Performance Analysis of Cache-conscious Hashing Techniques for Multi-core CPUs

Euihyeok Kim and Min-Soo Kim

Department of Information & Communication Engineering, Daegu Gyeongbuk Institute of Science & Technology, Hyeonpung-Myeon Dalseong-Gun, Daegu, Korea
keh@dgist.ac.kr, mskim@dgist.ac.kr

Abstract

A hash table is a fundamental data structure implementing an associative memory that maps a key to its associative value. Due to its very fast mapping operation of O(1), it has been widely used in various areas such as databases, bioinformatics, and distributed computing. Besides, the paradigm of micro-architecture design of CPUs is shifting away from faster uniprocessors toward slower chip multiprocessors. In order to fully exploit the performance of such modern computer architectures, the data structures and algorithms considering parallelism become more important than ever. This paper implements three cache-conscious hashing methods, linear hashing and chained hashing, and also, a modern hashing method, hopscotch hashing, and analyzes their performance under Intel 32-core CPU of Nehalem microarchitecture. We implement each hashing method using state-of-the-art techniques such as lock-free data structures, especially based on compare-and-swap (CAS) operations, and refinable data structures. To the best of our knowledge, the work done by this paper is the first work analyzing the performance of three all hashing methods under the same implementation framework. Experimental results using data of 2^{23} (i.e., about eight millions) key-value pairs shows that lock-free linear hashing is the best for insert operation among three hashing methods, and lock-free chained hashing is the best for lookup operation. Hopscotch hashing shows the second best performance of lookup operation. However, hopscotch hash table size is much bigger than other hash table size. Through experiments, we have found that the hopscotch hashing is relatively not efficient than other hash methods.

Keywords: *linear hashing, chained hashing, hopscotch hashing, parallel programming, lock-free hash tables, refinable hash tables, multicores, Intel microarchitecture, compare-and-swap, cache friendly*

1. Introduction

A hash table is a very fast data structure with its excellent lookup performance of O(1) implementing an associative memory that maps a key to its associative value. Due to its very fast mapping operation, it has been widely used in various areas such as databases, bioinformatics, and embedded systems. Representative hashing methods are linear hashing [1] and chained hashing [2]. Recently, hopscotch hashing [3] has been proposed.

Meanwhile, since the paradigm of micro-architecture design of CPUs is shifting to on-chip multi-core processors instead of increasing clock speed. For instance, recent desktop has CPU up to 6 and 10 CPU for server. Since CPU is developed as the clock speed does not increase, single thread program does not make the program to speed up automatically. Therefore, in order to fully exploit the performance of such modern

computer architectures, the data structures and algorithms considering multi-threading become more important than ever.

There have been many studies on implementing an efficient hashing method under multi-core architecture [4-10]. The major operations for hashing are insertion and lookup. When using multi threads, the insertion operation is required to solve the race condition problem since multiple threads tries to update a common data structure simultaneously. A common method solving that problem is (1) to acquire a mutex lock, (2) update the data structure inside a critical section, and then, (3) release the lock. However, this naïve method could degrade the performance severely, especially when there are many concurrent threads, since the threads compete with each other for getting a lock, which is called *thread contention*. In order to reduce the thread contention, *lock-free* hashing methods [4], which don't use a lock, and *refinable* hashing methods [5], which use fine-granularity locks, are proposed. Especially, the lock-free hashing methods are usually based on a hardware-level instruction called *Compare-And-Swap*(CAS).

In this paper, we have analyzed the performances of three hashing methods, especially cache-conscious hashing methods — linear hashing, chained hashing, and hopscotch hashing — for multi-threading. Nowadays, due to the memory-wall problem, the cache-conscious algorithms and data structures are as important as multithreaded ones. We have first implemented those three hashing methods with some modification of algorithms so as to achieve the best performance and best fairness. Then, we have compared the performance of insertion and lookup of them using Intel 32-core machine. To the best of our knowledge, the work done by this paper is the first work analyzing the performance of those cache-conscious hashing methods under the same implementation framework and under the machine of a large number of cores. In case of hopscotch hashing, although the authors insist that the performance of hopscotch hashing is better than other hashing methods under Sun UltraSPARC, it shows quite different results at least under Intel multi-core machine.

