A Capacity-Elastic Cuckoo Filter Design for Dynamic Set Representation

Lailong Luo, Deke Guo, *Senior Member, IEEE*, Ori Rottenstreich, *Member, IEEE*, Richard T. B. Ma, *Senior Member, IEEE*, Xueshan Luo, and Bangbang Ren

***Abstract*—The emergence of large-scale dynamic sets in networked and distributed applications attaches stringent requirements to approximate set representation. The existing data structures (including Bloom filter, Cuckoo filter, and their variants) preserve a tight dependency between the cells or buckets for an element and the lengths of the filters. This dependency, however, degrades the capacity elasticity, space efficiency and design flexibility of these data structures when representing dynamic sets. In this paper, we first propose the Index-Independent Cuckoo filter (I2CF), a probabilistic data structure that decouples the dependency between the length of the filter and the indices of buckets which store the information of elements. At its core, an I2CF maintains a consistent hash ring to assign buckets to the elements and generalizes the Cuckoo filter by providing optional *k* candidate buckets to each element. By adding and removing buckets adaptively, I2CF supports the bucket-level capacity alteration for dynamic set representation. Moreover, in case of a sudden increase or decrease of set cardinality, we further organize multiple I2CFs as a Consistent Cuckoo filter (CCF) to provide the filter-level capacity elasticity. By adding untapped I2CFs or merging under-utilized I2CFs, CCF is capable of resizing its capacity instantly. The trace-driven experiments indicate that CCF outperforms its alternatives and realizes our design rationales for dynamic set representation simultaneously, at the cost of a little higher complexity.**

***Index Terms*—Cuckoo filter, consistent hashing, elasticity.**

I. Introduction

SET REPRESENTATION while supporting membership queries is a fundamental problem in databases, caches, routers, storage, and distributed applications [1]. These systems often represent set elements with a probabilistic data structure and support constant-time approximate membership query with small false positive probability. The most widely-used probabilistic data structures for approximate membership query are Bloom filter [2], [3], Cuckoo filter [4], [5] and their variants [6], [7], [8], [9].

Bloom filter and Cuckoo filter represent sets in diverse ways. Bloom filter is a fixed-length array of bits which are initialized as 0s. To insert an element, k independent hash functions are employed to map the element into the bit vector. Thereafter, the corresponding bits are set to 1s. When testing the membership of any element x, Bloom filter just checks the k corresponding bits are non-zero. If they are all ones, Bloom filter concludes that x is a member of the set (possibly implying a false positive); otherwise, it correctly indicates that x is not a member (no false negatives). Unlike Bloom filter, Cuckoo filter stores the fingerprints of the elements with their candidate buckets directly. Cuckoo filter derives two candidate buckets for each element with the partial cuckoo hashing strategy [10] and tries to store the fingerprint into one of the candidate buckets. An element is identified as a member of the set if its fingerprint can be found in either of its candidate buckets. Bloom filter and Cuckoo filter, however, fail to represent dynamic set members because of their incapability of resizing their capacities.

To this end, Dynamic Bloom filter (DBF) [6] and Dynamic Cuckoo filter (DCF) [7] have been developed. Both DBF and DCF attempt to add and merge homogeneous Bloom filters and Cuckoo filters to extend and downsize their capacities on demand. In both DBF and DCF, the length of each filter is predefined and cannot be altered since the indices of cells or candidate buckets are determined by calculating the modulus based on the length of the filter. As a consequence, they can only resize their capacity by adding or merging homogeneous filters. In the worst case where one filter has to be added to store only one additional element, the resultant space utilization can be very low. Therefore, in space-scarce scenarios, the bucket-level capacity alteration is necessary to save space. Moreover, a major weakness of DBF is that it fails to support reliable element deletion [7] since there may be multiple BFs which satisfy the membership query condition. Although DCF guarantees reliable element deletion, it employs the XOR operation to derive the second candidate bucket during reallocations. Therefore, the length of each Cuckoo filter can only be of the form $m=2^\gamma$ ($\gamma\geq0$). Otherwise, the XOR results may go out of range.

Manuscript received July 9, 2020; revised March 27, 2021, June 28, 2021, and July 18, 2021; accepted July 20, 2021. Date of publication July 26, 2021; date of current version December 9, 2021. This work was supported in part by the National Key Research and Development Program of China under grant 2018YFB1800203, in part by the National Natural Science Foundation of China under Grants 62002378 and 61772544. The associate editor coordinating the review of this article and approving it for publication was A.-C. Pang. *(Corresponding authors: Xueshan Luo; Deke Guo.)*

Lailong Luo is with the Science and Technology on Information Systems Engineering Laboratory and the National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, Hunan, China (e-mail: luolailong09@nudt.edu.cn).

Deke Guo, Xueshan Luo, and Bangbang Ren are with the Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, Changsha 410073, Hunan, China (e-mail: dekeguo@nudt.edu.cn; xsluo@nudt.edu.cn; renbangbang11@nudt.edu.cn).

Ori Rottenstreich is with the Department of Computer Science and the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa 3200003, Israel (e-mail: or@technion.ac.il).

Richard T. B. Ma is with the School of Computing, National University of Singapore, Singapore (e-mail: tbma@comp.nus.edu.sg).

Digital Object Identifier 10.1109/TNSM.2021.3099433

TABLE I
THE FEATURES OF SET REPRESENTATION DATA STRUCTURES. THE NUMBER OF "+" QUANTIFIES THAT FEATURE

Name	BF	DBF	CF	DCF	ACF	SCF	I2CF	CCF
CE		++		++			++	+++
SE	+	+	++	++	++	++	+++	+++
DF	++	++	+	+	+	+	+++	+++

Consequently, we envision the design of a Cuckoo filter style data structure which properly concerns the following three design rationales for dynamic set representation.

- **Capacity elasticity (CE):** The data structure's capacity is adaptively adjustable according to the set cardinality. Despite the unpredictability of the number of elements to represent, the offered capacity shows coincident changing trends as the set cardinality adaptively.
- **Space efficiency (SE):** The space utilization remains at a high level irrespective of the variation of set cardinality. This is extremely important for space-scare scenarios, e.g., wireless sensor networks.
- **Design flexibility (DF):** All the parameters are adjustable so that users can customize their own configurations according to their design goals. For example, the number of hash functions may be increased for higher space utilization or decreased for better query throughput.

These rationales, if realized, will bring unprecedented benefits for set representation and membership query, in terms of space-saving and quality of service. The design flexibility further extends the applicability of the data structure to more general scenarios with diverse requirements.

The existing probabilistic data structures, however, fail to achieve the three rationales properly and simultaneously. As shown in Table I, Bloom filter and DBF achieve low space utilization. The reason is that they keep half of the bits as 0s, in order to incur the least false positive rate. By contrast, Cuckoo filter and its variants improve their space utilization with the reallocation strategy during each insertion. DBF and DCF offer capacity elasticity to some extent by adding and merging filters dynamically. However, in reality, a more fine-grained capacity scaling is needed to handle small-scale capacity overflows and recycle space timely when a few elements are removed. Furthermore, existing data structures are somehow hindered by their limited design flexibility. In the framework of Bloom filters, the parameters have to be carefully designed to guarantee their target false positive rate. Meanwhile, current proposals of Cuckoo filters must use a fixed number of hash functions and a power of two number of buckets.

A common reason for the existing data structures' deficiency of achieving the three rationales is that they preserve a tight dependency between the cells or buckets for an element and the lengths of the filters. As a result, their capacities have to be predefined and remain immutable irrespective of the change of dynamic sets. Therefore, in this paper, we first propose I2CF, a probabilistic data structure which decouples the dependency between the length of the filter and the indices of buckets which store the information of elements. At its core, an I2CF maintains *a consistent hash ring* [11], [12] to assign buckets to the elements and generalizes the Cuckoo filter by providing optional k candidate buckets to each element. By adding and removing buckets adaptively, I2CF supports bucket-level capacity alteration for dynamic set representation.

Moreover, in case of a sudden increase or sharp decrease of set cardinality, we further organize multiple I2CFs as a CCF to provide filter-level capacity elasticity. By adding untapped I2CFs or merging under-utilized I2CFs, CCF is capable of resizing its capacity instantly. As shown in Table I, both I2CF and CCF offer elegant space efficiency and design flexibility. CCF has better capacity elasticity than I2CF, since I2CF only provides bucket-level capacity alteration, while CCF additionally supports filter-level capacity adjustment for dynamic sets. In fact, I2CF is a special case of CCF when only one I2CF is maintained. To summarize, we achieve the following contributions.

- We first design I2CF (Index Independent Cuckoo filter), a probabilistic data structure which decouples the dependency between the length of the filter and the indices of buckets which store the information of elements. It allows flexibility in the memory size without the need for reallocating most elements. Thereafter, we organize multiple I2CFs as a CCF and present the algorithms for dynamic set representation and capacity resizing.
- For any I2CF with given parameters in a CCF, we present a new threshold for the ratio between the number of represented elements and the number of buckets. Additionally, we derive an upper bound for the probability that a given number of elements can be successfully stored in an I2CF.
- Trace-driven evaluations are conducted to measure the performance of our proposals. The results show that CCF outperforms DCF and realizes the three design rationales simultaneously at the cost of a little higher time-complexity.

The rest of this paper is organized as follows. Section II elaborates on the application of sketches in networks. Section III introduces the background and related work. Section IV presents the I2CF and CCF design and their operations. Section V details the theoretical performance analysis for CCF. Section VI reports the evaluation results and Section VII concludes the whole paper.

II. Applications of Sketches in Networks

Set representation is a fundamental task in networking for applications such as content caching, packet routing, privacy preserving, network measurement, blockchain and beyond [1], [13], [14]. In such scenarios, sketch data structures are employed to rep-resent a given set while supporting constant-time membership queries. According to the query result, subsequent actions will be triggered or avoided.

Content caching is widely implemented in network applications [15], [16], e.g., CDN, Web, DNS. A typical caching system includes a main memory which contains all the elements/items to query and a cache which only stores a subset of the union. Often, sketch data structures are employed to represent the elements in the cache memory. The access of an element x is first directed to the sketch. If the sketch indicates that x is an element in the cache, the access tries to read the element from the cache. Due to the potential false positive errors, x may be not found in the cache, then the access will be routed to the main memory. If the sketch judges that x is not stored by the cache, the request will fetch x from the main memory directly without accessing the cache. In summary, a cache system benefits from the sketches by refining unnecessary access to the main memory.

