Meep Hashing: Ultrafast and Compact Minimal Perfect Hashing for Practical Large-Scale Lookup Systems

SHOUQIAN SHI*, Nanjing University, China
DIANCHENG LUO, Nanjing University, China
JACQUES LIAO, Google, USA
YI LIU, University of California, Santa Cruz, USA
XINGSHENG ZHAO, Google, USA
MATTHEW FAHRBACH, Google, USA
CHEN QIAN, University of California, Santa Cruz, USA

Minimal perfect hashing (MPH) is a class of algorithms that maps a set of N keys to the range $\{0, 1, \cdots, N-1\}$ without collisions. MPH is widely employed in modern large-scale applications including distributed storage indexes, network packet classification, genomic encoding in Bioinformatics, *etc.* In practice, these systems are often augmented with alien-key filters to ensure robustness against unexpected inputs. This paper introduces Meep hashing, a novel minimal perfect hashing algorithm designed for ultrafast lookup with an integrated alien-key filtering mechanism. Among existing lookup systems, Meep achieves the lowest on-average and tail lookup latency while maintaining a memory footprint comparable to the most compact perfect hashing data structures. It supports configurable target false positive rates for alien-key filtering, enabling flexible trade-offs between accuracy and efficiency. The performance advantages of Meep are realized through a hybrid orchestration of Cuckoo-based and probing-based hashing techniques, combined with a collision resolution strategy that handles four distinct collision types with minimal or zero additional overhead. We implement Meep and evaluate its efficacy in lookup speed, memory footprint, and construction time using microbenchmarks. Experimental results show that Meep hashing is consistently 50% to 200% faster than the other MPHs and 100% faster than the other MHTs. Meep achieves more than 7 billion queries per second under Zipfian query key distribution on a single machine with 20 parallel lookup threads.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Information systems** → *Distributed storage.*

Additional Key Words and Phrases: Minimal Perfect Hashing, Key-value lookups, Compact data structures

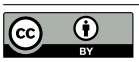
ACM Reference Format:
Shouqian Shi, Diancheng Luo, Jacques LIAO, Yi Liu, Xingsheng Zhao, Matthew Fahrbach, and Chen Qian. 2026. Meep Hashing: Ultrafast and Compact Minimal Perfect Hashing for Practical Large-Scale Lookup Systems. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 210 (June 2026), 22 pages. https://doi.org/10.1145/3802087

1 Introduction

Minimal perfect hashing (MPH) provides a space-efficient solution to the static "key-index mapping" problem: given a fixed set of N distinct keys, an MPH bijectively maps each key to a unique integer in the range $\{0, 1, \cdots, N-1\}$. The key-index mapping is determined by MPH algorithm itself instead

*Corresponding author

Authors' Contact Information: Shouqian Shi, sqlite@nju.edu.cn, Nanjing University, China; Diancheng Luo, Nanjing University, China, 231220136@smail.nju.edu.cn; Jacques LIAO, Google, USA, qhliao@google.com; Yi Liu, University of California, Santa Cruz, USA, yliu634@ucsc.edu; Xingsheng Zhao, Google, USA, xingshengzhao@google.com; Matthew Fahrbach, Google, USA, matthew.fahrbach@gmail.com; Chen Qian, University of California, Santa Cruz, USA, cqian12@ucsc.edu.



ACM 2836-6573/2026/6-ART210
https://doi.org/10.1145/3802087

of some predefined assignments. MPH is widely employed in modern large-scale applications including distributed storage indexes [11, 26, 29, 43], network packet classification [44, 46], and even genomic encoding in Bioinformatics [33].

These applications have four common requirements for the lookup algorithm. **1) Fast lookup.** Lower on-average latency improves the overall system throughput for a given computation capacity, and lower tail latency greatly improves round trip time for batched lookup services. **2) Space efficiency.** For large-scale lookup systems running in the cloud, using less DRAMs directly saves the hardware acquisition and operation costs; for in-memory caching systems, fitting more key-value items in a fixed-sized DRAM supports saves on-board bandwidth. **3) Alien key filter.** Detecting a query key that was not in the item set (i.e., an **"alien key"**) in an earlier stage of the system processing pipeline prevents resource wastes on non-existing key lookups, and the percentage of alien keys is an important indicator of system health. **4) Collision free.** To guarantee the system correctness, two different keys should not collide to the same lookup result.

Hash tables are the most adopted approach for key-value lookup systems due to their amortized $O(1)$ lookup time and alien key proof by design. Examples include traditional separate-chaining and open addressing hash tables [12], Cuckoo hash tables [22, 25, 30, 39], and some other modern variants [5, 23]. Most hash tables store keys to resolve collisions. In practical systems [11, 26, 29, 43], however, keys can be several thousand bytes long, causing a huge amount of space cost and increased lookup latency—for each lookup, one or more full-key comparisons are required to locate the queried key in the hash table.

While minimal perfect hashing (MPH) [3, 14, 21] algorithms are designed for key-to-index mapping, they can also be adapted to "key-value lookup" scenarios for fast lookup of a list of predetermined key-value pairs. Given N key-value items $(k_i, v_i), i = 0, \cdots, N-1$, an MPH function H bijiectively maps N keys to $\{0, 1, \ldots, N-1\}$, and store the v_i at the $H(k_i)$-th slot in an N-element value array. To look up a key k, just return the $H(k)$-th value in that array. Traditional MPH algorithms exhibit drawbacks when used in practical lookup systems, including high space cost, high latency, and prune to alien keys [3, 3, 14]. Several variations have been proposed such as Othello [37, 44], SetSep [17, 45], Ludo [36], and PTHash [34].

Othello and SetSep are fast key-value lookup algorithms. Similar to MPH, they both do not store full keys. Different from MPH, they can only support key-value lookup and lack key-index mapping ability. To distinguish them with typical MPH algorithms, we call these algorithms "**Minimal Hash Table (MHT)**". Besides, Othello and SetSep fall short in long value lookup scenarios — for a single value bit, they cost 2.33 or 1.5 bits per key, resulting in a "**value amplification**" of 133% or 50% respectively. Ludo supports both key-index and key-value lookup by its table-based design and enjoys low value amplification. However, Ludo is non-minimal: it maps N keys to $\sim 1.038N$ indexes, wasting 3.8% of the index space. We call this **"index amplification"**[1]. PTHash is a recently proposed MPH algorithm with decent construction and lookup speed by a careful parameter fine-tuning. By design, PTHash has long tail latency for lookup and does not support integrated value lookup.

It is also important to understand that for an alien key, an MPH or MHT algorithm still returns an arbitrary result. Meep, however, decimates the alien key rate by design: the alien key filter is built-in without compensating lookup latency and memory bandwidth.

This paper presents Meep hashing[2], a novel table-based MPH algorithm that provides the lowest average lookup latency, unparalleled tight bound on tail latency, configurable alien key filter, and

[1]Note that in table based lookup systems there is a similar concept **load factor**, which does not directly depict the index space amplification.

[2]"Meep! Meep!" is the sound of the greater roadrunner (Geococcyx californianus), a bird in the cuckoo family that has the fastest running speed of all flying birds.

a highly compact overall memory footprint. Specifically: **1)** It offers the lowest average lookup latency and the highest lookup throughput among all known MPH and MHT algorithms. **2)** It also provides the lowest tail lookup latency among all known lookup systems, enabling the fastest batched lookup. **3)** It has built-in alien key filter. **4)** Its space cost is among the lowest tier due to its compact memory layout and its zero index amplification. All these advantages make Meep a highly competitive solution for both key-value and key-index large-scale lookup systems.

The advantages of Meep are derived from a novel orchestration of all the benefits of Cuckoo hash table, SwissTable, Cuckoo filter, and linear-probing based hashing, while carefully avoiding their individual drawbacks. A series of innovations are proposed to overcome a variety of challenges without compromising performance in terms of latency, space, and alien key detection. The core idea of Meep can be demonstrated by two major steps. **Step 1)** Distribute N keys to $\lceil N/16 \rceil$ buckets, each bucket containing up to 16 keys. **Step 2)** Rearrange slots inside each bucket, to avoid all four types of collisions without extra space cost, including local collisions, remote collisions, overflow collisions, and empty collisions, which are covered in § 5. The first step is a novel use of (2,16)-Cuckoo hashing due to its unique advantages in high load factor and compatibility with SIMD-based ultrafast batched matching. The second step is nontrivial by design and is the hardcore of Meep algorithm. Existing minimal perfect hashing algorithms use extra storage to maintain the choice of hash function [14, 17, 36], but this approach is slow in lookup. Meep hashing avoids all collisions with zero extra space cost. This is achieved by a novel probing rule inside buckets and a careful rearrangement of slots in each bucket while avoiding all collisions. Overall, Meep hashing achieves an overflow rate of just 0.53% and exhibits zero index amplification, while remaining fully compatible with ultrafast batched lookups and effective at filtering out 95% of alien keys.

The core contributions of this paper are summarized below:

(1) A careful conceptual distinguish and numerical comparison between MPH and MHT.
(2) Identifications of four types of collisions in Cuckoo filter based MPH algorithms.
(3) An algorithm to resolve all collisions with two machine instructions during lookup.
(4) Overall design of Meep hashing, an ultrafast MPH with builtin key-value lookup, key-index lookup, and alien-key filter.
(5) Prototype implementation of Meep in C++ for reference and result reproduction [1].

We implemented the complete software of Meep, providing all its aforementioned advantages. The **open-sourced** version of Meep is available for result reproduction [1].

The rest of this paper is organized as follows. Section 2 presents the related work. Section 4 defines the problem model. We present the detailed design of Meep in Section 5 and the design optimizations in Section 6. The system implementation and evaluation results are shown in Section 7. We conclude this work in Section 8.

