Monkey Hashing: a Wait-Free Hashing Scheme with Worst-Case Constant-Time Operations

Judismar Arpini Junior*, Vinícius Gusmão Pereira de Sá†

March 3, 2023

Abstract

We exploit multiple-choice hashing to create a concurrent hashing scheme with $O(1)$ worst-case time for lookup, insert, update and remove operations, an eventually consistent hash map that does not use any kind of synchronization for thread-safety. It works particularly well in scenarios with a single writer and multiple reader threads, significantly outperforming popular solutions such as ConcurrentHashMap (Java) and Intel TBB concurrent_hash_map (C++) in heavily concurrent stress test scenarios. The price to pay is a non-zero probability that an insertion might fail, which can be made small enough, though, to suit all imaginable applications.

1 Introduction

Hashing-based data structures have enormous importance in computing, starring in a number of efficient algorithms for a variety of problems [3, 4, 7, 12, 19]. The basic intent consists of storing *keys* (unique identifiers), sometimes mapped to *values*, to be retrieved at some later time. Dictionary operations are made efficient by determining the position (*bucket*) in an underlying array (*table*) where the key (and its associated value object, if any) will be stored as a function of the key itself.

Not too long ago, advances in computer hardware usually meant advances in clock speed, so all existing software would speed up by itself over time. In recent times, manufacturers have been leaning towards architectures with multiple communicating processors, increasing the ability to execute parallel tasks first and foremost, with only modest increases in clock speed. Those systems are called *shared-memory multiprocessors*, or *multicores* [9], where multiple concurrent threads must coordinate their operations on some shared memory.

The traditional way to implement thread-safe data structures (structures that may be accessed by concurrent threads without entering an inconsistent

*Colégio Pedro II
†Universidade Federal do Rio de Janeiro

state) is via mutual exclusion, where only one thread at a time is allowed to operate on the structure. However, such blocking implementations will not always suit certain systems [8] where a lock-free or wait-free behavior may be of utmost importance [9].

Formally, a shared data structure is *lock-free* if it guarantees that infinitely often some thread completes its task within a finite number of steps. A shared data structure is *wait-free* if it satisfies a stronger condition, namely that *each thread* completes its execution in a finite number of steps.

An efficient thread-safe map implementation is Java's lock-based ConcurrentHashMap [14]. Instead of locking the entire map, it uses one lock per bucket or subset of buckets. It is designed to optimize lookups. Writer threads may update the hash map concurrently, and lookups may be executed without using locks.

The Intel Threading Building Blocks library (TBB) [11] provides the *concurrent_hash_map* class, quite popular in the C++ community. It resolves hash collisions by allocating linked lists for each buckets, and employs a lock per bucket to maximize concurrent execution.

The hash table in [16] is a high-performance lock-free scheme that resolves collisions via linked lists. It can perform resizes to grow or shrink the table size when needed. However, after a first resize is performed, the average time complexity of the operations will no longer be a constant function of the table size [16, 18].

The work in [18] presents another high-performance lock-free hash table that enforces the average constant time complexity of operations after resizes.

A lock-based concurrent cuckoo hashing is found in [9]. It improves concurrency by trading space; thus, its space overhead is higher than the basic cuckoo hashing [5].

The optimistic cuckoo hashing [5] has great memory efficiency. It was designed for scenarios where lookups dominate, with only one writer allowed at a time. It was later improved by [15] to support multiple writers, using the concurrent writer approach from the concurrent cuckoo hashing introduced in [9].

The hopscotch hashing [10] is designed for both sequential and concurrent use, with $O(1)$ worst-case lookup time. It combines the advantages of cuckoo hashing, chained hashing and linear probing. Like cuckoo hashing, an insertion may fail and a full rehash is required; thus, the worst-case insertion time is $\Omega(n)$.

In [13], Laborde, Feldman and Dechev present an ingenious wait-free hash map with constant worst-case time for all basic operations. It consists of a multi-level array, using multiple tables structured as a tree. It avoids global resizing by allocating new arrays when necessary. A drawback is that there is no freedom to choose hash functions, much like in a direct-addressing solution. Indeed, the hash value is, so to say, *the key itself*, seen as a bit sequence that uniquely determines a path from the tree root down to a leaf node where the key will be effectively stored, in a way that is very similar to how a *trie* operates [6]. Because there is no arbitrarily chosen hash function, its deterministic behavior is always known beforehand, and a malicious adversary may choose keys that will lead its operations to their very worst case (yielding bigger multiplicative factors,

but still within constant time). The number of collisions may also happen to be (intentionally) maximized so that $\Theta(n)$ arrays are allocated, yielding a higher memory overhead. Finally, the worst-case performance of all basic operations is affected when larger keys are used, since the maximum depth of the tree increases with the size of the keys (again, the worst-case time is still constant[1], albeit with possibly somewhat bigger multiplicative factors).