Set representation also plays an essential role in packet routing for both wired and wireless networks. In wired networks, sketches are implemented to speed-up IP lookups and multicast routing. Specifically, Lim *et al.* proposed to record the IP length to enable fast longest IP prefix matching [17]. Besides, the multicast tree is represented by a sketch which is thereafter embedded into the packet header or stored in the switch ports [18]. As a result, a packet can naturally find its next hop by referring to the in-packet or in-switch sketch. In wireless networks such as mobile ad hoc networks [19], wireless sensor networks [20], wireless named data networks [21], sketches are employed to propagate information of topology, neighborhood, nearby services, probed data, etc. With the help of sketches, nodes in the wireless networks route packets more efficiently.

Sketches naturally anonymize user-sensitive data (such as geo-location, biometric data, personal interest, etc) since they represent elements without recording the real content. Therefore, sketches are widely employed to preserve privacy. For location-based services, Calderoni *et al.* proposed to represent the location information with sketches such that the real user location is transparent for upper-level services [22]. Similarly, biometric data, which is widely adopted for authentication, can also be anonymized by sketches to provide fast membership query without the worry of data leakage [23]. Moreover, a core module of social APPs is to recommend potential friends with similar interests to users. To preserve privacy, Oriero *et al.* employ sketches to represent users' interested topics and thereafter share these sketches among trusted friends [24].

State-of-the-art collaborative network measurement techniques mostly collect the flow information with sketches and then aggregate them in a central node for further analysis [25]. For diverse measurement tasks, different sketches are designed and implemented. These sketches keep track of the flow size with counters, so that complex analysis such as heavy hitter, heavy changer, entropy evaluation, etc, are enabled.

Consider the wide use of sketches in the networking and communication community, this paper focus on the representation of dynamic sets which further restrict the elasticity, efficiency and flexibility of the employed sketch data structure.

III. Background and Related Work

In this section, we briefly introduce the background and the related work of this research. We mainly introduce the Cuckoo hash table, the recently proposed Cuckoo filter and its variants as well as the consistent hashing techniques.

A. Cuckoo Hash Table

A hash table provides constant query time but incurs only near 50% space utilization. By integrating the "power of two choices" into hash tables, cuckoo hash table [26], [27] achieves high space utilization with the guarantee of constant-time queries. Basically, a cuckoo hash table is an array of m buckets. Each bucket is allowed to store an element. To insert an element x, two independent hash functions h_1 and h_2 are employed to select two candidate buckets for x. If either bucket $h_1(x)\%m$ or $h_2(x)\%m$ is empty, x will be stored in either of them. In contrast, if both of them are occupied, the cuckoo hash table will randomly select a bucket and kick out the stored element to accommodate x. The victim will then be reallocated to its other alternative bucket. The cuckoo hash table keeps kicking and reallocating the stored elements until the victim is accommodated successfully or the number of iterations reaches a predefined upper bound *max*. When an element is failed to be inserted, the cuckoo hash table is considered as a full cuckoo hash table. To query or access the elements, the users only need to check the two corresponding buckets. The reallocations help cuckoo hash table refine the placements of previous elements, thus enabling the high space utilization. In a real implementation, each bucket is suggested to accommodate multiple elements [28], and the number of hash functions can also be an optionally k rather than the fixed value 2 [29]. Zentgraf *et al.* further optimize the element-bucket assignment with the Bellman-Ford and Hopcroft-Karp algorithms, such that an arbitrary query can find its content as soon as possible [30].

B. Consistent Hashing

An inherent shortcoming for a hash table is that its resizing calls for the remapping of all elements. The reason is that the location for an element is determined by the modular result between the hash value and the table length. Consistent hashing [11], [31] relaxes this situation such that only a small part of the stored elements will be moved when resizing the hash table, with the assumption that a bucket can accommodate multiple elements. Consistent hashing maps both the elements and the buckets in the hash table into a ring ranged from 0 to a given large integer M. Thereafter, the elements are assigned to the buckets with either clockwise or anti-clockwise order in the ring. When a new bucket is added into the ring, only the elements in its successor may be moved to it. Similarly, when a bucket is removed from the hash table, the elements in the bucket will be pushed to its successor directly. Given the number of buckets in the hash table as m and the number of elements as n, each update of the consistent hash table only affects n/m elements on average. Consistent hashing has been widely employed in distributed systems such as peer-to-peer network [12], content delivery network [32], OpenStack Swift [33], Amazon Dynamo [34], etc. To save power consumption, Xie and Chen leverage the consistent hashing technique by storing the files on *primary servers* and scaling down to a few active servers when necessary [35]. In this paper, we employ the consistent hashing to assign the fingerprints into the CCF buckets with the ability of scale-up and scale-down at will.

C. Cuckoo Filters

Cuckoo filter (CF) [5] is a light-weight probabilistic data structure to support constant-time membership query, based on the framework of cuckoo hash table [26]. Cuckoo filter replaces the actual contents of elements in the cuckoo hash table with the fingerprints of elements. Structurally, a CF consists of m buckets, each of which is capable of residing b fingerprints. Any element x is associated with a f-bit fingerprint (η_x) generated by a hash function h_0. An arising challenge for CF is to derive out the alternative bucket for a victim without the raw data of the victim. CF employs the *partial-key cuckoo hashing strategy* [10] to tackle this issue. Basically, the alternative bucket can be derived out by executing an XOR operation towards the current bucket and the hash value of the victim fingerprint. That is, the two bucket locations are derived as $h_1(x) = hash(x)$ and pair-wisely $h_2(x) = h_1(x) \oplus hash(\eta_x)$.

With the above design, to insert an element x, CF first calculates the fingerprint η_x of x and then generates the two candidate buckets based on the redesigned hash functions h_1 and h_2. Thereafter, the fingerprint η_x will be stored into either of the candidates and the victim will be reallocated in the CF vector if necessary. To query an element y is a member of set A or not, CF checks the two corresponding buckets of y. If the fingerprint η_y is stored in either them, CF judges $y \in A$; otherwise CF concludes $y \notin A$. Due to the potential hash collisions of the fingerprints, CF may suffer from false positive errors (reporting elements which do not belong to A as members of A). Theoretically, the false positive rate of CF is bounded as $\xi_{CF} = (1-(1-\frac{1}{2^f})^{2b})$, where f is the number of bits for a fingerprint and b is the number of slots in a bucket. There are no false negative errors for CF if all elements in A are inserted successfully.

Most recently, several CF variants have been proposed for further improvement. Experimental results validate CF's performance, but the merit is not theoretically guaranteed. The simplified cuckoo filter [8] (SCF) calculates the locations of buckets for an element x as $h_1(x)$ and $h_1(x) \oplus \eta_x$. The impact of the simplification can be visualized by a fingerprint-edge graph whose vertices are the buckets of the hash table, and whose edges connect the possible pairwise locations of a fingerprint. Based on graph theory, SCF provides theoretical performance analysis to its filter.

Adaptive cuckoo filter [9] (ACF) tries to remove false positive errors from the CF vector by resetting the collided fingerprints. The ACF consists of a CF and a corresponding cuckoo hash table. Such a design enables ACF to identify false positive errors and decouple the locations of buckets from the fingerprints. When a false positive error occurs, ACF generates a new fingerprint for the conflicted element which is directly available from the cuckoo hash table. As a consequence, the false positive error will not arise again in the future.

Inspired by the Dynamic Bloom filter [6], dynamic cuckoo filter [7] (DCF) dynamically maintains multiple homogeneous

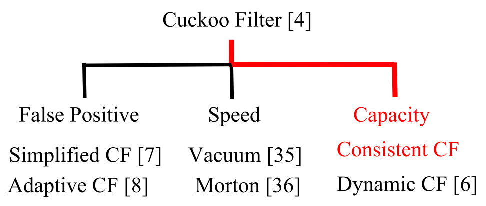


Fig. 1. The position of this work in the community.

CFs to enable elastic capacity alteration. Initially, only one Cuckoo filter is maintained and marked as active. The subsequent homogeneous CFs will be introduced in either an active or passive manner. A recycling mechanism is suggested to merge two underloaded CFs, thereby improving the space utilization. The false positive rate DCF is bounded as $1-(1-f_{CF})^s$, where f_{CF} is the false positive rate of each CF vector and s is the number CFs maintained in DCF.

Vacuum filter [36] and Morton filter [37] speed up the throughput of insertion, deletion, and query with different methodologies. Specifically, Morton filter [37] introduces a compressed format that permits a logically sparse filter to be stored compactly in memory. It prefers to store the elements with their first candidate buckets, such that subsequent retrieval of fingerprints requires fewer hardware cache accesses without accessing the alternative buckets. The Vacuum filter, by contrast, Vacuum filter proposes to put the two candidate buckets in a single *alternate range* which can be accessed by one memory access. The Vertical Cuckoo filter [38] proposes to offer more than 2 candidate buckets for each element for better space utilization.

The above variants of CF, however, fail to realize our design rationales properly. SCF and ACF leverage the employed hash functions only, but their capacities cannot be resized after implementation. DCF supports filter level capacity alteration but incurs limited design flexibility and untimely space recycling. Consequently, we present CCF (Consistent Cuckoo filter), a novel probabilistic data structure which promises capacity elasticity, high space utilization, and design flexibility simultaneously. Fig. 1 explicitly plots the position of this work.

IV. Consistent Cuckoo Filter

We describe the design of the CCF (Consistent Cuckoo filter) in detail here, including its data structure, operations for set representation and resizing strategies. Before that, we introduce the I2CF (Index-Independent Cuckoo filter), which is the basic component of CCF.