2 Related Work

Practical key-value lookup systems are large scale in both key volume and concurrent queries. Batched lookup speed, space cost, and alien key detection are their major concerns.

Hash tables are the most adopted approach for key-value lookup systems [5, 22, 25, 30, 39], due to their amortized $O(1)$ lookup time and alien key proof by design. However, many practical systems have large keys [11, 13, 26, 27, 32, 35, 38, 43], costing hash tables more space to store the key-value items and more time to lookup. In particular, **Cuckoo hashing** [30] is a key-value mapping data structure that achieves $O(1)$ lookup time in the worst case and amortized $O(1)$ update time [15, 28, 30, 41]. **SwissTable** [5] is a quadratic probing hash table with a highly optimized lookup speed.

Minimal perfect hashing algorithms [3, 14, 21, 34] are also used for key-value lookup and shows great potential to significantly reduce the space usage by not storing the full keys while retaining the

Solution	Full-key required	Alien-key proof	Key-index lookup		Key-value lookup		Bits per item	Throughput (Mqps)	
			Supported?	A_i	Supported?	A_v		uniform	Zipfian
(2,4)-Cuckoo Table	Yes	Yes	Amp	5%	Yes	5%	$1.05(L+l)$	18	22
SwissTable	Yes	Yes	Amp	12.5%	Amp	12.5%	$1.125(8+L+l)$	32	41
RecSplit $\ell=8$ $b=100$	No	No	Yes	0	Ext-array	0	$1.8+l'_d+l$	12	13
PTHash D-D	No	No	Yes	0	Ext-array	0	$4.05+l'_d+l$	52	56
PTHash C-C							$3.36+l'_d+l$	54	58
SetSep	No	No	No	-	Amp	50%	$0.5+1.5(l'_d+l)$	24	28
Othello	No	No	No	-	Amp	133%	$2.33(l'_d+l)$	63	65
Ludo	No	No	Amp	3.8%	Yes	5%	$3.76+1.05(l'_d+l)+0.012(L+l)$	16	18
Meep	No	>95%	Yes	0	Yes	0.53%	$3.77+l'_d+l+0.0053(L+l)$	**127**	**145**

Table 1. Comparison of lookup algorithms. L: key length in bits. l: value length in bits. A_i: index amplification. A_v: value amplification. $l'_d=-\log_2 R_f$, where R_f is the target false positive rate. Throughput is evaluated on 8 million keys. RecSplit and PTHash are non-table-based MPH and utilize an external array for key-value lookup.

ability of mapping keys correctly to their corresponding values. Ludo hashing [36] is a non-minimal perfect hashing algorithm that supports dynamic updates. PTHash [34] is an improvement over FCH [20] on space and construction time while keeping the original 3-step structure of *mapping, ordering, and searching*. RecSplit [14] uses brute-force to resolve collisions in small sets and split the large sets in recursive way until the resulting smaller sets can be resolved.

Minimal Hash Tables (MHTs) support key-value lookup while lacking key-index lookup ability. Bloomier filters [9, 10] use a bipartite graph and its acyclicity to resolve collisions. Othello Hashing is a data structure and a series of algorithms based on Bloomier filters designed for dynamic forwarding information bases [44]. Coloring Embedder [42] and SetSep [17, 45] use hierarchical hashing and brute force at last level to resolve collisions. Quotient filters [4, 31] are static lookup algorithms based on linear-probing hashing. In addition to alien key false positives, it further has wrong lookup answers for existing keys.

Filters are for approximate membership queries. For a given set S, a filter answers whether any given key k is in S. Since filters do not store full keys, they might wrongly recognize k' while $k' \notin S$. k' is called an **"alien key"**, and a **"false positive"** of that filter. Bloom filters [6, 18] are popular due to their simplicity but slow [16, 40]. Cuckoo filter [16, 40] is proposed to replace Bloom filter for less space cost and less hash computations and memory loads per lookup. The core design is to replace the keys in the Cuckoo hash table by the digest of keys. Vacuum filter [40] improves Cuckoo filter by solving several practical problems. Cuckoo Filtable [37] is a chained table design to address more collisions.

Meep alleviates the burden to save full keys, provides built-in alien key detection, is compatible for both key-index and key-value lookup, and enjoys highly compact overall memory footprint and ultrafast lookup. We compare Meep with the representative lookup algorithms, including hash tables, MPHs, and MHTs, in Table 1. All numerical results are under the same set of evaluation configurations as in § 7.

3 Background Knowledge

We elaborate on some details of Cuckoo hash table and SwissTable in this section as background knowledge.

Cuckoo hash table. We note that (2,4)-Cuckoo is the de facto standard configuration [15, 30, 36, 45, 46]. As shown in Fig. 1, a (2,4)-Cuckoo has a number of buckets, each bucket has 4 slots, and every key-value pair is stored in one slot of the two *alternate buckets* based on the two hash

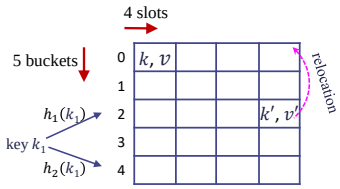


Fig. 1. (2,4)-Cuckoo Hash Table

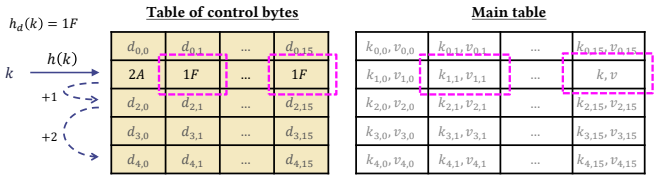


Fig. 2. SwissTable

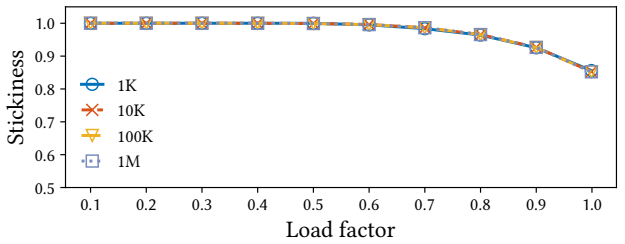


Fig. 3. Stickiness for (2,16)-Cuckoo

values $h_0(k)$ and $h_1(k)$. To insert an item with key k_1, a single empty slot should be found in bucket $h_0(k_1)$ or $h_1(k_1)$. If both the buckets are full, one existing item (e.g., the one with key k' in Fig. 1) will be relocated to its alternate bucket, and k_1 takes the slot of k'. If the alternate bucket of k' is full as well, an item in that bucket will be relocated recursively. This process stops when every item is placed in a slot. The selection of k' is typically done via breadth-first search.

SwissTable [5] is a quadratic probing hash table with a highly optimized lookup speed. As Fig. 2 shows, each bucket of the main table contains up to 16 key-value items, and each item is associated with a separate control byte. A control byte can be any of: empty flag, tombstone flag, or a 7-bit **"digest"** of the corresponding key, *e.g.*, the lower 7 bits of the key's hash value. To look up an item with key k, a hash function H maps k to a bucket, and another hash function H_d calculates its 7-bit digest d to match with the 16 control bytes via SIMD instructions. On a matched digest, compare the full key, in addition to the digest, to confirm the actual match. Since the match between d and 16 control bytes can be done via a single instruction[3] on a modern CPU, this lookup within a bucket is superfast. SwissTable resolves key clustering with quadratic probing (index +1, +2, +3, $\cdots$), until an empty flag is met. A major drawback of SwissTable is the unbounded lookup latency due to its probing-based design, especially when the table is near full. In practical systems, lookup requests are batched, which means long tail latency will occur on almost all batches, leading to a large latency for each batch.

Stickiness of Cuckoo hashing. For the following discussion, we define three terms: strong key, weak key, and stickiness. A key k is a **strong key** if it is located in its first-choice bucket $b_1(k)$; otherwise, it is a **weak key**. The **strong keys of a bucket** are the keys located at this bucket and take this bucket as their first choice bucket. The **weak keys of a bucket** are keys take this bucket as their first choice bucket but *not* located at this bucket. A Cuckoo-based table has a **stickiness** value, which equals to the portion of keys that are located in their first-choice buckets [36].

We show the stability of the stickiness value R_s for (2,16)-Cuckoo hash tables by a series of experiments. For each combination of the different table capacities $|S| \in \{1K, 10K, 100K, 1M\}$ and table load factors $R_l \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.999\}$, we build ten (2,16)-Cuckoo hash tables by inserting random elements to a pre-sized empty Cuckoo hash table, use breadth-first search to find the Cuckoo path during insertion, and calculate the average stickiness of all keys. The results are shown in Fig. 3. The stickiness is consistent across different table capacities, and only

[3] *e.g.*, Streaming SIMD Extension (SSE) instruction *_mm_cmpeq_epi8* in the Intel SSE2 instruction set [24].

Symbol	Description
S	set of key-value items
N	total number of valid keys, $N = \|S\|$
$\langle k,\ v\rangle$	item with key k and value v
$h_d(\cdot)$	hash function to generate the digest for any key
l	length of a value
l_d	length of the digest of a key
n_s	number of slots in a bucket (default 16)
$h_1(\cdot),\ h_2(\cdot)$	hash functions to locate the 1st and 2nd bucket of any key
$b_1(\cdot),\ b_2(\cdot)$	indexes of the 1st and 2nd bucket of any key
$s_1(\cdot),\ s_2(\cdot)$	indexes of the preferred slot in the 1st and 2nd bucket
R_f	target false positive rate configured before `construct`
R'_f	actual false positive rate achieved after `construct`
R_l	load factor of a Cuckoo table
A_i, A_v	index amplification and value amplification
R_s	stickiness of a table
R_o	overall overflow rate of the Meep main table
p	probing size

Table 2. Notation

depends on the load factor, and for a near-full (2,16)-Cuckoo hash table, the stickiness is around 85% for any table size. Hence, we make a key observation: **(O1) most keys in (2,16)-Cuckoo are strong keys**.

