# ACH: a Locality-aware Dispatcher for CDN Cache Clusters

Paper #1570745716

***Abstract*—Content Delivery Networks (CDN) are deployed globally to enhance the Internet users' experience. CDN nodes are typically composed of cache clusters that contain multiple servers with a static workload dispatcher. Intuitively, workload imbalances in a CDN cache cluster may provide worse performance than a single big CDN cache server of the same capacity. However, we demonstrate that a sophisticated dispatcher providing user request isolation in a cache cluster can achieve the same or better performance.**

**In this paper, we formulate the request dispatching problem and present Adaptive Consistent Hashing (ACH): a self-adaptive request dispatcher that boosts the hit ratio of a cache cluster. ACH profiles user requests and dynamically adjusts the request dispatching policy to balance workloads and maximize hit ratios.**

**ACH applies a modified access counter matrix to simplify the optimization process to linear time complexity. Our evaluation shows that ACH improves the hit ratio of the cache cluster by around 10%, reduces WAN traffic by 10-60% and performs as well as a single cache server of the same capacity. Furthermore, with video streaming traffic, a cache cluster with ACH achieves a hit ratio that is 8% higher than a single cache server of the same capacity.**

***Index Terms*—CDN, caching, dispatcher, load balancing, optimization**

## I. Introduction

Content Delivery Networks (CDN) are crucial to improving internet user experience by caching contents geographically closer to the users. CDNs have been widely deployed to improve latency and save bandwidth for over 20 years. In 2017, over 56% of the internet traffic was delivered via CDNs and that number is expected to rise to 72% by 2022 [1]. Improving the performance of cache servers in CDNs would make a significant impact on user experience. Increasing the hit ratio by 1% would reduce the request latency by 35% [2]. On each cache miss, the CDN cluster needs to fetch content from the remote origin server, resulting in significant latency and bandwidth costs. This latency degrades the user's experience and the bandwidth cost becomes a major part of CDNs' operation cost [3] [4] [5] [6] [7] [8]. Limited memory in a cache server is a major cause of low hit ratios. To increase total memory capacity, most CDN caches are implemented using a cluster consisting of multiple cache servers and a request dispatcher as shown in Fig. 1.

On the surface it would seem that a cache cluster cannot match the hit ratio of a single cache with the same memory capacity due to uneven workload distribution. Based on previous work on distributed databases, most cache cluster dispatchers are based on static dispatch policies like consistent hash [9] or size-based classification (Memcache) [10], which cannot adapt to dynamic workloads, leading to low hit ratios. Our

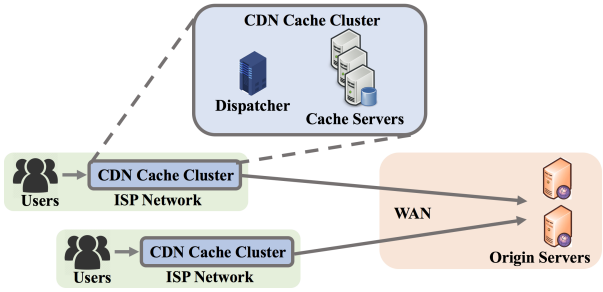
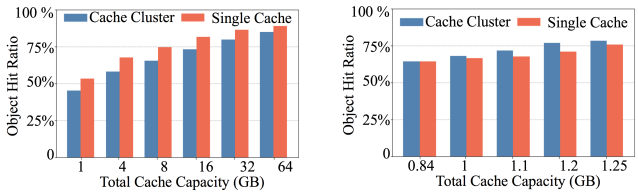



Fig. 1: CDN Cache Cluster Setup

simulation results in Fig. 2(a) show that a cache cluster with a hash-based dispatcher has a hit ratio 10% lower than a single cache of the same size.




(a) Wikipedia trace (b) Bilibili trace

Fig. 2: Hit Ratio of Single Cache and Cache Cluster

However, as shown in the simulation results in Fig. 2(b), a cache cluster can achieve a higher hit ratio than a single large LRU cache of equivalent capacity in some cases. We observe that the cache cluster provides the key property of *isolation*, which can potentially increase the hit ratio: User requests dispatched to different cache servers are isolated from each other and do not contend for the same memory resources. This property of isolation allows the dispatcher to protect high-demand contents from being evicted by others, thereby increasing the hit ratio of the cache cluster.

What is required therefore, is an intelligent dispatcher to analyze request locality and leverage the isolation property to boost the cluster's hit ratio.

We present ACH, a self-adaptive cache cluster dispatcher based on request locality. ACH profiles and dispatches requests to each cache server using a greedy-algorithm optimizer to maximize the cluster hit ratio and balance the workload. Our evaluation shows that ACH leverages isolation in a cache cluster and outperforms a single large cache hit ratio by 8% percent with certain types of content.

The contributions of our paper are three-fold:

- Identify the isolation property of cache clusters and use it effectively to improve the cache hit ratio. This paper further focuses on the optimization of request dispatchers in cache clusters to take advantage of the isolation property and defines a mathematical model for the request dispatching problem.
- Reduce the complexity of profiling the hit ratio to make it linear. Compared with existing algorithms with time complexity of $O(R^2)$[1], this paper adopts and modifies an access counter metric (ACM) based algorithm to profile the hit ratio of an LRU cache server in linear time complexity ($O(R)$).
- Present ACH, a self-adapting dispatcher that optimizes a cache cluster's hit ratio based on request locality and isolation. This paper further implements the ACH dispatcher simulator and prototype and provides both the hit ratio and overhead evaluation of ACH dispatchers. Our evaluation results show that ACH can reduce WAN traffic compared to the original hash-based cluster cache by up to 60% with only 3% additional CPU utilization on average. We represent that our algorithm is practical in real-world scenarios.

The rest of the paper is organized as follows. *Section II* introduces the background and motivation for our work. *Section III* presents the detailed design of ACH and the ACM-related algorithms. *Section IV* presents the ACH simulator and prototype with evaluation results and analysis. We discuss related work in *Section V* and conclude the paper in *Section VI*.

## II. Motivation

Memory capacity is a major bottleneck in CDN infrastructure [11]. To provide a reasonable memory capacity that ensures a decent hit ratio, CDNs typically deploy a cache cluster that consists of multiple cache servers as shown in Fig. 1. Typically, each user request that arrives at the cache cluster is dispatched to one of the cluster's cache servers by the request dispatcher, which maintains a mapping table between content IDs and cache server IDs. Upon arrival of a user request, the selected cache server handles it based on the admission/eviction schemes applied. Most cache servers apply the LRU (least recently used) policy that evicts the least recently accessed content when the memory is full [2], [10].

