# Cache on Track (CoT): Decentralized Elastic Caches for Cloud Environments

Victor Zakhary, Lawrence Lim, Divyakant Agrawal, Amr El Abbadi

UC Santa Barbara
Santa Barbara, California
victorzakhary,lawrenceklim,divyagrawal,elabbadi@ucsb.edu

## ABSTRACT

Distributed caches are widely deployed to serve social networks and web applications at billion-user scales. This paper presents *Cache-on-Track* (CoT), a decentralized, elastic, and predictive caching framework for cloud environments. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve skewed workloads with small update percentage. Small front-end caches are mainly used to mitigate the load-imbalance across servers in the distributed caching layer. Front-end servers use a heavy hitter tracking algorithm to continuously track the top-k hot keys. CoT dynamically caches the top-C hot keys out of the tracked keys. CoT's main advantage over other replacement policies is its ability to dynamically adapt its tracker and cache sizes in response to workload distribution changes. Our experiments show that CoT's replacement policy consistently outperforms the hit-rates of LRU, LFU, and ARC for the same cache size on different skewed workloads. Also, CoT slightly outperforms the hit-rate of LRU-2 when both policies are configured with the same tracking (history) size. CoT achieves server size load-balance with 50% to 93.75% less front-end cache in comparison to other replacement policies. Finally, experiments show that CoT's resizing algorithm successfully auto-configures the tracker and cache sizes to achieve back-end load-balance in the presence of workload distribution changes.

## 1 INTRODUCTION

Social networks, the web, and mobile applications have attracted hundreds of millions of users who need to be served in timely personalized way [9]. To enable this real-time experience, the underlying storage systems have to provide efficient, scalable, and highly available access to big data.

Figure 1 presents a typical web and social network system deployment [9] where user-data is stored in a distributed back-end storage layer in the cloud. The back-end storage layer consists of a distributed in-memory caching layer deployed on top of a distributed persistent storage layer. The caching layer aims to improve the request latency and system throughput and to alleviate the load on the persistent storage layer at scale [44]. Distributed caching systems such as Memcached [3] and Redis [4] are widely adopted by cloud service providers such as Amazon ElastiCache [1] and Azure Redis Cache [2]. As shown in Figure 1, hundreds of millions of end-users send streams of page-load and page-update requests to thousands of stateless front-end servers. These front-end servers are either deployed in the same core datacenter as the back-end storage layer or distributed among other core and edge datacenters near end-users. Each end-user request results in hundreds of data object lookups and updates served from the back-end storage layer. According to Facebook
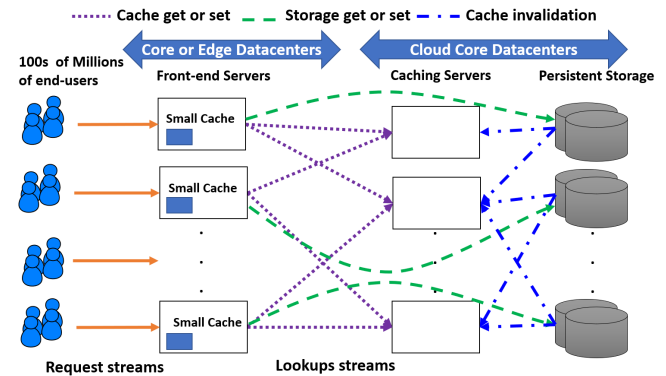
**Figure 1: A typical web and social network system deployment**

Tao [9], 99.8% of the accesses are reads and 0.2% of them are writes. Therefore, the storage system has to be **read optimized** to efficiently handle end-user requests at scale.

Redis and Memcached use consistent hashing [30] to distribute keys among several caching servers. Although consistent hashing ensures a fair distribution of the number of keys assigned to each caching shard, it does not consider the workload per key in the assignment process. Real-world workloads are typically skewed with few keys being significantly hotter than other keys [25]. This skew causes load-imbalance among caching servers.

Load imbalance in the caching layer can have significant impact on the overall application performance. In particular, it may cause drastic increases in the latency of operations at the tail end of the access frequency distribution [24]. In addition, the average throughput decreases and the average latency increases when the workload skew increases [11]. This increase in the average and tail latency is amplified for real workloads when operations are executed in chains of dependent data objects. A single page-load results in retrieving hundreds of objects in multiple rounds of data fetching operations [9, 38]. Finally, solutions that equally overprovision the caching layer resources to handle the most loaded caching server suffer from resource under-utilization in the least loaded caching servers.

In this paper, we propose Cache-on-Track (CoT); a decentralized, elastic, and predictive heavy hitter caching at front-end servers. CoT proposes a new cache replacement policy specifically tailored for small front-end caches that serve skewed workloads with small update percentage. CoT uses a small front-end cache to solve back-end load-imbalance as introduced in [20]. However, CoT does not assume perfect caching at the front-end. CoT uses the space saving algorithm [37] to track the top-k *heavy hitters*. The tracking information allows CoT to *cache* the exact top-C hot keys out of the approximate top-k tracked keys preventing cold and noisy keys from the long tail to replace hot keys in the cache. CoT is decentralized in the sense that each front-end independently determines its hot key set based on the key access

distribution served at this specific front-end. This allows CoT to address back-end load-imbalance without introducing single points of failure or bottlenecks that typically come with centralized solutions. In addition, this allows CoT to scale to thousands of front-end servers, a common requirement of social network and modern web applications. Unlike traditional replacement policies, CoT is elastic in the sense that each front-end uses its local load information to monitor its contribution to the back-end load-imbalance. Each front-end elastically adjusts its tracker and cache sizes to reduce the load-imbalance caused by this front-end. In the presence of workload changes, CoT dynamically adjusts front-end tracker to cache ratio in addition to both the tracker and cache sizes to eliminate any back-end load-imbalance.

In traditional architectures, memory sizes are static and caching algorithms strive to achieve the best usage of all the available resources. However, in cloud settings where there are theoretically infinite memory and processing resources and cloud instance migration is the norm, cloud end-users aim to achieve their SLOs while reducing the required cloud resources and thus decreasing their monetary deployment costs. CoT's main goal is to reduce the necessary front-end cache size *independently* at each front-end to eliminate server-side load-imbalance. In addition, CoT strives to dynamically find this minimum front-end cache size as the workload distribution changes. Reducing front-end cache size is crucial for the following reasons: 1) it reduces the monetary cost of deploying front-end caches. For this, we quote David Lomet in his recent works [33–35] where he shows that cost/performance is usually more important than sheer performance: *"the argument here is not that there is insufficient main memory to hold the data, but that there is a less costly way to manage data."*. 2) In the presence of data updates and when data consistency is a requirement, increasing front-end cache sizes significantly increases the cost of the data consistency management technique. Note that social networks and modern web applications run on thousands of front-end servers. Increasing front-end cache size not only multiplies the cost of deploying bigger cache by the number of front-end servers, but also increases several costs in the consistency management pipeline including a) the cost of tracking key incarnations in different front-end servers and b) the network and processing costs to propagate updates to front-end servers. 3) Since the workload is skewed, our experiments clearly demonstrate that the relative benefit of adding more front-end cache-lines, measured by the average cache-hits per cache-line and back-end load-imbalance reduction, drastically decreases as front-end cache sizes increase.

CoT's resizing algorithm dynamically increases or decreases front-end allocated memory in response to dynamic workload changes. CoT's dynamic resizing algorithm is valuable in different cloud settings where all front-end servers are deployed in the same datacenter or in different datacenters at the *edge*. These front-end servers obtain workloads of the same dynamically evolving distributions or different distributions. In particular, CoT aims to capture local trends from each individual front-end server perspective. In social network applications, front-end servers that serve different geographical regions might experience different key access distributions and different local trends (e.g., #miami vs. #ny).

We summarize our contributions in this paper as follows.

- Design and implement Cache-on-Track (CoT), a front-end cache replacement policy specifically tailored for small caches that serve skewed workloads with small update percentages.
- Design and implement CoT's resizing algorithm. The resizing algorithm dynamically minimizes the required front-end cache size to achieve back-end load-balance. CoT's built-in elasticity is a key novel advantage over other replacement policies.
- Evaluate CoT's replacement policy hit-rates to the hit-rates of traditional as well as state-of-the-art replacement policies, namely, LFU, LRU, ARC, and LRU-2. In addition, experimentally show that CoT achieves server size load-balance for different workload with **50% to 93.75%** less front-end cache in comparison to other replacement policies.
- Experimentally evaluate CoT's resizing algorithm showing that CoT successfully adjust its tracker and cache sizes in response to workload distribution changes.
- Report a *bug* at YCSB's [15] ScrambledZipfian workload generator. This generator generates workloads that are significantly less-skewed than the promised Zipfian distribution.

The rest of the paper is organized as follows. The related work is discussed in Section 2. In Section 3, the data model is explained. Section 4 further motivates CoT by presenting the main advantages and limitations of using LRU, LFU, ARC, and LRU-k caches at the front-end. We present the details of CoT in Section 5. In Section 6, we evaluate the performance and the overhead of CoT and the paper is concluded in Section 7.

