Hash Table Accelerator

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

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

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

ABSTRACT

Hash table is a widely used data structure in many applications. However, hash value lookup is a time consuming problem because of multiple-time key comparison and it cannot be avoided. 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 by combining chaining and probing in hash table. For each hash lookup, 64 keys can be compared simultaneously, which will highly increase the execution time. If the entries in HLB are not enough, memory hash table is used as a supplement to HLB. Corresponding HLB replacement policy is designed for data exchange in HLB and memory. Instructions are designed for hash lookup and hash insert. Once implemented, the result of HLB will be compared with Google hash table benchmark. The implementation will be finished on x-86 CPU system.

CCS CONCEPTS

• **Computer Architecture → Hash Accelerator**.

KEYWORDS

hash table, accelerator, table lookup

ACM Reference Format:
Yaodong Sheng, Zhankai Feng, and Anlan Yu. 2019. Hash Table Accelerator. In *ECE 401: Advanced Computer Architecture, Sep. - Dec., 2019, Bethlehem, PA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Alg.1 shows the pseudo code for traditional probing based hash lookup. We can notice that there is a loop to compare the two keys. Every time when hash lookup is executed, program has to do key comparison for several times. One can imagine that the efficiency cannot be promised. For chain based hash lookup, keys can be chained onto the corresponding index. However, for each key comparison, there is one memory reference since the key has to be got using pointer. If the chain is long, it will be quite time consuming.

Based on the drawback of traditional hash lookup, we propose to increase the throughput while decrease the latency of operations on hash table. We add a special unit called hash lookaside buffer (HLB) in following article which is between CPU and L1 cache. In Section 2, the detailed HLB structure and data flow description as long as the replacement policy are given.

Algorithm 1 Probing Based Hash Lookup Pseudo Code

```
1: $Vector < T > hash\_table$
2: $index \leftarrow hash(key)$
3: while $hash\_table.key\ != key\ \&\&\ index\ != hash\_table.size()$ do
4:     $index++$
5: end while
6: if $index == hash\_table.size()$ then
7:     $return\ -1$ // key not exist
8: else
9:     $return\ index$
10: end if
```

2 HLB STRUCTURE AND DATA FLOW

In this section, detailed structure of the HLB is given. The hash table in this design combined chain based hash and probing based hash. Different key-value pairs referred to the same index are stored in the same row in HLB. Only when the row is full, the key-value pair will be stored in the next row. 5 rows are reserved as backup positions for each row.

2.1 HLB Size Configuration

The goal is to store as much contents of hash table as possible. The number of rows of HLB is set as 1024 and each row can store 64 contents. Each content is consist of 32 bits key and 32 bits value pointer. There's also 3 bits reserved for row counter and 6 bits reserved for column counter, which are used for replacement policy. Hence, the size for each row should be $64 \times (32+32)+3+6 = 4.105$KB. Size for the whole HLB is $1024 \times 4.105 = 4.203520$MB.

2.2 HLB Structure

HLB structure is shown in Fig. 1. Totally there are 1024 rows so 1024 indexes are available in HLB. Each row can store 64 key-value pairs. Each key is 32 bits and also 32 bits for each value.

Besides, there are 3 bits used as a row counter to indicate from which row the replacement happens. There are also 6 bits used as a column counter to indicate which entry in the row are going to be replaced. Details of replacement policy will be illustrated in the following parts.

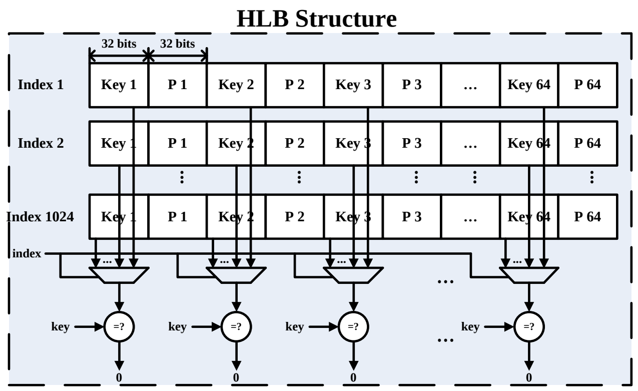

Figure 1: HLB Structure.