### A. Limitation of static request dispatchers

Most current cache clusters use a static request dispatcher that maps contents to servers with a fixed dispatch policy. There are mainly two kinds of dispatchers: class-based [10] and hash-based. Class-based dispatchers classify requests based on a specific parameter (e.g. content size, request frequency, etc) and each cache server handles the requests in a specific class. As an example, Memcache [10] uses a dispatcher that classifies requests based on content size and forwards them to the associated cache server that holds contents within that specific size range. Most cache clusters adopt hash-based dispatchers from distributed databases like Dynamo [12]. These hash-based dispatchers calculate the hash value of the content ID and forward the requests to the cache server associated with that hash value. The most widely used type of hashing function in CDN clusters is consistent hashing [9], which minimizes content migration when the cluster adds or removes cache servers.

The handling of dynamic request localities in static dispatchers may be suboptimal. In practical CDN and deployment scenarios, users' request patterns are dynamic and fast-changing. With a static hash function, it is hard to distribute the workload across the cache servers evenly at all times. Under an unbalanced workload distribution, popular contents on a busy cache server are more likely to be evicted than other cold contents in other idle servers, which leads to a low hit ratio when compared to a single cache with an equivalent memory capacity. We conducted a cache simulation based on an unbalanced Wikipedia trace [13] and observed a hit ratio decrease of around 10% as Fig. 2(a) shows.

Based on the above analysis, it is reasonable to conclude that a single cache would have a higher hit ratio than a cache cluster of the same memory capacity. However, is that always the case?

### B. Isolation

We observe that a cache cluster can achieve a higher hit ratio than a single cache with the same capacity in some cases. The simulation results in Fig. 2(b) show that the distributed cache cluster has a hit ratio about 6.7% higher than that of a single cache server with the same capacity. The key factor that makes a cache cluster outperform a single cache is that the dispatcher is able to split the user requests trace into multiple sub-traces for each cache server. This isolates the user requests on one cache server from those on other servers. Thus, the isolated user requests on different cache servers do not contend for the same shared memory space.

We use the example in Fig. 3 to illustrate how such isolation boosts the overall hit ratio in a cluster. In this example, users request five different contents in a loop. If the single cache server has memory capacity to hold only four contents, all requests result in cache misses. However, let's say that a dispatcher divides the requests into two sets, where Server 1 handles the requests for contents A and B while Server 2 handles the requests for contents C, D, and E. Although the requests for content C, D and, E result in cache misses on Server 2, the requests that go to Server 1 are mostly hits. In this case, the cache cluster achieves a hit ratio of 30%.

Different request dispatching strategies lead to different hit ratios in a cache cluster. By isolating the requests based on locality, a dispatcher can minimize the memory space contention and further boost the overall hit ratio of the cluster. The example in Fig. 4 shows that dispatching policy II better isolates the contending contents (A/B and C/D) and improves the overall hit ratio of the cache cluster cache by over 40%. Therefore, an intelligent request dispatcher needs to analyze

[1] $R$ is the number of requests in the user request trace

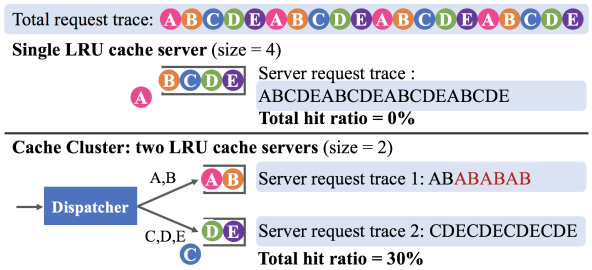



Fig. 3: Benefit of Isolation

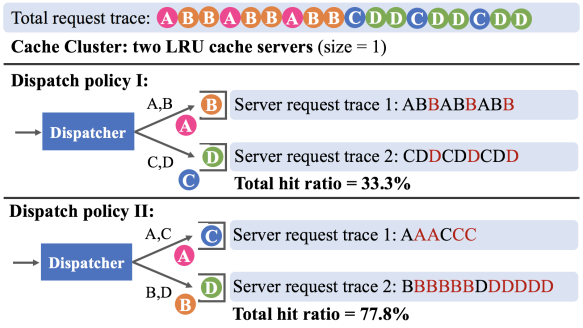



Fig. 4: Different Dispatching Policies

the locality of the user requests and decrease the contention between contents by the property of isolation to improve the overall hit ratio of the cache cluster.

### C. The request dispatching problem

We need to model the request dispatching problem to determine the optimized content-server mapping and implement a request dispatcher that maximizes the overall hit ratio. Therefore, we formulate the request distribution problem as the following cost function optimization:

minimize:

$$C(T) = \sum_{i=1}^{M} \frac{r_i}{R} C_i(t_i, \phi_i) \tag{1}$$

subject to:

$$\{t_1, ..., t_M\} = T \tag{2}$$

Function ***C(t)***, further defined below, is a cost function associated with the cache cluster and function $C_i(t_i, \phi_i)$ represents the cost associated with cache server $i$ as a function of its request trace ($t_i$) and its cache size ($\phi_i$).

$M$ is the total number of cache servers in the cluster, $R$ is the total number of requests, $r_i$ is the total number of requests dispatched to cache server $i$, $T$ is the user request trace of the entire cluster cache, and $t_i$ is the request trace that goes to server $i$. We further summarize the terms and notations in Table I.

The request dispatcher subdivides the total request trace $T$ into $M$ sub-traces $t_1, ..., t_M$ for each cache server. Here we define a filter function $f(T, f_i)$ to extract the corresponding sub-trace $t_i$ from the original trace according to the associated content subset $f_i$:

$$t_i = f(T, f_i) \tag{3}$$

TABLE I: TERMS AND NOTATIONS

| Notations | Descriptions |
|---|---|
| $R$ | Number of requests of the cache cluster |
| $r_i$ | Number of request of cache server $i$ |
| $T$ | Request trace of the cache cluster |
| $t_i$ | Request of cache server $i$ |
| $\Delta T$ | The length of the sampling window |
| $\phi_i$ | Cache size of server $i$ |
| $u$ | Total number of unique contents requested |
| $F$ | The total content set |
| $f_i$ | The content set associated with server $i$ |
| $P$ | Total content number in content set $F$ |
| $Z_n$ | File size of the $n^{th}$ request |
| $M$ | Number of cache servers in the cluster |
| $M'$ | Number of virtual nodes on the hash ring |
| $C(T)$ | Cost function of the cache cluster with trace $T$ |
| $C_i(t_i, \phi_i)$ | Cost function of cache server $i$ with trace $t_i$ and cache size $\phi_i$ |
| $\Delta m$ | Total missed bytes of the cluster cache |
| $\Delta m_i$ | Missed bytes of cache server $i$ |
| $v$ | Request workload variation |
| $A$ | Access counter matrix of the cache cluster |
| $A_i$ | Access counter matrix of cache server / virtual node $i$ |
| $\Delta X$ | $\Delta X$ matrix of the cache cluster |
| $\Delta X_i$ | $\Delta X$ matrix of cache server / virtual node $i$ |
| $\Delta Y$ | $\Delta Y$ matrix of the cluster cache |
| $\Delta Y_i$ | $\Delta Y$ matrix of cache server / virtual node $i$ |