A. Design of Index-Independent Cuckoo Filter (I2CF)

To represent dynamic sets, the employed data structure should offer elastic capacity. Although DBF and DCF are capable of filter-level capacity elasticity, they fail to provide the ability of fine-grained capacity alteration. The reason is that their lengths of filters are predefined and immutable throughout their lifetime. The using of XOR operations to compute hash values in Cuckoo filter further exacerbates the capacity elasticity by restricting the filter length to be a power

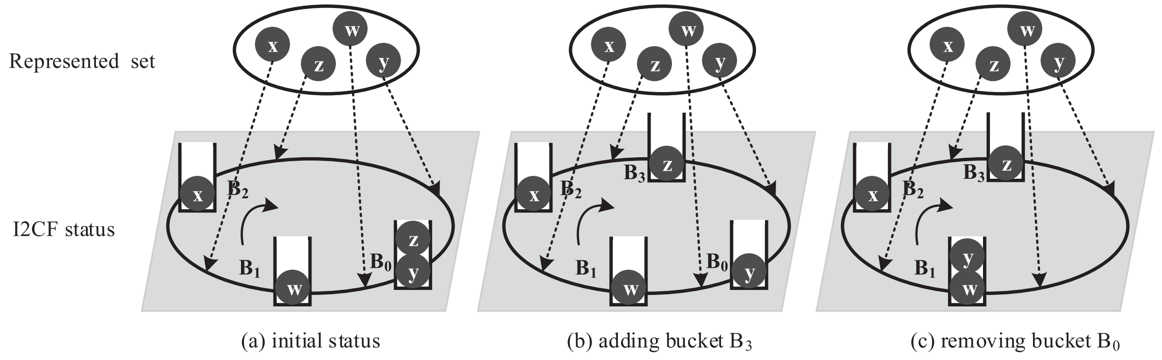


Fig. 2. An illustrative example of I2CF with $k = 1$, $b = 2$ and $v = 1$. It represents a set with 4 elements, including x, y, z and w. Initially, the I2CF employs 3 buckets to store these elements. When a new bucket B_3 is added as a successor of B_2, the fingerprint of z is reallocated from B_0 to B_3, since the hash value of η_z is mapped between B_2 and B_3. When B_0 is removed, the fingerprint η_y is reallocated to B_0's successor B_1.

of two. Therefore, we redesign the framework of Cuckoo filter and propose the Index-Independent Cuckoo filter (I2CF) here.

Basically, I2CF consists of multiple buckets, each of which has b slots. That is, each bucket can accommodate b fingerprints at most. As shown in Fig. 2, the buckets are mapped onto a consistent hash ring [11], [12] ranging from 1 to $M-1$. To ensure better load balance in the consistent hash ring, each bucket has $v \geq 1$ virtual nodes in the consistent hash ring. I2CF also stores the fingerprints of elements instead of the actual contents by offering each fingerprint $k \geq 1$ candidate buckets. An element is successfully represented if its fingerprint is stored in one of its candidate buckets. To determine the candidate buckets of an element x, k independent hash functions are employed to map the fingerprint η_x onto the consistent hash ring. Thereafter, the k nearest buckets (in a clockwise order by default) of the k hash values are regarded as the candidate buckets of η_x. In this way, the candidate buckets are index-independent and decided by the consistent hashing. A fingerprint can be stored in any one of these candidate buckets. If all the candidate buckets are fully occupied, I2CF randomly kicks out an existing fingerprint from one of these buckets to store the new fingerprint. The victim will be reallocated to one of its other candidate buckets. The reallocation ends successfully when a bucket has available space and fails when the number of such reallocations reaches the given threshold.

Compared with Cuckoo filter, I2CF has two major improvements. First, I2CF organizes the buckets as a consistent hash ring to decouple the dependency between candidate buckets and the length of the filter. As a consequence, I2CF naturally enables the capability of adding and removing buckets on demand. A toy example of adding and removing buckets from an I2CF is given in Fig. 2. Second, I2CF generalizes the number of candidate buckets from the fixed two in Cuckoo filter as a mutable variable k. This generalization further improves its design flexibility. Moreover, as analyzed later, larger k values also guarantee higher space utilization. With these improvements, I2CF achieves the bucket-level capacity elasticity and high space utilization to represent dynamic sets.

B. Overview of the Consistent Cuckoo Filter (CCF)

I2CF provides bucket-level capacity elasticity, but when set cardinality increases drastically, a single I2CF may fall short of offering enough space timely. Therefore, we further generalize I2CF as CCF which dynamically maintains multiple I2CFs. Just like existing CF variants, CCF also leverages fingerprints to represent elements in a set. The fingerprint for an element x is generated by mapping x into a given range $[0, 2^f - 1]$ with a hash function h_0. Basically, a CCF consists of s ($s \geq 1$ and initialized as 1) *heterogeneous* I2CFs. An arbitrary I2CF$_i$ ($i \in [0, s-1]$) has $m_i \geq 1$ buckets with $b_i \geq 1$ slots. The employed number of hash functions k_i and the value of M_i in I2CF$_i$ are also allowed to be different from other I2CFs. With such a framework, CCF enables ultimate design flexibility. Note that to multiplex the calculated hash values for each fingerprint among the I2CFs, we prefer $k_0 = \cdots = k_i = \cdots = k_{s-1} = k$, and $M_0 = \cdots = M_i = \cdots = M_{s-1} = M$ by default. More importantly, CCF provides capacity elasticity at both the bucket level and filter level. That is, its capacity can be altered by adding or removing buckets in any I2CF, as well as introducing untapped or compacting under-utilized I2CFs. The details are given in Section IV-D. When an I2CF is extended or introduced, it will be marked as active to store new elements.

Theorem 1: For an I2CF$_i$ ($i \in [0, s-1]$) in CCF, let b_i and k_i denote the number of slots in each bucket and the number of candidate buckets in the filter I2CF$_i$, respectively. The false positive rate for a CCF query can be calculated as:

$$\xi_{CCF} = 1 - \prod_{i=0}^{s-1}(1 - \xi_i) = 1 - \prod_{i=0}^{s-1}\left(1 - \frac{1}{2^f}\right)^{k_i \cdot b_i}. \quad (1)$$

When $k_0 = \cdots = k_i = \cdots = k_{s-1} = k$, $b_0 = \cdots = b_i = \cdots = b_{s-1} = b$,

$$\xi_{CCF} = 1 - \prod_{i=0}^{s-1}(1 - \xi_i) = 1 - \left(1 - \frac{1}{2^f}\right)^{s \cdot k \cdot b} \approx \frac{s \cdot k \cdot b}{2^f}. \quad (2)$$

The false positive error of CCF stems from the hash collisions of the fingerprints. If two elements $x \in A$ and $y \notin A$ share the same fingerprint, i.e., $\eta_x = \eta_y$, the membership query of y implies a false positive error due to the existence of x. Within the CCF framework, a membership query may check all of the s I2CF vectors. For I2CF$_i$, the false positive rate is $\xi_i = 1 - (1 - \frac{1}{2^f})^{k_i \cdot b_i}$. The global false positive rate is thus derived as $\xi_{CCF} = 1 - \prod_{i=0}^{s-1}(1 - \xi_i)$. Note that, both DCF and CCF have multiple filters and share the same false positive

Algorithm 1: CCF Insertion(η_x)

```
Input: The fingerprint to insert $\eta_x$
1  Calculate the hash values $h_1(\eta_x), \cdots, h_k(\eta_x)$;
2  Decide the candidate buckets for $\eta_x$ in the active I2CF based on $h_1(\eta_x), \cdots, h_2(\eta_x)$;
3  if $\eta_x$ can be successfully inserted into the active I2CF within max reallocations then
4      return True;
5  else
6      Extend CCF with its resizing strategy;
7      Insert $\eta_x$ into the extended or added I2CF;
8      return True;
9  end
```

Algorithm 2: CCF Query(x)

```
Input: The element to query $x$
1  $\eta_x = h_0(x) \ mod \ 2^f$;
2  Calculate the hash values $h_1(\eta_x), \cdots, h_k(\eta_x)$;
3  for $i$=0 to $s-1$ do
4      Determine candidate buckets $B_i^1(x), \cdots, B_i^k(x)$ in I2CF$_i$;
5      for $j$=0 to $k-1$ do
6          if $B_i^j(x)$ has $\eta_x$ then
7              return True;
8          end
9      end
10 end
11 return False;
```

Algorithm 3: CCF Deletion(x)

```
Input: The element to delete $x$
1  $\eta_x = h_0(x) \ mod \ 2^f$;
2  Calculate the hash values $h_1(\eta_x), \cdots, h_k(\eta_x)$;
3  for $i$=0 to $s-1$ do
4      Determine candidate buckets $B_i^1(x), \cdots, B_i^k(x)$ in I2CF$_i$;
5      for $j$=0 to $k-1$ do
6          if $B_i^j(x)$ has $\eta_x$ then
7              Remove $\eta_x$ from $B_i^j(x)$;
8              Downsize the CCF when necessary;
9              return True;
10         end
11     end
12 end
13 return False;
```

rate. Generally, larger f leads to lower false positive rate, while larger k, b and s result in higher false positive rate. However, DCF fails to support runtime false positive rate guarantee since the value of s increases continually with the increase of set cardinality. As a consequence, the false positive of DCF keeps increasing when more CFs are launched. CCF, on the contrary, provides runtime false positive rate guarantee by setting a threshold for s. If the value of s reaches the threshold, on one hand, CCF can conduct the compact operation to hopefully merge some I2CF vectors. On the other hand, CCF only employs the bucket-level resizing strategy to accommodate the coming elements, thereby the value of s will not be increased any more. Thus, the false positive rate can be reasonably bounded.

C. Dynamic Set Representation With CCF

In this subsection, we present the basic operations of CCF for dynamic set representation, including insertion, query, and deletion of elements. The associated resizing strategies are detailed later in Section IV-D.

Insertion: The CCF tracks the number of elements inserted into each of its I2CF and thereafter marks the I2CF which represents the least elements as an active I2CF. To insert an element x, CCF first generates its fingerprint by mapping x into the range $[0, 2^f-1]$. Then, k independent hash functions map η_x onto the consistent hash ring. Based on the generated hash values, the consistent hashing determines the candidate buckets for η_x in the active I2CF. After that, we try to insert η_x into the active I2CF by following the strategy provided by the cuckoo hashing [5]. If the active I2CF can successfully store fingerprint η_x, the insertion algorithm will be terminated. Otherwise, CCF capacity has to be extended at either the bucket level or filter level. Thereafter, η_x will be inserted into the extended or added I2CF. The pseudo-code is shown in Algorithm 1. Note that, upon being extended, CCF marks the manipulated I2CF as active, so that the coming elements will be stored by this I2CF vector. We suggest choosing the I2CF with least buckets for better balance when bucket level extension is performed. Sometimes multiple buckets have to be added to successfully reside η_x. If there are still many elements to be inserted after x, CCF will introduce a new I2CF vector within the constraints of false positive rate, such that the coming elements will be stored immediately.

Query: Membership query with CCF may check every I2CF vector. Let s denote the number of I2CF vectors in CCF. We need to check at most $s{\cdot}k$ buckets. Algorithm 2 presents a membership query in detail. The fingerprint η_x is hashed by k hash functions to determine the locations of η_x in the hash ring for I2CF$_i$ ($i{\in}[0, s-1]$). Based on the hash values, the consistent hashing tells CCF the candidate buckets for η_x in I2CF$_i$. Then, if any bucket holds η_x, the membership query terminates and returns true. By contrast, if η_x cannot be found in all I2CFs, CCF judges $x{\notin}A$ and returns false. There may be a potential false positive error for any queried element, but no false negative errors for the stored elements.

