# AnchorHash: A Scalable Consistent Hash

Gal Mendelson, Shay Vargaftik, Katherine Barabash, Dean Lorenz, Isaac Keslassy, and Ariel Orda

***Abstract*—Consistent hashing is a central building block in many networking applications, such as maintaining connection affinity of TCP flows. However, current consistent hashing solutions do not ensure full consistency under arbitrary changes or scale poorly in terms of memory footprint, update time and key lookup complexity.**

**We present AnchorHash, a scalable and fully-consistent hashing algorithm. AnchorHash achieves high key lookup rate, low memory footprint and low update time. We formally establish its strong theoretical guarantees, and present an advanced implementation with a memory footprint of only a few bytes per resource. Moreover, evaluations indicate that AnchorHash scales on a single core to 100 million resources while still achieving a key lookup rate of more than 15 million keys per second.**

***Index Terms*—Consistent hashing, load balancing.**

## I. Introduction

**Background.** Consistent hashing (CH) aims at mapping the identifiers (keys) of incoming objects into a set of resources, while achieving (1) *minimal disruption*, *i.e.,* minimum mapping changes as resources are arbitrarily removed or added, and (2) *balance*, *i.e.,* even spreading of the keys across resources such that no resource is overloaded.

CH is a central building block in many networking applications, such as datacenter load balancing, distributed hash tables, and distributed storage [1]–[7]. For instance, it is used by L4 load-balancers to evenly forward incoming packets to servers, while maintaining the affinity of TCP connections while servers are removed or added. However, as we next describe, despite recent advances [1], [8], [9], current CH solutions do not ensure full consistency under arbitrary changes or scale poorly in terms of memory footprint, update time and key lookup complexity.

**Related Work.** Consistent hashing was first introduced in the context of caching using the Ring algorithm (also called Consistent Hashing) [10], [11]. Several variations of the traditional Ring algorithm have been suggested in the literature to improve balance, *e.g.,* [12], [13]. Such Ring-based solutions face scalability issues, since they require a significant memory footprint and an increasing key lookup complexity.

Another well-known CH algorithm is Highest-Random-Weight (HRW) [14], also designed with the goal of increasing cache hit rates. It was later applied in the design of a location service for wireless networks [15] as well as in data storage systems [16]. While HRW offers good balance and small memory footprint, its computational complexity is prohibitive.

To achieve high key-lookup rate, MaglevHash [1] and similar techniques (*e.g.,* [8], [9]) rely on large memory tables.

Gal Mendelson is with the Technion and IBM Research, e-mail: galmendelson@gmail.com. Shay Vargaftik is with VMware Research, e-mail: shayv@vmware.com. Katherine Barabash and Dean Lorenz are with IBM Research, e-mail: {dean@il, kathy@il}.ibm.com. Isaac Keslassy and Ariel Orda are with the Technion, e-mail: {isaac@ee, ariel@ee}.technion.ac.il.

| | | HRW [14] | Ring [10] | Maglev-Hash [1] | **Anchor-Hash** |
|---|---|---|---|---|---|
| **Consistency** | **Minimal disrupt.** | ✓ | ✓ | × | ✓ |
| | **Balance** | ✓ | ✓* | ✓* | ✓ |
| **Scalability** | **Lookup rate** | × | × | ✓ | ✓ |
| | **Memory** | ✓ | ✓* | ✓* | ✓ |
| | **Update time** | ✓ | ✓* | × | ✓ |
| **Statelessness** | | ✓ | ✓ | ✓ | × |

TABLE I: Comparison of AnchorHash and common CH algorithms. Existing algorithms sacrifice full consistency and/or scalability, while AnchorHash aims at providing both. As we later show, AnchorHash leverages state information to achieve its properties and therefore is not stateless.

These solutions sacrifice full consistency, memory footprint, and update time upon resource additions and removals.

Several additional algorithms are designed for special cases where resources cannot be removed or added arbitrarily. For example, Jump Consistent Hash [17] assumes that resources can only be added or removed in a specific order. Two additional approaches that do not support resource additions are considered in [18]. The second approach shares some design features with our algorithm, but its implementation cannot scale due to a large memory footprint.

**AnchorHash.** In this paper, we present *AnchorHash*, a new hashing technique that guarantees minimal disruption, balance, high lookup rate, low memory footprint, and fast update time after resource additions and removals. Table I shows how AnchorHash is the only algorithm to achieve these goals at once. As opposed to the other algorithms, AnchorHash's decisions depend on past events in the system.

We first introduce AnchorHash, which hashes the incoming object's key into successively smaller sets of resources until eventually obtaining its unique mapped resource. We show how AnchorHash stays consistent under arbitrary resource removals and additions by keeping some history (Sec. III).

Then, we formally prove that AnchorHash is consistent, *i.e.,* guarantees minimal disruption and balance. We further prove that the average number of required hash computations in a key lookup depends only on the fraction of randomly failed resources and not on their absolute number. This allows for a very high key lookup rate at scale. We prove that even under extreme failure conditions, where $50\%$ of resources are removed in an adversarial manner, a key lookup by AnchorHash still requires less than $\approx 1.69$ hash computations on average and admits a low standard deviation of less than $\approx 0.83$ (Sec. IV).

Next, we focus on implementing AnchorHash. Using several successive improvements in the data representation structures, we show how AnchorHash can be reduced to an $O(1)$ memory footprint per resource, at the cost of a slight increase in complexity (Sec. V).

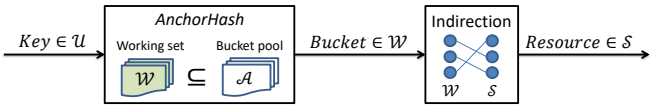



Fig. 1: AnchorHash uses indirection in order to compute the key-to-resource mapping. It first sets a bijective mapping between buckets and resources (right side), and then computes for each incoming key a key-to-bucket mapping (left side).

We then evaluate AnchorHash as well as HRW [14], Ring [10] and MaglevHash [1], using the criteria in Table I. AnchorHash and MaglevHash are the only algorithms that achieve high key lookup rate at scale, but MaglevHash sacrifices its consistency, and also requires a high memory footprint and a prohibitive update time. On the other hand, AnchorHash achieves low memory footprint and a negligible update time. In fact, we find that AnchorHash scales on a single core to 100 million resources while achieving a key lookup rate of more than 15 million keys per second (Sec. VI). Finally, the code for AnchorHash appears in [19].

## II. Preliminaries

We wish to map object keys to resources. Let $\mathcal{U}$ denote the set of keys, and let $\mathcal{S}$ denote the current set of resources. For example, in the context of datacenter L4 load-balancing, keys may correspond to packet 5-tuples and resources to servers.

**Mapping keys to resources.** As Fig. 1 illustrates, we use *indirection* by first mapping keys to *buckets*, then buckets to resources. Specifically, current existing resources, *i.e.,* members of $\mathcal{S}$, are assigned to buckets in a one-to-one correspondence. Buckets belong to a set denoted by $\mathcal{A}$, whose size corresponds to the number of available resources in the system, *e.g.,* the number of servers in the load balanced cluster. Let $\mathcal{W} \subseteq \mathcal{A}$ denote the subset of buckets that are currently assigned to resources, which we call the *working set*. We refer to buckets in $\mathcal{W}$ as *working* buckets. Note that the set $\mathcal{A}$ of all possible buckets is fixed, while its subset $\mathcal{W}$ changes upon resource removals/additions, *e.g.,* due to server failures or maintenance operations. Thus the mapping can be decomposed into two parts:
*(i) Keys to buckets.* A key is first mapped to a bucket in $\mathcal{W}$.
*(ii) Buckets to resources.* The corresponding resource in $\mathcal{S}$ is deduced from the bucket using the indirection.

**Resource removal and addition.** Resources can be added and removed arbitrarily. Upon a removal, the corresponding (bucket,resource) pair is removed from the indirection, and the bucket is removed from $\mathcal{W}$. When a resource is added, it is assigned a bucket in $\mathcal{A}\backslash\mathcal{W}$, the bucket is added to $\mathcal{W}$ and the pair (bucket,resource) is added to the indirection.

Note that a resource removal uniquely determines the bucket to remove from $\mathcal{W}$. However, when a resource is added, due to the indirection, any bucket in $\mathcal{A}\backslash\mathcal{W}$ can be added. This property is one of the building blocks we use to construct AnchorHash.

The rest of this section is devoted to the first part of mapping keys to buckets, since the second indirection-based part is straightforward. We henceforth refer to adding/removing a resource as adding/removing a bucket.

**Goals.** We start by formally defining our goals. We seek a consistent hash algorithm that maps keys to buckets and satisfies the following joint objectives of *minimal disruption* and *balance*:

**Definition 1 (Minimal disruption).** *A hash algorithm achieves* minimal disruption *iff*
(i) *Upon the addition of a bucket $b \in \mathcal{A}\backslash\mathcal{W}$ to $\mathcal{W}$, keys either maintain their mapping or are remapped to $b$.*
(ii) *Upon the removal of a bucket $b \in \mathcal{W}$, keys that were not mapped to $b$ keep their mapping, and keys that were mapped to $b$ are remapped to members of $\mathcal{W}\backslash\{b\}$.*

*Example* Consider a hash algorithm $\text{HASH}(\mathcal{W}, k) = \mathcal{W}[h]$, where

$$h = \begin{cases} h_1(k) \equiv \mathit{hash}(k) \mod |\mathcal{A}| & \text{if } h_1(k) < |\mathcal{W}|, \\ h_2(k) \equiv \mathit{hash'}(k) \mod |\mathcal{W}| & \text{otherwise,} \end{cases}$$

and $\mathcal{W} \subset \mathcal{A}$. Consider the returned hash upon removal of the last bucket $b$ from $\mathcal{W}$. If $\text{HASH}(\mathcal{W}, k) = \mathcal{W}[h_1(k)]$ then, since $h_1(k)$ does not depend on $\mathcal{W}$, $\text{HASH}(\mathcal{W} \setminus \{b\}, k) = \text{HASH}(\mathcal{W}, k)$, unless $\mathcal{W}[h] = b$. Minimal disruption holds for this subset of keys. However, if $\text{HASH}(\mathcal{W}, k) = \mathcal{W}[h_2(k)]$ then most keys would get a different result, including keys that were not previously mapped to $b$. Minimal disruption does not hold in this case.

