One Off-chip Memory Access Hash Table

Anonymous Authors

Abstract

Hash table is a classic data structure. It is widely used in various fields of computer science. Its worst-case query performance is critical for many applications. Among the literature, perfect hash tables can achieve one memory access per query in the worst case. However, they do not support fast incremental updates. In this paper, we propose the One-Memory Hash table, which is deployed in the increasingly widespread on-chip/off-chip hierarchical memories (*e.g.*, on-chip memory/off-chip memory in FPGA, GPU/CPU). Our hash table needs only one off-chip memory access per query in the worst case, supports fast incremental updates, has good extensibility and guarantees no update failure. Our key technique is to build *exclusive fingerprints* in the on-chip memory to guide query for the Key-Value pairs stored in the off-chip memory. We implement the One-Memory Hash on GPU/CPU and FPGA platforms, and conduct head-to-head comparisons with 9 prior hash schemes on 3 real-world and 1 synthetic datasets. Experimental results show that the query performance of One-Memory Hash significantly outperforms all prior works. All related source codes have been released anonymously at Github [9].

1 Introduction

1.1 Background and Motivation

Hash tables are one of the most classic data structures. Thanks to their average $O(1)$ time complexity for insertion, query and deletion, hash tables have been applied to a variety of areas, such as IP lookups [19, 49, 59], packet classification [28, 52], load balancing [11, 13, 42, 57], intrusion detection [45, 50], TCP/IP state management [60], MAC table lookup [62, 65], key-value stores [15, 25, 26, 39], in-memory databases [14, 17, 22, 29], NLP [27, 48, 56], social network [40], and more [31].

Many real-world workloads that hit hash tables are *read-heavy* [12, 47]: most of the operations are queries. The main shortcoming of hash tables is that they cannot guarantee $O(1)$ time complexity for queries in the worst case. Many algorithms (details in the survey [31]) achieve good average query performance by using certain collision resolution methods, *e.g.*, storing key-value pairs (KV pairs) that are mapped to the same bucket in a linked list. However, their worst-case query time is still of $O(n)$ complexity, which could be intolerable in practice.

In many applications, it is critical for the worst-case query time to be well-bounded. Here we give three use cases: 1) Hash tables are widely used in the routers [51], such as IP lookups [61], packet classification [55], per-flow measurement [37], and so on. However, the worst case performance cannot be guaranteed due to hash collisions, and thus the line rate cannot always be caught up. Consequently, high end routers are forced to use hardware (such as TCAM, BCAM) to store key-value pairs, leading to high cost and high power consumption. Obviously, if the worst-case query time is constant and short, expensive hardware can be replaced, reducing cost and power consumption. 2) A switch in a data center [62, 65] queries each incoming packet in its MAC table, and if query operations take too much time in the worst case, the buffer of the incoming packet might overflow, causing packet loss. 3) When DDoS attacks happen, the attacked server needs to check each incoming request in the black list which is usually stored as a hash table. If the lookup of the black list is too slow in the worst case, the server will deny services. Therefore, it is critical to design a hash table that guarantees a small bound for the worst-case query time.

Due to the significance of query performance of hash tables, a large number of works have studied how to reduce the number of memory access per query. There are mainly two kinds of solutions. The first kind achieves constant lookup time at the cost of slow update and possibility of update failures. Typical schemes include Cuckoo Hashing [43] and its variants [23, 24, 35, 65], and Perfect hash tables [21]. Cuckoo hashing needs two memory accesses per query in the worst case, and has attracted much attention [36, 44, 53]. Unfortunately, its bounded worst-case complexity comes at the cost of slow updates and the possibility of update failures. To achieve a high load factor (around 95%), some variants of cuckoo hashing [23–25] allow as many as 500 memory accesses per insertion in the worst case. After these 500 memory

accesses, if the insertion still fails, the whole table needs to be reconstructed, which is highly undesirable in many applications. Perfect Hashing [21] achieves one memory access per query through specially designed hash functions, but does not support fast incremental updates.

The second kind of solutions leverages the on-chip/off-chip hierarchical memories, and achieves in average around 1 off-chip memory access per query. Typical schemes include Segmented Hashing [32] and Peacock Hashing [33]. Typically, the access time of on-chip memory like SRAM is compared negligible to that of off-chip memory like DRAM [38]. When the on-chip and off-chip memories are pipelined, the query bottleneck usually lies in the access of the off-chip memory. Therefore, the second kind of solutions focuses on using auxiliary data structures (*e.g.*, fingerprints, Bloom filters [16]) in the on-chip memory to reduce the number of accesses of off-chip memory. Here we show how this kind of solution works by using **an example**, which will be discussed again in the next subsection to illustrate our proposed idea. Suppose we have two sub-tables, each associated with an independent hash function, $h_1(.)$ and $h_2(.)$. Each key-value (KV) pair has one candidate bucket in each of the two sub-tables. For simplicity, we suppose every KV pair can find one empty bucket. To query a key, it needs two memory accesses in the worst case. To minimize the number of memory accesses, for each bucket with a key x, we use a hash function $h^{\mathrm{F}}(.)$ to compute a n-bit value $h^{\mathrm{F}}(x)\&(2^n-1)$, which is used as the *fingerprint* of x. All fingerprints constitute two fingerprint arrays, which are usually very small (*e.g.*, 8 bits per fingerprint) and can be stored in the on-chip memory. The two sub-tables are stored in the off-chip memory. To query a key x, we first compute $h_1(x)$ and $h_2(x)$, and check the two corresponding fingerprints in the on-chip memory. There will be one match in most cases, and two matches in the worst case. Then we check the corresponding matched buckets in the sub-table in off-chip memory. The usage of fingerprints achieves only one off-chip memory access per query in most cases, but the worst-case query does not change –it still n two memory accesses.

In summary, existing works that optimize the number of memory accesses either suffer from slow update and possibility of update failures, or do not improve the worst-case performance. The *goal* of this paper is to design a hash table that needs at most one off-chip memory access per query in the worst case, and supports fast incremental update without update failures.

1.2 Our Proposed Approach

The *key novelty* of One-Memory Hash is that when querying an inserted key, it reports at most one sub-table contains the queried key. This is achieved through our *exclusive fingerprinting* strategy. We continue the **above example** to show how exclusive fingerprinting works. Recall that in the above example, we use only *one hash function* $h^{\mathrm{F}}(x)\&(2^n-1)$ to compute the fingerprint. When querying a key, there could be two matches of the fingerprints in the worst case. Our goal is to achieve **only one fingerprint match**. Towards this goal, we change the method of computing fingerprints. Suppose there is a bucket with three KV pairs mapped to it: $\langle k_1,\ v_1\rangle$, $\langle k_2,\ v_2\rangle$, and $\langle k_3,\ v_3\rangle$, with $\langle k_1,\ v_1\rangle$ arriving first and being stored in it. Instead of using one hash function, we deploy a number of hash functions $h_1^{\mathrm{F}}(.), h_2^{\mathrm{F}}(.), h_3^{\mathrm{F}}(.)\ldots$ to compute a number of fingerprints for k_1. Among these fingerprints, we choose one fingerprint $h_i^{\mathrm{F}}(k_1)$ which satisfies that $h_i^{\mathrm{F}}(k_1) \neq h_i^{\mathrm{F}}(k_2)$ and $h_i^{\mathrm{F}}(k_1) \neq h_i^{\mathrm{F}}(k_3)$. We use this method to compute fingerprints for every key. In this way, it can be guaranteed that when querying any key in the hash table, there will be exactly one fingerprint match, and then we can get the KV pair in the corresponding bucket with only one off-chip memory access. If there is more than one match, it indicates that the queried key is definitely not in the hash table, and we immediately report a query failure without accessing the off-chip memory. Therefore, using our exclusive fingerprinting strategy, there will be at most one[1] off-chip memory access per query in the worst case. We further propose a technique called *instant table extension* to avoid update failures.

Hierarchical implementation: As shown in Figure 1, One-Memory Hash is well suited to the on-chip/off-chip hierarchical memories widely used in modern systems. Specifically, the fingerprint arrays are stored in on-chip memory, such as the BRAM (Block RAM) inside a FPGA (Field Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) chip, or the global memory [2] in a GPU (Graphics Processing Unit). The sub-tables are stored in off-chip memory, such as the DRAM (Dynamic RAM). As aforementioned, when using a pipeline of on-chip and off-chip memory, the query performance bottleneck often lies in the access of off-chip memory. Since One-Memory Hash needs only one access of off-chip memory per query, it can achieve high query speed. In our experiments, One-Memory Hash achieves a query speed of 55 Mqps (Million query per second) on GPU/CPU platforms, and 200 Mqps on FPAG platforms.

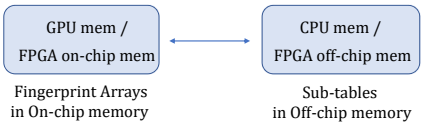

Figure 1: Hierarchical structure of One-Memory Hash: on-chip and off-chip memory combinations.

1.3 Key Contributions

- We propose a novel hierarchical hash table scheme, One-Memory Hash, which achieves one off-chip memory access per query in the worst case, supports fast incremental

[1] For convenience, in this paper we assume a KV pair fits into a machine word, and can be fetched through a single memory access.

[2] Strictly speaking, the global memory of GPU is not on-chip memory. However, it is much faster than the DRAM of CPU, and the data structures stored in GPU can be processed in parallel.

updates, has fast extension when the table is full, and guarantees no update failures.

- We derive detailed mathematical analysis of One-Memory Hash, showing it is practical in real-world scenarios.
- We implement One-Memory Hash on GPU/CPU and FPGA platforms, and conduct head-to-head comparisons against 9 common hashing schemes on 3 real-world datasets and 1 synthetic dataset. Experimental results show that One-Memory Hash significantly outperforms all 9 prior hashing schemes.

2 Related Work

Hashing schemes can be divided into four categories: classic hashing, multi-choice hashing, cuckoo hashing and perfect hashing. This section only covers typical algorithms. For additional hashing schemes, please refer to the survey [10].

Classic Hashing: Classic hashing uses a single hash table. Collision resolution approaches include chaining using linked lists, linear probing, and double probing.