4 Problem Definition and Notations

We formally define the problem scope of this work. Given a set of key-value items $S = \{\langle k_i,\ v_i\rangle \mid i \in [N].\ \forall l \neq m,\ k_l \neq k_m\}$. The goal of this work is to find a practical algorithm by providing the following functions with minimized time and space costs and a configurable false positive rate R_f.

(1) The construction function $\texttt{construct}(S, R_f)$ constructs a lookup table for set S at the target false positive rate R_f.
(2) The key-value lookup function $\texttt{query}(k)$ returns v, if $\langle k, v\rangle \in S$.
(3) The key-index lookup function $\texttt{index}(k)$ returns a number $n \in [N]$, where $\forall i \neq j \wedge k_i,\ k_j \in S, \texttt{index}(k_i) \neq \texttt{index}(k_j)$.
(4) For any key $k' \notin S$, $\texttt{query}(k)$ and $\texttt{index}(k)$ return $\perp$ for most cases, and only return an arbitrary value (or index) at a probability $R'_f \leqslant R_f$.

For scenarios where only key-index lookup is needed and key-value is not, the input set is a set of distinct keys. $S = \{k_i \mid i \in [N].\ \forall l \neq m,\ k_l \neq k_m\}$.

We summarize the notation used in the following discussions for reference in Table 2.

5 Design of Meep Hashing

5.1 Core framework

A practical key-value or key-index lookup system requires low lookup latency, less memory bandwidth consumption, small space cost, and alien key filter. Meep meets all the above requirements by carefully transforming a (2,16)-Cuckoo hash table to an MPH while overcoming four types of collisions.

The construction of Meep consists of 3 major steps, 1) template construction, 2) template transformation, 3) permutation and overflow. The steps are also illustrated in Fig 4, Fig 5, and Fig 6.

Step 1) Given N distinct keys, a (2,16)-Cuckoo hash table of $\lceil N/16\rceil$ buckets is built by inserting each of the keys into the table per the Cuckoo hashing. As we strictly assign N slots for N keys,

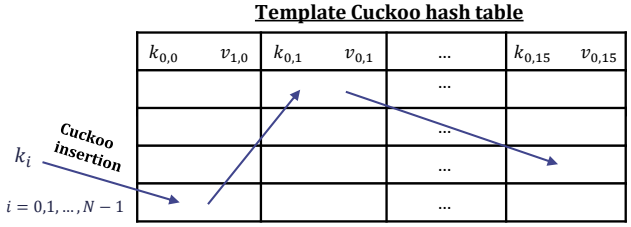


Fig. 4. Step 1, template construction

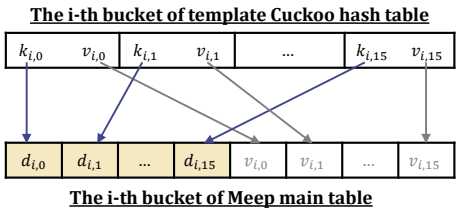


Fig. 5. Step 2, copy template to Meep

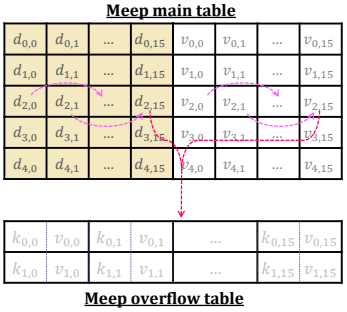


Fig. 6. Step 3, permute to resolve collisions

and the maximum load factor is less than 1, a small portion of insertions will fail. All the failed keys are inserted to a small overflow table, and all keys are tried for insertion regardless of previous failures if any. The Cuckoo insertion is intensively researched in theory as the graph k-orientation problem and analytical results show that the load factor of Cuckoo hashing is *asymptotically almost surely (a.a.s)* more than 99.93%[4] when n is sufficiently large by Cain *et al.* in 2007 [8].

Step 2) We use the constructed near-full (2,16)-Cuckoo hash table as a **"template"** and transform it to a same-sized new table with $\lceil N/16 \rceil$ buckets, each bucket containing 16 digests. And for key-value lookup, each bucket also contains 16 values. Each digest is 1 byte in length and calculated by $h_d(k)$. The overflow table is intact. This step greatly reduces the memory consumption, *e.g.*, Meep saves more than 90% memory compared to a Cuckoo hash set when keys are longer than 10 bytes, which is common in practice.

Step 3) The digests in each bucket are permuted to resolve digest collisions, and failures are relocated to the overflow table. In Fig. 6, the algorithm decides to relocate $d_{2,0}$ to $d_{2,2}$ and $d_{2,1}$ to $d_{2,15}$, $\cdots$, and $d_{2,15}$ to overflow. The values permute with the digests to make each digest-value pair consistent in their positions. The permutation algorithm is the hard core of Meep algorithm and is elaborated in the rest of this section.

Before all the design details, let us focus on the challenges and opportunities behind this approach. Step 2 outputs a lookup table "keyed" by digests. Each key or digest has 32 potential collisions because each key has two alternative buckets, each containing 16 digests. On one hand, the major challenge is how to resolve all collisions via merely 1-byte digests. On the other hand, if the collisions are efficiently resolved, we can implement the ultrafast lookup as follows.

Ultrafast key-index lookup. To look up the value of a key k, first the digest is calculated $d = h_d(k)$. Then d is matched against all the 16 slots in the $h_1(k)$-th bucket of the table in a single

[4]To verify this, we conduct experiments on the 1-grams and 2-grams from the English GoogleBook datasets for 1K, 10K, 100K, and 1M different keys with 10 set of different random hash seeds for the Cuckoo hash table. We use Abseil hash function and breadth-first search for Cuckoo path searching. The load factor is always around 99.95%, even better than the theory bound because on insertion failures, we continue to insert instead of stop and skip the rest.

SSE instruction. And we apply some efficient collision resolution to precisely determine the position of k in this bucket. If not in this bucket, match again with bucket $h_2(k)$. If both buckets fail, query the overflow table, which is a regular Cuckoo hash table.

```
index(k) :
    d ← h_d(k)
    for n = 1, 2 do
        i_b ← h_n(k)
        // Batch matching via single SSE instruction
        I ← 16-bit, i-th bit set iff. d equals the i-th digest of b_n
        // If k is not in this bucket, i_s is set to n_s
        i_s ← efficient collision resolution on I
        if i_s < n_s then return i_b n_s + i_s
    return query results of k on the overflow table
```

We make two interesting notes: 1) As the template Cuckoo hash table has stickiness 0.85, we expect 85% lookup finishes at 1 memory load, making Meep lookup very fast. 2) After step 3, the final overflow table contains only a tiny portion (0.53%) of all keys. Chaining the overflow table after the main table avoids two memory loads from the overflow for the majority (99.5%) keys in the main table.

Meep supports both key-index and key-value lookup. For N input keys, $\lceil N/n_s \rceil$ buckets are allocated for the main table, and the last N mod n_s slots of the last bucket are forbidden to use, resulting in exactly N available slots. After Meep construction, the overflow table contains R_oN keys, and the main table holds exactly $(1 - R_o)N$ digests and R_oN unoccupied slots. For the i_s-th slot at the i_b-th bucket, we define the global index of that slot as $i_b n_s + i_s$. Then **for key-index lookup**, each key is bijectively mapped to the index space $0, 1, 2, \cdots, N - 1$ by the position of its corresponding digest. The R_oN unoccupied slots can be freely assigned to the R_oN overflow keys, and hence, all N input keys are assigned an index without collision. **For key-value lookup**, the main table contains both digests and values, and each key is mapped to the corresponding value of the matched digest, and the overflow table contains the key-value pairs of the overflow keys. We make another important observation: **(O2)** the main table and overflow table together make Meep hashing an MPH algorithm for both key-value and key-index lookup.

Ultrafast key-value lookup. The key-value lookup algorithm `query`(k) is slightly different in that it returns the corresponding value as opposed to the global slot index. Fig. 7 also shows the memory layout and lookup workflow[5].

```
query(k) :
    d ← h_d(k)
    for n = 1, 2 do
        i_b ← h_n(k)
        I ← 16-bit, i-th bit set iff. d equals the i-th digest of b_n
        i_s ← efficient collision resolution on I
        if i_s < n_s then return i_s-th value
    return query results of k on the overflow table
```

With the ultrafast lookup algorithms above, the rest of this section is developed around the design on the "efficient collision resolution" part. We start this design by analyzing the collisions.

[5]Note for key-index lookup, Meep main table only holds digests and do not allocate value part in each bucket.

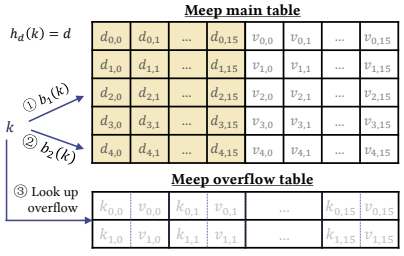


Fig. 7. Meep hashing lookup

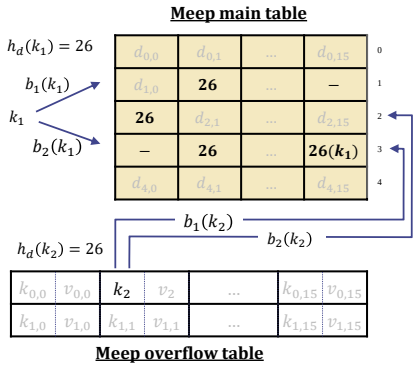


Fig. 8. Collisions during Meep construction

5.2 Meep collision types

All collisions originate from Meep's special data layout and query methodology. We use Fig. 8 to demonstrate all four types of collisions. In this example, the table has 5 buckets, each containing 16 digests, and we assume $h_1(k_1) = 1$, $h_2(k_1) = 3$, and $h_d(k_1) = \texttt{0x26}$, and the real position of k_1 is at bucket 3 slot 15. There happen to be 3 digests equal to `0x26` in buckets 1 and 3.

