objects, with the highest values barely reaching $5\%$. In other words, the intermediate cache servers do not help improve the object hit performance at all! Changing LRU to a more sophisticated cache replacement policy does not significantly improve the overall object hit performance of the intermediate cache servers. The right plot in Fig. 2 shows the object hit probabilities at the edge server and the three intermediate cache servers where the so-called *k-HIT* cache replacement strategy is employed with $k = 4$: an object is only placed in the cache if it has accumulated at least $k$ hits during a pre-specified time period at each server; if the cache is full, the object with the least hits during the time period is evicted. We see that while the object hit probabilities at the intermediate cache servers improve over LRU, their effectiveness in enhancing object hit probabilities is still limited; in particular, as $h$ increases, the role of the layer-$h$ cache server $L_h$ diminishes drastically. Only when $k$ is considerably large would intermediate cache servers have a marked impact on the object hit probabilities – increasing $k$ on the other hand reduces the ability of the cache network to adapt to changing access patterns.

The poor performance of a tandem cache network under "caching-along-the-path" is due to the problem of thrashing, in particular when a cache replacement policy such as LRU is employed at each cache: objects cached at the second cache server after the edge server are often quickly evicted, leading to poor hit rates; the problem exacerbates rapidly at higher-tiered cache servers, from the second to the third, fourth, and so forth, where these intermediate cache servers are severely *under-utilized* with low hit rates. There are several reasons behind this poor performance. Having a sequence of cache servers *operating independently* in accordance with their *own* cache replacement policies create two major issues: first, the request arrival processes to the cache servers are no longer independent – caching a more popular object at a previous cache server would reduce its access rate at subsequent cache servers, making it less popular (*i.e.*, causing changes in the request access pattern at higher layers after filtering popular content). This affects the ability of subsequent cache servers to properly estimate the "popularity" of objects. Second, when an object is evicted from, say, an edge or intermediate cache server, it is simply discarded – this is not only a wasteful operation; more importantly, past (access) information about this object is simply lost. The compounded effects of these two factors create a cascade of thrashings at the intermediate servers, resulting in *severe under-utilization* of cache resources. The poor hit rates at the cache servers due to thrashing also create another major problem that is of utmost importance in practice: missed content requests must be serviced by the origin content server; this causes the origin server to be *overloaded*; it also requires more network capacity at the origin server! As noted in the introduction, besides reducing the overall latency of serving user content requests, another major reason that today's large-scale online content providers employ one or more CDNs to

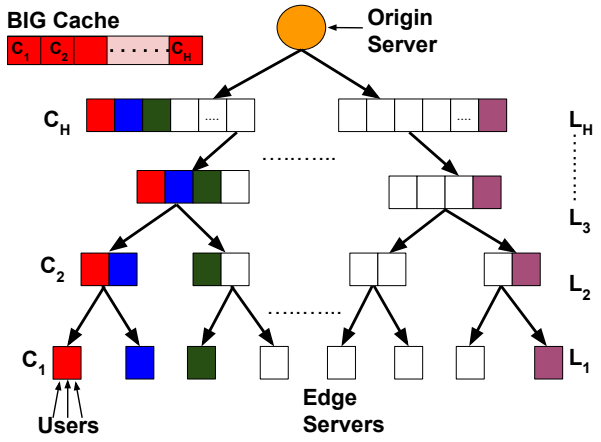



Fig. 3. "BIG" Cache Abstraction

help content delivery is to reduce the processing and network bandwidth demands on origin content servers (which cannot meet all user demands) by *scaling out* through *geographically dispersed* cache servers deployed by CDNs. In addition, the "wasteful" operations (*e.g.*, read or write operations to the local disks) at cache servers due to thrashing also produces higher energy consumption that is of an increasing concern in current and future content delivery networks [8].

**Related Work.** Caching has been a well studied subject with a rich literature. ICNs have attracted a flurry of new studies on caching strategies, in particular, in the context of CCN/NDN [5]. We refer the reader to the excellent paper [7] and the references therein for a survey and summary of various caching policies and their analyses. We also do not claim a novel contribution regarding the poor performance of cache servers in a tandem network. Similar observations have been suggested based on anecdotal evidences in [9], [10]. Partly due to these observations, the authors in [6] argue for "caching at edge servers" only, and propose an "incrementally deployable" ICN architecture. Object allocation strategies which slightly improve upon "caching at edge servers" such as *leave-copy-probabilistically* (LCP) and *leave-copy-down* (LCD) have been proposed and analyzed in [7]. In LCP, if $O_i$ is available at $C_h$, $1 \leq h \leq H+1$; as $O_i$ is returned from $C_h$ back to $C_1 = C_e$, each cache server $C_{h'}$, $1 \leq h' < h$ would cache a copy of $O_i$ with a probability $q$, while in LCD only the cache server $C_{h-1}$ caches $O_i$. However, these alternative object allocation strategies do not fully take advantage of the resources available at the entire cache network, and the thrashing problem is still applicable as at least one more copy is still cached on the way back to the end-user. As such, the overall scalability of the cache network will be limited by the resources available at the edge servers (and only a portion of other cache servers.) Thus, in this paper, we also provide analysis through simulation demonstrating the in-efficiency of these object allocation strategies compared to "BIG" cache abstraction as shown by the results in § III, VI. To the best of our knowledge, we are the first to propose the notion of "BIG" cache abstraction and demonstrate its ability in fully utilizing cache resources at intermediate cache servers.

