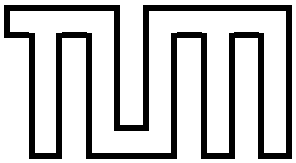


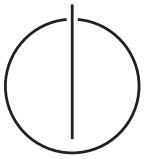
DEPARTMENT OF INFORMATICS

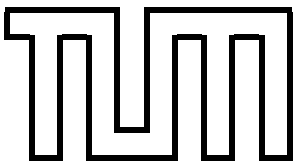
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Multi-Core Hash Tables

Adrian Riedl





DEPARTMENT OF INFORMATICS

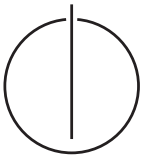
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Multi-Core Hash Tables

Effiziente Parallele Hashtabellen

Author:	Adrian Riedl
Supervisor:	Prof. Dr. Jana Giceva
Advisor:	Maximilian Bandle, M.Sc.
Submission Date:	September 15, 2020



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, September 14, 2020 Adrian Riedl

Acknowledgments

I would like to express my gratitude to my advisor Maximilian Bandle and my supervisor Prof. Dr. Jana Giceva for the opportunity to work on this interesting topic.
Furthermore, I would like to thank Maximilian for his support and time he spent with discussing the different approaches, giving hints, checking the results and their interpretation, and everything he did for me.
Many thanks to Johannes for proof-reading this thesis, the years of friendship, and the funny times inside and outside of the lecture halls.
I would like to thank many people, who supported me in my spare time – without you, this thesis would have turned out to be more difficult.

Lastly, thanks to my great parents, who supported me over all the years and without whom I would not be, where I am right now.

Abstract

Hash Tables play an important role in many modern applications as they offer several advantages like fast and efficient lookups for keys or constant time complexity regardless of the input size.
Not directly linked to hash tables, but an important way for speeding up data processing algorithms is thread parallelism on multi-core hardware.
This thesis presents and investigates the symbiosis of both different hashing schemes and multi-core hardware. For the Chained Hashing and Open Addressed Hashing Schemes, we present different synchronization strategies that allow concurrent access to the different hash tables. Therefore, we show how the implementation and the algorithms have to be adapted by adding necessary fields for reserving a bucket exclusively. Moreover, we mention some optimizations how to improve the lookup throughput.
With thread parallelism, work partitioning comes into play, which is looked at closely in our experiments. The analysis presents a hierarchy of the different strategies to find the best suited for a specific hash table. Furthermore, we compare the insert and lookup performance of the different hash tables for varying workloads to determine which ones are the best. We conclude with a comparison of already existing parallel hash table implementations.
Finally, we summarize our results, stating the hash tables with the highest performance, and providing a guideline about using specific hashing methods.

Kurzfassung

Hashtabellen sind ein wichtiger Bestandteil in vielen modernen Anwendungen, da sie zahlreiche Vorteile bieten, wie schnelles und effizientes Nachschlagen der Schlüssel oder konstante Laufzeitkomplexität, unabhängig von der Eingabegröße. Nicht direkt in Verbindung stehend mit Hashtabellen, jedoch eine andere Möglichkeit, Algorithmen zur Datenverarbeitung zu beschleunigen, ist das Ausnutzen von Parallelität auf Multi-Core-Prozessoren.
Diese Arbeit präsentiert und untersucht die Symbiose sowohl von unterschiedlichen Hashing Schemata als auch von Multi-Core Hardware. Für Chained Hashing (verkettete Listen) und Open Addresses Hashing (offene Adressierung) Schemata präsentieren wir verschiedene Strategien zur Synchronisation, welche gleichzeitigen Zugriff auf die unterschiedlichen Hashtabellen zulassen. Dafür zeigen wir, wie die Konstruktion der Hashtabelle und die Algorithmen angepasst werden müssen, indem zusätzliche Felder zum Reservieren eines Buckets hinzugefügt werden. Weiterhin erwähnen wir Optimierungen, die die Lesegeschwindigkeit verbessern sollen.
Aufgrund der Parallelität spielt das Partitionieren der Arbeit eine Rolle, die wir in unseren Experimenten genauer betrachten. Die Analyse präsentiert eine Hierarchie von unterschiedlichen Synchronisationsverfahren, um die beste für eine bestimmte Hashtabelle zu finden. Weiterhin vergleichen wir die Leistungsfähigkeit für das Einfügen und Nachschlagen, um für wechselnde Auslastungen die besten zu finden. Wir beenden die Analyse mit einem Vergleich mit bereits existierenden Implementierungen für parallele Hashtabellen.
Zum Schluss fassen wir unsere Ergebnisse zusammen und führen einen Leitfaden an, um die beste Hashtabelle zu nutzen.

Contents

1. Introduction

Hash tables are essential and expedient data structures which are required by many different applications. They are used as index structures in database management systems to look up data for a particular key efficiently, as data structures to calculate aggregation functions, or to perform a hash join in SQL queries.
Hash tables are available in many different programming languages, like `std::unordered_map` in C++ STL, `java.util.Hashtable` in Java, and also a lot of libraries offer their own implementations, such as `boost::intrusive::hashtable` in the Boost library[1]. Despite these various implementations for such data structures, the hash tables must perform the best possible way on modern hardware, so hashing and hash tables should not be a black box.
Regarding the newest hardware, we have to consider the development of the microprocessor industry. Since the end of Dennard's scaling in 2004 and the slowdown of Moore's law, there was a switch to increase the core count instead of a single processor's efficiency and performance. With the single-threaded performance stagnating, parallel execution of several tasks needs to be investigated in detail for many different applications and data structures.

1.1. Motivation

With the end of Dennard's scaling, the microprocessor industry started to increase the core count of a single processor instead of a single CPU's performance. Thus, single-threaded hash tables soon become a significant bottleneck, so both hashing and the utilization of hash tables need to be looked at very closely from the multi-threaded perspective to get the best performance.
As only a few research papers can be found that make hash tables on multi-core machines a subject of discussion, we want to provide an insight into this topic and contribute some ideas to it.
Parallel hash tables need to be sophisticated to guarantee correctness as well as high performance for all operations. As parallelism can easily cause mistakes, the different approaches have to be implemented carefully.
This thesis presents a detailed look at three different hashing schemes with several different synchronization strategies that allow concurrent access for insertion. These strategies are combined with two hash functions, four different data distributions, varying numbers of threads, and changing ratios regarding un/successful lookups. With these parameters that we can change, we try to simulate several cases in which hash tables might be used.
For the evaluation, we compare the different locking strategies of each hash table and the optimization we announce for the hash tables. Moreover, we look, how different hash tables perform for changing workloads and hardware settings.

[1] `https://www.boost.org` (accessed September 12, 2020)

Our experiments clearly state that picking the right combination of locking strategy and optimization may have an arbitrary influence on inserts and lookups.

1.2. State of the Art

In 2015, Richter et al. proposed in their work [RAD15] an analysis of different hash tables and hash functions regarding the change of data distributions, load factors, dataset sizes, and un/successful lookup ratio. At the end of their experiments, they suggest a decision graph based on the results they achieved that can be used to determine which hash table and hashing scheme fits best for a specific workload. However, they explicitly state in their paper to rely on single-threaded execution.
Van der Vegt et al. present a parallel compact hash table in their research paper [VL11]. Their hash table uses dynamic region-based locking that is a lockless and efficient mechanism to parallelize Cleary compact hash tables and order-preserving bidirectional linear probing hash tables. By avoiding operating system locks, they promise to be able to provide the performance required for systems designed for high throughput.
Laarmann et al. analyze in detail in [LPW10], that hash tables have many different points that need to be considered. They list several sensitive performance parameters, like bucket size, cache line size, data layout, probing sequence, etc., whose impact is explicitly considered and finally, they compare the resulting speedup.
Finally, Cliff Click presented a lock-free hash table in his talk on JavaOne in 2007 [Cli07]. With his implementation, he could get high-performance numbers for many cores compared with other concurrent hash tables. For his hash table, he used the Compare-and-Swap concept, a commonly occurring pattern in concurrent programming.

1.3. Structure

The outline of this thesis is as follows. Chapter 2 presents the foundations which are required for this thesis. Therefore, Sect. 2.1 repeats the hashing schemes considered throughout the work with an example for each hash table. The used hash functions are explained briefly in Sect. 2.2, while Sect. 2.3 presents the first idea of how to improve the performance of the data structures when dealing with large amounts of data.
In Chapter 3, we list the considerations which we made for each hash table. In general, it starts with the question, how to deal with unsuccessful lookups, in Sect. 3.1. After answering this question, we start with the Chained Hashing in Sect. 3.2. For this hash table, ideas how to synchronize the access are shown in Sect. 3.2.1, while Sect. 3.2.2 focuses on possible optimizations for lookups. Sect. 3.3 continues with Open Addressed Hashing. First, we list our thoughts about Linear Probing Hashing in Sect. 3.3.1, followed by Robin Hood Hashing in Sect. 3.3.2. Similar to the Chained Hashing, Sect. 3.3.3 presents an idea of a possible optimization for Open Addressed Hashing.
Next, Chapter 4 shows the evaluation of our implementations. In Sect. 4.1, we explain the experimental setup that needs to be known to comprehend the experiments. Sect. 4.2 shows the results we got for the various experiments. At first, we analyze the hash functions in Sect. 4.2.1, followed by Sect. 4.2.2 about partitioning the workload. Before we go deeper into dealing with parallelism, we show an efficient way to return lookup results in Sect. 4.2.3, which

we use mostly in the experiments. After this, we take a look at the different synchronization strategies in Sect. 4.2.4. Based on previous results, we move on to selected hash tables and compare their performance for the insertion, as Sect. 4.2.5 presents. The lookup performance is shown in Sect. 4.2.6. To finish this chapter, we also compare our implementation with already existing parallel hash tables in Sect. 4.2.7.
At the end of this thesis, the outcome of the experiments is presented in Chapter 5. Therefore, we summarize what we could figure out and present the facts split up in two parts; the first is in Sect. 5.1 and describes everything that applies for all hash tables, while Sect. 5.2 illustrates which hash table suits best for which workload. This is also summarized in a decision tree. Lastly, Sect. 5.3 presents some ideas of future work to be done on the basis of this project.

2. Foundations

Hashing is a useful concept in several applications, as it is well suited for point accesses and thus allows quick lookups and insertions. To realize this concept, we rely on hash functions that are used for distributing the data equally. The aim of these functions is that there are hardly any collisions between two different keys. Not only the way to calculate the hash is relevant, but also how the data is stored to allow fast access to it again. Despite the aim of hash functions, collisions are unavoidable and thus, the hash tables have to provide a way of dealing with collisions. These solutions might interfere with the aim of constant access times. In this chapter, we describe the basic hashing schemes, which we will look at in this thesis, in Sect. 2.1, before we continue with an overview of the hash functions in Sect. 2.2. Finally, Sect. 2.3 introduces a way of work balancing for parallel execution models, which was originally presented for another environment but can be adapted for the actual use case.

2.1. Hashing Schemes

In this thesis, we analyze three different hashing schemes:

1. Chained Hashing
2. Linear Probing Hashing
3. Robin Hood Hashing on an Open Addressed Scheme

For each hashing scheme, we present an exemplary representation, as well as pseudo-code for inserting key-value pairs $\langle k, v \rangle$ and looking up a key k.
Hereby, we explain the hash tables, which we focus on and how these are constructed in our project. However, we do not yet include any parallelism or specific adaptions for multi-core machines.

2.1.1. Chained Hashing

Chained Hashing, introduced by Luhn in 1953 and analyzed by Knott and de la Torre in 1987 [KT89], is a straightforward solution for resolving collisions, as the elements mapping to the same bucket of the hash table are stored in a linked list. Fig. 2.1 shows an example for this hashing scheme – a $\bullet$ indicates a pointer to the next entry, whereas $\perp$ represents a `nullptr`.
For storing the data, the hash table is an array of pointers, which are either a `nullptr` indicating that this bucket is empty or a pointer to an entry of the hash table. The entries include key k and value v, as well as a pointer p to the next entry of the linked list. For lookups, the correct bucket has to be figured out and the list needs to be traversed, checking for a match until reaching the end.
In general, Chained Hashing is very easy to implement and to use. However, it has a significant

drawback, namely causing several cache misses when traversing the linked list, as most pointer indirections are random memory accesses. Loading the data from memory to cache leads to stalling the processor and wasting computational power.

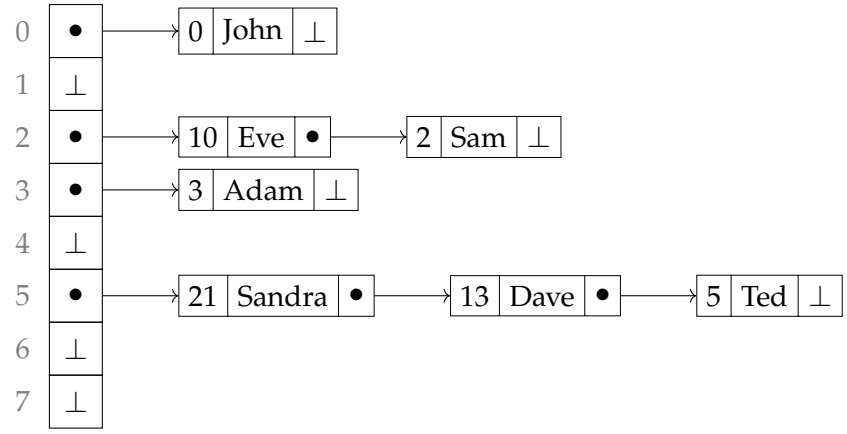


Figure 2.1.: Representation of Chained Hashing for inserting key-value pairs in ascending order of the keys with the hash function $h(k) = k \bmod 8$. The number on the left of the bucket is its index.

Insertion. Algo. 2.1 shows how a new key-value pair $\langle k, v \rangle$ is inserted into the Chained Hash Table HT with $buckets_{HT}$ buckets. The number of elements inserted into the Chained Hash Table is not limited, so an arbitrary number of elements can be stored.

Algorithm 2.1
Chained Hashing – Insertion

```
position ← h(k) mod buckets_HT
oldPointer ← HT[position]
newPointer ← Entry(⟨k, v⟩, oldPointer)
HT[position] ← newPointer
```

With this insertion algorithm, the elements which map to the same bucket are linked via the pointers. As the newest element is placed at the front of the linked list, the end of the list is indicated with a `nullptr`.

Lookup. For the lookup of a certain key k in the hash table, the same steps for determining the bucket are taken. Getting the pointer to the first entry of this bucket, the iteration over the list starts, until either the key is found or the end of the list is reached. For the first case, the iteration ends by returning the corresponding value v of this entry. In the second case, a way has to be found to indicate that there was no match, for example by throwing an exception or returning a specially defined value. The exact procedure is described in Algo. 2.2.

Algorithm 2.2
Chained Hashing – Lookup

```
position ← h(k) mod buckets_HT
entry ← HT[position]
while entry ≠ nullptr do
    if entry.key = k then return entry.value
    entry ← entry.next
throw NoMatchFound
```

2.1.2. Open Addressed Hashing

2.1.2.1. Linear Probing Hashing

In 1954, Gene Amdahl, Elaine McGraw, and Arthur Samuel came up with Open Addressed scheme based hash tables which Linear Probing belongs to. This was analyzed by Donald E. Knuth [Knu63].
Using this hashing scheme, the hash table contains one key-value pair per bucket. To resolve hash collisions, the next free bucket is searched where the pair is then stored. Lookups are performed in the same manner.
An example is shown in Fig. 2.2.

0	1	2	3	4	5	6	7
0 John		2 Eve	3 Adam	10 Sam	5 Sandra	13 Dave	21 Ted

Figure 2.2.: Representation of Linear Probing for inserting key-value pairs in ascending order of the keys with the hash function $h(k) = k \bmod 8$. The number above the bucket is its index.

Insertion. After calculating the position in the hash table with $buckets_{HT}$ buckets, the bucket is checked if it is already taken. If this is the case, the position is incremented by 1 w.r.t. the size of the hash table and the corresponding bucket is rechecked. This continues until a free bucket is found. Inserting an element into an already full Linear Probing Hash Table leads to an infinite loop, as no free bucket can be found. The insertion of a key-value pair $\langle k, v \rangle$ into a Linear Probing Hash Table is expounded in Algo. 2.3. This algorithm assumes never to be called on a full hash table, thus expects to find a free slot in the hash table. The number of elements is limited to the hash table's size, so the load factor can never be bigger than 1.

Algorithm 2.3
Linear Probing – Insertion

```
position ← h(k) mod buckets_HT
bucket ← HT[position]                          ▷ Initial bucket
while bucket taken do
    position ← (position + 1) mod buckets_HT
    bucket ← HT[position]
bucket ← ⟨k, v⟩
```

Linear Probing Hashing can be implemented very efficiently and thus allows fast execution, additional to very good cache locality, as the scanning for free buckets is done sequentially.

Lookup. Looking up keys in the Linear Probing Hash Table works quite similar to inserting. The bucket, where the data is assumed to be stored, is determined and it is checked if the key matches. If there is a mismatch, the position is incremented w.r.t. the size of the hash table and the key is checked again. This continues until either a match is found, a free bucket is detected, or, the initial bucket, from which the checking has begun, is reached again. For the first case, the search is finished by returning the value of this bucket, whereas the last two cases lead to a notification about the mismatch.
Algo. 2.4 shows a simplified version of the algorithm, assuming that the hash table is never full, so that there is always a free slot, thus leaving out the case of hitting the initial bucket again.

