HiCHT: High-performance Compact Hash Table

Biao Wang[1], Pengcheng Wang[2], Jifan Shi[1], Weiguang Wang[2], and YunXu[1](✉)

[1] School of Computer Science and Technology, University of Science and Technology of China, Hefei, China
{wangbiao0814,shijifan8}@mail.ustc.edu.cn,xuyun@ustc.edu.cn
[2] Huawei Technologies, Nanjing, China
{wangpengcheng25,weiguang.wang}@huawei.com

Abstract. Hash tables represent a fundamental data structure in the field of information retrieval. The optimization of space and performance for this structure is crucial and urgent. Clerry first proposed a compact hash table that allows the use of space close to the information-theoretic lower bound by storing only the quotient part of each key. However, it sacrifices the performance of basic operations due to the overhead of resolving hash conflicts. Subsequent optimized methods use the chaining strategy to handle hash collisions, but they suffer from the slow comparison of each element in the linked list. In this paper, we present HiCHT, a hash table with chaining buckets to minimize the overhead of handling hash collisions. Each bucket stores only the quotient of the key to reduce space and introduces a coarse fingerprint filter to identify elements in the linked list quickly. The experimental results demonstrate that HiCHT is significantly superior to all existing variants based on the compact hash table in terms of performance for basic operations. Moreover, its memory usage is also competitive with the best-performing method.

Keywords: Dynamic Dictionary · Compact Hash Table · Efficient Access

1 Introduction

Compact data structures can represent data in reduced space while allowing querying it in the compressed form, which is of great benefit to existing applications equipped with parsimonious storage or dealing with large-scale data [17]. In recent years, the compact representations of data structures such as arrays, dictionaries, texts, and graphs have shown to be extremely memory-efficient [5, 6, 1]. Representing dynamic hash tables compactly is of critical importance, as they are fundamental data structures in various applications. However, the advantages in the space of proposed methods come with sacrificing the performance of basic operations. Achieving high-performance access to compact hash tables while preserving their memory advantages has become a critical challenge.

Here we consider the problem of compactly representing a dynamic hash table. Given a dynamic set S storing key-value pairs $(x,\ y)$, where the key x is from *universe* $U = [0, u)$ and the value y comes from $\{0, 1, \cdots, 2^v - 1\}$,

where u is an upper bound on the set S of integers and v is the bit-width of each value. All key-value pairs in the set S are distinct. A dynamic hash table for S needs to support the following operations: *Insert(S, x, y)* inserts a key-value pair (x, y) into the set S if (x, y) is not in it; *Lookup(S, x)* returns y if the key-value pair (x, y) belongs to the set S; *Delete(S, x)* removes the key-value pair (x, y) from S. Typically, a dynamic hash table cannot use less space than the information-theoretic lower bound for storing the S, which is $\mathcal{B}(u,n)+nv = \log_2 \binom{u}{n} + nv = n\log_2 u - n\log_2 n + O(n) + nv$ bits [21], where n denotes the number of keys in the set S. In the following, we abbreviate $\mathcal{B}(u,n)$ by $\mathcal{B}$ and the logarithm to base 2 by lg for simplicity.

Rajeev et al. [21] and Yuriy et al. [3] proposed compact dynamic hash tables using $\mathcal{B} + o(\mathcal{B})$ bits while supporting lookup in $O(1)$ time and insertion and deletion in $O(1)$ expected amortized time. Recently, new advance has been made by Bercea et al. [4] and Liu et al. [15], which can achieve the $O(1)$ worst time operations with high probability, but their methods have restrictions on the storage of the value and the support of delete operation, respectively. Moreover, these data structures are rather complex and provide only theoretical results.

Some researchers have proposed compact hash tables (CHTs) to solve the problem. The core idea of CHT is to save memory by storing only the quotient part of each key, and the unstored part can be inferred from the location where the key is originally hashed. Currently, CHTs can be implemented using either open addressing or chaining. Clerry [7] and Poyias [20] first proposed CHTs based on linear probing, which require storing and maintaining additional information to handle the hash collisions, as keys may not be placed in their corresponding hashing positions. This incurs extra overhead when performing basic operations, and the issue is exacerbated when setting a high load factor to ensure low space consumption. For example, Clerry's method needs to move a lot of data to perform an insert operation, especially when the capacity of the hash table is insufficient and the hash collisions occur frequently. In contrast to compact hashing via open addressing, later designs of CHT using chaining strategies [12] map the conflicting keys into the same bucket and store them in a linked list. However, the linked list has obvious pointer-chasing overhead during accesses, and it also introduces extra space overhead of pointers. Increasing the size of keys stored in the linked node can reduce the overhead of pointers, but results in poor performance due to the slow query process within each bucket.

Towards the above shortcomings, we present HiCHT, which achieves both space-saving and high performance for basic operations. Similarly, we resolve the conflicts of keys using chaining buckets. As stated before, to reduce the memory consumption of pointers, the linked list needs a larger capacity to store more keys. If we continue to search for the keys sequentially, the amount of data accessed will increase, causing more cache misses. To fully utilize the cache, we add a summary structure to each bucket that contains the fingerprints of the keys. With these fingerprints, we can filter out many mismatched elements, and this approach has proven to be more cache-friendly than the serialized lookup strategy in many designs [18, 16]. The fingerprint is obtained directly from the remainder, without

introducing additional space overhead. In this way, each bucket can hold more keys while providing high performance for basic operations. Meanwhile, HiCHT stores only the quotients of keys as well, and the array of quotients in each bucket is allocated a reasonable size to use the space more efficiently. Moreover, we further improve the lookup performance by parallelizing the comparison of the fingerprints using SIMD instructions.