For the best fairness, we have implemented linear hashing and chained hashing in a lock-free manner based on CAS operation and hopscotch hashing using a refinable lock-based manner. In case of chained hashing, we have modified the data structure and algorithms with considering the size of cache line. In case of hopscotch hashing, there have not been proposed any lock-free data structures or algorithms so far due to its complexity, we have used a refinable lock-based method, the best method for the performance.

The rest of this paper is organized as follow. The existing work related to this study is described in Chapter 2. Chapter 3 explains the data structure and algorithms of each hashing method used in this paper. Chapter 4 analyzes and evaluates three hashing methods, and then, Chapter 5 concludes the paper.

2. Related work

We briefly introduce three cache-conscious hashing methods, linear hashing, chained hashing, and hopscotch hashing in this section. Linear hashing and chained hashing are the conventional methods, and hopscotch hashing is the method recently proposed.

2.1. Linear hashing

Linear hashing (1) first tries to insert a pair of <key, value> to the target bucket, (2) finds the first empty bucket with *linear probing* from the next bucket of the target one if a *hash*

collision occurs, and (3) inserts the pair to the empty bucket. Since the pair is usually stored near the target bucket when the fill factor is not too high, it is known that linear hashing shows a good performance with a moderate fill factor below 50% [3]. However, if the fill factor is larger than 50%, the performance of linear hashing might degrades much since the number of buckets to be linear probed increases, and so, many cache lines should be read from main memory [11].

2.2. Chained hashing

Chained hashing solves the hash collision problem by creating a linked list at the target bucket and inserting <key, value> into the list. Chained hashing has an advantage that the performance degradation is not severe compare with linear hashing even when the fill factor is very high. It is because chained hashing needs only to search the corresponding linked list while linear hashing needs to scan the overall hash table in a worst case. However, chained hashing has a drawback that it is usually not cache friendly due to the nodes of the linked lists non-contiguously allocated in memory space. Chained hashing also needs a thread-safe memory manager or a garbage collector for managing the linked lists efficiently.

2.3. Hopscotch hashing

Hopscotch hashing is the method that has all the features of linear hashing, chained hashing, and cuckoo hashing [12]. Hopscotch hashing first finds out the closest empty bucket from the target bucket with linear probing. Then, it inserts <key, value> into the bucket if the distance between two buckets is less than the size of a cache line. Otherwise, it performs repeatedly data displacement between the empty bucket and the bucket cache-line distant from the empty bucket in the reverse direction of linear probing. Hopscotch hashing has an advantage that could improve the performance of lookup by reducing the number of cache misses, which is achieved by data displacements performed only in the rage of a cache line. We denote the number of buckets in a single cache line by *H* meaning hop range. For instance, when the size of a cache line is 512 bits, and the size of a bucket is 8 bytes (*i.e.*, 64 bits), H = 8. Each bucket stores extra information, called *hop information*, of *H* bits for data placement, which is actually a bitmap that indicates in which buckets the set of keys displaced are located instead of the current bucket.