## III. Case For "BIG" Cache Abstraction

**"BIG" Cache Abstraction.** Given an edge server $C_e$ which serves content requests from its user populace for objects in $\mathcal{O}$ offered by the origin server $C_o$, let $C_2, \ldots, C_H$ be a sequence of intermediate cache servers along the path where content requests are routed from $C_e$ to $C_o$. The cache servers, $C_1 = C_e, C_2, \ldots, C_H$ together with $C_{H+1} = C_o$ form a *tandem* cache (sub-)network, a branch from an edge server to the origin server in the hierarchical cache network with a tree structure shown in Fig. 1. The object request arrival rates at $C_e$ are given by $\lambda_i^e$'s for $O_i$'s in $\mathcal{C}$. In the conventional paradigm, each cache server operates its own *separate* cache *independent of others*, serving content requests as they arrive and makes caching decisions on its own. As illustrated in the previous section using simple examples, this conventional paradigm creates complicated interactions among the cache servers along the path, producing the problem of *cascade thrashing*. As a result, the cache performance is poor; in particular, cache resources at higher layer cache servers (along the path towards the origin server) are severely under-utilized. To circumvent these issues, we propose the notion of "BIG" cache as an innovative abstraction for effectively utilizing the distributed storage and processing capacities of all cache servers in a cache network. This abstraction is formally defined below.

Consider a sequence of cache servers, $C_1 = C_e, C_2, \ldots, C_H$, where the intermediate cache servers, $C_h$, $2 \leq h \leq H$, may also serve content requests from other edge servers, $C_{e'} \neq C_e$ – namely, they are on the path from another edge server $C_{e'}$ towards the origin content server $C_o = C_{H+1}$ (see Fig. 1). Under the proposed *"BIG" cache* abstraction, each intermediate cache server would (*logically*) allot a portion of its cache capacity, $C_h^e$, $0 \leq C_h^e \leq C_h$, $h = 2, \ldots, H$, to service content requests routed from the edge server $C_e = C_1^e$ for content objects offered by the origin server $C_o$. Instead of treating the cache pieces as *separate* caches, we view them *collectively* as if these cache pieces were "glued" together to form *one "BIG" (*virtual*) cache* with capacity $C^e = \sum_{h=1}^{H} C_h^e$. This is schematically depicted in Fig. 3, where several "BIG" virtual caches are shown for different edge servers highlighted in colors. With this abstraction, objects are placed in the "BIG" virtual cache distributed across $H$ cache pieces *under one consistent cache replacement policy* (as opposed to $H$ independently operated policies): objects may be moved or swapped across the boundaries of these distributed cache pieces; an object is only considered *evicted* from the "BIG" cache if and only if it is evicted from one of the distributed cache pieces and *not* placed in another one. Perhaps more importantly, we maintain the object access patterns *globally* for each "BIG" (virtual) cache, and make cache replacement decisions *consistently* across the distributed cache pieces. In other words, we emulate the same operations and behavior of a cache replacement policy that is applied to a set of distributed caches *as if it were applied to a single "BIG" cache*. We will demonstrate the key advantages of the proposed "BIG" cache abstraction over conventional paradigm where

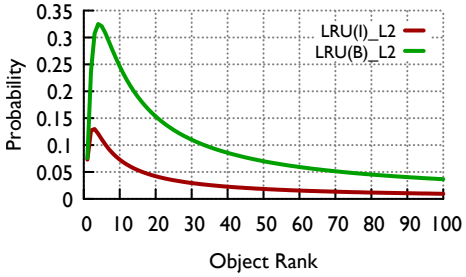



Fig. 4. Object hit probabilities at the second-layer cache $C_2$ under LRU(I) and LRU(B): $N = 100$ and $C = 10$

each cache server operates independently with its own cache replacement policy. Clearly, "BIG" cache abstraction also poses important new research challenges. For example, the emulation creates additional overheads, where, for example, moving or swapping objects across boundaries of cache pieces is more costly than simply moving or swapping them within one physical cache piece. Hence, it is desirable to minimize these swapping overheads. In § IV, we describe one caching mechanism for operating one "BIG" (virtual) cache across a distributed set of cache pieces. Another major problem is how to (logically) partition and allot the cache resources at intermediate servers among multiple edge servers so as to maximize the overall cache network performance. We discuss and tackle this problem in § V.

**Theoretical Performance Analysis.** "BIG" cache abstraction allows any existing cache replacement strategies such as LRU, $k$-HIT, FIFO, RAMDOM, $k$-LRU, etc. [7] to be applied *as a single consistent strategy* to the entire "BIG" (virtual) cache. We are also afforded an added benefit in that it is more amendable to conduct theoretical performance analysis, as will be illustrated shortly. Consider a cache of size $C$ and a request arrival process $\lambda = \{\lambda_i\}$, where $\lambda_i$ is the request rate for object $O_i \in \mathcal{O}$. Given a cache replacement policy $P$, let $\eta_i^P(C, \lambda)$ and $\rho_i^P(C, \lambda) = \lambda_i \eta_i^P(C, \lambda)$ be the hit probability and hit rate of object $O_i$ as a function of $C$ and $\lambda$ under $P$. Then $\eta^P(C, \lambda) = \sum_i \eta_i^P(C, \lambda)$ and $\rho^P(C, \lambda) = \sum_i \lambda_i \eta_i^P(C, \lambda)$ represent the overall cache hit probability and rate under $P$. The authors in paper [7] provide a nice summary of various cache replacement policies with approximate theoretical analysis of their performance under IRM and renewal arrival processes. We can directly apply these theoretical analyses to our "BIG" cache.

Given a "BIG" cache of size $C^e = \sum_{h=1}^{H} C_h^e$, the overall hit probability and hit rate of each object as well as the entire "BIG" cache under a cache replacement policy $P$ are simply given by $\eta_i^P(C^e, \lambda)$, $\rho_i^P(C^e, \lambda)$, $\eta^P(C^e, \lambda)$ and $\rho^P(C^e, \lambda)$. For $h = 1, \ldots, H$, define $C_{\leq h}^e = \sum_{l=1}^{h} C_l^e$. Hence $C_{\leq h}^e$ is the total cache size of the first $h$ layers staring from the edge server. We can provide a theoretical estimate of the cache performance or efficiency of the cache piece at the $h^{th}$ layer, $C_h^e$, directly. For example, the (additional) hit probability $\eta_{ih}^e$ of object $O_i$ contributed by the cache piece $C_h^e$ is given by $\eta_{ih}^e = \eta_i^P(C_{\leq h}^e) - \eta_i^P(C_{\leq h-1}^e)$ (for clarity, here we drop the parameter $\lambda$. Similarly, we can theoretically estimate the overall object hit rate contributed by the cache piece $C_h^e$: $\rho_h^e = \rho^P(C_{\leq h}^e) - \rho^P(C_{\leq h-1}^e)$. These results can further be leveraged to derive, for example, estimates for the overall service latency and other performance metrics. Due to space limitation, we do not elaborate further. The amenability of performance analysis for "BIG" cache abstraction is in stark contrast to the conventional approach: As each cache server operates independently, they create complex interactions between the cache replacement policy at one layer and the object request arrival process at the next layer. While Markov-modulated processes are employed to analyze a tandem network of independently operated cache servers in [7], such analysis quickly become unwieldy when $H$ grows larger.