Multi-choice hashing: The basic idea of multi-choice hashing is to allocate multiple candidate buckets through multiple hash functions for each element. D-left hashing [58] is one of the most typical multi-hashing schemes. It uses d equal size sub-tables with associated hash functions and linked lists. If there are multiple shortest linked lists, it always inserts the KV pair into the leftmost one to achieve a high load factor. Bloom filters (BF) [16] have been used in multi-choice hashing schemes to help determine which candidate bucket to check, so as to improve query speed. Typical algorithms include choice hashing [30], segment hashing [32], and peacock hashing [34]. Usually, a BF is maintained for each sub-table and records all elements that have been inserted into that sub-table. When querying a key, each BF is checked. Only when the BF reports positive, the corresponding sub-table will be checked. However, due to the false positive errors of BF, it does not improve the worst-case query performance.

Cuckoo hashing: Cuckoo hashing is an elegant hashing scheme that has a high load factor and query speed, but with update failures [43, 44]. The basic cuckoo hashing uses a single hash table T and two hash functions $h_1(.), h_2(.)$. To insert a KV pair $\langle k_l, v_l \rangle$, it locates two candidate buckets $T[h_1(k_l)]$ and $T[h_2(k_l)]$. If they are both full, cuckoo hashing randomly chooses a bucket, kicks out the KV pair in it, replaces it by $\langle k_l, v_l \rangle$, and repeats this process for the kicked pair recursively until an empty bucket is found. However, the kicking process is not guaranteed to terminate. To address this issue, cuckoo hashing sets a threshold of *500 kicks*: after 500 kicks, the kicking process stops and an update failure is reported. To query a key, it simply checks two candidate buckets. Many variants and applications of cuckoo hashing have been successful, including cuckoo filter [24], BCHT [46], SmartCuckoo [53], Horton Table [18], MegaKV [63] and MemC3 [25].

Perfect Hashing: Given n KV pairs and m ($m > n$) buckets, perfect hashing finds hash functions that can hash these n pairs into m buckets without any collision. When $m = n$, it is called Minimal Perfect Hashing. Typical schemes include CHD, BRZ, CHM, and FCH [1]. The main limitation of perfect hashing is that it does not support fast incremental updates.

3 One-Memory Hash Algorithms

In this section, we first present the basic version – OMH_1, which achieves the design goal, but has low memory efficiency. Then we propose two optimization versions: OMH_2 and OMH_3. Table 1 summarizes the symbols and abbreviations frequently used in the paper.

Table 1: Symbols frequently used in the paper.

Symbol	Description
$\langle k_l, v_l \rangle$	a KV pair with key k_l and value v_l
d	# of sub-tables
n	# of total KV pairs
m	# of buckets in a sub-table in OMH_1
m_i	# of buckets in the i^{th} sub-table in OMH_2 and OMH_3
T_i	the i^{th} sub-table in the main table
$T_i[j]$	the j^{th} bucket in the i^{th} sub-table
$L_i[j]$	linked list of the j^{th} bucket in the i^{th} sub-table
t_i	the i^{th} array in the fingerprint table
$t_i[j]$	the j^{th} bucket in the i^{th} fingerprint array
$t_i[j].index/fp$	index/fingerprint field of $t_i[j]$
w	# of bits in a bucket in the fingerprint table
w_1	# of bits in the index field
w_2	# of bits in the fingerprint field
r	common ratio of the geometric sequence in OMH_2 and OMH_3
$h_i(.)$	hash function associated with the i^{th} sub-table and the fingerprint array
$h^F(.)$	hash functions for calculating fingerprints
b	# of cells in one bucket in OMH_2 and OMH_3
s	# of entries in the stash
c	ratio of # of allocated buckets to # of total KV pairs

3.1 OMH_1– One Memory Access per Query

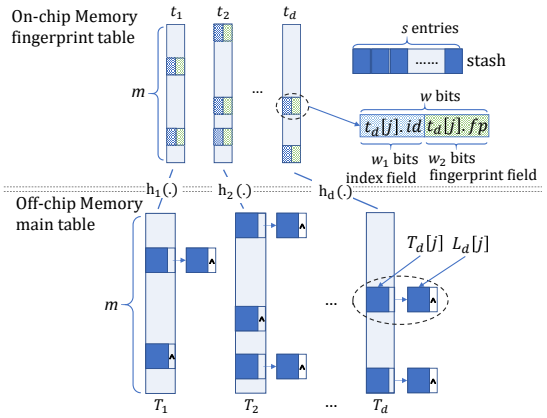

Figure 2: Data structure of OMH_1.

3.1.1 Data Structure

As shown in Figure 2, OMH_1 consists of three components: A fingerprint table and a stash in on-chip memory, and a main

table in off-chip memory. The main table is composed of d sub-tables $T_1, T_2, \ldots, T_d$. The fingerprint table is composed of d arrays $t_1, t_2, \ldots, t_d$. Both T_i and t_i $(1 \leqslant i \leqslant d)$ have m buckets, and are associated with the same hash function $h_i(.)$ $(1 \leqslant i \leqslant d)$, whose output is uniformly distributed in the range $[1, m]$. We denote the j^{th} bucket in the i^{th} sub-table and array by $T_i[j]$ and $t_i[j]$, respectively. Each bucket $T_i[j]$ $(1 \leqslant i \leqslant d, 1 \leqslant j \leqslant m)$ in the main table has a linked list which is denoted by $L_i[j]$. The stash is an array of s (s is usually very small, *e.g.*, 32) entries, and each entry is a KV pair.

A bucket $T_i[j]$ in the main table together with its corresponding bucket $t_i[j]$ in the fingerprint table records a KV pair $\langle k_l, v_l \rangle$: $T_i[j]$ is large and stores $\langle k_l, v_l \rangle$, while $t_i[j]$ is small and stores a fingerprint of $\langle k_l, v_l \rangle$. There are 2^{w_1} (w_1 is a parameter which will be explained later) independent FP-hashes h_q^{F} $(0 \leqslant q \leqslant 2^{w_1} - 1)$ used to compute fingerprints. q is called *the index* of h_q^{F}. A bucket $t_i[j]$ in the fingerprint table has w bits, and contains two fields: the *fingerprint field* ($t_i[j].fp$) and the *index field* ($t_i[j].index$). The fingerprint field has w_2 bits (the length of a fingerprint) and is used to record the fingerprint of the KV pair stored in $T_i[j]$. The index field has w_1 ($w_1 = w - w_2$) bits and is used to record the index of the FP-hash that generates the fingerprint. The initialization of OMH$_1$ is to set all the buckets $T_i[j]$ and $t_i[j]$ $(1 \leqslant i \leqslant d, 1 \leqslant j \leqslant m)$ to 0, representing the empty bucket.

3.1.2 Operations

Construction: Given a set of KV pairs $\mathbb{S}$, OMH$_1$ first builds the main table, and then the fingerprint table. During the process, we propose two techniques to meet the design goal: `footprint recording` and `exclusive fingerprinting`, both discussed below.

1) Main table construction: For each KV pair $\langle k_l, v_l \rangle$ in $\mathbb{S}$, OMH$_1$ maps it to d buckets in the main table: $T_1[h_1(k_l)]$, $T_2[h_2(k_l)]$, …, $T_d[h_d(k_l)]$. We call them the *candidate buckets* of $\langle k_l, v_l \rangle$. OMH$_1$ checks them sequentially and stops when an empty bucket is found. There are three possible cases:

Case 1: If an empty bucket is found, OMH$_1$ inserts $\langle k_l, v_l \rangle$ into it. For the other $d-1$ unchosen candidate buckets, OMH$_1$ inserts $\langle k_l, v_l \rangle$ into the linked lists of all of them. This process is called **footprint recording**.

Case 2: If no empty bucket is found and the stash is not full, OMH$_1$ inserts $\langle k_l, v_l \rangle$ into the stash.

Case 3: If no empty bucket is found and the stash is full, then OMH$_1$ cannot store $\langle k_l, v_l \rangle$. We call this *load overflow*. To address load overflow, OMH$_1$ activates a mechanism called **instant table extension**, which dynamically adds new empty sub-tables to store $\langle k_l, v_l \rangle$. We defer the details of the mechanism later in Section 3.1.3.

2) Fingerprint table construction: After the construction of the main table, for each bucket $T_i[j]$ that stores a KV pair $\langle k_l, v_l \rangle$, OMH$_1$ chooses an *adequate* FP-hash to compute a fingerprint, and records the index of this FP-hash and the fingerprint in $t_i[j]$. *Adequate* means that the computed fingerprint of $\langle k_l, v_l \rangle$ is different from all the fingerprints of KV pairs in the linked list of $T_i[j]$. OMH$_1$ utilizes a technique called *exclusive fingerprinting* to choose the adequate FP-hash.

Exclusive fingerprinting: This technique is inspired by SetSep [64]. Let $\langle k_l, v_l \rangle$ be a KV pair stored in $T_i[j]$, $L_i[j]$ be the linked list of $T_i[j]$, and $\mathbf{S}$ be the set of keys in $L_i[j]$. An FP-hash h_q^{F} is adequate if and only if $h_q^{\mathrm{F}}(k_l) \neq h_q^{\mathrm{F}}(s)$ $(\forall s \in \mathbf{S})$. OMH$_1$ tries the 2^{w_1} FP-hashes sequentially, and if it finds an adequate h_q^{F}, OMH$_1$ records q in $t_i[j].index$, and $h_q^{\mathrm{F}}(k_l)$ in $t_i[j].fp$. If no adequate FP-hash is found, we say that a *fingerprint construction failure* happens. If the stash is not full, OMH$_1$ inserts $\langle k_l, v_l \rangle$ into the stash; if the stash is full, OMH$_1$ again utilizes the instant table extension mechanism to handle the situation. The pseudo-code of the construction process is shown in Algorithm 1 in Appendix 8.1[3].

Due to footprint recording, a KV pair $\langle k_l, v_l \rangle$ is inserted into all the linked lists of its $d-1$ unchosen candidate buckets. Exclusive fingerprinting guarantees $\langle k_l, v_l \rangle$ has no matched fingerprint in these buckets. Therefore, $\langle k_l, v_l \rangle$ has *exactly one matched fingerprint*, right in its chosen candidate bucket.

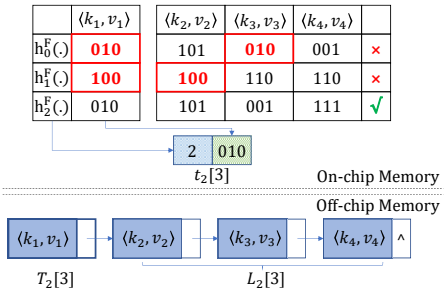

Figure 3: The process of exclusive fingerprinting with $w_1 = 2$ and $w_2 = 3$ for $\langle k_1, v_1 \rangle$ stored in $T_2[3]$.