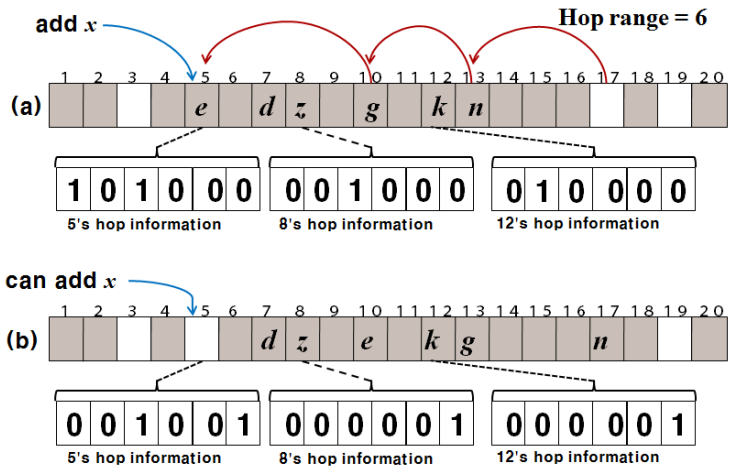

Figure 1. Example of insertion in Hopscotch hashing

Figure 1 presents an example of insertion in hopscotch hashing. The white colored buckets represent empty buckets and the grey colored ones occupied buckets. We assume the hop range *H* is 6. Figure 1(a) presents the situation where the insert operation tries to insert the data x into the bucket 5, but a hash collision occurs. The insert operation finds out the closes empty bucket 17 with linear probing, and checks the hop information of the bucket 12, which is of *H*-1 buckets preceding the bucket 17. The second bit value 1 in the corresponding hop information indicates that the data *n* in the next bucket of the bucket 12 (*i.e.*, bucket 13) could be displaced to the bucket 17. The hop information is updated after displacement of the data *n*. The insert operation checks again the hop information of the bucket 8, which is of *H*-1 buckets preceding the bucket 13. Then, it displaces the data *g* in the bucket 10 to the bucket 13 and updates the hop information of the bucket 10. Finally, the insert operation checks the hop information of the bucket 5, displaces the data *e* in the bucket 5 to the bucket 10, and inserts the data *x* into the bucket 5 with updating the corresponding hop information. Figure 1(b) presents the status of hash table and hop information after the insertion of the data *x* is completed.

3. Implementation

The lock-free data structure and algorithm based on CAS instructions are known as the multi-threading method of the best performance. Several lock-free hashing methods have been proposed for linear hashing and chained hashing [4, 7, 9, 10]. But, there is no lock-free method for hopscotch hashing since the algorithm is more complex than those of linear hashing or chained hashing. It is well known that designing a lock-free version of a certain algorithm is much more difficult than designing its lock-based version. The best implementation method for hopscotch hashing in terms of performance is a *refinable* lock-based method, which uses bucket-level locks instead of a table-level lock [5]. In this section, we present CAS-based lock-free data structures and algorithms for linear hashing and chained hashing and refinable lock-based ones for hopscotch hashing.

3.1. Lock-free linear hashing

Several lock-free methods for linear hashing have been proposed so far. Gao, *et al.*, [7] have proposed the method where the size of a hash table can grow and shrink dynamically. However, it has a drawback that thread safety is not guaranteed without PVS [8]. Purcell and Harris [9] have proposed the method that needs no garbage collection and has a small footprint. However, it has a drawback that it cannot be used for implementing a dictionary since it cannot store values together with keys [9]. Stivala, *et al.*, [4] have proposed the method that is based on CAS operations, is thread safe without PVS, and can be used for implementing a dictionary. Therefore, we use their data structure and algorithm.

Figure 2 presents the table structure of linear hashing. The sizes of keys and values are all 8 bytes. When creating a hash table at first, the keys and values are initialized with *NULL*.

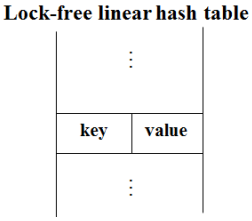

Figure 2. Data structure of lock-free linear hashing

Figures 3 and 4 presents insertion and lookup algorithms for linear hashing, respectively. Both operations use the *get_entry* function that tries to find out the bucket having a given key and returns its location if there is such bucket, *NULL* otherwise. The *insert* function tries to swap the content of the target bucket with the given key atomically using a CAS instruction in Line 19. If the content of the target bucket is not *NULL*, *i.e.*, other thread has already inserted a key into the bucket using CAS, then the original CAS instruction returns non-NULL value, which means CAS fails. In that case, the *insert* function tries to insert the key again by calling itself recursively in Line 20.

```
1  get_entry(key)
2       h ← hash(key)
3       ent ← hashtable[h]
4       probes ← 0
5      WHILE probes < TABLE_SIZE - 1 ∧ ent.key ≠ key ∧ ent.key ≠ NULL
6      DO
7            probes ←  probes + 1
8            h ←  (h + 1) mod TABLE_SIZE
9           ent ← hashtable[h]
10     IF probes ≥ TABLE_SIZE - 1 THEN
11         RETURN NULL
12     ELSE
13         RETURN ent
14 Insert(key, value)
15     ent ← get_entry(key)
16     IF ent = NULL THEN
17         error_exit("hash table full")
18     IF ent.key = NULL THEN
19         IF CompareAndSwap(ent,NULL,<key,value>)≠ NULL THEN
20             RETURN insert(key,value)
21     ent.value ← value
22     RETURN TRUE
```

Figure 3. Insert algorithm of lock-free linear hashing [5]

```
1  Lookup(key)
2       ent ← get_entry(key)
3       IF ent ≠ NULL ∧ ent.value ≠ NULL THEN
4           RETURN ent.value
5       ELSE
6           RETURN KEY_NOT_FOUND
```