As a concrete example, consider a tandem network with two cache servers, $C_e = C_1$ and $C_2$, each with a cache size $C_i = c$ and employing LRU for cache replacement. Using Che's approximation [11] and the theoretical analysis in [7], we can estimate the hit probability $\eta_{ih}^{(I)}$ of each object $i$ at cache $C_h$ as follows: $\eta_{i1}^{(I)} = 1 - e^{-\lambda_{i1} T_c^1}$ and $\eta_{i2}^{(I)} \approx 1 - e^{-\lambda_{i2}(T_c^2 - T_c^1)}$, where $T_c^h$ is the characteristic time of cache $C_h$ of size $c$, and $\lambda_{ih}$ is the arrival rate of requests for object $i$ at cache $C_h$, in particular $\lambda_{i1} = \lambda_i$ and $\lambda_{i2} = \lambda_{i1}(1 - \eta_{i1})$ (see [7] for details). In contrast, applying LRU to one "BIG" cache of size $C_1 + C_2 = 2c$, the overall hit probability of object $O_i$ is given by $\eta_i^{(B)} = 1 - e^{-\lambda_i T_{2c}}$. The hit probability of $O_i$ contributed by $C_2$ under one "BIG" cache is given by $\eta_{i2}^{(B)} = \eta_i^{(B)} - \eta_{i1}^{(B)} = e^{-\lambda_i T_c^1} - e^{-\lambda_i T_{2c}}$. We can mathematically prove that $\eta_{i2}^{(B)} > \eta_{i2}^{(I)}$; due to space limitation, we omit the proof here. Numerical results using these formulas for the object hit probabilities at the second layer cache $C_2$ are plotted in Fig. 4 with $N = 100$, $c = 10$ and the same object request arrival processes as before, where LRU(I) denotes that two caches operate independently using LRU and LRU(B) denotes that the two caches form one "BIG" cache operating under a single consistent LRU policy. (In the above and throughout the remainder of the paper, we will use the superscripts or suffices $(I)$ vs. $(B)$ to denote cache servers operating independently vs. as one "BIG" cache.) We see that the cache resource at the second cache server $C_2$ is much better utilized for all objects under the "BIG" cache abstraction. Therefore, two servers in a tandem network operating as one "BIG" cache significantly improve the overall efficiency of the cache network.

**Performance Comparisons via Simulations.** We now conduct simulations to further demonstrate the benefits of "BIG" cache abstraction over the conventional approach when $H$ increases. Consider a tandem network with four cache servers $C_h$, $1 \leq h \leq H$, each with cache size $C = 10$, serving a total of 100 objects with the same request arrival processes as before. With LRU applied independently to the four cache servers vs. as one "BIG" cache formed by the four servers. We also applied different object allocation strategies when LRU is applied independently, which specifies how objects are cached on the backward path while being sent from the cache where they are found to users. We used the following object allocation strategies: 1) *caching at edge servers only (OE)* only cache a copy of an object at an edge server, 2) *leave-copy-*

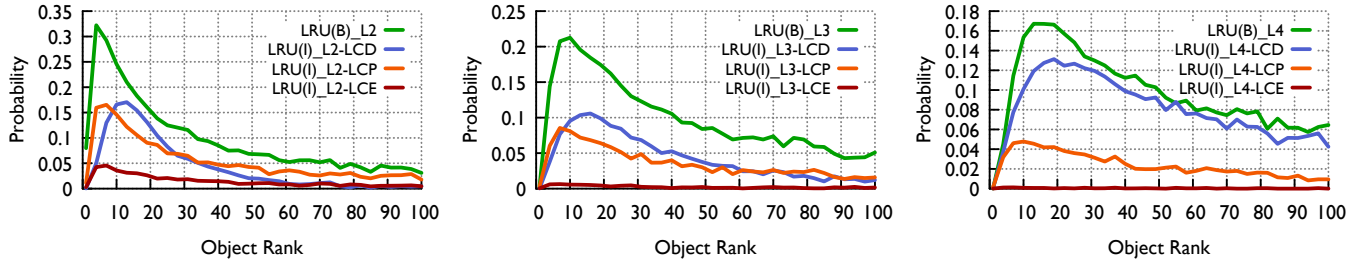



Fig. 5. LRU - Object hit probabilities, $N = 100$, $C = 10$, $H = 4$, $R = 1M$, $q = 0.5$ for LCP

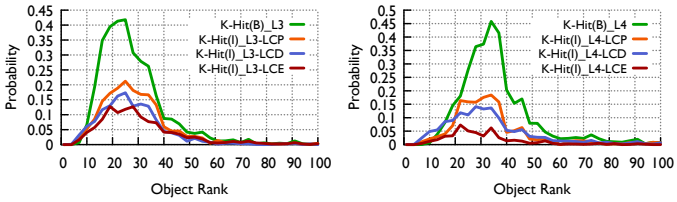



Fig. 6. $k$-HIT - Object hit probabilities: $N = 100$, $C = 10$, $H = 4$, $R = 1M$, $k = 4$, $timer = 500$ $hits$, $q = 0.5$ for LCP

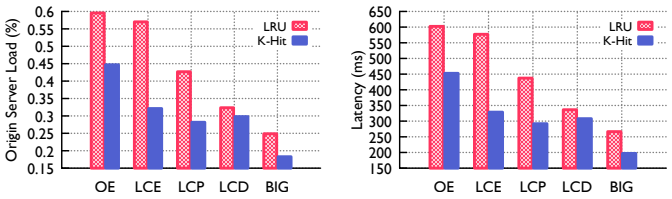



Fig. 7. LRU and $k$-HIT - Origin Server Load and Overall Latency

*everywhere (LCE)* always cache a copy of an object when it passes through a cache server, 3) *leave-copy-probabilistically (LCP)* cache a copy of an object with probability $q$ when it passes through a cache server, and 4) *leave-copy-down (LCD)* cache a copy of an object only at the cache preceding the one in which it is found (if any) (*i.e.*, if the object is found at $C_h$, it will be cached at $C_{h-1}$). In our simulation, we use $q = 0.5$ for LCP. This simple simulation is intended to make the case for "BIG" cache abstraction. More extensive simulation can be found in § VI.