Deletion: The deletion of an element x needs to first perform a membership query for finding its possible locations. If a corresponding fingerprint η_x is found, then the matched fingerprint will be removed from CCF. Algorithm 3 shows the details of the delete operation. If the fingerprint η_x is not found in CCF, the deletion algorithm returns fail. When a sufficient number of elements have been deleted from CCF, the resizing operations will be executed to downsize CCF capacity and maintain high space utilization. CCF prefers filter-level resizing since a smaller s ensures a lower false positive rate.

D. Resizing Operations of I2CF and CCF

An essential challenge for dynamic set representation is the unpredictable set cardinality n. This challenge puts forward new requirement for the employed data structure, i.e., the capability of capacity resizing. Moreover, the set cardinality n may vary irregularly, i.e., n may increase or decrease progressively or dramatically. To handle that, the data structure must be resized in diverse granularity. Therefore, we propose two options to extend the capacity of CCF, i.e., a scale-up method which adds buckets into an I2CF, and a scale-out method which adds an untapped I2CF into CCF. Pair-wisely, the CCF capacity can be downsized by either removing buckets from one specific I2CF or compacting sparse I2CFs. *Scale up* and *scale down* support the bucket-level capacity alterations, meanwhile, *scale out* and *compact* achieve the filter-level capacity adjustments. These methods generate ultimate elasticity for CCF when representing dynamic sets.

Scale up: When a new bucket is added into an I2CF, only the fingerprints stored in the bucket's successor may be affected. We consider that a new bucket B_{new} is mapped between two existing buckets B_i and B_j $(i, j \in [0, m-1])$, and B_j is the successor of B_{new}. In this case, only the fingerprints in B_j might have to be reallocated to bucket B_{new}. Specifically, if a fingerprint in B_j is mapped between B_i and B_{new}, it should be moved to B_{new}; otherwise, it should still remain in B_j. Especially, if B_j is empty, B_{new} will also be empty. A toy example for adding bucket can be found in Fig. 2(b).

Scale down: Correspondingly, CCF can remove buckets from an I2CF for higher space utilization. When an existing bucket is removed from an I2CF, only the fingerprints in this bucket should be reinserted into the CCF. We consider two buckets B_i and B_j in the hash ring such that B_j is the successor of B_i. CCF tries to store the fingerprints in B_i by pushing them into bucket B_j preferentially and then reallocating the rest of the fingerprints to other buckets. If all the fingerprints are successfully stored, B_i will be removed; otherwise, B_i cannot be removed. An illustrative example for removing bucket is shown in Fig. 2(c). When scaling down, CCF prefers removing empty or under-utilized buckets for time-saving.

Scale out: Another method to increase the capacity of CCF is to add untapped I2CFs. Initially, CCF maintains a single I2CF and scales up or scales down this filter according to the real demand. When the number of elements to represent increases dramatically, the capacity of CCF can be extended immediately by adding one or multiple untapped I2CFs into the system. Note that, the added I2CFs are allowed to be heterogeneous since they are totally independent. The number of buckets and the number of slots are all mutable.

Compact: When an I2CF becomes sparse due to the removal of elements from the set, CCF tries to remove this I2CF through the compact operation. As shown in Algorithm 4, CCF first selects a least-loaded I2CF vector I2CF$_L$ and removes it. The updated CCF is denoted as CCF$_T$. Thereafter, we try to reinsert the fingerprints in I2CF$_L$ into CCF$_T$. If all the fingerprints in I2CF$_L$ can be successfully inserted into CCF$_T$, the selected I2CF$_L$ is allowed to be removed; otherwise, the CCF is already condensed enough and cannot be further compressed. The compact algorithm keeps removing I2CF vectors until an undeletable I2CF is reached.

In practice, the set cardinality n varies due to the join or removal of elements. When the value of n increases (decreases) gradually, CCF executes the scale up (scale down) algorithm to adaptively adjust its capacity. In the case of dramatic growth (reduction) of n, the scale out (compact) operation will be employed to extend (downsize) the CCF instantly. With these strategies, CCF ensures capacity elasticity and high space utilization simultaneously.

Algorithm 4: CCF Compact()

```
Input: The current CCF
success = True;
while success do
    Select the least-loaded I2CF vector I2CF$_L$ in CCF;
    Declare a new CCF named CCF$_T$;
    Let CCF$_T$ = CCF.remove(I2CF$_L$);
    for all $\eta_x$ stored in I2CF$_L$ do
        if <EXCLAMATION>CCF$_T$.insertion($\eta_x$) then
            success = False;
            break;
        end
    end
    if success then
        CCF = CCF$_T$;
    end
end
```

E. Resizing Strategy in CCF

CCF provides the capacity elasticity both in the filter level and the bucket level. For further understanding, we state the resizing strategy in CCF here explicitly. Note that CCF mainly offers three functions to its users, i.e., element insertion, query, and deletion, while only insertion and deletion may trigger the resizing process.

We rely on the arrival rate of elements (number of inserted elements per unit time, denoted as α) and the upper bound of CCF false positive rate $\overline{\xi}$ to jointly decide the use of scale up and scale out. Let $\overline{\alpha}$ be a sentinel value: only when $\alpha \leq \overline{\alpha}$, CCF can insert the arrival elements with small-scale scale up operations (adding one bucket in each extension). We employ the scale out in a conservative manner, since increasing the number of I2CFs results in a higher overall false positive rate. Specifically, only when $\alpha > \overline{\alpha}$ and the overall false positive rate ξ_{CCF} after adding the empty I2CF is no more than $\overline{\xi}$, the scale out operation may be triggered. Otherwise, CCF will be extended by scale up only. Especially, when $\alpha > \overline{\alpha}$ but no more I2CFs are allowed to be added due to the constraint of $\overline{\xi}$, CCF has to extend the I2CFs with large-scale scale up operation (adding multiple buckets in each extension). The exact number of added buckets in each extension is proportional to the arrival rate α.

Compared with the large-scale scale up operation, the scale out operation is capable of extending the CCF instantly; while the former incurs more time-complexity to add buckets. Therefore, within the constraints of $\overline{\xi}$, CCF prefers the scale out operation than large-scale scale up.

Pair-wisely, for downsizing, CCF provides both the scale down operation to delete buckets from an I2CF and the compact operation to remove sparse I2CFs. Whenever a bucket gets empty because of element deletion, it will be removed from that I2CF with the scale down operation. By contrast, the compact operation will be triggered only if the number of stored fingerprints in an I2CF is less than a predefined threshold. To this end, CCF integrates a counter with each I2CF to track the number of stored fingerprints.

In reality, an online system inserts and deletes elements frequently; hence it may not be advisable to resizing the CCF repeatedly. Especially, when the element arrival rate approximates the removal rate (number of removed elements per unit time, denoted as β), the required capacity is stable. In that case, resizing CCF is not urgent, unless an insertion failure occurs or an I2CF gets extremely under-utilized. Therefore, from a high level, we suggest to resize the CCF with joint consideration of α and β. When $\alpha > \beta$, scale up or scale out operation is needed to extend CCF; when $\alpha < \beta$, scale down or compact operation is required to downsize CCF.

V. Performance Analysis of CCF

Here we first theoretically analyze the time-complexity of CCF. Then, we present a new method to calculate the threshold of the ratio between the number of elements to represent n_i and the number of buckets m_i for a given I2CF$_i$. Lastly, we offer a bounded probability of successfully inserting a given number of elements with a given I2CF$_i$.

A. Time-Complexities of CCF

Theorem 2: Consider a CCF with s homogeneous I2CFs and $k_0 = \cdots = k_i = \cdots = k_{s-1} = k$, $b_0 = \cdots = b_i = \cdots = b_{s-1} = b$. Let *max* and *m* denote the allowed reallocation times and the lengths of I2CFs. The time-complexities for CCF insertion, query and deletion are $O(max \cdot \log m)$, $O(s \cdot k \cdot b \cdot \log m)$ and $O(s \cdot k \cdot b \cdot \log m)$, respectively.

CCF introduces the consistent hashing to achieve capacity elasticity. Therefore the time-complexity of query and deletion is not constant anymore. Basically, whenever we need to know the indices of candidate buckets for an element, CCF has to refer to the underlying consistent hash ring. In a real implementation, the hash values of these buckets are organized as a binary search tree. Consequently, given a hash value of an element, the corresponding candidate bucket will be searched out in $O(\log m)$ time. To insert an element into the active I2CF, at most *max* reallocations is allowed, thus the time-complexity is $O(max \cdot \log m)$. As for a query and deletion, CCF might have to go through all the I2CFs, therefore the resultant time-complexity is $O(s \cdot k \cdot b \cdot \log m)$.

Compared with DCF, the time-complexities of CCF are a little higher with an additional multiplicand of $\log m$. Note that logarithmic overhead, in fact, is acceptable in practice since the logarithmic function increases slowly when the value of *m* grows drastically. Distributed systems that employ the consistent hashing techniques all incur $\log$ level complexity. They are proved to function well. Examples include Akamai [32], Swift [33], Dynamo [34], etc.

B. Threshold for CCF Insertion

Each I2CF in CCF can be extended or downsized by adding or removing buckets dynamically. But for a static I2CF with

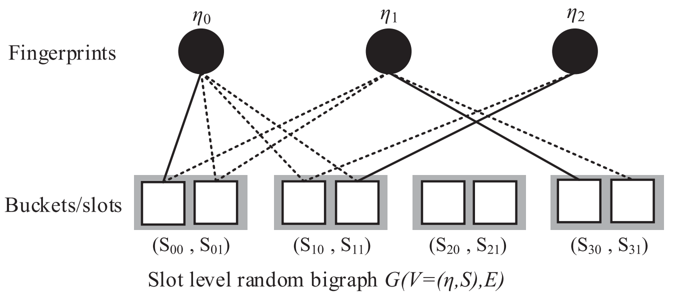


Fig. 3. An instance of slot level random bigraph for an I2CF$_i$ with m_i=4, k_i=2, b_i=2 and n_i=3. A “mapping conflict” happens to η_2 since only bucket 1 is assigned to it. The three solid edges forms a complete matching, meaning that the three fingerprints can stored successfully.

given parameters, we need to explore how many fingerprints can be successfully inserted. That is, given the number of elements to be represented n_i, a derivative problem is to seek a threshold T_i for the ratio between n_i and m_i. When $\frac{n_i}{m_i} \le T_i$, I2CF$_i$ can successfully store the n_i elements with probability $1-o(1)$; otherwise, I2CF$_i$ may fail to record all the n_i elements with probability $1-o(1)$.