Algorithm 2.4
Linear Probing – Lookup

```
position ← h(k) mod buckets_HT
bucket ← HT[position]                          ▷ Initial bucket
while bucket taken do
    if bucket.key = k then return bucket.value
    position ← (position + 1) mod buckets_HT
    bucket ← HT[position]
throw NoMatchFound
```

2.1.2.2. Robin Hood Hashing on Linear Probing

Robin Hood Hashing was announced by Celis et al. in 1985. In their preliminary report [CLM85], they present promising numbers w.r.t. deletions and successful lookups appearing to require constant time, while insertions and unsuccessful searches can be performed in logarithmic time. An example is shown in Fig. 2.3. This thesis focuses on Robin Hood Hashing on Open Addressed Hash Tables, here, in particular, the Linear Probing Hash Table, by adapting the insertion algorithm.

0	1	2	3	4	5	6	7
0 John		2 Eve	10 Sam	3 Adam	5 Sandra	13 Dave	21 Ted

Figure 2.3.: Representation of Robin Hood Hashing on Linear Probing for inserting key-value pairs in ascending order of the keys with the hash function $h(k) = k \bmod 8$. The number above the bucket is its index.

Insertion. The difference to Linear Probing Hashing, which was explained in Sect. 2.1.2.1, is to consider the distance of the newly inserted element to the original slot it should be placed in. Let $d(\langle k, v\rangle)$ be the distance of pair $\langle k, v\rangle$ from its original bucket. When probing the sequence during insertion of $\langle k_{new}, v_{new}\rangle$, the distance of the pair $\langle k_{stored}, v_{stored}\rangle$ – if the bucket contains one – is compared with the distance of the new pair. If a pair is encountered, which fulfills $d(\langle k_{new}, v_{new}\rangle) > d(\langle k_{stored}, v_{stored}\rangle)$, the new pair is placed in this bucket, while the search for an empty slot continues for the replaced pair. The insertion of a key-value pair $\langle k, v\rangle$ into a Robin Hood Hash Table on Linear Probing with $buckets_{HT}$ buckets is displayed in Algo. 2.5. This algorithm assumes never to be called on a full hash table, thus expects to find a free slot in the hash table. The number of elements is limited to the size of the hash table, so the load factor can never be bigger than 1.

Algorithm 2.5
Robin Hood on Linear Probing – Insertion

```
position ← h(k) mod buckets_HT
distance ← 0
bucket ← HT[position]                                   ▷ Initial bucket
while bucket taken do
    if distance > d(⟨bucket.key, bucket.value⟩) then
        ⟨k_previous, v_previous⟩ ← ⟨bucket.key, bucket.value⟩
        bucket ← ⟨k, v⟩
        distance ← d(⟨k_previous, v_previous⟩)
        ⟨k, v⟩ ← ⟨k_previous, v_previous⟩               ▷ Continue with replaced data
    position ← (position + 1) mod buckets_HT
    bucket ← HT[position]
    distance ← distance + 1
bucket ← ⟨k, v⟩
```

Lookup. The algorithm for the lookup in a Robin Hood Hash Table on Linear Probing does not differ from the Algo. 2.4, although Celis et al. pronounced a smart search algorithm in their paper. Their strategy would be to start the search at the bucket with the average displacement from its perfect slot. The search is then performed in a bidirectional manner.
Another idea would be to make use of the distance of the probed element, compared with the iteration number, and to abort early.
However, both ideas result in additional computational and internal overhead, which needs further investigation to make them more performant when dealing with parallelism.

2.2. Hash Functions

This section presents the hash functions used within the experiments.
We want to investigate if there is a big difference between the hash functions regarding key distributions. Therefore, we choose two hash functions, which are widely used in practice: (1) Cyclic Redundancy Codes (CRCs) as one engineered hash function and (2) Murmur hash. Both are implemented for hashing 64-bit integers.

2.2.1. Cyclic Redundancy Codes

Cyclic Redundancy Codes (CRCs) are mainly used in computer networks and data storage devices to provide cheap and effective error detection capabilities [Koo02; KC04]. Error detection codes need to be simple, cheap, fast, and robust to calculate, as the amount of data transferred and stored increases over the time.
Although CRCs are not explicitly designed for the use as a hash function, they work on any input data the same way with a promising throughput, as Gad et al. analyzed for the 32-bit ethernet protocol [GGN15]. This hash function is not limited by the datatype of the input but can process any data. Thus, this function is more complex than other hash functions which are designed for a certain input datatype.
So, the implementation takes a pointer to a buffer containing the data whose hash needs to be calculated and the size of the buffer in bytes.
With this information, the implementation combines the data in the buffer with a combination of shifts, additions, multiplications, and logical ANDs, ORs, and XORs, depending on the data's remaining size.

2.2.2. Murmur

The Murmur hash is a relatively new, but very common hash function used in many practical applications, like DuckDB [RM19].
Until now there is no formal analysis on this and thus no improvements to it, so the implementation is used as announced by A. Appleby [App]. Limiting to 64-bit keys in our analysis, Murmur3's 64-bit finalizer is used as shown below.

```
uint64_t murmur3_64(uint64_t key) {
    key ^= key >> 33;
    key *= 0xff51afd7ed558ccd;
    key ^= key >> 33;
    key *= 0xc4ceb9fe1a85ec53;
    key ^= key >> 33;
    return key;
}
```

2.3. Work Partitioning

When dealing with parallelism, the work needs to be distributed among individual workers. Therefore, work partitioning becomes relevant, as the challenge here is to get the best possible tradeoff between too small and too big tasks. When the work is too small, there is a large overhead, for example, due to communication, while for too much work, efficient load balancing is quite difficult.
The implementation partitions the workload for the tasks in two different ways:

Morsel-Driven Parallelism. Leis et al. announced in [Lei+14] a Morsel-Driven query execution model for database management systems. With this, the input data is split into constant-sized work units, called morsels, which a dispatcher then assigns to worker threads. The same idea can be implemented for building hash tables, either by handing over a list of key-value pairs for insertion or storing the pairs. Before starting the read phase, the hash table is built. Once all data is known, the threads are started, synchronized with an `atomic` counter, getting the next free morsel until there are no morsels left anymore. When processing a morsel, the thread iterates over its assigned morsel. After finishing this, it gets the next morsel until none is left anymore. The thread terminates, while the main thread waits for all other threads to finish their assigned work.

Tuple-Driven Parallelism. Similar to the Morsel-Driven Parallelism, we gather the data before building the hash table. Also, synchronization between individual threads is done by an `atomic` counter, which indicates the tuple's index to be inserted into the hash table. This method is simpler to implement, but has the following disadvantage: the counter is a hot spot in the code, as it is incremented for each tuple. Thus, all threads might want to access the counter simultaneously, which should not happen, leading to massive communication and synchronization overhead.

3. Exploiting Hash Tables on Modern Hardware

In this section, we start dealing with parallelism and present our approaches of synchronizing access to the hash tables.
There are two kinds of parallelism: One is Data-Level parallelism for which many data items are processed at the same time – the other one is Task-Level parallelism, which means that different tasks operate independently and in parallel. In our thesis, we focus on Thread-Level parallelism, which makes use of Task-Level parallelism by using hardware threads that work in parallel.
The major challenge of multi-threading is to allow the highest possible degree of concurrency to get the best performance improvement by reducing the synchronization to a minimum. Additional to this, we have to maintain the correctness of the outcome as the order of reads and writes is non-deterministic.
Moreover, the work is partitioned in parallel tasks, which are mapped to individual threads. With this model, task granularity plays an important role, as too little task sizes lead to a higher overhead, whereas too big sizes cause difficulties for efficient load balancing, as the work might not be distributed equally.
Furthermore, a Write-Once-Read-Many workload (WORM) is assumed, which means that at first, all elements are inserted into the hash table. After all insertions succeeded, the reading phase starts. Once this phase has started, no insertions are allowed anymore.
The data type of the keys is referred to as `KeyType`, the data type for the values is named `ValueType`.

3.1. Returning Lookup Results

Looking up a key in a hash table has two possible outcomes, either the key is found and the associated value can be returned, or the key is not found and no value can be given back. The first case, finding the key, is simple to implement – we return the value.
For the second case, we would need a value for every type which is returned if there is a mismatch. However, this is not possible for every type.
Thus, we consider two options in this project:

Throwing a `std::exception`. In C++, custom exceptions can be designed and used. Mostly, they are used to report both runtime errors and logical errors.
By designing an exception with the name `NoMatchFound`, the program has a simple solution for indicating a missing match for this key. Therefore, the exception is thrown if the key cannot be found in the hash table. The algorithms 2.2 and 2.4 have already presented this solution.

Returning a `std::optional`. Another possibility is to set a bit in the return value if the key cannot be found. As this is not possible for every datatype, an artificial bit is added by returning a pair with a `boolean` and the `ValueType`. If we find a match, the `boolean` is `true` and the value in the second part of the pair can be used. For a mismatch, the `boolean` is false and we ignore the value in the second part.
This solution can be implemented in many different ways; however, our implementation uses the C++ `optional` class, which offers the desired functionality to objects of this class. When an `optional` object is created, the `boolean` is false, until a value is assigned to the object.
The algorithms 2.2 and 2.4 for the lookups in the different hash tables can easily be changed to this approach by replacing the throwing of the exception with a return of the `std::optional` instance.

3.2. Chained Hashing

In the first part of this section, we present different ways how to synchronize Chained Hash Tables. Moreover, we also show what optimizations can be applied to this hash table to gain performance.
Analyzing the most straightforward implementation of the Chained Hash Table, we assume that the hash table is an array of pointers, each of which points to the head of a linked list as displayed in Fig. 2.1.

3.2.1. Synchronization

In order to allow multiple threads to work on the hash table, each bucket needs to have the possibility to be locked to guarantee exclusive access to it.

3.2.1.1. Locking with `std::mutex`

The simplest solution allowing a certain degree of parallelism would be to protect each bucket with an instance of `std::mutex`. This implementation allows us to de-schedule the waiting thread until the lock becomes free. However, this strategy has two significant drawbacks: (1) high cost, as two context switches are needed, one to put the thread to sleep and the second to wake the thread up (12–20μs) and (2) memory consuming size of the mutex, as the standard mutex can be up to 40–80 bytes in size. A mutex needs a flag indicating if it is locked and a queue that stores the waiting threads which are blocked, allowing the mutex to put the threads to sleep and to wake them again. This explains the memory consumption of the mutex.
Due to the mentioned disadvantages above, the locks – supported by the OS – are not considered in detail.

3.2.1.2. Locking with `std::atomic_flag`

Using this method of synchronization, each pointer from the hash table is associated with an instance of `std::atomic_flag` (later on referred to as: flag). The C++ class offers the function `test_and_set` (TAS), which sets the flag to `true` if it was not set before, otherwise it doesn't change anything. Most importantly, the value, the flag held before, is returned, indicating if the flag has been set with the call.

Before accessing a bucket via its pointer, the flag needs to be set by this thread. If it is not possible, as another thread has already taken the flag, the requesting one has to spin. The waiting thread repeatedly polls the lock until it is free. To unlock the bucket, the flag is cleared by the thread holding the exclusive access to it.
Spinlocks have the advantage that only a few cache miss penalties are caused, however the thread burns CPU cycles while it is waiting for the flag to be released.
How this can be implemented, is shown in Algo. 3.1.

Algorithm 3.1
Chained Hashing – Locking bucket via `std::atomic_flag`

```
position ← h(k) mod buckets_HT
while HT[position].flag.TAS() do
    {}                                  ▷ Bucket taken, retry
perform workload                        ▷ Bucket locked exclusively
HT[position].flag.clear()               ▷ Unlock bucket
```

3.2.1.3. Locking with Pointer Smashing

Running on a 64-bit operating system, pointers are 64 bits long, while the upper 16 bits are always 0. These bits can be used for encoding different information. When analyzing the memory consumption of the locking strategy mentioned earlier, each bucket needs to be associated with an additional flag.
As the information about the bucket's locking status can be encoded using only one bit, the unused part of the pointer to the linked list can be exploited.
If a pointer is locked, the requesting thread polls the pointer to get access to the bucket, so this strategy also turns out to be a spinlock.
In this project, the main functionality depends on a `std::atomic::exchange` operation (AE), which returns the original value the exchanged variable held before. Thereby, a globally predefined mask, whose 60^{th} bit is set to 1, is exchanged with the value of the element of the hash table array. This mask represents an invalid pointer, so it cannot be caused by any memory allocation, of course, any value that is no valid pointer can be chosen. Moreover, benefiting from memory alignments of modern processors, one could also decide to use a value whose lowest bit is set to 1 and the others 0. After the swap, the returned value of the AE operation is compared with the predefined mask. If it is the same, another thread has already locked the bucket and the requester retries; if not, the requester has exclusive access and the address of the first entry in the linked list.
To unlock the bucket again, a valid pointer is stored in the element in the hash tables.
Algo. 3.2 shows the procedure to lock a bucket via the pointer.

Algorithm 3.2
Chained Hashing – Locking bucket via Pointer Smashing

```
position ← h(k) mod buckets_HT
originalValue ← 0
mask ← 1 ≪ 60
while (originalValue ← AE(HT[position], mask)) = mask do
    {}                                          ▷ Bucket taken, retry
perform workload                                ▷ Bucket locked exclusively
store(HT[position], valid pointer)              ▷ Unlock bucket
```

3.2.1.4. Pointer Exchanging with `std::atomic::exchange`

Both synchronization strategies explained in 3.2.1.2 and 3.2.1.3 lead to spinning threads if the bucket they try to access is taken. Hence, we need two accesses to the synchronization structure, one for locking and the other one to unlock.
With announcing our last synchronization strategy, we only require one access to the synchronization structure. Therefore, when adding an element to the bucket, the thread first gets the memory position of the new entry, which it needs to store the key-value pair. After knowing the new element's address, it is swapped with the corresponding bucket's pointer, returning the previously-stored pointer using an AE operation. Thus, the bucket now points to the newly added element. To maintain the list, the thread places the returned pointer in the added element.
Algo. 3.3 displays how this can be implemented.

Algorithm 3.3
Chained Hashing – Pointer Exchanging with `std::atomic::exchange`

```
position ← h(k) mod buckets_HT
entry ← Entry(⟨k, v⟩)                           ▷ Placing data
oldHead ← AE(HT[position], entry)               ▷ Exchanging of the pointer
entry.setNext(oldHead)                          ▷ Maintaining linked list
```

The only drawback of this solution is that the list is incomplete between the exchange operation of the pointer and the maintaining of the list. So for Read-Write workloads being a mix of insertions, updates, deletions, and lookups, this solution needs careful evaluation.
The exact procedure of Pointer Exchanging is shown in Fig. 3.1. For visualization, we insert the key-value pair ⟨21, Sandra⟩ into an existing hash table with the hash function $h(k) = k \ mod \ 8$. Part a) shows the hash table before the insertion is executed, b) how the thread places the data, c) the exchange of the pointer in the hash table, and d) the maintaining of the list. The steps from Alg. 3.3 have the same descriptions as the parts of the visualization.

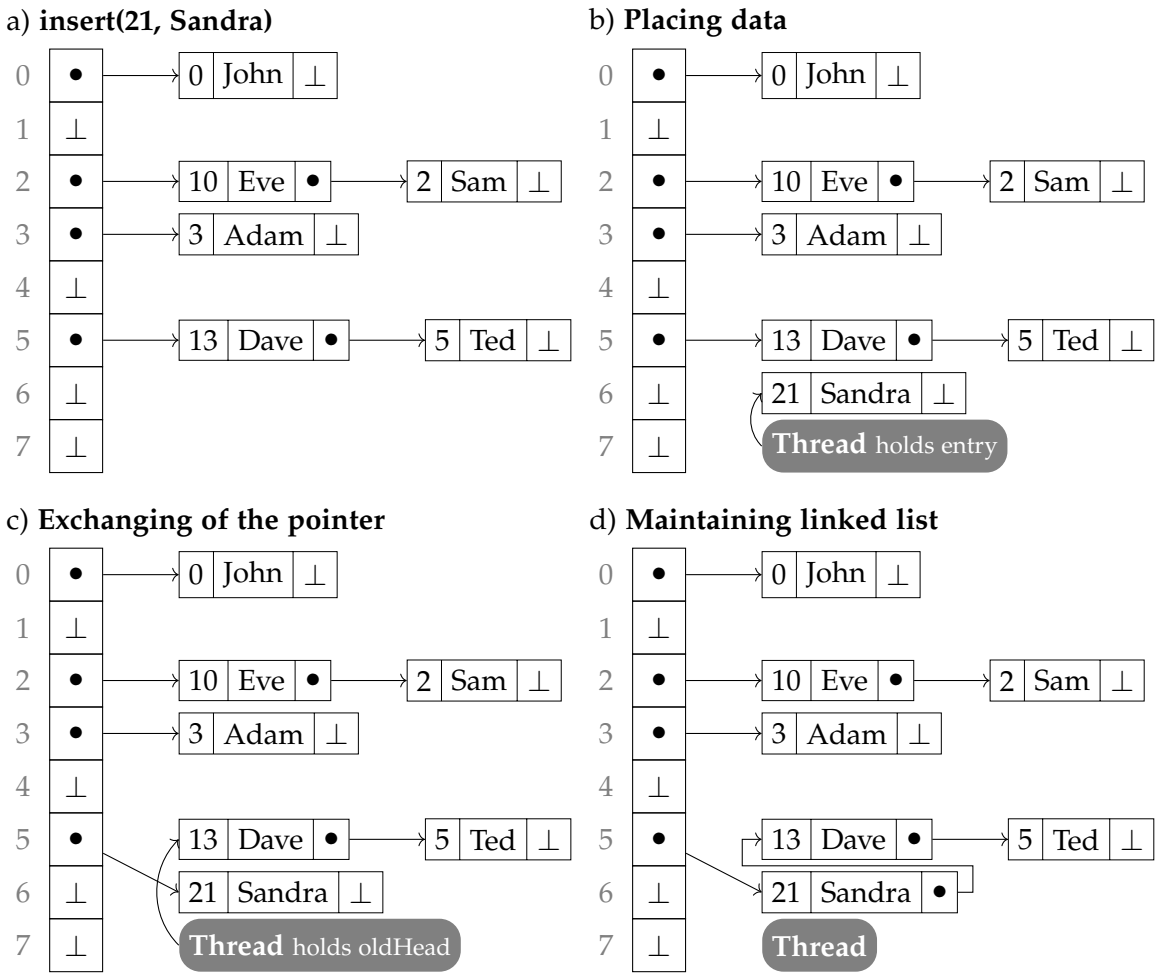


Figure 3.1.: Visualization of Algo. 3.3. The names of the parts of the pictures are the same as the lines of the algorithm.