In general, our main contributions are as follows.

- We design and implement HiCHT, a high-performance CHT using chaining buckets to avoid extra overhead for handling hash conflicts, which exploits *quotienting* for space reduction and *fingerprinting* for search acceleration.
- We prove that HiCHT uses $(1 + O(1/\lg\lg u))(\mathcal{B} + nv) + O(n \lg\lg u)$ bits, while supporting *lookup* and *insertion/deletion* in $O(\lg\lg u)$ expected time and $O(\lg\lg u)$ amortized time, respectively.
- Using various datasets and workloads, HiCHT outperforms the existing best CHT by up to 2.35× in terms of lookup performance, and its memory consumption is also competitive with the best-performing method.

2 Background and Related Works

In this section, we will first briefly introduce the concept of *quotienting*, and then review the existing designs of compact hash tables.

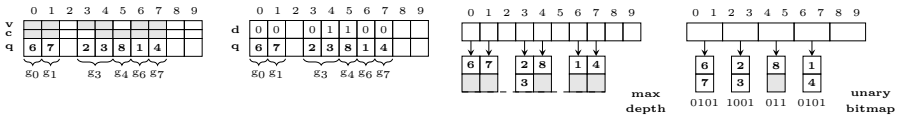
2.1 Basic Idea

In particular, a *quotienting* based hash table needs to supply two functions. (1) A hash function $h(x)$ that maps each key to a position in the hash table. (2) A quotient function $q(x)$ such that the key x can be determined once $h(x)$ and $q(x)$ are given. Taking division hashing as an example, we let $h(x) = x \bmod m$ and $q(x) = \lfloor x/m \rfloor$. $h(x)$ identifies the location of the key x in the hash table, and we only need to store $q(x)$ explicitly.

2.2 Related Work

We focus mainly on the practical work related to CHT, and they can be categorized into open-addressing and chaining-based methods. In the following, we discuss and graphically illustrate these approaches in detail, and the existing practical designs are depicted in Figure 1.

Compact Hash Table Using Open Addressing. We start with Clerry's CHT [7] using bidirectional linear probing [2], as shown in Figure 1(a), keys with the equal remainder are arranged into the same group and conflicts inside each group are resolved through quotients. Besides, the bitmaps V and C are introduced to obtain the start and end positions of each group, where $V[i]$ indicates whether group g_i exists, and each non-zero bit in C marks the end position of a group. To search for a key, we start from the initial hashing position and probe



(a) (b) (c) (d)

Fig. 1. Compact hash table variants storing set $S = \{16, 23, 33, 47, 60, 71, 84\}$, where $h(x) = x \bmod 10$ and $q(x) = \lfloor x/10 \rfloor$. (a) clerryCHT. (b) dCHT. (c) bucketCHT. (d) groupCHT.

to the left in the quotient array until an empty position is found, by counting the number of 1s in V and C bitmaps from the obtained empty position to the right, we can find the position of corresponding group. For the key 16, we have $h(16) = 6$ and $q(16) = 1$. $V[6] = 1$ so the search continues, and the first empty position by searching to the left of the quotient array is at position 2. Between position 2 and 6 we have $V[3] = 1$, $V[4] = 1$, and $V[6] = 1$, and by counting back from the empty position 2, three C bits are 1 at position 4, 5, and 6. Therefore, the g_6 starts at the position 6 and ends at the position 6. Finally, we check whether the quotient array has a quotient 1 in this interval. Poyias et al. [20] proposed a design that replaces the bitmaps V and C with an array d of displacements, which keeps the number of probes for each key starting from the initial hashing address. For example, the key 84 in Figure 1(b) is stored at position 5, which is at a displacement of 1 from its initial position. Each probe calculates the initial position by the displacement array and gets the complete key for comparison by combining it with the corresponding quotient.

The open-addressing strategy requires additional space to handle hash conflicts. As the index space approaches the theoretical information lower bound, the number of probes required to perform basic operations increases, leading to a sharp decrease in performance for both Clerry's and Poyias' methods.

Compact Hash Table Using Chaining. Köppl et al. [12] design practical CHTs using bucketing. Figure 1(c) gives an example of the designed data structure, which uses an array to store conflicting keys. Each array has a predefined maximum size. When any array reaches the maximum size, the hash table is augmented and rehashed [13]. To perform a lookup operation, the quotients in the mapped bucket are scanned. Figure 1(d) exhibits a more compact structure by grouping consecutive buckets in Figure 1(c). It uses a bitmap written in unary encoding [10] to distinguish between different groups. Specifically, if the number of keys in the buckets belonging to a group is $s_0, s_1, \cdots, s_k$, the corresponding bitmap is $0^{s_0}10^{s_1}1...0^{s_k}1$. To look up a given key x, assuming that there are m buckets in the hash table and g consecutive buckets are merged into one group, we first compute that it belongs to the j-th bucket in group i, where $h(x) = x \bmod m$, $i = \lfloor h(x)/g \rfloor$ and $j = h(x) \bmod g$. Then, the location of the bucket to be searched in the group can be obtained by iterating over the unary encoding bitmap. Finally, all quotients in the corresponding bucket are scanned.