The mapping between elements and buckets in I2CF$_i$ can be abstracted as a k_i-uniform hypergraph with m_i nodes and n_i hyperedges each of which is of fixed size k_i chosen independently from the m_i nodes. Based on the *core theory* of hypergraph, T_i can be derived out as a function of k_i and b_i. The details are given in [29] and [39].

In fact, the resulted hypergraph may not be k_i-uniform since the k_i independent hash functions may select the same buckets from I2CF$_i$ for an element *x*. We call this phenomenon “mapping conflict”. These mapping conflicts violate the k_i-uniform assumption towards the hypergraph. However, [29] and [39] did not consider the impact of the potential mapping conflicts. Therefore, in this paper, we present a novel new abstraction to I2CF and other Cuckoo filter-like data structures.

We notice that an I2CF$_i$ can be naturally represented as a slot level random bipartite graph (or bigraph) $G(V=(\eta, S), E)$, where η and *S* denote the fingerprints to be stored and the slots in I2CF$_i$, respectively. As shown in Fig. 3, each slot has two subscripts which demonstrate its host bucket and its location in that bucket. For example, S_{01} means the second slot of the first bucket. In the bigraph, edges demonstrate the assignment between the fingerprints and slots. If a bucket is a candidate bucket of a fingerprint, all slots of the bucket have an edge to that fingerprint, to explicitly indicate that these slots can be employed to store that fingerprint. In the generated bigraph, a matching indicates a possible way to store these fingerprints. Besides, this abstraction naturally provides us an important theorem in bigraphs, i.e., Hall's Theorem [40].

Theorem 3: (Hall's Theorem) Let $G(V=(X, Y), E)$ be a bigraph with bipartite sets *X* and *Y*. For a set of nodes $W \subseteq X$, let $N_G(W)$ denote the set of neighbors of *W* in *G*, i.e., the set of all nodes in *Y* which are adjacent to some element of *W*. There is a matching that entirely covers *X* if and only if for every subset *W* of *X*:

$$|W| \le |N_G(W)|. \quad (3)$$

Additionally, referring to the parameters of I2CF$_i$ as m_i, k_i, b_i and the number of fingerprints to be inserted n_i, we derive the following observations.

Observation 1: For an insertion of an arbitrary element *x*, let $\Theta \in [0, k_i]$ denote the total times that element *x* is mapped into a bucket. The value of Θ follows a typical binomial distribution since the employed hash functions are independent. Specifically, the probability that $\Theta = \theta$ can be calculated as:

$$p\{\Theta = \theta\} = \binom{k_i}{\theta}\left(\frac{1}{m_i}\right)^{\theta}\left(1 - \frac{1}{m_i}\right)^{(k_i - \theta)}. \tag{4}$$

Let p_0 denote the probability that an element *x* is mapped into the bucket. As $\Theta \geq 1$ means *x* is mapped into the bucket, the value of p_0 can be derived out as:

$$p_0 = 1 - p\{\Theta = 0\} = 1 - \left(1 - \frac{1}{m_i}\right)^{k_i}. \tag{5}$$

Observation 2: Let $\Phi \in [0, n_i]$ denote the total number of elements mapped into a bucket. Then the value of Φ also follows a typical binomial distribution since the insertions of elements are independent. To be specific, the probability that $\Phi = \phi$ is:

$$p\{\Phi = \phi\} = \binom{n_i}{\phi} p_0^{\phi}(1 - p_0)^{(n_i - \phi)}. \tag{6}$$

By jointly considering the observations and Hall's Theorem, we provide a new threshold $\hat{T}_i$ for an I2CF$_i$. When $\frac{n_i}{m_i} < \hat{T}_i$, the fingerprints can be stored successfully with high probability; by contrast, when $\frac{n_i}{m_i} \geq \hat{T}_i$, I2CF$_i$ may fail to store some fingerprints with high probability.

Theorem 4: If $\Phi < b_i$, it is impossible for I2CF$_i$ to store any fingerprint with the remained $b_i - \Phi$ slots. Then, the fraction of space that may be utilized for a bucket is:

$$\hat{u} = 1 - \sum_{\phi=0}^{b_i - 1}\left(1 - \frac{\phi}{b_i}\right) p\{\Phi = \phi\}, \tag{7}$$

where $1 - \frac{\phi}{b_i}$ is the fraction of bucket space which will never be utilized when $\Phi < b_i$. By contrast, if all the fingerprints are successfully stored, then the space utilization of I2CF$_i$ is:

$$\overline{u} = \frac{n_i}{m_i \cdot b_i}. \tag{8}$$

Then, the threshold $\hat{T}_i$ can be derived as the unique value of $\frac{n_i}{m_i}$ such that:

$$\hat{u} = \overline{u}. \tag{9}$$

Theorem 4 can be proved by jointly considering observation 1, observation 2 and the Hall's Theorem. Intuitively, when $\frac{n_i}{m_i}$ is less than $\hat{T}_i$, $\hat{u}$ is larger than $\overline{u}$, meaning there is sufficient space for the n_i fingerprints. As a result, when $\hat{u} \geq \overline{u}$, I2CF$_i$ satisfies the requirement of Hall's Theorem with high probability. In contrast, if $\frac{n_i}{m_i} \geq \hat{T}_i$, then $\hat{u} \leq \overline{u}$ and the space that can be utilized is not sufficient enough to accommodate the n_i fingerprints. In this situation, I2CF$_i$ will fail to satisfy Hall's Theorem with high probability. As shown in Fig. 4, given m_i=50 and b_i=2, the value of $\hat{T}_i$ grows significantly when k_i increases. Besides, with given m_i and k_i, the growth of b_i results in increasing of $\hat{T}_i$. Table II further presents the derived $\hat{T}_i$ when m_i is taken as 2^{30} while k_i and b_i varies. This threshold provides a guide for the users of CCF and I2CF for their parameter settings in practice. Intuitively, larger $\hat{T}_i$ guarantees higher space utilization. Therefore, given the same value of b_i, I2CF has the potentiality of realizing better space utilization than DCF by increasing the value of k_i.

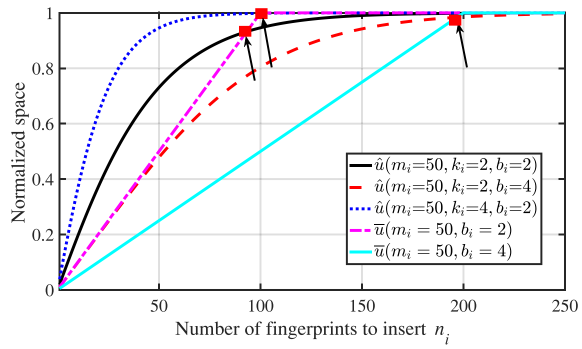


Fig. 4. The $\hat{T}_i$ with diverse parameter settings. The value of $\hat{T}_i$ can be derived out by dividing the x-axis value of the crossover point with b_i.

TABLE II
THE THRESHOLD $\hat{T}_i$ FOR I2CF$_i$ (WHEN $m_i = 2^{30}$)

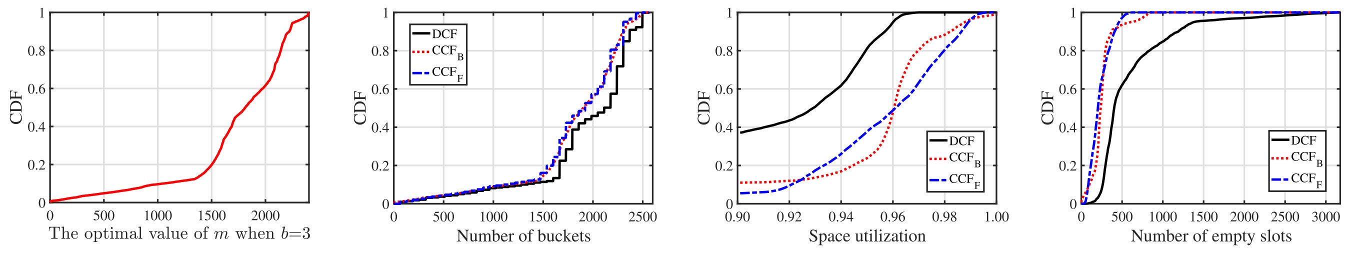
$k_i \backslash b_i$	1	2	3	4
2	0.796812130	1.861790807	2.905683863	3.934728166
3	0.940479791	1.979049536	2.992264312	3.997079786
4	0.980172599	1.996604114	2.999390447	3.999888473
5	0.993022846	1.999453835	2.999955482	3.999996295
6	0.997483538	1.999913939	2.999996938	3.999999888
7	0.999082240	1.999986694	2.999999798	3.999999996

C. Probability of a Successful Representation

Theorem 4 and [39] present the threshold of $\frac{n_i}{m_i}$ for a given I2CF. When $\frac{n_i}{m_i}$ is less than the threshold, the n_i fingerprints can be successfully stored with high probability. However, they fail to settle the derived question: What exactly is the probability of successfully inserting n_i fingerprints with a given I2CF, or alternatively, how to derive bound of that probability? We try to answer this question here.

Observation 3: For n_i given fingerprints, the number of edges in the maximum matching of the resultant $G(V{=}(\eta, S), E)$ implies the maximum number of fingerprints which can be successfully inserted into I2CF$_i$. If the maximum matching is a complete matching, then all the given fingerprints can be successfully stored by I2CF$_i$.

Note that the maximum matching in a specific bigraph can be solved by existing algorithms such as the Hungarian algorithm [41], Ford-Fulkerson algorithm [42], Hopcroft-Karp algorithm [43], etc. Let Ψ be a variable describing the number of successful inserted fingerprints in an I2CF$_i$ with the parameters of m_i, n_i, k_i and b_i. A brute force method to calculate the probability distribution of Ψ is possible by exploring the probability space of the $G(V{=}(\eta, S), E)$ and then count those bigraphs in which the maximum matching contains a number of $\Psi = \psi$ edges. This method, however, suffers from exponentially growing time complexity since it has to test all the $m_i^{n_i \cdot k_i}$ possible bigraphs. Therefore, alternatively, we derive out an



(a) The CDF of m_{opt} when b=3. (b) The CDF of # buckets. (c) The CDF of space utilization. (d) The CDF of # empty slots.