Example: Figure 3 illustrates the process of exclusive fingerprinting. $\langle k_1, v_1 \rangle$ is stored in $T_2[3]$, and its linked list $L_2[3]$ has three KV pairs: $\langle k_2, v_2 \rangle$, $\langle k_3, v_3 \rangle$, and $\langle k_4, v_4 \rangle$. OMH$_1$ checks the FP-hashes sequentially. Because $h_0^{\mathrm{F}}(k_1) = h_0^{\mathrm{F}}(k_3)$ and $h_1^{\mathrm{F}}(k_1) = h_1^{\mathrm{F}}(k_2)$, h_0^{F} and h_1^{F} are inadequate. h_2^{F} is found adequate, therefore OMH$_1$ records 2 in $t_2[3].id$ and the third fingerprint 010 in $t_2[3].fp$.

Insertion: To insert a KV pair $\langle k_l, v_l \rangle$, OMH$_1$ maps it to d buckets in the main table: $T_1[h_1(k_l)]$, $T_2[h_2(k_l)]$, …, $T_d[h_d(k_l)]$. OMH$_1$ checks them sequentially and stops if an empty bucket is found. During this process, there are three possible cases:

Case 1: If an empty bucket is found, OMH$_1$ inserts $\langle k_l, v_l \rangle$ into it, and performs exclusive fingerprinting for it. OMH$_1$ also inserts $\langle k_l, v_l \rangle$ into all the linked lists of the $d-1$ unchosen buckets for footprint recording. For the unchosen candidate buckets, if they contain a KV pair $\langle k_l', v_l' \rangle$ and record a FP-hash h_q^{F}, OMH$_1$ checks if h_q^{F} is still adequate, as $\langle k_l, v_l \rangle$ is added to

[3]Due to the length limit, we put the full paper with the appendix at the anonymous github repository https://github.com/onememoryhash/OMH

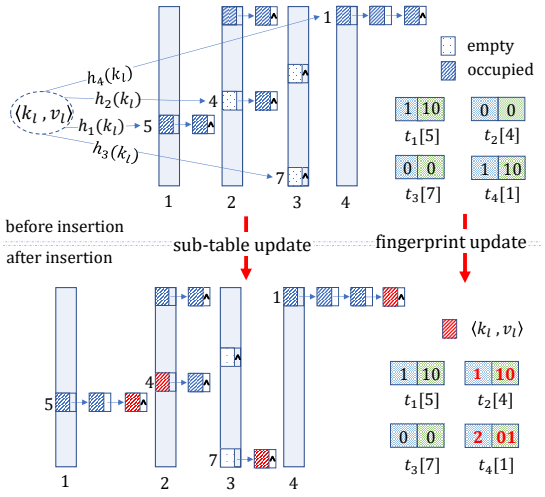

Figure 4: Insertion with $d = 4$, $w_1 = 2$, and $w_2 = 2$. Stash is not shown as it is not involved in the insertion process in this example.

the linked list. If h_q^{F} becomes inadequate, *i.e.*, $h_q^{\mathrm{F}}(k_l) = h_q^{\mathrm{F}}(k_l')$, OMH_1 applies the exclusive fingerprinting for $\langle k_l', v_l' \rangle$.

Case 2: If no empty bucket is found and the stash is not full, then OMH_1 inserts $\langle k_l, v_l \rangle$ into the stash.

Case 3: If no empty bucket is found and the stash is full, then the instant table extension mechanism is activated. The mechanism is discussed later in Section 3.1.3. Algorithm 3 in Appendix 8.1 shows the pseudo-code of the insertion process.

Example: Figure 4 illustrates the process of inserting $\langle k_l, v_l \rangle$ with $d = 4$, $w_1 = 2$ and $w_2 = 2$. The stash is not shown as it is not involved in the process. $\langle k_l, v_l \rangle$ has two empty candidate buckets: $T_2[4]$ and $T_3[7]$. It is thus inserted into $T_2[4]$, and added to the linked lists of its three unchosen candidate buckets, *i.e.*, $L_1[5]$, $L_3[7]$, and $L_4[1]$. OMH_1 then performs exclusive fingerprinting for $\langle k_l, v_l \rangle$, and checks whether the FP-hash of pairs in $T_1[5]$ and $T_4[1]$ are still adequate. Finally, $t_2[4]$ and $t_4[1]$ are updated.

Query: OMH_1 queries a key k_l in the following process: First, OMH_1 checks the stash. If k_l is found, OMH_1 reports the value and the query ends. Otherwise, OMH_1 locates d candidate buckets $t_1[h_1(k_l)]$, $t_2[h_2(k_l)]$, …, $t_d[h_d(k_l)]$. For each bucket $t_i[h_i(k_l)]$ $(1 \leqslant i \leqslant d)$, OMH_1 gets the FP-hash index q and fingerprint f, and computes $h_q^{\mathrm{F}}(k_l)$ to see if it matches with f. There are two possible situations:

Situation 1: If there is only one matched fingerprint, OMH_1 locates the bucket in the fingerprint table that contains the matched fingerprint, and checks its corresponding bucket in the main table: if k_l is found, OMH_1 reports the corresponding value; if k_l is not found, OMH_1 reports NULL, indicating the queried key does not exist.

Situation 2: If there is more than one matched fingerprints, or there is no matched fingerprint, OMH_1 reports a NULL.

In the above situations, no matter the query succeeds or not, there is *at most one off-chip memory access*. The pseudo-code of query is shown in Algorithm 4 in Appendix 8.1.

Example: Figure 5 illustrates the process of querying k_l with $d = 4$, $w_1 = 2$,$w_2 = 2$, and $s = 4$. First OMH_1 checks the stash, and k_l is not found. Then, OMH_1 checks four candidate buckets of k_l in the fingerprint table, and finds that there is only one matched fingerprint in $t_1[1]$. Therefore, OMH_1 checks the corresponding bucket $T_1[1]$ in the main table, which contains the key k_l, hence OMH_1 returns the value v_l.

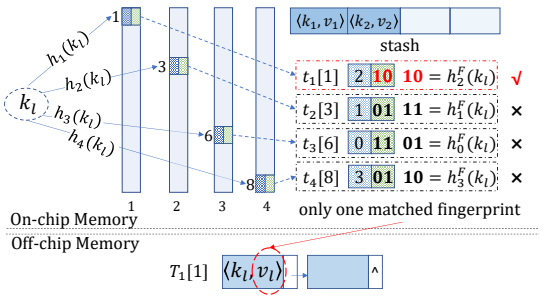

Figure 5: Query with $d = 4$, $w_1 = 2$ and $w_2 = 3$.

Deletion: To delete a KV pair $\langle k_l, v_l \rangle$, OMH_1 first queries it. If it is in the stash, OMH_1 removes it from the stash; if it is in a bucket in the main table, OMH_1 sets that bucket and its corresponding bucket in the fingerprint table to 0, and deletes $\langle k_l, v_l \rangle$ from the linked lists of its unchosen candidate buckets.

3.1.3 Eliminating Update Failures

In this part, we improve OMH_1 to make it free of update failures. Currently, OMH_1 has two types of update failures: *load overflow* and *fingerprint construction failure*. We propose the **instant table extension mechanism** to address both.

Handling load overflow: Hash schemes suffer from the problem of load overflow in many real-world scenarios for two reasons: 1) The total number of KV pairs is often unknown in advance; 2) Even if it is known, it can change dynamically. To handle load overflow, some hash schemes use very long linked lists, at the cost of a low query speed [54]. Some schemes choose to drop the incoming KV pair [33]. Some reconstruct the whole table, at an unbearable time cost [44]. In contrast to these solutions, we propose a novel instant table mechanism, which stores the incoming KV pair in newly added sub-tables. *Our mechanism has only $O(1)$ time complexity, and only slightly degrades the query and update performance.* When $\langle k_l, v_l \rangle$ encounters load overflow, the mechanism proceeds as follows:

Step 1: Add a new sub-table T_{d+1} and a new array t_{d+1}, which have the same structure and size as T_1 and t_1, and associate them with the hash function $h_1(.)$.

Step 2: Insert $\langle k_l, v_l \rangle$ into $T_{d+1}[h_1(k_l)]$, and the linked lists of $T_1[h_1(k_l)]$, $T_2[h_2(k_l)]$, $\cdots$, $T_d[h_d(k_l)]$.

Step 3: Apply exclusive fingerprinting for $\langle k_l, v_l \rangle$, and KV pairs in its unchosen candidate buckets if their FP-hashes need to be re-calculated.

The key part of this mechanism is to *reuse the hash function* $h_1(.)$ *for* T_{d+1} *and* t_{d+1}, through which it gains $O(1)$ time complexity. Note that the addition of T_{d+1} offers each of all the previously inserted KV pairs a new candidate bucket. The footprint recording then requires copying all the previously inserted KV pairs to the linked lists of their new candidate buckets in T_{d+1}, which introduces significant time overhead. The reuse of $h_1(.)$ avoids this overhead. Since T_{d+1} and t_{d+1} are associated with $h_1(.)$, KV pairs mapped to $T_1[j]$ $(1 \leqslant j \leqslant m)$ will also be mapped to $T_{d+1}[j]$, *i.e.*, their new candidate bucket will definitely be $T_{d+1}[j]$. Therefore, OMH$_1$ copies nothing since $L_{d+1}[j]$ can be obtained from $L_1[j]$ and $T_1[j]$, reducing the time overhead to be only the allocation of a new sub-table and a new fingerprint array. For the latter insertion and query, OMH$_1$ then checks not only the original sub-tables and fingerprint arrays, but also the newly added ones.

The instant table extension mechanism has very good **extensibility**: If load overflow happens again after T_{d+1} and t_{d+1} are added, OMH$_1$ can add more sub-tables and arrays using the same approach: it adds a new sub-table T_{d+j} $(j \geqslant 2)$ and a new fingerprint array t_{d+j} with the same structure and size as T_j and t_j, and associate them with h_j.

Handling fingerprint construction failures: When a KV pair $\langle k_l, v_l \rangle$ encounters a fingerprint construction failure and the stash is full, OMH$_1$ activates the instant table extension mechanism and inserts the pair into T_{d+1}, giving it another chance for exclusive fingerprinting. To guarantee that $\langle k_l, v_l \rangle$ gets an adequate fingerprint in T_{d+1}, OMH$_1$ adds two more bits in a bucket of t_{d+1}, with the fingerprint and index field both having one more bit. Twice longer fingerprints and twice more FP-hashes ensure with high probability that $\langle k_l, v_l \rangle$ obtains an adequate fingerprint. Even if the fingerprint construction fails again, OMH$_1$ can add more sub-tables and arrays until it finally gains an adequate fingerprint. Let us point out that this mechanism is really a "release valve" for our algorithm. The analysis in Section 4.2 shows that the probability of fingerprint construction failure is negligible with proper w_1 and w_2, and we have never encountered a fingerprint construction failure in experiments (see Figure 24 in Appendix 8.5). The pseudo-code of instant table extension is shown in Algorithm 5 in Appendix 8.1.