Per the lookup algorithm above, the first alternate bucket of k_1 is checked, which is bucket 1, even though it does not hold the real position of k_1. Hence, we call the `0x26` in bucket 1 slot 1 a **"remote collision"** of k_1, and the collision resolution algorithm should identify this and return n_s, or 16, to denote the bucket does not really hold k_1.

To continue the lookup of k_1, bucket 3 is checked, and the collision resolution algorithm should precisely resolve the ambiguity of two `0x26` values in this same bucket, and return the real position of k_1 in this bucket, which is 15. We call the two `0x26` digests a **"local collision"**.

In Fig. 8, assume key k_2 fails to resolve collisions and is located to the overflow table, and $b_1(k_2) = 3$, $b_2(k_2) = 2$, and $h_d(k_2) = \texttt{0x26}$. There are 3 occurrence of digest `0x26` in the buckets 2 and 3, which are called **"overflow collisions"** of k_2. Per the lookup workflow, the collision resolution algorithm should be able to reject all the three colliding positions before the correct lookup happen at the at the overflow table. Note the overflow is a hash table and the collisions are naturally resolved by keys.

The dashes "–" in Fig. 8 denote unoccupied or empty slots. We do not introduce a special flag or a special value, *e.g.,* `0xFF`, to indicate the emptiness. Instead, each empty slot is still assigned a 1-byte digest as a placeholder. In this way, we achieve both small memory footprint and ultrafast lookup speed, at the cost of potential digest collisions, which we call **"empty collisions"**.

With all possible collisions laid out above, our ultimate goal is to invent a novel co-design of both a digest arrangement algorithm during construction and a digest probing algorithm during lookup, to support the ultrafast lookup. As a result, **Meep uses merely two machine instructions to resolve all four types of collisions during lookup**, which corresponds to the "efficient collision resolution on I" in the above `index`$(\cdot)$ and `query`$(\cdot)$ algorithms. We cover the details below.

5.3 Type 1: local collisions

Revisiting the example in Fig. 8, key *shadowing* happens due to the digest collision of two digest-value items in the same bucket. This type of collisions are local to the bucket, and not relevant to any digests in any other buckets. We call them "local collisions".

The local collision problem is related to the design of Ludo hashing [36] and SetSep [17] in that they all need to reconcile the key collisions within a small group of 4 to 16 elements. Ludo faces with a much smaller search space—in each bucket, it arranges 4 keys to 4 slots. This can be

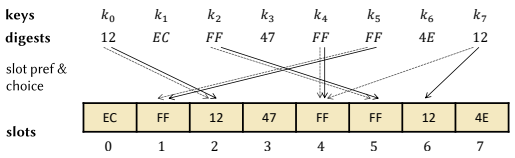


Fig. 9. Meep resolvable local collisions

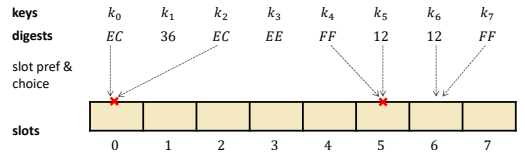


Fig. 10. Meep non-resolvable local collisions

trivially achieved via brute-force search. SetSep, however, needs to arrange 16 keys to 16 slots. Brute-force search of a SetSep bucket takes a long time, since the success probability of each attempt is $16!/16^{16} \approx 1.1 \times 10^{-6}$. Instead, SetSep uses an array together with a seeded hash to encode the arrangement.

Meep has 16 slots to reconcile, the same search space size as SetSep, but we want to invent and utilize the sub-structures inside a bucket to reduce the search space without brute force or additional array encoding. Fortunately, we have an important observation: **(O3) Only a partial order instead of a total order among the n_s slots is necessary** — in a single bucket, the slots with the same digest value should be fully ordered, while different digests can be in an arbitrary order. We claim that with the partial order defined, most of the local collisions can be resolved by naive linear probing as shown below. We call the keys with the same digest value in the same bucket a **"collision group"**.

We first modify the definition of $h_1(\cdot), h_2(\cdot)$: instead of returning only the bucket index, each hash function now returns a slot-level location $\langle b,\ s\rangle$, where b is the bucket index, and s is the index of its **"preferred slot"** in bucket b.[6] In the discussions below, we use $b_1(\cdot), b_2(\cdot), s_1(\cdot), s_2(\cdot)$ to denote the bucket and slot indexes generated by hash functions $h_1(\cdot), h_2(\cdot)$, such that $h_1(k) = n_s b_1(k) + s_1(k)$, $h_2(k) = n_s b_2(k) + s_2(k)$. And to avoid ambiguity, we always use b_1, s_1, b_2, s_2 in the following discussions and avoid using h_1, h_2.

We then introduce a **new rule** to the lookup process: to look up a key k, first try matching the digest of its preferred slot s, and if it is a mismatch, try slots $s+1, s+2, \cdots$ (in terms of $\bmod n_s$), as done in linear-probing style hash tables. With this rule, if two keys k_1, k_2 of the same colliding group have different preferred slots s_1, s_2, and $s_1 < s_2$, to resolve collisions, k_1 should be located to any of $s_1, s_1+1, \cdots, s_2-1$, and k_2 should be located to any of $s_2, s_2+1, \cdots, n_s-1, 0, 1, \cdots, s_1-1$.

We show a concrete example in Fig. 9, and for demonstration, the bucket contains 8 slots, as opposed to the actual number 16. We use 2 hexadecimal digits to denote the 1-byte digest. For this bucket, we have a two-key collision group for digest `0x12` and a three-key collision group for digest `0xFF`. One possible arrangement is shown in the figure, where the dashed arrows point to the preferred slots of the keys, and the solid arrows point to their final locations. Since all the non-colliding digests are free to arrange, their arrows are omitted. Per the new probing-style lookup, k_4 must be located at slot 4, or k_4 and k_2 will shadow each other. And after placing k_4 at slot 4, as the lookup of k_4 will stop at slot 4, k_2 is free to be located at slot 5, 6, 7, 0, ... If k_2 is placed at slot 5 as in the example, k_7 can be placed at 6, though its preferred slot is 4.

Another example in Fig. 10 shows two failures. The first is a direct failure: k_0 and k_2 share the same digest and the same preferred slot, and putting anyone before will shadow the other. Hence their collision is non-resolvable. We use a small table to store all non-resolvable collisions as discussed in Sec 5.5. In this example, both k_0 and k_2 should be relocated to the overflow. The second failure is caused by two collision groups: $h_d(k_4) = h_d(k_7) =$ `0xFF` and $h_d(k_5) = h_d(k_6) =$ `0x12`. k_4 should be placed at slot 5 to avoid collision with k_7, while k_5 must also be placed at slot 5 to avoid shadowing k_6. As a result, one of k_4 and k_5 should be moved to the overflow table. Note that unlike

[6]This modification is *free* because modern hash functions generate 64-bit or 128-bit hash results, and truncated anyway. Practical systems larger than 2^{64} always shard the data for purposes of scalability, performance, and maintainability.

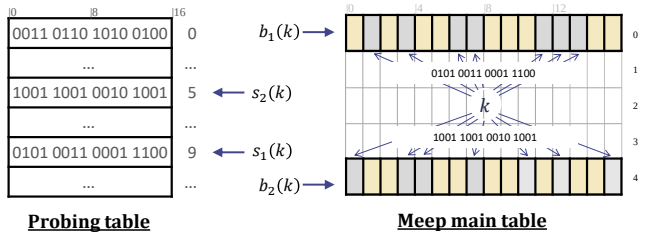


Fig. 11. Randomized probing and key shadows

the same-ground collision, only one should be relocated to resolve this collision. From the examples we gain some sense that slot rearrangement is non-trivial in some buckets. In general, it requires a graph search as covered in Sec. 5.6.3.

5.4 Type 2: remote collisions

Recall Fig. 8, bucket 1 contains a digest 26, colliding with the weak key k_1, which will first match its first-choice bucket, which is bucket 1. Digest 26 at bucket 1 slot 1 is a remote collision of k_1.

However, the current full-bucket probing methodology provides no flexibility, and the colliding digest in the first-choice bucket always shadows the key in the second-choice bucket. Recall the stickiness of (2,16)-Cuckoo hash is $R_s = 85\%$. Then we can estimate the portion of relocated keys due to remote collision as $R_r = (1 - R_s)(2^{-l_d} \cdot n_s)$. For $l_d = 8, n_s = 16$, we have $R_r = 0.94\%$. In the above example, 0.94% of keys like k' should be relocated to overflow to avoid shadowing the colliding remote weak keys like k. Interestingly, we cannot relocate the weak key k to overflow as in that way, k' still collides with k during lookup of k.

To further reduce R_r, we explore the flexibility in the probing algorithm. Revisit how we resolve the local collisions: each key starts matching from its preferred slot and stops at the first match or when the n_s slots have all been visited. However, we make another observation: **(O4) a local collision group usually resolves in a few search steps**, and each key can have a much smaller probing range than 16 slots, without incurring more failures in the local collision resolution.