Figure 4. Lookup algorithm of lock-free linear hashing [5]

3.2. Lock-free chained hashing

Several implementing methods for lock-free chained hashing have been proposed. Shalev and Shavit [10] have proposed a so-called split-ordered lists method and Stivala, *et al.*, [4] introduced a CAS-based method. Since the split-ordered lists method has a bigger overhead in term of memory usage and a higher cost in terms of maintaining linked lists than the CAS-based method, we adapted the CAS-based method. The original CAS-based method [4] is not cache-conscious, so we designed a new data structure that allocates a node of linked lists by a *chunk* which size is the same with a cache line, not by a pair of <key, value>.

Figure 5 presents the data structure we designed for cache-conscious lock-free chained hashing. Each bucket consists of 8-byte *key*, 8-byte *value*, and 8-byte pointer *next*. All *keys* and *values* are initialized with *NULL* when initializing the table. A node of each linked list is composed of a short array of *keys*, a short array of *values*, *blank*, and 8-byte pointer *next*. The *blank* variable is for memory alignment. If the size of a cache line is 64 bytes, the number of elements of the array *keys* (or *values*) is three, and the size of *blank* is 8 bytes. By allocating a node by the size of a cache line, the *lookup* operation using this new data structure can reduce the number of cache misses at most three times compared with the conventional data structure that allocates a node by the size of <key, value>.

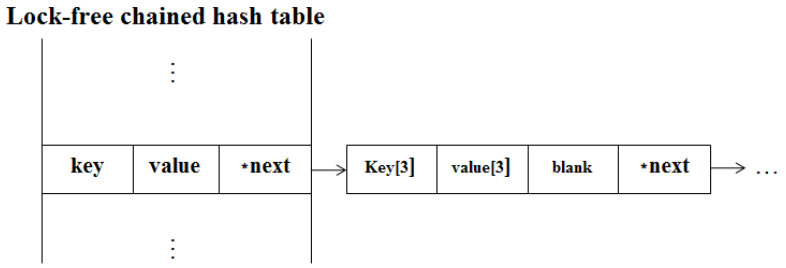

Figure 5. Data structure of lock-free chained hashing

Figure 6 presents the *insert* algorithm for lock-free chained hashing. In Line 3, it tries to insert a given data to the target bucket using a CAS operation. If the CAS instruction fails, *i.e.*, there is already other *key* in the bucket, the algorithm checks the existence of a linked list in Line 7, and then, tries to insert the data to an empty bucket of the first node of the linked list in Lines 8~9. If there is no linked list or no empty bucket in the first node, the operation allocates a new node, initializes it, and modifies the *next* pointer of the new node so as to point to the old node in Lines 11~13. Finally, the algorithm needs to modify the *next* pointer of the target bucket so as to point to the newly allocated node. Here, since there might be multiple threads trying to modify the *next* pointer of the target bucket simultaneously, the algorithm needs to modify the pointer by using the CAS instruction as in Line 14. If CAS fails, *i.e.*, other thread has already created a new node and modified the pointer, then, the current thread should release the memory that it allocated for the node as in Line 15. Otherwise, the amount of memory leaked gets bigger and bigger as time goes on.

```
1  Insert(key, value)
2      bucket ← hashtable[hash(key)]
3      IF CompareAndSwap(bucket.key,NULL,key) = NULL THEN
4          RETURN true
5      WHILE true
6          old_node ← bucket.next
7          IF old_node ≠ NULL THEN
8              for(i ← 0; i < list_length; i++)
9                  IF CompareAndSwap(old_node.key[i],NULL,key) = NULL THEN
10                     RETURN true
11         new_node ← malloc sizeof(entry_list)
12         new_node.init()
13         new_node.next ← old_node
14         IF CompareAndSwap(bucket.next,old_node,new_node) ≠ old_node
15             free new_node
```

Figure 6. Insert algorithm of lock-free chained hashing

Figure 7 presents the *lookup* algorithm for lock free chained hashing. If the algorithm finds out the given *key* in the target bucket immediately, it returns the corresponding *value*. Otherwise, it scans the corresponding linked list while checking whether the *key* of each bucket is the same with the given *key*. If the algorithm finds out such bucket in the linked list, then returns the corresponding *value*. Otherwise, it returns *KEY_NOT_FOUND*.

```
1  Lookup(key)
2      ent ← hashtable[hash(key)]
3      IF ent.key = key THEN
4          RETURN ent.value
5      IF ent.next THEN
6          list ← ent.next
7          WHILE (list)
8              for(i  ← 0; i < list_length; i++)
9                  IF list.key[i] = key THEN
10                     RETURN list.value[i]
11             list ← list.next
12         RETURN NULL
13     ELSE
14         RETURN KEY_NOT_FOUND
```