$$\sum_{i=1}^{m} f_i = F \tag{4}$$

The request dispatcher maps the user request to each cache server based on content subset $f_i$. When a user requests content within $f_i$, the request is forwarded to the corresponding cache server $i$. Therefore, the filter function $f(T, f_i)$ extracts the sub-trace $t_i$ by selecting the requests for content within subset $f_i$. We define the request distribution problem as finding the optimum content sets $\{f_1, ..., f_M\}^*$ that minimize the overall cost function ***C(t)*** of the cache cluster.

*a)* ***Cost function****:* In the request dispatching problem, the dispatcher needs a proper cost function $C_i(t_i, \phi_i)$ to reflect the desired performance metrics of the cache cluster. Cache clusters may have different performance metrics (e.g. hit ratio, bandwidth usage, response time, etc). In this paper, we focus on two aspects of cache cluster performance: 1) Hit ratio and 2) Request workload variation. We use the number of requests and cache misses $m_i$ to determine the hit ratio. In addition, we define the request workload variation $v$ in equation (5).

$$v = \frac{\max_{i \in M}\{r_i\} - \min_{j \in M}\{r_j\}}{R} \tag{5}$$

The request workload variation $v$ is a measure of the uniformity of distribution of the request workload among all the cache servers in the cluster. A lower request workload variation $v$ indicates a more balanced workload distribution, which avoids CPU and bandwidth usage peaks on a specific busy server in the cluster.

An ideal dispatcher needs to boost the overall hit ratio of a cache cluster as well as balance the request workloads among all the cache servers. Thus, we combine both the hit ratio and workload variation in the cost function shown in equation 6

$$C_i(t_i, \phi_i) = \alpha m_i + (1 - \alpha) v \tag{6}$$

$\alpha$ is a parameter that adjusts the weight between the cluster hit ratio and the workload variation. Using equations (1) and

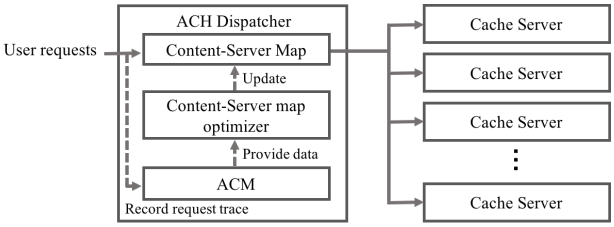
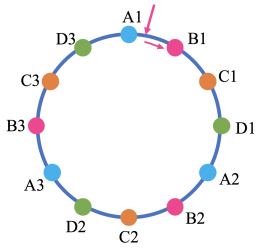



Fig. 5: Overview of ACH Workflow




| Cache server | Virtual Nodes |
|---|---|
| A | A1, A2, A3 |
| B | B1, B2, B3 |
| C | C1, C2, C3 |
| D | D1, D2, D3 |

Fig. 6: 'Virtual node'-based Consistent Hashing

(6), the overall cost function of the cache cluster *C(T)* is expressed as:

$$C(T) = \sum_{i=1}^{M'} \frac{r_i}{R} C_i(t_i, \phi_i) = \alpha \sum_{i=1}^{M'} \frac{r_i}{R} m_i + (1-\alpha) M' v \quad (7)$$

*b)* ***Optimal request dispatching****:* Brute force would be a simplistic approach to determine the optimum content subset $\{f_1, ..., f_M\}^*$. The optimizer can go through all the possible combinations of content subsets, calculate the cost function *C(T)*, and find the global optimum solution $\{f_1, ..., f_M\}^*$. This is impractical as the total number of possible content subsets can be extremely large. There are a total of $C_P^M$ possibilities, where $M$ is the number of cache servers and $P$ is the size of the total content set *F*. This leads to a time complexity of $O(P!)$, which is not practical. A better approach is described below.

## III. Design of Adaptive Consistent Hashing (ACH)

### A. ACH: Self-adaptive heuristic dispatching policy optimizer

We introduce Adaptive Consistent Hashing (ACH), a self-adaptive hash-based dispatcher with a heuristic optimizer. Rather than trying out all the possible content subsets, ACH optimizes the cost function *C(T)* step by step using a local greedy search.

*1) ACH overview:* As shown in Fig. 5 the ACH dispatcher includes three major components: a Content-Server Map, a Content-Server Map Optimizer, and an Access Counter Matrix (ACM).

The Content-Server Map serves as a look-up table. When a user request arrives at the ACH dispatcher, it looks up the cache server's IP address in the Content-Server Map based on the requested content ID and forwards the request to it as shown in Fig. 5. Such a look-up process is simple and fast and is the major task of the ACH dispatcher.

The Content-Server Map Optimizer is used to adjust the look-up table to boost the hit ratio based on the user request locality. This optimizer adjusts the Content-Server Map periodically as described in section III-C.

The ACM works as a database to store the content access information and feeds user request data to the optimizer. Upon arrival of each user request, ACH stores the access information in its ACM. ACM's operation is described in section III-B.

*2) 'Virtual node'-based consistent hashing:* ACH groups content using the concept of a 'virtual node' [9] to decrease the size of the request dispatching optimization problem. As Fig. 6 shows, consistent hashing places multiple 'virtual nodes' on a hash ring, dividing it into sections. When a content ID is mapped onto the hash ring, it is associated with the next node in the clockwise direction. In the consistent hashing approach, a much larger number of 'virtual nodes' is usually deployed on the hash ring than the number of cache servers. Therefore, multiple 'virtual nodes' are associated with a physical cache server, which allows the consistent hash to divide the hash ring into more sections with finer granularity for better load balancing. To illustrate the virtual node concept, consider the the example in Fig. 6. There are four cache servers in the cluster and 12 'virtual nodes' on the hash ring. Each server is associated with three 'virtual nodes' as the table shows. When a new request (marked as the pink arrow) arrives, the dispatcher calculates its hash value on the hash ring and finds the nearest virtual node in the clockwise direction, which is node B1. According to the virtual node association table, node B1 is associated with cache server B. Therefore, the new request is dispatched to server B.