3.1.4 Analysis of OMH$_1$

Advantages: The most salient advantage of OMH$_1$ is that it needs at most one off-chip memory access for each query, even in the worst case. Furthermore, OMH$_1$ has no update failure and allows dynamic updates.

Limitation: To achieve a high load factor[4], OMH$_1$ needs a large number of sub-tables (see Figure 22 in Appendix Appendix 8.4). As a result, a KV pair has some copies, which consumes too much off-chip memory.

[4]Here, the load factor is defined as the ratio of # of inserted KV pairs to # of buckets when the stash first becomes full.

3.2 OMH$_2$– Improving the Load Factor

The goal of OMH$_2$ is to improve the load factor. First, OMH$_2$ expands the buckets in the main table to have multiple *cells*, where each cell stores a KV pair. Second, recall that OMH$_1$ always inserts a pair $\langle k_l, v_l \rangle$ into its leftmost empty candidate bucket, thus more KV pairs will be inserted into the left sub-tables. Therefore, OMH$_2$ lets the left sub-tables have more buckets. This is similar to the d-left hash table [58].

Data structure: Compared with OMH$_1$, the data structure of OMH$_2$ have two differences: 1) A bucket is divided into b cells, and each cell can store a KV pair. Correspondingly, a bucket in the fingerprint table stores b fingerprints and indexes; 2) the sizes of sub-tables (and fingerprint arrays) follow a decreasing geometric sequence with a common ratio r.

Insertion: The insertion process of OMH$_2$ is very similar to that of OMH$_1$. Given a pair $\langle k_l, v_l \rangle$, OMH$_2$ maps it to d candidate buckets $T_1[h_1(k_l)\%m_1]$, $T_2[h_2(k_l)\%m_2]$, $\cdots$, $T_d[h_d(k_l)\%m_d]$. OMH$_2$ checks the cells in these buckets sequentially and stops when an empty cell is found. During the process, there are three possible cases:

Case 1: If an empty cell is found, OMH$_2$ inserts $\langle k_l, v_l \rangle$ into that cell and all linked lists of the $d-1$ unchosen candidate buckets. Then, exclusive fingerprinting is performed for $\langle k_l, v_l \rangle$, as well as KV pairs in the unchosen buckets if their FP-hashes need recalculation. Exclusive fingerprinting now has to ensure that the fingerprint of a KV pair is different from not only the fingerprints of KV pairs in the linked list, but also the fingerprints of KV pairs in the other cells of the bucket.

Case 2: If no empty cell is found and the stash is not full, OMH$_2$ inserts $\langle k_l, v_l \rangle$ into the stash.

Case 3: If no empty cell is found and the stash is full, then the instant table extension mechanism is activated (same as in OMH$_1$).

Query: The query process of OMH$_2$ is exactly the same as that of OMH$_1$, except that OMH$_2$ has to check b fingerprints in one bucket in the fingerprint table.

Deletion: Same as OMH$_1$.

Advantages: OMH$_2$ achieves a much higher load factor compared with OMH$_1$ (see Figure 22 in Appendix Section 8.4).

Limitations: OMH$_2$ has $d-1$ copies for a KV pair when employing d sub-tables, which still consumes too much off-chip memory.

3.3 OMH$_3$– Reducing the Copies

In OMH$_3$, we propose a new technique called `hash sharing` to reduce the number of copies for each KV pair.

Hash sharing: The key idea of hash sharing is to make the second half of sub-tables share the hash functions with the first half. Through hash sharing, the linked lists in the second half of the sub-tables can be removed, because they can be obtained from the linked lists in the first half of the sub-tables.

Figure 6 illustrates how hash sharing works in principle. Suppose there are two sub-tables T_1 and T_2 with length of 6 and 3. Instead of using two hash functions for each sub-table,

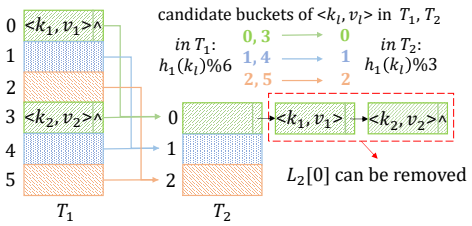

Figure 6: Principle of hash sharing.

OMH_3 uses only one hash function $h(.)$ for both of them. Hash sharing utilizes the property of the modulo operation: since 6 is divisible by 3, if $h(k_l)\%3 = 0$, then $h(k_l)\%6$ is either 0 or 3. This means that any KV pair mapped to $T_2[0]$ is definitely mapped to $T_1[0]$ or $T_1[3]$ (see $\langle k_1, v_1\rangle$ and $\langle k_2, v_2\rangle$ in Figure 6). Therefore, there is no need to maintain the linked list $L_2[0]$. In general, all linked lists in the second half of sub-tables $T_{d/2+1} \cdots T_d$ can be obtained from linked lists in the first half of sub-tables $T_1 \cdots T_{d/2}$, thus can be removed. Therefore, a KV pair has only $d/2-1$ copies, which is half as many compared to OMH_2. Note that the linked lists in the second half of sub-tables are usually long since they are composed of several linked lists in the first half of sub-tables. Therefore, the probability of fingerprint construction failure is relatively high there. To handle this, OMH_3 makes KV pairs in the second half of sub-tables have longer fingerprints and more FP-hashes. One may reasonably expect that hash sharing might incur a drop in randomness, increasing the hash collisions and decreasing the load factor. However, our experiments show that the effect of hash sharing on load factor is actually negligible (please refer to Figure 21 in Section 8.4).

Data structure: The data structure of OMH_3 is almost the same as that of OMH_2, except that OMH_3 has only $d/2$ hash functions $h_1, h_2, \cdots, h_{d/2}$. $T_{d/2+i}$ $(1 \leqslant i \leqslant d/2)$ and T_i is associated with the same hash function $h_i(.)$. A bucket in $t_{d/2+1}$, $t_{d/2+2} \cdots t_d$ has more than w bits, with w_1' $(> w_1)$ bits in the index field and w_2' $(> w_2)$ bits in the fingerprint field.

Insertion: The insertion process of OMH_3 is the same as that of OMH_2, except that a KV pair only has to be inserted to the linked lists of its first half of candidate buckets.

Query and Deletion: The query and deletion process of OMH_3 is the same as the one of OMH_2.

Advantages: OMH_3 overcomes all disadvantages of the first two versions, while inheriting all their advantages.

4 Analysis of Insertion

In this section, we give a detailed and comprehensive analysis of the insertion operation of One-Memory Hash, providing the readers with a better understanding of the algorithm.

The insertion of One-Memory Hash has four steps: 1) Find an empty bucket; 2) Perform footprint recording; 3) Compute a proper fingerprint using exclusive fingerprinting; 4) Check other fingerprints in the unchosen candidate buckets. The complexity of the first and second step if of $O(d)$. We analyze step 3 and 4 in details below. We only present the main lemmas and theorems here, and the proofs are shown in Appendix 8.2 .

4.1 Time Cost of Exclusive Fingerprinting

The time cost of exclusive fingerprinting depends on the length of the linked list where the technique is performed on. Therefore, we analyze the expected length of the linked lists in OMH_3. We first analyze in the case of OMH_1, then extends the results to OMH_3. Suppose there are n KV pairs and d sub-tables, and each sub-table contains $\frac{n}{d}$ buckets. Let l be a random variable that denotes the length of a linked list of any bucket in any sub-table. Lemma 1 gives a 99.7% confidential interval of l:

Lemma 1 *When n is very large, the following equation gives an approximating* 99.73% *confidential interval of l:*

$$P(d - 3\sqrt{d} \leqslant l \leqslant d + 3\sqrt{d}) = 99.73\% \tag{1}$$

Theorem 1 expands the result of Lemma 1 to OMH_3.

Theorem 1 *Suppose in* OMH_3, $\frac{m_{i+1}}{m_i} = r$, *there are n KV pairs, d sub-tables and each bucket has b cells. Suppose we allocate* $c \cdot n$ *cells for these KV pairs (c is a constant a bit greater than 1,* e.g., *1.02). Let* l_i *be a random variable denoting the length of a linked list of any bucket in the* i^{th} *sub-table. When n is very large, the following equation gives an approximating* 99.73% *confidential interval of* l_i*:*

$$\begin{aligned} P\Big(&\frac{b(1-r^d)}{c(1-r)r^{i-1}} - 3\sqrt{\frac{b(1-r^d)}{c(1-r)r^{i-1}}} \leqslant l_i \\ &\leqslant \frac{b(1-r^d)}{c(1-r)r^{i-1}} + 3\sqrt{\frac{b(1-r^d)}{c(1-r)r^{i-1}}}\Big) = 99.73\% \end{aligned} \tag{2}$$

With $r = 0.5$, $d = 8$, $b = 6$ and $c = 1.02$, Theorem 1 gives that l_1 approximates $N(11.71, 11.71)$, and with probability 99.73%, l_1 falls in $[1.44, 21.98]$. The correctness of Theorem 1 is verified as in Figure 20 in Appendix 8.3. Figure 7 shows the time overhead of exclusive fingerprinting on linked lists with different lengths on three different synthesized datasets[5]. It can be observed that the the time cost introduced by exclusive fingerprinting is small and acceptable.

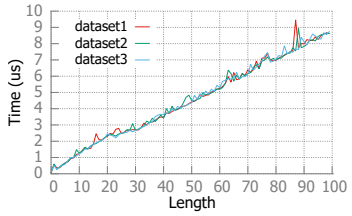

Figure 7: The time overhead of exclusive fingerprinting on linked lists with different length.

[5]the experiment is done on the CPU platform as described in Section 6.2

4.2 Fingerprint Construction Failure

We now analyze the probability of fingerprint construction failure.