## 2 RELATED WORK

Distributed caches are widely deployed to serve social networks and the web at scale [9, 38, 44]. Load-imbalancing among caching servers negatively affects the overall performance of the caching layer. Therefore, significant research has addressed the load-imbalacing problem from different angles. Solutions use different load-monitoring techniques (e.g., centralized tracking [6, 7, 26, 43], server-side tracking [11, 24], and client-side tracking [20, 28]). Based on the load-monitoring, different solutions redistribute keys among caching servers at different granularities. The following summarizes the related works under different categories.

**Centralized load-monitoring:** Slicer [7] and Centrifuge [6] are examples of centralized load-monitoring where a centralized control plane is separated from the data plane. Centrifuge uses consistent hashing to map keys to servers. However, Slicer propagates the key assignments from the control plane to the front-end servers. Slicer's control plane collects metadata about shard accesses and server workload. The control plane periodically runs an optimization algorithm that decides to redistribute, repartition, or replicate slices of the key space to achieve better back-end load-balance. Also, Slicer replicates the centralized control plane to achieve high availability and to solve the fault-tolerance problem in both Centrifuge [6] and in [11]. CoT is complementary to systems like Slicer and Centrifuge since CoT operates on a *fine-grain key* level at front-end servers while solutions like Slicer [7] operate on coarser grain slices or shards at the caching servers. Our goal is to cache heavy hitters at front-end servers to reduce key skew at back-end caching servers and hence, reduce Slicer's initiated re-configurations. Also, CoT is distributed and front-end driven that does not require any system component to develop a

global view of the workload. This allows CoT to scale to thousands of front-end servers without introducing any centralized points of failure or bottlenecks.

**Server side load-monitoring:** Another approach to load-monitoring is to distribute the load-monitoring among the caching shard servers. In [24], each caching server tracks its own hotspots. When the hotness of a key surpasses a certain threshold, this key is replicated to $\gamma$ caching servers and the replication decision is broadcast to all the front-end servers. Any further accesses on this hot key shall be equally distributed among these $\gamma$ servers. Cheng et al. [11] extend the work in [24] to allow moving coarse-grain key cachelets (shards) among threads and caching servers. Our approach reduces the need for server side load-monitoring. Instead, load-monitoring happens at the edge. This allows individual front-end servers to independently identify their local trends and cache them without adding the monitoring overhead to the caching layer, a critical layer for the performance of the overall system.

**Client side load-monitoring:** Fan et al. [20] use a distributed front-end load-monitoring approach. This approach shows that adding a small perfect cache in the front-end servers has significant impact on solving the back-end load-imbalance. Fan et al. *theoretically* show through analysis and simulation that a small *perfect cache* at each front-end solves the back-end load-imbalance problem. Following [20], Gavrielatos et al. [21] propose *symmetric caching* to track and cache the hot-most items at every front-end server. Symmetric caching assumes that all front-end servers obtain the same access distribution and hence statically allocates the same cache size to all front-end servers. However, different front-end servers might serve different geographical regions and therefore observe different access distributions. CoT discovers the workload access distribution independently at each front-end server and adjusts the cache size to achieve some target load-balance among caching servers. NetCache [28] uses programmable switches to implement heavy hitter tracking and caching at the network level. Like symmetric caching, NetCache assumes a fixed cache size for different access distributions. To the best of our knowledge, CoT is the first front-end caching algorithm that exploits the cloud elasticity allowing each front-end server to independently reduce the necessary required front-end cache memory to achieve back-end load-balance.

Other works in the literature focus on maximizing cache hit rates for fixed memory sizes. Cidon et al. [12, 13] redistribute available memory among memory slabs to maximize memory utilization and reduce cache miss rates. Fan et al. [19] use cuckoo hashing [41] to increase memory utilization. Lim et al. [32] increase memory locality by assigning requests that access the same data item to the same CPU. Bechmann et al. [8] propose Least Hit Density (LHD), a new cache replacement policy. LHD predicts the expected hit density of each object and evicts the object with the lowest hit density. LHD aims to evict objects that contribute low hit rates with respect to the cache space they occupy. Unlike these works, CoT does not assume a static cache size. In contrast, CoT maximizes the hit rate of the available cache and exploits the cloud elasticity allowing front-end servers to independently expand or shrink their cache memory sizes as needed.

## 3 DATA MODEL

We assume a typical key/value store interface between the front-end servers and the storage layer. The API consists of the following calls:

- $v = get(k)$ retrieves value $v$ corresponding to key $k$.
- $set(k, v)$ assigns value $v$ to key $k$. $set(k, null)$ to delete $k$.

Front-end servers use *consistent hashing* [30] to locate keys in the caching layer. Consistent hashing solves the *key discovery* problem and reduces key churn when a caching server is added to or removed from the caching layer. We extend this model by adding an additional layer in the cache hierarchy. As shown in Figure 1, each front-end server maintains a small cache of its hot keys. This cache is populated according to the accesses that are served by each front-end server.

We assume a client driven caching protocol similar to the protocol implemented by **Memcached** [3]. A cache client library is deployed in the front-end servers. *Get* requests are initially attempted to be served from the local cache. If the requested key is in the local cache, the *value* is returned and the request is marked as served. Otherwise, a *null* value is returned and the front-end has to request this key from the caching layer **at the back-end storage layer**. If the key is cached in the caching layer, its value is returned to the front-end. Otherwise, a *null* value is returned and the front-end has to request this key from the persistent storage layer and upon receiving the corresponding value, the front-end inserts the value in its front-end local cache and in the server-side caching layer as well. As in [38], a *set*, or an update, request invalidates the key in both the local cache and the caching layer. Updates are directly sent to the persistent storage, local values are set to null, and delete requests are sent to the caching layer to invalidate the updated keys. The Memcached client driven approach allows the deployment of a *stateless* caching layer. As requests are driven by the client, a caching server does not need to maintain the state of any request. This simplifies scaling and tolerating failures at the caching layer. Although, we adopt the Memcached client driven request handling protocol, our model works as well with write-through request handling protocols.

Our model is not tied to any replica consistency model. Each key can have multiple incarnations in the storage layer and the caching layer. Updates can be synchronously propagated if *strong consistency* guarantees are needed or asynchronously propagated if *weak consistency* guarantees suffice. Since the assumed workload is mostly read requests with very few update requests, we do not address consistency of updates in this paper. Achieving strong consistency guarantees among replicas of the same object has been widely studied in [11, 24]. Ghandeharizadeh et al. [22, 23] propose several complementary techniques to CoT to deal with consistency in the presence of updates and configuration changes. These techniques can be adopted in our model according to the application requirements. We understand that deploying an additional vertical layer of cache increases potential data inconsistencies and hence increases update propagation and synchronization overheads. Therefore, our goal in this paper is to reduce the front-end cache size in order to limit the inconsistencies and the synchronization overheads that result from deploying front-end caches, while maximizing their benefits on back-end load-imbalance.

## 4 FRONT-END CACHE ALTERNATIVES

Fan et al. [20] show that a small perfect cache in the front-end servers has big impact on the caching layer load-balance. A **perfect cache** of $C$ cache-lines is defined such that accesses to the $C$ hot-most keys always hit the cache while accesses to any other keys always miss the cache. The perfect caching assumption is impractical especially for dynamically changing and evolving workloads. Several replacement policies have been developed to

approximate perfect caching for different workloads. This section discusses the workload assumptions and various client caching objectives. This is followed by a discussion of the advantages and limitations of common caching replacement policies.

**Workload assumptions:** Real-world workloads are typically skewed with few keys being significantly hotter than other keys. In this paper, we assume skewed mostly read workloads with periods of stability (where hot keys remain hot during these periods).

**Client caching objectives:** Front-end servers construct their perspective of the key hotness distribution based on the requests they serve. Front-end servers aim to achieve the following caching objectives:

- The cache replacement policy should prevent cold keys from replacing hotter keys in the cache.
- Front-end caches should adapt to the changes in the workload. In particular, front-end servers should have a way to retire hot keys that are no longer accessed. In addition, front-end caches should have a mechanism to expand or shrink their local caches in response to changes in workload distribution. For example, front-end servers that serve uniform access distributions should dynamically shrink their cache size to *zero* since caching is of no value in this situation. On the other hand, front-end servers that serve highly skewed Zipfian (e.g., s = 1.5) should dynamically expand their cache size to capture all the hot keys that cause load-imbalance among the back-end caching servers.

A popular policy for implementing client caching is the LRU replacement policy. Least Recently Used (LRU) costs O(1) per access and caches keys based on their recency of access. This may allow cold keys that are recently accessed to replace hotter cached keys. Also, LRU cannot distinguish well between frequently and infrequently accessed keys [31]. Alternatively, Least Frequently Used (LFU) can be used as a replacement policy. LFU costs $O(log(C))$ per access where $C$ is the cache size. LFU is typically implemented using a min-heap and allows cold keys to replace hotter keys at the top levels of the heap. Also, LFU cannot distinguish between old references and recent ones. This means that LFU cannot adapt to changes in workload. Both LRU and LFU are limited in their knowledge to the content of the cache and cannot develop a wider perspective about the hotness distribution outside of their static cache size.