*3) Periodic request dispatching optimization:* ACH optimizes the request dispatcher periodically by adjusting the virtual node association based on the user request locality. As stated in section II-A, the static request dispatch policy cannot adapt to the variation in locality of dynamic user requests. ACH uses a sampling window $\Delta T$ to collect the user request trace and adjust its dispatching policy according to it. The goal of the request dispatch optimization is to minimize the cost function *C(T)* for the cluster cache. To achieve this goal, ACH profiles the cost function $C_i(t_i, \phi_i)$ for each cache server based on the user request trace collected from the previous sampling window. ACH adjusts the content-server association with the granularity of 'virtual nodes' to reduce time complexity. We discuss this optimization process in more details in section III-C.

### B. Profile hit ratio by access counter matrix

*1) Stack distance and number of request misses:* For the periodic optimization process, ACH uses the concept of *stack distance* [14] to calculate the number of cache misses $\Delta m$ [15] [16] [2] [17] [18] [19]. Stack distance is defined as the total size of unique content accessed between one request and the previous request accessing the same content. Under LRU eviction policy, if a request has a stack distance larger than

the cache capacity, the request is considered a cache miss, otherwise, it is a cache hit [14], [20]. Using this concept, we can easily determine the number of cache misses and calculate the hit ratio of the cache server using stack distance.

*2) Access Counter Matrix (ACM):* To calculate the stack distance of each request, we need to store and calculate the total size of unique content accessed between each request and the previous one accessing the same content. We adopt a matrix data structure called Access Counter Matrix (ACM) [16] to compute the stack distance of each request in the trace.

Upon arrival of a request, ACH constructs a new access counter. Every access counter accumulates the size of the unique content requested since its construction. Thus, when a new request arrives, every existing counter that has not previously seen the requested content would increase by the size of the content. ACH maintains one access counter array (ACA), an array that tracks the latest value of the total size of the unique content starting from each timestamp (as the index of the element in the array), for each virtual node. Every request appends one new access counter element into the ACA and updates all the other elements. We take a snapshot of the values in ACA after every request and store them as a column in a triangular matrix, namely ACM, as shown in Fig. 7[2]. In an ACM $A$, element $A_{(i,j)}$ represents the total size of unique content accessed between the $i^{th}$ and $j^{th}$ requests. In the example shown in Fig. 7, element $A_{(3,7)} = 4$ means the total unique content accessed between the third and the seventh request is 4.

To determine whether a certain content has been previously accessed, we compare the value changes of the access counter. If element $A_{(i,j-1)} < A_{(i,j)}$, it means the $j^{th}$ request accessed new content that has never been accessed after the $i^{th}$ request. Based on the above, if we subtract each access counter element $A_{(i,j)}$ by its neighbor element $A_{(i,j-1)}$ on the left, we get a matrix $\Delta X$ that shows the requests that accessed new content. Let us define $\Delta x_{(i,j)} = A_{(i,j)} - A_{(i,j-1)}$. If an element $\Delta x_{(i,j)}$ is greater than zero, it means the $j^{th}$ request accessed new content that has never been accessed after the $i^{th}$ request. Consider the example in Fig. 7. $\Delta x_{(3,6)} = A_{(3,6)} - A_{(3,5)} = 1$ means the content $A$ that is accessed in the sixth request (the column number) has never been accessed after the third request (the row number), and the size of this new content is 1. For element $\Delta x_{(3,7)} = A_{(3,7)} - A_{(3,7)} = 0$ in the $\Delta X$ matrix, the content $C$ accessed in the seventh request has been accessed after the third request.

To profile the stack distance of a request, we need to find the previous request that accesses the same content. Let's say the $i^{th}$ request is the previous request that accesses the same content $m$ as the $j^{th}$ request. This means starting from the $i^{th}$ request, content $m$ is not a new content. Therefore, element $\Delta x_{(k,j)} = 0$ in matrix $\Delta X$ for all $k > i$. On the other hand, after the $i^{th}$ request, content $m$ is not accessed until the $j^{th}$ request. This means $\Delta x_{(k,j)} > 0$ for all $i < k < j$. Therefore, we can find the previous request $i$ that accesses the same content as the $j^{th}$ request by checking matrix $\Delta X$ and finding the element satisfying $\Delta x_{(i,j)} = 0$ and $\Delta x_{(i+1,j)} > 0$. Let us consider content $C$ in Fig. 7 for an example. Content $C$ is accessed in the $3^{rd}$ and $7^{th}$ requests. This means element $\Delta x_{(k,7)} = 0$ for all $k \leq 3$ and all the elements $\Delta x_{(k,7)} > 0$ for all $3 < k \leq 7$ as the figure shows. The position where $\Delta x_{(i,j)}$ transitions from 0 to a non-zero value is where this content is last requested. In this example it is in the $3^{rd}$ request.

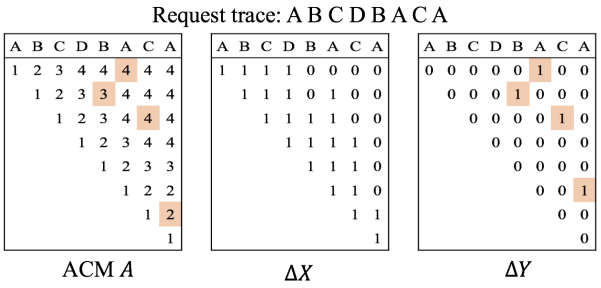



Fig. 7: Profiling Stack Distance by ACM

To simplify finding the last request accessing the same content, we define another matrix $\Delta Y$ in equation (8).

$$\Delta y_{(i,j)} = \begin{cases} x_{(i,j)} - x_{(i,j)} & , \quad i < j \\ 0 & , \quad i = j \end{cases} \tag{8}$$

Each non-zero element $\Delta y_{(i,j)}$ in $\Delta Y$ indicates that the $j^{th}$ request accessed the same content as the $i^{th}$ request. As a result, the stack distance of the $j^{th}$ request equals to the access counter matrix element $C_{(i,j)}$ with the same coordinate as $\Delta y_{(i,j)} > 0$. In the example in Fig. 7, there are four shaded non-zero elements in the matrix $\Delta Y$, namely $\Delta y_{(2,5)}$, $\Delta y_{(1,6)}$, $\Delta y_{(3,7)}$ and $\Delta y_{(6,8)}$. Element $\Delta y_{(3,7)}$ indicates that content $C$ that was accessed by the $7^{th}$ request was previously accessed by the $3^{rd}$ request, and the stack distance is equal to $C_{(3,7)} = 4$.

*3) Combination and maintenance of ACM:* To complete the optimization process described in section III-A3, ACH needs to compare the cost function $C_i(t_i, \phi_i)$ with different 'virtual node' associations. One of the advantages of the access counter matrix is that it makes it much easier to get the ACM of each cache server by combining the ACMs of all the virtual nodes associated with it.