Fig. 5 shows the object hit probabilities at the three intermediate servers with $R = 1M$ object requests. We see that as $H$ increases from 2 to 4, the cache performance of LRU(I)-LCE does not improve at all: cache resources at the third and fourth layer caches, $C_3$ and $C_4$, are practically not utilized at all; worse, comparing the results in Fig. 4 and Fig. 5, the object hit probabilities at the second-layer cache $C_2$ got worse with $H = 4$. This is due to the problem of cascade thrashing we alluded to earlier. In contrast, under "BIG" cache abstraction, the cache performance at $C_2$ is not much affected as we increase $H$, while the cache resources at the additional intermediate servers, $C_3$ and $C_4$ are effectively utilized to improve the object hit probabilities. Using other object allocation strategies such as LCP, LCD only help improve the performance of LRU very slightly, but still "BIG" cache abstraction provided better performance, even though LRU(I)-LCD has a similar behavior for $L4$, but this is not the case in the other layers and also for $k$-HIT as shown in the next figure. The overall hit probability for each object is shown in Fig. 8. The overall sum of object hit probabilities at each layer are shown in Fig. 9, where we recall $C = 10$: we see that under "BIG" cache abstraction, the cache efficiency at each layer approaches 100%; this is in contrast to LRU(I)-LCE where only the edge server achieves a nearly 100% cache efficiency, whereas the second server barely achieves 15% (=1.5/10) efficiency, and the cache efficiency at the third and fourth cache servers is nearly 0. The intermediate caches are still under-utilized using other object allocation strategies (LCP and LCD) compared to "BIG" cache abstraction. With a more sophisticated cache replacement such as $k$-HIT, the cache performance of $k$-HIT(I) is better than LRU(I), but similar observations still hold. With the same model parameters, Fig. 6 shows the object hit probabilities at the intermediate servers of the $3^{rd}$ and $4^{th}$ layers for $k$-HIT(I) vs. $k$-HIT(B) for $k = 4$. We see that the cache efficiency at the third and fourth server slightly improves under $k$-HIT(I) using different object allocation strategies, but the cache resources at these servers are still *under-utilized*. In contrast, $k$-HIT(B) increases the hit probabilities of the popular objects with at least 50% increase. Moreover, $k$-HIT(B) attains nearly 100% cache efficiency at all layers as shown in Fig. 10, while the efficiency decreases by approaching the origin server layer using different alternatives of $k$-HIT(I) ($k$-HIT(I)-LCE, $k$-HIT(I)-LCD, and $k$-HIT(I)-LCP). Thus, we can deduce that "BIG" cache abstraction results in better performance than the different object replication techniques. Finally, "BIG" cache abstraction results in the minimum origin server load and overall service latency ($\phi_1 = 1$, $\phi_2 = 10$, $\phi_3 = 50$, $\phi_4 = 100$, & $\phi_5 = 1000$) for both LRU and $k$-HIT as shown in Fig. 7.

**General Topologies.** Our proposed "BIG" cache abstraction can be applied to any general topology. Assuming a graph structure for the cache network, for each edge server, we need to find a line of cache servers on the way to the origin server and then apply "BIG" cache to these set of servers. This leads to some of the intermediate cache servers being shared among different edge servers and be part of their "BIG" cache, which raises an interesting research problem about how the cache space should be partitioned among edge servers to improve the overall performance. The cache partitioning problem is not the main focus of the this paper, however in § V, we highlight how it can be formalized as an optimization problem and provide a simple heuristic approach to address it. It is important to clarify that partitioning the available cache space does not mean that multiple copies of the same object

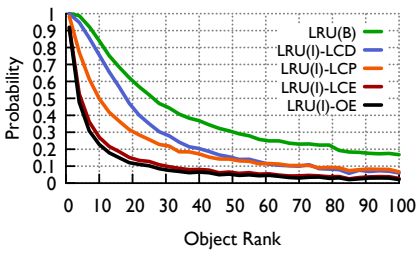



Fig. 8. Overall object hit probability

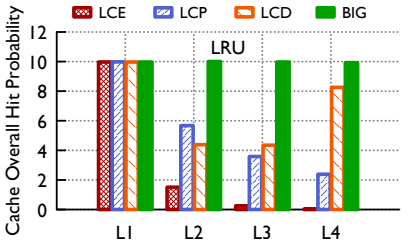



Fig. 9. Overall cache hit probability

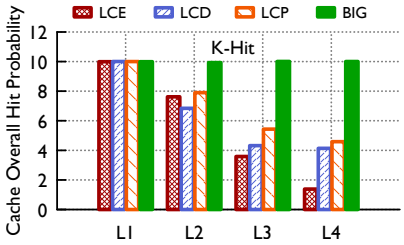
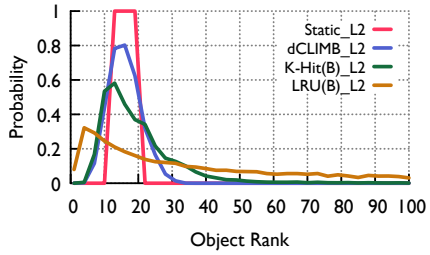
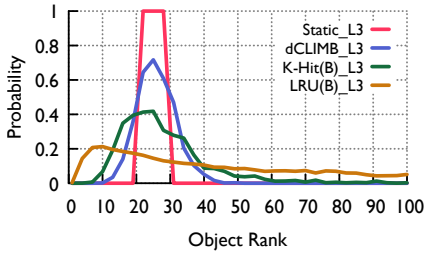