Adaptive Replacement Cache (ARC) [36] tries to realize the benefits of both LRU and LFU policies by maintaining two caching lists: one for *recency* and one for *frequency*. ARC dynamically changes the number of cache-lines allocated for each list to either favor recency or frequency of access in response to workload changes. In addition, ARC uses shadow queues to track more keys beyond the cache size. This helps ARC to maintain a broader perspective of the access distribution beyond the cache size. ARC is designed to find the fine balance between recent and frequent accesses. As a result, ARC pays the cost of caching every new cold key in the recency list evicting a hot key from the frequency list. This cost is significant especially when the cache size is much smaller than the key space and the workload is skewed favoring frequency over recency.

LRU-k [39] tracks the last k accesses of each cached key, in addition to a pre-configured *manually* fixed size history that includes the access information of the recently evicted keys from the cache. New keys replace the cached key with the least recently $k^{th}$ access. The evicted key is moved to the history, which is typically implemented using LRU. LRU-k is a suitable strategy to mock perfect caching of periodically stable skewed workloads when its cache and history sizes are perfectly pre-configured for this specific workload. However, due to the lack of LRU-k's dynamic resizing and elasticity of both its cache and history sizes, we choose to introduce CoT that is designed with native resizing and elasticity functionality. This functionality allows CoT to adapt its cache and tracker sizes in response to workload changes.

## 5 CACHE ON TRACK (COT)

Front-end caches serve two main purposes: *1) decrease the load on the back-end caching layer* and *2) reduce the load-imbalance among the back-end caching servers*. CoT focuses on the latter goal and considers back-end load reduction a complementary side effect. CoT's design philosophy is to track more keys beyond the cache size. This tracking serves as a filter that prevents cold keys from populating the small cache and therefore, only hot keys can populate the cache. In addition, the tracker and the cache are dynamically and adaptively resized to ensure that the load served by the back-end layer follows a load-balance target.

The idea of tracking more keys beyond the cache size has been widely used in replacement policies such as 2Q [29], MQ [45], LRU-k [39, 40], and ARC [36]. Both 2Q and MQ use multiple LRU queues to overcome the weaknesses of LRU of allowing cold keys to replace warmer keys in the cache. All these policies are desgined for fixed memory size environments. However, in a cloud environment where elastic resources can be requested on-demand, a new cache replacement policy is needed to take advantage of this elasticity.

CoT presents a new cache replacement policy that uses a *shadow heap* to track more keys beyond the cache size. Previous works have established the efficiency of heaps in tracking frequent items [37]. In this section, we explain how CoT uses tracking beyond the cache size to achieve the caching objectives listed in Section 4. In particular, CoT answers the following questions: 1) *how to prevent cold keys from replacing hotter keys in the cache?*, 2) *how to reduce the required front-end cache size that achieves lookup load-balance?*, 3) *how to adaptively resize the cache in response to changes in the workload distribution?* and finally 4) *how to dynamically retire old heavy hitters?*.

### 5.1 Notation

The key space, denoted by $S$, is assumed to be large in the scale of trillions of keys. Each front-end server maintains a cache of size $C <<< S$. The set of cached keys is denoted by $S_c$. To capture the top-$C$ hottest keys, each front-end server tracks $K > C$ keys. The set of tracked key is denoted by $S_k$. Front-end servers cache the top-$C$ hottest keys where $S_c \subset S_k$. A key hotness $h_k$ is determined using the dual cost model introduced in [18]. In this model, read accesses increase a key hotness by a read weight $r_w$ while update accesses decrease it by an update weight $u_w$. As update accesses cause cache invalidations. Therefore, frequently updated keys should not be cached and thus an update access decreases a key's hotness. For each tracked key, the read count $k.r_c$ and the update count $k.u_c$ are maintained to capture the number of read and update accesses of this key. Equation 1 shows how the hotness of key $k$ is calculated. We use $r_w = u_w = 1$ in our experiments.

$$h_k = k.r_c \times r_w - k.u_c \times u_w \tag{1}$$

$h_{min}$ refers to the minimum key hotness in the cache. $h_{min}$ splits the tracked keys into *two* subsets: 1) the set of cached keys (also tracked) $S_c$ of size $C$ and 2) the set of tracked but not cached

| $S$ | key space |
|---|---|
| $K$ | number of tracked keys at the front-end |
| $C$ | number of cached keys at the front-end |
| $h_k$ | hotness of a key $k$ |
| $k.r_c$ | read count of a key k |
| $k.u_c$ | update count of a key k |
| $r_w$ | the weight of a read operation |
| $u_w$ | the weight of an update operation |
| $h_{min}$ | the minimum hotness of keys in the cache |
| $S_k$ | the set of all tracked keys |
| $S_c$ | the set of cached keys (these keys are also tracked) |
| $S_{k-c}$ | the set of tracked but not cached keys |
| $I_c$ | the current local lookup load-imbalance |
| $I_t$ | the target lookup load-imbalance |
| $\alpha$ | the average hit-rate per cache-line in an epoch |
| $E$ | Epoch: a configurable number of accesses |

**Table 1: Summary of notation.**

keys $S_{k-c}$ of size $K - C$. The current local load-imbalance among caching servers lookup load is denoted by $I_c$. $I_c$ is a local variable at each front-end that determines the current contribution of this front-end to the back-end load-imbalance. $I_c$ is defined as the workload ratio between the most loaded back-end server and the least loaded back-end server as observed at a front-end server. For example, if a front-end server sends, during an epoch, a maximum of 5K key lookups to some back-end server and, during the same epoch, a minimum of 1K key lookups to another back-end server then $I_c$, at this front-end, equals 5. $I_t$ is the target load-imbalance among the caching servers. $I_t$ is the only input parameter set by the system administrator and is used by front-end servers to dynamically adjust their cache and tracker sizes. Ideally $I_t$ should be set close to 1. $I_t = 1.1$ means that back-end load-balance is achieved if the most loaded server observes at most 10% more key lookups than the least loaded server. Finally, we define another **local auto-adjusted** parameter $\alpha$. $\alpha$ is the average hits per cache-line and it determines the quality of the cached keys. A cache-line, or entry, contains one cached key and its corresponding value and $\alpha$ determines the average hits per cache-line during an epoch. For example, if a cache consists of 10 cache-lines and they cumulatively observe 2000 hits in an epoch, we consider $\alpha$ to be 200. $\alpha$ helps detect changes in workload and adjust the cache size accordingly. Note that CoT automatically infers the value of $\alpha$ based on the observed workload. Hence, the system administrator does not need to set the value of $\alpha$. $E$ is an epoch parameter defined by a configurable number of accesses. CoT runs its dynamic resizing algorithm every $E$ accesses. Table 1 summarizes the notation.

## 5.2 Space-Saving Tracking Algorithm

CoT uses the *space-saving* algorithm introduced in [37] to track the key hotness at front-end servers. Space-saving uses a min-heap to order keys based on their hotness and a hashmap to lookup keys in the tracker in $O(1)$. The space-saving algorithm is shown in Algorithm 1. If the accessed key $k$ is not in the tracker (Line 1), it replaces the key with minimum hotness at the root of the min-heap (Lines 2, 3, and 4). The algorithm gives the newly added key the benefit of doubt and assigns it the hotness of the replaced key. As a result, the newly added key gets the opportunity to survive immediate replacement in the tracker. Whether the accessed key $k$ was in the tracker or is newly added to the tracker, the hotness of the key is updated based on the

access type according to Equation 1 (Line 6) and the heap is accordingly adjusted (Line 7).

---

**Algorithm 1** The space-saving algorithm: track_key( key k, access_type t).

---

**State:** $S_k$: keys in the tracker.
**Input:** (key k, access_type t)

1: **if** $k \notin S_k$ **then**
2:     let $k'$ be the root of the min-heap
3:     replace $k'$ with $k$
4:     $h_k := h_{k'}$
5: **end if**
6: $h_k := $ update_hotness(k, t)
7: adjust_heap(k)
8: return $h_k$

---

## 5.3 CoT: Cache Replacement Policy

CoT's tracker captures the approximate top $K$ hot keys. Each front-end server should cache the exact top $C$ keys out of the tracked $K$ keys where $C < K$. The exactness of the top $C$ cached keys is considered with respect to the approximation of the top $K$ tracked keys. Caching the exact top $C$ keys prevents cold and noisy keys from replacing hotter keys in the cache and achieves the first caching objective. To determine the exact top $C$ keys, CoT maintains a cache of size C in a min-heap structure. Cached keys are partially ordered in the min-heap based on their hotness. The root of the cache min-heap gives the minimum hotness, $h_{min}$, among the cached keys. $h_{min}$ splits the tracked keys into *two unordered* subsets $S_c$ and $S_{k-c}$ such that:

- $|S_c| = C$ and $\forall_{x \in S_c} h_x \geq h_{min}$
- $|S_{k-c}| = K - C$ and $\forall_{x \in S_{k-c}} h_x < h_{min}$
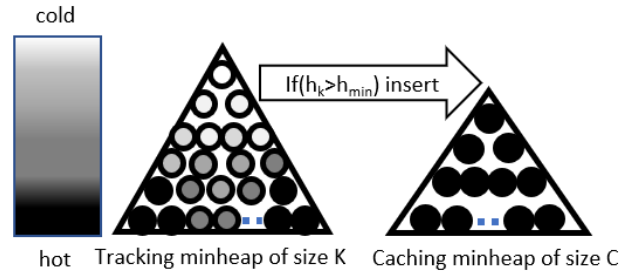


**Figure 2: CoT: a key is inserted to the cache if its hotness exceeds the minimum hotness of the cached keys.**

For every key access, the hotness information of the accessed key is updated in the tracker. If the accessed key is cached, its hotness information is updated in the cache as well. However, if the accessed key is not cached, its hotness is compared against $h_{min}$. As shown in Figure 2, the accessed key is inserted into the cache only if its hotness exceeds $h_{min}$. Algorithm 2 explains the details of CoT's cache replacement algorithm.