The example in Fig. 8 illustrates how ACH combines different ACMs. ACM $A_i$ only updates the content access information related to the requests handled by virtual node $i$. Otherwise ACA simply appends one more access counter without updating elements, thus the corresponding column in ACM $A_i$ remains same with the previous one. In the example, requests that access content $A$, $B$ are handled by node 1 and requests that access content $C$, $D$ are handled by node 2. When combining two ACMs, ACH simply performs matrix addition to get the combined matrix as $A$ shown in Fig. 8, where $A_{(i,j)} = A_{1(i,j)} + A_{2(i,j)}$.

[2]To better introduce the concept of stack counter and request matrix, we assume all the contents in this example have the same size of 1, which makes the total size of the unique contents equivalent to the total number of the unique contents.

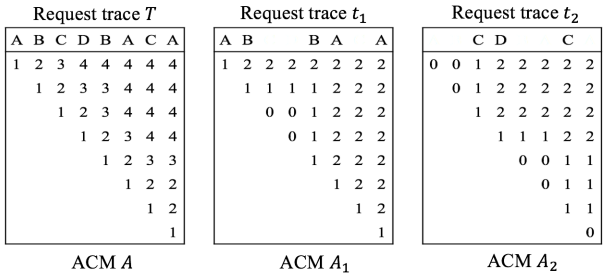



Fig. 8: Combination of ACM

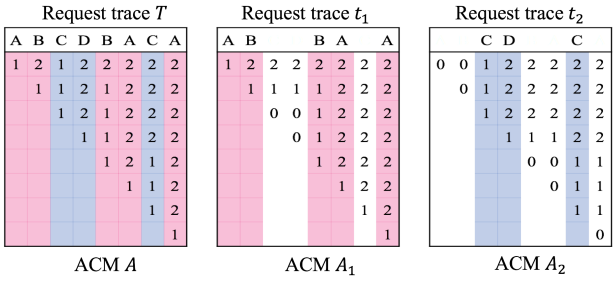



Fig. 9: Compressed ACM

Since the content association optimization is performed at the virtual node level, ACH needs to maintain an access counter matrix $A_i$ for each virtual node $i$.

### C. Virtual node association adjustment

ACH adjusts the content association between the busiest and idlest cache servers to minimize the overall cost function ***C(t)***. A busier cache server usually has a larger cost function value due to a higher miss ratio, caused by heavily requested contents that lead to severe memory contention. To optimize the request locality, ACH needs to migrate part of the content associated with the busiest cache server to the idlest cache server in order to balance the request workload and increase the overall hit ratio. At the end of each time window $\Delta T$, ACH profiles the cost function $C_i(t_i, \phi_i)$ of each cache server and finds the busiest and idlest servers with the highest and lowest cost function values, respectively. The next step is to select the optimum 'virtual node' to migrate between them.

ACH needs to traverse all the virtual nodes associated with the busiest server to find the best candidate that would minimize the cost function. This means that ACH needs to evaluate the cost functions after migrating each candidate virtual node. ACH can profile the cost function of different virtual node associations efficiently by combining the ACMs of the candidate virtual nodes as described above in III-B3. During the virtual node association adjustment process, ACH maintains ACMs for each virtual node $\{A_1, ..., A_{M'}\}$ during time window $\Delta T$. ACH gets the ACMs for the busiest (maximum cost) server $S^{max}$ and those of the idlest (minimum cost) server $S^{min}$ by combining the ACMs of all the virtual nodes associated with them. ACH then traverses each virtual node $i$ in server $S^{max}$ and simply subtracts matrix $A_i$ from $A^{max}$ and adds it to $A^{min}$ to get the resulting ACMs for both servers after migration. With the resulting ACMs, ACH profiles the updated number of misses based on section III-B3, calculates the new cost function, and selects the candidate virtual node that minimizes the cost function.

### D. Complexity analysis and optimization

*1) Space and time complexity analysis:* ACH maintains an ACM $A_i$ for each virtual node $i$ for the adjustment process described in section III-C. This results in a total of $M'$ $R \times R$ triangular matrices, where $M'$ is the number of the virtual nodes on the hash ring and $R$ is the number of requests during the time window $\Delta T$. Therefore, ACH uses a space complexity of $O(R^2M')$ to maintain the necessary information. The adjustment process requiring ACH to traverse through all the possible candidates makes the time complexity of this process $O(R^2)$. We reduce both space and time complexities as described below.

*2) Space complexity reduction:* We manage to reduce the space complexity by maintaining only one global ACM $A$ for the entire cache cluster while preserving the same content access information. According to section III-B3, $A_i$'s columns are considered as significant only when a request that lands on a virtual node causes an update of the corresponding ACA. Other $A_i$ columns are redundant copies of the significant ACA values to their left, as shown in Fig. 8. These redundant columns, shown as unshaded in the example of Fig. 9, can be eliminated, which would allow us to compress all virtual nodes ACMs into one global ACM $A$ for the cluster. For each request, only one virtual node's ACA is updated and recorded into the corresponding column in $A$, as shown in Fig. 9. In this example, the request dispatch policy is the same as the one in Fig. 8. Since requests $1, 2, 4, 5, 8$ land on node 1, these columns in ACM $A_1$ are considered significant and shaded in red. Likewise, the significant columns in ACM $A_2$ are shaded in blue. Now ACH only needs to maintain a single compressed ACM $A$ that contains the significant columns from ACM $A_i$ of each virtual node [3]. By compressing the ACM, the space complexity is reduced from $O(R^2M')$ to $O(R^2)$.

We can further reduce the space complexity of ACM $A$ by compressing the redundant information in each column by observing that the count values in each column may remain the same over several rows. Therefore, we can partition each column of the ACM into multiple sections, where all the elements in each section have the same value. This allows us to compress each column by only recording the starting index of each section and its corresponding element value, expressed as $[(i_1, e_1), ..., (i_n, e_n)]$, where $i_n$ is the row index of section $n$ and $e_n$ is the element value of section $n$. In Fig. 9, ACM $A$ has two sections in column 7. The first section starts at index $(1)$ with the element value of 2, and the second section starts at index $(5)$ with the element value of 1. Therefore, column 7 can be compressed as $[(1, 2), (5, 1)]$.

[3]We also record additional information for each column of $A$ that is required for processing, including virtual node index and server index.

Since the number of sections in a column is not larger than the unique content accessed by the virtual node, the space complexity of each column can be expressed as $O(u)$, where $u$ is the total unique content accessed during $\Delta T$. Therefore, the total space complexity of ACM $A$ is reduced to $O(uR)$.

*3) Time complexity reduction:* We introduce an improved algorithm to profile the cost function of each cache server in linear complexity with respect to the number of requests *R*.