Our contribution. We propose the wait-free *monkey hashing* scheme for scenarios with a single writer and multiple reader threads, where traditional, non-thread-safe schemes invariably incur in concurrency issues. Monkey hashing is a simpler alternative to the more involved, thread-safe, wait-free scheme in [13]. The monkey hashing is an actual hashing scheme, in the sense that one can pick whatever hash function (or rather hash *functions*, plural) they please. It does not use locks or any other kind of synchronization, and every operation runs in $O(1)$ worst-case time.[2] The price to pay is a (small) probability that an insertion might fail. As we shall see, we can make the failure probability small enough even to the most demanding applications.

In Section 2, we introduce the novel monkey hashing scheme. In Section 3, we discuss the eventual consistency of our construct. In Section 4, we obtain an upper bound for the failure probability of the insertion and argue that it can be made negligible without touching the asymptotic efficiency of either time or space. In Section 5, we present experimental results that compare the proposed scheme with Java's ConcurrentHashMap [14], with the TBB concurrent_hash_map [11] and with Laborde et al.'s wait-free hash map [13].

2 Monkey Hashing

Monkey hashing (MH) is a thread-safe hashing scheme that does not use locks or any other kind of synchronization, requiring only that single-word read and write operations are atomic. It consists of a single hash table and $k \geq 1$ hash functions, meaning multiple alternative locations for each key in the same table. Unlike cuckoo hashing, elements will never be evicted from where they first landed, so new keys being inserted must always find an unoccupied spot to call their own.[3]

Like the concurrent cuckoo hashing in [5], our hashing scheme is specially tailored for scenarios with a single writer and multiple reader threads, though it certainly works just fine even in single-thread applications.

[1] More precisely, it is proportional to the key size, much like the computation of typical hash functions. It can be considered $O(1)$, though, assuming the maximum key length is a (typically small) constant.

[2] Under the same fixed maximum key length assumption (see previous note).

[3] The name stems from the analogy to a monkey that jumps along branches of a tree until it finds a vacant one to rest. Some cultures even have popular sayings in the likes of "every monkey to their own branch", meaning "every jack to his trade".

The monkey hashing can be implemented as a *hash map*, that is, it can store key-value pairs, even though a pair consists of two memory words and cannot be read atomically. Of course it can also be implemented as a *hash set*, where only keys are stored.

It is required that one knows beforehand the maximum number of entries that may ever coexist in the map (i.e., its maximum capacity), since memory is pre-allocated to enforce the intended load factor without the need of rehashes.[4]

The k hash functions may be obtained from a universal class [2]. Such functions work just as well as idealized uniform hash functions do on average [17].

To look up some key x, we follow the sequence of positions $h_1(x), h_2(x), \ldots,$ $h_{k'}$ in the underlying array T (i.e., the table) until we have either found x or reached the maximum number of hash functions that were actually called for by any insert operation, denoted by k'. Such a *maximum in use* value k' is dynamically kept track of. The pseudo-code of the lookup algorithm is shown in Figure 1.

```
Lookup(x)
1.  for i = 1, 2, ..., k'
2.      if T[h_i(x)] = x
3.          return true
4.  return false
```

Figure 1: The monkey hashing lookup algorithm

When a key x is inserted, we insert x in the first position in the hash table that is not occupied along that same sequence $h_1(x), h_2(x), \ldots,$ up to $h_k(x)$, where k is the aforementioned number of available hash functions, i.e., the *maximum allowed* number of insertion attempts for a given key. The insertion fails if all k positions are occupied. The pseudo-code in Figure 2 shows the insertion algorithm, where an empty position is denoted by $\perp$. Note that we use a mapping A of all positive integers $i \in [1, k']$ onto the number of keys that required i hash functions during its insertion. Such a mapping will also be updated by the remove operation, and is used ultimately to keep track of $k' \leq k$.

To remove a key, we first look it up in the MH structure. If it is found, then, along with removing it from the table, we also make sure to remove its contribution to the mapping A of hash function counts. Figure 3 gives the pseudo-code for the remove operation.