Figure 7. Lookup algorithm of lock-free chained hashing

3.3. Refinable lock-based hopscotch hashing

Figure 8 presents the data structure of refinable lock-based hopscotch hashing. Each bucket consists of 8-byte *key*, 8-byte *value*, 8-byte *hop information*, 8-byte *lock*, 40-

byte *pthread_mutex*, and 48-byte *pthread_cond*. At first, the *key*, *value*, and *hop information* of each bucket are initialized with *NULL*. Here, we denote that the size of *hop information* is 8 bytes, not *H* bits, for memory alignment.

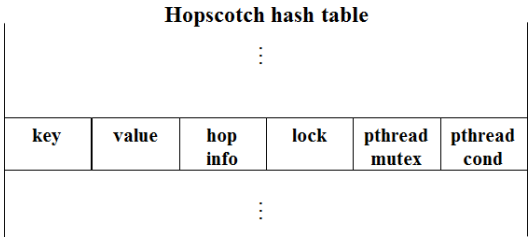

Figure 8. Data structure of refinable lock-based hopscotch hashing

The insert algorithm of hopscotch hashing is described in Figure 9. In this paper, we set *ADD_RANGE* to 128 and *HOP_RANGE* to 32. The algorithm first finds out the first empty bucket with linear probing in Lines 8~11. If the empty bucket is located within *HOP_RANGE*, the data is inserted without displacement in Lines 14~18. Otherwise, the algorithm performs repeatedly data displacement by calling the *Find_closer_bucket* function, which finds out the bucket to be displaced with the closest empty bucket.

```
1  Insert(key, value)
2      start_bucket ← hash(key)
3      hashtable[start_bucket].lock()
4      IF lookup(key) THEN
5          hashtable[start_bucket].unlock()
6          RETURN false
7      free_dist ← 0
8      for(; free_dist < ADD_RANGE; free_dist++)
9          IF hashtable[start_bucket].key = NULL THEN
10             BREAK
11         start_bucket++
12     IF free_dist < ADD_RANGE THEN
13         DO
14             IF free_dist < HOP_RANGE THEN
15                 hashtable[start_bucket].hopinfo |= 1 << (HOP_RANGE - free_dist)
16                 hashtable[start_bucket] ← <key, value>
17                 hashtable[start_bucket].unlock()
18                 RETURN true
19             Find_closer_bucket(start_bucket, free_dist, found)
20         WHILE found = false
21     hashtable[start_bucket].unlock()
22     RETURN false
```

Figure 9. Insert algorithm of hopscotch hashing

Figure 10 presents the *lookup* algorithm of hopscotch hashing. Since each of all *keys* in the hash table is stored within *HOP_RANGE* from its original target bucket thanks to the insert algorithm, the algorithm only needs to check the buckets within *HOP_RANGE* as in Lines 3~6. If it finds out such bucket, it returns value. Otherwise, it returns *KEY_NOT_FOUND*.

```
1  Lookup(key)
2      start_bucket ← hash(key)
3      for(i ← 0; i < HOP_RANGE; i++)
4          IF key = hashtable[start_bucket].key THEN
5              RETURN hashtable[start_bucket].value
6          start_bucket++
7      RETURN KEY_NOT_FOUND
```

Figure 10. Lookup algorithm of hopscotch hashing

4. Performance evaluation

4.1. Experimental data and experiment environment

The purpose of our experiments is to compare the performance of three multithreading cache-conscious hashing methods, linear hashing, chained hashing, and hopscotch hashing, and analyze their characteristics. We have measured the wall clock time of the insert and lookup operations while increasing the number of threads from 1 to 64. We repeated each test three times and used the average value.