**Definition 2 (Balance).** *Let $k \in \mathcal{U}$ be a key. A hash algorithm achieves* balance *iff $k$ has an equal probability of being mapped to each bucket in $\mathcal{W}$.*

**Definition 3 (Consistency).** *We define a hash algorithm as* consistent *iff it achieves both* minimal disruption *and* balance.

AnchorHash uniformly hashes keys to bucket sets using hash functions. Accordingly, for our theoretical exposition, we make the following standard assumption (*e.g.,* [20], [21]).

**Uniform hashing assumption.** For every subset $\mathcal{B} \subseteq \mathcal{A}$, let $H_{\mathcal{B}} : \mathcal{U} \to \mathcal{B}$ be a hash function which maps keys to $\mathcal{B}$ such that: (1) $\forall k \in \mathcal{U}$ we have that $H_{\mathcal{B}}(k)$ is a uniform random variable on $\mathcal{B}$, and (2) for any sequence of distinct subsets $\mathcal{B}_1, \mathcal{B}_2, \ldots$ the random variables $H_{\mathcal{B}_1}(k), H_{\mathcal{B}_2}(k), \ldots$ are independent. In practice, this can be approximated by introducing a random *seed* into a hash function, *e.g.,* $H_{\mathcal{B},Seed}$.

## III. AnchorHash

### A. AnchorHash principles

We now explain how AnchorHash maps keys to buckets. We start with an initial working set, and then discuss how buckets are removed and added.

**Initial mapping.** Suppose we begin with a working set $\mathcal{W}$. We use the hash function $H_{\mathcal{W}}$ to map keys to $\mathcal{W}$. By the uniform hashing assumption, for any key $k$, each member of $\mathcal{W}$ has an equal probability to be chosen, thus achieving balance (Def. 2).

**Bucket removal.** Now, suppose that we want to remove a bucket $b \in \mathcal{W}$. If we use the new hash function $H_{\mathcal{W}\backslash b}$ to map keys to buckets, keys that were mapped to members of $\mathcal{W} \setminus b$ by $H_{\mathcal{W}}$ might be remapped, and the minimal disruption property will not hold.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $\mathcal{W}$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\mathcal{W}_b$ | | | | | | | |

(a) Initial working set $\mathcal{W}$=$\{0,1,2,3,4,5,6\}$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $\mathcal{W}$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × |
| $\mathcal{W}_b$ | | | | | | | $\{0,1,2,3,4,5\}$ |

(b) Removing bucket 6 with $\mathcal{W}_6$=$\{0,1,2,3,4,5\}$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $\mathcal{W}$ | ✓ | ✓ | ✓ | ✓ | ✓ | × | × |
| $\mathcal{W}_b$ | | | | | | $\{0,1,2,3,4\}$ | $\{0,1,2,3,4,5\}$ |

(c) Removing bucket 5 with $\mathcal{W}_5$=$\{0,1,2,3,4\}$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $\mathcal{W}$ | ✓ | × | ✓ | ✓ | ✓ | × | × |
| $\mathcal{W}_b$ | | $\{0,2,3,4\}$ | | | | $\{0,1,2,3,4\}$ | $\{0,1,2,3,4,5\}$ |

(d) Removing bucket 1 with $\mathcal{W}_1$=$\{0,2,3,4\}$.

Fig. 2: Example with an initial working set $\mathcal{W} = \{0,1,2,3,4,5,6\}$ (Fig. 2(a)). Then, bucket 6, 5 and 1 are removed consecutively (in Figures 2(b), 2(c) and 2(d), respectively).

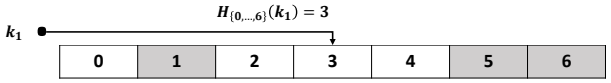
To address this issue, the key idea in AnchorHash is to keep using $H_{\mathcal{W}}(k)$ as long as $H_{\mathcal{W}}(k) \neq b$, and otherwise rehash the key to $\mathcal{W} \setminus b$ using $H_{\mathcal{W}\setminus b}(k)$. For instance, assume that the initial working set is $\mathcal{W} = \{0,\ldots,6\}$. Then we are hashing any key $k$ using $H_{\{0,\ldots,6\}}(k)$. Assume now that bucket 6 is removed. Then we continue to first hash any key $k$ using $H_{\{0,\ldots,6\}}(k)$. If it hits a bucket in $\{0,\ldots,5\}$, we are done. Otherwise, we rehash the key using $H_{\{0,\ldots,5\}}(k)$, with the result guaranteed to be a working bucket.

This approach preserves the consistency of the algorithm, as we later formally prove. First, only keys that were mapped to $b$ are remapped, thus minimal disruption is achieved. Second, by the uniform hashing assumption, keys that did not initially hit $b$ are spread uniformly over $\mathcal{W} \setminus b$, and the same is true for the keys that initially hit $b$ and are rehashed. Therefore, balance is also achieved.

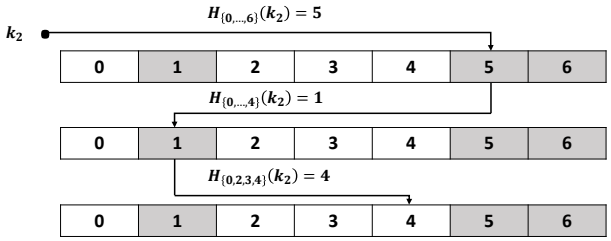
When several buckets are removed, we repeat this procedure iteratively until hitting a bucket in the working set. To simplify the notation, we denote by $\mathcal{W}_b$ the working set right after the removal of a bucket $b$.

*Example.* Fig. 2 illustrates this procedure with an initial working set $\mathcal{W}$=$\{0,1,2,3,4,5,6\}$ and buckets 6, 5 and 1 removed consecutively. Fig. 3(a) shows a simple example of a key that is immediately hashed to a bucket in the working set. Fig. 3(b) shows a more complex example in which the key is repeatedly hashed to decreasing subsets until reaching a bucket in the working set.

**Bucket addition.** Suppose that the last bucket that was removed was $b$, and the current working set is $\mathcal{W}$ (*i.e.,* $\mathcal{W}_b$=$\mathcal{W}$). Recall that AnchorHash may add any bucket not in $\mathcal{W}$ by virtue of the indirection. If we need to add a new bucket, we choose to add back bucket $b$. More generally, upon bucket addition, AnchorHash *always* adds the last removed bucket. We show in Sec. V that this allows for an extremely efficient implementation. This is because by our iterative construction, adding the last removed bucket $b$ simply brings us back to the state just before $b$'s removal. Specifically, upon the addition of $b$: (1) the only remapped keys are the ones remapped to $b$ (these are the same keys that hit $b$ and were rehashed after $b$ was previously removed), and *minimal disruption* holds; and (2) since *balance* was achieved before $b$ was removed, it is also achieved after it is added back. We prove these claims formally in Sec. IV.




(a) $k_1$ is immediately hashed to bucket $3 \in \mathcal{W}$.




(b) $k_2$ is initially hashed to bucket 5. Then, since $5 \notin \mathcal{W}$, following Fig. 2(d), $k_2$ is rehashed into the set $\{0,\ldots 4\}$. Assume that it is rehashed to $1 \in \mathcal{W}_5$. Since $1 \notin \mathcal{W}$ as well, $k_2$ is rehashed again to $4 \in \mathcal{W}_1$. Since $4 \in \mathcal{W}$, the process terminates.

Fig. 3: Example of possible key lookups in the state presented in Fig. 2(d).

*Example.* Consider Fig. 2(d). If we add the last removed bucket 1, we simply return to the state illustrated in Fig. 2(c). At this point, if we add the last removed bucket 5, we simply return to the state illustrated in Fig. 2(b), and so on.

We maintain a LIFO queue (*i.e.,* stack) for the removed buckets, denoted by $\mathcal{R}$. For example, in the state illustrated in Fig. 2(d), $\mathcal{R}$=$\{6 \leftarrow 5 \leftarrow 1\}$.

**Anchor.** By construction, $|\mathcal{A}|$ is an upper bound on the number of buckets that we allow. Therefore, in practice, we simply set the value of $|\mathcal{A}|$ to a larger value than may be needed (*e.g.,* $2\times$ the initial system size) and insert the unused buckets (*i.e.,* members of $\mathcal{A}\setminus\mathcal{W}$) into the stack $\mathcal{R}$. Note that this initial order within $\mathcal{R}$ may be arbitrary. We later leverage this observation to optimize implementation. Since $\mathcal{A}$ serves as the starting point of the algorithm on which everything is defined, we refer to it as the *Anchor*.

*Example.* Consider again Fig. 2(a). Assume that instead of beginning our operation with $\mathcal{W}$=$\{0,1,2,3,4,5,6\}$, we would like to start our system with only $\mathcal{W}$=$\{0,1,2,3,4\}$, but want to be prepared to increase $\mathcal{W}$ to include buckets 5 and 6 if needed. Then, we simply start our system with $\mathcal{A}$=$\{0,1,2,3,4,5,6\}$, and initially set $\mathcal{W}$=$\{0,1,2,3,4\}$ and $\mathcal{R}$=$\{6 \leftarrow 5\}$. This precise state is illustrated in Fig. 2(c).

### B. AnchorHash algorithm

The pseudo-code for AnchorHash is given in Alg. 1.

**Initialization.** INITANCHOR($\mathcal{A},\mathcal{W}$) receives as an input the Anchor $\mathcal{A}$ and the initial working set of buckets $\mathcal{W}$. We fill the stack $\mathcal{R}$ with the initially unused buckets. For each such unused bucket $b$, we remember $\mathcal{W}_b$, *i.e.,* the working set just after its removal.