Since there are no keys ever being moved and no collision lists to maintain, the threads accessing a MH structure are free to execute all basic operations without any kind of synchronization. Hence, for a constant number k of hash

[4] In the experiments of Section 5, Java's ConcurrentHashMap and Intel's TBB concurrent_hash_map are also pre-allocated to avoid resizes (something they do support), thus ensuring a fair comparison.

```
Insert(x)
1.  if Lookup(x)
2.      return
3.  for i = 1, 2, ..., k
4.      if T[h_i(x)] = ⊥
5.          T[h_i(x)] ← x
6.          A[i] ← A[i] + 1  // one more key requiring i functions
7.          k' ← max{k', i}  // k' may have changed
8.          return
9.  fail
```

Figure 2: The monkey hashing insertion algorithm

```
Remove(x)
1.      for i = 1, 2, ..., k'
2.          p ← h_i(x)
3.          if T[p] = x
4.              T[p] ← ⊥
5.              A[i] ← A[i] − 1  // one less key requiring i functions
6.              if A[i] = 0 and k' = i  // must update k'
7.                  for j = i − 1, i − 2, ..., 1
8.                      if A[j] > 0
9.                          k' ← j
10.                         return
11.                 k' ← 0  // empty table
12.     fail  // not found
```

Figure 3: The monkey hashing removal algorithm

functions, the operations are all wait-free, bounded by $O(k) = O(1)$ steps.

We can iterate through the entries in thread-safe fashion by traversing the underlying array while skipping empty positions. Insertions and deletions done while an iteration is taking place might not be immediately seen, as discussed in Section 3, something usually referred to as *eventual consistency*, which is inherent to—and all the most tolerable by—most multi-threaded applications.

3 Thread-Safety and Eventual Consistency

The writer thread and the reader threads execute their operations concurrently, ignoring synchronization issues.

An entry with key x that has just been inserted may not be seen immediately

thereafter by all reading threads. A reading thread might have started to look it up before it was inserted, and may have just checked position $h_j(x)$. Then, the writer thread inserts the entry in that same position and the reader thread resumes its operation from position $h_{j+1}(x)$ and fails to find the key. After it is first seen, though, never (again) should it fail to be successfully retrieved by each reading thread. Such a behavior—when no more updates are done, all readers will eventually read the latest written data—is commonly referred to as *eventual consistency*, and is inherent to many modern solutions, from distributed databases to event stores to the Internet Domain Name System (DNS) [1].

Deleting a key does not cause any trouble to the lookup operations in the reader threads: it takes a single atomic step to erase the key, be it of a primitive data type or an object reference (setting it to a *null* value). In the latter case, if the reference was already retrieved, and then removed shortly thereafter, that object will still be accessible to the reader, even if the writer replaces that reference with a null or some other reference.

When using MH to implement maps, the value, in each key-value pair, should be written first (and deleted last) to avoid the retrieval of a key without the intended corresponding value. Moreover, when storing key-value pairs, a mapped value that is changed may not be immediately seen by all reading threads. Indeed, a reader thread might have just retrieved the reference of a value, when the writer thread replaces it with a reference to another value. Hence, the reader thread will actually have accessed the *previous* value. This is an expected, harmless race condition, and the outcome would have been the same had the update operation started after the completion of the read operation. Note, though, that reading partially updated entries must be avoided by enforcing that the data types of both key and value have atomic writes (which is the case, for example, in modern Java, for pointers and wrappers of primitive types).

Note that key-value pairs will probably be implemented as objects allocated in the random-access memory. Each position in the underlying array (i.e., the table) stores a reference to one of those objects (see Figures 4(a) and 4(b)). To improve performance, we may want to reuse the referenced object when removing a key-value pair. This avoids the overhead of dynamically freeing and (re-)allocating memory for new entries, as well as reducing the garbage collection burden, when applicable. We can achieve that by logically erasing both the key and the value of the existing entry without actually freeing the key-value object. However, after a reader thread retrieves an entry, the writer thread might overwrite the fields in that reused key-value object pair (after, say, a logical deletion followed by an insertion), and thus the reader thread might end up reading the value associated with the new key. When the key can be inferred from the value object (e.g., the key is some *user id* and the value is some *User* object, containing, among other data attributes, the user id itself), we can avoid this concurrency issue altogether by checking that the retrieved value does indeed correspond to the informed key. This can be done by providing MH with a function that, given a value object, returns its (unique) key. If the key obtained from the value does not correspond to the search key, then a deletion surely took place in the meantime and a null would be returned instead; otherwise return

the retrieved, verified value object. In case such a verification function is not provided, the MH scheme will still work, albeit *not* reusing key-value objects, therefore actually freeing such objects during deletions, and reallocating a new key-value object from scratch at every insertion.