The bucketCHT uses $\mathcal{B} + O(n \lg \lg u)$ bits and supports both lookup and insertion in $O(\lg(u/n))$ expected time. The grouping strategy reduces the space-bound to $\mathcal{B} + O(n)$ bits and performs lookup in $O(1)$ expected time and insert in $O(\lg(u/n))$ worst-case time. Köppl's methods are proven to be competitive in terms of memory consumption compared with previous designs. However, the long scan time in the bucket becomes a major bottleneck in improving their performance for basic operations.

3 High-performance Compact Hash Table

This section presents the design of HiCHT. We first describe the organization of its data structures. Then, the lookup, insertion, and deletion algorithms for HiCHT are given, followed by an asymptotic analysis of HiCHT's space and performance. Finally, the synchronization protocol of HiCHT is shown.

3.1 Data Structures of HiCHT

In this section, we describe in detail all the data structures in HiCHT and give specific examples for illustration. Finally, we summarize the main advantages of HiCHT over other methods.

As shown in Figure 2, the bucket in HiCHT consists of fingerprints, a bitmap, a pointer to the quotients, and a pointer to the next bucket. Each bucket can hold multiple keys, where the quotients of keys are stored in a quotient array and can be accessed via the pointer within the bucket. The fingerprints are used to accelerate the search procedure, while each bit in the bitmap identifies whether the key at the corresponding position is valid or not. We set the capacity of a bucket $c = \Theta(\lg u)$ and the number of buckets $m = (n/c)$, where u is the universe of keys and n is the number of keys in the set. The size of the hash table is therefore $s = mc = n$. If the number of keys in a bucket exceeds its capacity, the part that exceeds the capacity must be stored in a new bucket, which can be accessed by the pointer in the original bucket.

Each bucket is further divided into c sub-buckets, and any sub-bucket can be identified by a fingerprint of $\lg c$ bits. Therefore, the fingerprint array occupies $c \lg c$ bits. The quotient array adaptively adjusts its size according to the number of keys stored in the current bucket, and we increment the size of the quotient array in base units of $i = \Theta(\lg u / \lg \lg u)$. This ensures that the size of the quotient array is increased gradually, thus reducing the space overhead caused by over-allocation.

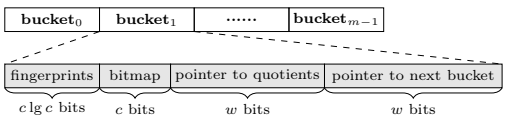


Fig. 2. The basic structure of HiCHT. It maps the keys of set S to buckets.

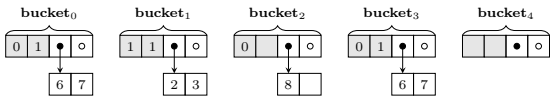


Fig. 3. Illustration of how HiCHT stores the set of keys $S = \{16, 23, 33, 47, 60, 71, 84\}$. For a given key x, x will be mapped to $bucket_j$ where $h(x) = x \bmod s$, $q(x) = \lfloor x/s \rfloor$ and $j = \lfloor h(x)/c \rfloor$. Moreover, we define $f(x) = h(x) \bmod c$ as the fingerprint of key x and store it in the bucket. Here we set $s = 10$ and $c = 2$. The values in the shaded boxes and the values pointed to by the pointer correspond to the fingerprints and the quotients, respectively.

Figure 3 gives an example of HiCHT, and there are five buckets for HiCHT instance. We set $s = 10$ and $c = 2$ for simplicity. According to the definition, for key 84, we have $h(84) = 84 \bmod 10 = 4$. Therefore, key 84 is mapped to $bucket_2$ as $\lfloor h(84)/2 \rfloor = 2$. We then compute $q(84) = \lfloor 84/10 \rfloor$ and $f(84) = h(84) \bmod 2$, which are 8 and 0, respectively. All this information will be stored in $bucket_2$.

Assuming that each element in the quotient array is stored in l bits and the bit width of an integer is w. To use the right amount of space, we allocate an array of $\lceil kl/w \rceil$ integers. To support the key-value pair storage pattern, the *value* of each key is added to the quotient field, and we can still separate the *quotient* and *value* from this field using simple shift and logical operations. This design is more cache-friendly in our approach since only a few elements have to be visited when performing basic operations.

The data structure and organization of HiCHT offer several specific advantages. First, the chaining buckets structure requires no extra overhead to recover the complete key compared to open addressing methods. Second, the chaining strategy enables dynamic allocation, while others may need to rebuild the hash table from scratch, such as the bucketCHT we mentioned before. Third, each bucket keeps an array of fingerprints to avoid unnecessary comparisons and can be further accelerated through SIMD.

3.2 Algorithms for Basic Operations

Next, we present the algorithms for lookup, insertion, update, and deletion in HiCHT.

Lookup: The pseudocode for the lookup is shown in Algorithm 1. First, we get the bucket of the search key and the corresponding quotient and fingerprint through some basic calculations (line 1), and the $\_mm\_set1\_epi8$ function will expand a 1-byte fingerprint into a vector by making multiple copies. Then the chaining list of buckets is traversed (lines 2-12). The $\_mm\_cmpeq\_epi8$ and $\_mm\_movemask\_epi8$ functions perform a parallel comparison and store the result of the comparison in a bitmap $bitfield$, and we can get which fingerprints match through the 1s in the bitmap $bitfield$. The keys that match the fingerprint will be further compared with the quotient field, and if the quotient is also consistent, the lookup for the given key returns true (lines 5-11).

Algorithm 1: Algorithm for lookup.