For every key access, the *track_key* function of Algorithm 1 is called (Line 1) to update the tracking information and the hotness of the accessed key. Then, a key access is served from the local cache only if the key is in the cache (Lines 3). Otherwise, the access is served from the caching server (Line 5). Serving an access from the local cache implicitly updates the accessed key hotness and location in the cache min-heap. If the accessed key

**Algorithm 2** CoT's caching algorithm

**State:** $S_k$: keys in the tracker and $S_c$: keys in the cache.
**Input:** (key k, access_type t)

1: $h_k$ = track_key(k, t) as in Algorithm 1
2: **if** $k \in S_c$ **then**
3:     let v = access($S_c$, k) // local cache access
4: **else**
5:     let v = server_access(k) // caching server access
6:     **if** $h_k > h_{min}$ **then**
7:         insert($S_c$, k, v) // local cache insert
8:     **end if**
9: **end if**
10: return v

is not cached, its hotness is compared against $h_{min}$ (Line 6). The accessed key is inserted to the local cache if its hotness exceeds $h_{min}$ (Line 7). This happens only if there is a tracked but not cached key that is hotter than one of the cached keys. Keys are inserted to the cache together with their tracked hotness information. Inserting keys into the cache follows the LFU replacement policy. This implies that a local cache insert (Line 7) would result in the replacement of the coldest key in the cache (the root of the cache heap) if the local cache is full.

## 5.4 CoT: Adaptive Cache Resizing

This section explains CoT's resizing algorithm. This algorithm reduces the necessary front-end cache size that achieves back-end lookup load-balance. In addition, this algorithm dynamically expands or shrinks CoT's tracker and cache sizes when the served workload changes. Also, this algorithm detects changes in the set of hot keys and retires old hot keys that are not hot any more. As explained in Section 1, reducing the front-end cache size decreases the front-end cache monetary cost, limits the overheads of data consistency management techniques, and maximizes the benefit of front-end caches measured by the average cache-hits per cache-line and back-end load-imbalance reduction.
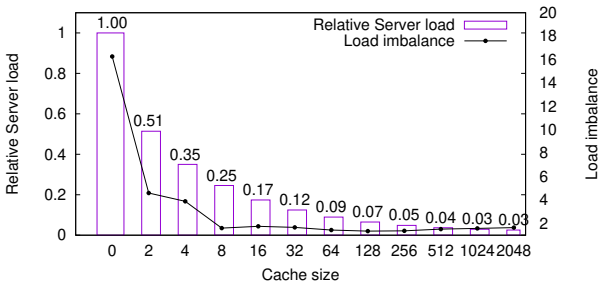


**Figure 3: Reduction in relative server load and load-imbalance among caching servers as front-end cache size increases.**

**The Need for Cache Resizing:** Figure 3 experimentally shows the effect of increasing the front-end cache size on both back-end *load-imbalance reduction* and decreasing the workload at the back-end. In this experiment, 8 memcached shards are deployed to serve back-end lookups and 20 clients send lookup requests following a significantly skewed Zipfian distribution (s = 1.5). The size of the key space is 1 million and the total number of lookups is 10 millions. The front-end cache size at each client is varied from 0 cachelines (no cache) to 2048 cachelines (≈0.2% of the key space). Front-end caches use CoT's replacement policy and a ratio of 4:1 is maintained between CoT's tracker

size and CoT's cache size. We define back-end load-imbalance as the workload ratio between the most loaded server and the least loaded server. The target load-imbalance $I_t$ is set to 1.5. As shown in Figure 3, processing all the lookups from the back-end caching servers (front-end cache size = 0) leads to a significant load-imbalance of 16.26 among the caching servers. This means that the most loaded caching server receives 16.26 times the number of lookup requests received by the least loaded caching server. As the front-end cache size increases, the server size load-imbalance drastically decreases. As shown, a front-end cache of size 64 cache lines at each client reduces the load-imbalance to 1.44 (an order of magnitude less load-imbalance across the caching servers) achieving the target load-imbalance $I_t = 1.5$. Increasing the front-end cache size beyond 64 cache lines only reduces the back-end aggregated load but not the back-end load-imbalance. The *relative server load* is calculated by comparing the server load for a given front-end cache size to the server load when there is no front-end caching (cache size = 0). Figure 3 demonstrates the reduction in the relative server load as the front-end cache size increases. However, the benefit of doubling the cache size proportionally decays with the key hotness distribution. As shown in Figure 3, the first 64 cachelines reduce the relative server load by 91% while the second 64 cachelines reduce the relative server load by only 2% more.

The failure of the "one size fits all" design strategy suggests that statically allocating fixed cache and tracker sizes to all front-end servers is not ideal. Each front-end server should independently and adaptively be configured according to the key access distribution it serves. Also, changes in workloads can alter the key access distribution, the skew level, or the set of hot keys. For example, social networks and web front-end servers that serve different geographical regions might experience different key access distributions and different local trends (e.g., #miami vs. #ny). Therefore, CoT's cache resizing algorithm learns the key access distribution independently at each front-end and dynamically resizes the cache and the tracker to achieve lookup load-imbalance target $I_t$. CoT is designed to reduce the front-end cache size that achieves $I_t$. Any increase in the front-end cache size beyond CoT's recommendation mainly decreases back-end load and should consider other conflicting parameters such as the additional cost of the memory cost, the cost of updates and maintaining the additional cached keys, and the percentage of back-end load reduction that results from allocating additional front-end caches.

**Cache Resizing Algorithm (parameter configuration):** Front-end servers use CoT to minimize the cache size that achieves a target load-imbalance $I_t$. Initially, front-end servers are configured with no front-end caches. The system administrator configures CoT by an input *target load-imbalance* parameter $I_t$ that determines the maximum tolerable imbalance between the most loaded and least loaded back-end caching servers. Afterwards, CoT expands both tracker and cache sizes until the current load-imbalance achieves the inequality $I_c \leq I_t$.

Algorithm 3 describes CoT's cache resizing algorithm. CoT divides the timeline into epochs and each epoch consists of $E$ accesses. Algorithm 3 is executed at the end of each epoch. The epoch size $E$ is proportional to the tracker size $K$ and is dynamically updated to guarantee that $E \geq K$ (Line 3). This condition helps ensure that CoT does not trigger consecutive resizes before the cache and the tracker are warmed up with keys. During each epoch, CoT tracks the number of lookups sent to every back-end caching server. In addition, CoT tracks the total number of cache

hits and tracker hits during this epoch. At the end of each epoch, CoT calculates the current load-imbalance $I_c$ as the ratio between the highest and the lowest load on back-end servers during this epoch. Also, CoT calculates the current average hit per cached key $\alpha_c$. $\alpha_c$ equals the total cache hits in the current epoch divided by the cache size. Similarly, CoT calculates the current average hit per tracked but not cache key $\alpha_{k-c}$. CoT compares $I_c$ to $I_t$ and decides on a resizing action as follows.

(1) $I_c > I_t$ and $\alpha_c \geq \alpha_{k-c}$ (Line 1), this means that the target load-imbalance is not achieved and cached keys observe more hits than the keys in the tracker. CoT follows the binary search algorithm in searching for the front-end cache size that achieves $I_t$. Therefore, CoT decides to double the front-end cache size (Line 2). As a result, CoT doubles the tracker size as well to maintain a tracker to cache size ratio of at least 2, $K \geq 2 \cdot C$ (Line 2). The cache and tracker sizes are doubled until either $I_t$ is achieved or a configurable upper limit cache size is hit. In addition, CoT uses a local variable $\alpha_t$ to capture the quality of the cached keys when $I_t$ is first achieved. Initially, $\alpha_t = 0$. CoT then sets $\alpha_t$ to the average hits per cache-line $\alpha_c$ during the current epoch (Line 4). In subsequent epochs, $\alpha_t$ is used to detect changes in workload.

(2) $I_c \leq I_t$ (Line 5), this means that the target load-imbalance has been achieved. However, changes in workload could alter the quality of the cached keys. Therefore, CoT uses $\alpha_t$ to detect and handle changes in workload in future epochs as explained below.