For the experimental data, we generated uniformly distributed <key, value> pairs of from 2^{20} (*i.e.*, one million) to 2^{23} (*i.e.*, about eight millions). We denote the data set of 2^{20}, 2^{21}, 2^{22}, and 2^{23} pairs as 1MB, 2 MB, 4MB, and 8MB, respectively. Since the sizes of keys and values are 8 bytes, the size of the smallest data set, *i.e.*, 1MB, is 2^{20}x(8+8) = 16MB, and that of the largest data set, *i.e.*, 8MB, is 2^{23}x(8+8) = 128MB. In each experiment, we set the number of buckets of each hash table to double of the number of <key, value> pairs of the data set such that fill factor becomes 50%.

For the experimental environment, we used a 32-core machine with four Intel Xeon E7-2830 8-core 2.13GHz CPUs of 24MB L3 cache, 128GB memory, and SUSE Enterprise 64-bit Linux operating system. We compiled our source codes by using g++ (version 4.3.4). In order to avoid the unexpected effect of logical cores, we turned off the hyper-threading (HT) option of the system.

4.2. Insert operation performance

Figure 11 presents the performance results of the insert operation of three hash tables. Both lock-free linear hashing and lock-free chained hashing shows that the performance increases continuously as the number of threads increases. The performance of linear hashing of 64 threads is improved 19 to 24 times compared to that of a single thread. Similarly, the performance of chained hashing is improved 8 to 12 times, and that of hopscotch hashing is improved by up to 8 times.

Lock-free linear hashing shows the best performance for the insert operation, and lock-free chained hashing shows a comparable performance with that. Even though we have not presented the result in Figure 11, we have identified that non-cache-conscious chained

hashing, which allocates a node of a linked list by a bucket, shows much worse performance than linear hashing.

The performance of hopscotch hashing is not much improved compared with the other two hashing methods, as the number of threads increases. It is because hopscotch hashing is a lock-based method and requires displacement process for insertion, which might cause many cache misses. There exist the locking overhead and the thread contention among threads, especially, when the number of threads increases. Due to such overhead and contention, the performance for insertion even degrades when the number of threads is 32 or 64.

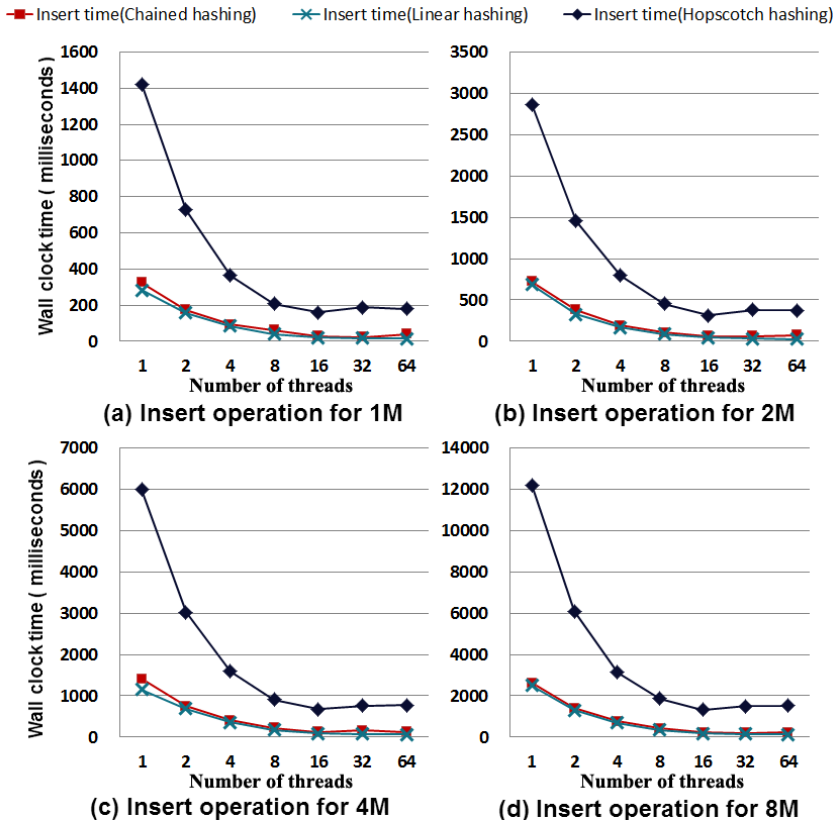

Figure 11. Results of the insert operation

4.3. Lookup operation performance

Figure 12 presents the performance results of the lookup operation of three hash tables. The performance of linear hashing of 64 threads is improved 11 to 22 times compared to that of a single thread. Similarly, the performance of chained hashing is improved 15 to 23 times, and that of hopscotch hashing is improved 14 to 20 times.