Fig. 5. The comparison between CCF and DCF with Yahoo! trace. The CDFs are calculated by counting the frequency of a specific result of metric.

upper bound of $p\{\Psi{=}n_i\}$ ($n_i{\in}[1, m_i{\cdot}b_i]$) based on the Hall's Theorem [40] and observation 3.

Theorem 5: For a given I2CF$_i$ with m_i, k_i, b_i and n_i ($n_i{\in}[1, m_i{\cdot}b_i]$) fingerprints to be inserted, the probability that all the n_i fingerprints can be successfully accommodated has the following upper bound:

$$p\{\Psi{=}n_i\} \le \sum_{j=\lceil n_i/b_i\rceil}^{\max\{n_i\cdot k_i, m_i\}} p\{\Omega = j\}, \qquad (10)$$

where $p\{\Omega = j\}$ denotes the probability that the n_i fingerprints are mapped into exactly *j* buckets of I2CF$_i$. The value $p\{\Omega = j\}$ can be calculated as:

$$p\{\Omega{=}j\} = \frac{\binom{m_i}{j}\sum_{l=0}^{F(j,n_i,k_i)}\left[D_l\prod_{r=0}^{j-1}\binom{n_i-\sum_{q=0}^{r}Q[l][q]}{Q[l][r]}\right]}{m_i^{n_i\cdot k_i}}, \qquad (11)$$

where Q is an array of vectors, each with *j* positive integers such that the sum of these integers is exactly $n_i{\cdot}k_i$. The number of vectors in *Q* is denoted as $F(j, n_i, k_i)$ and can be calculated with the input value of *j*, n_i and k_i. D_l is the number of possible combinations of the *j* integers in $Q[l]$. The factor $\prod_{r=0}^{j-1}\binom{n_i-\sum_{q=0}^{r}Q[l][q]}{Q[l][r]}$ counts all possible cases when the $n_i{\cdot}k_i$ mappings are distributed into the selected *j* buckets according to the distribution given by $Q[l]$.

Theorem 5 can be proved by considering both observation 3 and the Hall's Theorem. Basically, $p\{\Omega{=}j\}$ only counts the probability that the n_i fingerprints are mapped into exactly *j* buckets, but fails to consider the situation where a subset of the n_i fingerprints may not satisfy Hall's Theorem. Therefore, Eq. (10) offers the upper bound of $p\{\Psi{=}n_i\}$. Eq. (11) can be derived out by regarding each mapping as a ball then formulating it as a typical balls and bins problem.

We give a walk-through example by calculating the upper bound of $p\{\Psi{=}3\}$ with $m_i{=}5$, $b_i{=}2$, $k_i{=}2$ and $n_i{=}3$, respectively. From Eq. (10), we have $p\{\Psi{=}3\}{\le}p\{\Omega{=}2\}{+}p\{\Omega{=}3\}{+}p\{\Omega{=}4\}{+}p\{\Omega{=}5\}$. Then, by Eq. (11), $p\{\Omega{=}2\}{=}0.03968$, $p\{\Omega{=}3\}{=}0.3456$, $p\{\Omega{=}4\}{=}0.4992$, and $p\{\Omega{=}5\}{=}0.1152$. Therefore, the upper bound for $p\{\Psi{=}3\}$ is 0.99968. When calculating $p\{\Omega{=}3\}$, we have n_ik_i=6=1+1+4=1+2+3=2+2+2. Thus $F(j, n_i, k_i){=}3$, $Q{=}\{[1,1,4],[1,2,3],[2,2,2]\}$, $D_0{=}3$ since [1,1,4] has three permutations: {1, 1, 4}, {1, 4, 1}, and {4, 1, 1} such that $D_1{=}6$ and $D_2{=}1$. Consequently, $p\{\Omega{=}3\}{=}[\binom{5}{3}\cdot(3\cdot\binom{6}{1}\binom{5}{1}{+}6\cdot\binom{6}{1}\binom{5}{2}{+}\binom{6}{2}\binom{4}{2})]/5^6{=}5400/15625{=}0.3456$.

With the above analysis, we provide a better understanding of the proposed data structures, as well as a guide to the potential users about the parameter settings whenever CCF or I2CF are within their considerations.

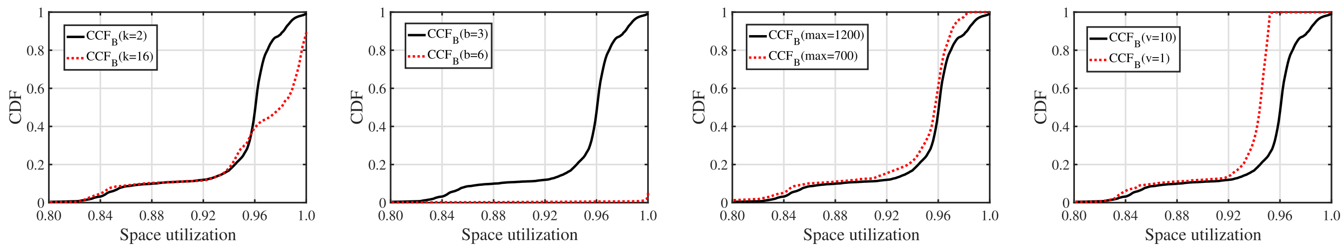
VI. Evaluation

In this section, we compare the performance of CCF with DCF for dynamic set representation and then quantify the impact of the parameters. All of the experiments are conducted on a host with 8 GB RAM and 3.4 GHz CPU. Especially, we conduct the evaluations based on the real network flow trace from Yahoo! [44]. The Yahoo! trace records the basic information for each flow in its 6 distributed data centers, including the IP addresses of both source and destination servers, the arriving and terminating time of each flow, etc. In this paper, we regard the combination of the source and destination IP addresses as the content of an element, coupled with the start and end time. We refer to a time period of 20 minutes from the sketch. In this period, there are in total 58,941 flows and Fig. 5(a) shows the CDF of the optimal number of buckets in theory m_{opt}. We have $m_{opt}{=}\lceil n/b\rceil$, since each bucket can store *b* fingerprints at most.

A. Comparison With DCF

We implement two versions of CCF, i.e., CCF with only bucket-level alteration CCF$_B$ and CCF with only filter-level alteration CCF$_F$. For a fair comparison, the parameters for CCF$_F$ and DCF are set as the same, i.e., $m_i{=}64$, $b_i{=}3$, $k_i{=}2$, $f{=}30$. For both CCF$_B$ and CCF$_F$, the value of *M* and the number of virtual nodes in the consistent hash ring *v* are given as 5×10^{10} and 10 respectively. Fig. 5(b), (c) and (d) depict the CDF of the resultant number of buckets, space utilization and the number of empty slots, respectively.

By jointly considering Fig. 5(a) and (b), we characterize the capacity elasticity of DCF and CCF. Obviously, CCF$_B$ achieves the best elasticity and keeps capacity up and down whenever the number of elements increases or decreases. The curve of CCF$_B$ in Fig. 5(b) matches the variation of m_{opt} in Fig. 5(a) perfectly. The CCF$_F$ also responds to the changes of m_{opt} rapidly by executing its compact and scale out algorithms dynamically. The DCF, however, fails to compact under-utilized CFs instantly when the value of m_{opt} decreases. The reason is that DCF moves any fingerprint in the under-utilized CF into its corresponding buckets of other CFs. Therefore, successful compaction is hard to achieve. Our



(a) Impact of # candidate buckets *k*. (b) Impact of # slots in each bucket *b*. (c) Impact of # reallocations *max*. (d) Impact of # virtual nodes *v*.

Fig. 6. The impact of different parameters in CCF$_B$ with the Yahoo! trace.

CCF$_F$, in contrast, always tries to insert the fingerprints in an under-utilized I2CF into other I2CFs, thereby releasing the fingerprints from their locations in the under-utilized I2CF. Thus, both CCF$_B$ and CCF$_F$ have better elasticity than DCF.

Moreover, the CDF of space utilization is depicted in Fig. 5(c). For DCF, about 37% of resultant space utilization is less than 0.90. However, CCF$_B$ and CCF$_F$ have less than 10% of results which are below 0.90. Moreover, the maximum space utilization for DCF is 0.970, which is much lower than that of CCF$_B$ (1.0) and CCF$_F$ (0.999). To be accurate, on average, the space utilization for DCF, CCF$_B$ and CCF$_F$ are 0.8809, 0.9481 and 0.9425, respectively. Correspondingly, the CDF of the number of empty slots is shown in Fig. 5(d). For CCF$_B$ and CCF$_F$, 93% and 97% results have less than 500 empty slots, while that value for DCF is 62% only. In the worst case, DCF remains 3,176 slots empty. More than 16% DCF results incur more than 1,000 empty slots. The reason is that DCF can only compact an under-utilized filter if all the stored fingerprints find their corresponding unfilled buckets in other CFs. As a result, when the value of *n* decreases but DCF may fail to recycle under-utilized CFs timely. Notice that CCF$_B$ has more empty slots than CCF$_F$. The reason is that we only try to merge the buckets which store less than 2 fingerprints in our experiments. At the end of our tests, due to the removals of flows, the proportion of buckets which accommodate 2 fingerprints gets higher, while CCF$_B$ doesn't recycle the empty slots immediately.

From the above experiments, we conclude that CCF achieves better capacity elasticity and higher space utilization than DCF. The feature of design flexibility, on the other hand, may not be quantified directly. Intuitively, DCF only adds or merges homogeneous CFs, while the I2CFs in CCF is allowed to have diverse parameter settings. This flexibility makes CCF more suitable for dynamic set representation than DCF.

B. Impact of Parameters in CCF$_B$

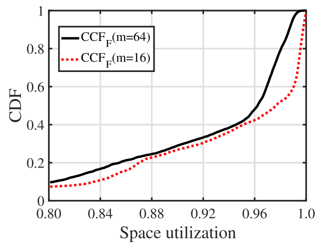
In this subsection, we quantify the impact of parameters for CCF$_B$. Especially, we consider four main parameters, i.e., the number of candidate buckets *k*, the number of slots in a bucket *b*, the maximum reallocations *max*, and the number of virtual nodes in the consistent hash ring *v*. Note that, we evaluate the space utilization of CCF$_B$ when the above parameters vary. The reference setting is a CCF$_B$ with *k*=2, *b*=3, *max*=1200, and *v*=10. We then vary the four parameters separately and show the results in Fig. 6.