---

**Algorithm 3** CoT's elastic resizing algorithm.

**State:** $S_c$: keys in the cache, $S_k$: keys in the tracker, C: cache capacity, K: tracker capacity, $\alpha_c$: average hits per key in $S_c$ in the current epoch, $\alpha_{k-c}$: average hits per key in $S_{k-c}$ in the current epoch, $I_c$: current load-imbalance, and $\alpha_t$: target average hit per key

**Input:** $I_t$

1: **if** $I_c > I_t$ && $\alpha_c >= \alpha_{k-c}$ **then**
2:     resize($S_c$, $2 \times C$) , resize($S_k$, $2 \times K$)
3:     E := max (E, $K$)
4:     Let $\alpha_t = \alpha_c$
5: **else**
6:     **if** $\alpha_c < (1 - \epsilon).\alpha_t$ and $\alpha_{k-c} < (1 - \epsilon).\alpha_t$ **then**
7:         resize($S_c$, $\frac{C}{2}$),  resize($S_k$, $\frac{K}{2}$)
8:     **else if** $\alpha_c < (1 - \epsilon).\alpha_t$ and $\alpha_{k-c} > (1 - \epsilon).\alpha_t$ **then**
9:         half_life_time_decay()
10:    **end if**
11: **end if**

---

$\alpha_t$ is reset whenever the inequality $I_c \leq I_t$ is violated and Algorithm 3 expands cache and tracker sizes. Ideally, when the inequality $I_c \leq I_t$ holds, keys in the cache (the set $S_c$) achieve $\alpha_t$ hits per cache-line during every epoch while keys in the tracker but not in the cache (the set $S_{k-c}$) do not achieve $\alpha_t$. This happens because keys in the set $S_{k-c}$ are colder than keys in the set $S_c$. $\alpha_t$ represents a target hit-rate per cache-line for future epochs. Therefore, if keys in the cache do not meet the target $\alpha_t$ in a following epoch, this indicates that the quality of the cached keys has changed and an action needs to be taken as follows.

(1) Case 1: keys in $S_c$, on the average, do not achieve $\alpha_t$ hits per cacheline and keys in $S_{k-c}$ do not achieve $\alpha_t$ hits as

well (Line 6). This indicates that the quality of the cached keys decreased. In response. CoT shrinks both the cache and the tracker sizes (Line 7). If shrinking both cache and tracker sizes results in a violation of the inequality $I_c < I_t$, Algorithm 3 doubles both tracker and cache sizes in the following epoch and $\alpha_t$ is reset as a result. In Line 6, we compare the average hits per key in both $S_c$ and $S_{k-c}$ to $(1 - \epsilon) \cdot \alpha_t$ instead of $\alpha_t$. Note that $\epsilon$ is a small constant $<<< 1$ that is used to avoid unnecessary resizing actions due to insignificant statistical variations.

(2) Case 2: keys in $S_c$ do not achieve $\alpha_t$ while keys in $S_{k-c}$ achieve $\alpha_t$ (Line 8). This signals that the set of hot keys is changing and keys in $S_{k-c}$ are becoming hotter than keys in $S_c$. For this, CoT triggers a half-life time decaying algorithm that halves the hotness of all cached and tracked keys (Line 9). This decaying algorithm aims to forget old trends that are no longer hot to be cached (e.g., Gangnam style song). Different decaying algorithms have been developed in the literature [14, 16, 17]. Therefore, this paper only focuses on the resizing algorithm details without implementing a specific decaying algorithm.

(3) Case 3: keys in $S_c$ achieve $\alpha_t$ while keys in $S_{k-c}$ do not achieve $\alpha_t$. This means that the quality of the cached keys has not changed and therefore, CoT does not take any action. Similarly, if keys in both sets $S_c$ and $S_{k-c}$ achieve $\alpha_t$, CoT does not take any action as long as the inequality $I_c < I_t$ holds.

## 6 EXPERIMENTAL EVALUATION

This section evaluates both CoT's caching and adaptive resizing algorithms. We choose to compare CoT to traditional and widely used replacement policies like LRU and LFU. In addition, we compare CoT to both ARC [36] and LRU-k [39]. As stated in [36], ARC, in its online auto-configuration setting, achieves comparable performance to LRU-2 (which is the most responsive LRU-k) [39, 40], 2Q [29], LRFU [31], and LIRS [27] even when these policies are perfectly tuned offline. Also, ARC outperforms the online adaptive replacement policy MQ [45]. Therefore, we compare with ARC and LRU-2 as representatives of these different polices. Section 6.1 explains the experimental setup. First, we compare the hit rates of CoT's cache algorithm to LRU, LFU, ARC, and LRU-2 hit rates for different front-end cache sizes in Section 6.2. Then, we compare the required front-end cache size for each replacement policy to achieve a target back-end load-imbalance $I_t$ in Section 6.3. In Section 6.4, we provide an end-to-end evaluation of front-end caches comparing the end-to-end performance of CoT, LRU, LFU, ARC, and LRU-2 on different workloads with the configuration where no front-end cache is deployed. Finally, CoT's resizing algorithm is evaluated in Section 6.5.

### 6.1 Experiment Setup

We deploy 8 instances of memcached [3] on a small cluster of 4 caching servers (2 memcached instance per server). Each caching server has an Intel(R) Xeon(R) CPU E3-1235 (8MB Cache and 16GB RAM) with 4GB RAM dedicated to each memcached instance. Caching servers and clients run ubuntu 18.04 and connected to the same 10Gbps Ethernet network. No explicit network or OS optimization are used.

A dedicated client machine with Intel(R) Core(TM) i7-6700HQ CPU and 16GB of RAM is used to generate client workloads. The client machine executes multiple client threads to submit workloads to caching servers. Client threads use Spymemcached

2.11.4 [5], a Java-based memcached client, to communicate with memcached cluster. Spymemcached provides communication abstractions that distribute workload among caching servers using *consistent hashing* [30]. We slightly modified Spymemcached to monitor the workload per back-end server at each front-end. Client threads use Yahoo! Cloud Serving Benchmark (YCSB) [15] to generate workloads for the experiments. YCSB is a standard key/value store benchmarking framework. YCSB is used to generate key/value store requests such as *Get*, *Set*, and *Insert*. YCSB enables configuring the ratio between read (Get) and write (Set) accesses. Also, YCSB allows the generation of accesses that follow different access distributions. As YCSB is CPU-intensive, the client machine runs at most 20 client threads per machine to avoid contention among client threads. During our experiments, we realized that YCSB's ScrambledZipfian workload generator has a bug as it generates Zipfian workload distributions with significantly less skew than the skew level it is configured with. Therefore, we use YCSB's Zipfian generator instead of YCSB's ScrambledZipfian. Figure 4 shows the hits per key generated for the top 1024 out 1 million keys and 100 million samples of a zipfian 0.99 distribution from YCSB-Zipfian, YCSB-ScrambledZipfian, and theoretical zipfian generators. As shown, YCSB-Zipfian generator (CDF 0.512) is almost identical to the theoretical generator (CDF 0.504) while YCSB-ScrambledZipfian generator (CDF 0.30) has much less skew than the theoretical zipfian generator (40% less hits in the top 1024 keys than the theoretical distribution).
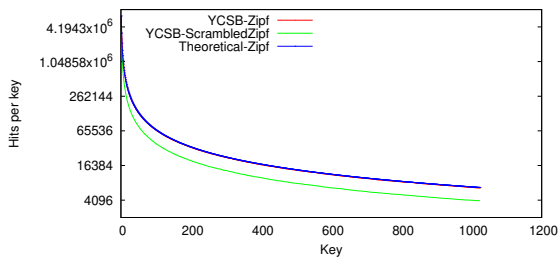


**Figure 4: YCSB-Zipf vs ScrambledZipf vs Theoretical-Zipf**

Our experiments use different variations of YCSB core workloads. Workloads consist of 1 million key/value pairs. Each key consists of a common prefix *"usertable:"* and a unique ID. We use a value size of 750 KB making a dataset of size 715GB. Experiments use read intensive workloads that follow Tao's [9] read-to-write ratio of 99.8% reads and 0.2% updates. Unless otherwise specified, experiments consist of 10 million key accesses sampled from different access distributions such as Zipfian (s = 0.90, 0.99, or 1.2) and uniform. Client threads submit access requests back-to-back. Each client thread can have only one outgoing request. Clients submit a new request as soon as they receive an acknowledgement for their outgoing request.

