Hash Table Accelerator

Yaodong Sheng
Lehigh University
Bethlehem, PA, U.S.A
yas616@lehigh.edu

Anlan Yu
Lehigh University
Bethlehem, PA, U.S.A
any218@lehigh.edu

Zhankai Feng
Lehigh University
Bethlehem, PA, U.S.A
zhf218@lehigh.edu

Abstract

Hash table is a widely used data structure in many applications. However, hash collision cannot be avoided especially in large data application. Even though chain-based and probing-based hash tables are applied to solve the hash collision problem, the efficiency still cannot be guaranteed because of lower spatial locality for chain-based hash table and traverse problem if the key is not in the table for probing-based hash table. In this case, following instructions dependent on this instruction will be delayed and the overall execution efficiency cannot be guaranteed. In this project, we design a hash lookaside buffer (HLB) for hash value lookup to increase the throughput as well as decrease the latency of operations on hash table. Used as a buffer dedicated for hash table, HLB can store 64KB key-value pairs. For each hash lookup, 64 keys can be compared simultaneously, which will highly decrease the execution time. If the entries of HLB are not enough, memory hash table (mem_table) is used as a supplement to HLB. According to the using frequency, the key-value pairs in mem_table will be determined whether to be moved into HLB or not. A victim key-value pair will be replaced in HLB correspondingly based on round robin method. Totally four instructions are designed to support the functions of HLB. Based on the instructions, 8 APIs are also provided. Experimental results shows that our proposed HLB structure achieves 3x to 4x efficiency compared with traditional probing-based hash table structure.

CCS Concepts

• **Computer Architecture** → **Hash Accelerator**.

Keywords

hash table, accelerator, table lookup

ACM Reference Format:
Yaodong Sheng, Anlan Yu, and Zhankai Feng. 2018. Hash Table Accelerator. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 Background

1.1 Hash Function

Hash function is defined as the function that can be used to map data from arbitrary size to fixed-size values. The results of hash function are used to index a fixed-size table which is called hash table. The process of using the result of hash function to index a hash table is called hashing or scatter storage addressing. [1]

Hash functions and hash tables are used in data storage and retrieval applications. They can access data in a small and nearly constant time per retrieval meanwhile storage space only fractionally greater than the space that all data actually occupy.

1.2 Hash Collision

All the hash functions have the following basic characteristics: If two indexes (output of a hash function) are not the same, then their two keys (input of a hash function) are not the same. This feature is the reason why the hash function is certainty. On the contrary, if two indexes are the same, two input values are likely to be the same, but may also be different. This situation is called hash collision. Good hash functions rarely have hash collisions in the input field. In hash table and data processing applications, if there is no method to suppress collisions, it's hard to find the hash value because collision means that computer needs to store this key-value pair in other location or store several key-value pairs in one index. Both of the methods would make computer spent more time to find the key-value pair.

1.3 Hash Table

Hash table is one of the main applications of hash functions, which allows users to quickly find data indexed by keyword. Hash table is the data structure that given a key, one can directly access to the storage location in memory. It can be regarded as a dictionary. Suppose that there is a batch of word information. If the data is stored arbitrarily, it would be time consuming to find a word information because the user has to check the word one by one until finally find the matched word. But if the word information is stored by the words' initial, the user will easily find the word information according to the initial. Hence, using hash table is efficient.

1.3.1 Hash Table Organization There are several commonly used hash functions to organize hash table.

Direct addressing : $hash(k) = k$ or $hash(k) = a * k + b$, where a, b are constants.

Numerical analysis : Analysis a set of data. For example, analysis a set of the employees' date of birth. then it can be found that the first few digits of birth date are more likely to be the same. In this case, the chances of collision will be very large. But the last few digits of birth date are more likely to be different. If the last part of

Unpublished working draft. Not for distribution.

the numbers is used to form an index, the collision rate would be significantly lower than using the first few digits to form a index. So numerical analysis is important in some cases to decide the hash function and achieve lower collision.

Middle-square method : When it is impossible to determine which bits of a key are distributed uniformly, the key can be first squared and then take the middle bits of the squared value as the index. The reason is that the middle digits after the square is related to each of the keywords. So different keywords will produce different indexes with high probability.

Folding method : The key is divided into several parts with the same size of digits (the last part can have different size), and the sum of these parts (carry off) is taken as the index.