Rather than subtracting $A_i$ from $A_{max}$ and adding it to $A_{min}$ as described in section III-C, we simplify this process by traversing ACM $A$'s columns from right to left to update the stack distance for the incoming requests. There are 4 different cases for each column $j$ in the compressed ACM $A$: the $j^{th}$ request that a column represents could be either 1) handled by server $S_{min}$; 2) handled by server $S_{max}$ and landing on virtual node $i$; 3) handled by server $S_{max}$ but not landing on virtual node $i$ and 4) handled by a third server. The requests corresponding to the first three cases need to update their stack distances, thus we use three queues: $q_{min}$, $q_i$, and $q_{max}$ to store the traversed requests that need to be updated in each case.

For the first case, since server $S_{min}$ is now combining with virtual node $i$, the request $j$ that handled by server $S_{min}$ will add its access counter value to all the requests that land on virtual node $i$ after request $j$. In this way, we pull out each request $k$ from $q_i$, find the last request $k'$ that accesses the same content and add the access counter value $A_{(k,k')}$ and $A_{(j,k')}$ to get the updated stack distance.

The second case needs two actions: As we move virtual node $i$ from server $S_{max}$ to server $S_{min}$, we need to decrease the access counter value of the requests on server $S_{max}$ and increase the access counter value of the requests on server $S_{min}$ after request $j$. The first action is the inverse of the first case. As we move virtual node $i$ out of server $S_{max}$, we need to decrease the access counter value of the requests on server $S_{max}$ after request $j$. Following a similar process to get the updated stack distances, we pull out each request $k$ from $q_{max}$, find the previous request $k'$ accesses the same content and decrease $A_{(k,k')}$ by $A_{(j,k')}$. The second action is very similar to the first case, which updates the stack distance of requests on server $S_{min}$ after request $j$ by adding the access counter value. In the third case, we only need to push these requests into the queue $q_{max}$ and then update the stack distance of these requests in the second case.

We summarize the improved virtual node association adjustment process in algorithm 1. [4] The improved algorithm updates the stack distance of the requests affected by the migration using a single computation. Thus, it has linear time complexity of $O(R)$ to profile the cost function for finding the optimum virtual node to migrate among all the candidates.

**Algorithm 1** Improved v-node association adjustment

**Input:** Max cost server $S_{max}$; Min cost server $S_{min}$; Compressed ACM $A$; Stack distance threshold $T$
**Initialize:** Optimum virtual node index $i^* \leftarrow 0$, Minimal missed bytes $m_{min} \leftarrow \infty$
**for** all virtual nodes $i$ associated with server $S_{max}$ **do**
  Reset missed bytes $m \leftarrow 0$
  Clear queues: $queue_{min}, queue_{max}, queue_i$
  **for** each column $j$ in $A$ from right to left **do**
    Get virtual node $V$, server $S$ associated with request $j$
    **if** $S == S_{min}$ **then** ▷ // Case 1
      Push request $j$ into $queue_{min}$
      **for** each request $k$ in $queue_i$ **do**
        $k' \leftarrow$ last request index accessing the same content as request $k$
        **if** cannot find $k'$ or $A_{(k,k')} + A_{(j,k')} > T$ **then**
          $m \leftarrow m + Z_k$ ▷ // Request k is a miss
        Remove request $k$ from $queue_i$
    **else if** $V == i$ **then** ▷ // Case 2
      Push request $j$ into $queue_i$
      **for** each request $k$ in $queue_{max}$ **do**
        $k' \leftarrow$ last request index accessing the same content as request $k$
        **if** cannot find $k'$ or $A_{(k,k')} - A_{(j,k')} > T$ **then**
          $m \leftarrow m + Z_k$ ▷ // Request k is a miss
        Remove request $k$ from $queue_{max}$
      **for** each request $k$ in $queue_{min}$ **do**
        $k' \leftarrow$ last request index accessing the same content as request $k$
        **if** cannot find $k'$ or $A_{(k,k')} + A_{(j,k')} > T$ **then**
          $m \leftarrow m + Z_k$ ▷ // Request k is a miss
        Remove request $k$ from $queue_{min}$
    **else if** $S == S_{max}$ **then** ▷ // Case 3
      Push request $j$ into $queue_{max}$
  Calculate the missed bytes in the remaining non-empty queues
  **if** $m < m_{min}$ **then**
    $m_{min} \leftarrow m$
    $i^* \leftarrow i$ ▷ // Update Optimum virtual node
Migrate virtual node $i^*$ from $S_{max}$ to $S_{min}$

## IV. Implementation and Evaluation of ACH

### A. Simulation platform

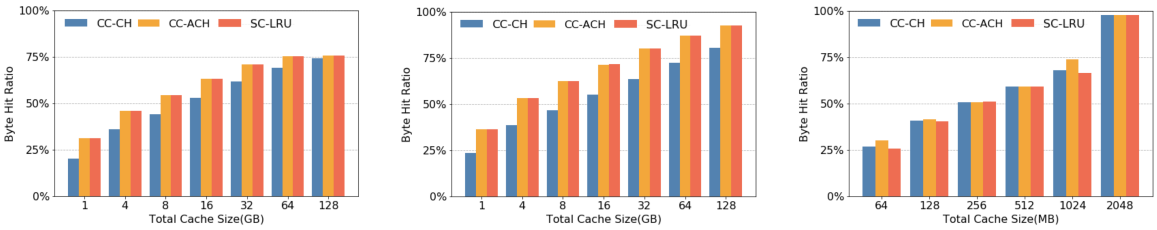
*1) Setup of simulator:* We extend the AdaptSize simulator [11] to cache clusters. In our simulation, we established four cache servers in the cache cluster with an LRU eviction policy. We set up two different request dispatch policies in our evaluation: ACH, denoted by CC-ACH and virtual-node-based static consistent hash [21], denoted by CC-CH. We also set up a single LRU cache server with a memory capacity equivalent to the cache cluster for our benchmark, denoted by SC-LRU.

*2) Trace information:* Our evaluation uses three traces from Wikipedia, ChinaCache, and Bilibili. The Wikipedia trace represents web-service traffic containing photos and other media content. The ChinaCache trace consists mainly of mobile software-related traffic and the Bilibili trace represents video streaming traffic. We summarize the properties of the three traces in Table II.

*3) Simulation results:*

*a) Byte hit ratio:* We show the byte hit ratio results in Fig. 10. In general, CC-ACH's performance is better than CC-CH's. For the Wikipedia and ChinaCache traces, CC-ACH has a byte hit ratio 15% higher than CC-CH in most cases and

[4] ACH pre-calculates the access counter values of server $S^{max}$ and $S^{min}$ and updates them in ACM $A$.




