A Fast Concurrent and Resizable Robin Hood Hash Table

by

Endrias Kahssay

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

Author ....
Department of Electrical Engineering and Computer Science
Jan 29, 2021

Certified by..
Julian Shun
Assistant Professor
Thesis Supervisor

Accepted by ....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Fast Concurrent and Resizable Robin Hood Hash Table

by

Endrias Kahssay

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 29, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Concurrent hash tables are among the most important primitives in concurrent programming and have been extensively studied in the literature. Robin Hood hashing is a variant of linear probing that moves around keys to reduce probe distances. It has been used to develop state of the art serial hash tables. However, there is only one existing previous work on a concurrent Robin Hood table. The difficulty in making Robin Hood concurrent lies in the potential for large memory reorganization by the different table operations. This thesis presents Bolt, a concurrent resizable Robin Hood hash table engineered for high performance. Bolt treads an intricate balance between an atomic fast path and a locking slow path to facilitate concurrency. It uses a novel scheme to interleave the two without compromising correctness in the concurrent setting. It maintains the low expected probe count and good cache locality of Robin Hood hashing. We compared Bolt to a wide range of existing concurrent hash tables in a comprehensive benchmark. We demonstrate that on a 36-core machine with hyper-threading, Bolt is between 1.57x - 1.73x faster than the fastest publicly available non-resizable concurrent hash table and 1.45 - 2x faster than the fastest publicly available concurrent resizable hash table. It also achieves speedups between 15x - 35x over a highly optimized serial implementation at 36-cores with hyper-threading.

Thesis Supervisor: Julian Shun
Title: Assistant Professor

Acknowledgments

This thesis is based on work done in collaboration with Rahul Yesantharao and Sameer Pusapaty who I had a pleasure working with. I would like to extend my sincere gratitude to Prof. Julian Shun for his guidance through two different research projects. Julian is very knowledgeable, and was patient in guiding me to become a better researcher.

I would like to thank Professor Nir Shavit who inspired my passion in Systems my freshman year through his passionate teaching of 6.816: Multicore Programming and for giving me the opportunity to do my first research project. I would like to also thank Prof. Charles Leiserson, TB Shardl, and Prof. Julian Shun for teaching me how to write fast software in 6.172: Performance Engineering and later giving me the opportunity to teach it.

Finally, I am grateful for the close friends I made at MIT, and my mom and dad and siblings who are always at my side.

Contents

List of Figures

Chapter 1

Introduction

Concurrent data structures are data structures that allow multiple threads to operate on them with proper correctness guarantees. Making them performant has become increasingly important in recent years due to the end of CPU frequency scaling and the shift towards multiprocessor machines. Concurrent data structures are inherently more challenging to implement properly than their sequential counterparts because of the complexity of the interactions between different threads and the underlying memory model.

A hash table is a fundamental, flexible, and dynamic data structure that stores a mapping from keys to data. The operations commonly provided are **insert**, **find**, and **delete**. It is an important data structure for sharing data between threads in the concurrent setting. A hash table uses a hash function to compute an index for a key and uses it to look up the associated data. A hash table has to handle collisions that occur when two keys map to the same index. The two common methods for resolving collisions are hash chaining and open addressing.

Hash chaining constructs an associated list per index for elements that collide in the table. To look for a colliding key's corresponding data, the key is hashed to its index, and then the corresponding list is searched. Open addressing, on the other hand, places colliding keys in nearby slots directly within the table. There are many variations of open addressing that differ in how they find slots for colliding keys. The most popular approach is linear probing, which linearly searches for an empty slot starting from the hash index.

Robin Hood Hashing [3] is a variant of open addressing that seeks to minimize variance of distances of keys to their hashed index, which is referred to as the **probe distance**. It does so by moving around keys in the table. Robin Hood Hashing has strong guarantees on the distribution of the probe distance across the table, which allows unsuccessful find operations to stop early without encountering an empty slot, unlike in linear probing.

Low variance of the probe distance is well suited for branch prediction and cache utilization. Because Robin Hood's operations proceed linearly in memory and the probe distance is low, serial Robin Hood should perform well on modern CPU architecture. It also has a natural solution for deletions that allow deleted slots to be reclaimed. Indeed, Ska [12] shows that Robin Hood can be used to develop a state-of–of-the–the-art hash table in the serial context.

However, Robin Hood hash table is difficult to make concurrent in a performant way because each operation may move around large amounts of memory. This is problematic because modern CPU architectures only support moving 16 bytes of memory atomically, which is only enough to fit one key and its associated value. In addition, Robin Hood imposes strong requirements on how keys are laid out in the table, which is challenging to maintain in the concurrent context.

Kelly et al. present the only existing concurrent implementation of Robin Hood [5]. However, it has poor performance, doesn't support associating values with keys, and lacks support for **resizing**, which is necessary when the table fills up.

In this thesis, we present Bolt, a re-sizable concurrent version of a Robin Hood hash table. Bolt is engineered for high performance. Bolt treads an intricate balance between an atomic "fast path" and a locking "slow path" to facilitate concurrency. It uses a novel scheme to interleave the two without compromising correctness in the concurrent setting.

Bolt's key insight is to distinguish between operations that can be resolved by looking at the location where the key hashes to, the **distance zero slot**, and operations that can't. In the former case, there is a fast path that bypasses locks and uses atomic instructions to execute the operation. This is the common case for reasonable load factors (≤ 0.6).

In Bolt, the table is divided into segments, each of which is protected by a lock. Operations that can be resolved by looking at the distance zero slot bypass the locks and directly operate on memory. Operations that require multiple memory look-ups take locks to properly interact with other operations operating over the same memory region. The complexity arises from the interaction between threads executing the locked path and the non locked path. Most of our find operations resolve using one atomic load, which makes them very fast. We also support updating the values of existing keys, which is done without locks in the fast path.

Bolt has very low overhead to support resizing because of a novel heuristic we developed that approximates how full the table is. This is based on counting the number of keys that are misplaced from their distance zero slot concurrently.

We conduct extensive experiments across the different operations of the table on both uniform and contented use cases. We demonstrate that on a 36-core machine with hyper-threading, Bolt is between 1.57x - 1.73x faster than the fastest publically available non-resizable concurrent hash table (folklore[8]) and 1.45 - 2x faster than the fastest publically available concurrent resizable hash table (growt[8]). It also achieves speedups between 15x - 35x over a highly optimized serial implementation at 36-cores with hyper-threading.

Chapter 2 gives background on serial Robin Hood and related work. Chapter 3 establishes the different structures utilized in Bolt and the API. Chapter 4 discusses the non-resizable version of Bolt. Chapter 5 argues the correctness of Bolt. Chapter 6 describes the modifications necessary to support resizing. Chapter 7 demonstrates the performance of Bolt in a comprehensive benchmark against the fastest existing tables and other widely used hash tables. Finally, chapter 8 concludes the thesis.

1.1 Terminology

Compare and set (CAS) is a hardware supported instruction that allows atomically comparing up to 16 bytes of memory to an expected value and changing it atomically if it matches. We refer to the 16-byte version as CAS in this thesis.

Robin Hood is a variant of linear probing that serves as the backbone of our implementation. Its described in section 2.1.

Lock-free is a property of an algorithm that requires at least one thread to make progress at any point in time. Locks violate this property as a thread might stop making progress and fail to release them, blocking other threads.

1.2 Related work

Maier et al. propose Growt [8], a state of the art concurrent resizable hash table based on folklore [8], a highly optimized concurrent linear probing hash table. Growt's operations are lock-free, except resizing. Growt has low complexity and excellent performance. However, the standard linear probing it employs suffers from high variance in probe distances affecting its performance. Moreover, its delete implementation doesn't reclaim slots, which negatively affects the performance of inserts and finds. A future resize is required to reclaim the deleted slots.

Li et al. [6] present a concurrent resizable cuckoo table that uses fine-grained locking per bucket for correctness. To reduce the number of acquired locks, they use a BFS-based algorithm to find each key's minimal displacement path from its current slot due to a cuckoo eviction.

Shun et al. [11] develop a phase concurrent deterministic hash table based on linear probing that supports one operation within a synchronized phase. They exploit the phase concurrency to implement a linear probing table in a fast and deterministic way. Because the implementation rearranges the table for inserts and deletes, the phase aspect is necessary to prevent finds from missing keys in the table.

Junction [10] is a popular re-sizable concurrent hash table that requires an invertible hash function and claims memory through *quiescent-state-based reclamation* (QSBR) protocol. This requires the user to regularly call a designated function.

Kelly et al. [5] present a lock-free concurrent Robin hood hash set without support for resizing. A hash set is a special case of a hash table where no values are associated with keys. It uses the K-CAS [2] primitive which allows changing multiple memory locations atomically and timestamps to make each operation atomic. In our measurements, we found K-CAS Robin Hood to have poor portability and high performance overhead. To our knowledge, this is the only existing concurrent implementation of a Robin Hood hash set, and no concurrent implementation exists that also supports resizing.

Chapter 2

Background

2.1 Robin Hood Background

Robin Hood hashing was a relatively obscure collision handling method invented in 1986 [3] that has recently gained popularity in part through Rust, which adopted Robin Hood as its standard hash table. Ska [12] demonstrates that Robin Hood can be used to develop a state of art hash table that outperforms Google's dense hash map significantly. Robin Hood is based on the concept of stealing from the "rich" and giving it to the "poor". In this case, the rich are elements of the hash table located close to their hash slot, while the poor are located far from it. All entries of a Robin Hood hash table maintain their distance from their original slot: the probe distance. When new keys are inserted into the hash table, the incoming key's current probe distance is compared with each consecutive slot's key's probe distance until one with a smaller probe distance is found. When such a key is found, the incoming key replaces the key. The evicted key then begins looking for a new slot. Wraparounds are allowed to deal with keys that hash close to the end of the array as in linear probing. The process repeats until the last moving entry finds an empty slot. This process can be visualized in figure 2-1.

For key lookups, the Robin Hood heuristic offers the benefit of pruning searches, which is especially effective for keys that don't exist in the table. A lookup proceeds as follows. Robin Hood maintains the probe distance for a key as the key is searched.

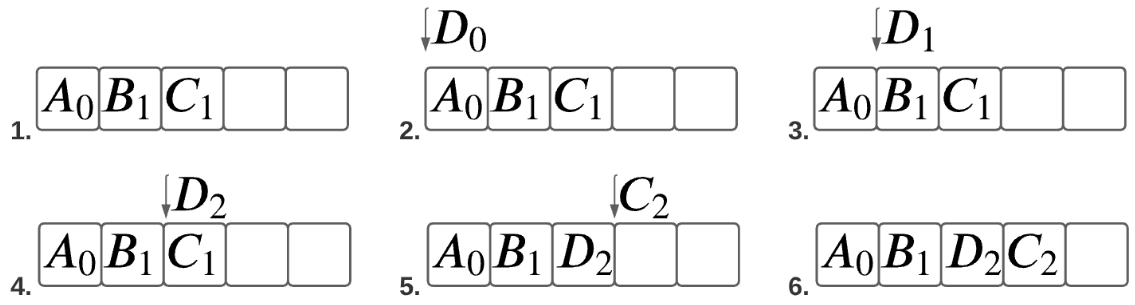
1. A_0 B_1 C_1
2. D_0 A_0 B_1 C_1
3. D_1 A_0 B_1 C_1
4. D_2 A_0 B_1 C_1
5. C_2 A_0 B_1 D_2
6. A_0 B_1 D_2 C_2

Figure 2-1: Inserting D with the Robin Hood heuristic. The under script represents the current probe distance. D starts from its hashed slot to find a slot to insert itself in (2). It keeps moving right until it encounters an element whose probe distance is lower than D's current probe distance (4). D swaps with C, and C goes to find a new slot (5). It encounters an empty slot in (6), which completes the insert.