**Algorithm 1** — AnchorHash

```
1: function InitAnchor(A, W)
2:     R ← ∅
3:     for b ∈ A\W do
4:         R.push(b)
5:         W_b ← A\R
6:
7: function GetBucket(k)
8:     b ← H_A(k)
9:     while b ∉ W do
10:        b ← H_{W_b}(k)
11:    return b
12:
13: function AddBucket( )
14:    b ← R.pop()
15:    delete W_b
16:    W ← W ∪ {b}
17:    return b
18:
19: function RemoveBucket(b)
20:    W ← W\ {b}
21:    W_b ← W
22:    R.push(b)
```

**Algorithm 2** — AnchorHash Wrapper

```
1: function InitWrapper(A, S)
2:     M ← ∅, W ← ∅
3:     for i ∈ (0, 1, ..., |S| − 1) do
4:         M ← M ∪ {(A[i], S[i])}
5:         W ← W ∪ {A[i]}
6:     InitAnchor(A, W)
7:
8: function GetResource(k)          ▷ Compute resource for key k
9:     b ← GetBucket(k)
10:    ξ ← M(b)
11:    return ξ
12:
13: function AddResource(ξ)
14:    b ← AddBucket( )
15:    M ← M ∪ {(b, ξ)}
16:
17: function RemoveResource(ξ)
18:    b ← M^{-1}(ξ)
19:    M ← M\ {(b, ξ)}
20:    RemoveBucket(b)
```

**GetBucket.** GetBucket($k$) receives a key $k \in \mathcal{U}$ as an input and returns a working bucket $b \in \mathcal{W}$ as an output. Initially, we hash the key uniformly over the Anchor $\mathcal{A}$; then, if the calculated bucket $b$ is not a member of $\mathcal{W}$, the key is rehashed into $\mathcal{W}_b$. This process continues until hitting a working bucket. We analyze the computational complexity of this procedure in Sec. IV and present empirical evaluation results in Sec. VI.

**AddBucket.** As mentioned, when adding a bucket, we add the the last removed bucket. Accordingly, AddBucket() has no input and simply returns the added bucket. It pops the last removed bucket $b$ from $\mathcal{R}$, deletes the no-longer-needed $\mathcal{W}_b$, adds $b$ to $\mathcal{W}$ and returns $b$.

**RemoveBucket.** RemoveBucket($b$) receives as an input the bucket we want to remove, and has no return value. We simply remove $b$ from $\mathcal{W}$, record the working set just after $b$'s removal $\mathcal{W}_b$ and push $b$ to the top of $\mathcal{R}$.

**Indirection.** For completeness, Alg. 2 presents the full key-to-resource mapping based on indirection (as presented in Fig. 1). It complements the key-to-bucket mapping of Alg. 1 with a standard bucket-to-resource bijection function $M$. For simplicity, we represent this bijection using a set of coupled pairs $(b, \xi) \in M$ such that $M(b) = \xi$ and $M^{-1}(\xi) = b$. Note that such indirection is trivially implemented using a standard map with $O(1)$ operations on average for each bucket or resource lookup.

## IV. AnchorHash properties

In this section we first prove that AnchorHash is consistent (*i.e.,* provides minimal disruption and balance), and then analyze its complexity.

**Theorem 1 (Minimal disruption).** *AnchorHash guarantees minimal disruption.*

*Proof.* **(i).** Assume a newly added bucket $b$. Consider function GetBucket($k$). Then, before $b$'s addition, each $k \in \mathcal{U}$ either encountered bucket $b$ before terminating or not. After the addition of $b$, keys that did not encounter $b$ are clearly not affected. Those that did now terminate at $b$.

**(ii).** Assume a newly removed bucket $b$. Consider again function GetBucket($k$). Before $b$'s removal, each $k \in \mathcal{U}$ either terminated at bucket $b$ or did not encounter it at all. After the removal of $b$, keys that did not encounter $b$ are clearly not affected. Those that did, now terminate at $H_{\mathcal{W}_b}(k)$. □

**Theorem 2 (Balance).** *AnchorHash achieves balance.*

*Proof.* We prove that given a possible sequence of operations, where an operation can be either a bucket removal or a bucket addition, balance holds in the initial state and after every operation. By the definition of GetBucket($k$) in Algorithm 1, its returned bucket for a specific key depends only on the stack $\mathcal{R}$ of removed buckets. Denote by $\mathcal{R}_j$ the stack after operation $j$ and denote $\mathcal{R}_0 = \emptyset$. We refer to the initial state $\mathcal{R}_0$ as the state after operation number 0. Our proof is by induction on the number of operations.

**Basis: initial state.** In this case, since $\mathcal{R}_0 = \emptyset$ we have $\mathcal{W}$=$\mathcal{A}$. According to the uniform hashing assumption, for every $k$, $H_{\mathcal{A}}(k)$ is a uniform random variable over $\mathcal{W}$.

**Induction hypothesis.** Assume that balance holds after every operation $j \leq i$; namely for every stack $\mathcal{R}_j$ such that $j \leq i$.

**Inductive step.** We now prove that balance holds after operation $i + 1$, which can either be a bucket removal (unless $|\mathcal{W}| = 1$) or a bucket addition (unless $\mathcal{R}_i = \emptyset$).

*Bucket removal.* Consider a newly removed bucket $b$. After $b$'s removal, according to Theorem 1 (minimal disruption), only keys that were mapped to $b$ are remapped. These are remapped using $H_{\mathcal{W}_b}$, which, by the uniform hashing assumption, assigns each of them with an equal probability to the members of $\mathcal{W}_b$, independently from their previous mappings.

*Bucket addition.* By the definition of AddBucket() in Algorithm 1, $\mathcal{R}_{i+1}$ is obtained by popping the bucket at the top of the stack $\mathcal{R}_i$; therefore $\mathcal{R}_{i+1} = \mathcal{R}_j$, for some $j < i$ (for example, if the previous operation was a bucket removal, then $\mathcal{R}_{i+1} = \mathcal{R}_{i-1}$, as an addition is an "undo" of removal). Thus, by the induction hypothesis, balance holds after operation $i + 1$. □

We now turn to providing a strong theoretical guarantee on the run-time complexity of GETBUCKET($k$), which explains why AnchorHash is able to process keys at a high rate at scale.

**Theorem 3 (Computational complexity).** *Fix $\mathcal{A}$, $\mathcal{W}$ and $\mathcal{R}$ such that $|\mathcal{A}|=a$ and $|\mathcal{W}|=w$. For a key $k$, denote by $\tau$ the number of hash operations performed by* GETBUCKET($k$). *Then:*

*1) The average of $\tau$ is upper-bounded by $1+\ln(\frac{a}{w})$.*
*2) The standard deviation of $\tau$ is upper-bounded by $\sqrt{\ln(\frac{a}{w})}$.*

*Proof.* Once GETBUCKET($k$) is invoked, we repeatedly hash $k$ into decreasing subsets of $\mathcal{A}$ until hitting a working bucket. The number of hash operations is 1 plus the number of iterations in the while loop, which is entered only if $b = H_{\mathcal{A}}(k) \notin \mathcal{W}$, i.e., bucket $b$ was removed. Consider a fixed sequence of removals $\mathcal{R} = \{r_{a-w-1} \leftarrow \ldots \leftarrow r_0\}$; namely, $r_{a-w-1}$ is the first removed bucket and $r_0$ is the last removed bucket (i.e., $\mathcal{W} = \mathcal{W}_{r_0}$).

Let $\tau_i$ denote the number of remaining iterations *after* the loop is entered with $b = r_i$. Let $b_i$ denote $H_{\mathcal{W}_{r_i}}(k)$; if $b_i \in \mathcal{W}$ then the loop terminates and $\tau_i = 1$. If $b_i \notin \mathcal{W}$, then $b_i = r_j$ for some $j < i$ ($r_i$ and all earlier removals are not in $\mathcal{W}_{r_i}$). Then, by the uniform hashing assumption, $\tau_i$ has the same distribution as $1 + \tau_j$.

For $i = 0$, since $\mathcal{W} = \mathcal{W}_{r_0}$, we have $b_0 \in \mathcal{W}$, thus $\tau_0 = 1$. With $\tau_0, \ldots, \tau_{a-w-1}$ at hand, for ease of exposition, we also define $\tau_{a-w} = \tau$ and $b_{a-w} = b$. We use these notations and observations to derive a recursive formula and find a closed-form expression for the moment generating function (MGF) of $\tau$. We then use it to find the first two moments of $\tau$.

For $0 \le i \le a - w$, define

$$\phi_i(s) = \mathbb{E}[e^{s\tau_i}]. \tag{1}$$

Then, by the law of total expectation, for $i > 0$,

$$\begin{aligned}\phi_i(s) &= \mathbb{P}(b_i{\in}\mathcal{W})\mathbb{E}[e^{s\tau_i}|b_i{\in}\mathcal{W}] + \sum_{j=0}^{i-1}\mathbb{P}(b_i{=}r_j)\mathbb{E}[e^{s\tau_i}|b_i{=}r_j] \\ &= \frac{w}{w+i}\mathbb{E}[e^{s\tau_i}|b_i{\in}\mathcal{W}] + \frac{1}{w+i}\sum_{j=0}^{i-1}\mathbb{E}[e^{s\tau_i}|b_i{=}r_j].\end{aligned} \tag{2}$$

First, if $b_i \in \mathcal{W}$, the loop terminates after a single hash calculation, *i.e.,* $\tau_i = 1$. Thus

$$\mathbb{E}[e^{s\tau_i}|b_i{\in}\mathcal{W}] = e^s. \tag{3}$$

Second, recall that the distribution of $\tau_i$ conditioned on $b_i = r_j$ follows the same distribution as $1 + \tau_j$. Therefore

$$\begin{aligned}\mathbb{E}[e^{s\tau_i} \mid b_i = r_j] &= \mathbb{E}[e^{s(1+\tau_j)}] \\ &= e^s\mathbb{E}[e^{s\tau_j}] = e^s\phi_j(s),\end{aligned} \tag{4}$$

Substituting (3) and (4) in (2) yields

$$\phi_i(s) = \frac{w\cdot e^s}{w+i} + \frac{e^s}{w+i}\cdot\sum_{j=0}^{i-1}\phi_j(s). \tag{5}$$