(a) Wikipedia (b) ChinaCache (c) Bilibili

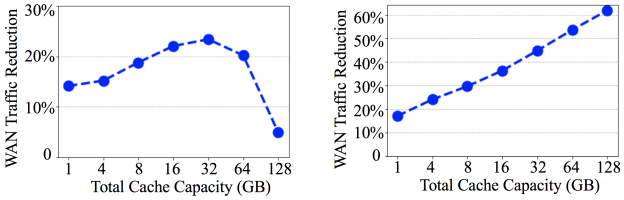
Fig. 10: Evaluation Results of Byte Hit Ratio

TABLE II: Trace Summary

| | Wikipedia | ChinaCache | Bilibili |
|---|---|---|---|
| **Total requests** | 10000000 | 55555251 | 17556378 |
| **Unique Obj requests** | 1032216 | 38846 | 545286 |
| **Total bytes requested (GB)** | 321 | 57969 | 14107 |
| **Unique bytes requested (GB)** | 75 | 39 | 470 |
| **Mean request obj size (KB)** | 73 | 998 | 862 |
| **Max request obj size (KB)** | 128755 | 1048 | 161923 |

matches SC-LRU's byte hit ratio within 1%. The Bilibili trace shows the benefit of cluster caches as discussed in section II-B. Both CC-ACH and CC-CH have a byte hit ratio equal or better than SC-LRU under most memory capacities thanks to the isolation property. CC-ACH leverages isolation more effectively and optimizes the dispatching policy based on the traffic locality, resulting in a byte hit ratio up to 8% higher than the SC-LRU benchmark. Considering all the simulation results, CC-ACH performs at least as well as and in certain cases better than the ideal single cache benchmark.

*b) WAN traffic reduction:* To better quantify the impact of CC-ACH on the user experience, we evaluate the WAN traffic reduction with the benchmark of the most widely applied consistent-hash-based cluster cache. We show CC-ACH's WAN traffic reduction relative to CC-CH in Fig. 11. CC-ACH achieves a reduction in WAN traffic of 5%-23% for the Wikipedia trace and up to 60% for the ChinaCache trace when compared to CC-CH.




(a) Wikipedia (b) ChinaCache

Fig. 11: WAN Traffic Reduction Over Consistent-hash-based Cluster Cache

*c) Stack distance analysis:* We further analyze the stack distance distribution to explain why cache clusters perform better than a single LRU cache with the Bilibili trace. We profile the request stack distance distribution of the single cache server and each cache server in the cache cluster as shown in Fig. 12. The blue curves show the request counts with different stack distances and the red line shows the miss ratio curve (MRC) [22] based on different cache sizes. According to Fig. 12(a), there are two peaks in the SC-LRU stack distance distribution: one around 0 and another around $1400MB$. This means that a large portion of the requests in this trace have a stack distance of around $1400MB$. Thus, an LRU cache server needs to have a memory capacity greater than $1400MB$ to guarantee a high byte hit ratio. On the other hand, CC-ACH managed to shift the second peak in the stack distance distribution to a smaller value by request dispatch optimization. In Fig.12(c) and 12(e), the second peak is below $300MB$ and most requests in Fig. 12(b) have a stack distance of less than $300MB$. This means three cache servers in CC-ACH only need a memory capacity of $300MB$ to reach a relatively high hit ratio. This shows that a CC-ACH with a total memory capacity of $1GB$ reaches a higher byte hit ratio than an SC-LRU server with the same capacity.

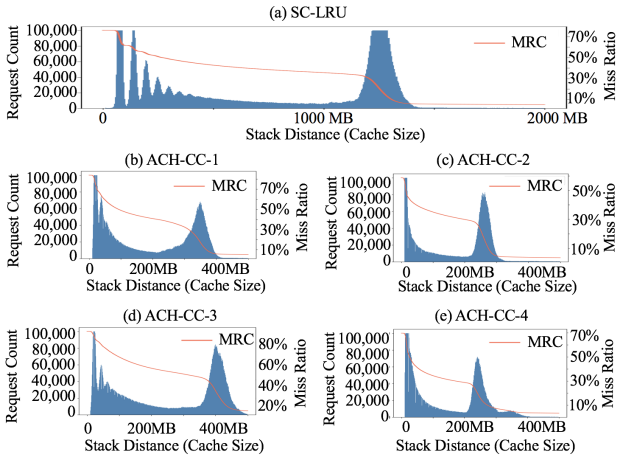



Fig. 12: Distribution of Stack Distance

## B. Implementation of ACH prototype

We implemented a cache cluster with an ACH dispatcher and four cache servers as shown in Fig. 13. Our prototype contains four physical servers with an *Intel Xeon Silver 4116* CPU and 256GB of memory. All the servers are interconnected by 10 Gbps Ethernet links. We deploy a client server from LRB [23] on server 1 to generate user requests as well as measure the user request throughput and latency. The setup of the client server is identical to the prototype in LRB, which contains 1024 threads to generate HTTP-based user requests in parallel. We deploy an OpenResty [24] agent on server

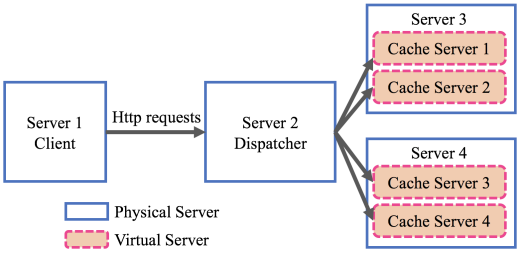



Fig. 13: Setup of ACH Prototype Testbed

2 to implement the request dispatcher, which will return a cache server IP address for each HTTP request from the user. We implemented an ACH dispatcher and a consistent hash dispatcher in the OpenResty agent for comparison.

Due to the limited number of physical servers, we deployed four virtual cache servers on physical servers 3 and 4. Each server has two virtual OpenResty backend web servers with LRU caches. When a user request is a cache miss in a virtual cache server, the server will generate a file with the corresponding size and send it to the user as well as store a copy in the LRU cache.

*1) Prototype evaluation results:* For our prototype evaluation, we use the Wikipedia trace shown in Table II. The client server generates HTTP requests based on the timestamp of each user request in the Wikipedia trace to represent the web-based traffic. The evaluation results are shown in Table III.