If a slot whose entry's distance is smaller is found during the linear scan, then the key cannot exist in the table. This is because if the key did exist, it would have a probe distance of at most the scanned entries. This is referred to as the **Robin Hood Invariant**. A successful lookup is visualized in figure 2-2. Unsuccessful lookup is visualized in figure 2-3.

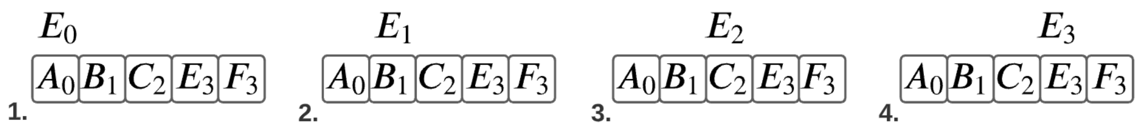
1. E_0 A_0 B_1 C_2 E_3 F_3
2. E_1 A_0 B_1 C_2 E_3 F_3
3. E_2 A_0 B_1 C_2 E_3 F_3
4. E_3 A_0 B_1 C_2 E_3 F_3

Figure 2-2: A successful lookup for key E using the Robin Hood heuristic. The find operation for key E starts from its hash slot. It iterates right past keys whose probe distance is lower than E's current probe distance (1 - 3). It finally finds E in (4).

Finally, deletes work by finding the desired key and removing the entry from the hash table. Instead of opting for a tombstone approach as is typical in linear probing and leaving "holes" in the hash table that can't be reused, Robin Hood hashing backshifts all entries to the right of the removed entry that are not in their hashed slot by one as to maintain the Robin Hood invariant and reduce probe distance. This is visualized in Figure 2-4.

Robin Hood hashing achieves amortized constant time in operations and boasts

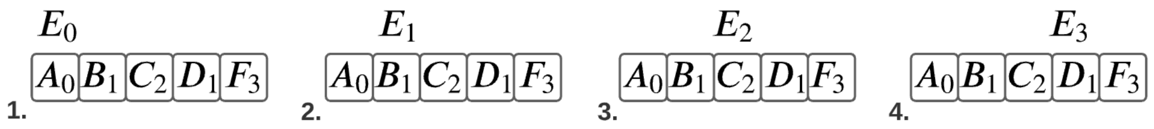
E_0
A_0 B_1 C_2 D_1 F_3
1.
E_1
A_0 B_1 C_2 D_1 F_3
2.
E_2
A_0 B_1 C_2 D_1 F_3
3.
E_3
A_0 B_1 C_2 D_1 F_3
4.

Figure 2-3: An unsuccessful lookup for key E using the Robin Hood heuristic. The find operation for E starts from its hashed slot and iterates right until it encounters D, which has a higher prob distance without encountering E (1 - 4). Because of the Robin Hood invariant, its not possible for E to exist. Thus the search for E ends. Unlike linear probing, reaching an empty slot to terminate the find was not necessary.

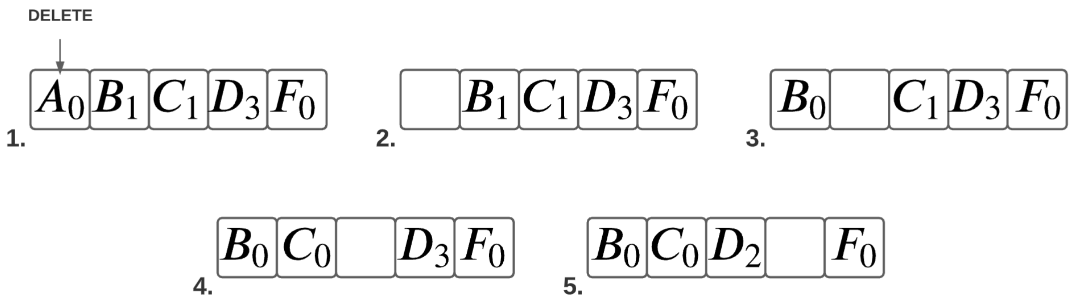

Figure 2-4: Deleting an element with the Robin Hood heuristic. After A is deleted, B,C and D are shifted back since they are at not at their hashed slot and there is no empty key between A and them (1 - 5). F remains at its hashed slot.

$\mathcal{O}(\log n)$ time [3] with high probability for a constant load factor < 1. It is cache-friendly due to good locality and performs well under high load because it minimizes variances of probe distances.

2.2 Challenges

The challenge of making the serial Robin Hood hash table concurrent lies in the nature of the algorithm used to enforce the Robin Hood invariant in insertions and deletions: it can potentially modify large amounts of memory. Keys that already existed in the table can go temporarily missing during an insert since it does a local reorganization of the table to enforce the invariant. Concurrent operations have to properly handle this, which requires some level of communication between different threads about keys that are being moved. Note this problem does not happen in the basic version of linear probing since existing keys will not be moved unless they are deleted.

Delete shifts keys back, while find operates forward in the table. As a result, a concurrent find can misses keys that are shifted to a region that it has already scanned due to a concurrent delete.

There are multiple solutions to these problems. A read/writer lock to protect indices of the table is the most straightforward one. However, care has to be made to avoid deadlocks in this solution; for example, this can happen if an operation wraps around the table, potentially taking locks in a non-increasing order. There is also performance scalability to locks that can hamper performance, since they require memory bandwidth, which is a limitation factor at high cores counts. Skewed distributions are especially problematic for lock-based approaches since many threads will contend for the same locks, causing massive performance deterioration and performance serialization.

Another solution adopted by K-CAS Robin Hood [5] is to use K-CAS [2],which allows atomically modifying large regions of memory. The limitation of this approach is that K-CAS is an expensive operator, which results in a performance penalty. It also poor contention mitigation since concurrent K-CAS operations in the same region will fail.

There are also problems not unique to Robin Hood that cause performance limitations in concurrent hash tables stemming from memory bandwidth limitations that we address.

2.3 K-CAS Robin Hood

Kelly et al. present K-CAS Robin Hood [5], which is a concurrent Robin Hood hash set algorithm. A hash set is a special case of a hash table where only the key is stored. The implementation maintains Robin Hood's properties in the concurrent context, including low expected probe distance and early search culling.

K-CAS [2] is a multiword compare and swap operation on K distinct memory locations that either succeed or fail together. Traditionally, K-CAS requires memory reclaimers, making it expensive to use in the concurrent context. K-CAS Robin Hood uses a variant developed by Arbel-Raviv and Brown [5] that doesn't require a memory reclaimer system.

K-CAS Robin Hood uses K-CAS to modify the memory moved by insertion and deletion atomically. This guarantees that insertion and deletion will be atomic with respect to each other. The find operation doesn't use a K-CAS, so K-CAS's guarantees don't apply. Therefore it's necessary to also have timestamps that are sharded across the table. More specifically, the table is split into multiple contagious shards, and there is a timestamp associated with each shard. The timestamps are updated when the update operations modify the underlying shard. A find operation has to collect the shards' timestamps and verify that they have not changed if it doesn't successfully find the key.

We found K-CAS Robin Hood to be orders of magnitude slower than Bolt in our measurements, which we suspect is due to K-CAS's high overhead. Because K-CAS Robin Hood is a hash set, it also doesn't support storing values for the key while Bolt does. This gives it less functionality, and a significant unfair advantage in the concurrent context than a hash table since atomically moving 8 bytes of memory (corresponding to a key) is cheaper than 16 bytes (a key and value). It also requires less memory bandwidth, which is the limiting factor for performance at high core counts. K-CAS Robin Hood doesn't support resizing, which would have added overhead. Even with all these advantages, Bolt significantly outperforms K-CAS Robin Hood. However, K-CAS Robin Hood is lock-free while Bolt is not. This provides stronger guarantees in terms of progress.

Bolt doesn't use K-CAS to rearrange memory; it instead uses a combination of compare and set (CAS) primitive to rearrange memory in the fast path and locks in the slow path. This allows it to be significantly faster and more portable than K-CAS Robin Hood. The main design similarity between K-CAS and Bolt is the use of timestamps. We also employ version numbers in a similar way. However, our find operation has a fast path that doesn't require checking the version numbers regardless of whether the key is found or not.

Chapter 3

Preliminaries

In this chapter, we give a high-level overview of Bolt and its different components. We describe the data structures utilized by Bolt in section 3.1, and the structure of Bolt and its underlying serial implementation in section 3.2. We then provide the API and helper functions in the section 3.3.

3.1 Underlying Data Structures

3.1.1 Lock Manager

Our algorithm treads an intricate balance between slow paths that use locks and fast paths that don't. We need a well-crafted locking scheme to make the interaction work. The lock manager data structure is used to implement that mechanism. At a high level, we divide the array into multiple contiguous segments, each of which is protected by a different lock.

Each thread has its own lock manager; however, the underlying locks that are managed are the same between all threads to facilitate synchronization. Each lock has an associated version number, which is increased atomically when the lock is taken.

The lock manager API is as follows:

lock(index)	The lock function attempts to acquire the index's lock. Blocks until the lock is acquired.
speculate(index)	Speculate returns whether the index's lock is free and internally records the version number for that index.
finish_speculate()	Returns true if locks speculated upon through speculate(index) have been grabbed between this function call and the first speculate(index) call. Returns false otherwise. Calling this function expunges all speculated indexes for a given thread.
release_locks()	Releases all acquired locks.

3.1.2 Thread Handlers

The hash-table is accessed through thread-local handlers. The handler object also has thread-local objects to facilitate table accesses, including thread-local lock manager operations. A thread grabs a handler the first time it accesses the table. It releases it when it's done using the table.

3.2 Bolt Table

We are now ready to describe the details of Bolt. Bolt is a resizable Hash table based on the Robin Hood heuristic, which seeks to minimize the variance of probe distances across the hash table. Through the Robin Hood invariant, we improve the performance of finds by terminating them early. The hash table's tail performance improves because probe distances are balanced across the table. Bolt is engineered for high performance and scalability and comes with contention and NUMA optimizations. Bolt is based on an intricate balance between an atomic "fast path" and a locking slow path to create a fast and highly scalable concurrent hashtable.

The key insight in Bolt is to make a distinction between insertions and finds that can be resolved by looking at the location where the key hashes to, i.e., the **distance zero slot**. In that case, there is a fast path that bypasses locks. This is the common case for reasonable load factors (≤ 0.6).

