One Slow Memory Access Hash Table*

Tong Yang[1], Yufei Wang[1], Haowei Zhang[1], Yunlong Wang[2]
Guojie Luo[1], Jiaxi Zhang[1], Xiaoming Li[1], Steve Uhlig[3]
[1]Department of Computer Science, Peking University, China
[2]Department of Electronic Engineering, Tsinghua University, China
[3]School of electronic engineering and computer science, Queen Mary University of London, UK

ABSTRACT

Existing hash tables cannot guarantee one memory access per query in the worst case without update failures. In this paper, we propose the OMH table, which is deployed in the fast-slow hierarchical memories (e.g., GPU/CPU). Our hash table builds exclusive fingerprints in the fast memory to guide query in the slow memory, and it needs one slow memory access per query in the worst case, supports fast incremental updates, and guarantees no update failure.

1 INTRODUCTION

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 address lookups [14], packet classification [6], MAC table lookup and more [3–5]. Most real-world workloads that hit hash tables are *read-heavy* [1]: most of the operations to hash tables are queries. The main shortcoming of hash tables is that they cannot guarantee $O(1)$ time complexity for queries in the worst case. In many applications, it is critical for the worst-case query time to be well-bounded, *e.g.*, a switch in a data center 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.

A large number of works have studied how to reduce the number of memory access per query in the worst case. 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 [9] and its variants, and Perfect hashing. Cuckoo hashing needs two memory accesses per query in the worst case, however, this bounded worst-case complexity comes at the cost of slow updates and the possibility of update failures. Perfect Hashing achieves one memory access per query through special designed hash functions, but does not support fast incremental updates. The Second kind of solutions achieve in average around 1 slow memory access per query by leveraging the fast-slow hierarchical memories, Typical schemes include Segmented Hashing [7] and Peacock Hashing [8]. Typically, the access time of fast memory like SRAM is compared negligible to that of slow memory like DRAM. Therefore, the second kind of solutions focus on using auxiliary data structure (*e.g.*, fingerprints, Bloom filters (BFs) [2]) in the fast memory to reduce the number of accesses of slow memory. However, both BFs and fingerprints have false positives, which means even with their assistance, there could still be multiple slow memory accesses in the worst case.

In summary, existing works 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 requires at most one memory access per query in the worst case, and supports fast incremental update without update failures. The *key idea* is to compute multiple fingerprints for a KV pair, and choose one so that each KV pair has only one matched fingerprint.

2 ONE MEMORY HASH TABLE

2.1 Data Structure

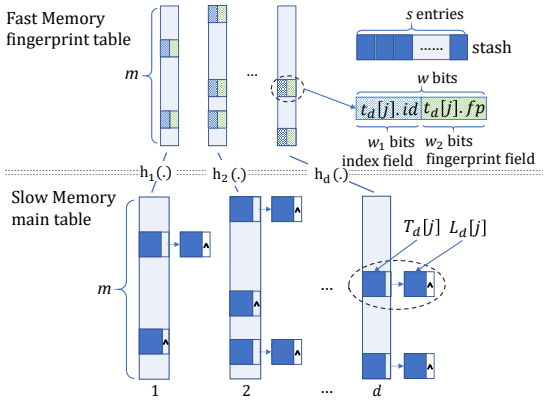


Figure 1: Data structure of OMH.

As shown in Figure 1, OMH consists of three components: A fingerprint table, a stash in fast memory, and a main table in slow 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 $T_i[j]$ 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 can store one KV pair. A bucket $T_i[j]$ in main table together with its corresponding bucket $t_i[j]$ in 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} 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 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.

*Co-primary authors: Tong Yang and Yufei Wang. This work is supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (973 Program, 2014CB340405), NSFC (61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences.

2.2 Operations

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

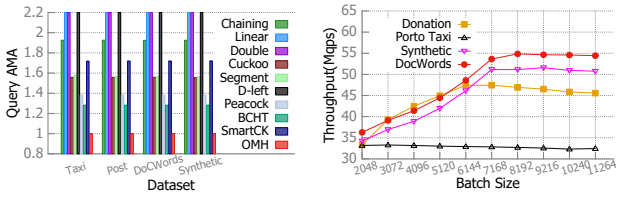
1) Main table construction: For each KV pair $\langle k_l, v_l \rangle$ in $\mathbb{S}$, OMH maps it to d buckets in the main table: $T_1[h_1(k_l)]$, $T_2[h_2(k_l)]$, ..., $T_d[h_d(k_l)]$. We call them the *candidate buckets* of $\langle k_l, v_l \rangle$. OMH checks them sequentially and stops when an empty bucket is found. During this process, there are three possible cases: **1)** If an empty bucket is found, OMH inserts $\langle k_l, v_l \rangle$ into it. For the other $d-1$ unchosen candidate buckets, OMH inserts $\langle k_l, v_l \rangle$ into the linked lists of all of them. This process is called **footprint recording**. **2)** If no empty bucket is found and the stash is not full, OMH inserts $\langle k_l, v_l \rangle$ into the stash. **3)** If no empty bucket is found and the stash is full, then OMH activates **instant table extension**: it adds a new pair of tables T_{d+1} and t_{d+1} associated with $h_1(.)$, which have the same structure and size as T_1 and t_1. OMH inserts $\langle k_l, v_l \rangle$ into $T_{d+1}[h_1(k_l)]$, and $L_1[h_1(k_l)]$, $L_2[h_2(k_l)]$, ..., $L_d[h_d(k_l)]$ due to footprint recording. Note that each previously inserted KV pair $\langle k'_l, v'_l \rangle$ should be copied to the linked list $L_{d+1}[h_1(k'_l)]$ due to footprint recording, however, the reuse of $h_1(.)$ avoids such heavy copy because $L_{d+1}[j]$ $(1 \leqslant j \leqslant m)$ can be obtained as $L_1[j] \cup T_1[j]$.