2.3 Data Flow for Hash Lookup

Fig. 2 shows the data flow of hash lookup. Given a key $key1$, CPU calculates the index i with HLB hash function i = HLB_hash($key1$). Index i is sent to HLB for lookup. As the lookup begins, multiplexers will be given the index i as a control signal as shown in Fig. 1 and the keys in line i will be sent to comparators for comparison with the given key. If all the outputs of comparators are 0, then the control signal given to multiplexers are index $i + 1$ and keys in line $i + 1$ are sent to comparators for comparison. If all the outputs are 0, then the control signal to multiplexers are index $i + 2$ and so on so forth until index is $i + 8$ and the outputs of comparators are still all 0. Then HLB will return 'null' to CPU. The given key will be rehashed using a different hash function dedicated for memory hash table j = MEM_hash in CPU. CPU will then figure out the address of line j in hash table and compare the keys in line j one by one. If the key is also not in memory, memory will return 'null' to CPU and CPU will know the key-value pair is not in the table. If the key is indeed in memory, memory will return the value to CPU and add 1 to the counter of this key-value pair. Then check the counter of this key-value pair, if it's larger than a value (to be determined), then using HLB replacement policy to insert it to HLB and a corresponding entry in HLB will be replaced.

2.4 Data Flow for Hash Insert and Remove

Similar to hash lookup, data flow for hash insert may also include memory operations. When the program attempt to insert a key-value pair into hash table, first CPU will calculate index based on hash function. Then CPU will send key and index to HLB for insertion. If the given line is full, then the next line will be checked until the next 8 line. If all the 8 lines are full, entry in HLB with row index equal to the given index plus the number in row counter and column index equal to the number in column counter are replaced to memory. The key-value pair requested to insert will be inserted in this position.

Hash remove can be done based on hash insert so here will not be illustrated too much.

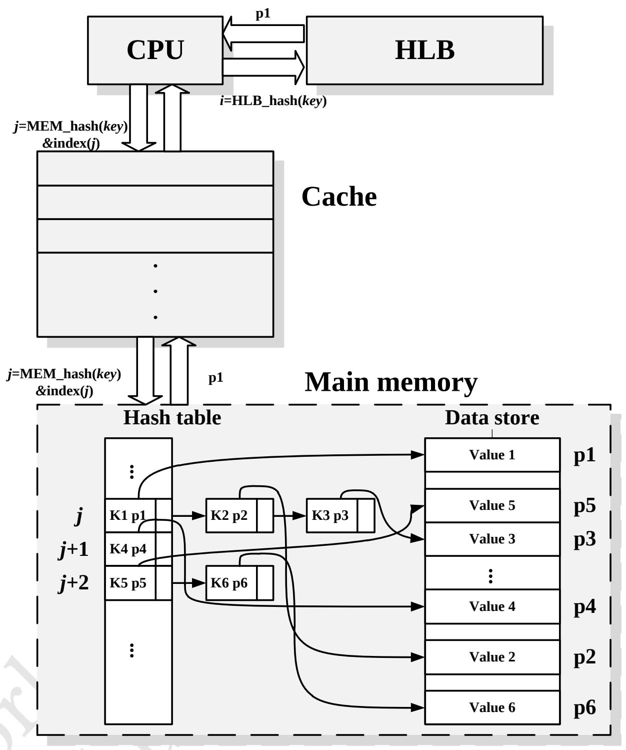

Figure 2: Data Flow of hash lookup.

2.5 HLB Replacement Policy

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

When hash lookup happens and the requested key is in memory, if the counter of the requested key show that it is referred many times, then the key-value pair will be sent to HLB and deleted in memory. If the corresponding lines of this key in HLB are full, the key-value pair with row index equal to the result of HLB_hash given the requested key plus the number in row counter and column index equal to the number in column counter will be replaced and be sent to memory. If the lines in HLB are not full, the key-value pair from memory can be directly inserted into HLB.

When hash insert happens and the HLB is full, entry in HLB with row index equal to the result of HLB_hash given the insert key plus the number in row counter and column index equal to the number in column counter will be replaced and sent to memory. The insert key-value pair can be inserted directly into HLB accordingly.

3 HLB ISA EXTENSIONS

Algorithm 2 shows the detailed instruction representation. Algorithm 3, 4 and 5 show the detailed hash lookup, insert and remove realizations.

Algorithm 2 HLB Instructions Pseudo Code

```
1:  HLB:
2:  global rc = 0; lc = 0;
3:  hash_lookup_inst(reg key, reg result):
4:  for int $i$ =HLB_hash(key); $i$ <=hash(key) + 8; $i++$ do
5:      parallel_compare(key) in all column of row i
6:      if HLB[i][j].key == key then
7:          result = value;
8:          return;
9:      end if
10: end for
11: result = NULL;
12: hash_insert_inst(reg key, reg value):
13: for int $i$ =HLB_hash(key); $i$ <=hash(key) + 8; $i++$ do
14:     parallel_compare(key) in all column of row i
15:     if HLB[i][j].key == key then
16:         HLB[i][j].value = value;
17:         return;
18:     end if
19: end for
20: key = HLB[h(key) + rc, lc].key;
21: $rc = (rc++)\%row\_number$;
22: $lc = (lc++)\%col\_number$;
23: $return$;
```