In Bolt, the table is divided into segments, each of which is protected by a lock. Operations that can be resolved by looking at the distance zero slot bypass the locks and directly operate on memory. Operations that require multiple memory look-ups take locks to properly interact with other operations operating over the same memory region. The complexity arises from the interaction between threads executing the locked path and the non locked path. Most of our find operations resolve using one atomic load, which makes them very fast. We employ version numbers that are embedded in the locks and change when the locks are grabbed in the slow path. A find operation that requires multiple memory loads records the version number of each segment it reads through and verifies they have not changed at the end of the operation. We designed the segment version numbers to only change in the slow path even under high contention, which allows the find operation to be fast. We also support updating the values of existing keys, which is done without locks in the fast path.

3.2.1 Bolt Structure

The underlying representation in Bolt is a flat array. Each entry in the array contains a key, value, and distance. The key and value are 8 bytes, while the distance is one byte. The distance is used to store how far away the key is from its hashed slot. The distances are all initialized to zero. There is also a special sentinel value for empty slots called EMPTY and one for slots locked by a concurrent operation called LOCKED.

In our first description, we omit the resizable aspects of Bolt for simplicity. Chapter 5 addresses the resizable version.

3.2.2 The underlying base algorithm

As the base of our hash table, we implemented a highly optimized Robin Hood hash table. One of the most notable modifications is that we don't allow **wraparounds**;

instead we allocate a buffer region to deal with elements that hash to the end of the array. This approach is outlined by Ska's Robin Hood [12].

The construction is as follows: we create a table of size $x + C \log x$ where x is the "capacity" of the hash-table, and $C > 1$ is a constant. The hash function only uses size x, so hash($\mathcal{K}$) $< x$ for all $\mathcal{K}$ where $\mathcal{K}$ is a key corresponding to an entry to the hash-table. We then disallow wraparound and use the extra space at the end instead for elements that would have wrapped around. For reasonable load factors, we can bound the extra space by $O(\log n)$ by noting that serial Robin Hood runs in $O(\log n)$ with high probability for insertion and deletion and iterates through memory linearly [3].

This approach has a few advantages:

1. It simplifies the approach for locking because it preserves global order of indices. Wraparound results in a loss of global order in indices, which can cause deadlock if not handled properly.

2. Compact code is generated for finding keys (no wraparound checks), resulting in better performance.

The main disadvantage is that support for resizing is necessary, although it is extremely unlikely for the buffer to fill up before the table does.

3.2.3 Terminology

Distance	The array index difference between the slot the key is stored in and the slot it hashes to.
Distance-zero Index:	The index that the key hashes to, i.e., the ideal slot for the key.
Table:	The underlying array used in the hash table.
Entry:	Entry is a structure containing the key, value associated with it, and the probe distance. This are represented by key, value, dist respectively. It is used to represent members of the hash table.

3.3 Hash table API

insert(key, val)	Inserts the key in the table with the value provided if the key doesn't exist. Returns whether the insert was successful.
InsertOrUpdate(key, val)	If the key exists in the table, it overwrites the value. Otherwise, it inserts the key with the value provided.
find(key)	Searches for the key in the table. It returns whether the key was found, and the value for the key.
delete(key)	Deletes the key if it exists, and returns whether the key was deleted.

3.3.1 Helper functions

Here we provide a high-level description of the helper functions utilized internally by the hash table API:

---

locked_insert(entry_to_insert, insert_index)

Inserts the entry into the table using locks at insert_index. It conducts necessary migration of evicted elements to maintain the Robin Hood invariant. Assumes that insert_index is the location for the entry that satisfies the Robin Hood invariant. Caller is responsible for acquiring a lock on insert_index, and releasing locks grabbed by the call.

---

distance_zero_insert(key, val, dist_zero_slot):

Attempts to insert the key at the index that the key hashes to, the dist_zero_slot, atomically. Returns if it was successful in inserting the key and whether the key was found in the table (in which case it is not inserted).

---

find_next_index_lock(manager, start_index, key, *distance_key):

Note *distance_key is a pointer, and is modified by this function. Searches for key starting from start_index using the Robin Hood invariant, locking indices before accessing them. Assumes distance_key is the current distance of key from its hash slot. Computes the insert index, which is where the key would have been inserted, and updates distance_key to correspond to the distance from the insert index. It returns the insert index and whether the key was found in the table. Caller is responsible for releasing locks.

---

get_thread_lock_manager():

Returns the lock manager for the thread.

---

compare_and_set_key_val(index, prev_key, new_key, new_val):

Atomically checks the key located at index and if it matches prev_key, it atomically changes the entry stored at that index to new_key and new_val. Returns the current key and val at the index before the change.

---

do_atomic_swap(swap_entry, index):

Does an atomic swap between the entry at the index provided and swap_entry. Returns the previous entry.

Chapter 4

Algorithm Details

In this chapter, we describe the algorithmic details of Bolt. Section 4.1 explains how insert and InsertOrUpdate work. Section 4.3 outlines delete. Finally, Section 4.4 explains the find operation.

4.1 Insertion

4.1.1 Distance zero insert

Algorithm 1: distance_zero_insert(key, val, dist_zero_slot): returns if key is inserted, and if it exists.

```
1  Entry entry_to_insert = {.key = key, .val = val, .dist = 0}
2  key_exists = false
3  inserted = false
4  if table[dist_zero_slot].key == EMPTY then
5      p = compare_and_set_key_val(index, EMPTY, {.key = key, .val = val })
6      if p == EMPTY then
7          inserted = true
8      end
9  end
10 if table[index].key == key then
11     key_exists = true
12 end
13 return inserted, key_exists;
```

Algorithm 1 shows the pseudocode for distance zero insert. distance_zero_insert first checks if dist_zero_slot (the hash slot of the key) is empty. If so, it attempts to claim this entry to insert the key atomically. If it is successful, it can return that the key was inserted and thus does not exist. Otherwise, it checks if the dist_zero_slot key matches the key. If so, then it returns that the key exists and that it was not inserted. Otherwise, it returns failure, indicating to the caller to try the insert in a different way.

4.1.2 Insert

Algorithm 2: insert(key, val): returns true if key was not found, and thus was inserted, and false otherwise.

```
1  index = h(k)
2  is_inserted, key_exists = distance_zero_insert(key, val, index)
3  if is_inserted then
4      return true
5  end
6  if key_exists then
7      return false
8  end
9  manager = get_thread_lock_manager()
10 entry_to_insert = {.key = key, .val = val, .dist= 0}
11 next_index, found = find_next_index_lock(manager, index, key,
     &entry_to_insert.dist)
12 if found then
13     manager.release_all_locks();
14     return false
15 end
16 locked_insert(entry_to_insert, next_index)
17 manager.release_all_locks()
18 return true
```

Algorithm 2 shows the pseudocode for insert. Insertion first attempts a distance zero insert. If that inserts the key, then it returns successfully. If it finds the key already exists, then it returns insert failed.

If both are not conclusive, then it has to trigger the slow path to insert the key. This proceeds in much the same way as the serial insertion. First, it checks if the key exists in the table. If it exists, then it returns that the key was found. Otherwise, it invokes locked_insert with the index that the key that is inserted should swap with. It then releases all the locks that were grabbed from calling locked_insert and then returns that the key was successfully inserted.

4.1.3 Locked Insert

Algorithm 3: locked_insert(entry_to_insert, swap_index)

```
1 swap_index = insert_index
2 while true do
3     entry_to_insert = do_atomic_swap(entry_to_insert, swap_index);
4     if entry_to_insert.key == EMPTY then
5         return true
6     end
7     swap_index = find_next_index_lock(manager, swap_index, key,
        entry_to_insert.dist)
8 end
```

Algorithm 3 shows the pseudocode for locked_insert. Locked insert proceeds very similarly to the part of the serial insert procedure that displaces existing keys to satisfy the Robin Hood invariant. It assumes that a lock has already been grabbed on insert_location. In a loop, It swaps the key that is being moved with the entry at swap_index (initially insert_index) atomically so it can properly handle a concurrent distance zero insert / InsertOrUpdate. If the index that was swapped with contained an EMPTY entry, then it returns as it didn't displace an existing key. Otherwise, it identifies the next index, which can swap with the displaced key using the robin-hood invariant by using find_next_index_lock. It then goes on to repeat the procedure and move the displaced key.

4.2 InsertOrUpdate

Algorithm 4: InsertOrUpdate(key, val)

```
manager = get_thread_lock_manager()
index = h(k)
i = index
for ( dist = 0; table[index].distance >= dist; ++dist, ++i ) {
    while table[i].key == key do
        prev_key, prev_val = compare_and_set_key_val(index, key, key,
          val)
        if prev_key == key then
            return;
        end
    end
}
is_inserted, is_found = distance_zero_insert(key, val, index)
if is_inserted then
    return
end
entry_to_insert = {.key = key, .val = val, .dist= 0 }
next_index, found = find_next_index_lock(manager, index, key,
  &entry_to_insert.dist)
if found then
    table[next_index].val = val
    manager.release_all_locks();
    return
end
locked_insert(entry_to_insert, next_index)
manager.release_all_locks();
return
```

Algorithm 4 shows the pseudocode for InsertOrUpdate. InsertOrUpdate first searches for the key using the Robin Hood invariant without acquiring any locks to update it if it exists. If the key is found, it attempts to change the value atomically. There is a subtle check to see if the procedure can return: it checks if the key present at that location when the CAS executed is the same key that it is updating. If so, it can return even if the CAS was unsuccessful since another update must have updated the key, and therefore the updates cancel out. Note that we use a strong CAS that cannot spontaneously fail.

If the key is not found through the search, it must trigger the slow path to insert/update the key. Note that this does not mean the key does not exist in the table; it might just have been moved around by concurrent operations.

It then attempts a distance zero insert. If this is successful in inserting the key, then it can return. Otherwise, it enters the slow path. It checks if the key exists in the table using find_next_index_lock. If it exists, then it updates the value of the key. If it does not exist, then it will insert the key. It does this by invoking locked_insert with the index that the key that is to be inserted should swap with. It then releases all the locks grabbed from find_next_index_lock and locked_insert and then returns that the element was inserted.

4.3 Deletion

Algorithm 5: delete(key, val)

```
index = h(k)
Entry entry_to_delete = {.key = key, .val = val, .dist = 0}
index, found = find_next_index_lock(index,key, &entry_to_delete.dist)
if !found then
    manager.release_all_locks();
    return false
end

manager = get_thread_lock_manager()
next_index = index+1
manager.lock(next_index)

while table[next_index].distance > 0 do
    entry_to_move = do_atomic_swap(LOCKED_ENTRY, next_index)
    table[current_index] = entry_to_move
    table[current_index].dist −= 1
    current_index = next_index
    next_index += 1
end
table[current_index] = EMPTY_ENTRY
manager.release_all_locks();
return true
```

Algorithm 5 shows the pseudocode for delete. In deletion, it first executes a locked search using FindNextIndex to find the key in the table. If it does not exist, then it can return failure. Otherwise, it starts the shift algorithm to eliminate the hole created by the delete as in the serial algorithm. Care has to be taken to maintain correctness in the concurrent context for shifting elements back.