Now that we have a recursive formula for $\phi_i(s)$, we are able to calculate its closed-form expression. For $i > 0$, rearranging (5) yields

$$\begin{aligned}\frac{w+i}{e^s}\cdot\phi_i(s) &= w + \sum_{j=0}^{i-1}\phi_j(s) \\ &= w + \sum_{j=0}^{i-2}\phi_j(s) + \phi_{i-1}(s) \\ &= \frac{w+i-1}{e^s}\cdot\phi_{i-1}(s) + \phi_{i-1}(s) \\ &= \frac{w+i-1+e^s}{e^s}\cdot\phi_{i-1}(s),\end{aligned} \tag{6}$$

where the third equality is derived by using the first equality but with $i-1$ instead of $i$. Therefore,

$$\phi_i(s) = \phi_{i-1}(s)\cdot\frac{w+i-1+e^s}{w+i}. \tag{7}$$

Now, using (7) and the stopping condition $\phi_0(s) = e^s$, we obtain for $i > 0$,

$$\phi_i(s) = e^s\prod_{j=1}^{i}\Big(\frac{w+j-1+e^s}{w+j}\Big), \tag{8}$$

Taking the logarithm and then differentiating with respect to $s$ yields

$$\frac{\phi_i'(s)}{\phi_i(s)} = 1 + \sum_{j=1}^{i}\Big(\frac{e^s}{w+j-1+e^s}\Big). \tag{9}$$

By (1), $\phi_i(0) = 1$. Hence, substituting $s = 0$ in (9) yields

$$\mathbb{E}[\tau_i] = \phi_i'(0) = 1 + \sum_{j=1}^{i}\frac{1}{w+j}, \tag{10}$$

and therefore

$$\begin{aligned}\mathbb{E}[\tau] = \mathbb{E}[\tau_{a-w}] = \phi_{a-w}'(0) &= 1 + \sum_{j=1}^{a-w}\frac{1}{w+j} \\ &\le 1 + \int_w^a\frac{1}{x}dx = 1 + \ln\Big(\frac{a}{w}\Big).\end{aligned}$$

Now, to obtain the bound on the standard deviation, we take the derivative with respect to $s$ in (9) and obtain

$$\frac{\phi_i''(s)\phi_i(s) - (\phi_i'(s))^2}{(\phi_i(s))^2} = \sum_{j=1}^{i}\Big(\frac{e^s(w+j-1+e^s) - e^{2s}}{(w+j-1+e^s)^2}\Big).$$

Setting $i = a - w$, $s = 0$ and using $\phi_{a-w}(0) = 1$ yields

$$\begin{aligned}\mathrm{Var}(\tau) = \mathrm{Var}(\tau_{a-w}) &= \phi_{a-w}''(0) - (\phi_{a-w}'(0))^2 \\ &= \sum_{j=1}^{a-w}\frac{w+j-1}{(w+j)^2} \le \sum_{j=1}^{a-w}\frac{1}{w+j} \le \ln(\frac{a}{w}).\end{aligned}$$

Thus the standard deviation is upper bounded by $\sqrt{\ln(\frac{a}{w})}$. Note that the bounds do not depend on the removal sequence we fixed. This concludes the proof. □

## V. AnchorHash implementation

Equation (8), from which the result of Theorem 3 is derived, implies that the distribution of the number of iterations (and by that also the number of hash operations) of GetBucket($k$) is *independent* of the implementation. However, the implementation does determine the amount of used memory and how many calculations and memory accesses are performed during each iteration of GetBucket($k$).

Specifically, each such iteration requires choosing a bucket uniformly at random from a known set (*i.e.,* $b \leftarrow H_{\mathcal{W}_b}(k)$) and checking if this bucket is working (*i.e.,* $b \notin \mathcal{W}$). The most challenging part is finding an efficient way to hold these different sets (*i.e.,* $\{\mathcal{W}_b \,|\, b \in \mathcal{R}\}$).

In the following, we first describe in detail the different components of AnchorHash implementation. Then, we present three distinct implementations of holding $\{\mathcal{W}_b \,|\, b \in \mathcal{R}\}$ that achieve different memory-computation complexity trade-offs which are summarized in Table II.

**Anchor representation.** We use an integer array $A$ of size $a$ to represent the Anchor. Each bucket $b \in \{0, 1, \ldots, a-1\}$ is represented by $A[b]$ that either equals 0 if $b$ is a working bucket (*i.e.,* $A[b] = 0$ if $b \in \mathcal{W}$), or else equals the size of the working set just after its removal (*i.e.,* $A[b] = |\mathcal{W}_b|$ if $b \in \mathcal{R}$).

*Example.* Considering again the example in Fig. 2(d), we have

| $b$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $A[b]$: | 0 | 4 | 0 | 0 | 0 | 5 | 6 |

By examining this array we can determine that buckets 0, 2, 3, and 4 are working, and buckets $1, 5, 6$ are removed, with $|\mathcal{W}_1| = 4$, $|\mathcal{W}_5| = 5$ and $|\mathcal{W}_6| = 6$.

**Hashing.** Denote $h_b(k) \equiv \text{hash}(b, k) \bmod A[b]$. We are using $b$ as "salt" in the hash function to make sure $\{h_b()\}_{b=0,1,\ldots}$ are independent (as assumed by the uniform hashing assumption). To implement hash$(b, k)$ efficiently, recent software-based solutions such as xxHash [22] and hardware-supported hashing such as crc32 [23] can be used.

**Removed buckets.** AnchorHash saves the removed buckets in a LIFO order for possible future bucket additions. Accordingly, we use an efficient implementation of a stack data structure $R$ to hold the removed buckets.

*Example.* In the example of Fig. 2(d), $R$ looks like:

| 6 | 5 | 1 | | | | ⇆ |
|---|---|---|---|---|---|---|

**Decreasing subsets.** For each removed bucket $b$, we need an efficient way of representing $\mathcal{W}_b$ and calculating $H_{\mathcal{W}_b}(k)$. For clarity, we tackle this challenge in stages: we begin with a *naive* implementation, which we successively improve to implementations with a *partial* then *minimal* memory usage.

### A. Naive implementation

A naive approach to representing $\{\mathcal{W}_b \,|\, b \in \mathcal{R}\}$ is using a key-value store, KV, that holds the pairs $\{(b, \mathcal{W}_b) \,|\, b \in \mathcal{R}\}$, where the *key* is a removed bucket $b$ and the *value* is $\mathcal{W}_b$, stored in KV$[b]$ as an array. This way, implementing $H_{\mathcal{W}_b}(k)$ simply translates to $H_{\mathcal{W}_b}(k) \equiv \text{KV}[b][h_b(k)]$.

*Example.* In the example in Fig. 2(d), KV$[1]$ looks like:

$\mathcal{W}_1 = \{0, 2, 3, 4\}$, KV$[1]$ :

| $h_1(k)$: | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| | 0 | 2 | 3 | 4 |

Unfortunately, albeit simple, this approach is not scalable, as it requires to maintain an array of size $|\mathcal{W}_b|$ for each removed bucket $b$, incurring a large memory footprint of $\Theta(|\mathcal{A}| + |\mathcal{A}||\mathcal{R}|)$. That is, arrays of sizes $|\mathcal{A}|-1, |\mathcal{A}|-2, \ldots, |\mathcal{A}|-|\mathcal{R}|$ are maintained for the members of $|\mathcal{R}|$. Since we also use an array of size $|\mathcal{A}|$ for the Anchor representation, the total memory footprint is given by

$$\begin{aligned} |\mathcal{A}| + \sum_{i=1}^{|\mathcal{R}|} \left(|\mathcal{A}| - i\right) &= |\mathcal{A}| + |\mathcal{A}||\mathcal{R}| - 0.5|\mathcal{R}|(|\mathcal{R}| + 1) \\ &\geq |\mathcal{A}| + |\mathcal{A}||\mathcal{R}| - 0.5|\mathcal{A}|(|\mathcal{R}| + 1) \\ &= 0.5(|\mathcal{A}| + |\mathcal{A}||\mathcal{R}|), \end{aligned}$$

and since

$$\begin{aligned} |\mathcal{A}| + \sum_{i=1}^{|\mathcal{R}|} \left(|\mathcal{A}| - i\right) &= |\mathcal{A}| + |\mathcal{A}||\mathcal{R}| - 0.5|\mathcal{R}|(|\mathcal{R}| + 1) \\ &\leq |\mathcal{A}| + |\mathcal{A}||\mathcal{R}|, \end{aligned}$$

the total memory footprint of the naive implementation is $\Theta(|\mathcal{A}| + |\mathcal{A}||\mathcal{R}|)$. Upon a bucket addition, the complexity accounts for $\Theta(|\mathcal{W}|)$ (*i.e.,* adding $\mathcal{W}_b$ to KV).

### B. Reduced-memory implementation

**Non-fixed points.** Consider again the naive implementation. Recall that all of the theoretical properties of AnchorHash are *independent* of the exact bucket order within the sets $\{\mathcal{W}_b \,|\, b \in \mathcal{R}\}$. Also, for any two *consecutively* removed buckets $b_1$ and $b_2$, the sets $\mathcal{W}_{b_1}$ and $\mathcal{W}_{b_2}$ only differ by a single bucket.

We want to leverage these properties to reduce the memory footprint of AnchorHash and accelerate its performance. Accordingly, we seek to *minimize the number of non-fixed point entries* in the members of $\{\mathcal{W}_b \,|\, b \in \mathcal{R}\}$, which we define as entries that respect $\mathcal{W}_b[h] \neq h$. This way we do not need to remember the full arrays, but only the *difference* between the initial order of buckets and each member of $\{\mathcal{W}_b \,|\, b \in \mathcal{R}\}$, *i.e.,* the *non-fixed points*.