We first try to limit the number of probed slots from 16 to 7 to reduce the remote collisions without noticeable impact on the local collision rate. We call the size of the probing range **"probing size"** p. For linear probing of probing size 7, it returns the list $\langle i_s,\ i_s+1,\ \cdots,\ i_s+6\rangle$ (in terms of $\bmod n_s$). We show the best probing size $p = 7$ in § 6.4. Given $p = 7$, now we have $R_r = (1 - R_s)(2^{-l_d} \cdot p) = 0.41\%$. We define a function `ProbingRange(·)` to denote a certain slot probing style. It takes a preferred slot i_s as input and returns the slot indexes to probe for this key.

Randomized probing style. To resolve key clustering in a bucket and to be compatible to SSE instructions, we introduce a randomized probing style. We generalize the idea of "preferred slot" as the index into a small "probing table" R of predefined probing ranges, and we call it "probing range selector". We make R a static 16x16 bitmap, and each row is randomly generated and manually adjusted to have exactly seven 1's. Then the probing range of key k with preferred slot i_s will be the slots whose corresponding bit is 1 in the i_s−th row of R.

Shadows of a key. For a key k and one of its alternative bucket, we call all slots in the probing range of k its **"shadows"**. For probing size 7, each key has 7 shadows in each of its two alternate buckets. We show an example of randomized probing table and key shadows in Fig. 11. For key k, its two alternative buckets are 0 and 4, and its probing range selectors are 9 and 5, respectively. Its shadows in buckets 0 and 4 are shown as the arrows and shades in the figure.

Based on the definition of key shadows, a remote collision can be interpreted as: there happen to be a colliding digest in any of a weak key's shadows at its first-choice bucket. Resolving this remote collision is to relocate that colliding digest to another slot which is within its shadow while do not overlap with any other some-digest key's shadows in this bucket.

5.5 Type 3: overflow collisions

If a key fails inserting into the template Cuckoo hash table or fails to resolve its local or remote collision, it is relocated to the overflow table. Meep handles the overflow table very differently from other approaches such as Ludo hashing: the overflow table is chained *after* the main table instead of in the front. We call this **zero-overhead overflow**. Ludo and other similar approaches do not support zero-overhead overflow by design because their main table does not reject any lookup.

To bound the tail latency, the overflow table is a (2,16)-Cuckoo hash table, such that the lookup of k is guaranteed to finish after up to 4 bucket loads—2 for the main table and 2 for the overflow. Note with the stickiness 85%, and 0.5% overall overflow rate, we expect about 1.15 on-average memory loads per lookup.

Relocating keys to the overflow table, however, might in turn cause more collisions. Suppose key k is relocated, then the lookup for k should not match any slots in the main table, or the return value for k is wrong. That requires that no colliding digest is in either of its probing ranges of its two alternative buckets, or it is called an **"overflow collision"**. Overflow collisions are solved similarly to remote collisions: move the colliding digests out of the shades of the colliding weak key.

5.6 Resolving collision types 1, 2, 3

5.6.1 Intuition. We identify a common structure of resolving local, remote, and overflow collisions: each key or digest is free to choose any slot in its probing range unless that slot happen to be in the shadow of another same-digest key, regardless it is local or remote. Based on this core insight, these collisions can be resolved together via graph search with specially tailored search space for each key by excluding shadows of colliding keys.

Based on the observations above, resolving the three types of collisions can be decoupled into two separate phases without shrinking the solution space. **1)** First determine the candidate slots for each key, honoring all the above three placement restrictions. **2)** Then each key is free to choose from its candidate slots, and the slot rearrangement is a graph search problem to reconcile the slot placement contentions.

Each key's search space is determined by excluding prohibited positions from its `ProbingRange`. A key k should be prohibited at a slot when some local/remote/overflow key share the same digest and also has this slot in its probing range.

5.6.2 Implementation. To resolve collisions, we propose to use a pending queue of buckets and resolve collisions in each bucket. Local digest-position collisions are easily calculated given the keys in that bucket. For the remote and overflow keys, the colliding digest and positions are tracked globally.

First, as the remote and overflow keys related to a bucket is known after construction Step 2, a two-dimensional array P is maintained so that for each slot at location $\langle i_b,\ i_s\rangle$, $P[i_b][i_s]$ is the set of prohibited digests (**"P-set"**) at that slot. During the bucket copy in Step 2, each key k is checked whether it is a weak key, by checking whether its current bucket i_b is its first-choice bucket $b_1(k)$. If k is weak, its digest d will be added to the P-sets of each slot within its probing range of its *first-choice bucket*, *e.g.*, for linear probing, $P[b_1(k)][s_1(k)], P[b_1(k)][s_1(k)+1 \bmod n_s], \cdots, P[b_1(k)][s_1(k)+6 \bmod n_s]$. P is only used during Meep construction and will be discarded thereafter.

After initializing the P-set P, the pseudocode of the overall collision resolution algorithm is shown as Algorithm 1. The algorithm traverses each bucket that needs rearrangement with the four inputs: the $(2,n_s)$-Cuckoo hash table T_c, Meep main table T_m, Meep overflow table T_o, and the list of P-sets P. For each bucket index i_b, the slots in two buckets $b_c = T_c[i_b], b_m = T_m[i_b]$ are in the same order. The algorithm repeatedly traverses all pending buckets, and tries to find a possible arrangement $\mathcal{A}$ of the slots for each bucket based on the candidate locations of slots. If a correct

Algorithm 1: Collision resolution algorithm core

Input: The (2,n_s)-Cuckoo hash table T_c
Input: The Meep main table T_m
Input: The Meep overflow table T_o
Input: The sets of prohibited digests P
Outcome: Local, remote, overflow collisions are resolved in T_m

```
Q ← a queue of {0, 1, 2, ··· , ⌈N/n_s⌉ − 1 }
while Q is not empty do
    i_b ← Q remove first
    b_c, b_m ← T_c[i_b], T_m[i_b]
    // A map from slots to their candidate locations
    L ← CandidateLocations(b_c, P[i_b])
    // Locations of each slot after rearrangement
    A ← GraphSearch(L)
    if A is not ⊥ then
        Rearrange b_c, b_m following A
        continue
    ⟨k, v⟩ ← the key-value with the samllest search space
    Relocate ⟨k, v⟩ from T_c to T_o
    d, ⟨b_1, s_1⟩, ⟨b_2, s_2⟩ ← h_d(k), h_1(k), h_2(k)
    for n = 1, 2 do
        for s ∈ ProbingRange(s_n) do
            Add d to P[b_n][s]
    Add b_1 and b_2 to Q
```

arrangement $\mathcal{A}$ is found, slots in b_c, b_m are rearranged per $\mathcal{A}$. Rearranging b_c is necessary because bucket i_b might need to retry later.

If a feasible arrangement $\mathcal{A}$ is not found, then some keys should be relocated to the overflow table T_o. We propose to relocate *the least flexible key* for two reasons. First, some keys have no candidate locations, *e.g.*, two keys involved in a triple-collision, a key shadowed by a weak key that has the same digest and preferred slot, a key shadowed partially by a weak key and partially by the local collision, etc. For these keys, the "least flexible" rule will naturally relocate these non-feasible keys as the first attempt. Second, even if all keys have candidate locations, the heuristic aligns with the intuition that removing a less flexible key will more likely reconcile the whole bucket. After relocating k to overflow table T_o, its digest $h_d(k)$ should be prohibited within its two probing ranges of both the buckets $b_1(k)$ and $b_2(k)$. And as the P-sets in $b_1(k)$ and $b_2(k)$ are updated, these buckets should be added to queue Q, to retry their slot arrangement.

For bucket i_b, procedure `CandidateLocations` takes two arguments: all keys in this bucket, stored in b_c, and all P-sets in bucket i_b, maintained in $P[i_b]$. The output is a list of candidate locations for each key in bucket i_b. To calculate the candidate locations for a key k, first find all the local colliding keys of k, and then k's candidate locations form a range starting from its preferred slot i_s, and ending at $min(i'_s, i_s + 6)$, where $i'_s \geqslant i_s$ is the nearest preferred slot among all colliding keys. After that, the candidate locations are further tailored by removing the locations where $h_d(k)$ is prohibited.

Given L the candidate locations of all keys in bucket i_b, procedure `GraphSearch` outputs any valid arrangement of all the keys, or $\perp$ if failed to find one. Intuitively, this problem is a variation of the classical eight queens puzzle, and it requires a full search of all possible placements via backtracking to find a solution, or conclude the problem is unsolvable.

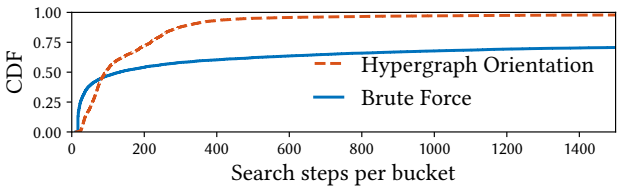


Fig. 12. Search steps of GraphSearch algorithms

5.6.3 Graph search algorithm. For the above GraphSearch, the full backtrack search is prohibitively expensive due to its large search space — up to 7^{16} search steps per bucket. We propose a heuristic to greatly reduce search steps for most cases and works well in practice. This heuristic arises from another important observation **(O5) the problem of GraphSearch is an irregular version of the hypergraph orientation problem** [8, 19]. In the original hypergraph orientation problem, a hypergraph is a pair $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of hyper edges, and each edge is associated to $\mathcal{H} \geqslant 2$ vertices. A $\mathcal{K}$-orientation of the hypergraph is to assign each hyper edge to one of its associated vertex, and each vertex is assigned with $\leqslant \mathcal{K}$ hyper edges. This problem is intensively researched in theory and practice, and the $(\mathcal{H}, \mathcal{K})$-Cuckoo hash table is the prevailing solution [22, 25, 30, 39].