Remainder method : The index is $hash(k) = k \ mod \ p$. It is not necessary to be limited into directly taking the modulus of the key. Instead it can be taken modulus operation after taking medium operation of folding square.

1.3.2 Mechanism For Handling Collision Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. Therefore, it's necessary to implement some collision resolution strategies to handle such events. Some common strategies are described below. All these methods require that the keys are stored in the table, together with the associated values.

Chain-based hash: For this method, each bucket is independent, and has a list of entries with the same index. The time for hash table operations is the time to find the bucket and the time for comparing the elements in this list.

Probing-based hash : For probing-based hash, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined starting with the slot indicated by hash function result and proceeding in some probe sequence, until an unoccupied slot is found. So the basics of this method is that if the original slot is occupied, then the data would be stored into another slot towards the end of this table, such as stored into next lost. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.

2 Introduction

2.1 The Problem Aim to Solve

In reality, the collisions cannot be avoided with vast amounts of data. As mentioned in background, there are two main methods to handle the collision, which are probing-based and chain-based. However both of them have drawbacks. Chain-based hash applications could cause pointer chase problems, thus reduce spatial locality and cause long memory reference time as a result. Probing-based application would traverse the whole hash table if a key is not in the hash table, which wastes a lot of time and reduces overall efficiency. In the big data applications, their drawbacks are magnified a lot and hashing would lose its advantage of finding data quickly. Obviously it will decrease the performance of hash applications. Though some of the hash accelerators are proposed to solve this problem, probing-based and chain-based hash still cannot be avoided. As a result, long memory stall and miss predictive path still keep the hash table away from efficiency.

2.2 Literature Review

The paper "Leveraging caches to accelerate hash tables and memorization" [1] discusses two inefficient points when hash table is implemented in conventional systems. First is poor core utilization, for a hash operation it includes a series of data dependent branch instructions because it has to identify whether the key is the same. These kinds of instructions reduce the utilization of CPU due to data dependency. Second is poor spatial locality, according to the property of hash operation, it's very likely that two keys are mapped into different place in memory. Thus if there are some keys accessed frequently, the cache would store some other key-value pairs which are not accessed frequently if they are stored in the cache line with frequently accessed key-value pairs.

2.2.1 Two Method To Do the Acceleration To solve the first problem, both Flat-HTA and Hierarchical-HTA add two new units to calculate the index of key then compare with the whole cache line, instead of a set of arithmetic instructions and comparison instructions. It replaces the execution stage into address calculation and line comparison stages when hash operations come. So that CPU will not have too many stalls and the resource utilization will be improved.

As for the second problem, Hierarchical-HTA uses L2 cache as HTA stash to store the hot key-value pairs. Compared with LLC, L2 cache does not have to manage the data by the whole cache line. So this paper changes the cache controller to let L2 cache operate key-value pair instead of cache line. Thus it can improve the spatial locality to some extent.

2.2.2 The Drawback of The Reference Paper First, their design aim to reduce wrong-path execution by comparing all keys in one cache line. However, there is only 2 key-value pairs stored in each cache line. If the application has more collisions than 2, the improvement on wrong-path execution reduction will be minor.

Second, the method used to handle hash collision is probing-based in this paper. So the disadvantage of probing-based hash also occurs in this design, which is that more branch instructions are needed to deal with lookup or insert operation.

2.3 The Improvement Of Our Work

In order to modify the drawbacks mentioned above, we add a special unit called hash lookaside buffer (HLB) which is located between CPU and L1 cache. As a dedicated storage for hash operation, multiple keys with the same index can be compared with the given key simultaneously. In this way, number of branch instructions will be decreased a lot and wrong-path execution time will also be lower since the comparison is achieved by comparators in hardware. Also, as a faster hash storage, the memory stall time will be highly decreased since most of the hash storage will be inside HLB and the number of key-value pairs stored in main memory will be smaller. In order to support general purpose, we also consider the mem_table, hash table in memory hierarchy, as the backup of HLB. In Section 3, HLB size configuration, the detailed HLB structure, as well as the replacement policy will be described.

Unpublished working draft. Not for distribution.

3 Hash Lookaside Buffer(HLB)

In this section, detailed structure of the HLB is given. The hash table in this design is chain-based hash. Different key-value pairs referred to the same index are stored in the same row in HLB. When the row is full, the key-value pair will be stored in memory.