For each element it has to shift back, it first takes the lock corresponding to the index it wants to move back, and then atomically swap the entries with the LOCKED key, which will make sure the key will not change due to a concurrent InsertOrUpdate operation (a distance zero insert cannot conflict because an entry inserted by a distance zero insert would not be shifted back). Then it shifts it back by one. This repeats until it encounters an element that does not need to be shifted back either because it is EMPTY or in its distance zero slot.

4.4 Find

Algorithm 6: find_speculate(key, start_index)

```
1  speculate_success = false
2  val = 0
3  found = false
4  index = start_index
5  for ( dist = 0; table[index].distance >= dist; ++dist, ++index ) {
6      manager.speculate_index(index)
7      if table[current_index].key == key then
8          val = table[current_index].val
9          found = true
10         break
11     end
12 }
13 speculate_success = manager.finish_speculate()
14 return val, found, speculate_success
```

Algorithm 7: find(key): Returns value of the key, and whether it was found

```
index = h(k)
zero_dist_key_pair = atomic_load_key_val(index);
if zero_dist_key_pair.key == key then
    return zero_dist_key_pair.val, true;
end
if zero_dist_key_pair.key == EMPTY then
    return -1, false;
end
val,found,speculative_success
tries = 0
while tries < MAX_TRIES do
    tries++
    val,found,speculative_success = Find_speculate(key, start_index)
    if speculative_success then
        return val, found
    end
end
entry_to_find = {.key = key, .val = val, .dist = 0}
index,found = find_next_index_lock(index,key, &entry_to_find.dist)
val = table[index]
manager.release_all_locks();
return val,found
```

Algorithm 7 shows the pseudocode for find. In find, it first attempts a **distance zero find**: it loads the entry in the table located at the hash slot of the key using an atomic load, getting both the key and value atomically. If the key is the requested key, then it returns the value. If the key is EMPTY (which indicates this slot is empty), it returns not found. Otherwise, we enter the slow path of find. This executes a speculative search for the key. This proceeds as follows:

It does the serial Robin Hood search algorithm, but before looking at a new index, it calls speculate on that index, which internally verifies the lock for that index is free and copies the version number. After the search completes following the normal serial Robin Hood find algorithm, speculative() is called, which will return if none of the locks that were speculated on earlier were taken at any point after they were speculated on based on their recorded version number. This ensures that no concurrent modify operations were invoked that could have moved keys around, causing find to return that the key did not exist when the key was actually present in the table.

If speculating succeeds, then it returns as in the serial algorithm. Otherwise, the speculative search is repeated a constant number of times. If it fails, then find executes a locked search using find_next_index_lock. It then releases all the locks acquired and then returns the result.

Chapter 5

Correctness

In this section, we show that Bolt is linearizable through an informal proof. We discuss each function and argue correctness. We establish the correctness of insertion in section 5.1, deletion in section 5.2, and InsertOrUpdate in section 5.3. Finally, we prove the correctness of find in section 5.4.

5.1 Insertion

Lemma 1. *A distance zero insert that inserts the key is linearizable with respect to a concurrent Insert.*

Proof. Let $\mathcal{Z}$ be the distance zero insert, and $\mathcal{L}$ be the slot corresponding to $\mathcal{Z}$. Consider a concurrent insert operation $\mathcal{I}$ that is operating over the same region. The only way $\mathcal{I}$ can conflict with $\mathcal{Z}$ is if it attempts to modify $\mathcal{L}$.

If $\mathcal{I}$ also attempts to insert on $\mathcal{L}$, then its CAS will fail since $\mathcal{Z}$ succeeded. If $\mathcal{Z}$ succeeds, then it executes an atomic swap with $\mathcal{L}$. This atomic swap only succeeds when it atomically reads $\mathcal{L}$'s value and sets it to the target value. Thus, $\mathcal{I}$ will see the most up-to-date value for $\mathcal{L}$. Therefore, it must appear that $\mathcal{I}$ occurs after $\mathcal{Z}$. □

Lemma 2. *A distance zero insert is linearizable with respect to a concurrent InsertOrUpdate.*

Proof. All line numbers are in reference to the pseudocode for InsertOrUpdate. Lemma 1 handles the case where InsertOrUpdate inserts the key. We only have to consider the case where InsertOrUpdate updates the value of the key because it already exists.

There are two ways they can happen: InsertOrUpdate finds the key and does a successful CAS at line 6 or it updates it after locking at line 19. If InsertOrUpdate happens through the CAS, then the CAS will linearize them. If it's the latter, they cannot conflict because distance zero insert only attempts CASing empty keys.

□

Lemma 3. *A distance zero insert is linearizable with respect to a concurrent Delete.*

Proof. Note that a concurrent delete will not shift an element inserted by a distance zero insert (unless a future insert moved the element) since the delete procedure only shifts elements with non zero distance. So the only collision possible is if a distance zero insert and delete are operating on the same key. In that case, if the concurrent delete does not see the key inserted, then it can be linearized to come before. If it does, it will delete it and is linearized later than the distance zero insert. □

Lemma 4. *A locked insert is linearizable with another locked insert / deletion / and the locked part of InsertOrUpdate.*

Proof. First, note that locked operations take the indices' lock before they read / modify them, and they do not release locks until the operation is complete. Further lock operations take locks of increasing indices, preventing deadlock. Note that even though deletion shifts elements back, it still takes locks of increasing indices. Let the two operations be G and F. WLOG, assume F has taken a lock of a later index than G. Then if G,F end up visiting the same index at some point through their procedure, then G has to stop and wait for F to complete because F would be holding the lock for that index. Further, G will not see any of the read or modified locations until F releases the locks. □

Lemma 5. *A locked insert is linearizable with respect to a concurrent InsertOrUpdate.*

Proof. Lemma 1 can be easily extended for the distance zero insert attempt by InsertOrUpdate (line 12) and the loop that searches for the key and does a CAS to update it. The only subtlety to consider is the CAS at line 6 can succeed without actually updating the value as long as the key doesn't change when it's CASed. However, this can't happen due to a concurrent insert since an insert always changes both the key and the value atomically. Starting from line 17, InsertOrUpdate acquires locks so we can apply Lemma 4. Therefore locked insert and InsertOrUpdate are linearizable with each other. □

Theorem 1. *Insert is linearizable with all concurrent modify operations.*

Proof. There are two ways an insert could complete: through a distance zero insert or a locked insert. Both are linearizable with all other modify operations based on the lemmas above. Therefore insert is linearizable. □

5.2 Deletion

Lemma 6. *Delete is linearizable with respect to a concurrent InsertOrUpdate.*

Proof. Lemma 3 can be extended to handle the linearization of the distance zero insert attempt by InsertOrUpdate (line 12). Starting from line 17, InsertOrUpdate acquires locks so we can apply Lemma 4. The remaining case deals with CAS that updates values in InsertOrUpdate for an existing key. Delete handles by making sure that it first atomically reads the key before moving a key and swaps it with LOCKED. This will result in line 6 in concurrent InsertOrUpdate that CASes a key that is being moved by delete in either succeeding and have it changes properly moved by Delete, or failing which are both linearizable. □

5.3 InsertOrUpdate

Lemma 7. *InsertOrUpdate is linearzable if it executes an update for the key.*

Proof. Let the InsertOrUpdate procedure be $\mathcal{I}$. Line numbers are in reference to Pseudocode for InsertOrUpdate. For $\mathcal{I}$ to return successfully after updating the key, there are two possibilities: it has to either execute the CAS at line 6 with the key not changing or execute a store at line 19 after acquiring locks.

In the first case, if the previous key has not changed, it means that either the CAS was successful, at which point its linearizable, or a concurrent modify operation updated the value of the key, making I's CAS fail. Note that the concurrent operation can't be a Delete or Insert since those operations change the key if they change the value atomically. Therefore the concurrent modify operation is another InsertOrUpdate, which would imply that it is updating the same key. Therefore it is linearizable.

The second case is similar to the first, except that the second case doesn't check if the key doesn't change and does a blind store. At this point, $\mathcal{I}$ has already grabbed locks, so the only operations that could be modifying the same memory are ones that don't take any locks. These are either distanceZeroInsert in which Lemma 2 applies or a concurrent InsertOrUpdate, in which case the analysis for the first case applies. □

Theorem 2. *InsertOrUpdate is linearizable with respect to all concurrent modify operations.*

Proof. InsertOrUpdate is linearizable with all modify operations: its linearizable with Delete by Lemma 6, with a concurrent Insert by Theorem 1, and with a concurrent InsertOrUpdate that does an insert by Theorem 1 and an update by Lemma 7. □

5.4 Find

Lemma 8. *A distance zero find that finds the key is linearizable.*

Proof. All modify operations atomically update both the key and value. Therefore since a distance zero find does an atomic load of the key and value, if it finds the key its looking for , the key must exist in the table at some point after the invocation of the find and before its return (it might be promptly deleted before the find returns, however). If the key was deleted or moved, it would have been replaced with

EMPTY or a different key respectively. Therefore, a distance zero find that finds a key linearizable. □

Lemma 9. *A distance zero find that returns the key is not found is linearizable.*

Proof. Let i be the index that the key $\mathcal{K}$ hashes to (the first index searched). A distance zero find returns key not found if it finds that table[i].key represent the EMPTY sentinel. Thus it's necessary that $\mathcal{K}$ doesn't exist later in the table / is not being moved for this to be correct.

Note that this holds trivially in the serial case because of the Robin Hood invariant (otherwise, the key would be unreachable). Therefore we only have to consider concurrent operations that could temporarily move key $\mathcal{K}$ while leaving index $\rangle$ empty. The two types of operations that move keys are locked inserts or locked deletes. A distance zero insert does not move keys, so it can safely be ignored. Only one operation can move key $\mathcal{K}$ at a time because the operations need to acquire a lock and find loads table[i] atomically.

If $\mathcal{K}$ is in the process of being moved later by an insert operation J, which inserts key $\mathcal{L}$ (for $\mathcal{L} \neq \mathcal{K}$), then it's not possible for index i to be empty. This is because if $\mathcal{K}$ was initially at distance zero, then $\mathcal{L}$ would atomically replace it first, so index i wouldn't be empty. Otherwise, if $\mathcal{K}$ had a probe distance > 0, there must not be empty slots before $\mathcal{K}$ starting from index i as this would violate the Robin Hood invariant.

If there is a concurrent delete operation that makes index i empty, it would be the last step in the delete procedure. After this, the delete operation is done. Because delete sets a slot empty after moving all the relevant keys, If $\mathcal{K}$ exists, there would be an empty slot between it, violating the Robin Hood invariant. Therefore, $\mathcal{K}$ is not within the hash table. □

Chapter 6

Resizing

In this chapter, we discuss how to support resizing in Bolt. Section 6.1 addresses relevant background. Section 6.2 addresses the resizing heuristics employed by Bolt. Finally, section 6.3 outlines the resize protocol.