TABLE III: Results of Prototype Evaluation

| **Dispatcher** | **Consistent Hash** | **ACH Dispatcher** |
|---|---|---|
| **Max throughput** | 21967 Rps | 19801 Rps |
| **Average throughput** | 2471 Rps | 2352 Rps |
| **Max CPU usage** | 6.3% | 12.2% |
| **Average CPU usage** | 3.1% | 6.0% |
| **Max memory usage** | 4.71 GB | 5.53 GB |
| **Average memory usage** | 4.67GB | 5.31GB |

Our hardware prototype evaluation shows that ACH has minimal overhead when compared with the consistent hash benchmark. ACH reaches an average throughput close to 95% of that of consistent hash. As for resource overhead, ACH uses 2.9% more CPU than consistent hash on average and 5.9% more in the worst case. Moreover, ACH's memory usage is 18% higher than that of the consistent hash benchmark. With the minor overhead shown in Table III, ACH can reach a hit ratio 10% higher than consistent hash in most cases.

## V. Related Work

### A. CDN admission and eviction policy

Researchers have been focusing on the admission and eviction policies of CDN servers to improve performance. Adpatsize [11] is an optimized admission policy that maximizes the object hit ratio by selectively admitting the popular content based on object size as well as request characteristics. ARC [25] adaptively adjusts the eviction policy by combining the LRU and LFU (Least Frequently Used) policies. LRB [23] is a sub-optimum eviction policy that approximates the optimum solution Belady [26] by utilizing machine learning methods. When these admission and eviction policies are applied on each cache server, the ACH dispatcher can complement them to boost the performance of the CDN cache cluster.

### B. Miss ratio based Memory Cache Strategy

The CDN cache cluster is similar to memory caches in distributed system for which a body of work exists to improve hit ratios. These works adopt the concept of MRC (miss ratio curve) to study the relation between hit ratio (miss ratio) and the available memory capacity. Dynacache [27] focuses on the adjustment strategy that re-allocates memory capacity among different caches to improve the overall hit ratio by analyzing the request distribution. Cliffhanger [2] solves the same memory allocation problem based on the profiled MRC and a hill-climbing algorithm. LAMA [28] optimizes memory allocation using dynamic programming and reaches a near-optimum solution. RobinHood [29] focuses on the tail-latency optimization using dynamic cache reallocation. Adaptive Hash System [30] use a well-designed hash function to adjusts the hash space based on the recorded hit ratio of each cache server. However, this paper fails to discover and utilize the isolation property provided by the cache cluster and has limited flexibility in hash-space re-allocation due to the nature of its adjustment algorithm.

### C. Other CDN optimizations

The CDN community has been working on performance optimization in multiple aspects. EdgeCache [31] focuses on minimizing stall duration tail probability (SDTP) by adjusting bandwidth allocation and parallel streams. Tang et al. [32] optimize cache deployment based on the varying distribution of user requests in different geographical areas. Proactive-Push [33] propose a video push mechanism to decrease bandwidth consumption by predicting popular content and pushing it to the cache proactively. NCDN [34] combines CDN with Named Data Networking (NDN), which allows CDNs to route user requests to the best node caching the content, increase robustness as well as support multicast better. Atre et al. [35] propose a latency-optimal offline caching algorithm to reduce the average delay by incorporating the 'aggregate delay'.

## VI. Conclusion

In this paper, we formulate and study the CDN request dispatching problem and identify the isolation property of a CDN cache cluster to improve the hit ratio performance. Based on our observations, we present and implement the ACH dispatcher that boosts the hit ratio of a cache cluster by analyzing the locality of the user requests and dynamically adjusting the request dispatching policy. We modify an ACM-based algorithm to simplify the optimization process to linear time complexity. Our evaluation shows that, in general, ACH performs as well as a single cache server with the same capacity and even outperforms it in certain cases by leveraging the isolation property. As future work, we plan to generalize ACH to interoperate with different admission/eviction policies on cache servers.

## REFERENCES


[1] Cisco, "Global mobile data traffic forecast update, 2017–2022," *Cisco visual networking index*, vol. 2017, p. 2022, 2019.

[2] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 379–392.

[3] M. Adler, R. K. Sitaraman, and H. Venkataramani, "Algorithms for optimizing the bandwidth cost of content delivery," *Computer Networks*, vol. 55, no. 18, pp. 4007–4020, 2011.

[4] N. Bartolini, E. Casalicchio, and S. Tucci, "A walk through content delivery networks," in *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Springer, 2003, pp. 1–25.

[5] D. S. Berger, "Towards lightweight and robust machine learning for cdn caching," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018, pp. 134–140.

[6] S. Hasan, S. Gorinsky, C. Dovrolis, and R. K. Sitaraman, "Trade-offs in optimizing the cache deployments of cdns," in *IEEE INFOCOM 2014-IEEE conference on computer communications*. IEEE, 2014, pp. 460–468.

[7] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 167–181.

[8] K. Mokhtarian and H.-A. Jacobsen, "Caching in video cdns: Building strong lines of defense," in *Proceedings of the ninth European conference on computer systems*, 2014, pp. 1–13.

[9] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11-16, pp. 1203–1213, 1999.

[10] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 124, 2004.

[11] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 483–498.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[13] E. ROCCA, "Running wikipedia.org," *https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf*, June 2016.

[14] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, pp. 78–117, 1970.

[15] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–14.

[16] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield, "Characterizing storage workloads with counter stacks," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 335–349.

[17] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 423–432.

[18] F. Olken, "E cient methods for calculating the success function of xed space replacement policies. master's thesis," *University of California at Berkeley, Berkeley, CA*, 1981.

[19] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 95–110.

[20] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "Parda: A fast parallel reuse distance analysis algorithm," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 1284–1294.

[21] A. Appleby, "Murmurhash 2.0," 2008.

[22] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 177–188, 2004.

[23] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, E. Witchel *et al.*, "Learning relaxed belady for content distribution network caching," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 529–544.

[24] Y. Zhang, "Openresty," *https://openresty.org/en/*, 2011.

[25] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *Fast*, vol. 3, no. 2003, 2003, pp. 115–130.

[26] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[27] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[28] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "Lama: Optimized locality-aware memory allocation for key-value cache," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 57–69.

[29] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 195–212.

[30] J. Hwang and T. Wood, "Adaptive performance-aware distributed memory caching," in *10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 33–43.

[31] A. O. Al-Abbasi and V. Aggarwal, "Edgecache: An optimized algorithm for cdn-based over-the-top video streaming services," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 202–207.

[32] G. Tang, K. Wu, and R. Brunner, "Rethinking cdn design with distributee time-varying traffic demands," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[33] Y. Zhang, C. Gao, Y. Guo, K. Bian, X. Jin, Z. Yang, L. Song, J. Cheng, H. Tuo, and X. Li, "Proactive video push for optimizing bandwidth consumption in hybrid cdn-p2p vod systems," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 2555–2563.

[34] X. Jiang and J. Bi, "ncdn: Cdn enhanced with ndn," in *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2014, pp. 440–445.

[35] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 495–513.