## 6.2 Hit Rate

The first experiment compares CoT's hit rate to LRU, LFU, ARC, and LRU-2 hit rates using equal cache sizes for all replacement policies. 20 client threads are provisioned on one client machine and each cache client thread maintains its own cache. **Configuration:** The cache size is varied from a very small cache of 2 cache-lines to 2048 cache-lines. The hit rate is compared using different Zipfian access distributions with skew parameter values s = 0.90, 0.99, and 1.2 as shown in Figures 5(a), 5(b), and 5(c) respectively. In addition, 14 days of Wikipedia's real

traces, collected by [42], are used to compare CoT to other replacement policies, including LHD [8], as shown in Figure 5(d). CoT's tracker to cache size ratio determines how many tracking nodes are used for every cache-line. CoT automatically detects the ideal tracker to cache ratio for any workload by fixing the cache size and doubling the tracker size until the observed hit-rate gains from increasing the tracker size are insignificant i.e., the observed hit-rate saturates. The tracker to cache size ratio decreases as the workload skew increases. A workload with high skew simplifies the task of distinguishing hot keys from cold keys and hence, CoT requires a smaller tracker size to successfully filter hot keys from cold keys. Note that LRU-2 is also configured with the same history to cache size as CoT's tracker to cache size. In this experiment, for each skew level, CoT's tracker to cache size ratio is varied as follows: 16:1 for Zipfian 0.9 and Wikipedia, 8:1 for Zipfian 0.99, and 4:1 for Zipfian 1.2. Note that CoT's tracker maintains only the meta-data of tracked keys. Each tracker node consists of a read counter and a write counter with 8 bytes of memory overhead per tracking node. In real-world workloads, value sizes vary from tens of KBs to few MBs. For example, Google's Bigtable [10] uses a value size of 64 KB. Therefore, a memory overhead of at most $\frac{1}{8}$ KB (16 tracker nodes * 8 bytes) per cache-line is negligible (0.2%).

In Figures 5, the *x-axis* represents the cache size expressed as the number of cache-lines. The *y-axis* represents the front-end cache hit rate (%) as a percentage of the total workload size. At each cache size, the cache hit rates are reported for LRU, LFU, ARC, LRU-2, and CoT cache replacement policies. In addition, TPC represents the theoretically calculated hit-rate from the Zipfian distribution CDF if a perfect cache with the same cache size is deployed. For each experiment, the TPC is configured with the same key space size N = (1 million keys), the same sample generated size (10 million samples), the same skew parameter s (e.g., 0.9, 0.99, 1.2) of the experiment, and k equals to the cache size (ranging from 2 – 2048). For example, a perfect cache of size 2 cache-lines stores the top-2 hot keys and hence any access to these 2 keys results in a cache hit while accesses to other keys result in cache misses. For Wikipedia traces, the TPC is the cumulative accesses of the top-C keys.

**Analysis:** As shown in Figure 5(a), CoT surpasses LRU, LFU, ARC, and LRU-2 hit rates at all cache sizes. In fact, CoT achieves almost similar hit-rate to the TPC hit-rate. In Figure 5(a), CoT outperforms TPC for some cache size which is counter intuitive. This happens as TPC is theoretically calculated using the Zipfian CDF while CoT's hit-rate is calculated out of YCSB's sampled distributions which are approximate distributions. In addition, CoT achieves higher hit-rates than both LRU and LFU with **75% less cache-lines**. As shown, CoT with 512 cache-lines achieves 10% more hits than both LRU and LFU with 2048 cache-lines. Also, CoT achieves higher hit rate than ARC using **50% less cache-lines**. In fact, CoT configured with 512 cache-lines achieves 2% more hits than ARC with 1024 cache-lines. Taking tracking memory overhead into account, CoT maintains a tracker to cache size ratio of 16:1 for this workload (Zipfian 0.9). This means that CoT adds an overhead of 128 bytes (16 tracking nodes * 8 bytes each) per cache-line. The percentage of CoT's tracking memory overhead decreases as the cache-line size increases. For example, CoT introduces a tracking overhead of 0.02% when the cache-line size is 750KB. Finally, CoT consistently achieves 8-10% higher hit-rate than LRU-2 configured with the same history and cache sizes as CoT's tracker and cache sizes.
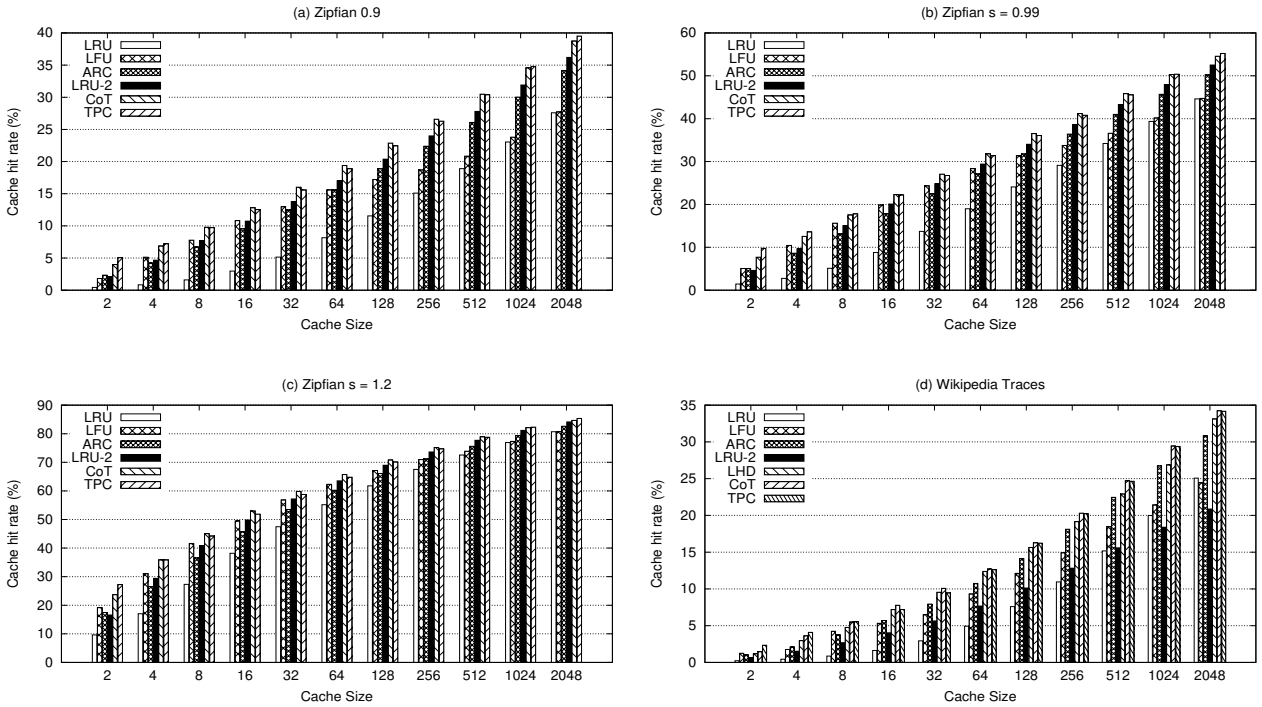
**Figure 5: Comparison of LRU, LFU, ARC, LRU-2, LHD, CoT and TPC's hit rates using various Zipfian and Wikipedia traces**

Similarly, as illustrated in Figures 5(b) and 5(c), CoT outpaces LRU, LFU, ARC, and LRU-2 hit rates at all different cache sizes. Figure 5(b) shows that a configuration of CoT using 512 cache-lines achieves 3% more hits than both configurations of LRU and LFU with 2048 cache-lines. Also, CoT consistently outperforms ARC's hit rate with 50% less cache-lines. Finally, CoT achieves 3-7% higher hit-rate than LRU-2 configured with the same history and cache sizes. Figures 5(b) and 5(c) highlight that increasing workload skew decreases the advantage of CoT. As workload skew increases, the ability of LRU, LFU, ARC, LRU-2 to distinguish between hot and cold keys increases and hence CoT's preeminence decreases.

In addition to YCSB synthetic traces, Wikipedia real traces show in Figure 5(d) that CoT significantly outperforms other replacement policies at every cache size. This Wikipedia real traces comparison includes LHD [8], a recently proposed caching policy specifically for cloud applications. As Figure 5(d) shows, CoT is the only replacement policy that achieves almost the same hit rate of the cumulative top-C keys beating LRU, LFU, ARC, LRU-2, and LHD by 23-84%, 14-51%, 9-42%, 37-58%, and 4-24% respectively for different cache sizes.

### 6.3 Back-End Load-Imbalance

In this section, we compare the required front-end cache sizes for different replacement policies to achieve a back-end load-imbalance target $I_t$. **Configuration:** Different skewed workloads are used, namely, Zipfian s = 0.9, s = 0.99, and s = 1.2. For each distribution, we first measure the back-end load-imbalance when no front-end cache is used. A back-end load-imbalance target $I_t$ is set to $I_t = 1.1$. This means that the back-end is load balanced if the most loaded back-end server processes at most 10% more lookups than the least loaded back-end server. We evaluate the back-end load-imbalance while increasing the front-end cache

size using different cache replacement policies, namely, LRU, LFU, ARC, LRU-2, and CoT. In this experiment, CoT uses the same tracker-to-cache size ratio as in Section 6.2. For each replacement policy, we report the minimum required number of cache-lines to achieve $I_t$.

| Dist. | Load-imbalance | Number of cache-lines to achieve $I_t = 1.1$ | | | | |
|---|---|---|---|---|---|---|
| | No cache | LRU | LFU | ARC | LRU-2 | CoT |
| Zipf 0.9 | 1.35 | 64 | 16 | 16 | 8 | 8 |
| Zipf 0.99 | 1.73 | 128 | 16 | 16 | 16 | 8 |
| Zipf 1.20 | 4.18 | 2048 | 2048 | 1024 | 1024 | 512 |

**Table 2: The minimum required number of cache-lines for different replacement policies to achieve a back-end load-imbalance target $I_t = 1.1$ for different workload distributions.**

**Analysis:** Table 2 summarizes the reported results for different distributions using LRU, LFU, ARC, LRU-2, and CoT replacement policies. For each distribution, the initial back-end load-imbalance is measured using no front-end cache. As shown, the initial load-imbalances for Zipf 0.9, Zipf 0.99, and Zipf 1.20 are 1.35, 1.73, and 4.18 respectively. For each distribution, the minimum required number of cache-lines for LRU, LFU, ARC, and CoT to achieve a target load-imbalance of $I_t = 1.1$ is reported. As shown, CoT requires **50% to 93.75% less cache-lines** than other replacement policies to achieve $I_t$. Since LRU-2 is configured with a history size equals to CoT's tracker size, LRU-2 requires the second least number of cache-lines to achieve $I_t$.