3.1 HLB Size Configuration

The goal of HLB is to store as many contents of hash table as possible. So the size of HLB is considered as 1MB. The HLB aims to be used on x86-64 system, so the key and value should be at most 64 bits, which is the length of address. If the value is larger than 64 bits, the address of it will be stored in HLB instead. The size for each key-value pair is 16 bytes. In order to support more indexes, number of rows of HLB is set as 1024 and number of columns is 64 as a result. So in this way, HLB supports maximum 1024 indexes and supports maximum 64 key-value pairs for each index.

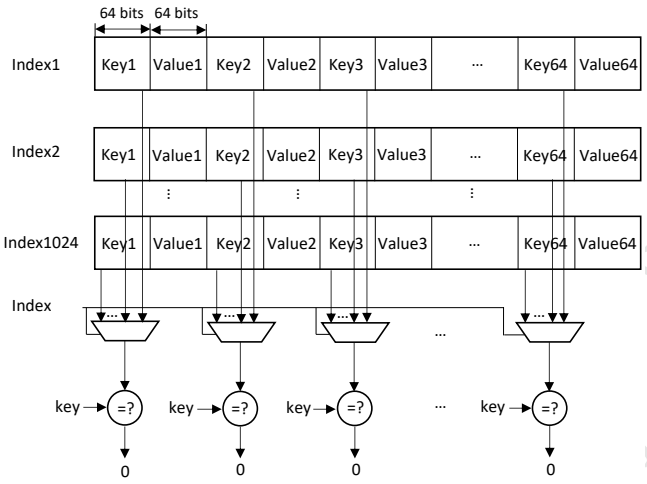


Figure 1: HLB Structure.

3.2 HLB Structure

HLB structure is shown in Fig. 1. Besides the 1024 rows and 64 columns structure mentioned in the previous subsection, 64 selectors as well as 64 comparators are designed. Selectors are used to select the key of index row that needed to be sent to comparator while comparators are used to do the comparison between the selected key and the given key (or NO_VALUE) in some cases. Each selector has 1024 inputs and each comparator has 2 inputs.

Hash index generator is also included in hardware, which performs as a role to calculate row index in HLB given a key. Hash function implemented in index generator is shown in Algorithm 1. Note that parameter s will be changed randomly during each rehash process. As hash lookup happens, the given key is sent to HLB as the input of the index generator. Then the index generator would output its index for lookup. As the lookup begins, selectors will be given the index i as a control signal as shown in Fig. 1 and the keys in line i will be selected and sent to comparators for comparison with the given key. If all the outputs of comparators are 0, then HLB will return 'NOT_FOUND' to CPU. It means that the given key is not in HLB so that CPU will lookup this key in memory.

Algorithm 1 Hash Function

1: $size\ t\ shift = 22$
2: $unsigned\ long\ long\ s = 2654435769ull$
3: $r = ((unsigned\ long\ long)key) * s$
4: $index = (size\ t)((r\&0xFFFFFFFF) >> shift)$

Once the given key not found in HLB, the given key will be sent as an input to the memory hash function, which is implemented by unordered_map library in c++. Then the lookup will happen in mem_table. If it's not in mem_table, return NOT_FOUND to CPU. Detailed lookup process will be explained in Section 4.2.3.

3.3 HLB Replacement Policy

Since mem_table is also used in our hash table, HLB replacement policy is needed.

In our work, the counters are used to record the reference times for each key-value pair in mem_table. When hash lookup happens and the requested key is in memory, then this key's corresponding counter would increase by 1. If the counter of the requested key shows that it is already referred 5 times, then the key-value pair will be sent to HLB and deleted in mem_table. If the corresponding line of this key in HLB is full, the key-value pair, namely, victim pair, pointed by column counter will be replaced and be sent to mem_table. If the line in HLB is not full, the key-value pair from memory can be directly inserted into HLB.

The victim pair is chosen by round robin method. The row index for victim pair is fixed as the index of the key which need to be inserted. A column counter is set by round robin method and the value of column counter is the column index of victim pair. Then the victim pair will be replaced and sent to mem_table, the insert key-value pair will be inserted directly into HLB accordingly.

4 Hash Instruction and Application Program Interface(API)

In this section, we describe the detailed implementation of our proposed HLB based hash operations, which includes both instruction supports and API supports.

4.1 Hash Instruction

To better apply our proposed HLB structure, we provide 4 new instructions: lookup, insert, erase and iterator. In this subsection, details about these 4 instructions will be given.