2) Fingerprint table construction: After the construction of main table, for each bucket $T_i[j]$ that stores a KV pair $\langle k_l, v_l \rangle$, OMH 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 $L_i[j]$. OMH utilizes a technique called **exclusive fingerprinting** to choose the FP-hash. Let $\langle k_l, v_l \rangle$ be a KV pair stored in $T_i[j]$, $L_i[j]$ be the linked list of $T_i[j]$, S be the set of keys in $L_i[j]$. An FP-hash h_q^{F} is adequate if and only if: $h_q^{\mathrm{F}}(k_l) \neq h_q^{\mathrm{F}}(s), \forall s \in \mathrm{S}$. OMH tries the 2^{w_1} FP-hashes sequentially, and if it finds an adequate h_q^{F}, OMH 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, OMH inserts it into stash to avoid the computation of a fingerprint. If the stash is full, instant table extension is performed.

2.2.2 Insertion: To insert a KV pair $\langle k_l, v_l \rangle$, OMH locates its d candidate buckets in main table, checks them sequentially, and stops if an empty bucket is found. During this process, there are three possible cases: **1)** If an empty bucket is found, OMH inserts $\langle k_l, v_l \rangle$ into it and all linked lists of the other $d-1$ unchosen buckets. Then OMH performs exclusive fingerprinting for $\langle k_l, v_l \rangle$. For the other $d-1$ unchosen candidate buckets, if they contain a KV pair $\langle k'_l, v'_l \rangle$ and record a FP-hash h_q^{F}, OMH checks if h_q^{F} is still adequate, because $\langle k_l, v_l \rangle$ is newly added to the linked list. If h_q^{F} becomes inadequate,*i.e.*, $h_q^{\mathrm{F}}(k_l) = h_q^{\mathrm{F}}(k'_l)$, OMH applies the exclusive fingerprinting for $\langle k'_l, v'_l \rangle$. **2)** If no empty bucket is found and the stash is not full, then OMH inserts $\langle k_l, v_l \rangle$ into the stash. **3)** If no empty bucket is found and the stash is full, then the instant table extension mechanism is activated.

2.2.3 Query: To query $\langle k_l, v_l \rangle$, OMH first checks the stash. If k_l is found, OMH reports the value and query ends. Otherwise, OMH 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 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: **1)** If there is more than one matched fingerprints, or there is no matched fingerprint, OMH reports a failure, meaning the queried key does not exist. **2)** If there is only one matched fingerprint, OMH 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 reports the value; if k_l is not found, OMH reports a failure. In both situations, there is *at most one slow memory access*.

2.2.4 Deletion: To delete a KV pair $\langle k_l, v_l \rangle$, OMH first queries to find it, and then removes it.



(a) $QueryAMA$ (b) $GPUthroughput$

Figure 2: Evaluations of OMH

3 EVALUATION

We use three real-world datasets and one synthetic dataset. We compare OMH with chaining [12], linear [**?**], double[**?**], cuckoo [9], d-left [13], peacock [8], segment [7], BCHT [10] and smartcuckoo hashing [11] in terms of query Average Memory Access (AMA). We also implement OMH on a fast-slow hierarchical memory platform: the GPU/CPU platform, and report its query throughput in Mqps (Million query per second).

As shown in Figure 2(a), we find that OMH needs only 1 AMA on all four datasets. No other hash scheme achieves this. The AMA of OMH 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. As shown in Figure 2(b), the maximum query throughput OMH achieves is around 55 Mqps, demonstrating its relevance to real-world scenarios.

REFERENCES

[1] Berk Atikoglu and Yuehai. et Xu. 2012. Workload analysis of a large-scale key-value store. *measurement and modeling of computer systems* 40, 1 (2012), 53–64.

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

[3] Haipeng Dai, L Meng, and Alex X Liu. 2018. Finding persistent items in distributed, datasets. In *IEEE INFOCOM*.

[4] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. 2016. Finding persistent items in data streams. *Proceedings of the VLDB Endowment* 10, 4 (2016), 289–300.

[5] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. 2016. Noisy Bloom Filters for Multi-Set Membership Testing. In *ACM SIGMETRICS*. 139–151.

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

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

[8] Sailesh Kumar and et Turner, Jonathan. 2008. Peacock hashing: Deterministic and updatable hashing for high performance networking. In *IEEE INFOCOM*.

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

[10] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. (2015), 1493–1508.

[11] Yuanyuan Sun and Yu Hua .et. [n. d.]. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. In *2017 USENIX ATC*.

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

[13] Berthold Vocking. [n. d.]. How asymmetry helps load balancing. *J. ACM* 50, 4 ([n. d.]).

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