As shown in Fig. 6(a), when *k* increases from 2 to 16, CCF achieves higher space utilization (rising from 0.9481 to 0.9599 on average). When k=16, nearly half of the results have more than 0.98 space utilization. However, less than 12% of the results achieve more than 0.98 space utilization when *k*=2. An element with more candidate buckets implies that a bucket may be assigned to more elements. Consequently, the probability that a bucket is assigned to less than *b* elements gets lower, which leads to higher space utilization. When *b* is increased from 3 to 6, we can see from Fig. 6(b) that the space utilization increases dramatically. To be specific, the space utilization is 0.9481 for *b*=3, but 0.9986 for *b*=6 on average. This phenomenon is reasonable since, with a larger *b*, there are fewer buckets in the CCF$_B$. In the Yahoo! trace, the maximum number of flows to store is about 7,290. So *max*=1,200 means the reallocations when inserting an element may cover the whole filter to explore potential empty slots. Also, with less number of buckets in the filter, the probability of buckets assigned to less than *b* elements gets lower. Accordingly, the resultant space utilization increases.

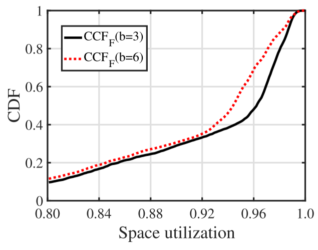
When the value of *max* is decreased from 1200 to 700, the CDF of the resultant space utilization is recorded in Fig. 6(c). Obviously, with more allowed reallocations, CCF achieves higher space utilization. Namely, with larger *max*, the insertion will search more buckets, and hopefully CCF may find an empty slot to store the fingerprint. Moreover, as depicted in Fig. 6(d), when the number of virtual nodes in the consistent hash ring decreases from 10 to 1, the space utilization experiences a significant drop (decreasing from 0.9481 to 0.9298 on average). When *v*=1, only about 16% of the results realize more than 0.95 space utilization. By contrast, when v=10, about 76% of the results realize more than 0.95 space utilization. Basically, with more virtual nodes, consistent hashing generates better load balance among the buckets. Therefore, the probability that a bucket is assigned with less than *b* elements gets lower and thereby resulting in higher space utilization.

C. Impact of Parameters in CCF$_F$

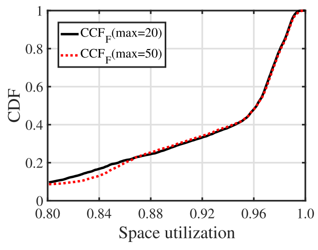
We further quantify the impact of parameters for CCF$_F$ in terms of space utilization. The parameters we considered include the number of buckets in each I2CF *m*, the number of slots in a bucket *b*, the maximum reallocations *max*, and the number of virtual nodes in the consistent hash ring *v*. The reference setting is a CCF$_F$ with *m*=64, *b*=3, *max*=20, and



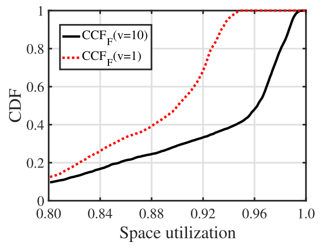
(a) Impact of # buckets in I2CF m.



(b) Impact of # slots in each bucket b.



(c) Impact of # reallocations max.



(d) Impact of # virtual nodes v.

Fig. 7. The impact of different parameters in CCF$_F$ with the Yahoo! trace.

v=10. We then vary the four parameters separately and show the results in Fig. 7.

As depicted in Fig. 7(a), when m decreases from 64 to 16, CCF$_F$ achieves a better space utilization. The average space utilization increases from 0.912 to 0.931. About half (48.3% to be exact) of the results realize more than 0.98 space utilization when $m = 16$, while that proportion is just 22.4% when $m = 64$. The reason is that, smaller m implies more fine-grained capacity control when adding or merging I2CFs. For instance, when one extra I2CF is needed to represent 5 elements, adding an I2CF with 16 buckets is definitely more space-saving than introducing an I2CF with 64 buckets.

When the number of slots in each bucket, i.e., b, varies from 3 to 6, as depicted in Fig. 7(b), CCF$_F$ becomes a bit more space-consuming. Specifically, the average space utilization drops from 0.912 to 0.899. About 57% of the resultant space utilizations when $b = 6$ fall into the interval [0.920, 0.984]. When $b = 3$, however, 52% of the generated space utilizations are more than 0.940. This phenomenon is reasonable since increasing b with fixed m means more space resources when adding an untapped I2CF, and higher difficulty when merging under-utilized I2CFs.

Moreover, as recorded in Fig. 7(c), when we increase the maximum reallocation times for each insertion, i.e., *max*, from 20 to 50, the space utilization shows a minor growth (i.e., 0.912 to 0.913). Theoretically, let w denote the number of buckets with empty slots in an I2CF with m buckets. We suppose that the candidate buckets for the victims are random. Then the probability that the empty slot can be searched out from this I2CF is: $1 - \binom{m-w}{max}/\binom{m}{max}$. With given m and w, this probability indeed increases with the growth of *max*, however, in a marginal manner. Therefore, when *max* is already large enough, even if we attach a significant increment to it, the resulted space utilization may show only a slight growth.

Last, Fig. 7(c) plots the CDF of space utilization of CCF$_F$ when the value of v changes from 10 to 1. Apparently, the CCF$_F$ experiences a significant performance drop (the average space utilization degrades from 0.912 to 0.863). Especially, when $v = 1$, the best space utilization of CCF$_F$ is just 0.952 and half of the results achieve less than 0.90 space utilization. More virtual nodes indicates better load balance among the buckets in the consistent hashing ring. The probability that a bucket is assigned with less than b elements gets lower, therefore resulting in a higher space utilization.

From the above results, we conclude that the parameters of CCF have diverse impacts on its performance. The users can customize their own configurations to achieve their goals by leveraging these parameters.

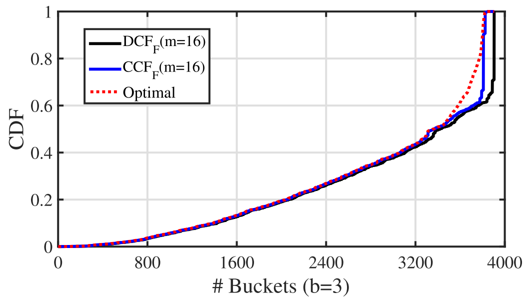
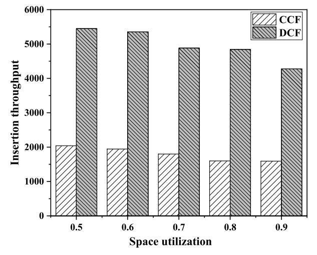


Fig. 8. The CDF of # buckets in the WIDE MAWI dataset.

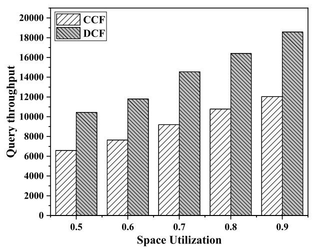
D. Further Comparison Between DCF and CCF$_F$

We further test CCF and DCF with a new flow dataset from the WIDE MAWI working group [45]. This trace contains 20,338,775 IPv4 packets collected by a WIDE MAWI sample point on June 10, 2020. We group these packets as 96,240 flows with the five tuples src.IP, dst.IP, src.port, dst.port, and protocol in packet headers. We add/remove the flows into/from DCF and CCF ($b = 3$, $m = 16$) according to their timestamps. To be fair, we implement the filter-level CCF and count the number of buckets whenever a flow is leaving. As shown in Fig. 8, CCF still outperforms the DCF significantly. To be specific, the optimal solution (a list that represents the flows with 100% space utilization) only needs 3,826 buckets at most. About 38.5% of results in DCF need more buckets than that. By contrast, this value in CCF is just 9%. The reason is that CCF can extend the filter more conservatively while recycling the space more aggressively.

Besides, we measure the insertion and query throughput of CCF (each bucket has 5 virtual nodes in the ring) and DCF when only one filter is maintained with identical parameter settings. The numerical results are shown in Fig. 9. As depicted in Fig. 9(a), when the space utilization goes up, the insertion throughput of both DCF and CCF decreases constantly, because more subsequent insertions may trigger the *kick-out-and-reallocate* process. The consistent hashing technique in CCF can significantly slow down the element insertion since more hash computations are required. DCF can be 2 to 3 times faster than CCF when representing elements. We then query the filters with a set that refers to the inserted elements as a subset. As shown in Fig. 9(b), DCF still outperforms CCF in

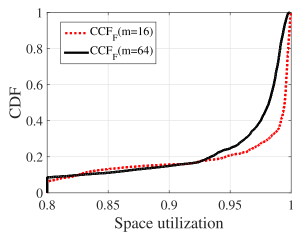


(a) The throughput of insertion.

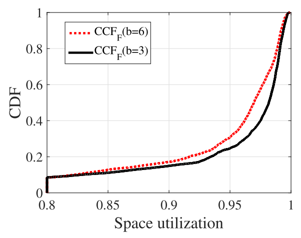


(b) The throughput of query.

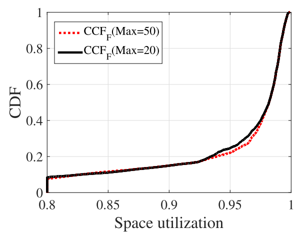
Fig. 9. The throughput of insertion and query with the WIDE MAWI dataset.



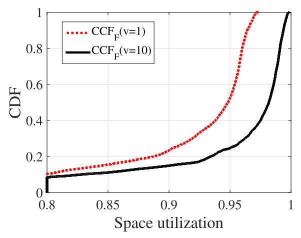
(a) Impact of # buckets in I2CF m.



(b) Impact of # slots in a bucket b.



(c) Impact of # reallocations max.



(d) Impact of # virtual nodes v.

Fig. 10. The impact of parameters in CCF$_F$ with the WIDE MAWI dataset.

Fig. 10. The impact of parameters in CCF$_F$ with the WIDE MAWI dataset.

a large scale. Since more inserted elements are queried, the resultant throughput increases when space utilization grows.

However, we argue that to obtain the features of capacity elasticity and design flexibility, such a compromise of insertion throughput may be acceptable for users with strict requirement of insertion speed. Further optimization techniques such as jump table [46] or red-black tree [47] can be employed to speed up CCF. The potential users can trade off such pros and cons according to their own needs.