Algorithm 2 HLB Lookup Instruction Pseudo Code

```
1: hash_lookup_inst(reg key, reg result):
2: i = HLB_hash(key);
3: parallel_compare (key) in all columns of row
4: if HLB[i][j].key == key then
5:     result = value;
6:     return;
7: end if
8: result = NOT_FOUND;
```

Unpublished working draft. Not for distribution.

4.1.1 Hash Lookup Instruction Given a key key_1, CPU sets one register to the value of key_1 and gives another register to serve as the result register. Then index generator reads out key_1 stored in the first register. In index generator, given hash function is used to calculate index based on the given key key_1. In our experiment the hash function is shown in Algoritm 1. After the calculation of index generator, HLB would simultaneously compare the 64 key entries in the index row. If one of the keys matches key_1, value of the second register is set as the value of the found key. In this case, the lookup succeeds. Otherwise, lookup in HLB fails and the second register is set as NOT_FOUND. Detailed pseudo code can be found in Algorithm 2.

Algorithm 3 HLB Insert Instruction Pseudo Code

```
1: global lc = 0;
2: hash_insert_inst(reg key, reg value):
3: i =HLB_hash(key);
4: parallel_compare (key) in all columns of row
5: if HLB[i][j].key == key then
6:     HLB[i][j].value = value;
7:     return;
8: end if
9: parallel_compare(NO_VALUE) in all columns of row i
10: k=min j, ∀ j, HLB[i][j].key = NO_VALUE
11: HLB[i][k].key = key;
12: HLB[i][k].value = value;
13: return;
14: key = HLB[i][lc].key;
15: lc = (lc + +)%col_number;
16: return;
```

4.1.2 Hash Insert Instruction Similar to hash lookup, HLB insert instruction also needs two registers for key and value respectively and also needs index generation to generate row index in HLB. After that, 64 keys in index row are also simultaneously compared with the given key stored in key register to find out whether it's an update or not. If one of the keys matches the given key, then the value entry with that key will be set as the value stored in value register. If none of the keys matches the given key, then simultaneously compare the 64 keys in index row with NO_VALUE to find whether there's an empty slot. Then assign the key and value in registers to the empty slot with smallest column index and finish the insertion. If none of the keys equals to NO_VALUE, it means the current row is full. In this case, a victim key-value pair is chosen to be the option to be replaced. Victim key-value pair is chosen by parameter *lc*, which is a column index. *lc* begins with value 0, and will be added by 1 every time when a victim key-value pair is selected. After the victim key-value pair is chosen, the key of this victim key-value pair will be assigned to register key for further use. Detailed logic is presented in Algorithm 3.

4.1.3 Hash Erase Instruction In this part, hash erase instruction is explained. Two registers, key and value, are used here. Key is used to store the given key from CPU and result is used to indicate whether the erase is successful or not. After getting row index from index generator, all keys in index row are simultaneously compared with the given key. If one of the keys matches the given key, then set the key and value in this slot to NO_VALUE. If none of the keys matches the given key, HLB sets the result register to FAIL, which means the given key is not in HLB and it asks for further operation. Detailed pseudo code is given in Algorithm 4.

Algorithm 4 HLB Erase Instruction Pseudo Code

```
1: hash_erase_inst(reg key, reg result):
2: i=HLB_hash(key);
3: parallel_compare (key) in all columns of row i
4: if HLB[i][j].key == key then
5:     HLB[i][j].key == NO_VALUE;
6:     HLB[i][j].value == NO_VALUE;
7:     return;
8: end if
9: result = FAIL;
10: return;
```

4.1.4 Hash Iterator Instruction The main purpose of hash iterator instruction is to provide users ability to do iteration of this HLB. So in this instruction, the first and last not empty slots are found and the first and last slots' key are assigned to indicator register for further use. Also, the next non-empty slot can also be found in this instruction and return the next non-empty slot's key to indicator for iteration use. Details are given in Algorithm 5.

4.2 Application Program Interface(API)

In this subsection, we provide the APIs of our HLB structure, which include: clear, rehash, lookup, insert, erase, begin, end and next.