*Example.* Recall the example in Fig. 2(d). In this example, the naive approach holds three arrays: KV$[6]$, KV$[5]$, and KV$[1]$. Our goal is to minimize the number of non-fixed point entries between the initial order of buckets $\{0, 1, 2, \ldots, 6\}$ and the order of buckets in the members of $\{\mathcal{W}_b \,|\, b \in \{1, 5, 6\}\}$. For example, to obtain the desired order for $\mathcal{W}_1 = \{0, 2, 3, 4\}$ and minimize the difference with $\boxed{0}\boxed{1}\boxed{2}\boxed{3}$, we simply use $\boxed{0}\boxed{\mathbf{4}}\boxed{2}\boxed{3}$, *i.e.,* take bucket 4 which is the last element in KV$[5]$, and put it instead of the removed bucket 1. This yields

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| $\mathcal{A} = \{0, 1, 2, 3, 4, 5, 6\}$, | Init : | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| $\mathcal{W}_6 = \{0, 1, 2, 3, 4, 5\}$, | KV$[6]$ : | 0 | 1 | 2 | 3 | 4 | 5 | |
| $\mathcal{W}_5 = \{0, 1, 2, 3, 4\}$, | KV$[5]$ : | 0 | 1 | 2 | 3 | 4 | | |
| $\mathcal{W}_1 = \{0, 2, 3, 4\}$, | KV$[1]$ : | 0 | **4** | 2 | 3 | | | |

(11)

Examining (11) reveals that instead of remembering all three arrays, we can just remember that KV[1][1]=4 (recall that $A$ provides the length of each array). Namely, all other elements are simply fixed points. Each time we calculate $H_{\mathcal{W}_b}(k)$, it equals $h_b(k)$ without the need to access any data structure. The only exception is when an entering key hits bucket 1 and then hashes to 1 again (*i.e.,* calculating $H_{\mathcal{W}_1}(k)$ yields $h_1(k) = 1$). For this specific case, we need to remember that we hit bucket KV[1][1]=4 instead of 1.

Now, assume that in this state bucket 0 is removed. Similarly, the desired ordering for $\mathcal{W}_0$ is obtained by taking the last element in KV[1], which is bucket 3, and putting it instead of the removed bucket 0. This yields

$$\mathcal{W}_0 = \{2,3,4\}, \qquad \begin{array}{l} h_0(k): \\ \text{KV}[0]: \end{array} \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline \mathbf{3} & \mathbf{4} & 2 \\ \hline \end{array} \tag{12}$$

Again, we only need to store KV[0][0]←3 and KV[0][1]←4, since working bucket 2 is a fixed point and its location is identical to its location in the initial ordering. To summarize, in this example we only need to remember 3 elements (the bold numbers in (11) and (12)) instead of the original 6+5+4+3 = 18.

**Individual KV entries.** To leverage this solution with reduced memory requirements, we stop organizing the key-value store using arrays. Instead of keeping an entry KV[$b$][$h$], we keep an entry KV[$(b,h)$] where the pair $(b,h)$ is the key. This can be efficiently implemented by simply concatenating $b$ and $h$ to form a single key. For example, in (12), instead of using KV[0][1]=4 with an array, we use KV[$(0,1)$]=4. Thus,

$$H_{\mathcal{W}_b}(k) = \begin{cases} \text{KV}[(b, h_b(k))] & \text{if the entry exists} \\ h_b(k) & \text{otherwise} \end{cases}$$

To efficiently determine the desired order within $\mathcal{W}_b$ for a newly removed bucket $b$ and the exact elements that we need to store, we maintain two additional arrays: (1) $W$, which always contains the *current* set of working buckets in their desired order, and (2) $L$, which stores for each bucket its *most recent* location in $W$. Both arrays are initialized identically: $W[b] = L[b] = b \quad \forall b \in \{0,1,\ldots,a-1\}$. For instance, after bucket 1 is removed (*i.e.,* last array in (11)), $W$ and $L$ obtain the following form:

$$\begin{array}{l} b: \\ W[b]: \end{array} \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 0 & 4 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array} \quad \begin{array}{l} b: \\ L[b]: \end{array} \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 0 & 1 & 2 & 3 & 1 & 5 & 6 \\ \hline \end{array}.$$

That is, bucket 4 replaced bucket 1 in $W$ and the most recent location of bucket 4 updated to index 1. Note that the removals of buckets 5 and 6 did not require any updates in both $W$ and $L$. With this example at hand, we now detail the update rules for $W$ and $L$ upon bucket removals and additions.

**Removal.** Assume a newly removed bucket $b$ and let $N = |\mathcal{W}_b|$. Then in $W$, $b$ is replaced by the last positioned working bucket (*i.e.,* $W[N]$), and its most recent location (*i.e.,* $L[b]$) is correspondingly updated in $L$. This yields

$$W[L[b]] \leftarrow W[N], \;\; L[W[N]] \leftarrow L[b].$$

Now, we use the updated array $W$ to determine which entries to store in KV: for all $h \in \{0,1,\ldots,|\mathcal{W}_b|-1\}$ such that $W[h] \neq h$, we store KV[$(b,h)$] ← $W[h]$.

**Addition.** Upon bucket addition, we need to *restore* the state prior to the last removal. To do so, we delete the corresponding entries in KV by the same rule we used to remember them. Then, we restore $W$ and $L$ to their previous state using:

$$L[W[N]] \leftarrow N, \;\; W[L[b]] \leftarrow b.$$

For example, given the state in (11), if we now add back bucket 1 then we simply restore $W$ and $L$ to their initial state, since using the rules yields $L[4] \leftarrow 4$ and $W[1] \leftarrow 1$.

**Complexity.** In the worst case, each consecutive removed bucket may require one additional entry in addition to the entries required by the previously removed bucket. Accordingly, this method for resolving $H_{\mathcal{W}_b}(k)$ results in a memory footprint of $1+2+\ldots+|\mathcal{R}| = 0.5|\mathcal{R}|(|\mathcal{R}|+1)$, together with three arrays of size (at most) $|\mathcal{A}|$ to represent $A$, $W$ and $L$. The total memory footprint is therefore $\Theta(|\mathcal{A}| + |\mathcal{R}|^2)$. Updating KV upon a bucket addition or removal incurs a complexity of $O(|\mathcal{W}|)$.

### C. Minimal-memory implementation

While the previous implementation may be sufficient for systems with a small $|\mathcal{R}|$ value, we present our final implementation of AnchorHash that results in a remarkably low-memory footprint, negligible response time to changes and high key lookup rate. Specifically, we show how to efficiently calculate KV[$(b,h)$] for all $(b,h)$ pairs, using a single array that replaces the key-value store functionality.

**Successors.** To do so, for each removed bucket $b$, we are only storing its *successor*, *i.e.,* the bucket that replaced it in $W$. That is, we define an array $K$, such that its entry for each removed bucket $b$ is $K[b] = \text{KV}[(b, L[b])]$. We initiate $K[b] \leftarrow b \; \forall b \in \{0,1,\ldots,a-1\}$, as initially a working bucket $b$ appears at $W[b]$ (*i.e.,* replaces itself). For instance, in the example of (11), we just remember that bucket 4 replaced bucket 1 (*i.e.,* $K[1] = 4$), and later in the example of (12), that bucket 3 replaced bucket 0 (*i.e.,* $K[0] = 3$). This yields

$$\begin{array}{l} b: \\ K[b] = \\ A[b] = \end{array} \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \mathbf{3} & \mathbf{4} & 2 & 3 & 4 & 5 & 6 \\ \hline 3 & 4 & 0 & 0 & 0 & 5 & 6 \\ \hline \end{array}. \tag{13}$$

We next show that we can exploit this information to reconstruct the individual KV entries used by the reduced-memory footprint implementation. Our key observation is that when trying to resolve KV[$(b,h)$], we are actually searching for $W[h]$ just after $b$'s removal. Therefore, we can trace through the history of $W[h]$, until we reach $\mathcal{W}_b[h]$. We start from $h$, which is the initial value of $W[h]$. When bucket $h$ was removed, $W[h]$ was updated to its successor, *i.e.,* $K[h]$. Similarly, when $K[h]$ was removed, it was updated to its successor as well, *i.e.,* $K[K[h]]$, and so on. Accordingly, we iteratively set $h \leftarrow K[h]$, until we reach the first working bucket at $W[h]$ just after $b$'s removal. We determine the stopping condition by looking at the sizes of $\mathcal{W}_b$ and $\mathcal{W}_h$: when $A[b] \geq A[h]$ we know that $K[h]$ was working when $b$ was removed, and can terminate.

**Algorithm 3** — AnchorHash Implementation

```
1: function InitAnchor(a, w)
2:     A[b] ← 0 for b = 0, 1, ..., a−1            ▷ |W_b| ← 0 for b ∈ A
3:     R ← ∅                                       ▷ Empty stack
4:     N ← w                                       ▷ Number of initially working buckets
5:     K[b] ← L[b] ← W[b] ← b for b = 0, 1, ..., a − 1
6:     for b = a−1 downto w do                     ▷ Remove initially unused buckets
7:         R.push(b)
8:         A[b] ← b
9:
10: function GetBucket(k)
11:     b ← hash(k) mod a
12:     while A[b]>0 do                             ▷ b is removed
13:         h ← h_b(k)                              ▷ h ← hash(b, k) mod A[b]
14:         while A[h] ≥ A[b] do                    ▷ W_b[h] ≠ h, b removed prior to h
15:             h ← K[h]                            ▷ search for W_b[h]
16:         b ← h                                   ▷ b ← H_{W_b}(k)
17:     return b
18:
19: function AddBucket( )
20:     b ← R.pop()
21:     A[b] ← 0                                    ▷ W ← W ∪ {b}, delete W_b
22:     L[W[N]] ← N
23:     W[L[b]] ← K[b] ← b
24:     N ← N + 1
25:     return b
26:
27: function RemoveBucket(b)
28:     R.push(b)
29:     N ← N − 1
30:     A[b] ← N                                    ▷ W_b ← W\b, A[b] ← |W_b|
31:     W[L[b]] ← K[b] ← W[N]
32:     L[W[N]] ← L[b]
```

*Example.* Consider the example in (13).