6.1 Background

Resizing is the scheme that hash tables implement to deal with an unknown number of keys that will be resident in the table in the future at initialization. In this thesis, we focus on a resize that increases the table's size due to insertions. Reducing the size of the table triggered by too many deletes is left for future work.

The standard approach for resizing is as follows: when the hash table's size crosses a certain fraction, the hash table internally triggers a resize of the underlying table. In general, the constant used in deciding when to resize is implementation dependant. It represents a trade-off between how often the table resizes and the maximum load factor of the table.

When a resize is triggered, the hash table creates a new array of a larger size and copies the new table's old data. In the serial case, resizing a table is relatively straightforward. A counter is kept that counts the number of keys that are currently in the table. This is updated by the insert and delete operations. If this counter exceeds the threshold in an insert operation, a resize is triggered, which inserts the data into a bigger table. The bigger table is a constant multiple of the old in size so that the work for resizing can be amortized based on previous inserts to be $O(1)$.

6.1.1 Challenges

Unlike serial resizing, concurrent resize is challenging to implement correctly in a performant manner. It is non-trivial to keep track of the table's size as the basic solution of using an atomic counter updated by concurrent threads introduces a single point of contention, which is not feasible due to the cache coherency protocol in modern multicore architectures. There also needs to be a protocol to deal with multiple threads deciding to resize and concurrent operations when a resize is initialized. It is also necessary to distribute work among threads dynamically. Finally, care must be taken to minimize contention when moving data from the old table to the new table.

6.2 Bolt Resizing Scheme

The main design principle we applied for Bolt is to maximize resizing performance while minimizing the performance cost incurred to enable resizing. Our resize protocol achieves this with an overhead of $< 5\%$ for the table's operations while still outperforming all other algorithms on the resizing benchmark. Our protocol for resizing is inspired by Growt [8]. We will first address several subproblems and then outline our resizing algorithm.

6.2.1 Heuristics for Resizing

Keeping an exact count of the number of keys in a concurrent table usually requires a shared variable or counting networks, which are too expensive to incorporate in a hash table. Instead, we developed a heuristic to allow us to resize without knowing the table's size in a fast and scalable manner, taking advantage of the Robin Hood heuristic. We designed our heuristic to work well for load factor above 25%, which is sufficient for resizing as most tables don't resize below that load factor since that would be too frequent and unnecessary.

The goal of resizing is two folds: to avoid performance deterioration and avoid running of space. The obvious way to do this is by keeping track of the load factor directly. This predicts the table's performance and trivially avoids running out of space (or some direct approximation). However, that is not strictly necessary, and our table exploits this. Instead, we keep a counter that allows us to estimate the hash table's performance while ensuring that it doesn't run out of space.

We do this by keeping track of the number of keys that have distance ≥ 1 in the table per lock segment. This count is **significantly cheaper** to compute than the size of the table because it only changes when an inserted key collides with an existing key, while the size changes every time there is an insert. This count gives a sense of how many keys are not being placed at their original spot, giving a proxy to estimate the table's performance. If more keys are placed in their hashed slot, then the table has low probe distance, which implies good performance. Conversely, if more are placed away from their hashed slot, then we expect the probe distance to be higher. Because only keys with dist ≥ 1 contribute to the probe distance, we can use this to give a lower bound on the probe distance. The Robin Hood invariant is also key to making this measure more accurate: Robin Hood kicks out keys from their original slot as load factor increases, which means we get a stronger correlation between load factor and distance ≥ 1 keys. It's worth noting that one could take a similar approach in linear probing with some metadata, but we believe it would be a worse measure. This is because linear probing does not kick out existing keys from their hashed slot as the load factor increases.

We combine this heuristic with one of the conditions that ska [12] uses for resizing. In particular, we also resize the table when the probe sequence length reaches a maximum size. As outlined in section 3.2.2, we size our table to have an overflow buffer room to avoid having to wrap around. We set the size of this wrap-around region to be bigger than the maximum size so that a resize is triggered if an insert ends up reaching the end of the buffer. This avoids the case when an insert can't find a slot in the table.

By combing the two heuristics, we can satisfy both conditions: by bounding the number of distance ≥ 1 keys we can get good performance, and by the max probe length heuristic, we can guarantee to resize before inserts fail because they can't find a spot in the table, while also guaranteeing that no wrap around is necessary.

The overhead of keeping track of distance ≥ 1 count is minimal in Bolt because it only has to be updated when keys are moved from distance zero to distance one, which requires locks in our approach. Therefore by incorporating the counters into the lock segments, we can update them when the locks are released, requiring no extra synchronization.

6.2.2 Distance ≥ 1 count and load factor

It turns out there is a strong relationship between the load factor the number of keys with distance ≥ 1 that allows a very accurate prediction of load factor > 0.25. The mathematical intuition for this comes from this analogy: we can think of inserting a new element as randomly picking an index in the table (assuming a uniform hash function), and from there doing an insert procedure. The probability that the distance ≥ 1 count up is related to the number of keys existing in the table (i.e the load factor). Each insert repeats this experiment. Note that if we assume the hash function is random, this is independent of the key distribution that the user employs (ignoring duplicates), so we expect this to converge regardless of the underlying key distribution for a sufficient large number of keys in the table.

6.3 Resize Outline

Algorithm 8 outlines the resize protocol. Resize is triggered once the distance ≥ 1 load factor of a lock segment in the table exceeds λ, or the max probe distance exceeds γ. We choose λ to correspond to a load factor of 0.5 exploiting 6.2.2. We set γ to 256, which is the maximum buffer size. Note that technically γ should be $O(\log n)$, however, since the maximum size of memory is 2^{64}, 256 is a sufficient upper bound. To facilitate resizing, a key structure is the thread handler, which is local to each thread.

This is a wrapper for Bolt, and all accesses to Bolt go through it. To support resizing, we augmented Bolt to have a **version number**, and a shared counter representing the outstanding references to the table and is used to manage the life cycle.

When a thread first access Bolt, it creates a new thread handler, acquires the global lock, copies the pointer of the latest version of Bolt, increments, and then release the lock. It keeps this handler until it's finished using the table. The lock is necessary to ensure the table is not freed in the middle of a thread creating the thread handler object. However, since this only happens once and future accesses to the table by this thread don't acquire the locks, it doesn't impact performance. During resizing, care must be taken to avoid elements being changed/inserted/deleted as the table is migrated. To accomplish this, we drain all locks in the table in parallel, which blocks any operations that take locks. This introduces a complication for table operations such as insert because a resize can take locks out of order. To handle this, we changed operations that take locks to attempt instead to acquire the lock. If it fails, then they check if a resize has started. If so, then they undo the changes they made to the table. Since their changes are not visible yet, this does not affect correctness.

To handle the fast paths which don't take locks (such as distance zero insert and insert or update), we use the following protocol which we refer to as the **migrator protocol** : each entry in the table is atomically replaced with a LOCKED entry by a resize operation. This entry is then migrated in place into the new table. This requires no modification of the fast path operation that doesn't take locks because they are already designed to handle locked entries.

Growt [8] implements a similar approach; however ours avoids reducing the keyspace by a factor of two, which is necessary for their approach as they have a special marked bit. They describe an extension of their table to get around this, but they don't implement it, and as described, it would seem to have significant performance overhead.

Each thread keeps their local table up-to-date with the latest table by checking if the local table has initialized/undergone resizing before executing an operation. If so, they join resizing if it's still in progress, decrement the counter on their local table (freeing it if it reaches zero), and update their local table to the latest version

6.3.1 Resize Steps

Resize is triggered once the distance ≥ 1 load factor of a lock segment in the table exceeds λ or the max probe distance exceeds γ. We choose λ to correspond to a load factor of 0.5 exploiting 6.2.2. We set γ to 256. Below is the procedure to handle resizing:

Algorithm 8: Resize Algorithm

```
1 If a thread detects the condition for resizing, it attempts to start a resize. If
    its the first thread to do so, it becomes the leader. Otherwise it helps with
    resizing.
2 The leader initializes that a resize has started by setting a global flag.
3 Threads coordinate in grabbing all the locks in the table.
4 The leader creates the next table and allocates memory for it.
5 Threads coordinate in initializing the next table with EMPTY entries.
6 Threads coordinate in migrating elements from the old table to next table.
    Elements from the old are replaced with a LOCKED entry, and then moved
    to the new table with synchronization (they are treated as a concurrent
    insert into the table).
7 The leader publishes the next table as the most recent table.
```

Correctness

Here we summarize the correctness argument established throughout section 6.3. What is necessary for correctness is:

1. All keys are migrated into the new table.
2. Operations that happen concurrently with resize are handled properly.
3. Threads are aware of the new version of the table after it resizes so that they can operate on the latest version.

By draining the locks first, no locked operations can modify the table once the migration has started. Because of the migrator protocol, operations that don't use locks are prevented from modifying the table while a migration is in progress.

Draining locks can cause a locked operation that has already modified memory to fail, but because this operation's changes are not visible yet, the operation first undoes its changes and join resizing. These two combined allow (1) to hold.

Before starting, each operation checks for the most recent version of the table. If the table changes after the operation starts, then there is two possibilities. The first is that the operation completes on the old table, in which case the change is guaranteed to be in the new table if it's a modified operation by (1). Or if its find operation, then it's can be linearized to come before the resize since it's concurrent. The second possibility is that it is interrupted before completing by the resize, in which case it will undo its changes, join resizing, and reattempt its operation on the new table. This results in (2) and (3) holding.

Chapter 7

Experimental Evaluation

We performed various experiments to investigate the performance of Bolt and compared the results with other concurrent hash tables. We start with a description of the environment for the test and discuss our competitors in section 7.1. We describe results for operations in isolation in section 7.2. We go over mixed results in section 7.3 and contention results in section 7.4. We discuss results for resizing in section 7.5 and finally compare against K-CAS Robin Hood in section 7.6.

We demonstrate that on a 36-core machine with hyper-threading, Bolt is between 1.57x - 1.73x faster than the fastest publically available non-resizable concurrent hash table (folklore[8]) and 1.45 - 2x faster than the fastest publically available concurrent resizable hash table (growt[8]). It also achieves speedups between 15x - 35x over a highly optimized serial implementation at 36-cores with hyper-threading.

We modified the experiments in Growt [8] that we retrieved from their GitHub repo [7]. We also used their framework to build the different algorithms we compared against.

7.1 Testing Setup

7.1.1 Tested Algorithms

We compare against **folklore** [8], which is the fastest concurrent hash table. It does not support resizing or deleting and is based on linear probing. We also compare against **Growt** [8], which is a resizable concurrent hash table and the fastest among tables that support deleting. Its implementation is based on modifying folklore to support resizing and deletes. Specifically, we compare against their two fastest variants: uaGrowt and usGrowt. We also compare with **Cuckoo** [6], which is part of the libcukoo library; the three variants of **Junction** [10] (junction_grampa, junction_linear, junction_leap), which were published by Pershing and are a variant of open addressing; and the two variants of **TBB** (tbbHM, TBBUM), hash tables provided in the popular concurrent library developed by Intel: the Threading Building Blocks [9]. **K-CAS Robin Hood** [5] is the only other concurrent implementation of Robin Hood. However, it's a hash-set, which does not allow storing a value per key, and its performance is orders of magnitude slower on our machine than all other algorithms, so we don't use it for our experiments. We believe its lackluster performance stems from the lack of portability of K-CAS primitive. We do a back-of-the-envelope calculation to compare with the results presented in the K-CAS Robin Hood paper [5].