In case only keys are stored in the scheme (i.e., when implementing sets, not maps), keys (or, alternatively, references to key objects) may be written directly to the underlying array (see Figures 4(c) and 4(d)).

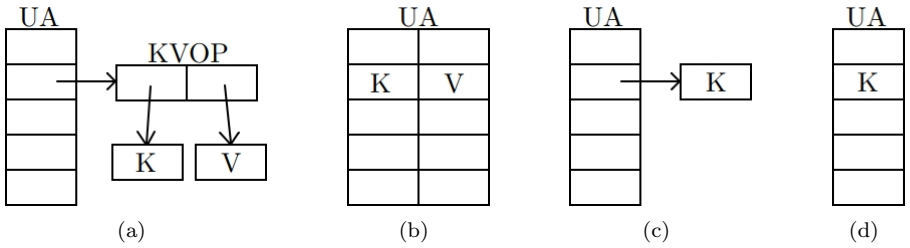

Figure 4: Some possible MH setups: (a) hash map for objects; (b) hash map for primitive types; (c) hash set for objects; (d) hash set for primitive types. K, V, UA and KVOP indicate, respectively, the stored key, the mapped value, the underlying array (table), and the key-value object pair.

4 Insertion Failure Probability

The insertion method is strictly speaking a randomized algorithm with a probability of failure. We can make such a probability be so small that the MH would work just as reliably as any deterministic algorithm would, in practice.

The upper bound on the probability that an insertion might fail decreases exponentially with the predefined maximum number of hash functions. Say we set 50 as the maximum number of hash functions, as we did in the tests depicted in Section 5, and $m = 2n$ as the size of the hash table, where n is the maximum number of entries stored at any given time. Because the load factor never exceeds $1/2$, when inserting key x the probability that position $h_i(x)$ is occupied is at most $1/2$ for all $i \in \{1, ..., 50\}$. Thus, the probability that the insertion of x fails is at most 2^{-50}. The number of insertions until the first failure occurs is a geometric random variable with expectation at least 2^{50}, which is more than one quadrillion. If we perform one million insertions (and one million deletions, keeping the table at its maximum capacity) every second, non-stop, this means we would expect to see our first failure in 35.7 years.[5] If we increase the number of functions very slightly, from 50 to just 52, the expected time before the first failure increases to nearly one and a half century.

[5]To keep things in perspective, Twitter usually registers an average of 6000 new tweets per second, worldwide, at the time of writing this paper, with a registered peak of 143,199 tweets per second in August, 2020.

5 Experiments

We have conducted experiments to compare the proposed MH scheme against existing, well-known solutions. All experiments were conducted on a Linux Mint 20.3 64-bit machine, with a multicore Intel(R) Core(TM) i3-2100 CPU with 4 cores of 3.10 GHz, and with 3.7 GB of RAM memory.

First, we compared a MH implementation in Java against Java's inbuilt thread-safe hash map solution, the ConcurrentHashMap (CHM), and against Java's ConcurrentSkipListMap (CSLM), a non-hash-based map whose thread-safety is known to be very well implemented, to the extent that it outperforms both the CHM in scenarios with a big number of threads and Java's TreeMap (a non-thread-safe red-black tree) in single-threaded scenarios. Monkey hashing significantly outperformed CHM in all multi-thread test scenarios. It also outperformed CSLM when the number of threads was the same or smaller than the number of CPU cores.[6] Second, we compared a C++ implementation of MH against Intel's TBB concurrent_hash_map.[7] Again, MH performed significantly better.

Figures 5 and 6 show the average running time of a sequence of read and write operations as a function of the number of concurrent threads.

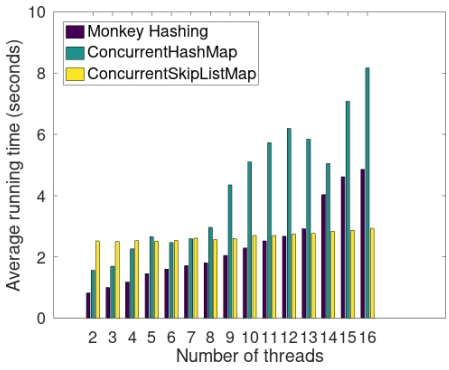

Figure 5: 25,000 inserted items.