Lemma 2 *Suppose the fingerprint field has w_2 bits. Let x be a random variable. x denotes the required number of FP-hashes to obtain an adequate fingerprint when performing exclusive fingerprinting on a linked list of length l. Then, x follows the Geometric distribution $G((1-\frac{1}{2^{w_2}})^l)$,* i.e.,

$$P(x=k) = (1-(1-\frac{1}{2^{w_2}})^l)^{k-1} \times (1-\frac{1}{2^{w_2}})^l \tag{3}$$

Theorem 2 *Given a constant α, if we want to obtain an adequate fingerprint on a linked list of length l with probability larger than α, the number of FP-hashes x should satisfy:*

$$x \geqslant \frac{ln(1-\alpha)}{ln(1-q)}, \quad where \;\; q=(1-\frac{1}{2^{w_2}})^l \tag{4}$$

(l,α)	99%	99.99%	99.9999%
50	2	4	6
500	10	20	30
1000	31	61	91

Table 2: The required minimum x for obtaining a proper fingerprint given different l and α, using Theorem 2.

Table 2 shows the minimum x given different α and l, with $w_2 = 9$. In our experiments, we allocate 6 bits (64 FP-hashes) for the index field of the first half of sub-tables (which have very short linked lists), and 9 bits (512 FP-hashes) for the index field of the second half of sub-tables (which have longer linked lists). According to Table 2, these number of FP-hashes are large enough to obtain a proper fingerprint with probability larger than 99.99%, even on a linked list with length of 1000. The fact that we never encounter a fingerprint construction failure in our experiments (see Figure 24 in Appendix 8.5) verifies the correctness of Theorem 2.

With the above analysis and evaluation, we can conclude that the insertion operation of One-Memory Hash is totally practical in real-world scenarios.

5 Implementation

To test the actual performance of One-Memory Hash, we implement it on two hierarchical memory platforms: GPU/CPU platform, and FPGA platform. We also simulate One-Memory Hash using pure CPU to tune its parameters.

5.1 CPU Simulation

We implement all considered hash schemes in C++. For SmartCuckoo, we use the code provided by its authors at [2]. The hash functions we use in the experiments are BOB Hash obtained from [5]. All source codes of our experiments are available at GitHub [9].

5.2 GPU/CPU Implementation

With the fast development of GPU (Graphics Processing Unit), its performance increases rapidly while the cost decreases rapidly. The main memory in GPU is called global memory, which is much faster than DRAM memory in CPU. With thousands of cores, the GPU is often much faster than the CPU. For hash tables, the key metric is the query speed. Therefore, we implement OMH_3 through a GPU/CPU pipeline to accelerate query operations using the CUDA Toolkit [6]. The CPU maintains the sub-tables of OMH_3, while the GPU maintains the stash and fingerprint arrays. For each query, the GPU does the comparison of fingerprints, and the CPU does the lookup in the sub-tables. Specifically, the queried keys are organized into many batches. Each batch of keys is processed as follows. First, a batch of keys is copied from the CPU into the GPU. Second, the GPU leverages a large number of threads to process those keys concurrently and generates intermediate results. Third, the results of this batch of keys are copied from the GPU back into the CPU. Finally, the CPU generates the final lookup results for this batch of keys. We also leverage multi-streaming to process different batches concurrently as long as they are going through different steps.

5.3 FPGA Implementation

FPGA (Field-Programmable Gate Array) is becoming a convenient application-specific computing solution thanks to its low power, high hardware parallelism, and fast time-to-market. FPGA's BRAM (Block RAM) and URAM (UltraRAM) are two kinds of on-chip memories with low latency [3]. Their flexible memory management suits well to our One-Memory Hash algorithm. We develop an FPGA implementation of the query operation of OMH_3. After building the hash table, the limited-bandwidth off-chip memory holds the sub-tables, while the high-bandwidth on-chip BRAM maintains the stash and the fingerprint arrays. For each queried key, the FPGA implementation performs the query process as described previously. If the hash table has the queried key, it eventually gives the address of the key-value pair in the off-chip memory or gives the corresponding value in the stash.

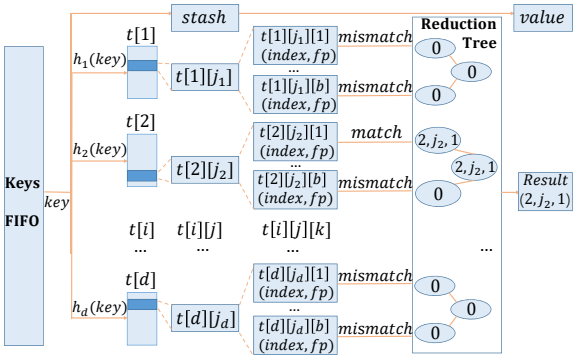

Figure 8: FPGA Implementation.

As shown in Figure 8, the input of the implementation is a FIFO of the queried keys. We use $t[i][j][k]$ to denote the k^{th} cell in the j^{th} bucket in the i^{th} fingerprint array. For each key in the FIFO, the FPGA implementation first checks the stash. If found, it reports the corresponding value. Otherwise, it lo-

cates d candidate buckets in parallel. For each bucket, it reads the FP-hash indexes and fingerprints of b cells in parallel, checks the matching situations, and obtains the results. Then, a reduction tree is developed to find the matching address from these $d \cdot b$ results. Finally, it puts the address in the result FIFO. All of these steps are pipelined, so the implementation can query one key per clock cycle of the FPGA. Therefore, the query processing throughput is proportional to the FPGA clock frequency within a typical range.

6 Experimental Results

In this section, we conduct extensive experiments on CPU, GPU/CPU, and FPGA platforms. In the CPU simulation part, we conduct head-to-head comparisons with 9 typical hash tables in terms of memory accesses and memory efficiency. In the GPU and FPGA part, we report the actual query performance of One-Memory Hash.

6.1 Datasets

We use in total 4 datasets in our experiments, including 3 real-world datasets and 1 synthetic dataset.

Taxi: This dataset includes around 500k records of the trajectories performed by 442 taxis running in the city of Porto, in Portugal [8]. Each record includes a trip ID, a timestamp, a taxi ID as well as other fields. We use the trip ID and taxi ID as our $< key, value >$ pairs.

Post: This dataset includes over 1.5 million records [7]. Each record includes six fields: ID1, ID2, name, date of birth, postcode and matching status. We joint ID1 and ID2 in one record together as our key and use the matching status as our value.

DocWords: This dataset includes multiple text collections in the form of bag-of-words [4]. It contains nearly 70 million items in total. Each item contains several fields. We combine the DocID and WordID fields as our $< key, value >$ pairs. Among the 70 million items, we sample 5 million to form the experimental dataset.

Synthetic: This dataset is generated using the C++ STL Mersenne Twister random integer generator [41]. The generated integers are unsigned 32-bit long, and follow a pseudo-random uniform distribution. 10 million integers are generated in total.

6.2 CPU Simulation

Setup: We performed all the experiments on a machine with 4-cores (8 threads, Intel Core i7@2.5 GHz) and 16 GB of DRAM memory. It has two 32KB L1 caches for each core, one 256KB L2 cache for each core, and one 6MB L3 cache shared by all cores.

Metrics: The performance of hash tables are usually measured in terms of operation speed and memory efficiency. In the simulation, we use *Average number of Memory Accesses* and *Worst-Case number of Memory Accesses* as the metrics for operation speed, and use *Pressure Load Factor* as the metric for memory efficiency.

1) Average number of Memory Accesses (AMA): Suppose it takes M memory accesses to operate (insert, query, or delete) N KV pairs. The average number of memory access is calculated as: AMA = $\frac{M}{N}$.

2) Worst-Case number of Memory Accesses (WCMA): Let $\{\langle k_i, v_i\rangle\}_{i=1}^{N}$ be a set of KV pairs, and m_j the number of memory accesses when doing operations on pair $\langle k_j, v_j\rangle$. The worst-case number of memory access is calculated as: WCMA = $\max\{m_i\}_{i=1}^{N}$.

3) Pressure Load Factor (PLF): Pressure Load Factor is defined as the load factor when the first update failure happens. When a collision happens during insertions, linear hashing, double hashing, cuckoo hashing, BCHT, SmartCuckoo and segment hashing with linear probing will try to probe a new bucket. We set the maximum number of probes to 500. For chaining hashing, half of the memory is used for hash table buckets, and the other half for linked lists [44]. If chaining hashing runs out of linked lists, an update failure occurs. For d-left hashing, 80% of the memory is used for the buckets and 20% for the linked lists [58]. For peacock hashing, an update failure occurs when all candidate buckets are full. For OMH_3, we ignore the instant table extension mechanism here, and an update failure occurs when the stash is full.

We first compare the three versions of OMH to prove the effectiveness of the techniques proposed in OMH_2 and OMH_3, and the comparison results are shown in Appendix 8.4. Then, we compare OMH_3 with prior hash schemes including chaining hashing [54], linear hashing [20], double hashing [20], cuckoo hashing [44], d-left hashing [58], peacock hashing [33], segment hashing [32], BCHT [46] and smartcuckoo [53] in terms of operation (insertion, query, deletion) AMA, operation WCMA, and PLF. For all these hash schemes, the deletion has exactly the same number of memory accesses as the query, therefore we only show the experimental results for the query operation.

Parameter Setting: In the following experiments, segment hashing has 16 segments, and d-left hashing has 8 sub-tables. For hash schemes using auxiliary data structures in on-chip memory, *i.e.*, segment hashing, peacock hashing and OMH_3, we allocate 15 on-chip memory bits for each KV pair as estimated before. We set the length of the optional probing sequence of Peacock hashing to be 2 [34]. Segment hashing [32] uses linear probing as the collision resolution policy, and maintains a on-chip memory bitmap to help ensure $O(1)$ insertion. BCHT [46, 63] uses 4 cells in a bucket. The memory allocation between buckets and linked lists for chaining hashing and d-left hashing are as described before in Section 6.2. For OMH_3, we tuned its parameters d, b, w_1, w_1', s, r to be $8, 6, 6, 9, 32, 0.5$, respectively (the tuning experiments are shown in Appendix 8.5). In the following memory access experiments, we set c to 1.02; in the PLF experiments, we set c to 1. SmartCuckoo is denoted by *SmartCK* in figures.

Insertion AMA as a function of load (Figure 9): As the result is almost the same on the four datasets, we only show

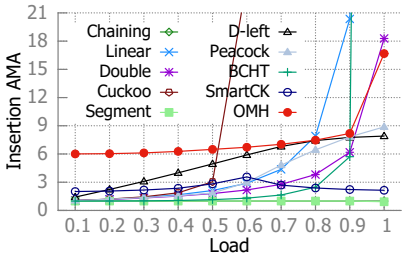

Fig. 9: Insertion AMA (DocWords).