3.2.2. Optimizations

The memory subsystem of modern hardware is designed for sequential access to data, while pointer chasing causes stalls of the CPU if the data does not fit into the cache [San+19, p.130]. As this problem arises when looking up in the hash table, there are two possibilities to make the lookup more performant: (1) by giving the CPU hints, which entry will be needed in the next iteration and (2) to narrow down the number of keys for which the pointer chasing is performed. The first improvement can be realized by prefetching the needed entries, the second by tagging the pointers to the linked lists.

3.2.2.1. Next Entry Prefetching

Software prefetching is used to hide cache miss latencies. Therefore, the source code is modified by using compiler intrinsics on any pointer in the program.
With the instruction `__mm_prefetch`, the CPU is aware that the data at this memory position will be needed and already loads it into the cache beforehand.

Algo. 3.4 shows the extension of Algo. 2.2 for prefetching.

Algorithm 3.4
Chained Hashing – Lookup with Prefetching

```
position ← h(k) mod buckets_HT
entry ← HT[position]
while entry ≠ nullptr do
    if entry.key = k then return entry.value
    entry ← entry.next
    __mm_prefetch(entry)
throw NoMatchFound
```

3.2.2.2. Pointer Tagging

As already mentioned in Sect. 3.2.1.3, on 64-bit machines, the upper 16 bits of a pointer are set to 0. These bits cannot only be used for locking this bucket but also for storing additional information about the entries of the linked list.
When inserting an element into the hash table of the size 2^n, the lowest n bits are considered. Without loss of generality, let n be smaller than 32. Thus, the upper 32 bits are not crucial for calculating the position in the hash table.
As the hashes of the keys mapping to this position share the same lower n bits, which are already determined by the bucket's index, they might differ in the upper part of the hash. With this knowledge, a bitmask can be constructed in the upper 16 bits of the pointer in each hash table bucket during the insertion. Therefore, $log_2(16) = 4$ bits are chosen from the hash to calculate the bit of the upper part of the pointer set to 1. In our case, we choose the lowest four bits of the upper 32 bits of the hash, so the bits 32 to 35.
If we chose to place the four bits in the n bits of the hash, we would not benefit from this, as all n bits for all hashes to this bucket are the same. This results in a bitmask, which has just one bit set to 1. Furthermore, by doing this, the idea of Pointer Tagging does not pay off, as all keys which hash to this bucket, independent of it being in the hash table or not, will cause a scan of the linked list.
So the bits used for building the mask for Pointer Tagging must not belong to the same bits which are used by the hash table to determine the position of the bucket.
Algo. 3.5 presents how to build up the tag during insertion phase, while Algo. 3.6 shows how to make use of the tagged pointers for lookups.

Algorithm 3.5
Chained Hashing – Insertion with Pointer Tagging

```
position ← h(k) mod buckets_HT
oldPointerWithTag ← HT[position]
[oldTag, oldPointer] ← splitPointer(oldPointerWithTag)
newPointer ← Entry(⟨k, v⟩, oldPointer)
bit ← h(k) & (0xF ≪ 32)
newTag ← setBit(oldTag, bit)
newPointerWithTag ← combinePointerWithTag(newPointer, newTag)
HT[position] ← newPointerWithTag
```

Algorithm 3.6
Chained Hashing – Lookup with Pointer Tagging

```
position ← h(k) mod buckets_HT
pointerWithTag ← HT[position]
[tag, pointer] ← splitPointer(pointerWithTag)
bit ← h(k) & (0xF ≪ 32)
if checkBit(tag, bit) then
    while pointer ≠ nullptr do
        if pointer.key = k then return pointer.value
        pointer ← pointer.next
throw NoMatchFound
```

3.3. Open Addressed Hashing

For the synchronization of an Open Addressed Hashing Schema, each thread inserting a key-value pair needs to check if the bucket is still free and if it is, it has to place the data directly in it to prevent another thread from taking the bucket between checking and storing. Moreover, one has to find a way how to indicate if a bucket is still available, as any value can be placed in the bucket.
For any Open Addressed Hashing Scheme, the hash table is an array of buckets containing the key and the value. Each bucket can hold exactly one key-value pair.

3.3.1. Linear Probing Hashing

Starting with the Linear Probing Hash Table, we announce two different ideas on how to enable parallelism when inserting elements.
The first one is to use a `std::atomic_flag`, while the second one uses the storing of the key's hash as synchronization.

3.3.1.1. Locking with `std::atomic_flag`

For this locking strategy, each bucket of the hash table is extended by an instance of `std::atomic_flag`. So, this flag has two functionalities: (1) avoiding multiple threads from

accessing the same bucket and (2) indicating if the bucket is already taken.
Once these considerations are done, the implementation is quite easy, as Algo. 3.7 shows. The critical thing to note is that a bucket's flag is not cleared again after the data has been stored. When inserting a key-value pair $\langle k, v \rangle$, the threads start to iterate from the initial bucket over all the buckets, which are already taken until it reaches one bucket that is still available. This bucket is then directly reserved by the thread by setting the flag and terminating the search for the next free bucket, as TAS returns the value the flag contained previously. Finally, the data can be stored in the bucket.

Algorithm 3.7
Linear Probing – Locking bucket via `std::atomic_flag`

```
position ← h(k) mod buckets_HT
bucket ← HT[position]                                   ▷ Initial bucket
while bucket.flag.TAS() do
    position ← (position + 1) mod buckets_HT
    bucket ← HT[position]
bucket ← ⟨k, v⟩
```

3.3.1.2. Locking with `std::atomic::compare_exchange`

Similar to the synchronization with the flags, this can also be implemented by storing the hash in the bucket. By doing this, the hash indicates if the bucket is free or not.
However, any value for the hash is possible; thus, a specific value must be defined to represent an empty bucket. So, settling on the value 0 stored in the hash field of a bucket indicating that it is free, hashes with this value need to be normalized. This means that the hash gets a predefined non-zero value.
With this adaption, the functionality of the hash field can be changed to represent important information about the bucket.
In C++, the function `std::atomic::compare_exchange` (CAS) performs a comparison of an instance of `std::atomic` (*compared*) with a predefined value (*expected*). If both are bitwise equal, the value of *compared* is replaced with a third value (*desired*), if both aren't the same, the actual value of *compared* is stored in *expected*.
With this function, the value of the hash field of any bucket can be checked if it is 0 and then directly store the hash in the bucket.
With the CAS operation, the thread reserves the bucket and stores the data in it.
Algo. 3.8 shows how synchronization can be implemented using the idea mentioned above.

Algorithm 3.8
Linear Probing – Locking bucket via `std::atomic::compare_exchange`

```
position ← h(k) mod buckets_HT
desired ← normalize(h(k))
expected ← 0
bucket ← HT[position]                                  ▷ Initial bucket
while bucket.hash.CAS(expected, desired) do
    expected ← 0                                       ▷ Reset required
    position ← (position + 1) mod buckets_HT
    bucket ← HT[position]
bucket ← ⟨k, v⟩
```

3.3.2. Robin Hood Hashing

The challenge of a parallel Robin Hood Hash Table is the rehashing of the key-value pairs whose distance is smaller than the distance of the newly inserted element.
When inserting an element into the Linear Probing Hash Table, the bucket is reserved by the thread either by setting the flag or the hash value; no other thread needs to access this bucket afterwards during the insertion phase.
Robin Hood Hashing, in contrast, reenters these buckets, which already contain a key-value pair if it has to replace the data. Thus, the inserting thread has to wait if it gets to a bucket, in which data is concurrently stored to maintain the invariant of the hash table.
To resolve this issue, each bucket of the hash table gets an additional marker which is set when a thread works on the corresponding bucket – this marker tells all other threads, that they have to wait until it is reset.
Moreover, with the same idea of using the hash to represent an empty or full bucket, Algo. 3.9 describes the procedure of inserting a new key-value pair $\langle k, v \rangle$ into a Robin Hood Hash Table.

Algorithm 3.9
Robin Hood on Linear Probing – Insertion

```
position ← h(k) mod buckets_HT
normalizedHash ← normalize(h(k))
expected ← 0
marked ← 42                                          ▷ Random, non-zero number
distance ← 0
while true do
    bucket ← HT[position]
    while bucket.marker.CAS(expected, marked) = false do
        expected ← 0
    if bucket.hash = 0 then                                          ▷ Bucket free
        bucket ← ⟨k, v⟩
        bucket.hash ← normalizedHash
        bucket.marker.store(0)
        break
    if bucket.hash ≠ 0 then                                          ▷ Bucket taken
        if distance > d(⟨bucket.key, bucket.value⟩) then
            oldData ← bucket
            bucket ← ⟨k, v⟩
            bucket.hash ← normalizedHash
            ⟨k, v⟩ ← oldData                          ▷ Rehash the replaced data
            distance ← d(⟨k, v⟩)
        distance ← distance + 1
    position ← (position + 1) mod buckets_HT
    bucket.marker.store(0)
```

3.3.3. Idea for Optimization

When analyzing the implementation of Linear Probing and Robin Hood Hashing, the hash tables have an increased memory consumption as both need additional attributes per bucket. This also leads to fewer buckets, which fit into one cache line.
However, the hash's additional storing can be exploited when storing compressed data as a key-value pair. When a lookup is performed, first, the hash is compared and only if it is the same hash, the data is decompressed and further processing is done based on this data.

4. Evaluation

Throughout the evaluation, the data types of both key and value are 64-bit integers. For the evaluation, a Write-Once-Read-Many workload (WORM) is assumed, like it can be found in database management systems when executing a hash join.
The chapter is structured the following way. Sect. 4.1.1 explains the hardware, on which the experiments were executed. The keys follow four different distributions, as described in Sect. 4.1.2. Insertions with different load factors (explained in Sect. 4.1.3) and lookups with different unsuccessful ratios (see Sect. 4.1.4) on the proposed hash tables are performed and analyzed.
After describing both hardware and data, we start presenting our results. Therefore, we start with the experiments that apply for all hash tables. First comes the analysis of the hash functions in Sect. 4.2.1, followed by Sect. 4.2.2 analyzing the work partitioning we announced in this thesis. Moreover, we answer the question of how to return the values for the lookups efficiently in Sect. 4.2.3.
Finishing everything independent of the hash table, we look at the different synchronization strategies which were presented in Sect. 3.2 and 3.3. Our first hashing scheme is Chained Hashing in Sect. 4.2.4.1 followed by Linear Probing in Sect. 4.2.4.2.
With the information of the experiments before, we look at the performance for the insertion for low load factors in Sect. 4.2.5.1 and for high load factors in Sect. 4.2.5.2.
The performance for lookups is studied in Sect. 4.2.6. The benefits of the optimizations for Chained Hashing are shown in Sect. 4.2.6.1. Sect. 4.2.6.2 evaluates parallel lookups in the different hash tables, depending on the number of threads and the unsuccessful ratio. The evaluation is concluded with the comparison of our implementation's performance with existing parallel hash tables, presented in Sect. 4.2.7.

4.1. Experimental Setup

4.1.1. Hardware Specification

All experiments are executed with a certain number of threads (#threads $\in \{1, 2, 5, 10, 15, 20\}$). For the experiments, a single core (one NUMA region) of a dual-socket machine running two Intel Xeon E5-2660 processors at 2.20 GHz has been used. The machine has in total 220 GB of DDR3-RAM running at 1866 MHz. The OS is a 64-bit Ubuntu 18.04.4 LTS with a Linux 4.15.0-106-generic kernel.
The project is implemented in C++ and compiled with `gcc-7.5.0`. The Makefile is generated by CMake (at least `cmake-3.08` required) and generated in `Release` mode. To build the project, just run `./script/build.sh` in the root directory of the project.

4.1.2. Data Distribution

As already mentioned, the keys and the values are 64-bit integers. We consider four different distributions of the data: **Normal**, **Dense**, **Sparse**, and **Grid**.
The normal distribution is generated with `boost`[1]. We decided to use this library as it offered us the calculation of cumulative distribution function for specific input.
The dense distribution is every key in $[1 : n] \coloneqq \{1, 2, \ldots, n\}$ with n being the number of keys which are inserted.
In the sparse distribution, n keys from $[1 : 2^{64} - 1]$ with $n \ll 2^{64}$ are chosen randomly – uniqueness of the keys is ensured.
For the grid distribution, every byte of every key is in $[1 : 14]$. Thus, there are $14^8 = 1,475,789,056$ possible keys. From these, the first n keys are used in sorted order. n is the number of keys, which are inserted.
For the lookup set, s keys are chosen from the keys inserted, while the rest is filled with keys that are not in the set of inserted ones. This allows us to control the successful/unsuccessful ratio.

4.1.3. Load Factor

The load factor defines the ratio of key-value pairs stored in the hash table to the hash table's size. For the project, the hash tables' size is set to 2^{30} buckets; the load factor is chosen from the set $\{25\%, 35\%, 45\%, 50\%, 70\%, 90\%\}$. The hash tables of Open Addressing Hashing have the same utilization in the buckets as each bucket stores one key-value pair. Therefore, the load factor of these hash tables cannot exceed 1.
In contrast, the number of key-value pairs inserted into a Chained Hash Table is not limited by the hash table's size. However, we define the term **load factor** as the ratio of elements in the hash table to the number of buckets of the hash table. So, for a load factor of 25%, a hash table with 2^{30} buckets stores 2^{28} key-value pairs.

4.1.4. Unsuccessful Ratio

The unsuccessful ratio is chosen from the set $\{0\%, 25\%, 50\%, 75\%, 100\%\}$.
Moreover, the number of keys looked up in the hash tables is predefined with 2^{25} keys. Depending on this number, the lookup amount defines how many keys are taken from the inserted key-value pairs. This means that at 0% each key can be found in the hash table, while for an unsuccessful ratio at 100%, no key can be found.
When generating the data for the lookups, at first the keys from the inserted ones are chosen, then the remaining amount of keys is filled with random numbers which are not in the inserted data.
After all keys are generated, these are shuffled to avoid a regular pattern of successful and unsuccessful queries.

[1] `https://www.boost.org/doc/libs/1_73_0/boost/math/distributions/normal.hpp` (accessed June 26, 2020)

4.2. Results

In this section, we present the numbers we received for the numerous experiments we did. We start with an analysis of the hash functions that we presented in Sect. 2.2. As the hash function needs to be both fast and well distributed in the hash table, we investigate both properties.

4.2.1. Analysis of Hash Functions

Before starting with the analysis of the different hash tables, we want to find out how well the hash functions from Sect. 2.2 perform for different workloads. Therefore, we have a look at the throughput of the hash functions for a selected number of keys and the resulting distribution of the keys in a hash table with 2^{30} buckets. With this, we can say for the Chained Hash Table, how many linked lists with a certain length will occur.

Throughput. For the analysis of the throughout, each hash function gets a vector with 64-bit keys following a specific data distribution. The number of keys is increased tenfold from 10^6 up to 10^9 keys. These keys are then hashed single-threadedly, and the time is measured, it takes to hash the complete vector. The result of this experiment can be seen in Fig. 4.1.

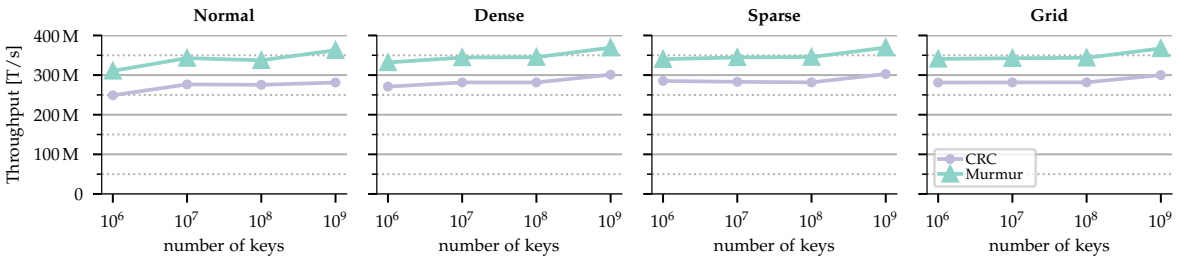


Figure 4.1.: Throughput of the hash functions depending on data distribution and the number of keys.