Fig. 10. Overall cache hit probability



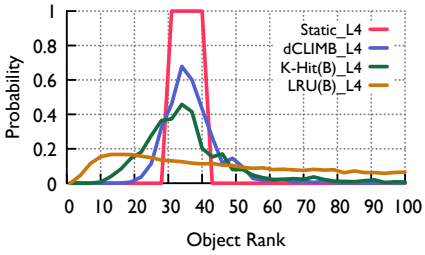




Fig. 11. Object hit probabilities, $N = 100$, $C = 10$, $H = 4$ , $R = 1M$, $k = 4$, $timer = 500$ $hits$

are cached when multiple edge servers share the same cache server. Only one physical copy is cached among different edge servers and this leaves more space to cache other objects from higher layers which helps improve the hit probability and also user latency. Cache partitioning is only virtual, but at the same time we keep track of the statistics of each big cache separately and objects in each big cache are managed independently of other big caches sharing the same physical cache server. This independent management of objects does not impact the performance of the overall system as shown in the results of § VI. "BIG" cache abstraction promotes cooperating cache servers vertically (from edge server to the origin server) to decide which objects to cache at each layer to improve the performance than cooperating requests from different edge servers at intermediate layers which then act independently from each other. Moreover, intermediate servers can also be edge cache servers for other user populace, in such cases there are requests coming directly from users and other requests to this intermediate server from other edge servers. These requests may vary in the objects of interest and also the number of requests. This variation is handled by the cache allotment which considers all possible streams of requests arriving at the current node and then partition the cache using a utility function which could be related to the hit rate.

In summary, "BIG" cache abstraction decouples the cache replacement polices from the problem of content/object allocation across multiple distributed caches. It allows any existing cache replacement strategies such as LRU, FIFO, $k$-HIT, $k$-LRU to be applied *as a single consistent strategy* to the entire (virtual) cache. One can judiciously choose a cache replacement policy that best matches the content object cache patterns, *e.g.*, LRU when there is significant *temporal locality of reference* in the access patterns, or static caching [12], [7] when object access patterns are known *a priori* and are Zipf-distributed. *Independently of* what cache replacement policy is employed, "BIG" cache abstraction enables us to efficiently utilize the additional cache resources available at all cache servers along the path to the origin server as there is only one copy stored (vertically) on the path from the edge server to the origin server and also one copy is stored (horizontally) at each intermediate layer irrespective of the big caches sharing this node; it also enables to leverage the server processing capacities and network bandwidth made available by these additional cache servers to fully *scale out* a cache network to meet growing user demands for information and content. The much higher overall hit rates under "BIG" cache abstraction also significantly decrease the need to fetch content from the origin server, thereby reducing its load as well as the network bandwidth demand. By minimizing or eliminating the problem of thrashing, we also create significant energy savings at cache servers. Finally, as discussed above the performance of cache replacement policies can be estimated by applying them to the big cache with the summation of the allocated cache sizes from each layer, the latency can then be calculated based on the hit ratio at each layer.

## IV. Caching Strategy For "BIG" Cache

When an edge cache server receives a request for an object, a copy is returned to the user if it is cached; otherwise the request is forwarded to other cache layers along the path to the origin server. When a copy is found, the object is returned to the user along the reverse path. However, this process does not specify how the object should be cached in these cache layers along the way back to the user. As shown in the previous section, "BIG" cache abstraction helps improve the object hit probability, the origin server load and also the overall service latency for all objects. While any existing cache replacement policy can be used in conjunction with "BIG" cache abstraction, some cache replacement policies may incur more overheads than others when emulating them across a distributed set of caches. In this section, we compare the performance of some of these cache replacement policies and the overhead associated with them.

For instance, applying LRU to "BIG" cache, a request for an object not in the edge cache results in inserting it at the head of the queue (at the edge server), which would trigger a series of object movements across boundaries of the distributed caches. Since similar behavior applies for other cache replacement policies, it is therefore desirable to minimize object movements across the boundaries of distributed caches. This motivates us to design a cache replacement mechanism for "BIG" cache abstraction. The key idea is that we view all cache pieces stacked together with $C_1^e = C_e$ on the top and $C_H^e$ at the bottom, and the entire "BIG" cache is organized in a single (global) *stack* data structure (linked together by individual stacks maintained at each cache server): intuitively, the goal is to have the object with most frequent and recent access is at the top of the stack, and remaining objects are organized in the order of their frequency and recent access, from high to low (see below for a more formal description). At any time a request for an object already cached in the "BIG" cache causes it to swap its position with the object currently ahead of it. If a request for a new object currently not cached anywhere in the "BIG" cache, it is appended at the end of the stack, *i.e.*, at the end of the stack maintained at the last server, $C_H^e$. We refer to this cache mechanism as ***dCLIMB***, which is a generalization of the *CLIMB* algorithm first studied in [13] for a single cache after applying it to a set of distributed caches acting as one "BIG" cache. Under *dCLIMB*, each object access triggers at most one object swap operation. If the accessed object is currently at the head of the $C_h^e$ cache, it would cause a swap operation across the cache boundaries; otherwise, the swap operation is performed locally within the $C_h^e$ cache.

The *dCLIMB* algorithm is formally described as following: assuming each cache $C_h^e$ is organized as a stack of $C$ slots (*i.e.*, cache size). Each slot $c$, where $1 \leq c \leq C$, is able to hold one object $O_i$. When a request for object $O_i$ is received at $C_h^e$, assuming $O_i$ is cached at slot $c$ at any layer $C_h^e$, $1 \leq h \leq H$, there are three possible cases: i) if $c \neq 1$, then $O_i$ is swapped with $O_j$ at slot $c-1$ in $C_h^e$, ii) if $c = 1$ and $h = 1$, nothing happens, and iii) if $c = 1$ and $h = 2, 3, \ldots H$, $O_i$ is swapped with $O_j$ at slot $C$ in $C_{h-1}^e$. In addition to the main cache at cache server $L_H$, *dCLIMB* implements a temporary cache in order to avoid caching every requested object at the main cache $C_H^e$, which leads to the eviction of more popular objects. When an object is requested, only its meta-data is inserted in this temporary cache, and only objects at the head of the temporary cache are moved to the actual cache of $L_H$.