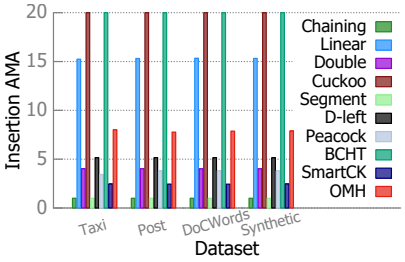

Fig. 10: Insertion AMA on four datasets.

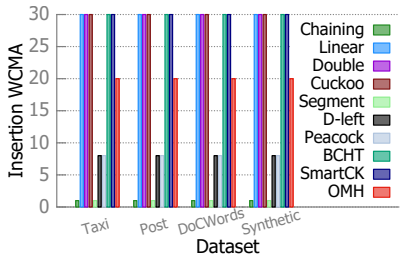

Fig. 11: Insertion WCMA on four datasets.

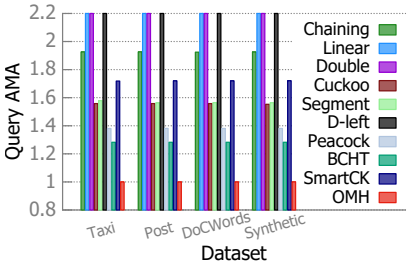

Fig. 12: Query AMA on positive set.

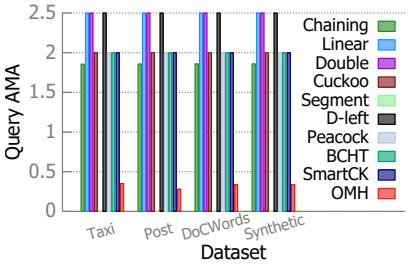

Fig. 13: Query AMA on negative set.

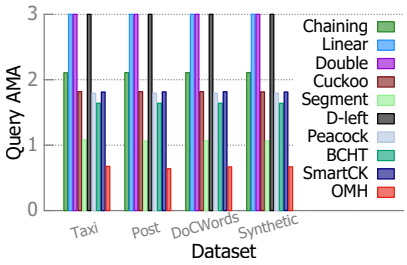

Fig. 14: Query AMA on mixed set.

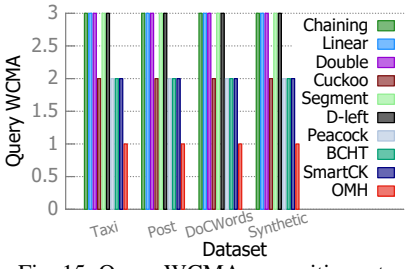

Fig. 15: Query WCMA on positive set.

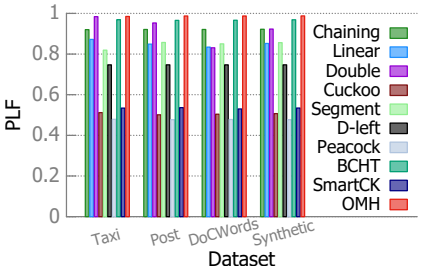

Fig. 16: PLF on four datasets.

the result on the Doc-Words dataset here. We observe that as the load increases from 10% to 100%, OMH_3 needs from 6.07 to 16.72 memory accesses on average, which is comparable to other hash schemes.

Insertion AMA for different datasets (Figure 10): We observe that OMH_3 needs 8, 7.87, 7.88, 7.88 memory accesses on average on the Taxi, Post, DocWord and Synthetic datasets, respectively, comparable to other hash schemes in the literature. The insertion AMA of OMH_3 is a bit higher than other multi-choice hashing schemes due to footprint recording.

Insertion WCMA for different datasets (Figure 11): We observe that OMH_3 needs 20 memory accesses in the worst case when performing insertion on all the four datasets. Again, the insertion WCMA of OMH_3 is a little bit higher than those of other multi-choice hashing schemes because it has to perform footprint recording. Meanwhile, the insertion WCMA of OMH_3 is much lower than that of cuckoo, smartcuckoo, linear, double, and BCHT.

Query Memory Access: In this set of experiments, we first insert a whole dataset into the hash tables, then run query sets. There are three kinds of query sets: 1) positive query set: the query set is exactly the same as the inserted dataset, 2) negative query set: the query set has no intersection with the inserted dataset, and 3) mixed query set: half of the query set is taken from the positive query set, and the other half from the negative query set.

Query AMA on Positive Query Set (Figure 12): As expected, we find that OMH_3 needs only 1 AMA on all four datasets. No other hash scheme can achieve this. The AMA of OMH_3 is 1.92, 5.48, 4.01, 1.55, 1.56, 15.32, 1.38, 1.28 and 1.65 times lower than that of chaining, linear, double, cuckoo, segment, d-left, peacock, BCHT and SmartCuckoo, respectively. This result validates the design of OMH_3.

Query AMA on Negative Query Set (Figure 13): We find that OMH_3 needs fewer AMA than all other hash schemes on all four datasets, except segment hashing. The AMA of OMH_3 is 5.3, 712.3, 145.7, 5.7, 27.8, 5.7, 5.7 and 5.7 times lower than the one of chaining, linear, double, cuckoo, d-left, peacock, BCHT and SmartCuckoo, respectively. Due to false positives, a KV pair in the negative query set may also have a matched fingerprint, incurring a memory access. Segment hashing has a lower AMA thanks to its proposed selective bloom filter algorithm [32].

Query AMA on Mixed Query Set (Figure 14): We find that OMH_3 needs fewer AMA than all other hash schemes on all four datasets, which is about 3.1, 206.3, 42.7, 2.7, 1.6, 12, 2.7, 2.4, 2.7 and 2.7 times lower than that of chaining, linear, double, cuckoo, segment, d-left, peacock, BCHT and SmartCuckoo, respectively.

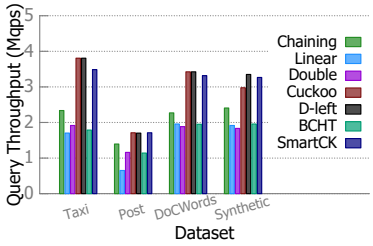

Fig. 17: Query throughput of some hash schemes on CPU.

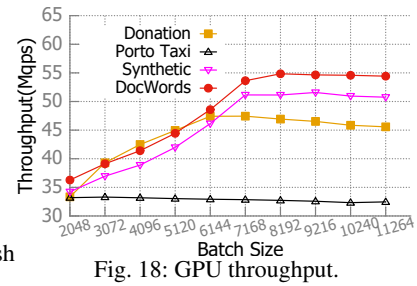

Fig. 18: GPU throughput.

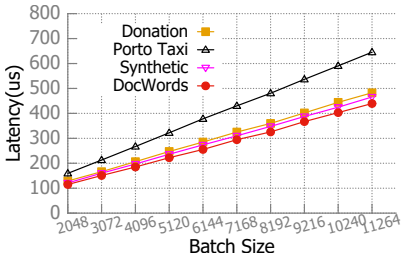

Fig. 19: GPU latency.

Query WCMP on Positive Query Set (Figure 15): We find that the WCMP of OMH_3 is exactly one on all four datasets, which is 12, 500, 500, 2, 62, 13, 2, 2 and 2 times lower than that of chaining, linear, double, cuckoo, segment, d-left, peacock, BCHT and SmartCuckoo, respectively.

PLF for different datasets (Figure 16): We find that the PLF of OMH_3 is the highest among the compared hash schemes, which is 1.07, 1.16, 1.07, 1.95, 1.15, 1.32, 2.07, 1.02 and 1.87 times higher than the one of chaining, linear, double, cuckoo, segment, d-left, peacock, BCHT and SmartCuckoo, respectively.

6.3 Performance on GPU/CPU Platform

Setup: The GPU used in our experiment is a GeForce GTX 1080, with a memory clock rate of 5 GHz, and a peak memory bandwidth of 320 GB/s. Our GPU has 20 multiprocessors, each has at most 2048 resident threads. The size of global memory available on GPU is 7.9GB.

Query performance: We present the query performance of OMH_3 in terms of throughput and latency. We use *Million Queries Per Second (Mqps)* and *Microseconds* as the units to measure the throughput and latency, respectively. As shown in Figure 18, the maximum query throughput OMH_3 achieves is around 55 Mqps, demonstrating its relevance to real-world scenarios. For reference and comparison, we also show the query speed of some hash tables implemented with pure CPU in Figure 17. It can be seen that with the assistance of GPU and our proposed techniques, OMH_3 outperforms these hash schemes up to an order of magnitude, meeting our design goal. As shown in Figure 19, although the batch latency increases linearly with the batch size, the throughput is not affected at all, thanks to the multi-streaming we have applied. The linearity of the delay completely depends upon the transfer time between/from the CPU and GPU, which reflects the speed of the PCIe Bus in our experimental platform.

6.4 Performance on FPGA Platform

Setup: We use Xilinx xcvu9p-flga2577-2-e (VU9P) as the target FPGA chip, which provides around 75.9Mb fast BRAMs and 270Mb URAMs. The slow memory bandwidth on an FPGA board is 3.2GB/s. We implement the OMH_3 query algorithm in HLS C and synthesize the hardware design using Xilinx Vivado HLS 2017.2. Then, we import the synthesized design into Xilinx Vivado 2017.2 to generate the bitstream and get the resource utilization and the performance report.

Performance: The resource utilization and the performance are shown in Table 3. The look-up tables (LUTs) and flip-flops (FFs) are consumed by the hash functions and other logic operations. The BRAMs and URAMs are used to store the FP-hash indexes, fingerprints, and other temporary variables. For all datasets, the clock frequency of the FPGA design achieves 200MHz. As described in Section 5.3, the performance of the query reaches 200Mqps. The utilization of LUTs and FFs in these three datasets are very close, because the FPGA design implements the same functionality for the same amount of hash functions and other logic operations. Taxi consumes less fast memory so we implement all the FP-indexes and fingerprints with only BRAM. Post and DocWords need more fast memory to store the FP-indexes and fingerprints with larger amounts of data, so we use both BRAM and URAM and balance the utilization of these two resources.

Table 3: Resource Utilization, frequency, and performance on FPGA platform.