Input: *buckets*, *key*, size of the hash table s, capacity of the bucket c
Output: *true* or *false*

```
1  r = key % s, q = key / s, f = (r & (c - 1)), r = (r ≫ lg c), b = buckets[r];
2  while bucket ≠ NULL do
3  |   cmp = _mm_cmpeq_epi8(_mm_set1_epi8(f), b.fingerprints);
4  |   bitfield = _mm_movemask_epi8(cmp) & b.bitmap;
5  |   while bitfield ≠ 0 do
6  |   |   idx = ctz(bitfield);                    /* count trailing zero */
7  |   |   if b.quotients[idx] == q then
8  |   |   |   return true;
9  |   |   end
10 |   |   bitfield = bitfield⊕(1 ≪ idx);
11 |   end
12 end
13 return false;
```

Insertion/update/deletion: The insertion is similar to the lookup operation. As shown in Algorithm 2, the search procedure in the chaining list is the same as in the lookup. If the key to be inserted already exists, we return a status that the inserted key is in the hash table (lines 2-4). Otherwise, the key is added to the empty entry of a chaining bucket, which may increase the size of the quotient array or create a new bucket(lines 5-16). The update operation can be implemented based on the insert operation, which only requires the modification of the payload corresponding to an existing key. The delete operation first searches the hash table and finds the location of the corresponding key. Then the deleted key is marked as invalid using the bitmap in the bucket. At the end of the delete operation, the bitmap of the bucket is used to determine whether the difference between the capacity of the current bucket and the number of elements stored is i. If so, the data of all chaining buckets is merged into new chaining buckets.

3.3 Asymptotic Analysis

In this section, we begin with the worst-case space consumption of HiCHT, taking into account the dynamic memory allocation model [21]. Then we analyze the performance of HiCHT using a simple uniform hashing assumption, which means that keys are uniformly hashed to different buckets [12, 3].

Memory consumption. The memory footprint of HiCHT includes the memory occupied by all allocated buckets and quotient arrays. Considering that there are n key-value pairs (x, y), where the key x comes from the universe $[0, u)$ and the value y is from $\{0, 1, \cdots, 2^v - 1\}$, assuming that the capacity of a bucket is set to $c = \Theta(\lg u)$, the incremental size of any quotient array is $i = \Theta(\lg u / \lg \lg u)$ and the number of all allocated buckets is T. In addition, the initial number of buckets is m and the size of the hash table is s, where

Algorithm 2: Algorithm for insertion.

Input: *buckets*, *key*, size of the hash table s, capacity of the bucket c
Output: *status*

```
1  r = key % s, q = key / s, f = (r & (c - 1)), r >>= lg c, b = buckets[r];
2  if lookup_bucket(b, f, q) then
3  |  return existed;
4  end
5  while b ≠ NULL do
6  |  if b.bitmap == (1 << lg c - 1) then
7  |  |  if b.next == NULL then
8  |  |  |  b.next = generate_empty_bucket();
9  |  |  |  insert_into_empty_bucket(b.next, f, q); return inserted;
10 |  |  end
11 |  |  else
12 |  |  |  b = b.next; continue;
13 |  |  end
14 |  end
15 |  idx = ctz(¬b.bitmap); insert_into_bucket(b, f, idx); return inserted;
16 end
```

$m = (n/c)$ and $s = mc = n$. Furthermore, we assume that each pointer occupies $w = \lg u$ bits using the word RAM model.

Proposition 31 *For n key-value pairs into HiCHT, the number of all allocated buckets $T < 2m$.*

Proof. Assuming that the j-th chaining list maintains s_j key-value pairs, then the number of allocated buckets in j-th chaining list is $T_j = \lceil s_j/c \rceil$. Therefore, we have $n = \sum_{j=0}^{m-1} s_j, T = \sum_{j=0}^{m-1} T_j = \sum_{j=0}^{m-1} \lceil s_j/c \rceil$. According to the property of the ceiling function, we have $\lceil s_j/c \rceil < s_j/c + 1$. Taking one step further, we have $T = \sum_{j=0}^{m-1} T_j = \sum_{j=0}^{m-1} \lceil s_j/c \rceil < \sum_{j=0}^{m-1} (s_j/c + 1) < 2m$.

Proposition 32 *HiCHT takes at most $(1+O(1/\lg\lg u))(\mathcal{B}+nv) + O(n \lg\lg u)$ bits for storing n key-value pairs.*

Proof. There are $T = \Theta(m) = \Theta(n/c)$ allocated buckets in HiCHT, each bucket takes $c \lg c + c + 2w$ bits and $c = \Theta(\lg u)$, so the memory consumption of all allocated buckets is $\Theta(n \lg\lg u)$ bits. Assuming that the j-th chaining list of maintains s_j key-value pairs and the incremental size of the quotient array is $i = \Theta(\lg u/\lg\lg u)$, then the number of allocated quotients with values in j-th chaining list is $\lceil s_j/i \rceil i$, and the total number of allocated quotients with values are $S = \sum_{j=0}^{m-1} \lceil s_j/i \rceil i < n + mi = (1 + \Theta(1/\lg\lg u))n$. Each quotient and value uses $\lg(u/s) + O(1) = \lg u - \lg n + O(1)$ and v bits, respectively. Then the overall space occupied by all quotients and values is $S \lg u - S \lg n + O(S) + Sv$ bits. Summed over the above memory cost, the overall space bound of HiCHT is $(1 + O(1/\lg\lg u))(\mathcal{B} + nv) + O(n \lg\lg u)$ bits.