We report our speedup with respect to **Ska Robin Hood** [12], which is a state of the art serial table based on Robin Hood.

7.1.2 Experimental Setup

We evaluate the performance of Bolt on an AWS EC2 c5n.18xlarge machine. It is a two-socket machine with two Intel Xeon Platinum 8000 Series Processors. Each socket runs at 3.5 GHz, has 18 cores, and 38 MB of L3 cache. In total, there are 36 cores and 72 hyper-threads. We compiled all programs with gcc 9.3 at -O3 optimization, using -flto for linking, and -mcx16, -msse4.2 for processor-specific instruction support.

7.1.3 Test Methodology

We used the test infrastructure used in Growt to conduct benchmarks [7]. Each test measures the time to execute operations, and we compute each data point by taking the median of five separate runs after discarding the first one for cache warm-up. We report *absolute speedups* of our algorithm on 36 cores with hyper-threading with respect to Ska [12], a state of the art serial hash table based on Robin Hood that outperforms Google's dense hash map. The keys used in the benchmark are generated ahead of time, and the work is distributed among threads. The inputs are scattered across the NUMA nodes by threads.

Most of the experiments are performed using uniformly generated keys. We also use an input with Zipf [4] distribution, which is a skewed distribution for testing contention.

We first do benchmarks on inserts and finds separately without resizing. This allows us to get a sense of how the most common table operations scale in isolation and their overheads. Then we do a mixed benchmark combining finds, insertions, and deletions. Finally, we test the performance of resizing.

7.2 Single Operation

7.2.1 Insertion

We insert 10M uniform random keys into an empty table that has been sized appropriately so that resizing is not required. However, tables that support resizing will still keep track of how full the table is. This gives an advantage to folklore, which does not support resizing. The results are presented in the figure 7-1. We note that Bolt is the fastest across all core counts. It achieves an absolute speedup of 33x on 36 cores with hyper-threading and is 1.61x faster than folklore, our closest competitor.

The performance of Bolt and folklore is comparable up to 18 cores, at which point NUMA effects kick in and Bolt is able to create a large gap. Note that folklore has a notable advantage since it does not support resizing.

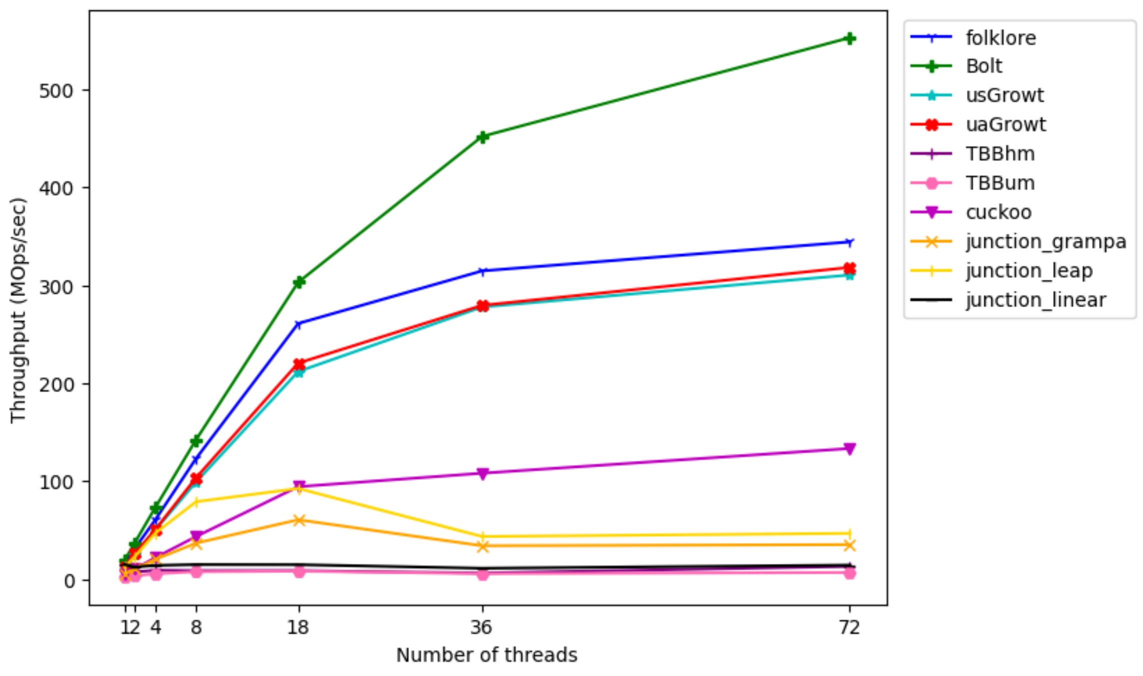

Figure 7-1: Insert Benchmark

The next table is uaGrowt, which is 1.74x slower than Bolt. The difference between usGrowt and uaGrowt is <5%. The next best table is Cuckoo, which is about 4x slower than Bolt. The best variant of Junction is junction_leap, which is 11x slower than Bolt. The best variant of TBB for this benchmark is TBBhm; however its severely outclassed by Bolt and is 43x slower.

7.2.2 Find

We test the performance of both successful and unsuccessful finds on pre-initialized tables. We expect different performance characteristics since unsuccessful finds often require looking at more entries in the table. We look up the 10M keys that were inserted previously in the table (successful finds), as well as 10M random keys (unsuccessful finds).

The results for successful finds (find+) are presented in 7-2, and unsuccessful finds (find-) in 7-3 .

Bolt is the fastest across all core counts and achieves an absolute speedup of 25.9

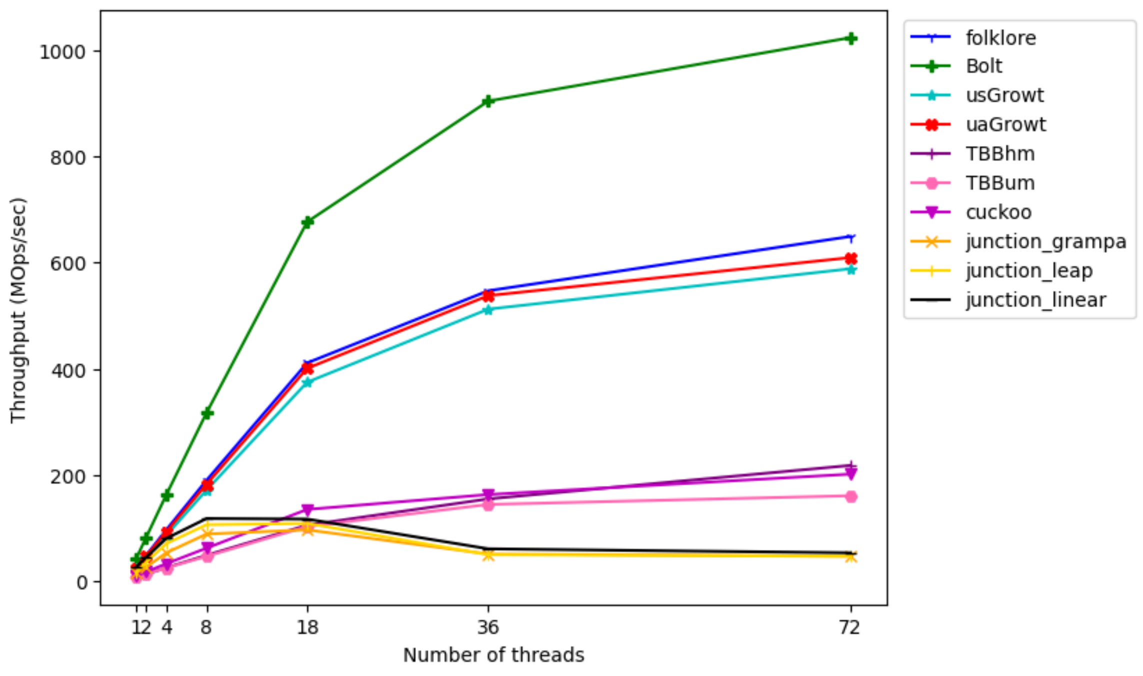

Figure 7-2: Successful Find (Find+) Benchmark

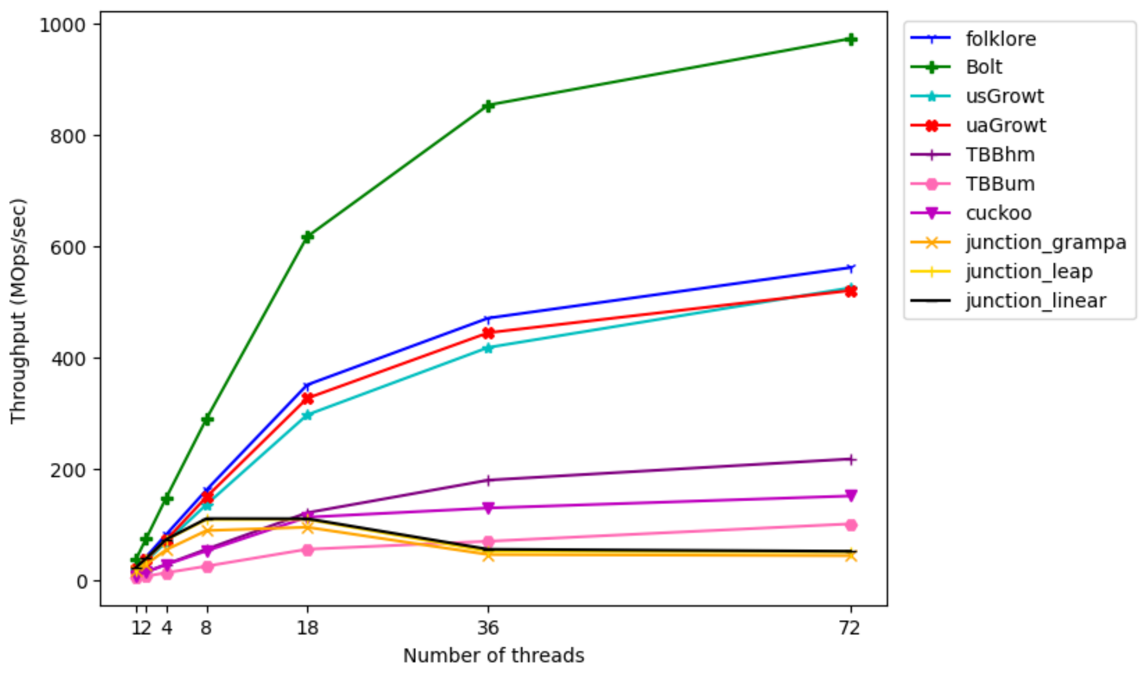

Figure 7-3: Unsuccessful Find (Find-) Benchmark

and 24.9 on find- and find+ respectively at 36 cores with hyper-threading. It scales better with find+ because a successful find requires less memory lookup and memory bandwidth, which is the main limiting factor for Bolt at high core counts. It is faster by 1.73x and 1.58x than its closest competitor folklore. The difference in performance can be explained by the more performant serial implementation of Bolt as well as the power of the Robin Hood invariant over linear probing (which is used by folklore) in allowing fast searches and culling searches early for unsuccessful finds.