The procedure GraphSearch can also be interpreted as a hypergraph orientation: $\mathcal{V}$ is the set of n_s locations in the bucket, and all the candidate locations of a key is a hyper edge in $\mathcal{E}$, and arranging the keys in the bucket is a 1-orientation of the hypergraph. One distinction, however, is that each hyper edge in our problem has a different number of vertex association $\mathcal{H}' \in \{0, 1, \cdots, p = 7\}$. Keeping this difference in mind, we still solve the problem with Cuckoo insertion. We make a slight modification to the $(\mathcal{H}', 1)$-Cuckoo hash table, such that the table has exactly n_s buckets, and each bucket has 1 slot, and each key's candidate buckets are determined, instead of hash functions, by its candidate locations depicted in L. Cuckoo avoids combinatorial explosion by turning a global assignment problem into a local, random walk with high success probability under sparse constraints. The pressures on hot positions are naturally distributed via Cuckoo-styled kicking.

To compare the efficiency of the brute force search and the Cuckoo hashing based hypergraph orientation, for each algorithm, we construct 10 Meep tables of 1M keys and record the number of search steps for each bucket, and show the cumulative distribution functions (CDFs) as in Fig. 12. The tail of the brute force algorithm turns out too long, and we truncate the number of steps at 1500. The average number of search steps of the brute force search is 2558791, which is 3358 times slower than the hypergraph orientation, which is 762 steps on average.

5.7 Type 4: empty collisions

After the construction above, the Meep main table contains a small portion R_o of unoccupied slots, or empty slots. An empty slot still holds a digest, and might shadow some local or remote keys. A naive solution is to introduce a bitmap per bucket to indicate whether a slot is empty. That introduces an additional 1-bit overhead per key. Another solution is to reserve a single special value `0xFF` as an indicator. That slightly degrades the alien key filter due to the occupation of 1/256 digest space. Further, originally the digest hash $h_d(\cdot)$ is a simple truncation of the last l_d bits of a long hash result, but now we should use more expensive computations *i.e.*, $\bmod (2^{l_d} - 1)$, incurring extra computation cost.

Instead of all the tradeoffs above, we propose a zero-overhead algorithm to assign a proper digest to each empty slot such that the placeholder digest does not shadow any local/remote/overflow key. With the P-set tracking the remote and overflow collisions and the probing ranges of local keys known, it is straightforward to know the digests that will shadow other keys in each slot. After

Algorithm 2: Fully SSE-accelerated `query`(k)

Input: Main table T_m, overflow table T_o, probing table R
Input: The query key k
Output: The value v associated with key k

$d, b_1, b_2, s_1, s_2 \leftarrow H(k)$
for $n = 1, 2$ **do**
    `// Batch digest matching via single SSE instruction`
    $I \leftarrow$ 16-bitmap, i-th bit set *iff.* d **equals** the i-th digest of b_n
    `// Single machine instruction, 16-bit bitwise AND`
    $I' \leftarrow I \,\&\, R[s_n]$
    `// Single machine instruction, leading zeros count`
    $i_m \leftarrow$ number of leading zeros in I'
    if $i_m < n_s$ **then**
        return i_m-th value
return query result of k on T_o, or $\perp$ if still not found

excluding the few prohibited digests, each empty slot randomly choose a digest from the remaining digest space. In this way, empty slots do not collide with any other slots without increasing overflow rate, memory cost, or computation overhead.

5.8 Meep ultrafast lookup

With all collisions resolved, we revisit the Meep lookup procedure based on an observation: **(O6) after the necessary collision resolution, there is no colliding digest in the key shadows**. That means when looking up a key, if multiple slots match the digest, the first is the actual match.

We already use the SSE instruction to match 16 digests in one machine instruction, and the rest steps are to 1) limit the matches in the key's probing range, and 2) then pick the first match. And either step can be finished at 1 machine instruction. Further, we split the 64-bit hash function result so that d, b_1, b_2, s_1, s_2 can be derived via a single hash computation. The fully optimized `query` is illustrated in Algorithm 2.

Moreover, table R is small and static, and 32 bytes in total. We pad and align it to 64 bytes so that it takes an entire cache line in modern CPUs, eliminating all possible false sharing on multi-cores. Constantly being referenced, this cache line is expected to always be located at the L1 data cache of all cores performing lookup.

6 Optimizations and design choices

In this section, we investigate the best combination of design parameters. The goals are high lookup speed and low memory cost. We show these two goals are not contradictory and can be achieved together with a carefully derived best configuration.

6.1 Number of alternative buckets n_b

Tuning n_b is not recommended. Setting n_b to 1 will have a much lower load factor and thus a much higher overflow rate, making the overall space cost worse than a probing-based hash table. Having a larger $n_b \geqslant 3$ will cause much more remote collisions, and the increased number of overflow keys will in turn collide with more keys, generating more overflow. Besides, a large n_b will degrade the lookup tail latency by adding more number of bucket loads.

6.2 Number of slots in a bucket n_s

The number of slots in a bucket n_s is default to 16, to best align with the SSE batch digest matching.

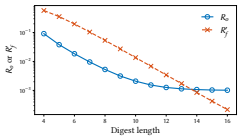


Fig. 13. Overflow and false positive rates with different l_d

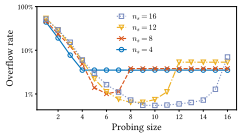


Fig. 14. Overflow rates for different p and n_s

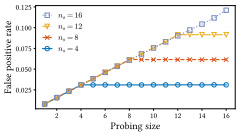


Fig. 15. False positive rates for different p and n_s

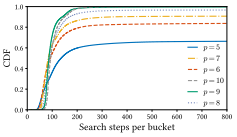


Fig. 16. Search steps per bucket for different p

The core design of Meep, however, does not rely on $n_s = 16$. Tuning n_s makes sense for several application scenarios, for example, when Meep is used as a lookup algorithm and the value length is 4 bytes, then $n_s = 12$ will make sure a bucket is less than 64 bytes, fitting into a single cache line. Tuning n_s, however, requires additional attention due to its deep coupling with probing size p. With a smaller n_s, for example, the best probing size p might be less than 7, so that the construction time is shorter, and the false positive rate R'_f will also be smaller.

For scenarios that fast lookup is the primary goal, $n_s = 16$ is the only best choice. n_s smaller than 16 looses the acceleration of vectorized machine instruction, and many loops are needed for one lookup. While $n_s = 32$ is also compatible to vectorized acceleration like AVX256 to match 32 digests in one instruction, compared with $n_s = 16$, $n_s = 32$ costs twice of the CPU-memory bandwidth and CPU-internal computation for a single lookup, with little improvement in increasing the load factor of Cuckoo template (already 99.95% for $n_s = 16$).

6.3 Digest length l_d

Longer digests enable a lower false positive rate and a lower overflow rate. For each digest length $l_d = 4, 5, \cdots, 16$, we construct 10 Meep tables of 1M keys and record the overflow rate R_o and actual false positive rate R'_f, as shown in Fig. 13. R'_f drops exponentially with the digest length, while R_o has a lower bound 0.1%. The lower bound is due to the local collisions because all keys are distinguished via their different digests.

If an alien key is to match with any slot in the Meep main table, there should be at least one digest match in its two probing ranges. Hence, the false positive rate of Meep is $R'_{f_0} = 1 - (1 - 2^{-l_d})^{2p} = 5.33\%$, where $l_d = 8$ and $p = 7$. Intuitively, changing the digest length l_d will help tune the false positive rate R'_f to meet the target R_f, when $R_f \neq 5.33\%$, while it is subtle in practice.

To enable SSE-accelerated ultrafast lookup, l_d must be at least 8 bits. When $l_d > 8$, we recommend splitting the digest into two parts: the first part consists of the lower 8 bits, used for SSE-based matching, and all 16 such 8-bit digests are stored contiguously. The remaining $l_d - 8$ bits of each digest are placed together with its corresponding value. As a result, $R'_f = 2^{8-l_d} \cdot R'_{f_0}$, instead of $1 - (1 - 2^{-l_d})^{2p}$. Also note that, longer digests also enable a lower overflow rate as shown in Fig. 13.

6.4 Probing size p

In discussions above, we assume the size of each key's probing range p is 7. The probing size is a critical design parameter which directly shapes the algorithm's key performance metrics. Also, p is deeply coupled with n_s. For each probing size $p = 1, \cdots, 16$ and $n_s = 4, 8, 12, 16$, we construct 10 Meep tables of 1M keys and record the overflow rate R_o, actual false positive rate R'_f, and number of search steps per bucket.

The overflow rates of different configurations are shown in Fig. 14 in logarithmic scale. For small probing sizes, R_o is high due to limited flexibility to rearrange slots in `GraphSearch`. We observe a rise of R_o when p grows from $n_s - 1$ to n_s. The reason is when $p = n_s - 1$, a remote or overflow key of a bucket will shadow all but one slot, and a colliding key can be placed at that non-shadowed slot. However, when $p = n_s$, a remote key or an overflow key of the bucket will shadow all slots,

and any colliding key should be relocated to the overflow table, causing more shadows in turn. Across all n_s values, $p = 7$ shows a consistent low overflow rate.

R'_f of different configurations are shown in Fig. 15, and we use quadratic probing to show a potential drawback of this very probing style. In theory, $R'_f = 1 - (1 - 2^{-l_d})^{2p}$, and the experimental results fit the theory well.

The CDF of number of search steps for $n_s = 16$ is shown in Fig. 16. With a larger probing size, all keys have more flexibility to resolve local collisions, and a bucket is usually resolved in 150 search steps.