Performance analysis. Now, we analyze the performance of HiCHT for lookup and insert operations based on the premise that keys are uniformly distributed into different buckets.

Proposition 33 *For n key-value pairs HiCHT, it uses $(1 + O(1/\lg\lg u))(\mathcal{B} + nv) + O(n \lg\lg u)$ bits while supporting lookup in $O(\lg\lg u)$ expected time and insert/delete in $O(\lg\lg u)$ amortized time.*

Proof. Let t_0 be the cost of computing the quotient, the remainder and the fingerprint, while let t_1, t_2 be the cost of checking the fingerprint array and comparing a single quotient, respectively ($t_0 = O(1), t_2 = O(1)$).

If there are k elements in a bucket, considering that keys with the same fingerprint will be further assigned to the same sub-bucket, k elements are uniformly distributed to c different sub-buckets when the fingerprint is used. The lookup operation in this bucket will examine at most $\lceil k/c \rceil$ fingerprint arrays and compare k/c quotients on average. Therefore, the average running time for a lookup in the chaining list with k elements is bounded by $t_0 + \lceil k/c \rceil t_1 + (k/c)t_2$. The expected time to perform a lookup operation in HiCHT is bounded by $t_0 + \lceil n/mc \rceil t_1 + (n/mc)t_2 = O(t_0 + t_1 + t_2)$. Using standard word-parallel tricks [11], the lookup time of the fingerprint array is proportional to the number of words, therefore we have $O(t_0 + t_1 + t_2) = O(1) + O(c \lg c / \lg u) = O(\lg\lg u)$.

Any insertion needs to look at the hash table first, which requires $O(\lg\lg u)$ time. Meanwhile, the insert operation also includes the cost of inserting the quotient into the quotient array. The size of the quotient array is initially set to i and is incremented by $i = \Theta(\lg u / \lg\lg u)$. Therefore, the cost of inserting k elements into the quotient array is $k + \sum_{j=1}^{\lfloor k/i \rfloor} ji < \frac{3}{2}k + \frac{k^2}{2i}$, and amortized cost is $O(1) + O(k/i)$. As we mentioned before, the average number of quotients in each quotient array is $k = \Theta(\lg u)$, so inserting a quotient into the quotient array costs $O(\lg\lg u)$ amortized time. Overall, the insertion can be completed in $O(\lg\lg u)$ amortized time. The delete operation is the opposite of the insert operation in that it involves shrinking the quotient array on demand, and through a similar analysis, we can see that the delete operation can also be done in $O(\lg\lg u)$ amortized time.

3.4 Concurrency Support

Traditional indexes are usually protected by fine-grained locks. Fine-grained locks make contention less likely, but acquiring or releasing a lock still requires a write to shared memory, making it less scalable. Lock-free data structures use a single compare-and-swap (CAS) operation to perform updates automatically. They do not acquire any locks and provide better scalability. A single CAS operation cannot synchronize HiCHT because inserts may require modification of data for multiple fields simultaneously.

To synchronize HiCHT, we combine optimistic locking and fine-grained locking. Instead of acquiring locks, the version numbers are used to detect contention. Our synchronization protocol does not require readers to acquire any locks, they

only need to detect if the version number is the same before and after the query. The writers need to acquire the lock before modifying the data and increase the version number before releasing the lock. Another critical issue of HiCHT's synchronization strategy is safe memory reclamation. The insertion may result in freeing old memory, and the deleting thread cannot reclaim the memory immediately because other threads may still be optimistically reading the memory. Therefore, we use epoch-based reclamation to free unused space. To realize these operations, each lock can be viewed as a w-bit version counter. The last bit of the version counter indicates the status of the lock, and the other bits indicate the current version. Acquiring a lock means setting the last bit of the version counter to 1 while releasing a lock causes the version counter to increment by 1.

Using fine-grained optimistic locks, the read-only access of the bucket in HiCHT is performed as follows: (i) read lock version (wait until the lock is free); (ii) access bucket; (iii) validate lock version. The write operation of the bucket is similar to traditional locking: (i) acquire lock; (ii) modify bucket; (iii) increment version and release lock.

Assuming that two threads T_1 and T_2 concurrently access the same bucket. If both threads attempt to modify the same bucket, they are unable to acquire locks concurrently. If both threads need to access the same bucket, there is no lock contention, and the accesses are considered valid. If thread T_1 seeks to modify the data in the bucket and thread T_2 seeks to access the data in the same bucket, when T_1 acquires the lock, T_2 will be blocked until T_1 completes the modification and releases the lock. Conversely, when T_2 is not blocked, the condition for T_2 to return the correct result is that T_1 does not modify the version number during the entire query process. This demonstrates that the concurrency protocol ensures the consistency of concurrent operations.

3.5 Implementation Issues

This section briefly describes the implementation details of HiCHT, including parameter settings and memory allocation strategies. For the parameters used in HiCHT, the maximum capacity of each bucket c is set to 64, and the incremental size i of the quotient array is 8.

4 Evaluation

In the following, we evaluate HiCHT and compare it with the state-of-the-art CHTs and other hash tables.

4.1 Setup

We conduct all experiments on a server running Ubuntu 22.04 LTS with Linux 5.15 kernel. All code in the experiment is compiled using GCC.