### 6.4 End-to-End Evaluation

In this section, we evaluate the effect of front-end caches using LRU, LFU, ARC, LRU-2, and CoT replacement policies on

the overall running time of different workloads. This experiment also demonstrates the overhead of front-end caches on the overall running time. **Configuration:** This experiment uses 3 different workload distributions, namely, uniform, Zipfian (s = 0.99), and Zipfian (s = 1.2) distributions as shown in Figure 6. For all the three workloads, each replacement policy is configured with 512 cache-lines. Also, CoT and LRU-2 maintain a tracker (history) to cache size ratio of 8:1 for Zipfian 0.99 and 4:1 for both Zipfian 1.2 and uniform distributions. In this experiment, a total of 1M accesses are sent to the caching servers by 20 client threads running on one client machine. Each experiment is executed 10 times and the average overall running time with 95% confidence intervals are reported.
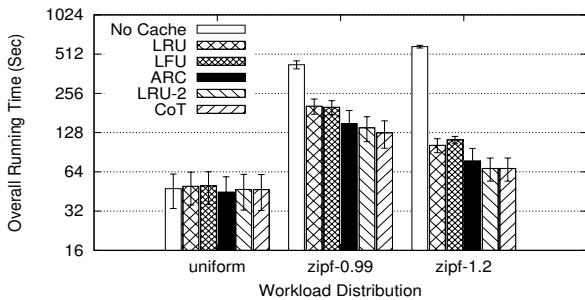


**Figure 6: Front-end caching effect on the end-to-end overall running time using different workload distributions.**

In this experiment, the front-end servers are allocated in the same cluster as the back-end servers. The average Round-Trip Time (RTT) between front-end machines and back-end machines is 244$\mu$s. This small RTT allows us to fairly measure the overhead of front-end caches by minimizing the performance advantages achieved by front-end cache hits. In real-world deployments where front-end servers are deployed in edge-datacenters and the RTT between front-end servers and back-end servers is in order of milliseconds, front-end caches achieve more significant performance gains.

**Analysis:** The uniform workload is used to measure the overhead of front-end caches. In a uniform workload, all keys in the key space are equally hot and front-end caches cannot take any advantage of workload skew to benefit some keys over others. Therefore, front-end caches only introduce the overhead of maintaining the cache without achieving any significant performance gains. As shown in Figure 6, there is no significant statistical difference between the overall run time when there is no front-end cache and when there is a small front-end cache with different replacement policies. Adding a small front-end cache does not incur run time overhead even for replacement policies that use a heap, e.g., LFU, LRU-2, and CoT.

The workloads Zipfian 0.99 and Zipfian 1.2 are used to show the advantage of front-end caches even when the network delays between front-end servers and back-end servers are minimal. As shown in Figure 6, workload skew results in significant overall running time overhead in the absence of front-end caches. This happens because the most loaded server introduces a performance bottleneck especially under thrashing (managing 20 connections, one from each client thread). As the load-imbalance increases, the effect of this bottleneck is worsen. Specifically, in Figure 6, the overall running time of Zipfian 0.99 and Zipfian 1.2 workloads are respectively 8.9x and 12.27x of the uniform workload when no front-end cache is deployed. Deploying a small

front-end cache of 512 cachelines significantly reduces the effect of back-end bottlenecks. Deploying a CoT small cache in the front-end results in 70% running time reduction for Zipfian 0.99 and 88% running time reduction for Zipfian 1.2 in comparison to having no front-end cache. Other replacement policies achieve running time reductions of 52% to 67% for Zipfian 0.99 and 80% to 88% for Zipfian 1.2. LRU-2 achieves the second best average overall running time after CoT with no significant statistical difference between the two policies. Since both policies use the same tracker (history) size, this again suggests that having a bigger tracker helps separate cold and noisy keys from hot keys. Since the ideal tracker to cache size ratio differs from one workload to another, having an automatic and dynamic way to configure this ratio at run-time while serving workload gives CoT a big leap over statically configured replacement policies.
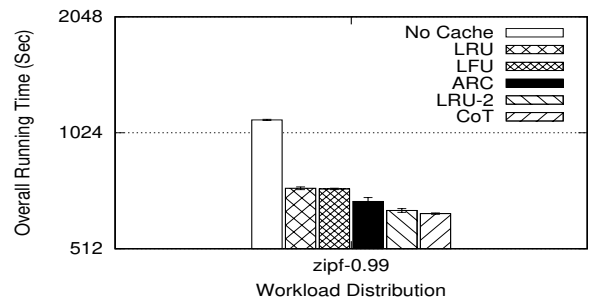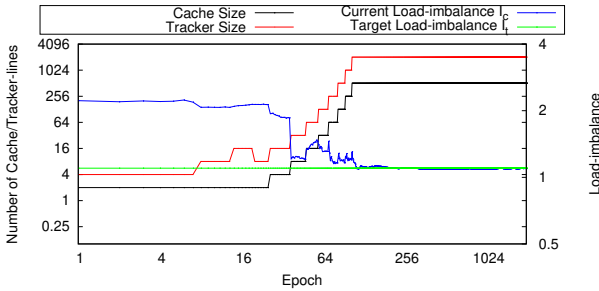


**Figure 7: Front-end caching effect on the end-to-end overall running time in Amazon EC2 cloud.**

Similar results have been observed in a cloud setup as well. We run the same end-to-end evaluation on Amazon EC2's cloud. 20 client threads run on a t2.xlarge machine in California to send 1M requests following zipfian 0.99 distribution to 8 m3.medium ElastiCache cluster in Oregon. Front-end cache and tracker sizes are similar to the local experiments configuration. The key space consists of 1M keys and each key has a value of 10KB (smaller values are used since network latency is large in comparison to local experiments). As shown in Figure 7, a small front-end cache of 512 cache-lines has significant performance gains in comparison to the setup where no front-end cache is deployed. Also, both CoT and LRU-2 outperform other front-end cache replacement policies. Also, CoT slightly outperforms LRU-2 achieving 2% performance gain on the average. The main advantage of CoT over LRU-2 is its ability to dynamically discover the ideal cache and tracker sizes that achieve backend load-balance as the workload distribution changes. The following section illustrates CoT's performance in response to workload changes.
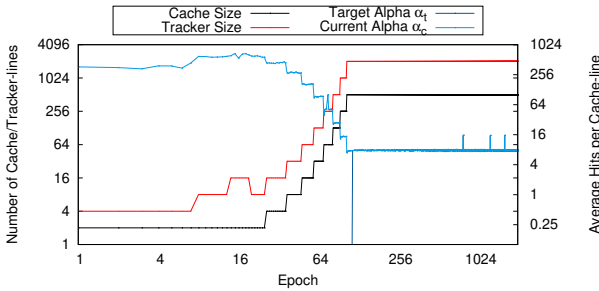
### 6.5 Adaptive Resizing

This section evaluates CoT's auto-configure and resizing algorithms. **Configuration:** *First*, we configure a front-end client that serves a Zipfian 1.2 workload with a tiny cache of size *two* cachelines and a tracker of size of *four* tracking entries. This experiment aims to show how CoT expands cache and tracker sizes to achieve a target load-imbalance $I_t$ as shown in Figure 8. After CoT reaches the cache size that achieves $I_t$, the average hit per cache-line $\alpha_t$ is recorded as explained in Algorithm 3. *Second*, we alter the workload distribution to uniform and monitors how CoT shrinks tracker and cache sizes in response to workload changes without violating the load-imbalance target $I_t$ in Figure 9. In both experiments, $I_t$ is set to 1.1 and the epoch

size is 5000 accesses. In both Figures 8a and 9a, the x-axis represents the epoch number, the left y-axis represents the number of tracker and cache lines, and the right y-axis represents the load-imbalance. The black and red lines represent cache and tracker sizes respectively with respect to the left y-axis. The blue and green lines represent the current load-imbalance and the target load-imbalance respectively with respect to the right y-axis. Same axis description applies for both Figures 8b and 9b except that the right y-axis represents the average hit per cache-line during each epoch. Also, the light blue and the dark blue lines represent the current average hit per cache-line and the target hit per cache-line at each epoch with respect to the right y-axis.