As the implementation of the Murmur hash (see Sect. 2.2.2) is specialized for working on 64-bit integers, the throughput is always higher than for the CRC hash function, independent of the data distribution or the number of keys. This behavior can be explained when looking at the implementation of both functions.
While the Murmur hash only performs one right shift, one exclusive OR (XOR), and one multiplication two times in a row, the CRC hash uses a combination of additions, multiplications, and logical ANDs, ORs, and XORs. On average, this leads to 30 instructions per key compared with 13 instructions per key for the Murmur hash. This can also be seen when looking at the produced assembly code.
The assembly code in Listing 4.1 shows the important part for the CRC hash of 64-bit integers; the original code is more extended, as this function is implemented for hashing any input data, but not relevant for this length of input type. Therefore, we iterate over the given data and perform several calculations on the input buffer to get a hash value. However, the function checks the remaining sizes of the input and decides how to continue based on this information. This explains the jumps in the lines 8, 12, 16, and 27 which are used if there is a certain size of

data left. When assuming only to hash 64-bit integers, we could get rid of the comparisons and the jumps and thus get about 20 instructions, but this is not the intention behind the CRC hash, so we decided to keep the function as general as possible.
Murmur hash is much shorter, as Listing 4.2 shows. The code presents the necessary steps for getting the hash of a specific key. As already mentioned above, the calculation just depends on right shifts, XORs, and multiplications with constants.

```
1  mov rax, QWORD PTR k0[rip]
2  lea r8, [rdi+rsi]
3  movabs rcx, -49064778989728563
4  add rcx, QWORD PTR k2[rip]
5  mov rdx, rcx
6  imul rdx, rax
7  cmp rsi, 31
8  jbe .L2
9  .L2:
10 lea rcx, [rdi+16]
11 cmp r8, rcx
12 jb .L3
13 .L3:
14 lea rsi, [rdi+8]
15 cmp rsi, r8
16 ja .L4
17 mov rcx, QWORD PTR k3[rip]
18 imul rcx, QWORD PTR [rdi]
19 mov rdi, rsi
20 add rcx, rdx
21 mov rdx, rcx
22 rol rdx, 9
23 imul rdx, QWORD PTR k1[rip]
24 xor rdx, rcx
25 .L4:
26 cmp rdi, r8
27 je .L5
28 .L5:
29 mov rcx, rdx
30 ror rcx, 28
31 xor rdx, rcx
32 imul rdx, rax
33 mov rax, rdx
34 ror rax, 29
35 xor rdx, rax
```

Listing 4.1: Assembly code for CRC hash.

```
1  mov rax, rdi
2  shr rax, 33
3  xor rax, rdi
4  movabs rdi, -49064778989728563
5  imul rdi, rax
6  mov rax, rdi
7  shr rax, 33
8  xor rax, rdi
9  movabs rdi, -4265267296055464877
10 imul rax, rdi
11 mov rdi, rax
12 shr rdi, 33
13 xor rax, rdi
```

Listing 4.2: Assembly code for Murmur hash.

Resulting Distribution. Not only the throughputs of the hash function matter but also the resulting distribution of the keys in a hash table with a specific size. Therefore, we looked at the distribution of the lengths of linked lists that arise in a Chained Hash Table. We set the size of the hash table to 2^{30} buckets.
As there is hardly any difference between the Murmur and the CRC hash, we decided to show

only the Murmur hash ratio. All distributions of the lengths of the lists follow a geometric distribution.
Fig. 4.2 shows the results of the ratios for all load factors of the normal distribution.
However, these plots show that the same load factor's data distributions do not significantly impact the resulting ratios. The higher the load factor gets, the longer the linked lists get and the ratio for linked lists with the length 1, so only one element, shrinks. Moreover, the last number at the horizontal axis indicates the maximum length which occurs.
The reason and the mathematical background of hash collisions can be explained with the Birthday Paradox [Suz+06].

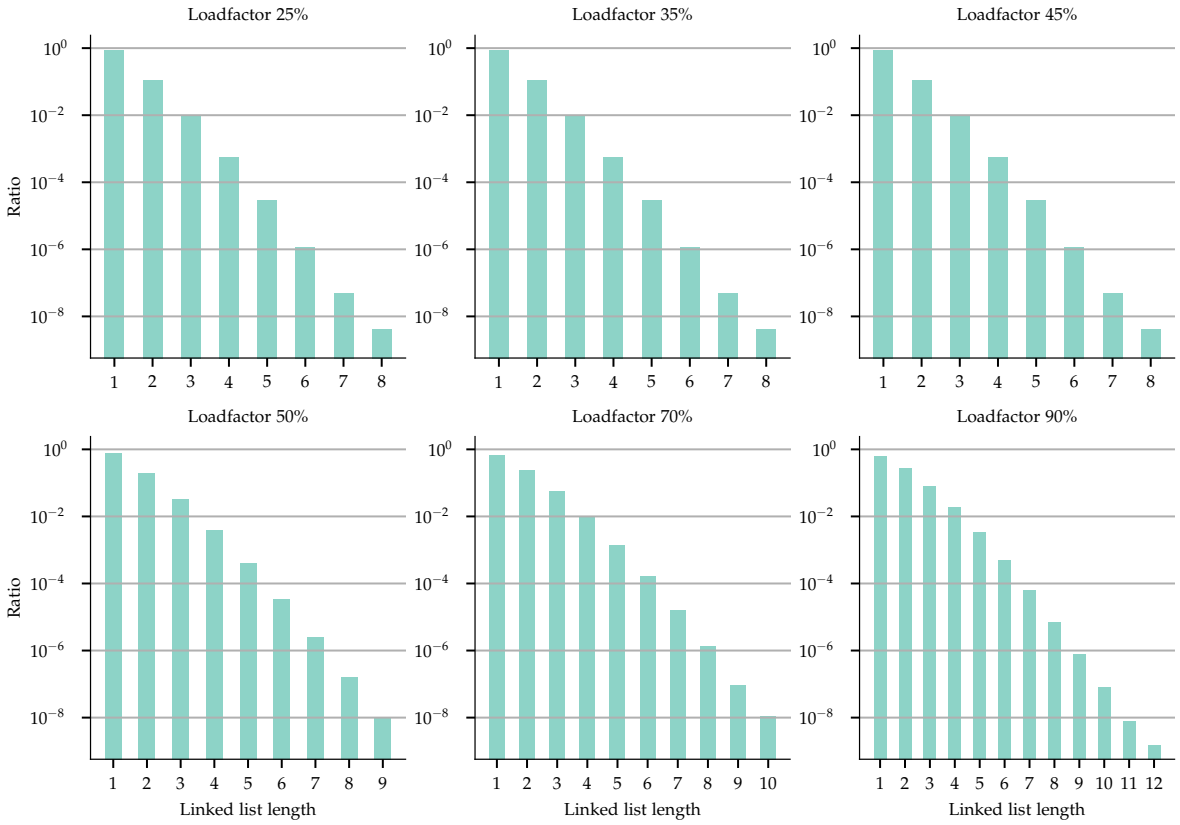


Figure 4.2.: Distribution of linked list length for normal distribution, all load factors, and Murmur hash.

With this experiment, we could show that both hash functions are distributed equally well. The only advantage of Murmur hash is the higher throughput, while the CRC hash has the advantage that it can hash an arbitrary type and is not limited to 64-bit integers. For the following experiments, we only present the numbers for the Murmur hash.

4.2.2. Morsel-Driven vs. Tuple-Driven Parallelism

As explained in Sect. 2.3, we differentiate between the Tuple-Driven and Morsel-Driven work partitioning model.
To analyze which one pays off, we insert the different datasets into the Linear Probing Hash

Table for both variants. For the Morsel-Driven model, the size of the morsels is set to 1024 tuples. The number of threads was varied between 1, 2, 5, 10, 15, or 20 threads.
Fig. 4.3 presents the results for the experiment with 25% load factor, this plot is representative for the other load factors.

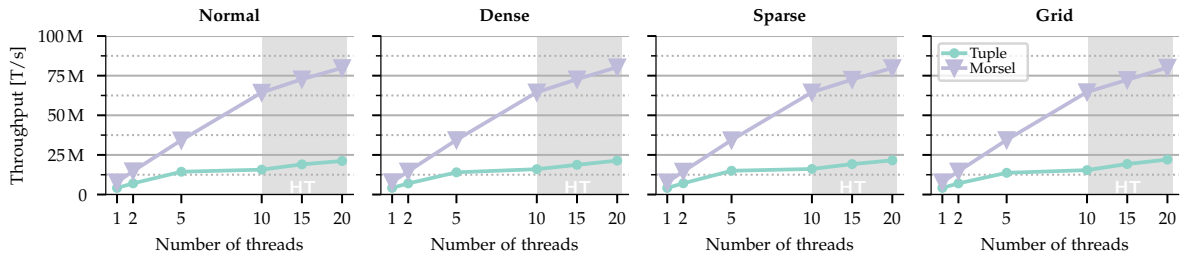


Figure 4.3.: Comparison of Morsel-Driven and Tuple-Driven work partitioning model for Linear Probing Hash Table at 25% load factor.

The Morsel-Driven model's main benefit is that the `atomic` counter, which is used for getting the morsel, is no frequently accessed counter, so, one thread directly gets a whole range of indexes that it needs to process. Thus, it is less likely that many threads want to access the counter simultaneously. We also tried different morsel sizes, but we could figure out that the throughput of the different sizes has soon approached the one displayed. For even bigger sizes, no significant speedup could be recorded. So, we decided to set the size of the Morsel-Driven work partitioning model to 1024 tuples.
In contrast, the Tuple-Driven model needs its counter to be incremented for every single tuple. For the Tuple-Driven model, this leads to the problem that threads are delayed until they can increment the counter safely, as it is more likely that other threads want to access the counter. So this thread cannot process any data in the meantime.
With this knowledge, we use our Morsel-Driven partitioning model for every other experiment.

4.2.3. Returning Lookup Results

As discussed in Sect. 3.1, one problem is how to return results for the lookups. One possibility is to design a custom exception that is thrown every time the key could not be found. The second option is to use the C++ `optional` header, which allows us to combine the information if a value was set to a variable and the value if one was found. Both ideas were evaluated and the results are shown in Fig. 4.4, which is representative for all other hash tables.

The idea of using a custom exception does not pay off, as exceptions start to become very expensive when the unsuccessful ratio increases. Using the `optional` header, we can see a slight decrease in the throughput when hitting 75% unsuccessful rate.
However, this plot clearly shows us that it is not worth to use an exception, but to go with the `optional` class.

4.2.4. Synchronization Strategies

In this section, we want to discuss the different locking strategies, which we announced in Chapter 3. We use each hash table with its different synchronization techniques and

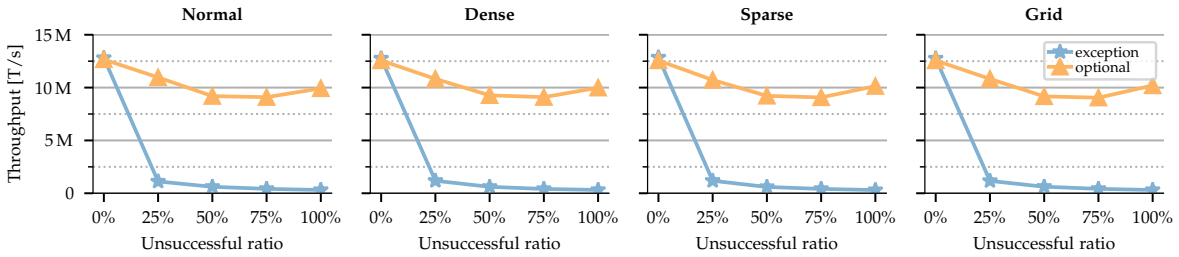


Figure 4.4.: Comparison of returning lookup results using a single thread for Robin Hood Hashing.

insert datasets with a particular load factor for this experiment. As we are interested in the performance when using multiple threads, we vary the number of threads. The hash tables use the Morsel-Driven execution model, that we have validated before.

4.2.4.1. Chained Hashing

The strategies are shown in Sect. 3.2.1. As we have already stated in Sect. 3.2.1.1, the synchronization with `std::mutex` is not investigated explicitly due to massive memory overhead. Thus, we have three hash tables left: (1) locking with a flag (short: **fCH**, Sect. 3.2.1.2), (2) locking with pointer smashing (short: **sCH**, Sect. 3.2.1.3), and (3) exchanging the pointers atomically (short: **eCH**, Sect. 3.2.1.4).
Fig. 4.5 and 4.6 present the throughput we can achieve using a certain synchronization strategy.

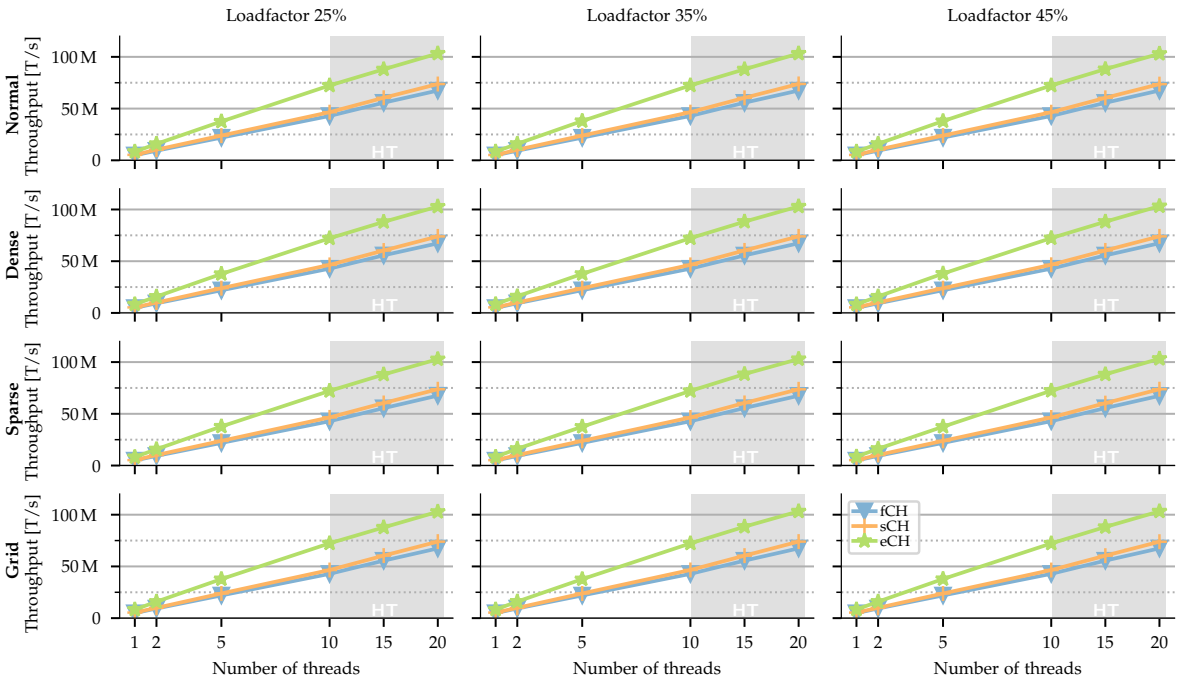


Figure 4.5.: Comparison of the locking strategies for Chained Hash Table (see Sect. 3.2.1) and 25%, 35%, and 45% load factor.

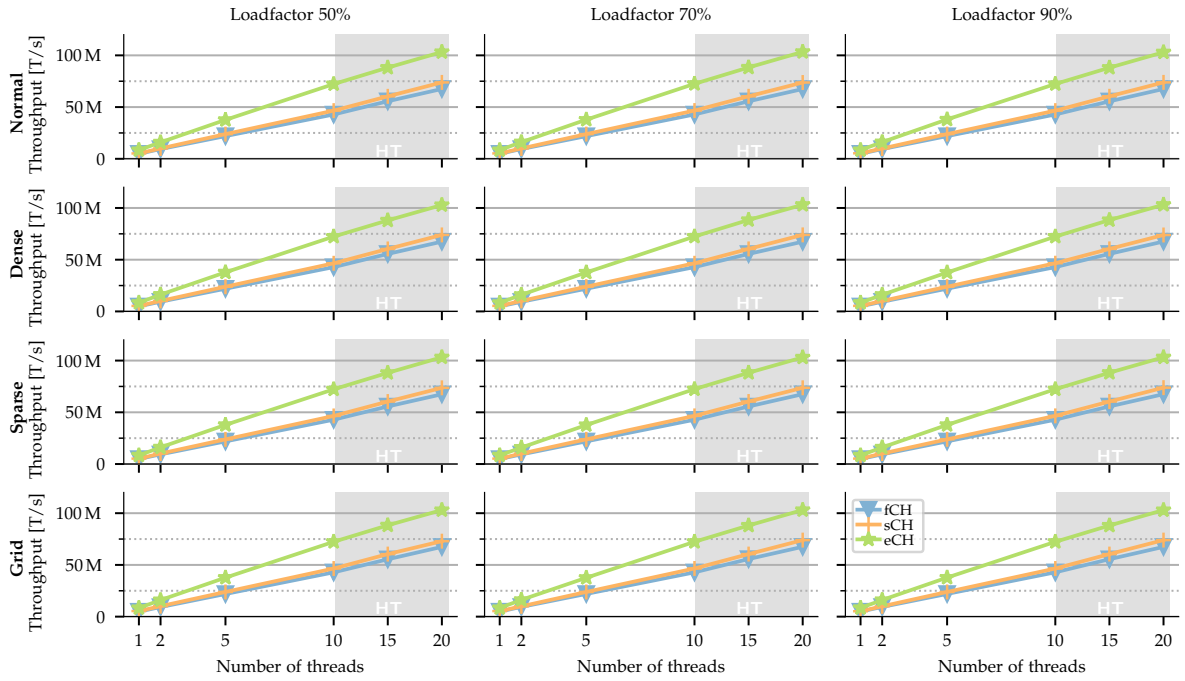


Figure 4.6.: Comparison of the locking strategies for Chained Hash Table (see Sect. 3.2.1) and 50%, 70%, and 90% load factor.