4.2.1 Clear Clear function is used to set every entries in HLB to NO_VALUE by calling 'hash_iterator_inst(2)'.

4.2.2 Rehash Rehash will happen when hash collision occurs and it will only happen when insertion happens. After calling hash insert instruction, if the content in key register is not equal to the given key wanted to be inserted, which means the index row is full and hash collision happens and HLB will be rehashed. Entries in HLB will be first moved into memory for a copy and then be set as NO_VALUE. By randomly changing the parameter *s* in the given hash function, entries will be rehashed and inserted by the new hash function. After doing rehash, if the given key-value pair is still not able to be inserted, another round of rehash will begin. A maximum rehash number 3 has been set to avoid deadlock happening. If rehash has already happened 3 times and the given key-value pair still cannot be inserted into HLB, then the victim key given by hash_insert_inst(key,result)'s key register will be evicted from HLB and inserted into mem_table. Then the given key-value pair can be inserted into HLB correspondingly.

4.2.3 Lookup For lookup API, we first search the given key in HLB by using 'hash_lookup_inst'. If the result register returns NOT_FOUND, we search it in mem_table. Otherwise it is successfully evicted from HLB. If we can find it in mem_table, depending on its recent using record the insert API may be called to put the frequently-used entry in memory into HLB. If it's also not in mem_table, then return NOT_FOUND means the key is never inserted before.

Unpublished working draft. Not for distribution.

Algorithm 5 HLB Iterator Instruction Pseudo Code

```
1: hash_iterator_inst(reg indicator):
2: if indicator == -1 // return the last not empty slot's key then
3:    for int i =ROW_NUM-1; i ≥ 0; i++ do
4:       parallel_compare (NO_VALUE) in all columns of row i
5:       k=max j, ∀ j, HLB[i][j].key != NO_VALUE
6:       indicator = HLB[i][k].key;
7:       return;
8:    end for
9:    indicator = NOT_FOUND;
10:   return;
11: end if
12: if indicator == 0 // return the first not empty slot's key then
13:   for int i = 0; i ≤ROW_NUM-1;i++ do
14:      parallel_compare (NO_VALUE) in all columns of row i
15:      k=min j, ∀ j, HLB[i][j].key != NO_VALUE
16:      indicator = HLB[i][k].key;
17:      ROW_iter = i;
18:      COL_iter = k;
19:      return;
20:   end for
21:   indicator = NOT_FOUND;
22:   return;
23: end if
24: if indicator == 1 //return the next not empty slot's key then
25:   if COL_iter == COL_NUM - 1 then
26:      COL_iter = 0;
27:      if ROW_iter != ROW_NUM -1 then
28:         ROW_iter ++;
29:      else
30:         indicator = NOT_FOUND;
31:         return;
32:      end if
33:   end if
34:   COL_iter ++;
35:   while HLB[ROW_iter][COL_iter].key == NO_VALUE do
36:      if COL_iter == COL_NUM - 1 then
37:         COL_iter = 0;
38:         if ROW_iter != ROW_NUM -1 then
39:            ROW_iter ++;
40:         else
41:            indicator = NOT_FOUND;
42:            return;
43:         end if
44:      end if
45:   end while
46:   indicator = HLB[ROW_iter][COL_iter].key;
47:   return;
48: end if
49: if indicator == 2 //set the whole table to NO_VALUE then
50:   for int i = 0;i <ROW_NUM;i + + do
51:      for int j = 0;j <COL_NUM;j + + do
52:         HLB[i][j].key = NO_VALUE;
53:         HLB[i][j].value = NO_VALUE;
54:      end for
55:   end for
56: end if
```

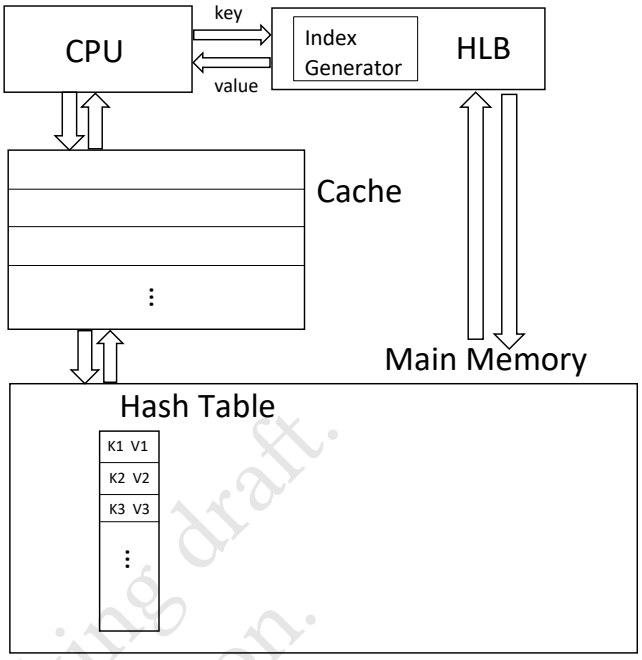


Figure 2: Data Flow of hash insert.