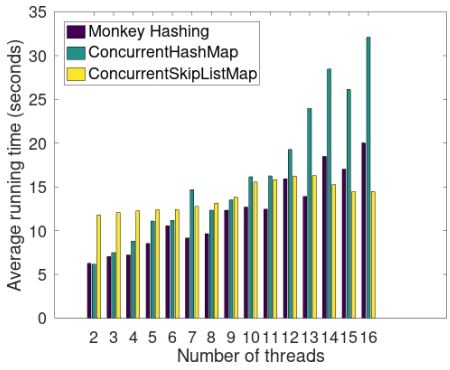

Figure 6: 100,000 inserted items.

Our scenario consists of a single writer thread and multiple reader threads. The number of items inserted in the map is 25,000 in the experiment of Figure 5, and 100,000 in the experiment of Figure 6. The target load factor was 1/2 in all experiments. The maximum capacity and the load factor are provided as constructor arguments to pre-allocate the table size, thus avoiding resizes. This was done to MH and CHM schemes all the same. The number of concurrent threads varied from 2 to 16 (always one writer, several readers).

When we increased the number of threads, we also increased the total amount of work, because each reading thread iterates through the entries 2,000 times. The writer thread inserts 25,000 or 100,000 items and overwrites the

[6]Source code: https://github.com/judismar/MonkeyHashMapJavaExperiments
[7]Source code: https://github.com/judismar/MonkeyCppExperimentsAgainstTBB

values of all these items 1,000 times. This sequence of operations was repeated 4 times to obtain the average total running time.

In Figures 7 and 8, we compare the monkey hashing against the TBB concurrent_hash_map[8] in C++. They show the average running time of a sequence of read and write operations as a function of the number of concurrent threads. The monkey hashing significantly outperforms the TBB concurrent_hash_map.

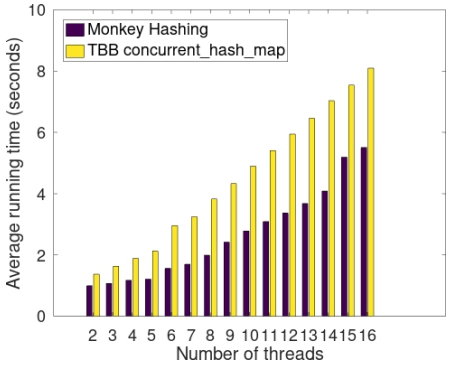

Figure 7: 25,000 inserted items.

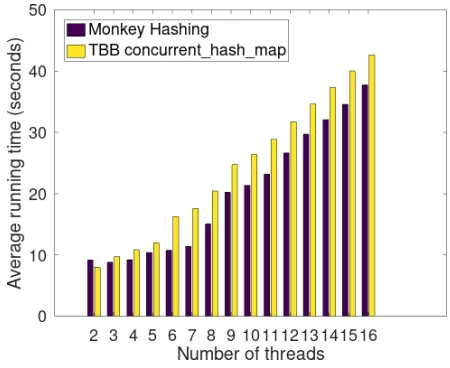

Figure 8: 100,000 inserted items.

In this scenario of a single writer thread and multiple reader threads, the number of items inserted by the writer thread is 25,000 in the experiment of Figure 7 and 100,000 in the experiment of Figure 8. The intended load factor is 1/2. The maximum capacity and the load factor are provided as constructor arguments of the monkey hashing to pre-allocate the table size. Twice the maximum capacity is provided as the constructor argument of the TBB concurrent_hash_map to pre-allocate the table size, thus avoiding resizes. The number of threads running concurrently is varied from 2 to 16.

When we increased the number of threads, the total amount of work also increased, because each reading thread iterates through the entries a fixed 1,000 times. The writer thread inserts 25,000, or 100,000 items. This sequence of operations is repeated 5 times to obtain the average total running time.

Figures 9 and 10 show the average running time of a sequence of read and write operations as a function of the number of concurrent threads in order to compare the monkey hashing with the wait-free hash map in [13][9], written in C++. MH ties with Laborde et al.'s hash map.

The scenario of the simulation in Figures 9 and 10 consists of a single writer thread and multiple reader threads. In the experiment of Figure 9, the writer thread inserts 25,000 items, and each reader thread performs 25,000 get operations. In the experiment of Figure 10, the writer thread inserts 100,000 items,

[8]Source code: https://github.com/judismar/MonkeyCppExperimentsAgainstTBB

[9]Source code: https://github.com/judismar/MonkeyHashMapCppExperiments. The code of the wait-free hash map in [13] is from https://github.com/AgamAgarwal/wait-free-extensible-hash-map.