As we can see from these plots, both strategies which are based on spinlocks (**fCH** and **sCH**) have a similar throughput. The idea which swaps the pointers in the hash table performs best in all experiments.
Now, we combine our observations with our code:

fCH: Locking with `std::atomic_flag`. After the work is done, the flag needs to be cleared again. The operation to clear the flag is implemented internally with `__cxx_atomic_store`.

sCH: Locking with Pointer Smashing. To unlock the bucket after insertion of the new element, a new pointer is stored. Again, this operation is implemented internally with `__cxx_atomic_store`.

In the end, both implementations for **fCH** and **sCH** map to the same basic instructions on machine code level. When profiling our implementations, we can figure out that the produced assembly codes for the strategies **fCH** and **sCH** have an `mfence` instruction. According to Intel's manual, this instruction serializes all store and load operations that occurred before the `mfence` instruction in the program instruction stream [Int16].
This instruction ensures that the data has been written before the thread continues. Thus, the threads have to wait until they are allowed to pass the memory fence.

eCH: Pointer Exchanging with `std::atomic::exchange`. Our last implementation does not need any explicit storing operations, as it just swaps the pointers. The exchanging is implemented with `std::atomic::exchange`, which itself is realized with `__cxx_atomic_exchange`.

A small example with Godbolt[2] allows us to verify our assumption about the produced assembly instructions with the same compiler and optimization level. Storing operations cause a `mfence` instruction, while exchanging operations don't.
The effect of different strategies can also be observed when looking at the number of cycles per tuple, as shown in Table A.1 in the Appendix. After the cycles first decrease from 1 to 5 threads, which might be due to the parallel overhead, they increase with a growing number of threads. However, the difference between the cycles of various synchronizing strategies remains nearly constant throughout the different experiments. We also notice, that the cycles for the strategies **fCH** and **sCH** only differ by about 70 cycles, while strategy **eCH** needs approximately 200 cycles less than **sCH**. These differences also explain the development of the throughput with more threads.
Another crucial point to realize is that the Chained Hashing scheme is independent of the load factor and the data distribution.
This cannot be seen only by the development of the throughput but also by considering the instructions in Table A.4 during insertion. For every load factor, synchronization strategy, and amount of threads, the number of executed instructions is nearly the same. The Chained Hash Table simply gets the bucket and the position where to store the new key-value pair. No other processing or calculations need to be done when inserting a new element.
The outcome of this experiment is to use the hash table synchronized with Pointer Exchanging. However, this hash table does not allow us to use the optimization idea of Pointer Tagging. It is challenging to implement the placing of the bit in the upper 16 bits together with the replacement of the real pointer in the lower 48 bits at the same time. When wanting to use Pointer Tagging, we can switch to the hash table, synchronized by Pointer Smashing. Moreover, we can avoid the additional flags which consume quite a significant amount of memory for huge hash tables.

4.2.4.2. Linear Probing Hashing

In Sect. 3.3.1, we announced two different mechanisms how to realize the synchronization for Linear Probing Hashing. We have two strategies: (1) locking with flag (short: **fLP**, Sect. 3.3.1.1) and (2) locking with exchanging the hash (short: **eLP**, Sect. 3.3.1.2). For the evaluation, we perform insertions of various datasets using differing numbers of threads.
As Fig. 4.7 and 4.8 show, there is hardly any difference between both strategies. However, the hash table using the hash for synchronization is a little bit slower than the one with the flags.

[2] `https://godbolt.org/z/deGbcW`

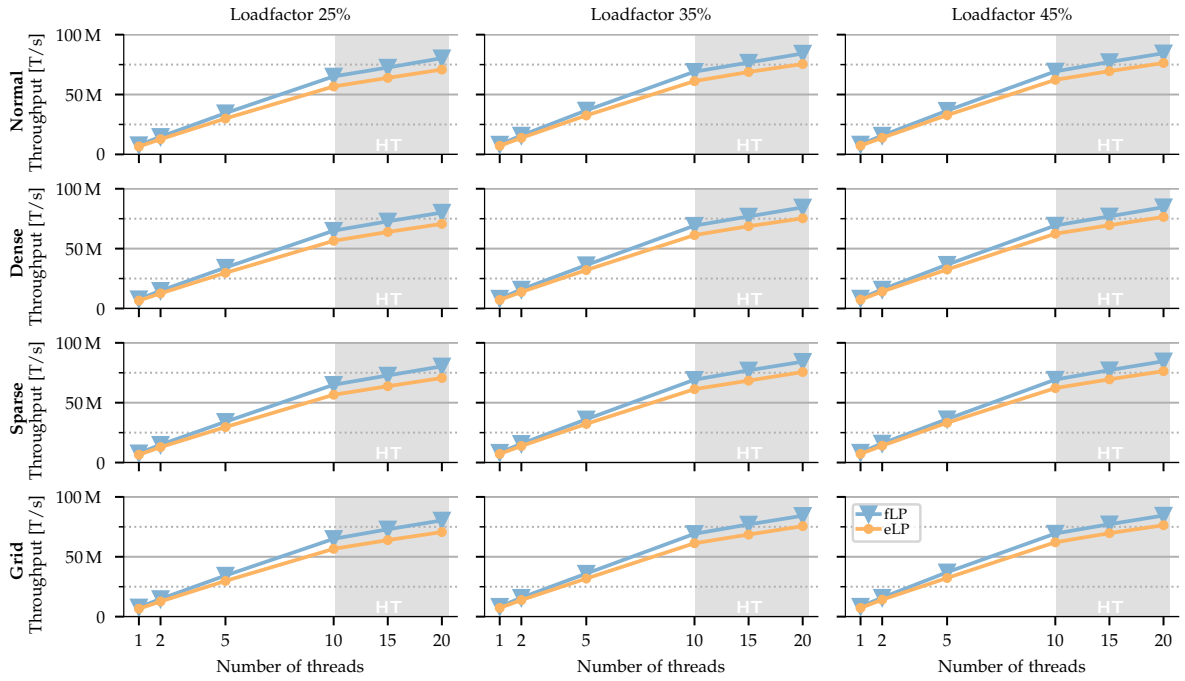


Figure 4.7.: Comparison of the locking strategies for Linear Probing Hash Table (see Sect. 3.3.1) and 25%, 35%, and 45% load factor.

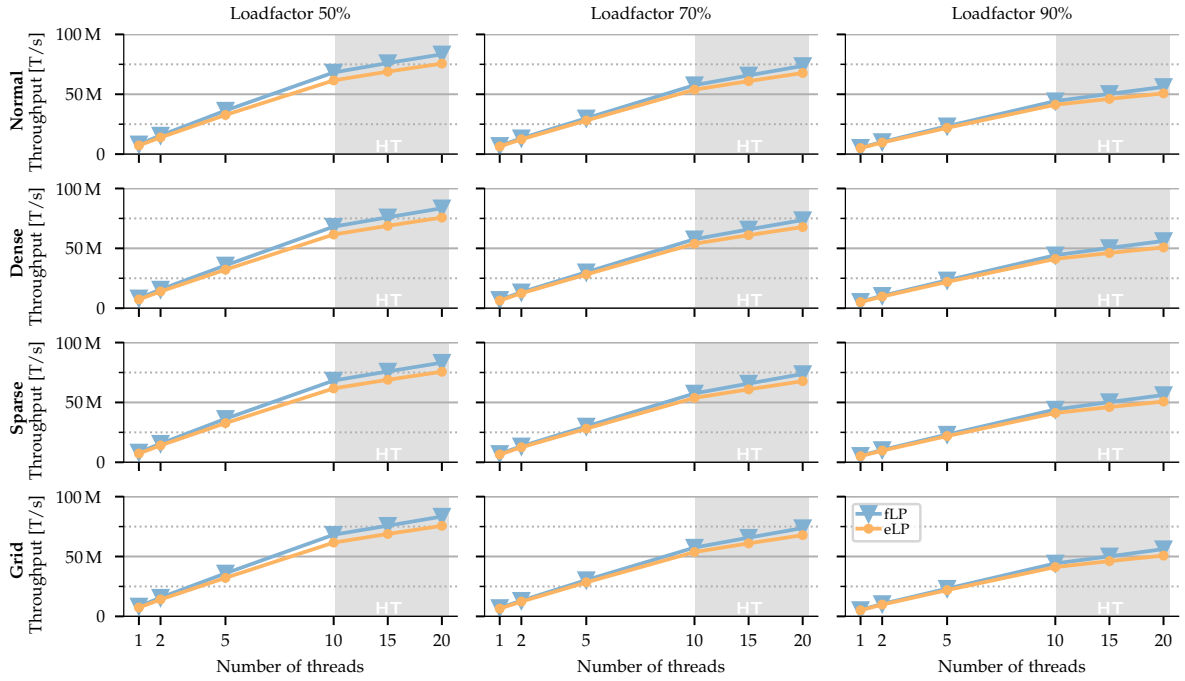


Figure 4.8.: Comparison of the locking strategies for Linear Probing Hash Table (see Sect. 3.3.1) and 50%, 70%, and 90% load factor.

fLP: Locking with `std::atomic_flag`. The function `test_and_set`, that we use for reserving the bucket, is implemented internally with `__cxx_atomic_exchange` – the produced assembly code is shown in Listing 4.3. We can see that this function is realized with a simple `xchg` instruction (line 9), followed by a `test` to check if the flag was set before (line 10). When profiling our implementation, this is also the only hotspot in the code, as every bucket touched by a thread needs to be tested and, if available, locked before further processing. The relevant part to notice is that the flag is not cleared again, as the flags are used to indicate the bucket's availability. Thus, we do not trigger a memory fence instruction as it was the case in Chained Hashing.

```
1  jmp 1cb03                          jump to bucket reservation (line 5)
2  mov 0x38(%rdi),%r8                 begin of the loop for iteration over the buckets
3  add $0x1,%rbp                      incrementation of bucket's position
4  and %r9,%rbp                       limit position to hash table size
5  mov %rbp,%rax                      start of bucket reservation: get flag of corresponding bucket
6  mov %r10d,%r13d
7  shl $0x4,%rax
8  add %rbp,%rax
9  xchg %r13b,0x10(%r8,%rax,1)        set operation of flag.test_and_set()
10 test %r13b,%r13b                   check if flag was set by operation before
11 jne 1caf8                          jump to loop begin (line 2) as bucket is not free
12                                    bucket reserved, continue with workload
```

Listing 4.3: Produced assembly code for Linear Probing Hashing, synchronized with flag.

eLP: Locking with `std::atomic::compare_exchange`. For this option, the value in the hash field indicates if the bucket is free or not. The compare and exchange operation needs to be done without any other thread being able to interrupt.
This is implemented with `compare_exchange_weak` (CAS), which itself is implemented with `__cxx_atomic_compare_exchange_weak`. By looking at the assembly code, we see that the produced instruction is `cmpxchg`. This compare and exchange instruction is itself protected by a `lock` instruction. To understand the need for this protecting instruction, the Software Developer's Manual [Int16] says that a locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may be interpreted by the system as a lock for a larger memory area. It may also lead to a bus lock.
Thus, the instruction prevents the internal one from being disturbed by another thread working on the same memory area. Using this concept in our use case, the internal comparison and exchange can take place safely. However, the assembly code for this strategy shown in Listing 4.4 includes the instruction for CAS in the lines 1 and 9 twice. The reason is that the bucket's position is incremented by one if the hash was not swapped successfully. So, before entering the loop, the CAS instruction in line 1 checks if the initial bucket was free. On success, the program jumps to the part where placing the data takes place. Otherwise, the loop begins, the position is incremented, and the bucket is rechecked. This second check is the second CAS at the end of the while loop (line 9), again on success, the program goes to the part where the data is stored, while on failure, the program jumps to the begin of the loop.

```
1  lock cmpxchg %rcx,0x10(%r10,%rbp,1)                    check if initial bucket is free
2  je 1c180                                               jump to storing the data on success
3  add $0x1,%rbx          begin of the loop for iteration over the buckets, increment position
4  mov 0x38(%rdi),%r10
5  xor %eax,%eax
6  and %r12,%rbx                                          limit position to hash table size
7  lea (%rbx,%rbx,2),%rbp                                 get hash field of next bucket
8  shl $0x3,%rbp
9  lock cmpxchg %rcx,0x10(%r10,%rbp,1)                    check if bucket is free
10 je 1c180                                               jump to storing the data on success
11 jmp 1c1fd                                              check next bucket (line 3)
```

Listing 4.4: Produced assembly code for Linear Probing Hashing, synchronized with hash.

When profiling the Linear Probing Hash Table for different load factors, we get the cycle numbers shown in Table A.2. These numbers indicate that more cycles are needed with higher load factors as it takes longer for the threads to find an empty bucket. So, more buckets need to be checked, which then leads to more cycles. Moreover, this can also be observed when taking a look at the rate of how often a particular assembly line was executed. Therefore, we focus on the two lines which perform the CAS operation. For the load factor of 25%, about 85% of all bucket accesses succeed for the first time, while for a load factor of 90%, only approximately 60% succeed directly. The rest needs to start iterating, which then leads to more cycles. All of these effects lead to a shrinking throughput when dealing with a higher load factor.
The outcome is that we use the synchronization strategy with the flag, as this offers slightly higher throughput.

4.2.5. Insertion

Now, we want to compare the performance of the different hash tables which performed best in the analysis before. Therefore, we use the following hash tables:

1. **eCH**: Chained Hash Table with Pointer Exchanging
2. **sCH**: Chained Hash Table with Pointer Smashing
3. **fLP**: Linear Probing Hash Table with flag
4. **RHH**: Robin Hood on Linear Probing Hash Table

We decided to present two Chained Hashing options, as the first combination does not support Pointer Tagging, which is investigated later on. With this selection of hash tables, we cover all essential information which is needed in the next experiments.
Before we get to the throughput analysis, we want to consider each hash table's memory consumption.

Memory Footprint. The used keys and values are 8B integers, as well as the pointers are 8 Bytes big. For the flags, we assume a size of 1 Byte.
Thus, we get the following sizes for the hash tables, lf is replaced by the load factor:

1. **eCH**: $2^{30} * 8B + 2^{30} * (8B + 8B + 8B) * lf$
2. **sCH**: $2^{30} * (8B + 1B) + 2^{30} * (8B + 8B + 8B) * lf$
3. **fLP**: $2^{30} * (8B + 8B + 1B)$
4. **RHH**: $2^{30} * (8B + 8B + 8B)$

With the calculations above, we can see that the size for Chained Hashing depends on the load factor, while Open Addressed Hashing result in a constant size.
By reference to our results for the different synchronization strategies, we could see that collision resolution for Open Addressed Hashing becomes relevant at a load factor of 50%. So, we split them into low and high load factors.

4.2.5.1. Low load factors: 25%, 35%, 45%

Our first experiment focuses on the lower load factors, which are less than 50%. The results can be seen in Fig. 4.9. It shows the different results for the hash tables, when inserting varying datasets with changing load factors. Each plot shows the throughput the corresponding hash table could achieve when using a certain number of threads. When running the Chained Hash Table with Pointer Exchanging (**eCH**), we see that this hash table has the highest throughput. However, **fLP** can stick with **eCH** quite well when running up to 10 threads. When exceeding the 10 threads, the throughput of **fLP** starts to stagnate. This observation can also be confirmed when we look at the development of cycles for the corresponding hash table in Table A.1 for **eCH** and in Table A.2 for **fLP**. Up to 10 threads, the number of cycles is in the same range while for more threads, the cycles per tuple for **fLP** start to increase more than for **eCH**.
Compared with the other Chained Hash Table (**sCH**), **fLP** performs better. Due to the bend of the throughput of the Linear Probing Hash Table, the **sCH** has nearly the same throughput for 20 threads as **fLP**.
The last hash table would be the Robin Hood Hash Table (**RHH**). This is the slowest of all the investigated hash tables, thus has the lowest throughput. We can confirm this when looking at the number of instructions; the hash table has to perform for one element. Therefore, Table A.6 shows, that the amount of instructions is nearly twice as many as for the Linear Probing Hash Table.
By reference to the hardware of the server, we can explain what causes this bend. One processor has 10 cores that support hyper-threading so that we can run up to 10 threads per core and 10 additional threads as hyper-threads. The Intel Hyper-Threading technology enables a single physical processor to execute two or more separate code streams (called threads) concurrently [Int16]. However, when using hyper-threads, a single processor core offers two logical processors that share execution units. In our implementations, the **fLP** has to access much more memory than the **eCH** as the implementation needs to iterate over already taken buckets. Due to this, the processor's memory unit is busier than in the **eCH**, which then leads to more workload for the memory unit. As this unit has to be shared, the throughput starts to decrease when starting to use hyper-threads.

