Performance Analysis of Lock-Free Linear Hashing for Multi-core CPUs

Euihyeok kim and Min-soo Kim

Department of Information & Communication Engineering
Daegu Gyeongbuk Institute of Science & Technology (DGIST), Hyeonpung-Myeon
Dalseong-Gun, Daegu, Korea
{keh, mskim}@dgistac.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 0(1), it has been widely used in various areas such as databases, bioinformatics, and embedded systems. 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 a linear hashing method and analyzes its performance under Intel 32-core CPU microarchitecture. We implement the method in a *lock-free* manner, especially by using the *compare-and-swap(CAS)* operation, which is the state-of-the-art technique for parallelism. To the best of our knowledge, the work done by this paper is the first work analyzing the performance of a lock-free linear hash table under the Intel microarchitecture of a large number of cores. Experimental results using data of 2^{23} (i.e., about eight millions) key-value pairs shows the performance of both *insert* and *lookup* operations is improved by up to more than 20 times compared with that of a single-threaded method. Through experiments, we also uncover the potential performance problem of the CAS-based parallelism of the Intel microarchitecture.

Keywords: linear hashing, parallelism, lock-free hash tables, multicores, Intel microarchitecture, compare-and-swap.

1 Introduction

A hash table is a fundamental data structure implementing an associative memory that maps a *key* to its associative *value.* Due to its excellent lookup performance of 0(1), it has been widely used in a lot of practical software systems. Meanwhile, since the paradigm of micro-architecture design of CPUs is shifting to on-chip multi-core processors, it becomes more and more important to develop parallel algorithms for fully exploiting the performance of hardware.

In this paper, we implement a linear hashing method, which is practically the most widely used method, and analyzes its performance under Intel CPU of Nehalem microarchitecture. We implement the method in a *lock-free* manner, especially by using the *compare-and-swap* (CAS) operation, which is the state-of-the-art technique for parallelism. To the best of our knowledge, the work done by this paper is the first work analyzing the performance of a lock-free linear hash table under the Intel microarchitecture, especially of a large number of cores. The previous work has done the performance analysis only under the other microarchitectures of AMD, IBM, Toshiba and SUN[2].

The rest of this paper is organized as follow. Section 2 describes existing hashing methods including a lock-free linear hashing method. Section 3 evaluates the performance of lock-free linear hashing under the Intel microarchitecture. Section 4 concludes the paper.

2 Related Work

This section we briefly introduce several major hashing methods: chained hashing, linear hashing, cuckoo hashing[5], and hopscotch hashing[6]. Chained hashing is general hashing method where the chained items stored as linked list. Linear hashing is also one of traditional hashing method and practically the most widely used method, that effectively control chaining problem using linear probing. Cuckoo hashing inserts and lookups items by using two alternative positions instead of using just one position, but, insertion sequence of displacements can be cyclic. Hopscotch hashing, its main idea is that "moves the empty slot towards the desired bucket"[6], so that it can use CPU cache line well to speed up the performance. Among four hashing methods, we describe the parallel version of linear hashing in more detail since it is practically the most widely used one.