We further quantify the impact of parameters in CCF$_F$ with the WIDE MAWI dataset. The reference setting is a CCF$_F$ with m=64, b=3, max=20, and v=10. We then vary the four parameters separately and record the results in Fig. 10. When $m = 16$, the CCF$_F$ realizes better space utilization due to finer-grained capacity expansion and recycling, as shown in Fig. 10(a). By contrast, given $m = 64$, increasing the number of slots b in each bucket results in a significant degrade of space utilization. The reason is that with the smaller capacity of each I2CF, CCF can introduce new filters or recycle under-utilized filters timely. Increasing max, on the contrary, brings limited space utilization growth to CCF$_F$. This is because the filters are already highly occupied and increasing the reallocation threshold presents a marginal effect. Lastly, enlarging the number of virtual nodes in the consistent hash ring from 1 to 10 can significantly increase the space utilization by providing better load balance among the buckets on the ring. The parameters show the same trends as the Yahoo! trace, meaning CCF is general to represent dynamic datasets.

VII. Conclusion

In this paper, we present the CCF design for dynamic set representation and membership query, with the targets of capacity elasticity, space efficiency, and design flexibility. CCF is composed of an adjustable number of I2CFs. At its core, each I2CF enables bucket-level capacity alteration with the use of consistent hashing. At the filter level, CCF resizes its capacity by adding untapped I2CFs or merging under-utilized I2CFs adaptively. Without any inner dependency and constraints, all the parameters of CCF are mutable and can be customized by its users. Theoretical analysis and trace-driven experiments show that CCF outperforms DCF and achieves the design rationales simultaneously at the cost of a little higher time-complexity. Users need to consider the gains and losses before implementing CCF in their systems.

References

[1] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 1, pp. 131–155, 1st Quart., 2012.
[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[3] L. Luo, D. Guo, R. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1912–1949, 2nd Quart., 2018.
[4] L. Luo, D. Guo, Y. Zhao, O. Rottenstreich, R. Ma, and X. Luo, "MCFsyn: A multi-party set reconciliation protocol with the marked cuckoo filter," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2705–2718, Nov. 2021.
[5] B. Fan, D. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proc. ACM CoNEXT*, Sydney, NSW, Australia, 2014, pp. 75–88.
[6] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, Jan. 2010.
[7] H. Chen, L. Liao, H. Jin, and J. Wu, "The dynamic cuckoo filter," in *Proc. IEEE ICNP*, 2017, pp. 1–10.
[8] D. Eppstein, "Cuckoo filter: Simplification and analysis," 2016. [Online]. Available: arXiv:1604.06067.
[9] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," *ACM J. Exp. Algorithmics*, vol. 25, no. 1, pp. 1–20, 2020.
[10] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proc. USENIX NSDI*, Lombard, IL, USA, 2013, pp. 371–384.
[11] D. Karger, E. Lehman, F. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. ACM STOC*, 1997, pp. 654–663.
[12] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup protocol for Internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003.
[13] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.
[14] O. Rottenstreich, "Sketches for blockchains," in *Proc. COMSNETS*, 2021, pp. 254–262.
[15] O. Rottenstreich and I. Keslassy, "The Bloom paradox: When not to use a Bloom filter?" in *Proc. IEEE INFOCOM*, Orlando, FL, USA, 2012, pp. 1638–1646.
[16] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," in *Proc. ACM SIGCOMM*, London, U.K., 2015, pp. 52–66.
[17] H. Lim, K. Lim, N. Lee, and K. H. Park, "On adding Bloom filters to longest prefix matching algorithms," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 411–423, Feb. 2014.

[18] C. Rothenberg, C. Macapuna, M. Magalhaes, F. Verdi, and A. Wiesmaier, "In-packet Bloom filters: Design and networking applications," *Comput. Netw.*, vol. 55, no. 6, pp. 1364–1378, 2011.

[19] J. Trindade and T. Vazao, "HRAN: A scalable routing protocol for multihop wireless networks using bloom filters," in *Proc. Wired/Wireless Internet Commun.*, 2011, pp. 434–445.

[20] X. Li, J. Wu, and J. Xu, "Hint-based routing in WSNs using scope decay bloom filters," in *Proc. IEEE NAS*, Shenyang, China, 2006, pp. 111–118.

[21] F. Angius, M. Gerla, and G. Pau, "Bloogo: Bloom filter based Gossip algorithm for wireless NDN," in *Proc. ACM Workshop Emerg. Name Orient. Mobile Netw. Design Architect. Algorithms Appl.*, Hilton Head, SC, USA, 2012, pp. 1–9.

[22] L. Calderoni, P. Palmieri, and D. Maio, "Location privacy without mutual trust: The spatial Bloom filter," *Comput. Commun.*, vol. 68, pp. 4–12, Sep. 2015.

[23] M. Gomez-Barrero, C. Rathgeb, J. Galbally, J. Fierrez, and C. Busch, "Protected facial biometric templates based on local Gabor patterns and adaptive Bloom filters," in *Proc. ICPR*, Stockholm, Sweden, 2014, pp. 4483–4488.

[24] E. Oriero, K. Rabieh, M. Mahmoud, M. Ismail, E. Serpedin, and K. Qaraqe, "Trust-based and privacy-preserving fine-grained data retrieval scheme for MSNs," in *Proc. IEEE WCNC*, 2016, pp. 1–6.

[25] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better NetFlow for data centers," in *Proc. USENIX NSDI*, Santa Clara, CA, USA, 2016, pp. 311–324.

[26] R. Pagh and F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[27] N. Nguyen and P. Tsigas, "Lock-free cuckoo hashing," in *Proc. IEEE ICDCS*, Madrid, Spain, 2014, pp. 627–636.

[28] M. Dietzfelbinger and C. Weidling, "Balanced allocation and dictionaries with tightly packed constant size bins," *Theor. Comput. Sci.*, vol. 380, nos. 1–2, pp. 47–68, 2007.

[29] N. Fountoulakis, M. Khosla, and K. Panagiotou, "The multiple-orientability thresholds for random hypergraphs," in *Proc. ACM-SIAM SODA*, San Francisco, CA, USA, 2011, pp. 1222–1236.

[30] J. Zentgraf, H. Timm, and S. Rahmann, "Cost-optimal assignment of elements in genome-scale multi-way bucketed Cuckoo hash tables," in *Proc. SIAM ALENEX*, 2020, pp. 186–198.

[31] S. Pontarelli, P. Reviriego, and J. Maestro, "Parallel *D*-pipeline: A cuckoo hashing implementation for increased throughput," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 326–331, Jan. 2016.

[32] L. Wang, V. Pai, and L. Peterson, "The effectiveness of request redirection on CDN robustness," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 345–360, 2002.

[33] T. Chekam, E. Zhai, Z. Li, Y. Cui, and K. Ren, "On the synchronization bottleneck of OpenStack Swift-like cloud storage systems," in *Proc. IEEE INFOCOM*, San Francisco, CA, USA, 2016, pp. 1–9.

[34] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proc. ACM SOSP*, Stevenson, WA, USA, 2007, pp. 205–220.

[35] W. Xie and Y. Chen, "Elastic consistent hashing for distributed storage systems," in *Proc. IEEE IPDPS*, Orlando, FL, USA, 2017, pp. 876–885.

[36] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," *Proc. VLDB Endow.*, vol. 13, no. 2, pp. 197–210, 2019.

[37] A. D. Breslow and N. Jayasena, "Morton filters: Fast, compressed sparse cuckoo filters," *VLDB J.*, vol. 29, nos. 2–3, pp. 731–754, 2020.

[38] P. Fu, L. Luo, S. Li, D. Guo, G. Cheng, and Y. Zhou, "The vertical cuckoo filters: A family of insertion-friendly sketches for online applications," in *IEEE ICDCS*, 2021, p. 9.

[39] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, "Tight thresholds for cuckoo hashing via XORSAT," in *Proc. ICALP*, Bordeaux, France, 2010 pp. 213–225.

[40] P. Hall, "On representatives of subsets," *J. London Math. Soc.*, vol. 1, no. 1, pp. 26–30, 1935.

[41] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Res. Logist. Quart.*, vol. 1, nos. 1–2, pp. 83–97, 1955.

[42] L. Ford and D. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton Univ. Press, 1962.

[43] J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs," *SIAM J. Comput.*, vol. 2, no. 4, pp. 225–231, 1973.

[44] M. Alizadeh, A. Greenberg, D. A. Maltz, and J. Padhye, "DCTCP: Efficient packet transport for the commoditized data center," in *Proc. ACM SIGCOMM*, New Delhi, India, 2010, pp. 63–74.

[45] *WIDE MAWI WorkingGroup, Traffic Trace Info*. Accessed: Jan. 10, 2021. [Online]. Available: https://mawi.wide.ad.jp/mawi/samplepoint-G/2020/202006101400.html

[46] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," 2014. [Online]. Available: arXiv:1406.2294.

[47] K. Elshad, "Red–black tree," in *Data Structures and Algorithms in Swift*. Berkeley, CA, USA: Apress, 2020, pp. 101–120.

Lailong Luo received the B.S. and M.S. degree with the College of Systems Engineering, National University of Defense Technology, Changsha, China, in 2013 and 2015, respectively, where he is currently pursuing the Ph.D. degree. His research interests include data structure and distributed networking systems.

Deke Guo (Senior Member, IEEE) received the B.S. degree in industry engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2008, where he is currently a Professor with the College of Systems Engineering. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of the ACM.

Ori Rottenstreich (Member, IEEE) received the B.Sc. degree in computer engineering (*summa cum laude*) and the Ph.D. degree from the Technion, Haifa, Israel, in 2008 and 2014, respectively. From 2015 to 2017, he was a Postdoctoral Research Fellow with the Department of Computer Science, Princeton University and from 2017 to 2019, he was the Chief Scientist of Orbs, a blockchain startup company. He is currently an Assistant Professor with the Department of Computer Science and the Department of Electrical Engineering, Technion. His research interests include computer networks and blockchain algorithms.

Richard T. B. Ma (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from Columbia University, New York, in May 2010. During his Ph.D. study, he worked as a Research Intern with IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, and Telefonica Research, Barcelona, Spain. He is currently a Research Scientist with the Advanced Digital Science Center, University of Illinois, USA, and an Assistant Professor with the School of Computing, National University of Singapore. His research interests include distributed systems and network economics.

Xueshan Luo received the B.E. degree in information engineering from the Huazhong Institute of Technology, Wuhan, China, in 1985, and the M.S. and Ph.D. degrees in system engineering from the National University of Defense Technology, Changsha, China, in 1988 and 1992, respectively, where he is currently a Professor with the College of Systems Engineering. His research interests are in the general areas of information system and operation research.

Bangbang Ren received the B.S. degree and the M.S. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2015, where he is currently working pursuing the Ph.D. degree with the College of Systems Engineering. His research interests include software-defined networking, network function virtualization, and approximation algorithm.