Unlike the insert operation, chained hashing shows the best performance for lookup, and linear hashing shows the worst performance among three hashing methods. Both insert and lookup algorithms of linear hashing are based on linear probing. So, the difference between the performances of insert and lookup operations of linear hashing is not much. However, chained hashing and hopscotch hashing have a relatively large overhead for the insert operation, which are allocating liked lists or displacement process, but a relatively small overhead for the lookup operation. Thus, the lookup performances of chained hashing and hopscotch hashing are much improved and become better than that of linear hashing.

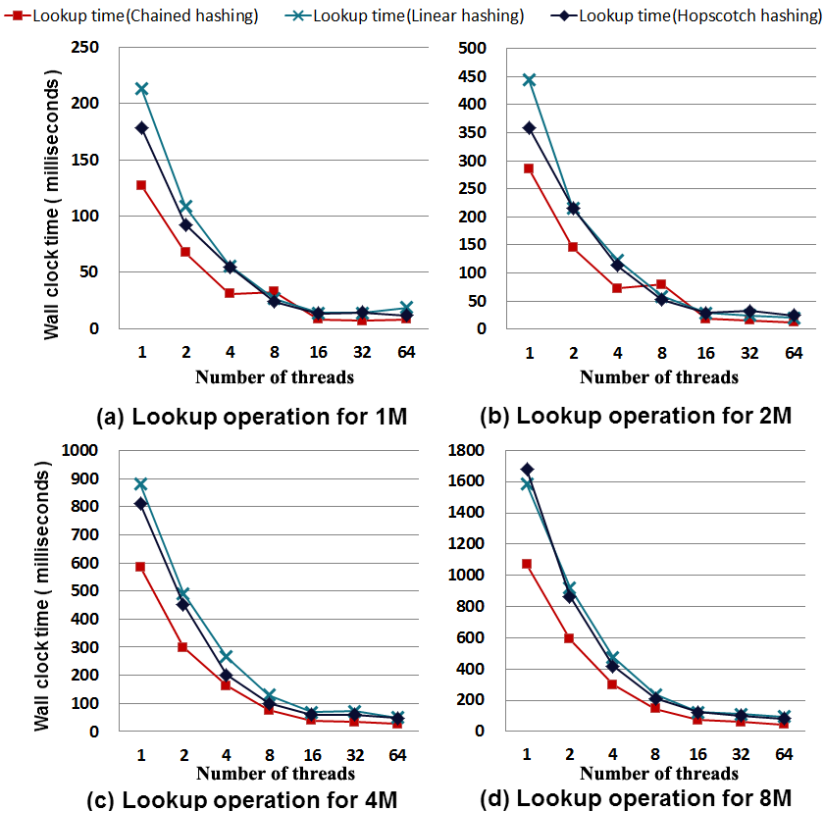

Figure 12. Results of the lookup operation

4.4. Hash table size

Figure 13 presents actual hash table sizes of three hash tables. For the 8MB data set, the size of linear hash table is $2^{24}\times(8+8)$ = 256MB, that of chained hash table including all linked lists is 385MB, and that of hopscotch hash table is $2^{24}\times(8+8+8+8+40+48)$ = 1.875GB. We note that linear hashing and chained hashing consumes much smaller memory than hopscotch hashing does, although they provide as good performance of the insert and lookup operations as hopscotch hashing does.