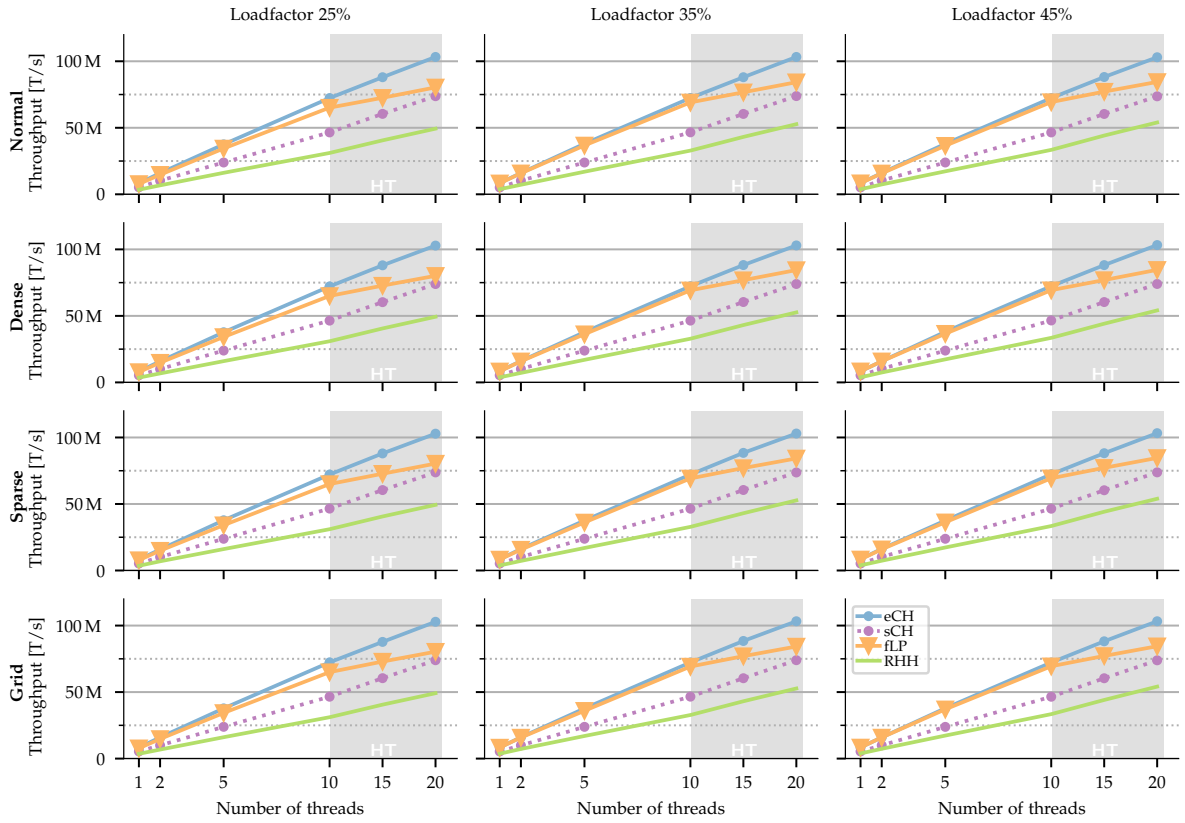


Figure 4.9.: Comparison for insertion into the hash tables for low load factors 25%, 35%, and 45%.

4.2.5.2. High load factors: 50%, 70%, 90%

In our second experiment, we focus on the higher load factors, which lead to the utilization of at least 50%. The results can be seen in Fig. 4.10. The performance of both **eCH** and **sCH** remains the same as before through all the experiments. However, when looking at the **fLP**, we notice a decrease in the throughput by increasing the load factor from 50% to 90%. This behavior was explained explicitly in Sect. 4.2.4.2. We also notice that the bend when hitting the 10 threads and moving on to hyper-threads remains the same in these load factors. Moreover, for 90% utilization, **sCH** performs the same as **fLP** for up to 10 threads and then outperforms it with more threads. The Robin Hood Hash Table again is the slowest one, but this is due to the overhead these hash tables have when inserting a new entry. The more entries are already stored in the hash table, the more likely it is that another element needs to be replaced and rehashed. This leads to an increasing number of instructions and thus cycles as Table A.6 and A.3 show.

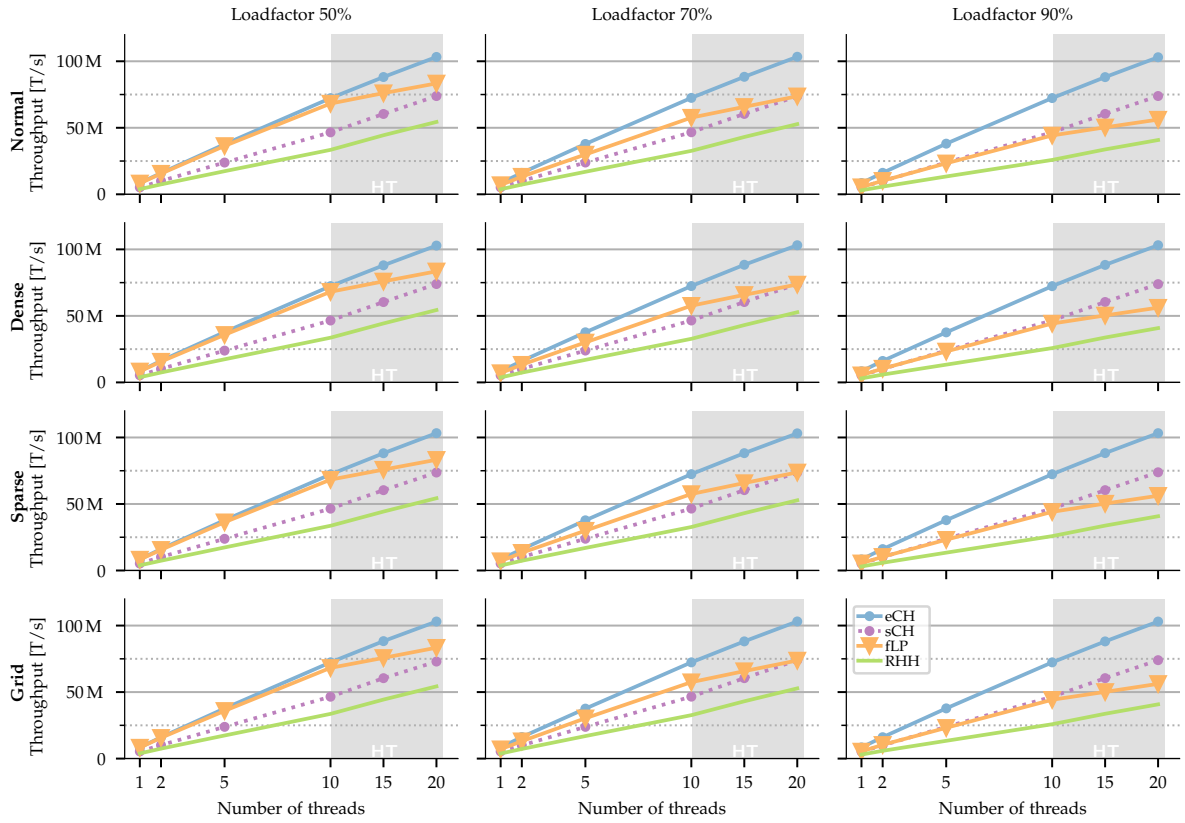


Figure 4.10.: Comparison for insertion into the hash tables for high load factors 50%, 70%, and 90%.

4.2.6. Lookup

Not only does the performance for inserting elements into the hash tables matter but also the time it takes to look up a key. We now want to analyze the lookup performance of the hash tables we have presented before.
Therefore, we first analyze in Sect. 4.2.6.1 how the optimizations of Chained Hashing pay off before Sect. 4.2.6.2 presents the performance of parallel lookups in the hash tables.
Moreover, we vary the unsuccessful ratio of our lookups, as explained in Sect. 4.1.4.

4.2.6.1. Optimizations for Chained Hashing

In Sect. 3.2.2, we announced two ideas how to optimize Chained Hashing. Fig. 4.11 presents the throughput we could achieve for 90% utilization of the hash table, so a 90% load factor, running the lookup with only one thread. Based on our experiment from Sect. 4.2.3, we decided that the values are returned using instances of the `optional` class.
Our first attempt to improve the performance of lookups in Chained Hashing is prefetching the next entry of the linked list when traversing it to check for the key. The results show us that this idea does not pay off. Prefetching has an imperceptible higher throughout than the basic implementation. The reason for this might be that the prefetch does not have enough time to get the data from memory. With this knowledge, we do not need to consider this kind

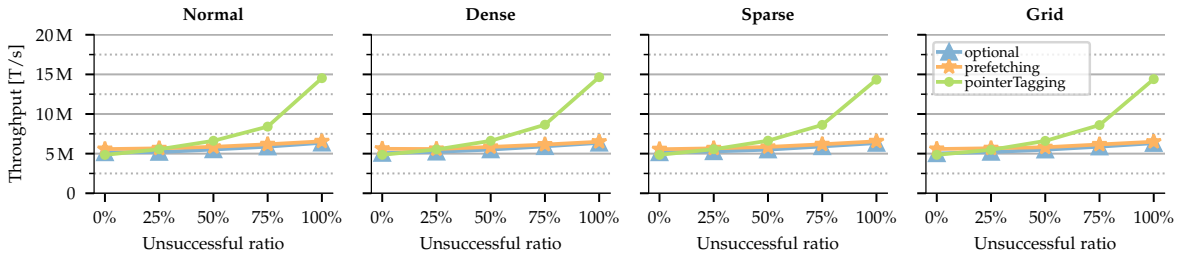


Figure 4.11.: Comparison of the optimizations for Chained Hashing for 90% load factor running single-threaded.

of optimization for the single-threaded case anymore. However, prefetching is just analyzed when running a single thread. When more threads are used for lookups, then the threads could make use of group prefetching by communicating the pointers of the elements they will need and giving the prefetcher enough time to get the data to cache.
The second attempt is to use Pointer Tagging. As we have explained in Sect. 3.2.2.2, Pointer Tagging intends to discard keys as early as possible to avoid the expensive traversing of the linked list. From the throughput, we can see that the idea of Pointer Tagging pays off very well, especially when the ratio of unsuccessful queries increases. For a ratio of 0% or 25%, the efficiency of both implementations – the basic one and the improved one – are approximately the same. A slightly slower throughput (mainly for 0% unsuccessful ratio) is due to the overhead of computations which need to be done to get the mask from the modified pointer and to check for a match, as well as getting the pointer to the entry of the list, but this does not have a massive impact on the performance. However, when the unsuccessful ratio is 50% or higher, the idea of Pointer Tagging leads to improved throughput. Especially for 100% of unsuccessful queries, we can detect a performance about twice as fast as the base case.

4.2.6.2. Parallel Lookups

After looking at the optimizations of Chained Hashing closely, we compare the different hash tables concerning their performance for lookups in parallel. Hereby, we run the experiment on four different hash tables:

1. **eCH**: Chained Hash Table with Pointer Exchanging
2. **sCH**: Chained Hash Table with Pointer Smashing and Pointer Tagging
3. **fLP**: Linear Probing Hash Table with flag
4. **RHH**: Robin Hood Hash Table

As several threads which look up keys in the hash table do not interfere with each other, we do not need any synchronization between the threads. Thus, we do not have to differentiate between the synchronization strategies of the hash tables. However, as we have seen that the optimization idea of Pointer Tagging already gives a significant performance improvement in the single-threaded case, we decided to take a closer look at the parallel performance for the Chained Hash Table.
To better visualize the results, we limit the unsuccessful ratio to $\{0\%, 50\%, 100\%\}$.

Low load factors: 25%, 35%, 45%. Starting with the low load factors, Fig. 4.12 presents the throughput of the hash tables above and Fig. 4.13 shows the cache misses each key causes for its lookup when running 10 threads.

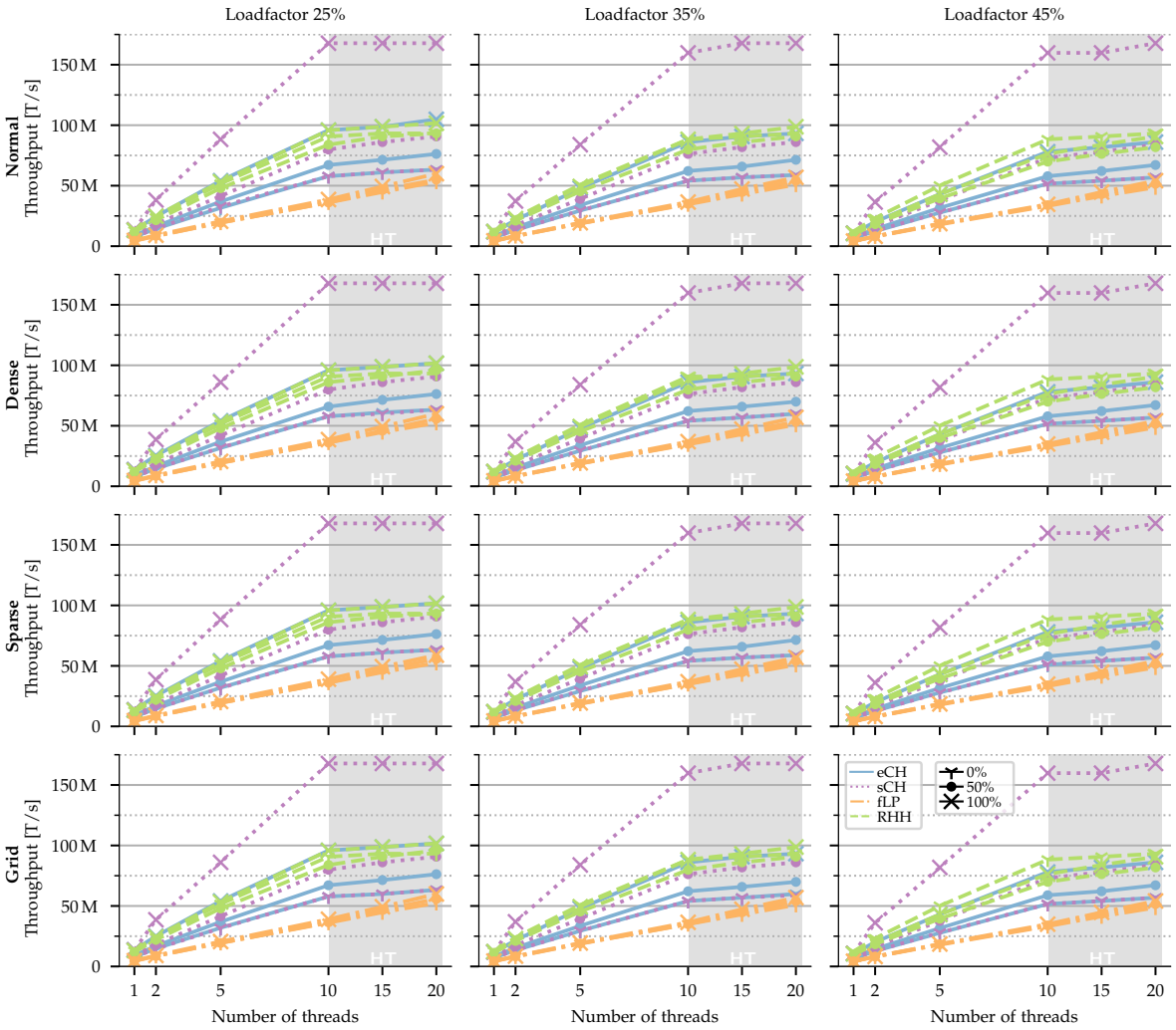


Figure 4.12.: Throughput for parallel lookups with different unsuccessful ratios (0%, 50%, 100%) for 25%, 35%, and 45% load factor.