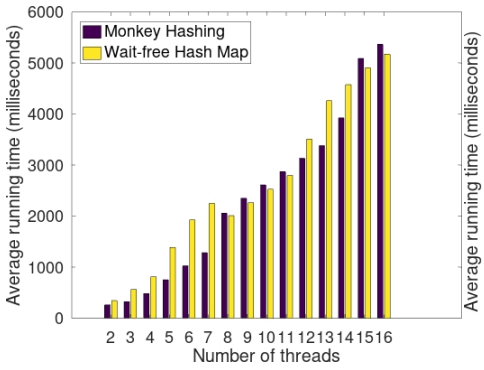

Figure 9: 25,000 inserted items.

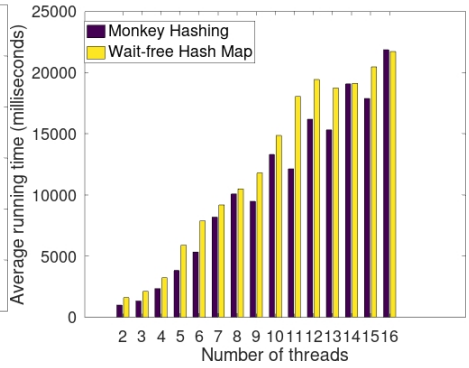

Figure 10: 100,000 inserted items.

and each reader thread performs 100,000 get operations. The intended load factor for MH was 1/2, and the maximum capacity and the load factor were again provided as constructor arguments to pre-allocate the tables. The number of threads running concurrently varied from 2 to 16. The writer thread inserts 25,000 or 100,000 items. This sequence of operations was repeated 5 times to obtain the average total running time.

6 Conclusion

We proposed a novel thread-safe hashing scheme that is simple to understand and to implement. High concurrent performance is attained by dispensing with locks or other forms of thread synchronization, for the price of a strictly positive probability of failure during an insertion operation, a probability which can be made negligible for all intents and purposes. In the quite common scenario of a single writer and multiple readers, for which our contribution is particularly well-suited, our scheme consistently outperforms well-known, widely used solutions such as Java's ConcurrentHashMap and C++ Intel TBB concurrent_hash_map, while not presenting the drawbacks of previous works with roughly the same objectives.

We believe it should be possible to devise a variant of the monkey hash set — one which uses CAS during insertions and removals — that allows multiple writers while still purposefully ignoring mutual exclusion.

We thank Paulo Casaes and Juan Lopes for the insightful discussions.

References

[1] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually consis-

tent infrastructure given no guarantee of safety? *ACM Queue*, 11(3):20–32, 2013.

[2] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[3] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (CSUR)*, 50(1):1–36, 2017.

[4] Lucila Maria de Souza Bento, Vinícius Gusmão Pereira de Sá, and Jayme Luiz Szwarcfiter. Some illustrative examples on the use of hash tables. *Pesquisa Operacional, 35 2*, 2015.

[5] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 371–384, 2013.

[6] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, sep 1960.

[7] GeeksforGeeks. Applications of hashing. https://www.geeksforgeeks.org/applications-of-hashing/. Accessed: 2022-04-22.

[8] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[9] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[10] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008.

[11] Intel. Intel threading building blocks. https://github.com/oneapi-src/oneTBB. Accessed: 2022-04-22.

[12] Donald E. Knuth. *"The Art of Computer Programming, Vol. 3: Sorting and Searching"*. Addison-Wesley Publishing Co., Reading, Massachusetts, USA, second edition, 1998.

[13] Pierre Laborde, Steven Feldman, and Damian Dechev. A wait-free hash map. *International Journal of Parallel Programming*, 45(3):421–448, 2017.

[14] Doug Lea. Hash table util.concurrent.concurrenthashmap. *JSR-166, the proposed Java Concurrency Package*, 2003.

[15] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[16] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.

[17] Michael Mitzenmacher and Salil P Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *SODA*, volume 8, pages 746–755. Citeseer, 2008.

[18] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

[19] Martin Thoma. The 3 applications of hash functions. https://levelup.gitconnected.com/the-3-applications-of-hash-functions-fab1a75f4d3d. Accessed: 2022-04-22.