2.1 Implementation of lock-free linear hashing

Several lock-free versions for linear hashing have been proposed so far. Gao et al.[1] proposed the dynamic algorithm that allows the hash table to grow and shrink as needed. Their algorithm used PVS[4] for the safety property that no process can access to de-allocated memory. One weak point of their algorithm is that, without PVS, it is not sure to archive such size and complexity like they insist[1]. Purcell and Harris[3] proposed the algorithm that has no garbage collection and low operation foot print. However, the problem is that it cannot implement a dictionary, storing a value with a key, as there is no way to replace keys[3]. Stivala et al.[2] proposed the method based on CAS operations. Since Stivala et al. is the most efficient method among [1], [2], and [3] due to atomic instructions, we implemented their lock-free linear hashing algorithm. The pseudo code for the *insert* and *lookup* operations are described in Fig. 1.

```
par_insert(key,value)
  ent .-get_entry(key)
    if ent = NULL then
      error_exit("hash table full")
    if ent.key = NO_KEY then
      if CompareAndSwap(ent.key,NO_KEY,key)*NO_KEY then
        return par_insert(key,value)
  ent.value.-value
  return TRUE

par_lookup(key)
  ent 4-get_entry(key)
  if ent * NULL A ent.value * NO_VALUE then
    return ent.value
  else
    return KEY_NOT_FOUND

get_entry(key)
  h+-hash(key)
  ent<--hashtable[h]
  probes <-0
  while probes < TABLE_SIZE - 1 A ent.key * key A ent.key * NO_KEY
do
    probes <-probes + 1
    h <--(h + 1) mod TABLE_SIZE
    ent <-hashtable[h]
  if probes   TABLE_SIZE - 1 then
    return NULL
  else
    return ent
```

Fig. 1. Pseudo code for the insertion and lookup operation on lock-free linear hashing [2].

3 Performance Evaluation

3.1. Experimental data and Environment

The purpose of our experiments is to identify the trends and potential problems of the performance of a lock-free linear hashing method under the Intel microarchitecture. We implemented three functions *par insert, par lookup* and *get entry* in the C programming language and measured the wall clock time of *insert* and *lookup* operations while increasing the number of threads from 1 to 64. Each test was repeated three times and the average value was used to draw figures.

For the experimental data, we generated and used random data sets of uniform distribution. We varied the number of data elements (i.e., key-value pairs) from 2^{20} *(i.e.,* one million) to 2^{23} *(i.e.,* about eight millions). We used 8 bytes as the key size and 8 bytes as the value size. Thus, the size of the smallest data set is 2^{20}x(8+8) = 16MB, and that of the largest data set is 2^{23}x(8+8) = 128MB.

For the simplicity, we used a linear hash table supporting only insertions and lookups. Hence, updates and deletions are not required. Since it is generally known that the typical fill factor of linear hash tables is 50%, we set the size of hash table, *i.e.*, the number of buckets of each hash table to two times of the number of data elements. That is, the size of the largest hash table is 128x2=256MB.

For the experimental environment, we used a system with four Intel Xeon E7-2830 8-core 2.13GHz CPU *(i.e.*, total 32 cores) of 24MB L3 cache, 128 GB memory, and SUSE Enterprise 64-bit Linux operating system. We compiled our source codes by using gcc (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.

3.2. **Results of the Experiments**

Fig. 2 and 3 show the performance of the insert and lookup operations of the lock-free linear hash table under the Intel microarchitecture. The performance of both operations is continuously improved as the number of threads increases. When the number of threads is 64, the performance of both insert and lookup is improved by up to more than 20 times compared with a single-threaded hash table. The wall clock time also linearly increases as the size of data set increases.

We note that the wall clock time of the insert operation in Fig. 2 is just a little bit larger than that of the lookup operation in Fig. 3. This result means there is almost no overhead when using the CAS instruction in the *par insert* function. We also note that the performance is not much degraded even when the size of hash table becomes larger than that of L3 cache, so the whole hash table isn't fit into the CPU cache. The total size of L3 caches of the experimental system is 4x24=96MB. When the numbers of data elements are 2^{22} and 2^{23}, the sizes of hash tables are 128MB and 256MB, respectively, the wall clock time of those data sets shows simply two times and four times, respectively, of that of the data set of 2^{21}.