Algorithm 3 HLB Lookup Pseudo Code

```
1:  word hash_lookup(word key)
2:  if hash_lookup_inst(key) == NULL then
3:      if mem_hash.find(key) ! = NULL then
4:          value = mem_hash[key].value;
5:          mem_hash[key].counter ++;
6:          if mem_hash[key].counter>=Threshold then
7:              mem_hash.remove(key);
8:              hash_insert(key, value);
9:          end if
10:         return value;
11:     else
12:         return NULL;
13:     end if
14: else
15:     return hash_lookup_inst(key);
16: end if
```

Algorithm 4 HLB Insert Pseudo Code

```
1:  bool hash_insert(word key, word value)
2:  bool result;
3:  k = key;
4:  hash_insert_inst(k, value);
5:  if k ! = key then
6:      value_k = hash_lookup_inst(k);
7:      result = mem_hash.insert(k, value_k);
8:      hash_insert_inst(key, value);
9:  end if
10: return result;
```

Algorithm 5 HLB Remove Pseudo Code

```
1: bool hash_remove(word key)
2: bool result;
3: hash_remove_inst(key, result);
4: if result == FAIL then
5:     result = mem_hash_remove(key);
6: end if
7: return result;
```

4 LITERATURE REVIEW

4.1 Leveraging Caches to Accelerate Hash Tables and Memoization [1]

In this paper it 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 is some keys accessed frequently, the cache would store some key-value pairs which is not accessed frequently because cache load the data as block size.

This paper mentioned two methods to accelerate hash operations with the modification of these two problems, which are FLAT-HTA and Hierarchical-HTA.

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 these two stages when hash operations come. So that CPU will not have too many stalls then increase the resource utilization.

As for the second problem, Hierarchical-HTA use 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 change the cache controller to let L2 cache can operate key-value instead of cache line. Thus it can improve the spatial locality to some extent.

5 BENCHMARK

The benchmark to be used in this project is Google hash table benchmark.

6 EXPERIMENTAL SETUPS

Tools: C++, LLVM, Pin tool, ZSim.

Parameters of architectural simulation: X86, 32 bits.

Number of cores: one.

Depth of pipeline: 12-Stage.

Architectural resource sizes: 4M L1-cache as HLB, 64 32bits-comparators, 2 registers for row_counter and col_counter. intel-i7 core CPU with 2g DDR3 memory.

Cache sizes: 1M L1 cache. 2M L2 cache. 4M LLC cache.

7 IMPLEMENTATION AND EVALUATION STEPS

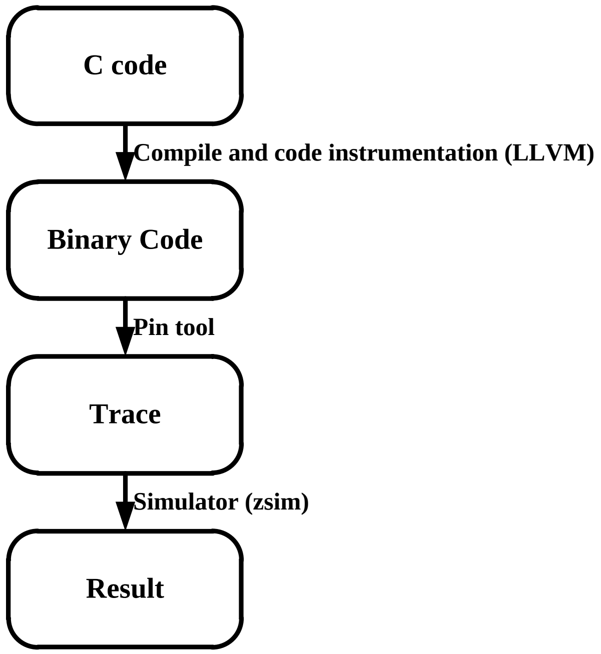

Figure 3: Implementation and Evaluation Steps.

Fig. 3 shows the implementation and evaluation steps of this project. First, the operations of hash table need to be defined in the top level c++ code. Using compiler LLVM, c++ code can be complied and instrumentation need to be done, which turns the c++ function into instruction. The compile result will be binary code, which is machine code. Then use pin tool to generate the trace of the code. Final step is to use simulator to turn a selected op code into our needed function (like hash lookup and insert) and the result will be got.

Currently we have already installed and started to learn LLVM, pin tool and zsim.

8 WEEKLY TIMELINE

8.1 First Week

Optimize the HLB. Currently our design is focused on fixed length hash table. Next step is to finish the design of shrinking and extending the hash table. Then more instructions like iterate and random access will be designed.

8.2 Second Week

Finish the compiler setup and using llvm to instrument the code.

8.3 Third Week

Using pin tool to get the trace.

8.4 Forth Week

Implement our instruction on simulator and feed trace to simulator.

8.5 Fifth Week

Same as Forth week

8.6 Sixth Week

Compare between our result with state of art and finish report.

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.