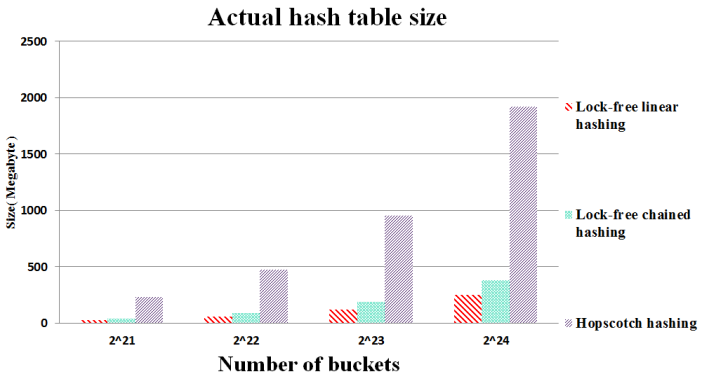

Figure 13. The actual sizes of the hash tables

5. Conclusions

In this paper, we have compared and analyzed the performance of three multithreading cache-conscious hashing methods, linear hashing, chained hashing, and hopscotch hashing, in terms of the insert and lookup operations. We have implemented each hashing method under the same implementation framework with the state-of-the-art techniques, especially CAS-based lock-free technique for linear hashing and chained hashing and refinable lock-based technique for hopscotch hashing. To the best of our knowledge, the work done by this paper is the first work that implements fairly those three hash methods and compares strictly them under the machine of a large number of cores (*i.e.*, 32) of the latest computer architecture. Experimental results using up to 2^{23} (*i.e.*, about eight millions) key-value pairs shows lock-free linear hashing is the best one for the insert operation, and lock-free chained hashing is the best one for the lookup operation. Hopscotch hashing shows the result that the performance of insertion rather degrades with 32 or 64 cores due to overhead from an excessive number of lock operations. In terms of actual sizes of hash tables, the size of hopscotch hash table is much bigger than those of other hash tables. That result means hopscotch hashing is practically not as good as other two hash methods that provide both small footprint and good performance.

Acknowledgements

This work was supported by the IT R&D program of MKE/KEIT. [10041145, Self-Organized Software-platform(SOS) for welfare devices].

References

[1] W. Litwin, "Linear hashing: a new tool for file and table addressing", Proceedings of 6th International Conference on Very Large Databases, **(1980)** October 1-3; Montreal, Canada.
[2] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, "Extendible Hashing-A Fast Access Method for Dynamic Files", ACM Trans. Database Syst., vol. 4, **(1979)**, pp. 315.
[3] M. Herlihy, N. Shavit and M. Tzafrir, "Hopscotch hashing", Distributed Computing, vol. 5218, **(2008)**, pp. 350.
[4] A. Stivala, P. J. Struckey, M. G. d. l. Banda, M. Hermenegildo and A. Wirth, "Lock-free Parallel Dynamic Programming", J. Parallel and Distributed Computing, vol. 70, **(2010)**, pp. 839.
[5] M. Herlihy and N. Shavit, "The Art of Multiprocessor Programming", Morgan Kaufmann, Burlington, **(2008)**.
[6] Ú. Erlingsson, M. Manasse and F. McSherry, "A cool and practical alternative to traditional hash tables", Proceedings of the 7th Workshop on Distributed Data and Structures, **(2006)** January 4-6; Santa Clara, United States.
[7] H. Gao, J. F. Groote and W. H. Hesselink, "Lock-free dynamic hash tables with open addressing", Distributed Computing, vol. 18, **(2005)**, pp. 21.
[8] PVS Language Reference, http://pvs.csl.sri.com.
[9] C. Purcell and T. Harris, "Non-blocking Hashtables with Open Addressing", Distributed Computing, vol. 3724, **(2005)**, pp. 108.
[10] O. Shalev, and N. Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables", J. the ACM, vol. 53, **(2006)**, pp. 379.
[11] D. E. Knuth, "The Art of Computer Programming", vol. 3, Sorting and Searching, Addison-Wesley Professional, Boston, **(1998)**.
[12] R. Pagh and F. F. Rodler, "Cuckoo hashing", J. Algorithms, vol. 51, **(2004)**, pp. 122.

Authors

Euihyeok Kim

Euihyeok Kim received the BS degree in computer engineering from Changwon National University in 2011. He is a master degree student in the Department of Information and Communication Engineering at DGIST, and he is a lab member of the InfoLab. He is interested in database and data mining area.

Min-Soo Kim

Min-Soo Kim received the BS, MS, and PhD degrees in computer science from KAIST in 1998, 2000, and 2006, respectively. He is currently an assistant professor in the Department of Information and Communication Engineering at DGIST. In the past, he worked as a postdoctoral research associate at Department of Computer Science, University of Illinois at Urbana-Champaign (UIUC), working on data mining area with Prof. Jiawei Han. He also worked as a postdoctoral research staff at IBM Almaden Research Center, working on developing the query engine of IBM Smart Analytics Optimizer for DB2 for z/OS. He is currently leading InfoLab at DGIST, and the research areas of InfoLab include database, data mining, machine learning, bioinformatics, and neuroinformatics.