Fig. 11 shows the performance of LRU, $k$-HIT, and *dCLIMB* applied to one "BIG" cache compared to static caching, we can notice that *dCLIMB* actually utilizes the intermediate layers of the cache hierarchy with a performance close to static caching, which we consider as the baseline. Without the knowledge of user access patterns a priori, *dCLIMB* leads to caching the most popular content objects at edge servers, and less popular objects in higher layers in the hierarchy.

The advantages of *dCLIMB* are multi-fold: It is a *self-adaptive* request-driven strategy which decides automatically how objects should be placed along different cache layers without the knowledge of user access patterns *a priori*. It attains better (higher hit ratio) than the other classical cache replacement policies by embedding the observed user access pattern for each object in its position in the stack. Moreover, *dCLIMB* is capable of dynamically adapting automatically to changes in user access patterns and flash crowds. For example, when an object receives a sudden burst of requests, *dCLIMB* would gradually move it to a lower cache layer closer to users, thus reducing service latency. However, this process may take some time, depending on how large the burst of requests is.

**Complexity.** Finally, *dCLIMB* incurs minimal object movement overheads compared to other cache replacement policies. For example, for each request *dCLIMB* needs to update the indexes of only two objects; only if the object is at the head of its cache layer, it needs to be swapped with the previous layer. In contrast, LRU needs to maintain a queue of requested objects; each time an object is accessed, move or insert it at the head of the queue, which requires more operations; LFU and $k$-HIT need to keep track of access counts (and sometimes timers), and require queue operations for object insertions or movements. The number of insertions and evictions operations for all cache layers approximately were: LRU: 2.7M, $K$-HIT: 222K, *dCLIMB*: 109K. We can notice that *dCLIMB* nearly requires half the operations required by $k$-HIT and at the same time provides the closest performance to static caching and increases the object hit probabilities at different caches without the need to keep track of counters and timers. Thus, *dCLIMB* minimizes the problem of thrashing (which produces wasteful disk operations that consume a lot of energy), and creates significant energy savings at the cache servers. Therefore, *dCLIMB* presents a distributed, coordinated and collective cache replacement policy with minimal complexity and without the need for a global central mechanism.

## V. "BIG" Cache Allotment

One major new research problem in the design of "BIG" cache abstraction is how to (*logically*) partition the cache resources at intermediate cache servers that are shared by multiple edge servers and allot cache pieces to form one "BIG" (virtual) cache with respect to each edge server $C_e$ so as to maximize the overall cache network performance. Addressing this problem thoroughly warrants a separate paper. For completeness, in the following we will briefly describe how this problem can be formulated as an optimization problem. We then outline a simple heuristic approach to tackle this problem in practice and illustrate the effectiveness of this approach via a toy example. We conclude by touching on a couple of additional research issues.

**Optimization.** Following the notations in § II and III, we use $C_h \in \mathcal{P}^e$ to denote that $C_h$ is on the path $\mathcal{P}^e$ (*i.e.*, a sequence of cache servers) from the edge cache server $C_e$ to the origin content server $C_o$. Hence, each intermediate cache server $C_h \in \mathcal{P}^e$ will allot a piece of its cache, $C_h^e$ ($0 \leq C_h^e \leq C_h$), to form one "BIG" cache for edge server $C_e$. The problem of cache allotment for "BIG" caches can be formulated as the following

optimization problem, where we want to find an optimal set of partitions, $\{C_h^e, h \in \mathcal{H}\}$, so as to maximize the overall cache network performance:

$$\begin{aligned} \underset{\{C_h^e, h\in\mathcal{H}\}}{\text{maximize}} \quad & \sum_{e\in\mathcal{E}} \sum_{i=1}^{N} \lambda_i^e \eta_i^e(C^e, \{\lambda_i^e\}) \\ \text{subject to} \quad & C^e = \sum_{h\in\mathcal{P}^e} C_h^e,\ e \in \mathcal{E} \text{ and} \\ & \sum_{e\in\mathcal{E}:h\in\mathcal{P}^e} C_h^e = C_h,\ h \in \mathcal{H}. \end{aligned}$$

For example, if the object hit probability $\eta_i^e(C^e, \{\lambda_i^e\}$ is a concave function of cache size $C^e$ (under the assumption of Poisson request arrival processes $\{\lambda_i^e\}$, $e \in \mathcal{E}$), then the above optimization problem can be solved via convex programming. In particular, if LRU is used, then $\eta_i^e(C^e, \{\lambda_i^e\}) = 1 - e^{-\lambda_i^e T^e}$ via Che's approximation, where $T^e$ is the characteristic time satisfying the constraint $\sum_i (1 - e^{-\lambda_i^e T^e}) = C^e$.

**Heuristic.** We outline a simple yet effective heuristic that can be useful to solve the cache allotment problem quickly in practice. The heuristic is based on the fact that for a single cache of size $C$, the *static caching* [12] which allocates the cache to the top $C$ objects (assuming all objects are of unit size) yields the best cache hit rate. Applying this fact to a cache network leads us to the following heuristic for cache allotment: starting from the bottom layer of edge servers, we allot the cache of an edge server to the top $C_e$ objects, and remove these objects from further consideration; more generally, at an intermediate cache server $C_h$ at the next layer, we merge and sort the request rates for the (remaining) objects from all paths $\mathcal{P}^e$ that traverse $C_h$ (we treat objects/object requests from different edge servers $C_e$ as *distinct*, even though some are for the same objects), and allot the cache to $C_h$ objects with the highest request rates. The number of objects (object request rates) from $C_e$ that is among these top $C_h$ objects would be the size of cache piece allotted to $C_e$. In the following, we will use a toy example to illustrate that this simple heuristic works well in practice, often yielding the "optimal" solution.