Bolt is at least 4.4x faster than all other remaining competitors on both find- and find+. Find- is between 5% - 10% slower than Find+ across all thread counts for Bolt. Find benchmarks are about two times faster than the insertions for Bolt, as no writes are required on the fast path.

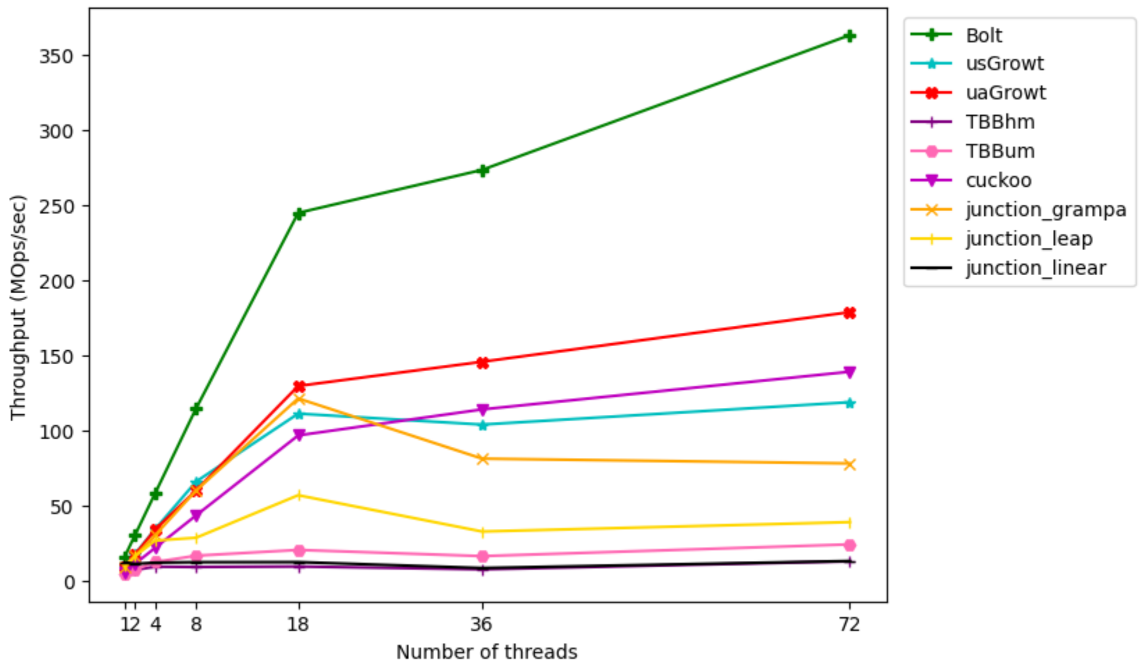

Figure 7-4: Delete/Insert Benchmark.

7.3 Mixed Inputs

Now that we have demonstrated the state of the art performance of inserts and finds in Bolt, we turn our attention to mixed operation benchmarks that are more representative of real-world use cases. Because folklore does not support deletions, it's omitted from these benchmarks.

7.3.1 Insertion/Delete

To test the performance of the interaction between insertion and deletion, we execute a test when an insert follows a delete. This keeps the size of the table constant. More specifically, the table starts prefilled with 2.5M keys, and then the delete of an existing key is followed by an insert of a random key. This is executed 10M times. The results are presented in figure 7-4.

Bolt is the fastest across all core counts and achieves an absolute speedup of 15.8x. The absolute speedup of Bolt is worse than the insert benchmark because deletes do not have a fast path and require locks. However, even with this limitation, it's faster by 2.03x than its closest competitor uaGrowt, and 3.06x faster than usGrowt. Unlike the previous benchmarks, there is a large gap between uaGrowt and usGrowt. It's not clear why this is the case as the overhead in delete between the two approaches is similar to insert. The performance gap between Bolt and uaGrowt and usGrowt is wider than insertion because Growt deletes do not reclaim slots and instead initialize a resize in the future to clean up deleted slots. The next best table is cuckoo, which is 2.61 slower than Bolt. It's worth noting that cuckoo is actually faster than usGrowt in this benchmark.

7.3.2 Insert/Find/Delete

In this benchmark, 100M operations are executed among different threads. Each operation is randomly sampled with probability: x finds, $(1-x)/2$ inserts, and $(1-x)/2$ deletes.

We consider two cases for the mix benchmark: mix_90 (90% finds) and mix_50

(50% finds). The rest of the operations are equally split between inserts and deletes. Mix_90 captures the typical real-world case where most operations are finds while mix_50 represents a more write-heavy operation. Again, folklore is omitted from this benchmark because it does not support delete.

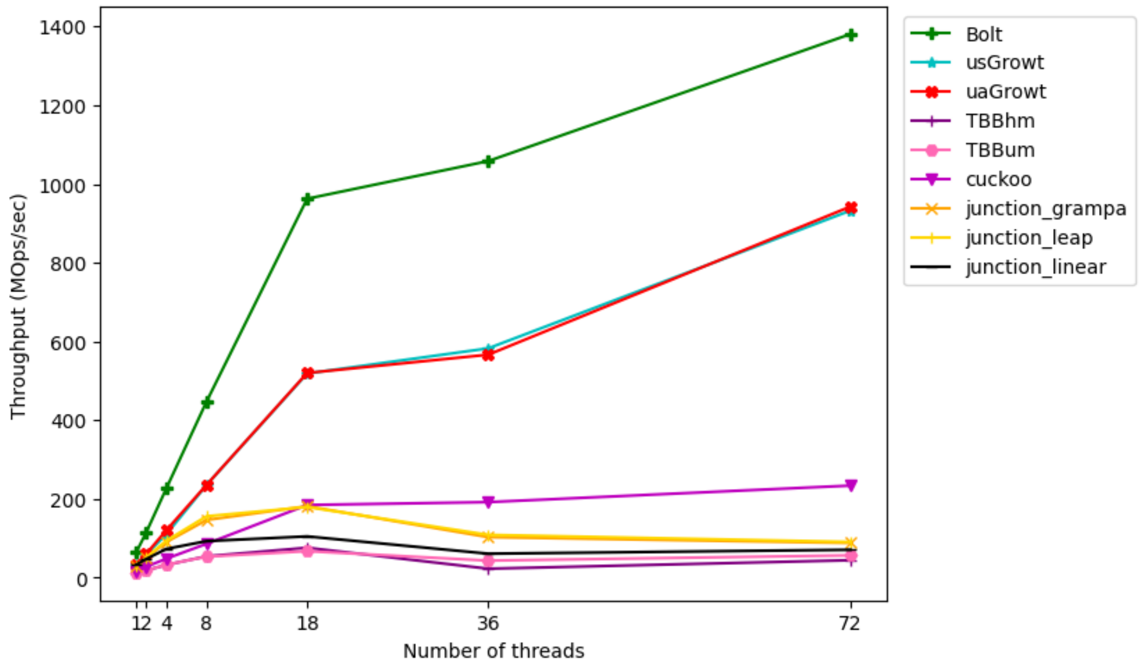

Figure 7-5: Mix Benchmark. 90% find, 5% delete, 5% insert

Results for mix_90 and mix_50 are presented in figure 7-5 and figure 7-6 respectively. As expected, the throughput is 2x less for mix_50 compared to mix_90 because find heavy workloads scale better on modern cache architectures.

Bolt is again the fastest across all core counts, achieving an absolute speedup of 30x on mix_90 and 15x on mix_50. Mix_90 scales better because most of the operations are finds, which scale well on modern multicore architectures. Bolt is faster by 1.45x and 1.75x than its closest competitor uaGrowt on mix_90 and mix_50, respectively.

Bolt is faster by at least 5.73x on mix_90 and 2.86x on mix_50 compared to other competitors besides Growt.

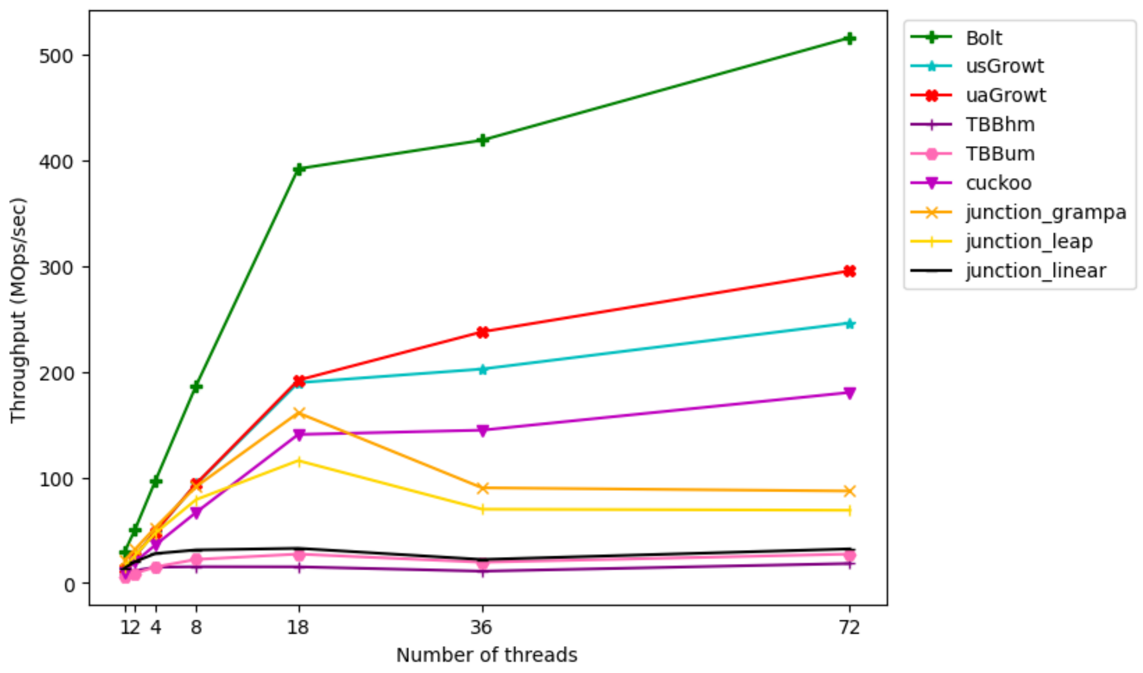

Figure 7-6: Mix Benchmark. 50% find, 25% delete, 25% insert

7.4 Contention

So far, we have considered uniform inputs. However, some real-world use cases exhibit a skewed distribution, which causes contention in a concurrent hash table. Skewed distributions are favorable for serial hash tables because of caching effects but are problematic for concurrent algorithms because of the memory cache coherency protocol. Traditionally, lock-based algorithms suffer extreme performance deterioration because a lock can only be grabbed by one thread at a time. Even though Bolt uses locks, we designed it with optimizations to handle contention.