As the plots show, all hash tables benefit from parallelism, but the throughput starts to slow down, when hitting the boarder to hyper-threads, so that it nearly gets constant. This is due to frequent communication with memory, which is needed to load the data. Thus, the hyper-threads do not provide any benefit to the implementations.
Additional to this perception, we can see two important parts from these plots. The first is that the optimization of Pointer Tagging for the Chained Hashing (**sCH**) pays off, especially when it comes to a high ratio of unsuccessful queries. When each looked up key is not in the hash table, so a 100% unsuccessful ratio, the throughout is nearly 1.5 times higher than for all the other hash tables. However, already for about 50% of unsuccessful lookups, the throughput can be increased. This behavior can be explained when looking at the cache misses, a lookup

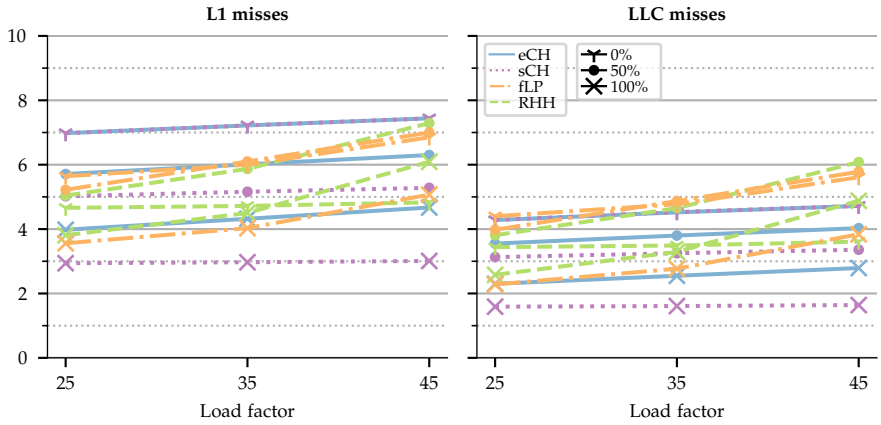


Figure 4.13.: Development of the cache misses during the parallel lookup for parallel hash tables for 25%, 35%, and 45% load factor.

causes. The number of both L1 misses and LLC misses is always smaller than for **eCH** for all different load factors; the only exception is when all keys can be found, as then Pointer Tagging does not reject any key. So, by using this optimization, many keys can be discarded before starting traversing the linked list. Additional to this, the throughput also shows that we can use **sCH** also for 0% unsuccessful ratio, as there is no difference between **sCH** and **eCH**. The second point is that Linear Probing is outperformed by all other hash tables, independently from the lookups' ratio. The number of cache misses continuously increases with a higher load factor. **fLP** causes approximately the same number of cache misses for 0% and 50% unsuccessful lookups. The data in a Linear Probing Hash Table build up clusters during insertion, as collisions are resolved by looking for the next free bucket using a sequential scan. So, the hash table needs to iterate over the buckets for these two unsuccessful ratios until it can terminate. There is also no invariance which gives information where the element can be expected; the whole clusters need to be scanned.
Getting a closer look at the Robin Hood Hash Table, we can see that the re-organization during the insertion and the overhead we have to pay for replacing and rehashing the element are worth its cost, especially when nearly every key is in the hash table. We ensure to keep the key as close to its initial position as possible by doing this re-organization. With this, some cache misses which are caused by the iteration over the hash table are avoided. This behavior can be observed very clearly when looking at the cache misses for a 45% load factor. The lowest number of cache misses is caused when the unsuccessful ratio is 0%. The reason for this is that in the best case, the element is only a few positions away from its initial bucket so that the sequential scan can be performed quickly and the element is found soon. This means that only a few cache lines are needed to be loaded to cache. The next higher number is for a 100% unsuccessful ratio, while the lookup for 50% causes most misses. For a 100% unsuccessful ratio, combined with this load factor, the lookup directly hits an empty bucket or it needs to iterate until it hits an empty one. In contrast, the 50% unsuccessful ratio is a combination of both, so the lookup either hits an empty bucket directly or also needs to iterate. However, the iteration takes longer, as 50% of the keys can be found. Thus, this leads to the highest number of cache misses.

High load factors: 50%, 70%, 90%. For the high load factors, the throughput is presented in Fig. 4.14 and the development of the cache misses for the normal distribution with 10 threads for changing load factors and unsuccessful ratios in Fig. 4.15.

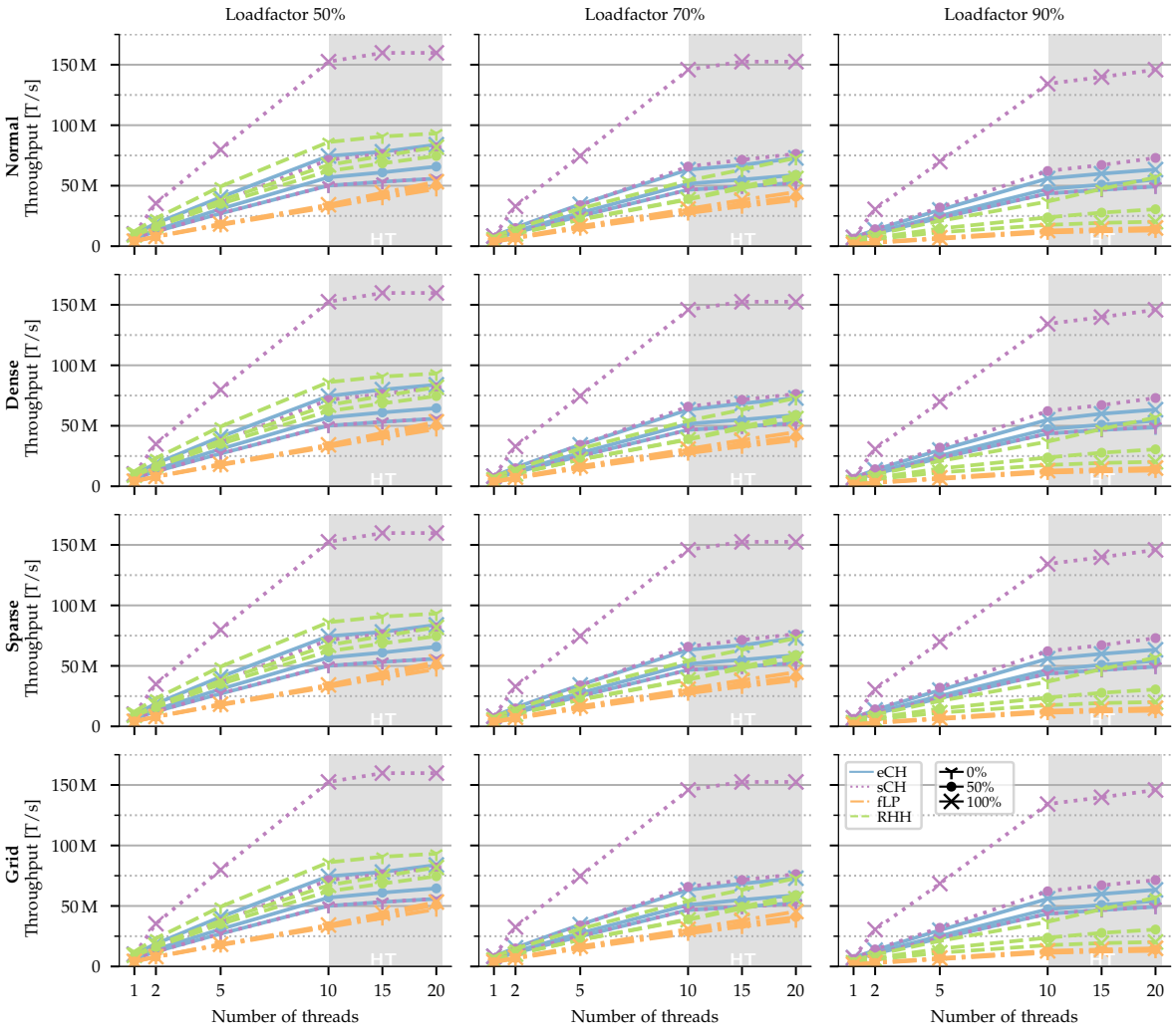


Figure 4.14.: Throughput for parallel lookups with different unsuccessful ratios (0%, 50%, 100%) for 50%, 70%, and 90% load factor.

From the results, we can see that **sCH** performs best again. For a 0% unsuccessful ratio, it is as fast as **eCH**, which does not include any additional computational overhead by calculating the bit for the bitmask of the pointer. This again pays off when it comes to higher unsuccessful ratios, as we can reach a throughput twice as high as for the other hash tables. Looking at the cache misses this hash tables causes, we still observe that the number of misses is constant for changing load factors. We can also see that for both Chained Hash Tables, the number of cache misses per load factor decreases when the unsuccessful ratio increases. This is due to the hash table's construction, as the thread either hits a bucket with a `nullptr`, which indicates that the bucket is empty or the chain is relatively short and does not cause much cache misses. However, **sCH** again shows its benefit by rejecting keys earlier, causing only the load of the

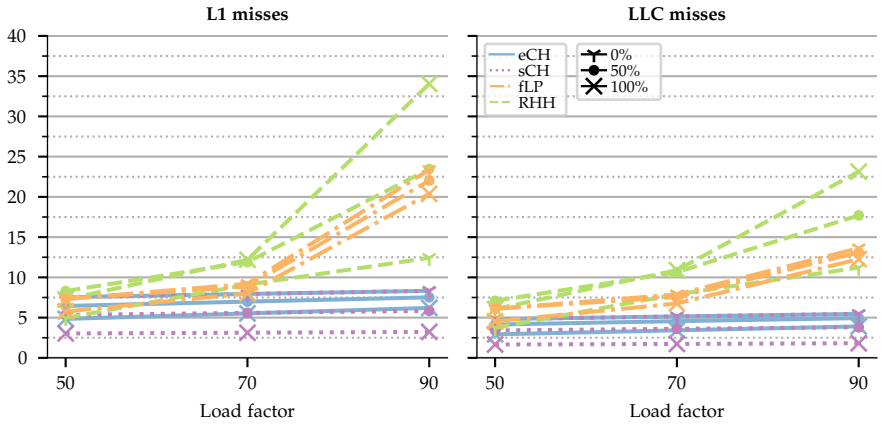


Figure 4.15.: Development of the cache misses during the parallel lookup for parallel hash tables for 50%, 70%, and 90% load factor.

pointer. The behavior also explains that the cache misses for 100% unsuccessful ratio in **sCH** is only half of the cache misses for the same ratio for **eCH**.
Similar to the low load factors, **fLP** again has the lowest performance for any unsuccessful ratio. This is again because of the scans of the clusters which build up during insertion. The length of these clusters increases when the load factor grows, so more cache lines have to be loaded from memory to be scanned.
The same effect can be observed when looking at **RHH**. This hash table has the same clusters as the Linear Probing one but contains the data in another permutation to maintain the invariance of Robin Hood Hashing. According to this, the data is kept closer to the initial bucket, so the lookups are faster than for Linear Probing Hashing when the key is in the hash table. This can also be seen for 90% load factor and an unsuccessful ratio of 0%; **RHH** nearly performs as well as **eCH** for the same workload, while **fLP** is outperformed by a factor of 2. This is also verified by the number of cache misses the hash tables cause, as for the case of high success rate, only a few cache lines need to be checked while for a huge load factor and high unsuccessful rates, an increasing number of data has to be accessed.

4.2.7. Comparison with other Parallel Hash Tables

Finally, we want to compare our implementation with existing parallel hash tables. This experiment was executed on an Intel Core i9-7900X CPU with 3.30GHz and 128GB of memory at 2133MHz DDR4, running Ubuntu 19.10 with kernel version 5.3.0-64-generic.
Therefore, we look at `tbb::concurrent_hash_map`[3] developed by Intel in their Thread Building Blocks library, `libcuckoo::cuckoohash_map`[4] by Fan et al. published in [FAK13; Li+14], and `phmap::parallel_flat_hash_map`[5]. We test the throughput when inserting 10 million and 100 million key-value pairs (dense distribution), using 8 threads, as well as looking up all keys with 8 threads. The returned values are validated by comparing with the input data.

[3] `https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/concurrent_hash_map.html` (accessed June 15, 2020)
[4] `https://github.com/efficient/libcuckoo` (accessed June 15, 2020)
[5] `https://github.com/greg7mdp/parallel-hashmap` (accessed July 3, 2020)

The size of **eCH** is set to 2^{30} buckets, while the size for **fLP** is varied depending on the number of inserted elements. For smaller numbers of elements, the size is either 2^{25} or 2^{30} buckets, while for the 100 million elements, the size is set to 2^{30} buckets. Fig. 4.16 shows the results we could gain from the experiment.

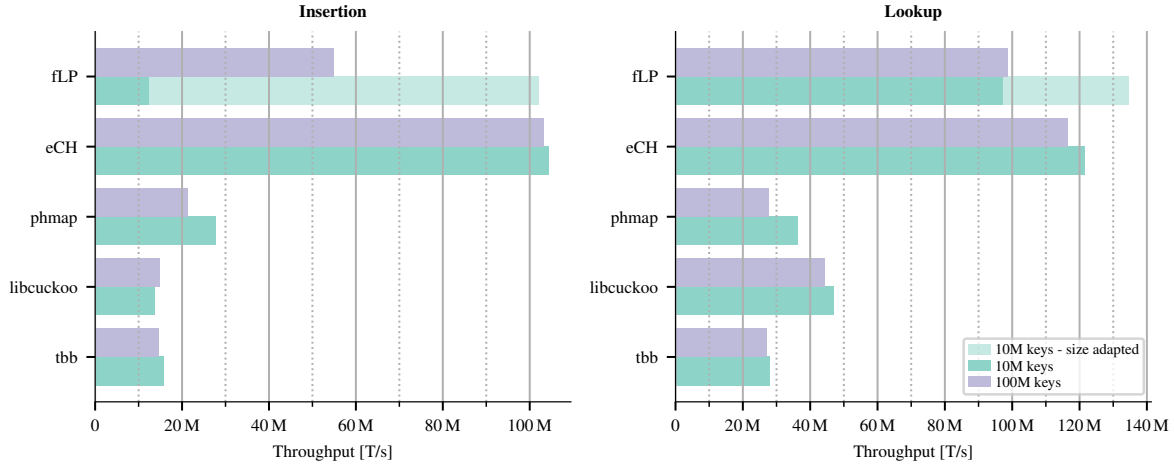


Figure 4.16.: Comparison of the performance for inserts and lookups of other parallel hash tables and our implementation running 8 threads.

Insertion. For our Chained Hash Table, we can see that the throughput is much higher than the implementations from the repositories and projects mentioned above.
However, our Linear Probing Hash Table shows different throughputs for the two varying sizes of our dataset. The higher throughput is achieved when adapting the hash table's size to the number of key-value pairs. With this insight, we can see that the size of Open Addressed Schemes influences the throughput the hash tables can achieve.

Lookup. Dealing with lookups, our implementation again outperforms the other hash tables for any workload. Furthermore, we can observe that for Open Addressed Schemes, it is important, not only for inserting but also for looking up keys, to select an adequate size for the hash table. By doing so, we can improve the throughput of the hash table by about 30 million keys per second. When inserting 10 million keys in the hash table with 2^{30} buckets, the lookup is slightly slower than when 100 million keys are inserted. This might be due to the storing of the data in the hash table, as when inserting more keys, these build clusters in the hash table when resolving collisions. Therefore, for some lookups, the data for the following lookup might already be loaded to cache, while for a small number of keys in a large hash table, it is very unlikely that many clusters build up.
Moreover, looking at the Chained Hash Table, each lookup in the hash table results in at least one cache miss, as pointer chasing is needed when traversing the linked lists. However, the length of the linked lists may be relatively short when inserting only a few keys into the hash table, which then positively affects the lookup throughput, as only a few elements need to be traversed. The throughput decreases slightly when the hash table holds 100 million keys, as now it is more likely that a collision occurred during the insertion, and thus the linked lists

are a bit longer than for the case with less inserted keys.

Nevertheless, these experiments show us that our implementation offers higher throughput for Chained Hashing for both inserts and lookups, as well as for the Linear Probing Hash Table when the size of the hash table is chosen appropriately.

5. Conclusions and Outlook

In this chapter, we want to discuss the results we got from the experiments, summarize them, and give a small guideline that simplifies the decisions when using hash tables. Therefore, we differ between hash table independent and hash table dependent insights.
With our experiments, we are confident to show the bearing of hash tables; however, no experiment can be complete enough to represent each hash table's behavior in every situation. Moreover, we hope that hash tables' users recall that choosing the wrong hash table can significantly affect performance and runtime.

5.1. Hash Table Independent Insights

All of the points listed here apply for all hash tables we looked at in our experiments.

Used hash function. When deciding for a hash function, the relevant points that need to be considered are to get the highest possible throughput and an excellent distribution of the keys. However, in our case, the Murmur hash turned out to be the hash function to choose as the throughput was higher than for the CRC hash, as Sect. 4.2.1 showed. Nevertheless, one might also need to regard that the Murmur hash, which we used, is a hash function designed for 64-bit integers only. As in several applications, e.g. in database management systems, this might not be enough, because strings need to be hashed, one has to go back to more generic functions like the CRC hash or to analyze the performance and correctness when combining several specific hash functions to hash any input type.

Data distribution. In the second part of Sect. 4.2.1, we looked at the length of the linked lists in a Chained Hash Table. The outcome of this experiment was that the data distribution of the keys, as announced in Sect. 4.1.2, does not have a relevant impact on the resulting behavior of the hash tables. This can also be seen when looking at the results in Fig. 4.9, 4.10, 4.12, and 4.14, as the plots in each column do not really differ.

Work partitioning. As the results in Sect. 4.2.2 present, Morsel-Driven parallelism pays off, as the overhead of communication can be decreased drastically. Though this approach needs further investigation, as the morsel sizes and the number of threads can impact the behavior of Morsel-Driven Parallelism. So, the adequate size must be determined, depending on the hardware and thread count, as the threads should interfere with each other as little as possible when getting the next morsel.

Returning results. In our experiments, we could decide between two possibilities how to indicate that a key could not be found. However, the performance results, we presented

in Sect. 4.2.3, clearly show that our first naive approach to throw an exception leads to an extreme slowdown of the throughput. Our second approach is more promising and allows more flexible implementations that do not require the C++ class.

5.2. Hash Table Dependent Insights

In this section, we want to make it easier for developers to decide which hash table to use. Therefore, we combine the knowledge from Sect. 4.2.4, 4.2.5, and 4.2.6.
In Fig. 5.1, we provide a guideline that can be used to determine the best hash table. With our knowledge, we gained from the experiments before, we can limit our decision making to the load factor and the unsuccessful ratio, as the data distribution of the input keys does not have an effect on the behavior of the hash table.

Size selection for Open Addressed Hashing. Before analyzing the different hash tables, the first point to note is that especially for Open Addressed Schemes it is very important to choose an appropriate size of the hash table. As shown in Sect. 4.2.7, we could improve the behavior of our Linear Probing Hash Table significantly for both insertions and lookups. Due to this, when using Open Addressed Hash Tables, the user has to investigate which size of the hash table fits best for the specific workload.