Consider a cache network of six cache servers, with $C_{11}^{(1)}, C_{12}^{(1)}, C_{21}^{(1)}, C_{22}^{(1)}$ as edge servers, and $C_1^{(2)}$ and $C_2^{(2)}$ as intermediate servers : $C_{11}^{(1)}$ and $C_{12}^{(1)}$ are connected to $C_1^{(2)}$, $C_{21}^{(1)}$ and $C_{22}^{(1)}$ are connected to $C_2^{(2)}$ and both $C_1^{(2)}$ and $C_2^{(2)}$ are connected to the origin server $C_o$, which offers a total of 100 objects. All caches are of size $C = 10$. Users serviced by $C_{11}^{(1)}$ and $C_{21}^{(1)}$ are interested in all 100 objects, where users serviced by $C_{12}^{(1)}$ and $C_{22}^{(1)}$ are only interested in the top 30 most popular objects. At all edge servers, object access probabilities follow a Zipf distribution with $\alpha = 1.0$. Since $C_1^{(2)}$ and $C_2^{(2)}$ are symmetric, we will focus only on $C_1^{(2)}$. Using our heuristic, $C_1^{(2)}$ would allot 3 units to $C_{11}^{(1)}$ and 7 units to $C_{12}^{(1)}$, yielding an allotment of $(3, 7)$. For comparison, we consider the following three allotments: $(9, 1)$, $(7, 3)$ and $(5, 5)$. Fig. 12 shows the overall hit rate with a simulation run of $1M$ requests at each edge server, where the *dCLIMB* cache replacement policy is employed. We see that our heuristic yields the best overall hit rate. Clearly, if $C_{11}^{(1)}$ is allotted more cache at $C_1^{(2)}$, the overall hit rate for its requests would increase, whereas the overall hit rates for requests from $C_{12}^{(1)}$ would decrease. Fig. 13 shows the respective gains or losses in hit rates for $C_{11}^{(1)}$ and $C_{12}^{(1)}$ under other three allotments relative to the "optimal" allotment $(3, 7)$.

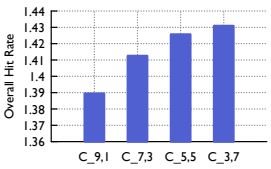



Fig. 12. Overall Hit Rate

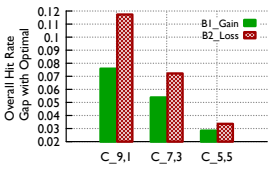



Fig. 13. B1_Gain, B2_Loss

As alluded in the description of our heuristic cache allotment algorithm above, requests arriving at an intermediate server $C_h$ from different edge servers may be for common objects. For requests for the same object coming from two different edge servers, we in fact only cache *one copy* of the same object in $C_h$ to attain better storage utilization. Instead, we simply keep track of the access patterns from each edge server separately using separate counters. In other words, under *dCLIMB*, the positions of objects in each "BIG" cache (logically) allocated to each edge server (along a path of the cache hierarchy) are maintained in separate counters (metadata structures) and adjusted independently for each edge server. The (*logical*) cache partition among the edge servers at $C_h$ is thus dynamically adjusted (instead of fixed) in accordance with changing access patterns. Due to space limitation, we will omit the detailed description of the algorithms for maintaining the statistics and managing the movement of objects in the intermediate cache servers. In a nutshell, under our scheme for the "BIG" cache abstraction with $dCLIMB$, the virtual caches allotted to the edge servers could in fact be larger than the physical cache size, due to common objects stored in an intermediate cache server that are accessed by multiple downstream edge servers. In order to coordinate and control the distributed cache servers as one "BIG" cache abstraction in a *global* manner, the framework proposed in [14] can be employed. However, how to implement such a framework for coordinating and controlling the distributed cache servers is outside the scope of this paper.

## VI. Evaluation

We evaluate "BIG" cache abstraction using a topology following the multi-layered architecture of YouTube video delivery system revealed in [2]. In this architecture, cache servers are organized in a 3-tier cache hierarchy (primary, secondary and tertiary servers), and the tertiary servers are connected to the origin server. User requests are first sent to the primary (or edge) cache servers. The primary cache server will service the request if it has the object cached. If not, the request is forwarded to the secondary layer. This process continues till the object is found. If none of the cache servers have the object, the request is forwarded to the origin server. With this notion, we consider a similar topology in the form of a binary tree, in which the root is the origin server (O) and the leaf nodes are the primary (*i.e.*, L1) servers which receive user requests. The intermediate servers comprises of

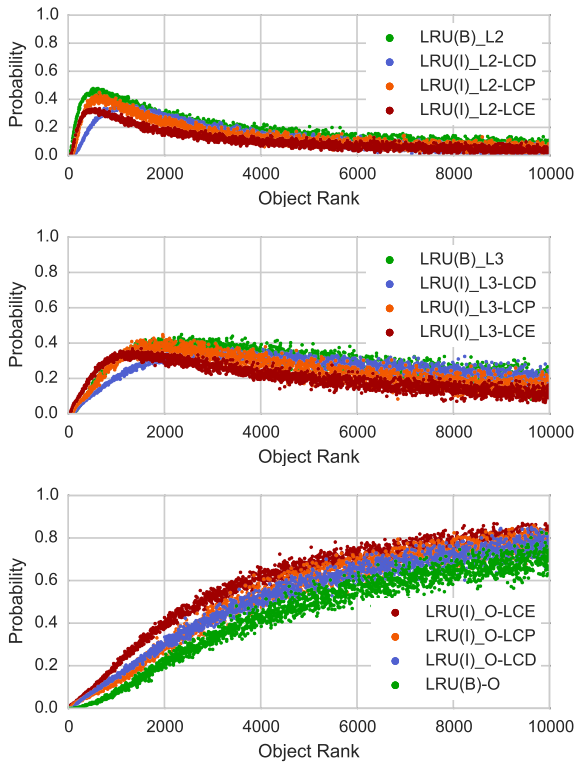



Fig. 14. Object hit probability at layer 2 (L2), layer 3 (L3) & origin (O)