We compare HiCHT with the following practical implementations of CHT, including clearyCHT [7], dCHT [20], bucketCHT [12], and groupCHT [12], which

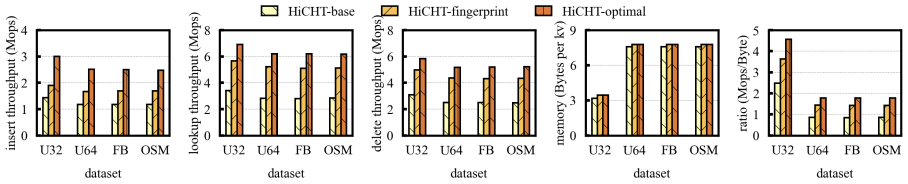


Fig. 4. Performance and memory consumption of different HiCHT designs.

are implementations of the schemes shown in Figure 1. In our experiments, for the chained design hash tables, we set the number of buckets of the hash table to $n/c\alpha$, where n denotes the number of keys, c denotes the capacity of a bucket, and α denotes the load factor.

Datasets. In our experiments, we use synthetic and real-world datasets to evaluate the performance and space of HiCHT and its competitors. All datasets used in our experiments have 60 million keys, where U32 and U64 contain 32-bit and 64-bit randomly generated integers, and they obey the uniform distribution in the range $[0, 2^{32})$ and $[0, 2^{64})$, respectively. While FB and OSM contain 64-bit user IDs and cell IDs.

Evaluation Metrics. Our evaluation of performance focuses on insert and lookup operations, and we use throughput to measure the average speed, which is expressed in million operations per second (Mops). The memory efficiency of the tested method is given by the average number of bytes consumed per key-value pair. We also define a composite metric, which is the ratio of performance/memory ($\frac{\text{inserts per sec + lookups per sec + deletes per sec}}{\text{average bytes per element}}$). The higher the P/M ratio of a method, the faster the method can perform basic operations using less space. To measure the evaluation metrics, we first load keys of the dataset into the hash table. Once the data has been loaded into the hash table, the memory consumption of each method is measured. Then we randomly insert new keys, delete old keys, and search keys in the dataset. For the sake of simplicity, the initial capacity of the hash table is sufficient to store all keys at a given maximum load factor of 0.85 so that no rehash operation is performed.

4.2 HiCHT Optimization

In this subsection, we evaluate how the proposed techniques and optimizations contribute to the performance and space of our method using various datasets.

Experiment 1 (Evaluation of Different Designs): HiCHT-base is the simplest CHT using chaining buckets, where each bucket stores the corresponding quotients, while HiCHT-fingerprint adds the fingerprint information to each bucket to avoid querying unmatched elements. HiCHT-optimal introduces SIMD instructions to compare the fingerprint information in parallel. As shown in Figure 4, we can see that that combining all optimizations can improve the performance of insert, lookup, and delete operation by 2.09 to 2.14 times, 2.02 to 2.21, and 1.89 to 2.10 times compared with the naive chaining design, respectively.

Table 1. Hardware performance counters per lookup (prefix HiCHT is omitted, and L3_TCM, TOT_CYC, MSP_BR and TOT_INS represent the number of L3 cache misses, total CPU cycles, mispredict branches and total instructions.).

Dataset	Method	L3_TCM	TOT_CYC	MSP_BP	TOT_INS
U32	base	4.9	888.6	1.1	965.9
	fingerprint	4.0	529.4	1.3	350.3
	optimal	3.5	437.9	1.8	138.5
U64	base	7.2	1053.4	1.1	1163.5
	fingerprint	4.2	565.4	1.3	365.0
	optimal	3.7	470.4	2.1	148.7

Figure 4 also shows that both HiCHT-fingerprint and HiCHT-optimal add about 0.2 bytes of space per key compared with HiCHT-base. This space overhead is acceptable and well worth it compared to the performance gains. Considering the performance/memory ratio, HiCHT-optimal is 1.83 to 2.10 times better than HiCHT-base. In summary, fingerprints and SIMD instructions are the key factors in the high performance of HiCHT. Meanwhile, the memory overhead introduced by fingerprints is almost negligible.

Experiment 2 (Lookup Performance Analysis): To further analyze the lookup performance improvement from these optimizations, we recorded some hardware performance counters in our experiment. Table 1 shows hardware performance counters per random lookup.

It can be seen that enabling *fingerprinting* reduces the number of L3 cache misses, total CPU cycles, and total instructions by 18.3% to 41.6%, 40.4% to 46.3%, and 63.7% to 68.6% compared to HiCHT-base, respectively. Combining *fingerprinting* and SIMD instructions, the number of L3 cache misses, total CPU cycles, and total instructions are reduced by 28.5% to 48.6%, 50.7% to 55.3% and 85.6% to 87.2% compared to HiCHT-base, respectively. The introduction of *fingerprinting* and SIMD instructions avoids the comparison of unmatched elements, thus reducing the number of instructions and cycles required per lookup. Meanwhile, it requires less data access and has better data locality, especially for 64-bit keys.

4.3 Comparing with Compact Hash Tables

In this subsection, we measure the throughput of different hash table methods for insert, lookup, and delete operations as well as their memory footprint. For the evaluation of CHTs, we add an 8-bit value to each key in the dataset, which is the same configuration as in [12]. We omit the comparison of the delete operation for clearCHT and dCHT as these implementations do not support deletion.