	Available (VU9P)	Taxi	Post	DocWords
BRAM	75.9Mbits	12.6 %	10.8%	52.9%
URAM	270Mbits	0%	15.7%	46.2%
FF	2364480	5.8%	5.8%	6.0%
LUT	1182240	12.7%	12.8%	12.8%
Frequency	-	200MHz	200MHz	200MHz
Performance	-	200Mqps	200Mqps	200Mqps

7 Conclusion

Improvements to hash tables potentially bring significant benefits to many applications nowadays, given their popularity. The worst-case query performance is a key aspect of hash tables, and has not been sufficiently addressed in the literature. In this paper, we propose a novel hierarchical hash table called One-Memory Hash table, which achieves one off-chip memory access per query in the worst case, while supporting fast incremental updates and has no update failure. Our key technique is to build exclusive fingerprint arrays which report that at most one sub-table contains the incoming key. We implement our solution on GPU/CPU and FPGA platforms, and conduct head-to-head comparisons with 9 prior hash schemes on 4 datasets. Experimental results show that One-Memory Hash significantly outperforms previous work.

References

[1] C minimal perfect hashing library. http://cmph.sourceforge.net/.

[2] Source code of smartcuckoo provided by its authors. https:https://github.com/syy804123097/SmartCuckoo.

[3] Ultrascale architecture and product data sheet. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[4] "bags-of-words data set". http://archive.ics.uci.edu/ml/datasets/Bag+of+Words.

[5] "bob hash website". http://burtleburtle.net/bob/hash/evahash.html.

[6] "cuda toolkit documentation". http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[7] "post dataset". https://archive.ics.uci.edu/ml/machine-learning-databases/00210/.

[8] "proto taxi dataset". https://archive.ics.uci.edu/ml/machine-learning-databases/00339/.

[9] "source code of omh". https://github.com/onememoryhash/OMH.

[10] M. Mitzenmacher A. Kirsch and G. Varghese. Hash-based techniques for high-speed packet processing. In *In Algorithms for Next Generation Networks*, page 181, 2010.

[11] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of ACM symposium on Theory of computing*, pages 238–247. ACM, 1995.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *measurement and modeling of computer systems*, 40(1):53–64, 2012.

[13] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. *SIAM journal on computing*, 29(1):180–200, 1999.

[14] Ronald J Barber, Guy M Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard S Sidle, Gopi K Attaluri, Naresh K Chainani, Sam Lightstone, and David C Sharpe. Memory-efficient hash joins. *very large data bases*, 8(4):353–364, 2014.

[15] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Green Computing Conference and Workshops*, pages 1–8, 2011.

[16] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[17] Peter Boncz, Stefan Manegold, and Martin L Kersten. Database architecture optimized for the new bottleneck: Memory access. *very large data bases*, pages 54–65, 1999.

[18] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 281–294, Denver, CO, 2016. USENIX Association.

[19] Andrei Z Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. 3:1454–1463, 2001.

[20] J. Lawrence Carter and Mark N Wegman. Universal classes of hash functions (extended abstract). *Journal of Computer and System Sciences*, 18(2):106–112, 1977.

[21] Zbigniew J Czech, George Havas, and Bohdan S Majewski. Perfect hashing. *Theoretical Computer Science*, 1997.

[22] David J Dewitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael Stonebraker, and David A Wood. Implementation techniques for main memory database systems. *international conference on management of data*, 14(2):1–8, 1984.

[23] David Eppstein, Michael T Goodrich, Michael Mitzenmacher, and Manuel R Torres. Cuckoo filters for faster triangle listing and set intersection. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 247–260. ACM, 2017.

[24] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. ACM CoNEXT*, 2014.

[25] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 371–384, 2013.

[26] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[27] Amit Goyal, Hal Daumé III, and Suresh Venkatasubramanian. Streaming for large scale nlp: Language modeling. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 512–520. Association for Computational Linguistics, 2009.

[28] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *Network, IEEE*, 15(2):24–32, 2001.

[29] Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[30] Adam Kirsch and Michael Mitzenmacher. Simple summaries for hashing with choices. *IEEE ACM Transactions on Networking*, 16(1):218–231, 2008.

[31] Adam Kirsch, Michael Mitzenmacher, and George Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer, 2010.

[32] Sailesh Kumar and Patrick Crowley. Segmented hash: an efficient hash table implementation for high performance networking subsystems. pages 91–103, 2005.

[33] Sailesh Kumar, Jonathan Turner, and Patrick Crowley. Peacock hashing: Deterministic and updatable hashing for high performance networking. In *Proc. IEEE INFOCOM*, 2008.

[34] Sailesh Kumar, Jonathan S Turner, and Patrick Crowley. Peacock hashing: Deterministic and updatable hashing for high performance networking. pages 101–105, 2008.

[35] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[36] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. page 27, 2014.

[37] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *Nsdi*, pages 311–324, 2016.

[38] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. pages 311–324, 2016.

[39] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. *management*, 15(32):36, 2014.

[40] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. Dht routing using social links. In *IPTPS*, volume 3279, pages 100–111. Springer, 2004.

[41] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[42] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[43] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[44] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

[45] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[46] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. pages 1493–1508, 2015.

[47] Vijayshankar Raman, Gopi K Attaluri, Ronald J Barber, Naresh K Chainani, David Kalmuk, Vincent Kulandaisamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. Db2 with blu acceleration: so much more than just a column store. *very large data bases*, 6(11):1080–1091, 2013.

[48] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 622–629. Association for Computational Linguistics, 2005.

[49] Dharmapurikar Sarang, Krishnamurthy Praveen, and Taylor David E. Longest prefix matching using bloom filters. In *Proc. ACM SIGCOMM*, pages 201–212, 2003.

[50] David V Schuehler, James Moscola, and John Lockwood. Architecture for a hardware based, tcp/ip content scanning system [intrusion detection system applications]. In *Proc. IEEE HPI*, pages 89–94, 2003.

[51] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing.

ACM SIGCOMM Computer Communication Review, 35(4):181–192, 2005.

[52] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 135–146. ACM, 1999.

[53] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. Smartcuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 553–565, Santa Clara, CA, 2017. USENIX Association.

[54] R. L. Rives T. H. Cormen, C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, 2009.

[55] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[56] Benjamin Van Durme and Ashwin Lall. Online generation of locality sensitive hash signatures. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 231–235. Association for Computational Linguistics, 2010.

[57] Berthold Vöcking. How asymmetry helps load balancing. *Journal of the ACM (JACM)*, 50(4):568–589, 2003.

[58] Berthold Vocking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4):568–589, 2003.

[59] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. *Scalable high speed IP routing lookups*, volume 27. ACM, 1997.

[60] Gary R Wright and W Richard Stevens. *TcP/IP Illustrated*, volume 2. Addison-Wesley Professional, 1995.

[61] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 39–50. ACM, 2014.

[62] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. Buffalo: bloom filter forwarding architecture for large organizations. In *Proc. ACM CoNext*, pages 313–324. ACM, 2009.

[63] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *very large data bases*, 8(11):1226–1237, 2015.

[64] Dong Zhou, Bin Fan, Hyeontaek Lim, David G Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. Scaling up clustered network appliances with scalebricks. *acm special interest group on data communication*, 45(4):241–254, 2015.

[65] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. ACM CoNEXT*, 2013.

8 Appendix

8.1 Algorithm Pseudocode

Algorithm 1: Construction of OMH_1.

```
Function Construct(S):
    // S is a set of KV pairs
    MainTableConstruct(S)
    FingerprintArrayConstruct()

Function MainTableConstruct(S):
    for ⟨k_l, v_l⟩ in S do
        findempty ← False, chosenidx ← 0
        for i = 1 to d do
            pos ← h_i(k_l)
            if T_i[pos] is empty then
                insert ⟨k_l, v_l⟩ into T_i[pos]
                findempty ← True
                chosenidx ← i
                break
        if findemptybucket is True then
            for i = 1 to d do
                if i ≠ chosenidx then
                    pos ← h_i(k_l)
                    insert ⟨k_l, v_l⟩ into L_i[pos]
        else
            if stash is not full then
                insert ⟨k_l, v_l⟩ into stash
            else
                InstantTableExtension(⟨k_l, v_l⟩)

Function FingerprintArrayConstruct():
    for i = 1 to d do
        for j = 1 to m do
            if T_i[j] contains a pair ⟨k_l, v_l⟩ then
                ExclusiveFingerprinting(i,j)
```

Algorithm 4: Query of OMH_1

```
Function Query(k_l):
    if k_l is in the stash then
        v ← the corresponding value of k_l
        return v
    else
        count ← 0        // number of matched buckets
        idx ← 0          // index of the matched bucket
        for i = 1 to d do
            pos ← h_i(k_l)
            fpidx ← t_i[pos].id
            fp ← t_i[pos].fp
            if h^F_{fpidx} == fp then
                count ← count + 1
                idx ← i
        if count == 1 then
            pos ← h_{idx}(k_l)
            if T_{idx}[pos] contains k_l then
                v ← the corresponding value of k_l
                return v
            else
                return NULL
        else
            return NULL
```

Algorithm 5: Instant Table Extension of OMH_1

```
Maintain a global variable ExtNum, which records the number
  of extensions. Initialize ExtNum ← 0.
Function InstantTableExtension(⟨k_l, v_l⟩):
    ExtNum ← ExtNum + 1
    allocate a new sub-table T_{d+ExtNum}
    allocate a new fingerprint array t_{d+ExtNum}, with each
      bucket having two more bits
    idx ← ExtNum%d, make T_{d+ExtNum} and t_{d+ExtNum} be
      associated with h_{idx}
    pos ← h_{idx}(k_l), insert ⟨k_l, v_l⟩ into T_{d+ExtNum}[pos]
    ExclusiveFingerprinting(d + ExtNum, pos)
    for i = 1 to d + ExtNum − 1 do
        pos ← h_i(k_l)
        if i <= d then
            insert ⟨k_l, v_l⟩ into L_i[pos]
        if T_i[pos] contains a pair ⟨k'_l, v'_l⟩ then
            fpidx ← t_i[pos].id
            fp ← t_i[pos].fp
            if h^F_{fpidx}(k_l) == fp then
                ExclusiveFingerprinting(i,pos)
```

Algorithm 2: Exclusive Fingerprinting

```
ExclusiveFingerprinting(i,j)  get the pair ⟨k_l, v_l⟩ in T_i[j]
for q = 0 to 2^{w_1} − 1 do
    adequate ← True
    for ⟨k'_l, v'_l⟩ in L_i[j] do
        if h^F_q(k_l) == h^F_q(k'_l) then
            adequate ← False
            break
    if adequate is True then
        t_i[j].id ← q
        t_i[j].fp ← h^F_q(k_l)
        return
if stash is not full then
    insert ⟨k_l, v_l⟩ into stash
else
    InstantTableExtension(⟨k_l, v_l⟩)
```