6.5 Meep best practice

With aforementioned design on Meep construction, we achieve perfect collision resolution and low overall overflow rate, and arrive at the best configuration of Meep hashing for both fast lookup and low memory footprint: Cuckoo style `GraphSearch`, 16x16 tabled probing with cache line padding, $n_b = 2, n_s = 16, p = 7, l_d \geqslant 8$. Following the construction algorithm and the recommended configurations, we build ten Meep tables for sizes 1K, 10K, 100K, 1M, and measure that the overflow rate R_o and the false positive rate R'_f are consistently around 0.53% and 5.33%, respectively.

When target false positive rate is more strict than 5.33%, we can tune l_d and p per Fig. 13 and discussions above.

7 Evaluations and Case Study

In this section we conduct two sets of experiments, each comparing Meep with a group of state-of-the-art techniques. The first set of experiments focus on pure MPHs for key-index lookup, while the second set put attention on key-value lookup algorithms. All experiments runs on two AMD EPYC 9554 64-core CPUs at 3.1GHz, with 192GB 4800MT/s DDR5 memory, 2 hardware thread per core, 32 KiB L1 data (L1d) cache per core, 1 MiB L2 cache per core, and 512 MB L3 cache every 8 cores. We implement the Meep prototype in C++ following the best configuration – Cuckoo style `GraphSearch`, 16x16 tabled probing with cache line padding, $n_b = 2, n_s = 16, p = 7, l_d = 8$. The source code of Meep is available for results reproducing [1].

We evaluate the performance of different designs by tracing following metrics:

(1) **Lookup efficiency** on both single thread and multiple threads.
(2) **Construction time** of different key set sizes.
(3) **Memory cost** to characterizes the space efficiency.

We use 64-bit keys and 8-bit values for all following evaluations. We generate keys and values randomly with a seeded random generator to guarantee reproducibility. Each data point shown in the figures is the average of 3 independent experimental runs with different random seeds. For lookup throughput evaluations, the request workloads are in two types: in the uniform distribution and Zipfian distribution. For the uniform distribution, all items are requested with an equal probability. For the Zipfian distribution, items are requested with biased probabilities, which better simulates the workload in most practical systems.

7.1 Evaluation on MPH and MHT algorithms

In the first set of experiments, we compare Meep with other minimal perfect hashing (MPH) algorithms and minimal hash table (MHT) algorithms.

The related MPH algorithms are PThash [34], Ludo [36] and RecSplit [14]. PTHash D-D is based on dictionary-based encoding and PTHash C-C is based on compact fixed-length encoding. Both schemes support $O(1)$ lookup, but C-C prioritizes speed, while D-D balances space and speed. We pick (D-D, $\alpha = 0.94, c = 7$) and (C-C, $\alpha = 0.99, c = 7$) which are recommended by the original paper. For RecSplit, b is the average bucket size when splitting and ℓ is the maximum leaf size to use

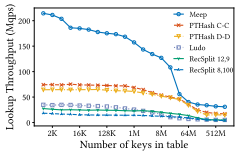


Fig. 17. MPH key-index lookup throughput (uniform)

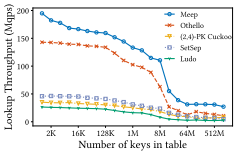


Fig. 18. MHT key-value lookup throughput (uniform)

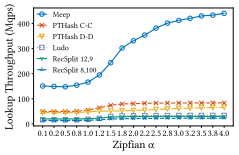


Fig. 19. MPH key-index Zipf lookup throughput (8M keys)

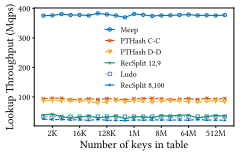


Fig. 20. MPH key-index lookup throughput (Zipf $\alpha = 4$)

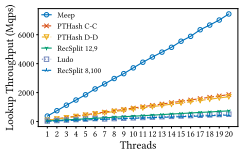


Fig. 21. MPH multi-thread lookup (Zipf $\alpha = 4$, 8M keys)

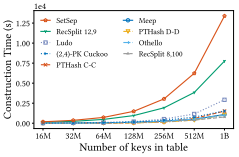


Fig. 22. MPH and MHT construction time

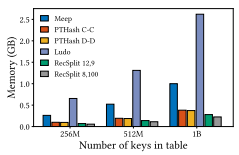


Fig. 23. MPH memory footprint

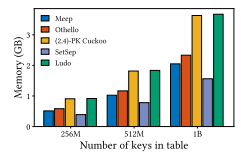


Fig. 24. MHT memory footprint (64 bit keys, 8 bit values)

brute-force collision resolution. We also pick the recommended configurations ($\ell = 8,\ b = 100$) and ($\ell = 12,\ b = 9$) for evaluation.

The related MHT algorithms are Othello [44], SetSep [17], (2,4)-PK Cuckoo [37], and Ludo [36]. Note that all MHT structures support key-value lookup, either via encoded mapping as in Othello and SetSep, or via table-embedded values as in Meep, PK-Cuckoo and Ludo. Like Meep, Ludo is both MPH and MHT, while unlike Meep, Ludo has a 1.2% bubble rate. PK-Cuckoo lacks the ability to perfectly distinguish valid keys, and hence not an MPH.

Lookup efficiency. Fig. 17 and Fig. 18 show the lookup throughput of MPH and MHT algorithms on different numbers of keys in one thread, under uniform key distribution. In every single run, we preform $2^{25} \approx 33$ million lookup operations on each MPH, and collect the overall throughput in terms of million queries per second (Mqps). Meep is the fastest among all MPHs and MHTs tested.

MPH, Meep is consistently 50%-200% faster than PTHash (C-C), the quickest among the other MPHs, under any size of key set. On small sets of 1K to 4K keys, Meep is ultrafast because it fits in 32 KiB L1d cache, and the computation only involves a single hash function call and three machine instructions to match the digest. As a result, Meep stably exhibits more than 200 Mqps single thread throughput, while the throughput of PTHash (C-C) is at 73 Mqps, around 30% of Meep's. On large sets of 512 million or 1 billion keys, Meep still leads all MPHs at a single thread throughput around 30 Mqps, while PTHash (C-C) is 17 Mqps. Uniform distribution of the query keys maximize the cache miss probability and hence emphasis on the memory load costs during lookup. Though Meep takes 1.15 memory loads compared to 1 of PTHash, Meep is still faster than PTHash because Meep takes much less computation, and the computation of PTHash consistently takes longer than 0.15 memory load. Among MHTs, Meep is still the fastest structure. Meep performs around 170 Mqps and 27 Mqps on small tables and large tables respectively. Othello takes consistently 2 memory loads to lookup a key, and hence Othello provides throughput around 143 Mqps and 10 Mqps, about 30%-70% of Meep's. Other MHTs are at least 3 times slower than Meep. PK-Cuckoo needs to resolve collisions and to compare the digests sequentially in the bucket, while Ludo needs 5 to 6 memory loads to look up a key in the main table due to its overflow table and bucket locator in front.

Lookup under Zipfian distribution. We fix the key set size at 8 million, and change the Zipfian parameter α from 0 to 4 in Fig. 19. All MPHs runs faster at higher α as the hot-cold distribution is more skewed. Meep is >3x faster than other algorithms and also grows faster as α grows.

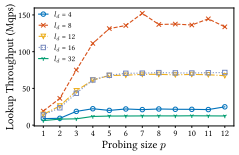


Fig. 25. Meep lookup throughput varying l_d and p

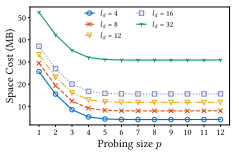


Fig. 26. Meep space cost varying l_d and p

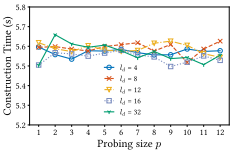


Fig. 27. Meep construction time varying l_d and p

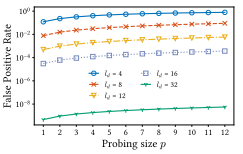


Fig. 28. Meep false positive rate varying l_d and p

Algorithm	GoogleBooks 1-grams			GoogleBooks 2-grams			UbiCrawler URLs		
	lookup (Mqps)	constr. (secs)	space (MiB)	lookup (Mqps)	constr. (secs)	space (MiB)	lookup (Mqps)	constr. (secs)	space (MiB)
Meep	**12.13**	**32.99**	24.23	**6.53**	231.87	92.91	**4.99**	153.20	39.25
PTHash (D-D)	6.81	61.20	9.25	3.71	**222.54**	36.13	2.75	**109.69**	14.62
RecSplit ($\ell = 12,\ b = 9$)	4.39	178.58	**6.49**	1.72	689.84	**24.87**	1.72	295.92	**10.51**

Table 3. Construction time, memory cost, and lookup throughput of MPHs.

Then we set the Zipfian parameter $\alpha = 4$ in Fig. 20 and vary the key set size from 1 thousand to 1 billion. As a small set of hot keys are looked up for most of the time, and the L1d cache can hold all the hot keys, all MPHs show stable lookup throughput across all key set sizes, similar as their throughput under the uniform distribution for small key set sizes when keys fits into the L1d cache. As a result, Meep lookup is consistently around 400 Mqps for all sizes, while others are less than 100 Mqps.

Fig. 21 shows the lookup throughput with different concurrent lookup thread numbers on 8 million of keys under Zipfian $\alpha = 4$. All MPHs' lookup performance scales linearly with the number of threads, meaning all their performance bottleneck is at computation instead of memory bandwidth. Meep is consistently >3x faster than all other structures under all concurrency due to fast computation. And with 20 parallel threads, Meep shows 7433 Mqps, or 7.4 Bqps.