4.2.4 Insert Fig. 2 shows the data flow of hash insert. First key-value pair will be tried to insert in HLB by calling 'hash_insert_inst'. If the value stored in key register is not equal to the given key, it means the index row in HLB is full. As explained in Section 4.2.2, rehash will happen for at most 3 times. If the given key-value pair still cannot be inserted into HLB, victim key-value pair will be evicted from HLB and be inserted into mem_table. Then the given key-value pair can be inserted into HLB.

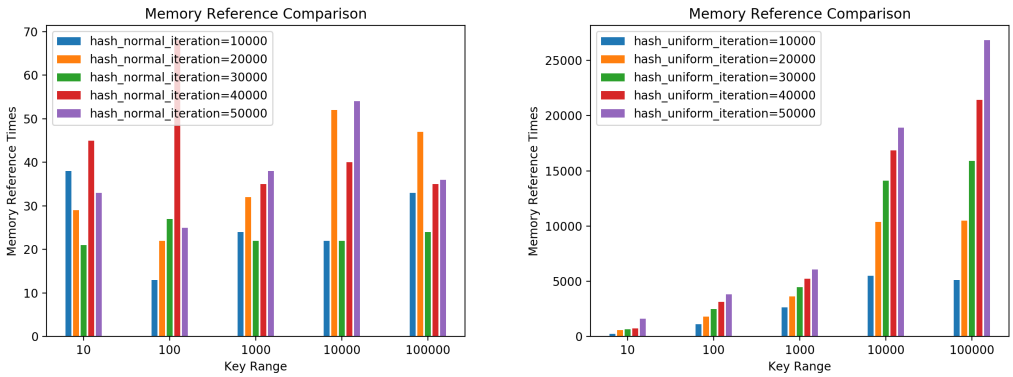
4.2.5 Erase For erase API, the given key will be first searched in HLB by using 'hash_erase_inst'. If the result register returns NOT_FOUNT, we search it in mem_table. Otherwise it is successfully evicted from HLB. If we can find it in mem_table, evict it from mem_table. If it's also not in mem_table, then return FAIL.

4.2.6 Begin If HLB is not empty, then the key of the first entry in HLB is returned as the first key in HLB and mem_table (HLB_table) structure. When refer the first key in HLB, we use 'hash_iterator_inst(0)' instruction.

4.2.7 End For the end of HLB_table, we assume it the end of mem_table if mem_table is not empty. Then we can use end() function in unordered_map library to get the end of mem_table. If mem_table is empty, we use 'hash_iterator_inst(-1)' to get the end of HLB.

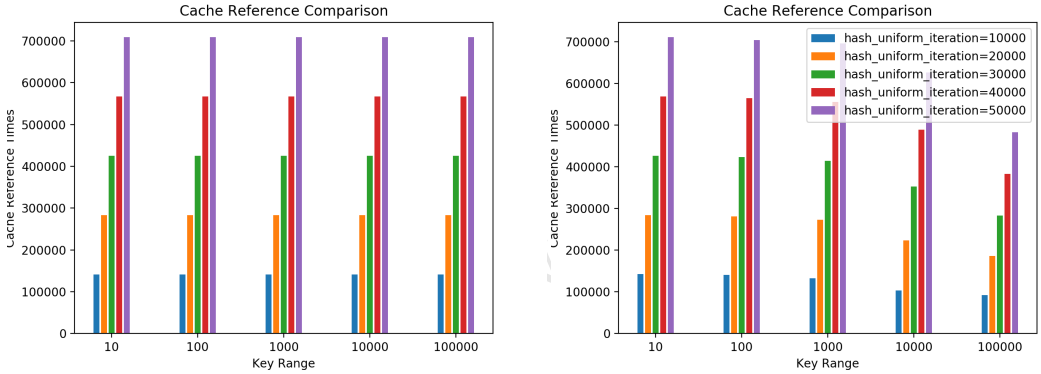
4.2.8 Next The next API is used to find the next not empty key for the current position in iterator. If the current iterator is in HLB and not the end of HLB, then 'hash_iterator_inst(1)' will be used to find the next iterator in HLB. If the current iterator is the end of HLB, then return the fist entry in mem_table. If the current iterator is in mem_table and not the end of mem_table, then return the next iterator in mem_table.