Unsuccessful ratios: 50%, 75%, 100%. Starting with an unsuccessful ratio that is higher than 50%, the choice of hash table should be **sCH** with Pointer Tagging. Although it is always slightly slower than **fLP** (up to a load factor of 90%) and **eCH** for the insertion, the lookup throughput remains nearly unchanged when altering the parameters, as Fig. 4.12 and 4.14 show. Thus, we can benefit from Pointer Tagging when performing the lookups in parallel and get a higher throughput than for the other hash tables.

Low load factors: 25%, 35%, 45%. Moving on to the low load factors and focusing on **RHH**, **eCH**, and **fLP**, Fig. 4.9 presents the throughput which could be achieved for our hash tables. From this graph, we can see for insertion that **RHH** performs half as good as **fLP**, while **eCH** is a bit above **fLP**.
Taking the lookup into consideration, Fig. 4.12 shows that **RHH** is higher than for the other two hash tables. We can also observe that despite the different unsuccessful ratios, the throughput is quite close to each other. The same holds for **fLP**, but its throughput is only half of **RHH**. **eCH** is the hash table, which is affected most by varying unsuccessful ratios, so for 100% unsuccessful queries, the performance is the same as for **RHH**, for 0% unsuccessful queries, it is a bit higher than half of **RHH**'s throughput.
Based on this knowledge, we can say that either **RHH** or **fLP** can be used. For the first case, the additional time needed for inserting the elements is amortized by the lookup time. In contrast, the higher performance for inserting amortizes the extra time for the lookups in **fLP**. If the lookup algorithm for **RHH** is improved and leads to even higher throughput, the decision should be **RHH**, as the lookup throughput compensates the slower insertion.

Medium load factor: 50%. Again limiting ourselves to **RHH**, **eCH**, and **fLP**, for this load factor, the decision is not very easy. Looking at the insertions in Fig. 4.10, we still see that **eCH** and **fLP** perform best up to 10 threads, while **RHH** again just has about half of the performance. However, Fig. 4.14 gives the important information to make the decision: for 0% unsuccessful ratio, the performance of **RHH** is nearly 1.7 times the performance for **eCH**. With this observation, we would advise choosing **RHH** for unsuccessful ratios of 0% up to 25%.
When the ratio is higher, the lookup performance of **RHH** does not correct the lower insertion throughput. Thus, for this case, we would tend to use **eCH**.

High load factors: 70%, 90%. For the last two load factors, we can see from Fig. 4.10, that the insert performance of **RHH** is the lowest. With 70% load factor, **fLP** slightly dominates **eCH**, while it is the other way around for 90%.
The lookup performance from Fig. 4.14 shows that for both load factor, **fLP** has the worst performance. With a 70% load factor, **RHH** and **eCH** have quite similar throughputs, while for 90%, **eCH** dominates **RHH** for nearly all workloads.
For these load factors, the only hash table that offers the same insert performance (or even a better one) is **fLP**, but the lookups are too slow. For **RHH**, it is the other way around, as the lookups are in the same range, but the inserts are too slow.
With these insights, the decision is to use **eCH** for both load factors, as it offers a very high insert throughput combined with a quite adequate lookup performance.

We created the following decision graph that presents the best hash tables for a specific workload with the explanations above.

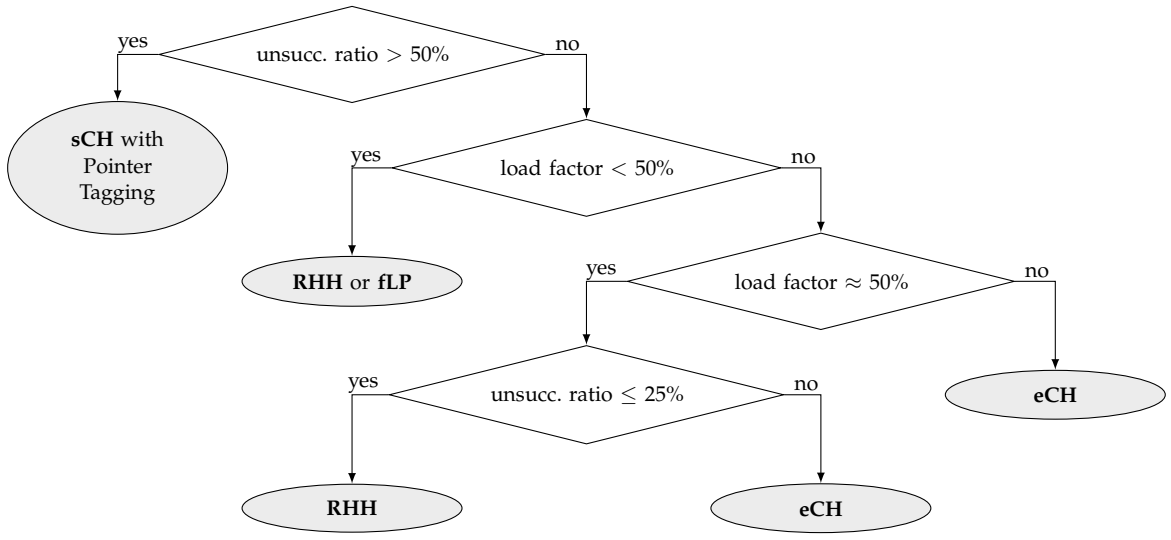


Figure 5.1.: Suggested guideline for Hashing Scheme selection.

5.3. Outlook

With the previous experiments, we limited ourselves to the Write-Once-Read-Many workload. Due to this, the synchronization strategies are also designed for this use case. In some applications, however, it is essential that mixed workloads can be executed. One future project could be to extend the strategies to allow mixed workloads and to analyze the performance of the proposed ideas.
Another point is to improve the lookup algorithm for Robin Hood Hashing by including the possibilities of early aborts to allow unsuccessful queries to stop before having scanned the whole cluster. Therefore, the ideas, Richter et al. proposed in [RAD15], can be used as they state several ideas of adapting the lookup algorithm.
Finally, parallelism cannot only be realized by thread parallelism, but also by data parallelism using SIMD instructions. An interesting point would be to explore a combination of both, so to run the execution multi-threaded and each thread itself works with SIMD instructions. For this, the hash tables' layout might need to be altered a bit, as well as the synchronization strategies need to be reconsidered.

A. Appendix

A.1. Cycles

Chained Hashing

Table A.1.: Cycles for Chained Hashing with different load factors of normal distribution combined with a varying number of threads and synchronization strategy. The abbreviations in the row labeled with *sync.* refer to the locking strategies mentioned in Sect. 4.2.4.1.

		loadfactor								
		25%			35%			45%		
sync.		**fCH**	**sCH**	**eCH**	**fCH**	**sCH**	**eCH**	**fCH**	**sCH**	**eCH**
threads	1	605.2	554.2	350.2	601.8	543.2	346.2	604.2	542.3	351.7
	2	602.2	560.7	349.3	608.7	560.1	350.4	608.0	559.1	348.4
	5	592.8	543.8	346.2	593.5	544.2	342.0	591.4	543.6	342.5
	10	606.5	558.5	359.3	606.0	557.9	358.9	606.5	558.9	359.2
	15	700.1	645.3	442.3	700.3	645.2	442.8	701.0	645.6	442.1
	20	770.4	705.0	502.3	770.0	704.5	503.1	770.4	705.6	504.7
		50%			70%			90%		
threads	1	606.1	547.6	346.5	604.3	547.3	351.1	604.7	556.2	347.5
	2	609.4	551.1	348.7	608.2	551.2	343.2	607.9	548.7	351.5
	5	592.3	544.7	345.2	593.0	544.3	343.4	592.4	544.3	341.5
	10	606.7	558.5	359.5	606.4	557.7	358.6	606.6	558.1	359.2
	15	700.6	645.8	442.4	701.2	645.2	441.2	701.9	645.3	442.4
	20	769.6	704.3	503.2	770.3	704.6	502.9	770.0	703.6	504.5

Linear Probing Hashing

Table A.2.: Cycles for Linear Probing Hashing with different load factors of normal distribution combined with a varying number of threads and synchronization strategy. The abbreviations in the row labeled with *sync.* refer to the locking strategies mentioned in Sect. 4.2.4.2.

		load factor					
		25%		35%		45%	
sync.		**fLP**	**eLP**	**fLP**	**eLP**	**fLP**	**eLP**
threads	1	371.7	438.9	354.5	402.1	350.5	397.3
	2	377.6	432.6	353.0	400.8	352.1	395.0
	5	378.8	433.6	355.5	399.8	352.7	396.6
	10	404.1	458.5	378.4	425.3	374.1	417.5
	15	536.3	610.4	508.0	566.8	506.1	560.9
	20	649.2	735.0	616.9	689.1	615.3	681.3
		50%		70%		90%	
threads	1	358.7	401.1	424.5	451.0	547.0	577.8
	2	358.1	401.2	426.5	452.4	546.0	587.5
	5	361.4	397.3	425.8	462.6	550.2	592.6
	10	382.2	422.2	453.8	482.9	586.6	630.3
	15	514.0	566.1	594.7	639.5	772.7	845.1
	20	623.6	687.4	706.6	767.2	923.6	1025.2

Robin Hood Hashing

Table A.3.: Cycles for Robin Hood Hashing for normal distribution depending on varying number of threads and load factors.

		load factor					
		25%	35%	45%	50%	70%	90%
threads	1	790.7	779.1	757.7	767.4	782.4	980.6
	2	796.4	778.1	763.6	761.3	774.7	981.5
	5	805.0	765.8	753.2	748.4	768.4	970.9
	10	837.0	791.5	778.6	778.4	796.2	1006.5
	15	961.8	903.5	882.7	878.7	905.4	1156.0
	20	1053.1	986.1	962.2	955.5	985.8	1274.5

A.2. Instructions

Chained Hashing

Table A.4.: Instructions for Chained Hashing with different load factors of normal distribution combined with a varying number of threads and synchronization strategy. The abbreviations in the row labeled with *sync.* refer to the locking strategies mentioned in Sect. 4.2.4.1.

		loadfactor								
		25%			35%			45%		
sync.		**fCH**	**sCH**	**eCH**	**fCH**	**sCH**	**eCH**	**fCH**	**sCH**	**eCH**
threads	1	52.1	49.0	41.9	52.0	49.0	41.9	52.0	48.9	41.9
	2	52.0	48.5	42.0	52.0	48.5	41.9	52.0	48.5	42.0
	5	52.1	48.2	42.0	52.1	48.2	42.0	52.0	48.2	42.0
	10	52.1	48.2	42.1	52.1	48.2	42.0	52.1	48.2	42.0
	15	52.2	48.2	42.3	52.2	48.3	42.3	52.2	48.2	42.3
	20	52.3	48.4	42.4	52.3	48.4	42.4	52.3	48.4	42.4
		50%			70%			90%		
threads	1	52.0	49.0	42.0	52.0	48.9	42.0	52.1	49.0	41.9
	2	52.1	48.5	42.0	52.1	48.5	42.0	52.0	48.5	41.9
	5	52.1	48.2	42.0	52.1	48.2	42.0	52.1	48.2	42.0
	10	52.1	48.2	42.0	52.1	48.2	42.1	52.1	48.2	42.0
	15	52.2	48.3	42.3	52.2	48.3	42.3	52.2	48.3	42.2
	20	52.3	48.4	42.4	52.4	48.4	42.4	52.4	48.4	42.5

Linear Probing Hashing

Table A.5.: Instructions for Linear Probing Hashing with different load factors of normal distribution combined with a varying number of threads and synchronization strategy. The abbreviations in the row labeled with *sync.* refer to the locking strategies mentioned in Sect. 4.2.4.2.

		load factor					
		25%		35%		45%	
sync.		**fLP**	**eLP**	**fLP**	**eLP**	**fLP**	**eLP**
threads	1	68.4	83.7	59.5	70.8	55.8	64.8
	2	68.6	83.1	60.3	70.7	56.3	64.3
	5	69.1	82.9	60.5	70.3	56.6	64.1
	10	69.3	83.0	60.7	70.4	56.8	64.0
	15	69.6	83.5	61.0	70.8	57.0	64.3
	20	69.8	83.8	61.1	71.0	57.1	64.5
		50%		70%		90%	
threads	1	55.0	63.1	56.8	62.2	87.6	88.4
	2	55.4	62.5	57.3	61.7	88.0	88.0
	5	55.7	62.3	57.6	61.4	88.3	87.7
	10	55.8	62.2	57.8	61.4	88.5	87.6
	15	56.0	62.5	57.9	61.5	88.7	87.7
	20	56.2	62.7	58.0	61.7	88.8	87.9

Robin Hood Hashing

Table A.6.: Instructions for Robin Hood Hashing for normal distribution depending on varying number of threads and load factors.

		load factor					
		25%	35%	45%	50%	70%	90%
threads	1	121.1	106.3	100.1	99.2	110.3	209.5
	2	121.0	106.2	100.1	99.1	110.1	209.5
	5	121.1	106.1	100.0	99.2	110.2	209.4
	10	121.5	106.3	100.2	99.2	110.3	209.5
	15	122.0	106.6	100.5	99.6	110.4	209.7
	20	122.4	106.9	100.7	99.7	110.6	209.8

List of Figures

List of Tables

List of Algorithms

Bibliography

[App] A. Appleby. *MurmurHash3 64-bit finalizer*. URL: https://github.com/aappleby/smhasher/wiki/MurmurHash3 (visited on 05/03/2020).

[Cli07] C. Click. "Lock-free hash table." Talk at JavaOne 2007. 2007.

[CLM85] P. Celis, P. Larson, and J. I. Munro. "Robin Hood Hashing (Preliminary Report)." In: *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 1985, pp. 281–288. DOI: 10.1109/SFCS.1985.48.

[FAK13] B. Fan, D. G. Andersen, and M. Kaminsky. "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing." In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. Ed. by N. Feamster and J. C. Mogul. USENIX Association, 2013, pp. 371–384.

[GGN15] V. R. Gad, R. S. Gad, and G. M. Naik. "Configurable CRC Error Detection Model for Performance Analysis of Polynomial: Case Study for the 32-Bits Ethernet Protocol." In: *Internet of Things, Smart Spaces, and Next Generation Networks and Systems - 15th International Conference, NEW2AN 2015, and 8th Conference, ruSMART 2015, St. Petersburg, Russia, August 26-28, 2015, Proceedings*. Ed. by S. I. Balandin, S. D. Andreev, and Y. Koucheryavy. Vol. 9247. Lecture Notes in Computer Science. Springer, 2015, pp. 529–542. DOI: 10.1007/978-3-319-23126-6_46.

[Int16] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A*. Intel Corporation. Sept. 2016.

[KC04] P. Koopman and T. Chakravarty. "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks." In: *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*. IEEE Computer Society, 2004, p. 145. DOI: 10.1109/DSN.2004.1311885.

[Knu63] D. E. Knuth. *Notes on "open" addressing*. Unpublished memorandum. (Memo dated July 22, 1963. With annotation "*My first analysis of an algorithm, originally done during Summer 1962 in Madison*". Also conjectures the asymptotics of the Q-function, with annotation "*Proved May 24, 1965*"). 1963.

[Koo02] P. Koopman. "32-Bit Cyclic Redundancy Codes for Internet Applications." In: *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. IEEE Computer Society, 2002, pp. 459–472. DOI: 10.1109/DSN.2002.1028931.

[KT89] G. D. Knott and P. de la Torre. "Hash Table Collision Resolution with Direct Chaining." In: *J. Algorithms* 10.1 (1989), pp. 20–34. DOI: 10.1016/0196-6774(89)90021-7.

[Lei+14] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by C. E. Dyreson, F. Li, and M. T. Özsu. ACM, 2014, pp. 743–754. DOI: 10.1145/2588555.2610507.

[Li+14] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. "Algorithmic improvements for fast concurrent Cuckoo hashing." In: *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. Ed. by D. C. A. Bulterman, H. Bos, A. I. T. Rowstron, and P. Druschel. ACM, 2014, 27:1–27:14. DOI: 10.1145/2592798.2592820.

[LPW10] A. Laarman, J. van de Pol, and M. Weber. "Boosting multi-core reachability performance with shared hash tables." In: *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. Ed. by R. Bloem and N. Sharygina. IEEE, 2010, pp. 247–255.

[RAD15] S. Richter, V. Alvarez, and J. Dittrich. "A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing." In: *Proc. VLDB Endow.* 9.3 (2015), pp. 96–107. DOI: 10.14778/2850583.2850585.

[RM19] M. Raasveldt and H. Mühleisen. "DuckDB: an Embeddable Analytical Database." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska. ACM, 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212.

[San+19] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0.

[Suz+06] K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota. "Birthday Paradox for Multi-collisions." In: *Information Security and Cryptology - ICISC 2006, 9th International Conference, Busan, Korea, November 30 - December 1, 2006, Proceedings*. Ed. by M. S. Rhee and B. Lee. Vol. 4296. Lecture Notes in Computer Science. Springer, 2006, pp. 29–40. DOI: 10.1007/11927587_5.

[VL11] S. van der Vegt and A. Laarman. "A Parallel Compact Hash Table." In: *Mathematical and Engineering Methods in Computer Science - 7th International Doctoral Workshop, MEMICS 2011, Lednice, Czech Republic, October 14-16, 2011, Revised Selected Papers*. Ed. by Z. Kotásek, J. Bouda, I. Cerná, L. Sekanina, T. Vojnar, and D. Antos. Vol. 7119. Lecture Notes in Computer Science. Springer, 2011, pp. 191–204. DOI: 10.1007/978-3-642-25929-6_18.