**(a) Changes in cache and tracker sizes and the current load-imbalance $I_c$ over epochs.**
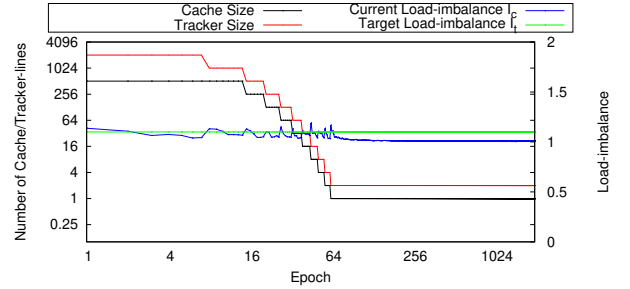


**(b) Changes in cache and tracker sizes and the current hit rate per cacheline $\alpha_c$ over epochs.**
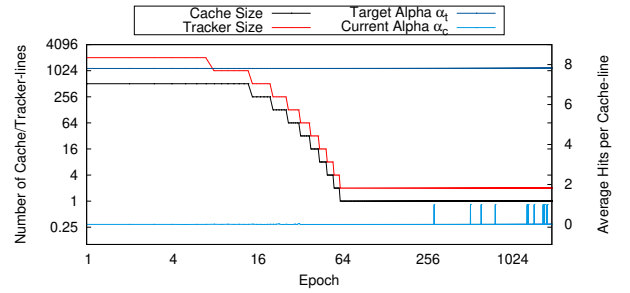
**Figure 8: CoT adaptively expands tracker and cache sizes to achieve a target load-imbalance $I_t = 1.1$ for a Zipfian 1.2 workload.**

**Analysis:** In Figure 8a, CoT is initially configured with a cache of size 2 and a tracker of size 4. CoT's resizing algorithm runs in 2 phases. In the first phase, CoT discovers the ideal tracker-to-cache size ratio that maximizes the hit rate for a fixed cache size for the current workload. For this, CoT fixes the cache size and doubles the tracker size until doubling the tracker size achieves no significant benefit on the hit rate. This is shown in Figure 8b in the first 15 epochs. CoT allows a warm up period of 5 epochs after each tracker or cache resizing decision. Notice that increasing the tracker size while fixing the cache size reduces the current load-imbalance $I_c$ (shown in Figure 8a) and increases the current observed hit per cache-line $\alpha_c$ (shown in Figure 8b). Figure 8b shows that CoT first expands the tracker size to 16 and during the warm up epochs (epochs 10-15), CoT observes no significant benefit in terms of $\alpha_c$ when compared to a tracker size of 8. In response, CoT therefore shrinks the tracker size to 8 as shown in the dip in the red line in Figure 8b at epoch 16. Afterwards, CoT starts phase 2 searching for the smallest cache size that achieves $I_t$. For this, CoT doubles the tracker and caches sizes

until the target load-imbalance is achieved and the inequality $I_c \leq I_t$ holds as shown in Figure 8a. CoT captures $\alpha_t$ when $I_t$ is first achieved. $\alpha_t$ determines the quality of the cached keys when $I_t$ is reached for the first time. In this experiment, CoT does not trigger resizing if $I_c$ is within 2% of $I_t$. Also, as the cache size increases, $\alpha_c$ decreases as the skew of the additionally cached keys decreases. For a Zipfian 1.2 workload and to achieve $I_t = 1.1$, CoT requires 512 cache-lines and 2048 tracker lines and achieves an average hit per cache-line of $\alpha_t = 7.8$ per epoch.



**(a) Changes in cache and tracker sizes and the current load-imbalance $I_c$ over epochs.**



**(b) Changes in cache and tracker sizes and the current hit rate per cache-line $\alpha_c$ over epochs.**

**Figure 9: CoT adaptively shrinks tracker and cache sizes in response to changing the workload to uniform.**

Figure 9 shows how CoT successfully shrinks tracker and cache sizes in response to workload skew drop without violating $I_t$. After running the experiment in Figure 8, we alter the workload to uniform. Therefore, CoT detects a drop in the current average hit per cache-line as shown in Figure 9b. At the same time, CoT observe that the current load-imbalance $I_c$ achieves the inequality $I_c \leq I_t = 1.1$. Therefore, CoT decides to shrink both the tracker and cache sizes until either $\alpha_c \approx \alpha_t = 7.8$ or $I_t$ is violated or until cache and tracker sizes are negligible. First, CoT resets the tracker to cache size ratio to 2:1 and then searches for the right tracker to cache size ratio for the current workload. Since the workload is uniform, expanding the tracker size beyond double the cache size achieves no hit-rate gains as shown in Figure 9b. Therefore, CoT moves to the second phase of shrinking both tracker and cache sizes as long $\alpha_t$ is not achieved and $I_t$ is not violated. As shown, in Figure 9, CoT shrinks both the tracker and the cache sizes until front-end cache size becomes negligible. As shown in Figure 9a, CoT shrinks cache and tracker sizes while ensuring that the target load-imbalance is not violated.

# 7 CONCLUSION

*Cache on Track* (CoT) is a decentralized, elastic and predictive cache at the edge of a distributed cloud-based caching infrastructure. CoT's novel cache replacement policy is specifically tailored

for small front-end caches that serve skewed workloads. Using CoT, system administrators do not need to statically specify the cache size at each front-end. Instead, they specify a target back-end load-imbalance $I_t$ and CoT dynamically adjusts front-end cache sizes to achieve $I_t$. Our experiments show that CoT's replacement policy outperforms the hit-rates of LRU, LFU, ARC, and LRU-2 for the same cache size on different skewed workloads. CoT achieves a target server size load-imbalance with 50% to 93.75% less front-end cache in comparison to other replacement policies. Finally, our experiments show that CoT successfully auto-configures the size of front-end caches in the presence of workload distribution changes.

## 8 ACKNOWLEDGEMENT

## REFERENCES

[1] 2018. Amazon ElastiCache in-memory data store and cache. https://aws.amazon.com/elasticache/.
[2] 2018. Azure Redis Cache. https://azure.microsoft.com/en-us/services/cache/.
[3] 2018. Memcached. A distributed memory object caching system. https://memcached.org/.
[4] 2018. Redis. http://redis.io/.
[5] 2018. A simple, asynchronous, single-threaded memcached client written in java. http://code.google.com/p/spymemcached/.
[6] Atul Adya, John Dunagan, and Alec Wolman. 2010. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 1–1.
[7] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-Sharding for Datacenter Applications.. In *OSDI*. 739–753.
[8] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann
[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.
[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.
[11] Yue Cheng, Aayush Gupta, and Ali R Butt. 2015. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 4.
[12] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*.
[13] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches.. In *NSDI*. 379–392.
[14] Edith Cohen and Martin Strauss. 2003. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 223–233.
[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
[16] Graham Cormode, Flip Korn, and Srikanta Tirthapura. 2008. Exponentially decayed aggregates on data streams. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 1379–1381.
[17] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. 2009. Forward decay: A practical time decay model for streaming systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 138–149.
[18] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. 2017. Caching with Dual Costs. In *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 643–652.
[19] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 371–384.
[20] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2011. Small cache, big effect: Provable load balancing for randomly partitioned

cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 23.
[21] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 21.
[22] Shahram Ghandeharizadeh, Marwan Almaymoni, and Haoyu Huang. 2019. Rejig: a scalable online algorithm for cache server configuration changes. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLII*. Springer, 111–134.
[23] Shahram Ghandeharizadeh and Hieu Nguyen. 2019. Design, implementation, and evaluation of write-back policy with cache augmented data stores. *Proceedings of the VLDB Endowment* 12, 8 (2019), 836–849.
[24] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 13.
[25] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 8.
[26] Jinho Hwang and Timothy Wood. 2013. Adaptive Performance-Aware Distributed Memory Caching.. In *ICAC*, Vol. 13. 33–43.
[27] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.
[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.
[29] Theodore Johnson and Dennis Shasha. 1994. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*.
[30] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 654–663.
[31] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers* 50, 12 (2001), 1352–1361.
[32] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 429–444.
[33] David Lomet. 2018. Caching Data Stores: High Performance at Low Cost. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1661–1661.
[34] David Lomet. 2018. Cost/performance in modern data stores: how data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. ACM, 9.
[35] David B Lomet. 2019. Data Caching Systems Win the Cost/Performance Game. *IEEE Data Eng. Bull.* 42, 1 (2019), 3–5.
[36] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *FAST*, Vol. 3. 115–130.
[37] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*. Springer, 398–412.
[38] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.
[39] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.
[40] Elizabeth J O'neil, Patrick E O'Neil, and Gerhard Weikum. 1999. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM (JACM)* 46, 1 (1999), 92–112.
[41] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.
[42] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. 2020. Learning relaxed belady for content distribution network caching. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 529–544.
[43] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019), 624–638.
[44] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2017. Caching at the web scale. *Proceedings of the VLDB Endowment* 10, 12 (2017), 2002–2005.
[45] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches.. In *USENIX Annual Technical Conference, General Track*. 91–104.