Unpublished working draft. Not for distribution.



(a) Cache miss with normal distribution.

(b) Cache miss with uniform distribution.



(c) Cache hit with normal distribution.

(d) Cache hit with uniform distribution.

Figure 3: Memory reference comparison for different distributions.

5 Experiment

In this section, we provide experiment setups, implementation details as well as evaluation results of our experiments.

5.1 Experiment Set Up

5.1.1 Tools C++.

5.1.2 Number of cores one.

5.1.3 Baseline Baseline used in our experiment is a probing-based hash table. The number of entries in this table is the same as the number of entries in HLB, which is $1024 \times 64 = 64$K. The hash function used for the baseline is the hash function implemented in c++ library.

5.1.4 Architectural resource sizes In our proposed structure, size of HLB is 1MB. Besides the storage of HLB, we also implement a 1MB LRU cache as a memory backup. So in this case, the storage is 2MB totally. While for the baseline, a 2MB LRU cache is used for storage.

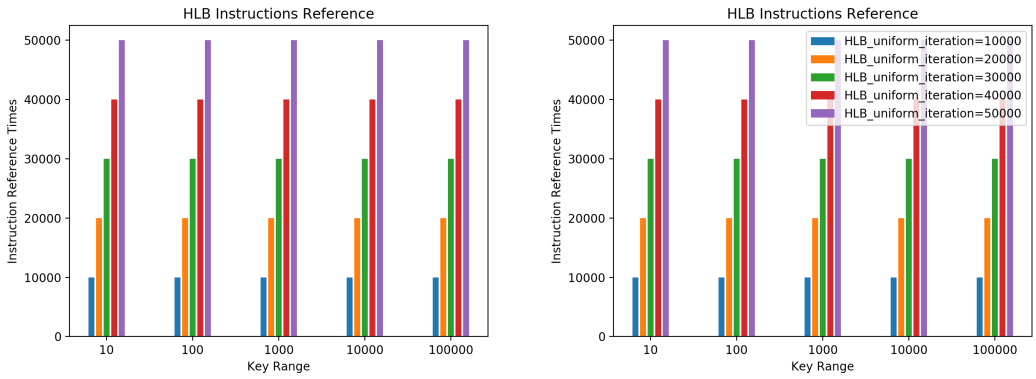
5.2 Implementation Details

In our experiment, our proposed HLB structure with 1MB LRU cache and traditional probing-based hash table with 2MB LRU cache are compared for evaluation. To be more detailed, cache hits and cache misses are counted to estimate the total latency. Also the latency for proposed 4 instructions are also estimated. Hence, the total latency for our proposed HLB structure is calculated by adding the cache miss time, cache hit time and HLB hit time together while the total latency for the baseline is calculated by adding the cache miss time and cache hit time together.

To implement memory reference time, every memory reference in one function will be recorded in our LRU cache if it hasn't been referred to in this function before. If the following instruction refers the cache line which is already in LRU cache, it will be counted as a cache hit. Otherwise, it would be a cache miss.

For the process of simulation, first we run the test with certain number of iterations on proposed HLB structure and baseline, respectively. For each iteration, keys and values are generated randomly. Based on the keys and values generated, do insertion with 50% probability, do erase with 20% probability and do lookup with the rest 30% probability. In the experiment, the iteration numbers

Unpublished working draft. Not for distribution.



(a) Normal distribution. (b) Uniform distribution.

Figure 4: Instruction reference comparison for different distributions.

Table 1: Memory reference count and latency comparison.

		Reference count			Latency (ns)			
Key range	**Structure**	**Cache hit**	**Cache miss**	**HLB**	**Cache**	**Memory**	**HLB**	**Total**
10	HLB	0	0	50,000	1	22.5	5	250,000
	Baseline	710,434	1,631	0				747,132
100000	HLB	0	0	50,000				250,000
	Baseline	483,039	26,810	0				1,086,264

are chosen as 10000, 20000, 30000, 40000 and 50000. Key ranges, which is the possible range the key can reach, are set as 10, 100, 1000, 10000 and 100000. The distribution of key and value is generated by normal and uniform distributions.