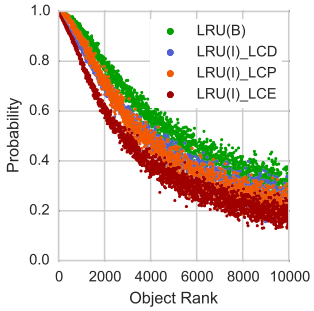



Fig. 15. Overall Object hit probability

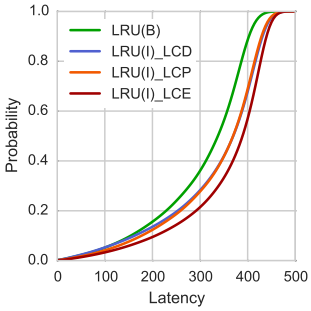



Fig. 16. Overall Latency (CDF)

the secondary (L2) and tertiary (L3) servers, thus $H = 3$. We consider a collection of 10000 objects of unit size whose access probabilities follow a Zipf distribution with $\alpha = 1$. Two million requests are generated for each edge server. We assume the rate of requests to be uniform across all primary servers. Cache sizes of primary, secondary and tertiary servers are set to be 1000, 2000 and 4000, respectively. Thus, the intermediate cache servers are partitioned equally among the different edge servers sharing them (*i.e.*, the total size of each big cache is 3000, that's 1000 from each cache layer.)

We compare the performance of applying LRU in "BIG" cache *LRU(B)* and applying LRU independently *LRU(I)* in each cache server. We also applied different object allocation methods (LCE, LCD, LCP) when LRU is applied independently, which specifies how objects are cached on the backward path while being sent from the cache where they were found to users. Fig. 14 shows the object hit probability at the different layers and the origin server. The object hit probability specifies the percentage of requests served by each layer and also the origin server. Although not shown, we find that LRU(I) and LRU(B) served almost a similar percentage at edge servers. However, Fig. 15 shows LRU(B) served a higher percentage of requests for almost all objects in the intermediate layers (L2 and L3). We also find that LRU(B) has the least percentage of requests being served by the origin server. All of this suggests that "BIG" cache abstraction leads the objects to being served more from the intermediate layers than going all the way to the origin server. We further compare the overall object hit probability from cache servers at layers L1, L2 and L3 (see Fig. 15) and the average estimated latency in fetching each of the objects (see Fig. 16). We clearly find that "BIG" cache abstraction outperforms all other methods. The main reason is that by caching only one copy at each path from the edge server to the origin server, we allow space for more objects to be cached at lower layers, and hence increase their hit probability and at the same time minimize the load on the origin server. Moreover, in "BIG" cache abstraction, objects evicted from one layer are not totally discarded but they are cached in the higher layer. Thus, "BIG" cache abstraction enables us to efficiently utilize cache resources available at cache servers along the path to the origin server while taking into consideration the requests from all edge servers sharing the same node. It is worth mentioning that the improvement in the performance shown in this section is just by using the simple LRU caching policy. We expect the performance to be even better using more sophisticated caching policies.

## VII. Conclusion

We have made a strong case for "BIG" cache abstraction to effectively utilize the distributed storage of all cache servers in a cache network. Through examples and simulations, we demonstrated that "BIG" cache abstraction can indeed eliminate the problem of (cascade) thrashing when cache servers operate independently with their own cache replacement policies and take full advantage of the additional cache resources available at intermediate cache servers. We are also afforded the added benefit that it is more amenable to theoretical performance analysis. "BIG" cache abstraction significantly improves the overall performance of a cache network while also drastically reducing the loads and other performance constraints at origin content servers. It therefore makes ICNs more efficient and scalable as a whole. "BIG" cache abstraction also opens up a number of new and challenging research questions and directions. As an initial step towards addressing some of these issues, we have developed the *dCLIMB* cache mechanism for "BIG" cache to minimize the overheads of moving objects across distributed cache boundaries. We also outlined a simple heuristic to address the cache allotment problem in the design of "BIG" cache abstraction.

**Acknowledgement.** This research was supported in part by NSF grants CNS-1411636, CNS 1618339 and CNS 1617729 and a Huawei gift.

## REFERENCES


[1] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang, "Unreeling netflix: Understanding and improving multi-cdn movie delivery," in *INFOCOM, 2012*.

[2] V. K. Adhikari, S. Jain, Y. Chen, and Z.-L. Zhang, "Vivisecting youtube: An active measurement study," in *INFOCOM, 2012*.

[3] A. Flavel, P. Mani, D. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev, "Fastroute: A scalable load-aware anycast routing architecture for modern cdns," in *NSDI*, 2015.

[4] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *SIGOPS*, 2010.

[5] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *CoNEXT*, 2009.

[6] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: Incrementally deployable icn," in *SIGCOMM, 2013*.

[7] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," in *INFOCOM, 2014*.

[8] V. Mathew, R. K. Sitaraman, and P. Shenoy, "Energy-aware load balancing in content delivery networks," in *INFOCOM, 2012*.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, "Finding a needle in haystack: Facebook's photo storage." in *OSDI*, 2010.

[10] A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy, "On the scale and performance of cooperative web proxy caching," in *SIGOPS*, 1999.

[11] H. Che, Z. Wang, and Y. Tung, "Analysis and design of hierarchical web caching systems," in *INFOCOM, 2001*.

[12] Z. Liu, P. Nain, N. Niclausse, and D. Towsley, "Static caching of web servers," in *Multimedia Computing and Networking 1998*.

[13] D. Starobinski and D. Tse, "Probabilistic methods for web caching," *Performance evaluation*, 2001.

[14] E. Ramadan, A. Narayanan, and Z.-L. Zhang, "Conia: Content (provider)-oriented, namespace-independent architecture for multimedia information delivery," in *ICMEW*, 2015.