Experiment 3 (Evaluation of Different Compact Hash Tables): Figure 5 reports the experimental results of lookup, insert, and delete performance of the existing CHTs on all datasets, respectively. It can be seen that HiCHT has a major advantage in the performance of basic operations. For insert operations, the throughput of HiCHT is 1.82 to 2.35 times better than that of

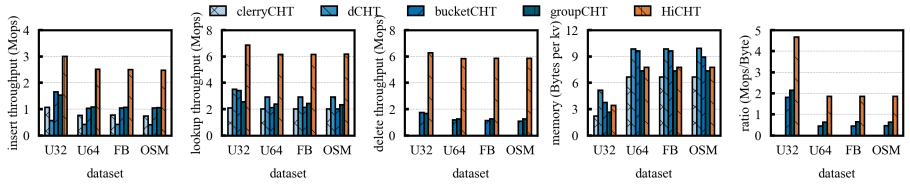


Fig. 5. Comparison of performance and memory consumption with CHTs.

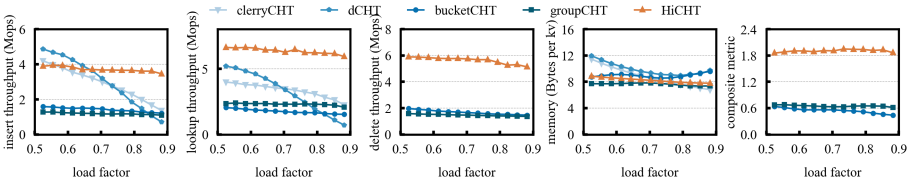


Fig. 6. Performance and memory consumption of CHTs with different load factors.

the best method. For lookup operations, the throughput of HiCHT improves by 1.96 to 2.11 times compared with that of the best method. For delete operations, HiCHT has a 3.59 to 4.66 times improvement in throughput over the best method. HiCHT is the fastest CHT for all basic operations, and the main credit for the property comes from the *fingerprinting* and SIMD techniques we introduced. Meanwhile, the existing CHTs require more overhead to handle hash collisions or compare the quotients, resulting in poorer performance.

Figure 5 also shows the memory consumption and P/M ratio of all tested CHTs. We can observe that clearyCHT is the most space-efficient CHT variant, followed by groupCHT and HiCHT. In addition, the performance/memory ratio of HiCHT is 2.18 to 2.95 times better than the best-performing competitor.

Experiment 4 (Impact of Different Load Factors): The load factor is also one of the key factors affecting the performance of the hash table. In this experiment, we set different load factors ranging from 0.5 to 0.9 to evaluate the performance of different CHT methods using the U64 dataset.

We first populate the hash table with a number of keys about half the capacity of the hash table and then continue to insert, lookup, and delete keys in batches until the maximum load factor is reached. As shown in Figure 6, the insertion and lookup performance of clearyCHT and dCHT decreases significantly as the load factor increases, while that of HiCHT remains almost unchanged. The reason for the poor performance of clearyCHT and dCHT is that these operations require more probes when there is less space available in these hash tables. Figure 6 also shows the memory usage and the performance/memory ratio of different CHTs. The memory consumption of HiCHT varies slightly, and it achieves better memory efficiency than clearyCHT when the load factor is low. Meanwhile, HiCHT's performance/memory ratio remains the best among its competitors. To summarize, the space and performance of CHTs based on open

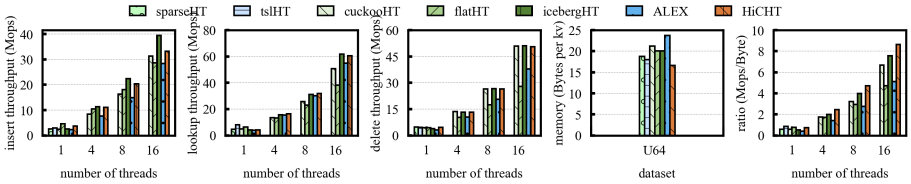


Fig. 7. Comparison with non-compact hash tables using multiple reader and writer threads.

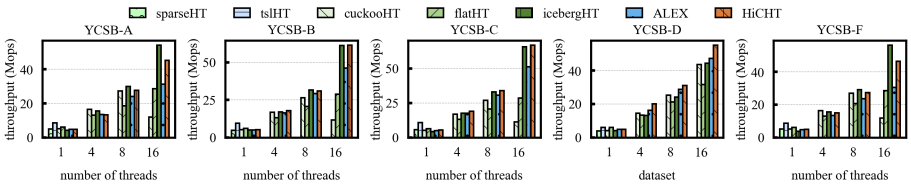


Fig. 8. Comparison with non-compact hash tables using YCSB workloads.

addressing depend heavily on the setting of the load factor, while our approach performs well at both low and high load factors.

4.4 Comparing with Non-compact Hash Tables

In this subsection, we repeat the experiments using different reader and writer threads on the U64 dataset and compare HiCHT with fine-grained optimistic locking with other hash tables including Google's sparse hash table, Tessil's sparse hash table, Google's flat hash table, Cuckoo hash table [14], and icebergHT [19]. We also compare HiCHT with the learned index ALEX [9, 22], which uses multi-level linear models to replace hash functions. To minimize the impact of memory alignment in these indexes, a 64-bit value is appended to each key. We also use YCSB workloads [8] to compare HiCHT with other methods in scenarios involving access skewness and mix operations.

Experiment 5 (Evaluation of Basic Operations for Different Non-compact Hash Tables): As shown in Figure 7. HiCHT has better lookup performance compared to fine-grained locking designs (cuckooHT) due to the fact that it does not need to acquire any lock when performing a lookup operation. The performance of the insertion/deletion operation in HiCHT is inferior to that of the best competitor since HiCHT needs to reorganize the hash bucket.