We use the Zipf distribution for this benchmark, which is a good approximation of real-world skewness [1]. We generate 10M keys from Zipf using parameter $\theta = 0.75$ and range 10M. We prefill the table with the 10M keys. We then execute a mixed benchmark of 90% finds and 10% updates to keys that were prefilled in the table. The updates overwrite the value of the key in the table to a unique value.

The results are presented in figure 7-7. Bolt is again the fastest across all core counts and achieves an absolute speedup of 35.77x. It is 1.57x faster than the closest

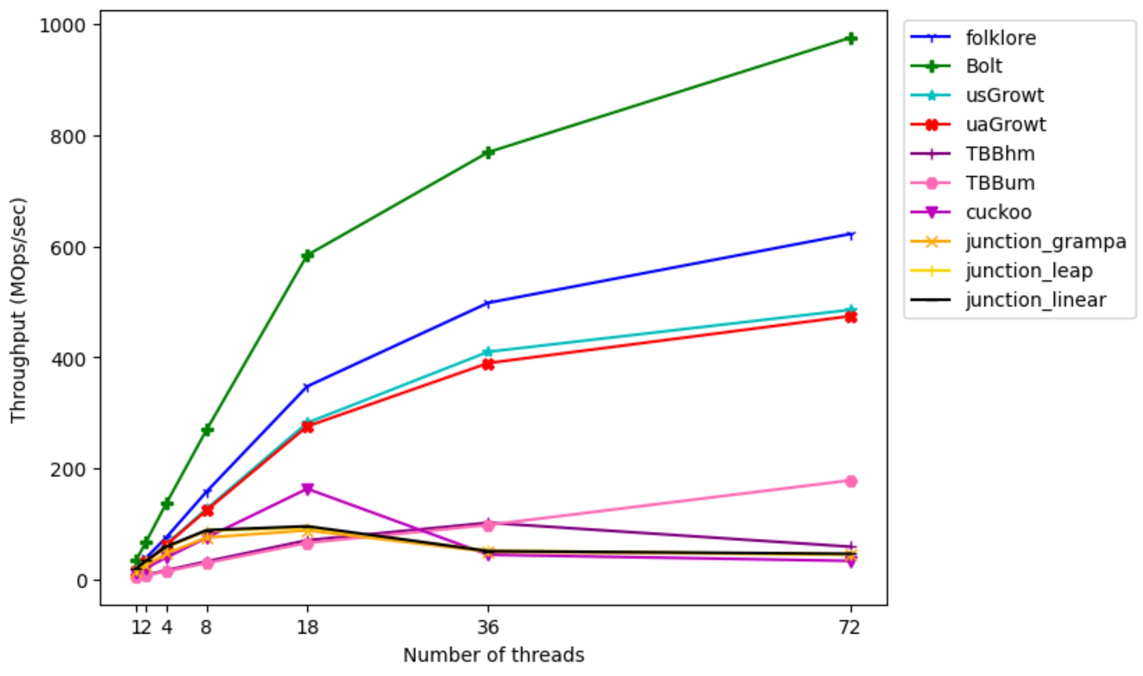

Figure 7-7: Contention Benchmark. 90% finds and 10% updates.

competitor folklore and 2x faster than the Growt variants. All other hash tables are slower by at least a factor of 5.46x. This benchmark demonstrates that Bolt fast paths for updating and finding keys are triggered with a high enough probability that contention does not cause performance deterioration.

7.5 Resize

Finally, we run a benchmark that demonstrates the performance of resize. We insert 100M uniform random keys into an empty table that has been sized for 10M keys. The results are presented in figure 7-8. Bolt is the fastest across all core counts and achieves an absolute speedup of 33x. It is 1.57x faster than the closest competitor, uaGrowt. All other tables besides Growt variants are clearly outclassed and are at least 5x slower.

The throughput for the resize benchmark is lower than the insert benchmark because resize is a heavy memory bandwidth operation, which is especially problematic at high core counts where memory bandwidth is scarce. This is because the threads will generate large amounts of memory traffic when moving the table. This benchmark also pushes the table to a high load factor before the table decides to resize, causing high probe distances, negatively impacting performance.

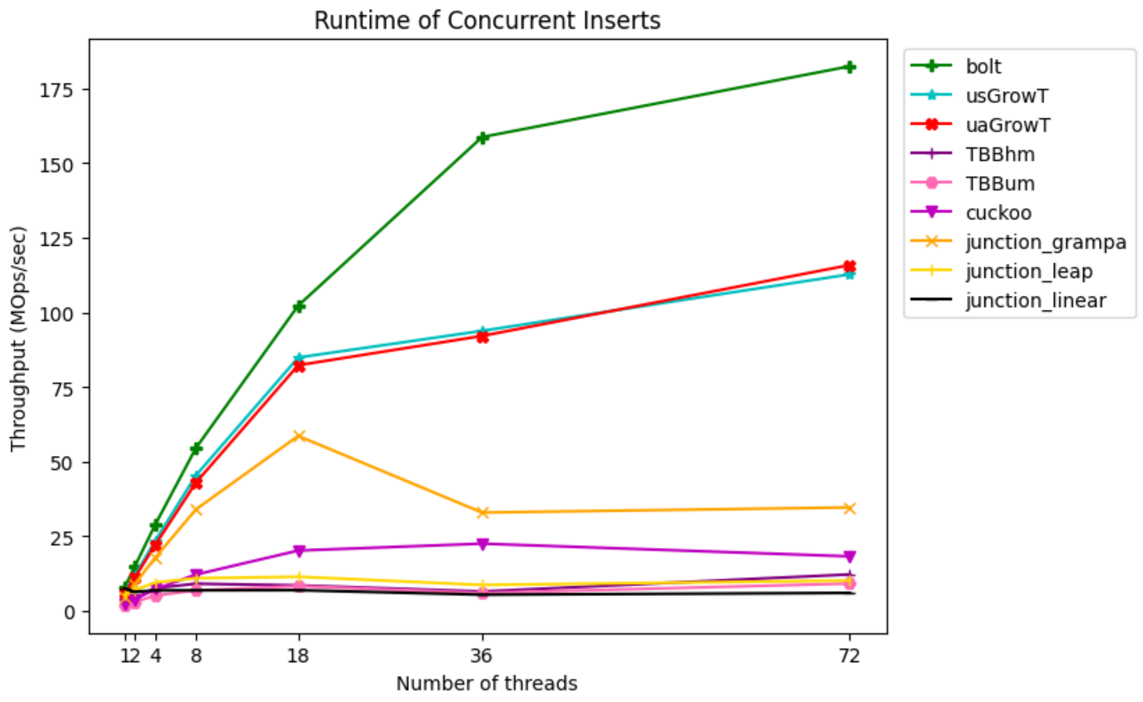

Figure 7-8: Resize Benchmark.

7.6 K-CAS Robin Hood Comparison

When we measured K-CAS Robin Hood's performance in our benchmark suite, we found it to be orders of magnitude slower than Bolt. We believe this is due to the high overheads of K-CAS and portability issues. Instead, we present a back-of-the-envelope calculation to compare Bolt with K-CAS Robin Hood based on numbers reported in their paper [5]. K-CAS Robin Hood reports results for 10% update between 20% - 40% load factor. This experiment is closest to Mix_90 in section 7.3.2. The machine used in their benchmark has double the number of cores, and similar single-core performance as our experiment machine. They achieve a throughout of 500 MOps/s at 144 threads. Adjusting for our machine, which has 72 cores, we would expect this to translate to around 250 MOps/s. Bolt achieves 1400 MOps/s in this benchmark. Based on these results, we are significantly faster. Note that K-CAS does not store values since it's a hash set, which gives it a significant advantage at high core counts because of memory bandwidth constraints.

Chapter 8

Conclusion

In this thesis, we introduced Bolt, a resizable concurrent hash table based on Robin Hood. It achieves state-of-art performance outperforming the fastest publicly available resizable and non-resizable hash tables.

We show how to support a fast path that doesn't require locks and a locked path in a scalable manner. We support resizing in the concurrent setting with minimal overhead using a heuristic to estimate how full the table is based on counting the number of displaced keys.

Future work includes expanding our fast path to cover more cases, more optimizations to deal with contention, developing a lock-free Robin Hood that is competitive to our algorithm, optimizations to make resize more efficient for a NUMA architecture, and extending our ideas to other serial hash tables that have been traditionally difficult to parallelize efficiently.

Bibliography

[1] Lada A Adamic and Bernardo A Huberman. “Zipf’s law and the Internet.” In: *Glottometrics* 3.1 (2002), pp. 143–150.

[2] Maya Arbel-Raviv and Trevor Brown. “Reuse, Don’t Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors”. In: *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*. Ed. by Andréa W. Richa. Vol. 91. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 4:1–4:16. DOI: `10.4230/LIPIcs.DISC.2017.4`. URL: `https://doi.org/10.4230/LIPIcs.DISC.2017.4`.

[3] Pedro Celis, Per-Ake Larson, and J. Ian Munro. “Robin hood hashing”. In: *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 1985. DOI: `10.1109/sfcs.1985.48`. URL: `https://doi.org/10.1109/sfcs.1985.48`.

[4] Jim Gray et al. “Quickly generating billion-record synthetic databases”. In: *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 1994, pp. 243–252.

[5] Robert Kelly, Barak A. Pearlmutter, and Phil Maguire. “Concurrent Robin Hood Hashing”. In: *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Ed. by Jiannong Cao et al. Vol. 125. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 10:1–10:16. ISBN: 978-3-95977-098-9. DOI: `10.4230/LIPIcs.OPODIS.2018.10`. URL: `http://drops.dagstuhl.de/opus/volltexte/2018/10070`.

[6] Xiaozhou Li et al. "Algorithmic improvements for fast concurrent Cuckoo hashing". In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*. 2014, nil. DOI: `10.1145/2592798.2592820`. URL: `https://doi.org/10.1145/2592798.2592820`.

[7] Tobias Maier. *growt*. 2020. URL: `https://github.com/TooBiased/growt`.

[8] Tobias Maier, Peter Sanders, and Roman Dementiev. "Concurrent Hash Tables". In: *ACM Transactions on Parallel Computing* 5.4, 16 (2019), pp. 1–32. DOI: `10.1145/3309206`. URL: `https://doi.org/10.1145/3309206`.

[9] Chuck Pheatt. "Intel threading building blocks". In: *J. Comput. Sci. Coll* 23.4 (Apr. 2008), pp. 298–298. URL: `https://dl.acm.org/doi/10.5555/1352079.1352134`.

[10] Jeff Preshing. *Junction*. 2016. URL: `https://github.com/preshing/junction`.

[11] Julian Shun and Guy E. Blelloch. "Phase-concurrent hash tables for determinism". In: *In SPAA*. 2014.

[12] Malte Skarupke. *I Wrote The Fastest Hashtable*. 2016. URL: `https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable`.