5.3 Evaluation Results

In this part, we show the evaluation results of our proposed HLB structure and baseline structure. First, the number of cache misses, cache hits and HLB references are counted. Then the latency for referring to 1MB HLB, 2MB LRU cache are estimated by *CACTI* and memory reference time is estimated by the latency of DDR-400. Based on the above information, the latencies for proposed HLB structure and baseline structure are estimated and compared.

5.3.1 Counts of different references Fig. 3(a) and 3(b) show the cache misses for normal and uniform distributions, respectively. Note that in both cases, cache misses for our proposed HLB structure are 0. So in these figures, only the cache misses for baseline case are shown. Similar to Fig. 3(c) and 3(d), which are cache hits for normal and uniform distributions. Fig. 4(a) and 4(b) show the number of new instructions we proposed used during the testing phase for our proposed HLB structure. For all these 6 figures, x-axis shows the possible range for the key and value and y-axis shows the number of references for cache miss, cache hit and new instructions.

Normal and uniform distribution are chosen here to generate keys and values because they are used widely for all kinds of applications. The mean for normal distribution is half of the key range and variance is 0.167 times key range in order to follow the standard normal distribution. While the range for uniform distribution is $[0, key\_range]$. Comparing the results of these two distributions, the influence of key ranges will be small in normal distribution case. This is because of the property of normal distribution, most of the keys generated are concentrated among the mean of the normal distribution so the actual number of keys in the experiment is small. As a result, the influence of key range is small in the normal distribution case. On the contrary, the keys generated by uniform distribution spread out so if the key range is larger, the actual number of keys will be larger. In this case, the number of the cache misses will be larger as key range increases, while the cache hits will be smaller as cache misses increase. As for the influence of iteration number, it is obvious that as iteration number increases, both cache hits and cache misses will increase.

As for HLB references, they are strictly equal to the number of iterations because for each iteration, one instruction is called for one hash operation no matter what are the key range and the distribution. So the trend of Fig. 4(a) and 4(b) is more predictable.

5.3.2 Latency of HLB, cache and memory The latency of HLB and cache are generated by *CACTI*. For HLB, the latency is simulated by a 64-way set associative, 64 bit tag, 1MB size and 8B block size cache. Since *CACTI* only supports with a maximum 16-way cache, the latency for 64-way cache is estimated based on the latency of 2-way, 4-way and 8-way set associative cache. The latency is chosen based on the maximum between the tag array component and data array component and after the approximating the latency of 64-way set associative should be 10 cycles. As for the 2MB LRU cache used

Unpublished working draft. Not for distribution.

in the baseline simulation, we choose 4-way set associative, 45 bit tag, 2MB size and 64B block size cache. The latency is 2 cycles and it is also chosen by the maximum of the tag array component and data array component. Frequency we choose is the frequency of Intel Ice Lake i7-1068G7, which is 2.3GHz. So the latency of HLB is 5ns and the latency of LRU cache is 1ns.

The latency of memory is chosen as the latency of DDR-400, which is 22.5ns.

5.3.3 Latency Comparison Based on the previous two subsections, latency for the proposed HLB structure and baseline structure are compared in this part. Choose the reference counting with 50000 iterations, 100000 key ranges and uniform distribution, the latency for proposed HLB structure is 0.25ms while the latency for baseline structure is 1.1ms. So the total speedup is 4x, which is the worst case of baseline structure. For the best case of baseline structure with 50000 iterations, key range is 10 since in this case cache misses is the smallest. The latency for proposed HLB structure is also 0.25ms while the latency for baseline structure is 0.75ms. So the total speedup is 3x. The memory reference count and latency comparison for 50000 iterations are shown in Table 1.

6 Conclusion

In this paper, we propose HLB based general purpose hash table application. To reduce the hash operation latency, we add a hardware HLB which can compare multiple keys parallelly. To support HLB, We design four instructions, and build our HLB_table API based on HLB instructions. Our proposed HLB_table consists 2 parts, HLB and in-memory hash table. The in-memory hash table serves as a backup for HLB, whenever the HLB is not sufficient, in-memory hash table would be applied. We show the efficiency of HLB_table by comparing it with traditional probing-based hash table. Based on the latency generated by *CACTI*, overall reference latency is also compared. Result shows that the total speedup is 3x to 4x, which will improve the efficiency.

References

[1] Guowei Zhang and Daniel Sanchez. 2019. Leveraging Caches to Accelerate Hash Tables and Memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 440–452.

Unpublished working draft. Not for distribution.