If in this state we further remove bucket 4, we obtain $\mathcal{W}_4 = \{2, 3\}$ and

$$\begin{array}{rl} b: & \begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \end{array} \\ K[b] = & \begin{array}{|c|c|c|c|c|c|c|}\hline \mathbf{3} & \mathbf{4} & 2 & 3 & \mathbf{2} & 5 & 6 \\ \hline \end{array}, \\ A[b] = & \begin{array}{|c|c|c|c|c|c|c|}\hline 3 & 4 & 0 & 0 & 2 & 5 & 6 \\ \hline \end{array}. \end{array} \tag{14}$$

As an example, we show how the arrays $K$ and $A$ can be used to calculate KV($[5, 1]$), KV($[1, 1]$), and KV($[4, 1]$). Recall that $A[b]$ holds $|\mathcal{W}_b|$ and KV($[b, h]$) holds $H_{\mathcal{W}_b}(k)$, where $h = h_b(k)$. Also, if $A[h] < A[b]$, then $h \in \mathcal{W}_b$ ($h$ was not removed before $b$) and KV($[b, h]$) $= h$.

We use $A$ to check if KV($[b, h]$) $= h$. In our case, KV($[5, 1]$) $= 1$, but KV($[4, 1]$) $\neq 1$ and KV($[1, 1]$) $\neq 1$, since $A[4] < A[1] < A[5]$. In other words, we know from $A$ that $1 \in \mathcal{W}_5$, but $1 \notin \mathcal{W}_4$ (and, obviously, $1 \notin \mathcal{W}_1$). We use $K$ to calculate KV($[1, 1]$); since $K[1] = 4$ and $4 \in \mathcal{W}_1$ ($A[4] < A[1]$), we conclude that KV($[1, 1]$) $= 4$. Similarly, we use $K$ twice to calculate KV($[4, 1]$); since $K[1] = 4$ and $4 \notin \mathcal{W}_4$, we examine $K[K[1]] = 2$. Since $2 \in \mathcal{W}_4$ ($A[2] < A[4]$), we conclude that KV($[4, 1]$) $= 2$.

All three of the above calculations may be needed to find a working bucket by GetBucket($k$). For example, consider a key $k$ for which $H_{\mathcal{A}}(k) = 5$ and $h_5(k) = h_1(k) = h_4(k) = 1$. First we examine bucket 5 and since $5 \notin \mathcal{W}$ ($A[5] > 0$) we rehash to $H_{\mathcal{W}_5}(k) =$ KV($[5, h_5(k)]$) $=$ KV($[5, 1]$) $= 1$. Since $1 \notin \mathcal{W}$, we rehash to $H_{\mathcal{W}_1}(k) =$ KV($[1, h_1(k)]$) $=$ KV($[1, 1]$) $= 4$. Since $4 \notin \mathcal{W}$, we rehash yet again to $H_{\mathcal{W}_4}(k) =$ KV($[4, h_4(k)]$) $=$ KV($[4, 1]$) $= 2$. Finally, since $2 \in \mathcal{W}$ it can be returned as the bucket for key $k$.

**Complexity.** Alg. 3 provides the pseudo-code for AnchorHash's final array-based implementation. The memory footprint for this solution is $\Theta(|\mathcal{A}|)$ *independently of the system state— e.g.,* , independently of the number of removed buckets or of their identity. That is, we keep four arrays $A, L, W, K$ of size $|\mathcal{A}|$ and the stack $\mathcal{R}$. The update time upon a bucket removal or addition accounts for $O(1)$ operations and is negligible for any $\mathcal{A}$ and $\mathcal{R}$. Note that $\Theta(|\mathcal{A}|)$ is required to save resource details (*e.g.,* server IP addresses).

While we already established bounds on the number of *hash operations*, we now provide an upper bound on the average number of *memory accesses* required by a key lookup when using our final minimal-memory implementation.

**Theorem 4 (Memory accesses).** *Assume* random *removals. Let $|\mathcal{W}| = w$ and $|\mathcal{A}| = a$. Denote by $\xi$ the total number of memory accesses performed by* GetBucket($k$) *for a randomly chosen key $k$ when using the minimal-memory implementation. Then, the average of $\xi$ is $O\big(\big(1 + \ln\big(\frac{a}{w}\big)\big)^2\big)$.*

*Proof.* Denote by $\mathcal{R}$ the (random) sequence of bucket removals. We will prove the result recursively on the size of $\mathcal{R}$. Suppose $|\mathcal{R}| = r$ buckets were randomly removed, and now we randomly remove an additional bucket.

By the minimal disruption property, only the keys that were mapped to this newly removed bucket are remapped. Likewise, by the balance property, this occurs with probability $\frac{1}{a-r}$ for a randomly chosen key. The keys that are remapped require an additional access to the array $A$ and possibly the *resolution* of the bucket's identity using the array K, where the latter depends on the sequence of removals and the last index the key hits.

The length of the required resolution when hitting index $i$ is upper-bounded by the number of times the bucket associated with index $i$ was removed. Denote this quantity by $\Delta_i^r$.Since the remapped keys have an equal probability of hitting any index in $\{0, 1, \ldots, a - r - 2\}$, we obtain

$$\begin{aligned} \mathbb{E}[\xi \mid |\mathcal{R}| = r + 1] &\leq \mathbb{E}[\xi \mid |\mathcal{R}| = r] \\ &+ \cdot \frac{1}{a-r}\Big(1 + \frac{1}{a-r-1}\sum_{i=0}^{a-r-2}\mathbb{E}[\Delta_i^{r+1}]\Big), \end{aligned} \tag{15}$$

Now, we observe that since index 0 is always associated with a working bucket, $\Delta_0^{r+1}$ is stochastically larger than $\Delta_i^{r+1}$ for all $i$. Also,

$$\begin{aligned} \mathbb{E}[\Delta_0^{r+1}] &= \frac{1}{a} + \ldots + \frac{1}{a-r-1} \\ &= \sum_{k=1}^{r+1}\frac{1}{a-k} \leq \ln\Big(\frac{a}{w}\Big). \end{aligned} \tag{16}$$

Thus, using (16) in (15) yields the following recurrence,

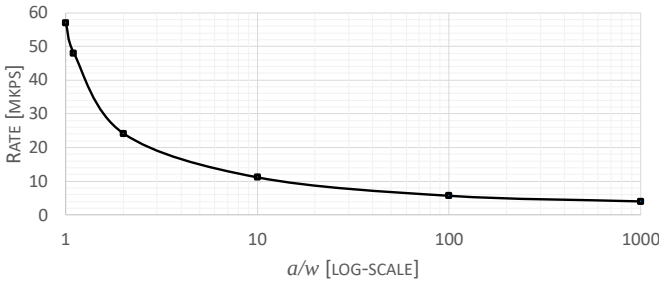
$$\begin{aligned} &\mathbb{E}[\xi \mid |\mathcal{R}| = r + 1] \leq \\ &\mathbb{E}[\xi \mid |\mathcal{R}| = r] + \frac{1}{a-r}\Big(1 + \ln\Big(\frac{a}{w}\Big)\Big), \end{aligned} \tag{17}$$

with the initial condition given by

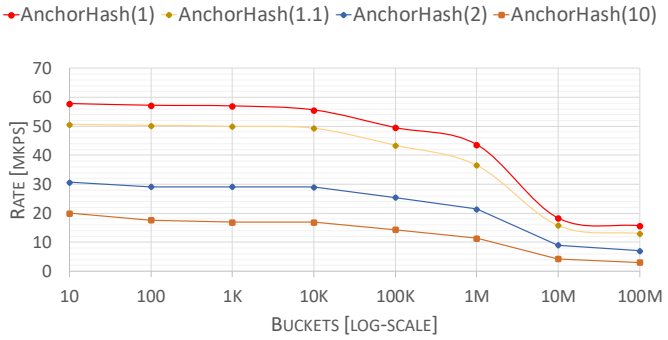
$$\mathbb{E}[\xi \mid |\mathcal{R}| = 0] = 1. \tag{18}$$

| | Hash operations | Memory accesses | Memory | Update |
|---|---|---|---|---|
| Naive | $O(1+\log(\frac{a}{w}))$ | $O(1+\log(\frac{a}{w}))$ | $\Theta(a+ar)$ | $\Theta(w)$ |
| Reduced | $O(1+\log(\frac{a}{w}))$ | $O(1+\log(\frac{a}{w}))$ | $O(a+r^2)$ | $O(w)$ |
| Minimal | $O(1+\log(\frac{a}{w}))$ | $O((1+\log(\frac{a}{w}))^2)$ | $\Theta(a)$ | $O(1)$ |

TABLE II: AnchorHash implementation evolution: naive, then reduced-memory, then minimal-memory implementations. Successive implementations reduce the memory footprint to improve scalability, but the final implementation also slightly increases the upper bound on the expected number of memory accesses.




(a) AnchorHash key lookup rate with 1,000 working buckets with respect to different $a/w$ ratios.




(b) AnchorHash key lookup rate with respect to the number of working buckets for different fixed $\frac{a}{w}$ ratios. *AnchorHash(x)* stands for an AnchorHash instance with $\frac{a}{w} = x$.

Fig. 4: AnchorHash key lookup rate in millions of keys per second (Mkps).

Now, by solving the recurrence given by (17) and (18) we obtain

$$\mathbb{E}[\xi \mid |\mathcal{R}| = a - w] \leq 1 + \ln\left(\frac{a}{w}\right) + \ln^2\left(\frac{a}{w}\right) \leq \left(1 + \ln\left(\frac{a}{w}\right)\right)^2, \quad (19)$$

which concludes the proof. □

Finally, Table II summarizes the differences between the naive, reduced-memory, and the final minimal-memory implementations.

## VI. Evaluation

**Algorithms.** In this section we test and compare AnchorHash to HRW, Ring, and MaglevHash, according to the evaluation metrics of Table I: consistency (*i.e.,* minimal disruption and balance), key lookup rate, memory footprint, and update time upon additions and removals.

**Testbed.** All our experiments were conducted on a single core of a commodity machine with an Intel i7-7000 CPU at 3.6 GHz, 16 GB of RAM and an Ubuntu 16.04 LTS operating system. All algorithm implementations are in C++ and are optimized for run-time purposes. In our evaluation, each bucket has a 32-bit identifier (*i.e.,* up to $2^{32}$ buckets are supported), and we use 64-bit randomly-generated keys. For all algorithms we use the crc32 [23] hash function with two 64-bit inputs (key and seed) for uniform hashing.