Algorithm 3: Insertion of OMH_1

```
Function Insert(⟨k_l, v_l⟩):
    findempty ← False, chosenidx ← 0
    for i = 1 to d do
        pos ← h_i(k_l)
        if T_i[pos] is empty then
            insert ⟨k_l, v_l⟩ into T_i[pos]
            findempty ← True
            chosenidx ← i
            ExclusiveFingerprinting (i,pos)
            break
    if findempty is True then
        for i = 1 to d do
            if i ≠ chosenidx then
                pos ← h_i(k_l)
                insert ⟨k_l, v_l⟩ into L_i[pos]
                if T_i[pos] contains a pair ⟨k'_l, v'_l⟩ then
                    fpidx ← t_i[pos].id
                    fp ← t_i[pos].fp
                    if h^F_{fpidx}(k_l) == fp then
                        ExclusiveFingerprinting(i,pos)
    else
        if stash is not full then
            insert ⟨k_l, v_l⟩ into stash
        else
            InstantTableExtension (⟨k_l, v_l⟩)
```

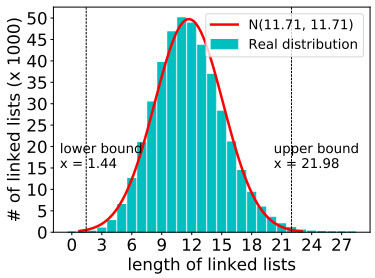

Figure 20: The bar shows the distribution of l_1, with $d = 8, b = 6, r = 0.5$ and $c = 1.02$ after inserting a dataset containing 5 million KV pairs. The line shows the normal distribution $N(11.71, 11.71)$, obtained using theorem 1. The vertical line indicates the upper and lower bound of the confidential interval 1.44 and 21.98. The first sub-table has 425000 linked lists, and around only 0.4% (1829) of linked lists have length greater than 21.

8.2 Proof of Theorems

Proof of Lemma 1 Since each KV pair has a probability of $\frac{d}{n}$ to be inserted into a certain bucket, and there are in total n KV pairs, it naturally derives that the length l of a linked list of any bucket follows the *binomial distribution*, $l \sim B(n, \frac{d}{n})$. Therefore, it can be easily computed that the expectation of l is $E(l) = n \times \frac{d}{n} = d$, and the variance is $Var(l) = n \times \frac{d}{n} \times (1 - \frac{d}{n}) = d - \frac{d^2}{n}$. As $n \to \infty$, the variance $Var(l) \to d$. According to the *Central Limit Theorem*, when n is very large, the binomial distribution $B(n, \frac{d}{n})$ approximates the normal distribution $N(d, d)$. According to the *three-sigma principle* of the normal distribution, we can get the probability that l falls in the range of $[d - 3\sqrt{d}, d + 3\sqrt{d}]$ is 99.73%.

Proof of Lemma 3 Since there are w_2 bits in the fingerprint field, the length of a fingerprint is 2^{w_2}. Therefore, when using a single FP-hash, the probability of obtaining an adequate fingerprint is $q = (1 - \frac{1}{2^{w_2}})^l$. We need k FP-hashes when all the first $k-1$ FP-hashes fail to compute an adequate fingerprint and the last k^{th} FP-hash succeeds. This means that x follows a geometric distribution with success probability of q, which directly gives: $P(x = k) = (1 - q)^{k-1} \times q$.

Proof of Theorem 1 First, it can be easily computed that $m_i = \frac{cn(1-r)r^{i-1}}{b(1-r^d)}$. Then, from the analysis above, we can similarly get that l_i follows the binomial distribution $B(n, \frac{1}{m_i})$. Therefore, the expectation of l_i is $E(l_i) = n \times \frac{b(1-r^d)}{cn(1-r)r^{i-1}} = \frac{b(1-r^d)}{c(1-r)r^{i-1}}$. The variance $Var(m_l)$ can be computed as $n \times \frac{b(1-r^d)}{cn(1-r)r^{i-1}} \times (1 - \frac{b(1-r^d)}{cn(1-r)r^{i-1}}) \approx \frac{b(1-r^d)}{c(1-r)r^{i-1}}$. As n grows very large, we can again use the normal distribution $N(\frac{b(1-r^d)}{c(1-r)r^{i-1}}, \frac{b(1-r^d)}{c(1-r)r^{i-1}})$ as an approximation of the binomial distribution. Applying the *three-sigma principle*, we get that with probability 99.73%, l_i falls in the range of $[\frac{b(1-r^d)}{c(1-r)r^{i-1}} - 3\sqrt{\frac{b(1-r^d)}{c(1-r)r^{i-1}}}, \frac{b(1-r^d)}{c(1-r)r^{i-1}} + 3\sqrt{\frac{b(1-r^d)}{c(1-r)r^{i-1}}}]$.

Proof of Theorem 2 With x FP-hashes, the probability of obtaining a proper fingerprint is:

$$\begin{aligned} &\sum_{i=1}^{x}(1-q)^{i-1}q = q\sum_{i=1}^{x}(1-q)^{i-1} \\ =&q \times \frac{1-(1-q)^x}{q} = 1-(1-q)^x \end{aligned} \tag{5}$$

Solving $(1-q)^x \geqslant \alpha$, we get $x \geqslant \frac{ln(1-\alpha)}{ln(1-q)}$.

8.3 Validation of Theorem 1

We give a concrete example to validate theorem 1. In the experiments described later in Appendix 8.5, we have tuned r to be 0.5, d to 8, b to 6, and c to 1.02. Applying theorem 1 with these configurations, it gives that the length of a linked list of any bucket in the first sub-table l_1 approximates the distribution $N(11.71, 11.71)$, and with probability 99.73%, l_1 falls in the range $[1.44, 21.98]$. Figure 20 verifies this result.

8.4 Comparison of the Three Versions of OMH

In this part, we compare the three versions of One-Memory Hash. The main metric is PLF. For OMH_2 and OMH_3, we set the number of cells in a bucket b to 6.

PLF of three versions (Figure 21 and 22): We observe that when using the same number of sub-tables, OMH_2 and OMH_3 have a much higher PLF than OMH_1. The increase in PLF shows that our improvements made in OMH_2 to achieve a higher load factor work well. We also observe that the PLF of OMH_2 and OMH_3 are very close, showing that the hash sharing technique has little impact on PLF of OMH_3. Note that to achieve a PLF over 90%, OMH_1 needs more than 30 sub-tables, which means it has more than 30 copies for each KV pair; OMH_2 needs only 6 sub-tables, therefore only 5 copies, which is more reasonable; OMH_3 needs 6 sub-tables as OMH_2 does, but the hash sharing technique reduces the number of copies from 5 to 2.

8.5 Tuning of Parameters of OMH_3

Here we measure how much the parameters of OMH_3 affect its performance. We focus on the following: 1) the number of sub-tables d and the number of cells in a bucket b, 2) the number of bits in the index field w_1 and the fingerprint field w_2 of the first half of fingerprint arrays, and their counterparts w_1' and w_2' in the second half. In the following experiments, we fix the number of entries in the stash s at 32, and the common ratio r at 0.5.

Effect of d and b on PLF: In this set of experiments, we set c to 1.0 and sample values of d ($\{4, 6, 8\}$) and b ($\{4, 5, 6, 7, 8\}$).

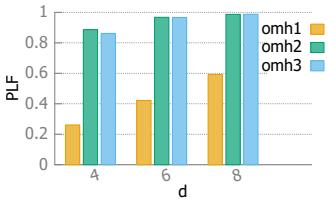

Fig. 21: PLF of three versions of our algorithm.

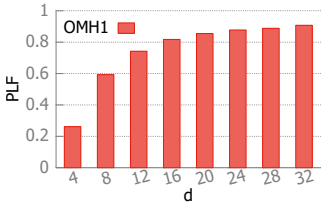

Fig. 22: PLF of OMH_1 as a function of d.

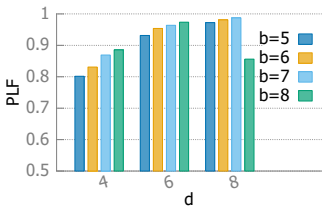

Fig. 23: PLF of OMH_3 as a function of d and b.

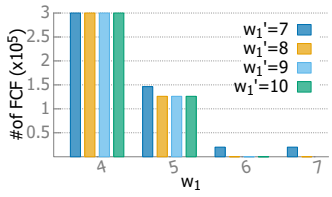

Fig. 24: # of FCF as a function of w_1 and w_1'.

For each combination of d and b, we do 10 independent runs, each on a dataset containing 10 billion KV pairs, and report the minimum PLF.

PLF as a function of d and b (Figure 23): We observe that the PLF increases as d and b increase, except for a drop when (d,b) is $(8,8)$. The reason for the monotonically increasing relationship is that since $d \cdot b$ represents the number of candidate cells a KV pair has, a larger $d \cdot b$ implies more candidate cells and therefore a higher PLF. The reason for the drop of PLF at $(d,b) = (8,8)$ is that when $d \cdot b$ is too large, the length of each sub-table is too small, and the higher number of collisions into the same bucket overcompensate the increased number of candidate cells. Therefore, we choose $(8,6)$ as the final choice for (d,b) in OMH_3.

Impact of w_1, w_2, w_1' and w_2' on number of Fingerprint Construction Failures (FCF): In this set of experiments, we set d, b, and c to 8, 6, and 1.02, respectively. We set $w_1 = w_2$ and $w_1' = w_2'$, *i.e.*, the index and fingerprint fields have the same number of bits. We vary w_1 and w_1' in the ranges $\{4,5,6,7\}$ and $\{7,8,9,10\}$, respectively. For each combination of w_1 and w_1', we perform 10 runs on 10 different datasets, each contains 10 million KV pairs, and report the maximum number of FCF.

FCF as a function of w_1, w_1' (Figure 24): As expected, we find that the number of FCF decreases as w_1 and w_1' increase. Note that when w_1 is small, most of the FCF occur in the first half of sub-tables, explaining the very limited impact of w_1' on the results. When w_1 is above 5, most of the FCF occur in the second half of sub-tables, so that the effect of w_1' becomes more explicit. With $w_1 = 6$ and $w_1' = 9$, the number of FCF goes down to 0, *i.e.*, no failure happens. Combined with the fact that $c = 1.02$ seems large enough to guarantee that OMH_3 needs no dynamic memory allocation in practice, we can estimate that OMH_3 uses roughly $(w_1 + w_1')/c \approx 15$ bits in on-chip memory for each KV pair. We therefore choose $6,9$ as the final value of w_1, w_1' in OMH_3.