Construction time. Fig. 22 shows the construction time needed by each structure. Sizes smaller than 16M are omitted. Meep is either faster or comparable to others. For 1 billion keys, Meep only needs 18 minutes to construct, while PTHash (D-D) is the fastest at 16 minutes, and all other MPHs takes longer time than Meep to construct, especially, RecSplit (12, 9) spends more than 2 hours and Setsep takes more than 3.5 hours. SetSep and RecSplit are slow due to their large search space in the brute-force searching. Meep also needs to resolve three types of collisions via search, but the Cuckoo-styled hyper graph orientation strategy reduces the search space greatly as in Fig. 12.

Memory cost Fig. 23 and Fig. 24 show the memory cost of all MPHs and MHTs on different key set sizes, respectively. For 1 billion keys, all MPH designs except Ludo only use less than 1GB memory to hold the entire structure, unlikely to become the system bottleneck. Meep has a high memory cost compared to PTHash, RecSplit, and SetSep. For applications where memory budget is super tight, RecSplit (8, 100) might be a competitive candidate though its construction is 6 times slower and lookup is 7-10 times slower than Meep.

7.2 Evaluation on Meep configurations

Some applications require more strict alien key filter than 5% provided by the Meep reference setting. Other applications might require shorter digests to save space. We construct a series of Meep tables under different l_d and p to show how the lookup performance (Fig. 25), space cost (Fig. 26), construction time (Fig. 27), and false positive rate (Fig. 28) changes under different settings.

$l_d = 8$ is the fastest because shorter digest, *e.g.*, $l_d = 4$, requires bitwise operations to extract the digest, while longer digest, *e.g.*, $l_d = 12$ or 32, requires more digest comparisons in addition to

possible bitwise operations. $ld = 8,\ p = 7$ is the most compact among all $ld = 8$ variations because it achieves the smallest overflow by balancing local and remote collisions. Inefficient probing when $p < 4$ generates too many overflows. The construction time is similar across all variations. The false positive rates align with § 6.3. $l_d = 8,\ p = 7$ is the best balance

7.3 Evaluation on real-world datasets

We evaluate the performance of MPHs on three sets of real-world datasets, representative for database and machine learning use cases. Specifically, we employ the English GoogleBook version 2 for its 1-grams and 2-grams as keys [7]. They contains 24.4 million and 93.4 million keys, the average length of keys being 10.3 and 17.9 bytes, and hence, they take 276 MiB and 1.77 GiB to store the full keys, respectively. We also evaluate MPHs on longer URL keys, collected by UbiCrawler [7], containing 39.5 million URLs, average length being 71.4 bytes, taking 2.86 GiB memory to store.

Table 3 shows the lookup throughput, construction time, and space cost of Meep, PTHash, and RecSplit. Meep is about x2 faster than PTHash and x3 faster than RecSplit on all three datasets. PTHash constructs faster when keys are long because each key is hashed multiple times during Meep construction, and this might be improved in the future. All algorithms greatly reduce the space cost, and the space costs are linear to the number of keys, instead of dataset size.

7.4 Case study

We have deployed Meep hashing in several industrial-scale production systems at Google since 2022 to support factorized retrieval like Search Ads pCTR system [2], whose total memory consumption is several hundreds of TiB. These systems are highly suitable for Meep because they struggle on memory and computation costs of lookup, while they can tolerate the 5.33% false positive rate shipped with Meep. The tolerance is because on one hand, the query keys are from internal data pipelines and hence rarely alien, and on the other hand, the downstream pipeline can identify and reject alien keys.

In different subsystems, we saw around 23% reduction in p99 latency and 17%-59% reduction in memory cost, compared with the naive SwissTable-based hash table index. Note the memory cost reduction is measured at the entire system level. The net reduction in lookup index space cost is around 95% as the hash table keys are around 20 bytes and is reduced to 1 byte digest.

8 Conclusion

Meep hashing is a novel minimal perfect hashing algorithm with unparalleled ultrafast lookup speed and highly compact memory footprint, supporting both key-index lookup as other MPHs and key-value lookup due to its table-based layout. Its built-in alien key filter is essential for robust large-scale lookup systems. We present Meep's core design via resolving 4 types of collisions and its fully optimized lookup. We explore a series of possible design choices and arrive at a single best setting with a configurable target false positive rate. Experimental results show that Meep hashing is consistently 50% to 200% faster than the other MPHs and 100% faster than the other MHTs. Meep achieves more than 7 billion queries per second under Zipfian query key distribution on a single machine with 20 parallel lookup threads.

Acknowledgments

We would like to thank Yuanxi Wang and the anonymous reviewers for their suggestions and comments which inspired our revisions of the manuscript.

References

[1] [n. d.]. Implementation of Meep Hashing in C++. https://anonymous.4open.science/r/meep-hashing.

[7] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

[2] Rohan Anil, Sandra Gadanho, Da Huang, Nijith Jacob, Zhuoshu Li, Dong Lin, Todd Phillips, Cristina Pop, Kevin Regan, Gil I Shamir, et al. 2022. On the factory floor: ML engineering for industrial-scale ads recommendation models. *arXiv preprint arXiv:2209.05310* (2022).

[3] Djamal Belazzougui and Fabiano C. Botelho. 2009. Hash, displace, and compress. In *Proc. of Algorithms-ESA*.

[4] Michael A Bender, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2011. Don't thrash: how to cache your hash on flash. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*.

[5] Sam Benzaquen, Alkis Evlogimenos, Matt Kulukundis, and Roman Perepelitsa. 2017. Swiss Tables Design Notes. https://abseil.io/about/design/swisstables#swiss-tables-design-notes Accessed 2022-12-13.

[6] B. H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726.

[8] Julie Anne Cain, Peter Sanders, and Nick Wormald. 2007. The Random Graph Threshold for k-orientiability and a Fast Algorithm for Optimal Multiple-Choice Allocation. In *Proc. of ACM-SIAM SODA*.

[9] Denis Charles and Kumar Chellapilla. 2008. Bloomier Filters: A Second Look. In *Proc. of ESA*.

[10] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA*. 30–39.

[11] Cloudflare 2022. Cloudflare: a global network built for the cloud. http://cloudflare.com/ Accessed 2022-12-13.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (4th ed.). MIT press, Cambridge, Massachusetts.

[13] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*.

[14] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. 2020. RecSplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 175–185.

[15] Bin Fan, Dave Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*.

[16] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM.

[17] Bin Fan, Dong Zhou, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. When cycles are cheap, some tables can be huge. In *Proc. of USENIX HotOS*.

[18] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking* (2000).

[19] Daniel Fernholz and Vijaya Ramachandran. 2007. The k-orientability Thresholds for $G_{n,p}$. In *Proc. of ACM/SIAM SODA*.

[20] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. 1992. Practical minimal perfect hash functions for large databases. *Commun. ACM* (1992).

[21] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2016. Fast Scalable Construction of (Minimal Perfect Hash) Functions. In *Proceedings of the International Symposium on Experimental Algorithms*.

[22] Manu Goyal, Bin Fan, Xiaozhou Li, David G. Anderson, and Michael Kaminsky. 2022. libcuckoo: A high-performance, concurrent hash table. https://github.com/efficient/libcuckoo Accessed 2022-12-13.

[23] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *International Symposium on Distributed Computing*. Springer, 350–364.

[24] Intel 2022. Intel Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_cmpeq_epi8&techs=SSE2 Version 3.6.3.

[25] X. Li, D. Andersen, M. Kaminsky, and M. J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys*.

[26] Bruce M. Maggs and Ramesh K. Sitaraman. 2015. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review* (2015).

[27] Rui Mao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*.

[28] Martn Dietzfelbinger and Christoph Weidling. 2007. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science* (2007).

[29] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of*

the ACM SIGOPS 28th Symposium on Operating Systems Principles. 243–262.
[30] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* (2004).
[31] Prashant Pandey, Alex Conway, Joe Durie, Michael A Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*. 1386–1399.
[32] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. *Proc. of ACM SIGCOMM*.
[33] Giulio Ermanno Pibiri, Yoshihiro Shibuya, and Antoine Limasset. 2023. Locality-preserving minimal perfect hashing of k-mers. *Bioinformatics* 39, Supplement-1 (06 2023), i534–i543.
[34] Giulio Ermanno Pibiri and Roberto Trani. 2021. PTHash: Revisiting FCH Minimal Perfect Hashing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*.
[35] B. Schlinker et al. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proc. of ACM SIGCOMM*.
[36] Shouqian Shi and Chen Qian. 2020. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2020).
[37] S. Shi, C. Qian, and M. Wang. 2019. Re-designing Compact-structure based Forwarding for Programmable Networks. In *Proc. of IEEE ICNP*.
[38] Shouqian Shi, Ye Yu, Minghao Xie, Xin Li, Xiaozhou Li, Ying Zhang, and Chen Qian. 2020. Concury: A Fast and Light-weight Software Cloud Load Balancer. In *Proceedings of the eleventh ACM symposium on cloud computing (SOCC)*.
[39] TensorFlow Authors. [n. d.].
[40] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment* (2019).
[41] Udi Wieder. 2017. *Hashing, Load Balancing and Multiple Choice*. Now Publishers.
[42] Tong Yang, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. 2019. Coloring Embedder: a Memory Efficient Data Structure for Answering Multi-set Query. In *Proc. of IEEE ICDE*.
[43] Kok-Kiong Yap et al. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proc. of ACM SIGCOMM*.
[44] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. 2018. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking* (2018).
[45] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. 2015. Scaling Up Clustered Network Appliances with ScaleBricks. In *SIGCOMM*.
[46] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of ACM CoNEXT*.

Received October 2025; revised February 2026; accepted February 2026