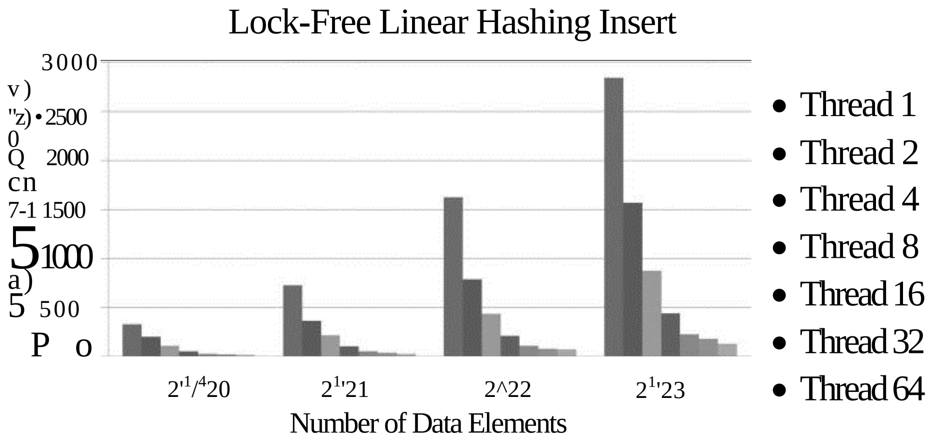

Fig. 2. Insert performance for lock-free linear hashing.

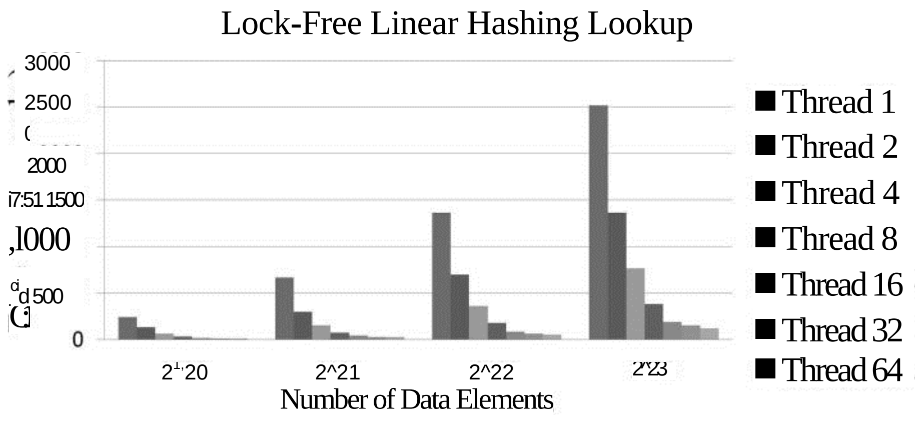

Fig. 3. Lookup performance for lock-free linear hashing.

One interesting result is the speedup ratio of the performance is suddenly degraded for both operations when the number of threads becomes larger than 16 (i.e., 32 and 64). Since the number of physical cores is 32, and also, there is almost no overhead of CAS instructions, we expected the performance increases linearly at least up to 32 threads. We consider here might be a certain performance problem unknown so far of the CAS-based parallelism of the Intel microarchitecture.

4 Conclusions

We analyzed the performance of the lock-free linear hashing based on CAS under Intel 32-core CPU microarchitecture while varying the number of data elements and the number of threads. Experimental results shows the performance of both *insert* and *lookup* operations is improved by up to more than 20 times compared with that of a single-threaded method. As a result, it can insert or lookup more than eight millions data elements within 200 milliseconds. We have also found out the potential performance problem of the CAS-based parallelism of the Intel microarchitecture.

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. Gao, H., Groote, J. F., Hesselink, W. H.: Lock-free dynamic hash tables with open addressing. DISC. 18, 21--42 (2005)
2. Stivala, A., Stuckey P. J., Banda,M. G. d. 1., Hermenegildo M., Wirth A.: Lock-free Parallel Dynamic Programming. J. Par. Dis. Comp. 70, 839--848 (2010)
3. Purcell, C., Harris, T.: Non-blocking hashtables with open addressing. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 108--121. Springer, Heidelberg (2005)
4. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference, http://pvs.csl.sri.com
5. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algo. 51, 122--144 (2004)
6. Herlihy, M., Shavit, N., Tzafrir, M..: Hopscotch hashing. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 305--364. Springer, Heidelberg (2008)