We can also find that HiCHT achieves the best memory efficiency compared to other hash tables, which saves at most 29.7% space budget, and the P/M ratio of HiCHT also outperforms other methods on the U64 dataset. HiCHT achieves its space advantage primarily by using a compact storage format for key-value pairs, whereas other non-compact methods must store the entire key-value pair.

Experiment 6 (Evaluation of Different Non-compact Hash Tables Using YCSB workloads): The YCSB workloads contain six workloads with

various combinations of operations including reads, updates, inserts, scans, and read-modify-writes. Specifically, the six workloads are A (50% reads and 50% updates), B (95% reads and 5% updates), C (100% reads), D (95% reads and 5% inserts), E (95% scans and 5% inserts), F (50% reads and 50% read-modify-writes). All workloads expect D to follow the Zipfian distribution (the alpha parameter defaults to 100). Figure 8 shows the performance of various methods under different YCSB workloads except workload E since hash tables do not support range queries. It can be observed that the performance of HiCHT under highly skewed access patterns are still competitive with icebergHT.

We draw the conclusion that HiCHT provides a space-efficient hash table alternative, and it still has excellent performance for all basic operations compared to existing widely used hash tables. At the same time, it scales well in multi-core environments via fine-grained optimistic locking.

5 Conclusion

We present HiCHT, a high-performance CHT using *quotienting* and *fingerprinting* as its core building blocks. HiCHT stores the conflicting keys in the chaining buckets. Each bucket can hold more keys to reduce the space overhead of pointers, and it chooses the best-fit size based on the number of keys stored in the bucket. The key enabling component for high performance is *fingerprinting*, which takes full advantage of caching and greatly reduces the amount of data accessed. Moreover, we compare the fingerprints in parallel through SIMD instructions to obtain further performance promotion.

Overall, HiCHT can approach the theoretical information space lower bound while still maintaining high performance for basic operations. The experiments reveal that HiCHT outperforms existing CHTs in terms of performance by a factor of 1.82 to 2.35 times, and it also offers a competitive memory efficiency compared to the best CHT.

Acknowledgments. This work was supported by Anhui Province Development and Reform Commission 2021 New Energy and Intelligent Connected Vehicle Innovation Project, the National Nature Science Foundation of China (61672480), the 111 Project 2.0 (BP0719016).

References

1. Agarwal, R., Khandelwal, A., Stoica, I.: Succinct: Enabling queries on compressed data. In: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15). pp. 337–350 (2015)
2. Amble, O., Knuth, D.E.: Ordered hash tables. The Computer Journal **17**(2), 135–142 (1974)
3. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: 2010 IEEE 51st Annual Symposium on Foundations of Computer Science. pp. 787–796. IEEE (2010)

4. Bercea, I.O., Even, G.: A space-efficient dynamic dictionary for multisets with constant time operations. arXiv preprint arXiv:2005.02143 (2020)
5. Blandford, D.K., Blelloch, G.E.: Compact dictionaries for variable-length keys and data with applications. ACM Transactions on Algorithms (TALG) **4**(2), 1–25 (2008), publisher: ACM New York, NY, USA
6. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de bruijn graphs. In: International workshop on algorithms in bioinformatics. pp. 225–235. Springer (2012)
7. Clerry, J.: Compact hash tables using bidirectional linear probing. IEEE Transactions on Computers **100**(9), 828–834 (1984)
8. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
9. Ding, J., Minhas, U.F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., et al.: Alex: an updatable adaptive learned index. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 969–984 (2020)
10. Kak, S.: Generalized unary coding. Circuits, Systems, and Signal Processing **35**(4), 1419–1426 (2016)
11. Knuth, D.E., et al.: The art of computer programming, vol. 3. Addison-Wesley Reading, MA (1973)
12. Köppl, D., Puglisi, S.J., Raman, R.: Fast and simple compact hashing via bucketing. Algorithmica **84**(9), 2735–2766 (2022)
13. Larson, P.A.: Dynamic hash tables. Communications of the ACM **31**(4), 446–457 (1988)
14. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 1–14 (2014)
15. Liu, M., Yin, Y., Yu, H.: Succinct filters for sets of unknown sizes. arXiv preprint arXiv:2004.12465 (2020)
16. Lu, B., Hao, X., Wang, T., Lo, E.: Dash: Scalable hashing on persistent memory. arXiv preprint arXiv:2003.07302 (2020)
17. Navarro, G.: Compact data structures: A practical approach. Cambridge University Press (2016)
18. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In: Proceedings of the 2016 International Conference on Management of Data. pp. 371–386 (2016)
19. Pandey, P., Bender, M.A., Conway, A., Farach-Colton, M., Kuszmaul, W., Tagliavini, G., Johnson, R.: Iceberght: High performance hash tables through stability and low associativity. Proceedings of the ACM on Management of Data **1**(1), 1–26 (2023)
20. Poyias, A., Puglisi, S.J., Raman, R.: m-bonsai: A practical compact dynamic trie. International Journal of Foundations of Computer Science **29**(08), 1257–1278 (2018)
21. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: International Colloquium on Automata, Languages, and Programming. pp. 357–368. Springer (2003)
22. Wongkham, C., Lu, B., Liu, C., Zhong, Z., Lo, E., Wang, T.: Are updatable learned indexes ready? arXiv preprint arXiv:2207.02900 (2022)