**Memory footprint.** Before turning to empirical evaluation, we first discuss the memory footprint of the four approaches, as it has a significant impact on all other qualities such as key lookup rate and update time.

The memory footprint of Ring and MaglevHash depends on the theoretical hash-space balance guarantee these algorithms provide. For example, in MaglevHash, reaching a maximum of 1% hash space imbalance requires *at least* $\frac{1}{0.01} = 100$ copies for each resource. Throughout our evaluation, for MaglevHash and Ring we use 100 copies for each resource [1]. On the other hand, HRW and AnchorHash provide perfect hash-space balance and do not require copies to do so.

In our implementation, AnchorHash requires only 16 Bytes of memory per resource. This means that even for $10^6$ resources, AnchorHash uses 16 MB of space, whereas MaglevHash requires at least 400 MB to achieve a reasonable balance for the same scenario.

**Lookup rate.** We test AnchorHash's key lookup rate for different Anchor sizes (up to $10^8$) and different $\frac{a}{w}$ ratios (up to $10^3$). For example, $w = 1,000$ and $\frac{a}{w} = 100$ means that only $w = 1,000$ resources are still active out of $a = 100,000$ (*i.e.,* a scenario with 99,000 random removals).

The results are depicted in Fig. 4. Fig. 4(a) shows the key lookup rate achieved by AnchorHash with 1,000 working buckets with respect to different $\frac{a}{w}$ ratios. Fig. 4(b) depicts AnchorHash rate with respect to the number of working buckets for different fixed $\frac{a}{w}$ ratios. Note that, even for a fixed $\frac{a}{w}$ ratio, the rate slightly decreases as the number of buckets increases. This is because of the increased percentage of L3 cache misses as follows from the increased memory footprint. Remarkably, even for a million buckets, AnchorHash achieves a rate of tens of millions of keys per second for reasonable and even extreme operating points (*e.g.,* half of the buckets have been randomly removed).

Next, Fig. 5 compares the key lookup rates achieved by the four approaches for different number of resources. For AnchorHash, we depict three scenarios with 10%, 50% and 90% random removals, corresponding to AnchorHash(1.1), AnchorHash(2) and AnchorHash(10). AnchorHash(1.1) reaches a high key lookup rate that is similar to MaglevHash. As the resource count increases, MaglevHash suffers from a more significant rate degradation due to increased L3 cache misses that stem from its much larger memory footprint. On the other hand, as expected, the rate of AnchorHash decreases for higher percentages of random removals, due to the larger number of hash computations.

Additionally, we tested the lookup rate of the four approaches using a backbone router CAIDA trace [24]. The results follow similar trends. Interestingly, all approaches run faster since the often reoccurring flow packets increase the cache hit rate.

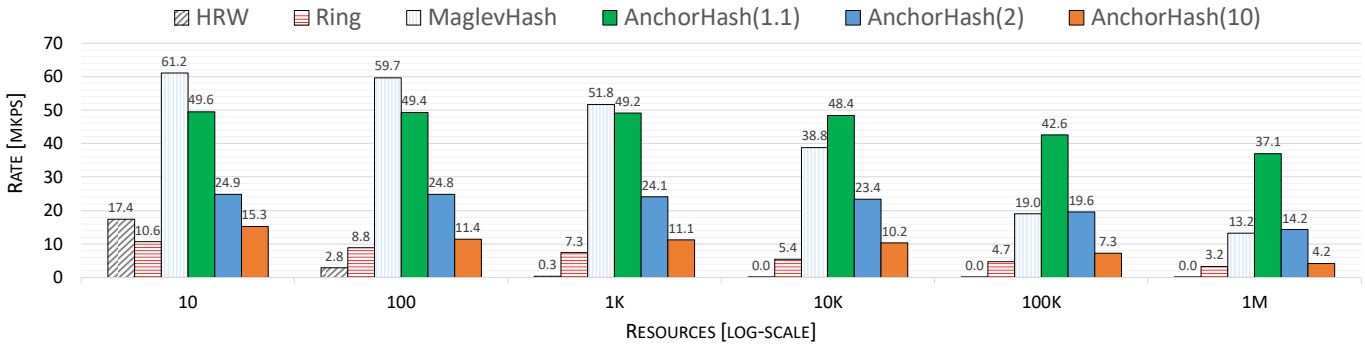



Fig. 5: Comparing the key lookup rates between HRW, Ring, MaglevHash and AnchorHash for different resource counts. Due to the significantly smaller memory footprint, AnchorHash maintains an extremely high rate even for $10^5$ resources.

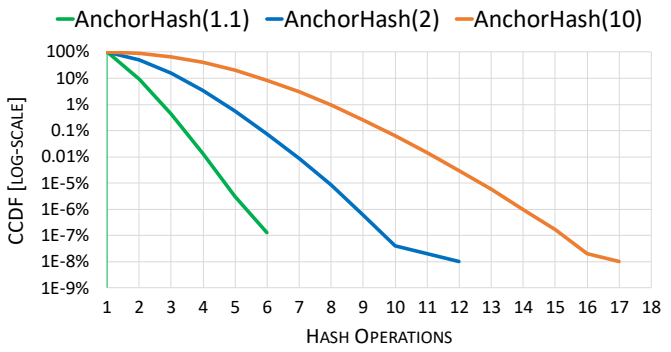



Fig. 6: CCDF for the number of hash operations performed by AnchorHash per key lookup. For example, for AnchorHash(2), 99.9% of keys would require 6 or less hash operations.

We also measured the number of hash operations for a key lookup of AnchorHash with 1,000 resources with an Anchor of 1,100, 2000 and 10000 buckets with 100, 1000 and 9000 random removals respectively (*i.e.,* AnchorHash(1.1), AnchorHash(2) and AnchorHash(10)). The number of simulated keys is $10^8$. The results are depicted in Fig. 6. For all versions of AnchorHash, it is evident that the number of hash operations is exponentially decreasing. Moreover, while the worst case in terms of hash operations is 101, 1001 and 9001 hash operations for the three versions of AnchorHash, out of $10^8$ keys no key required more than 6, 12 and 17 hash operations respectively. In AnchorHash(1.1), more than 90% of keys terminate after a single hash operation and less than 0.5% require more than 2. Even for AnchorHash(10), 99% terminate with less than 7 operations.

**Balance.** Essentially, there are three sources of imbalance, all reflected in an algorithm's load-balancing abilities: (1) hash space imbalance; (2) quality of the hash function; and (3) arriving keys. While the last two are implementation- and workload-dependent, the first is algorithm-dependent. Thus, in terms of balance, assuming uniform hashing, HRW and AnchorHash have an inherent advantage over MaglevHash and Ring. To demonstrate this, we tested the four approaches using the same hash function and a random stream of keys.

By standard practice [1] we measure the worst-case resource oversubscription in %. For instance, an oversubscription of 10% means that the most loaded resource has 10% more load than the average. All instances run with 1,000 resources. For AnchorHash we have an Anchor of 1,100, 2000 and 10000 buckets (corresponding to 100, 1000 and 9000 random removals respectively, *i.e.,* AnchorHash(1.1), AnchorHash(2) and AnchorHash(10)). Ring and MaglevHash both run with 100 copies per resource. The results are depicted in Fig. 7. As expected, the oversubscription improves for all algorithms as the number of keys increases. The oversubscription of MaglevHash and Ring are theoretically lower-bounded. Specifically, for MaglevHash it is at least 1.01 with 100 copies per resource and for Ring it is the inherent imbalance created by different size intervals in the ring (with high probability). Since HRW and AnchorHash (for all its versions) theoretically provide perfect balance, by the Law of Large Numbers, the oversubscription approaches zero as the number of keys increases. All three versions of AnchorHash have almost the same oversubscription indicating that the size of the anchor has no effect on the resulting balance (as expected). Note that HRW converges to 0 slightly faster than AnchorHash due to the large number of hash operations performed for each key, leading to better randomization.

**Update time.** We next test for the time it takes to update the data structure of each of the algorithms with a newly added or removed resource. The results are averaged over 100 trials, and depicted in Fig. 8. Both HRW and AnchorHash respond in nanosecond scale nearly independently of the size of the system. On the other hand, Ring and MaglevHash respond slower as the system size increases. For example, with $10^5$ resources, MaglevHash requires more than 4 seconds to respond.

**Minimal disruption.** We also test the minimal-disruption property for all approaches. Following theory, HRW, Ring and AnchorHash achieve the minimal-disruption property in practice as well. Unfortunately, MaglevHash fails to achieve minimal disruption and therefore is not fully consistent. For example, in a scenario with 900 resources and 100 consecutive resource additions, we find that *at each resource addition*, MaglevHash wrongfully reassigns a near-constant fraction of $\approx$ 0.6% of the hash space, *i.e.,* $\approx$ 0.6% of the keys are needlessly remapped at each of the 100 resource additions. While such *flips* may be acceptable when used together with key tracking (*e.g.,* connection tracking in datacenter load-

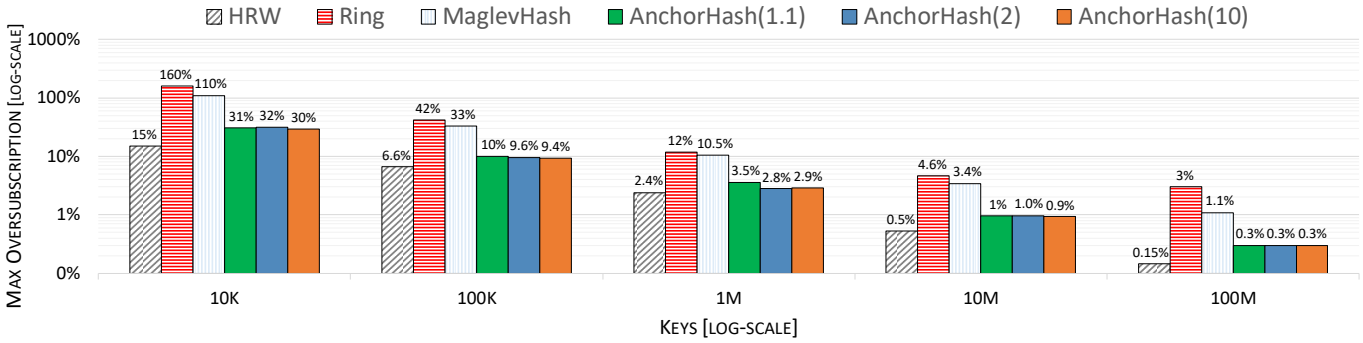



Fig. 7: Comparing worst-case oversubscription. Lower is better (better balance). All instances have 1,000 resources. For AnchorHash we have an Anchor of 1,100, 2000 and 10000 buckets with 100, 1000 and 9000 random removals accordingly (*i.e.,* AnchorHash(1.1), AnchorHash(2) and AnchorHash(10)).

