Thesis Proposal: Supersuper Fast and Efficient Hashing

Chenyao Lou

1 Introduction

The Hash table is a very crucial component for systems. For example, the load balancer uses the hash table to decide the associated backend server for specified connections. It can store the Forwarding Information Base (FIB) for network applications [3] like SDN. The lookup of the FIB is the biggest bottleneck either for software packets forwarding or hardware packets forwarding. Thus, developers desire a high-performance hash table for such tasks.

Besides, the space efficiency is also an important metric for hash tables. The space efficient hasing has two benefits. First, the small query structure can fit into the CPU cache, which is ten times faster than the main memory (DRAM). Thus the lookup performance is accelerated further with the same hardware. Second, it can store more items for memory-constanted tasks, to imporve the capacity of a single machine.

Therefore, we propose a new space efficient hash table SSFE (Supersuper Fast Hashing) design with the high lookup performance. The basic functionality of a hash table is that, given a set of tuples $\{(k_0, v_0), (k_1, v_1), ..., (k_{n-1}, v_{n-1})\}$, the hash table can give v_i for the question $q(k_i)$. There are variations of this definition: For a $k \notin \{k_0, ..., k_{n-1}\}$, (1) the hash table returns $\perp$ (key is not existing), or (2) returns a random $v \in \mathcal{V}$. SSFE is designed for the second definition, i.e. it returns a random value when the query key has never been seen before.

SSFE consists of two structures, the *Query Structure* and the *Maintenance Structure*. The query structure has less memory footprint than the maintenance structure, and only serves for the lookup. It does not store any keys of tuples. That is why quering a non-existing key results in a random value. The maintenance structure is used to store all tuples so that it supports update operations. This seprating structures design fully utilize the CPU cache since the query structure are more compact by discarding keys. The system can update the query structure on a dedicated core without sacrificing the lookup performance.

In section 2 we will disscuss the full design of SSFE. We compare SSFE with existing works in section 3.

2 Design

There are two main structures for SSFE, the *Query Structure* and the *Maintenance Structure*.

Query Structure The *Query Structure* only serves for lookups. The tuples are partitioned into groups. SSFE uses a hash function to associate a key with a group. Each group contains n tuples. In pratical, $n \leq 300$. Physically, a group is an array x with length m, the result of a query $q(k_i)$ is

$$v'_i = x[h_1(k_i)] \oplus x[h_2(k_i)] \oplus x[h_3(k_i)]$$

h_1, h_2 and h_3 are 3 different hash functions, the ranges are all $[0, m)$. Thus, the hard part is to find a value setting of x satisfying v'_i is equal to v_i for all i.

Maintenance Structure The maintenance structure explicitly maintains tuples of each group. It can naively use `std::vector` to maintain that since updates are not frequently. The *Maintenance Structure* use these explicit information to construct x for all groups.

2.1 Construction

WLOG, we only consider how to construct the x for one group. Suppose there are n tuples for this group, and the length of x is m. The tuples are $\{(k_0, v_0), (k_1, v_1), ..., (k_{n-1}, v_{n-1})\}$. Frist, we construct a matrix A_{n*m}. Each row of A represents a key. Initially, the A is all zeros. For k_i, we increase $A_{i,h_1(k_i)}, A_{i,h_2(k_i)}$ and $A_{i,h_3(k_i)}$ with 1.

According the definition, we want $Ax = (v_0, v_1, ..., v_{n-1})^T = v$. Therefore, $x = A^T v$. Note that all addition operations are *xor* operations in above steps.

The construction may fail. We do experiments shows that it needs 1.3 times to find a feasible A if the A is randomly generated, when $m = 1.1n$. Thus, we can choose h_1, h_2, h_3 among a hash family. Then, the A can be thought as a pesudo-random matrix. This needs SSFE to store the index of h_1, h_2, h_3.

2.2 Group Size

There are two alternative ways to decide the size of groups:

- Fix the group size by a constant, e.g., 256. This makes the group are aligned to the cache line size. Also, SSFE can compute the position of a group without readding any memory. A problem is that the associated tuples are not even among groups. SSFE needs more groups to guarantee the group with most tuples has enough capacity. A workaround is to add an inidcator to the group, to show whether it has enough space. If it is not, it follows a pointer to a dedicated memory area to extend the group. The cost per key is $1.3 size(value)$ bits.
- Set the group size to $m = 1.1n$, n is the tuples associated to this group. The program needs a pointer array to index the position of each group,

	Bits Per Key	False Positive
Cuckoo[3]	$1.05(size(value) + size(key))$	No
SepSet[2]	$2 + 2size(value)$	Yes
Othello[1]	$\leq 4size(value)$	Yes
SSFE	$1.3size(value)$ or $1.1size(value)$	Yes

Table 1: Comparsion between SSFE and existing hashing algorithms.

since sizes are varying. However, the group can be more compact, the cost per key is $1.1size(value)$ bits.

3 Related Work

Table 1 shows the comparsion among exiting hashings. It shows SSFE has the best space efficiency when the size of the value is relatively small.

Cuckoo Hashing [3] Cuckoo has a fixed group size 4. Each key are mapped into two possible groups. So that, while one group is full, the tuple can store into the another group. The utilization rate can be 95%. It stores keys into the group. That prevents the false positive but increase the space overhead.

SepSet [2] SepSet divides tuples into small groups. Each group contains 16 tuples in average. It uses 0.5 bit per key to divide tuples more evenly. In each group, SepSet brute forcely find a hash function that can map all keys into the correct values.

Othello Hashing [1] Othello does not partition keys. It uses two array a and b. The value of a query is $a[h_1(k_i)] \oplus b[h_2(k_i)]$. Othello abstracts those arrays as a bipartite graph, $(h_1(k_i), h_2(k_i))$ is a edge between two sides. When the bipartite group is acyclic, it can trivally set array values to find a feasible setting. They claim that the probability of the group is acyclic is larger than 0.5 when total bits are $4n$, so they only need to try $O(1)$ times to find a possile hash function.

4 Plans

- Implement SSFE with two group allocation strategies.
- Compare the performance with different strategies.
- Investigate the fast and pratical way to accelerate the construction phase.
- Compare the performance with existing hashings.

References

[1] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. Memory-efficient and ultra-fast network lookup and forwarding using othello hashing. *IEEE/ACM Transactions on Networking*, 26(3):1151–1164, 2018.

[2] 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. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 241–254. ACM, 2015.

[3] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.