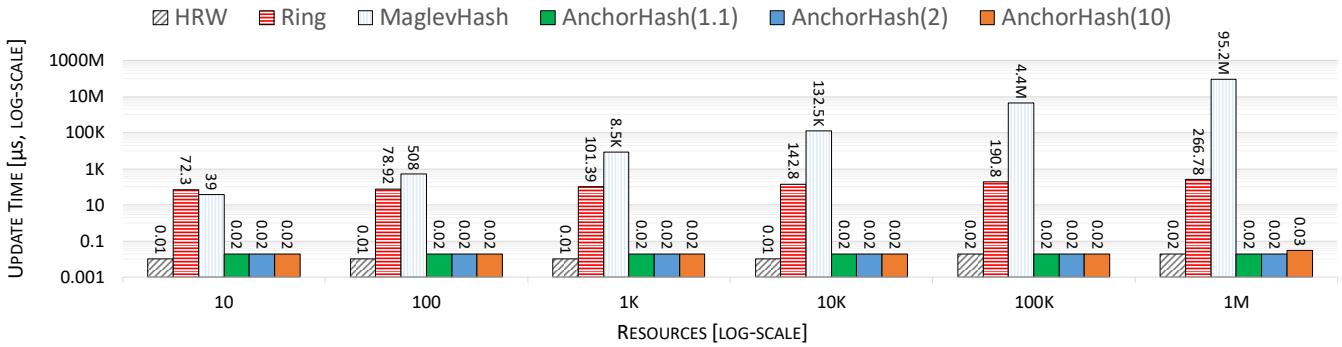



Fig. 8: Comparing update time for resource removals/additions. HRW and AnchorHash require only a few tens of nano-seconds independently of the size of the system. For Ring and especially MaglevHash, the update time increases with the size of the system. For example, for $10^5$ resources, MaglevHash requires more than 4 seconds to repopulate its array.

balancing), they may not be acceptable in other systems such as cache servers.

## VII. Conclusion

In this paper we introduced AnchorHash, a new consistent hashing technique. We provided implementation details and theoretical guarantees for AnchorHash. We then conducted evaluations comparing AnchorHash to existing algorithms. Evaluation results indicate that AnchorHash is a scalable and a fully-consistent hashing technique. It is capable of handling millions of resources while maintaining high key lookup rate, low memory footprint, and small update times upon resource additions and removals. Finally, the code for AnchorHash appears in [19].

## VIII. Future Work

Unlike other approaches, AnchorHash leverages state information to achieve its properties. Thus, in a distributed environment (*i.e.,* where multiple dispatchers run AnchorHash in parallel), AnchorHash requires an agreement on the removal order (*i.e.,* the content of $\mathcal{R}$) to ensure full consistency. While this overhead is small in terms of communication overhead (happens only once upon removal), it is of interest to study whether AnchorHash can be extended to maintain full consistency in a setting in which the dispatchers do not necessarily agree on the order of removals.

## Acknowledgments

This work was partly supported by the Hasso Plattner Institute Research School, the Israel Science Foundation (grant No. 1119/19), the Technion Hiroshi Fujiwara Cyber Security Research Center, and the Israel Cyber Bureau.

## References

[1] D. E. Eisenbud, C. Yi, C. Contavalli *et al.*, "Maglev: A fast and reliable software network load balancer." in *Usenix NSDI*, 2016.
[2] P. Goel, K. Rishabh, and V. Varma, "An alternate load distribution scheme in dhts," in *IEEE CloudCom*, 2017.
[3] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, 2003.
[4] D. Halperin, V. Teixeira de Almeida, L. L. Choo *et al.*, "Demonstration of the Myria big data management service," in *ACM SIGMOD*, 2014.
[5] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica, "Beacon vector routing: Scalable point-to-point routing in wireless sensornets," in *Usenix NSDI*, 2005.
[6] G. DeCandia, D. Hastorun, M. Jampani *et al.*, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS*, 2007.
[7] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*, 2002.
[8] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with Beamer," in *Usenix NSDI*, 2018.
[9] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa, "Balancing on the edge: Transport affinity without network state," in *Usenix NSDI*, 2018.
[10] D. Karger, E. Lehman, T. Leighton *et al.*, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM STOC*, 1997.
[11] D. Karger, A. Sherman, A. Berkheimer *et al.*, "Web caching with consistent hashing," *Comp. Netw.*, 1999.

[12] B. Appleton and M. O'Reilly, "Multi-probe consistent hashing," *arXiv preprint arXiv:1505.00062*, 2015.
[13] M. Sackman, "Perfect consistent hashing," *arXiv preprint arXiv:1503.04988*, 2015.
[14] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Trans. Netw.*, 1998.
[15] W. Wang and C. V. Ravishankar, "Hash-based virtual hierarchies for scalable location service in mobile ad-hoc networks," *Mobile Networks and Applications*, 2009.
[16] K. Park and V. S. Pai, "Scale and performance in the CoBlitz large-file distribution service." in *Usenix NSDI*, 2006.
[17] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," *arXiv preprint arXiv:1406.2294*, 2014.
[18] M. Uruena, D. Larrabeiti, and P. Serrano, "Fast robust hashing." in *IEEE Globecom*, 2006.
[19] "AnchorHash code." https://github.com/anchorhash.
[20] G. H. Gonnet and R. Baeza-Yates, *Handbook of algorithms and data structures: in Pascal and C, vol. 2*, 1991.
[21] D. E. Knuth, *Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3*, 1998.
[22] "xxHash for small keys: the impressive power of modern compilers," http://fastcompression.blogspot.com/2018/03/xxhash-for-small-keys-impressive-power.html.
[23] R. Singhal, "Inside intel® core microarchitecture (nehalem)," in *Hot Chips 20 Symposium (HCS), 2008 IEEE*. IEEE, 2008.
[24] P. Hick, "CAIDA Anonymized 2016 Internet Trace, equinix-chicago 2016-02-18 13:00-13:05 UTC, Direction A."


**Gal Mendelson** received his BSc, MSc (summa cum laude) and Ph.D. degrees from the Viterbi department of Electrical Engineering, Technion -– Israel Institute of Technology, in 2009, 2015 and 2020, respectively. He was the recipient of the Hasso Plattner Institute Ph.D. fellowship award and the INFORMS Applied Probability Society best student paper award. He is mainly interested in stochastic analysis, algorithms and communication networks.

**Shay Vargaftik** received his B.Sc. and Ph.D. degrees from the Viterbi department of Electrical Engineering, Technion -– Israel Institute of Technology, in 2012 and 2019, respectively. He was the recipient of the Hasso Plattner Institute and the IBM Ph.D. fellowship awards. He is currently a postdoctoral researcher in the VMware Research Group (VRG). He is mainly interested in the theory and practice of networking and machine learning with an emphasis on scalability and efficient resource usage.

**Katherine Barabash** received her B.Sc. in Applied Mathematics and M.Sc. in Computer Science degrees from the Technion – Israel Institute of Technology, in 1994 and 2010 respectively. Kathy is a researcher in IBM Research since 1997 and has contributed to system research in areas of memory management, storage, software defined networking, as well as other data center and Cloud technologies. Kathy's current research is devoted to Hybrid Cloud and 5G networking.

**Dean H. Lorenz** received his B.Sc. (summa com laude) in Computer Engineering and Ph.D. in Electrical Engineering, from the Technion, Haifa, Israel. He is Researcher at IBM Research – Haifa, where he is a technical leader in the Cloud Architecture Networking group, in the Hybrid Cloud department. Dr. Lorenz has more than 20 years of experience in research, hands-on development, and innovation in Networking, Virtualization, Storage, and Mobile Technologies; and has held technical positions at leading companies in these industries, including IBM Research, Akamai, Adobe Omniture, and Qualcomm. His current research is Cloud technologies, with focus on Cloud networking, AIOps, elasticity, and operation efficiency.

**Isaac Keslassy** (M'02, SM'11) received his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University, Stanford, CA, in 2000 and 2004, respectively. He is currently a full professor in the Viterbi department of Electrical Engineering at the Technion, Israel. His recent research interests include the design and analysis of data-center networks and high-performance routers. He was the recipient of an ACM SIGCOMM test-of-time award, of an ERC Starting Grant, and of the Allon, Mani, Yanai, and Taub awards. He was associate editor for the IEEE/ACM Transactions on Networking.

**Ariel Orda** (S'84, M'92, SM'97, F'06) received the BSc (summa cum laude), MSc, and DSc degrees in electrical engineering from the Technion, Haifa, Israel, in 1983, 1985, and 1991, respectively. During 1.1.2014-31.12.2017, he was the dean of the Viterbi Department of Electrical Engineering, Technion. Since 1994, he has been with the Department of Electrical Engineering, Technion, where he is the Herman and Gertrude Gross professor of communications. His research interests include network routing, the application of game theory to computer networking, survivability, QoS provisioning, wireless networks, and network pricing. He served as program co-chair of IEEE INFOCOM 2002, WiOpt 2010 and Netgcoop 2020, and general chair of Netgcoop 2012. He was an editor of the IEEE/ACM Transactions on Networking and Computer Networks. He received several awards for research, teaching, and service.