Scalable Hash Tables

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik des

Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Tobias Maier

Tag der mündlichen Prüfung: 23.07.21

1. Referent: Prof. Dr. Peter Sanders
Karlsruher Institut für Technologie
Deutschland

2. Referent: Julian Shun
Massachusetts Institute of Technology
USA

This document is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0): https://creativecommons.org/licenses/by/4.0/deed.en

Abstract

The term scalability with regards to this dissertation has two meanings: It means taking the best possible advantage of the provided resources (both computational and memory resources) and it also means scaling data structures in the literal sense, i.e., growing the capacity, by "rescaling" the table.

Scaling well to computational resources implies constructing the fastest best performing algorithms and data structures. On today's many-core machines the best performance is immediately associated with parallelism. Since CPU frequencies have stopped growing about 10-15 years ago, parallelism is the only way to take advantage of growing computational resources. But for data structures in general and hash tables in particular performance is not only linked to faster computations. The most execution time is actually spent waiting for memory. Thus optimizing data structures to reduce the amount of memory accesses or to take better advantage of the memory hierarchy especially through predictable access patterns and prefetching is just as important.

In terms of scaling the size of hash tables we have identified three domains where scaling hash-based data structures have been lacking previously, i.e., space efficient growing, concurrent hash tables, and Approximate Membership Query data structures (AMQ-filter). Throughout this dissertation, we describe the problems in these areas and develop efficient solutions. We highlight three different libraries that we have developed over the course of this dissertation, each containing multiple implementations that have shown throughout our testing to be among the best implementations in their respective domains. In this composition they offer a comprehensive toolbox that can be used to solve many kinds of hashing related problems or to develop individual solutions for further ones.

DySECT is a library for space efficient hash tables specifically growing space efficient hash tables that scale with their input size. It contains the namesake DySECT data structure in addition to a number of different probing and cuckoo based implementations. Growt is a library for highly efficient concurrent hash tables. It contains a very fast base table and a number of extensions to adapt this table to match any purpose. All extension can be combined to create a variety of different interfaces. In our extensive experimental evaluation, each adaptation has shown

to be among the best hash tables for their specific purpose. Lpqfilter is a library for concurrent *approximate membership query* (AMQ) data structures. It contains some original data structures, like the linear probing quotient filter, as well as some novel approaches to dynamically sized quotient filters.

It seems to be at the time to say **thank you** to a lot of people. First and foremost I want to thank my family who have always been there when I needed support or advice. To my mother and father, together they have shown me what it means to have each others' backs. To my brothers in hard times—mine and yours—having them has helped me a lot. To my wife—you're the best—who had to sit next to me throughout the pandemic and is probably even happier than me that the stress of the past weeks is ending now.

I want to thank Peter for being a great supervisor and never pressuring me too hard—he would have given me the opportunity to work on this dissertation even longer—and for giving me all these opportunities like sending me to Hawaii and Sweden. I also want to thank Julian for agreeing to review this dissertation it is nice that people voluntarily read my work.

Thank you to all my current and former colleagues. They have made my time working on this dissertation truly memorable: to Christian for hiring me as a tutor long before I worked here; to Daniel Funke for being the first person to actually use one of my hash tables; to Darren for the great time we had on Hawaii; to Demian for always being up for anything, I truly appreciate that; to Dominik for his funny song lyrics; to Jochen for teaching me to guess running times; to Lorenz for all the fun activities outside of work and around pizza ovens; to Lukas for the short time we shared an office before heading to home office; to Marvin for his willingness to play any board game I bring to board game night; to Matthias for him sometimes loosing at pool; to Michael for writing the best work emails; to Sascha for all the work he put into supervising the C++ lab course; to Sebastian Lamm for all the non-work talk about painting and games; to Sebastian Schlag for the most legendary parties; to Simon for his engaging and positive attitude; to Timo for all the fun activities including either pesto or LEDs or both; to Tobias Heuer for being fine with nicknames because I like my name; to Tomáš for his dry sense of humor at the most unexpected times; to Yaroslav for sharing an office with me even though I bought blue switches; to all the new guys Markus, Florian, Daniel Seemaier, and Hans-Peter I'm looking forward to finding out what I would have written here.

A special thank you to the people who have read this dissertation in an unfinished state, helping me edit this "mess" Demian, Tobias, Sebastian, Marvin, Hans-Peter, and Lorenz.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

A hash table is a data structure that stores elements associated with their keys. After an element and its key have been inserted, the original element can be retrieved (efficiently) using only the key. Hash tables are some of the most common data structures in research codes, industry applications, and also small (private) software projects. Their invention dates back to the 1950s, where multiple groups came up with the concept independently [37]. Since then they have become ubiquitous in many programming contexts. As such there are default hash table implementations in all common programming languages, e.g., C++s `std::unordered_map` in the Standard Template Library (STL), rust's `HashMap` from the rust standard library, and java's `HashMap` in the utils library. Some programming languages—like python, ruby, and perl—even have integrated support for hash tables.

associative storage

ubiquity in all areas of coding

every programming language has access to one

The term scalability can have multiple meanings. An online dictionary defines the term as follows:

> **scalability**
>
> **1** the capacity to be changed in size or scale. [. . .]
>
> **1.1** the ability of a computing process to be used or produced in a range of capabilities. [. . .]
>
> Lexico [40] (retrieved 20. May 2021,lexico.com/en/definition/scalability)

The entry identifies two aspects to the term scalability. Scaling in size and scaling to computational resources. This dyadic nature is at the core of this dissertation. A central focus of our work has been to provide fast and efficient hash table architectures that can scale their capacity in a scalable (efficient) manner. The basic methodology of growing a hash table's capacity, by allocating a new table and

scaling in size and scaling to resources migrating elements from the old table, is taught in most algorithm classes or beginner level textbooks, for example in “Algorithms and Data Structures: The Basic Toolbox” [55] (page 85, Exercise 4.5) or “Introduction to Algorithms” [16] (Section 17.4.1). This same migration technique is also implemented in all of the default library hash tables mentioned above. However, very little research exists into efficient migration algorithms. With our research we try to close this gap by designing tables with efficiency and adaptability in mind.

migration is well known

One strength of hash table data structures is that they inherently scale really well to the size of input problems. All basic operations have expected constant running times. In fact this is one of the main draws to hash tables and one of the reasons for their ubiquity (e.g., compared to search trees). Nonetheless, for algorithms to scale well to larger inputs compute power has to scale with the size of the problem. However, for the past 10-15 years increasing compute power has only come in the form of increased processor counts. Thus, algorithms scaling well with compute power means they have to scale well in multi-processor environments. For data structures like hash tables scaling to multi-processors necessitates concurrency, i.e., data structures that can be accessed by many cores in parallel. Hence, concurrent data structures are a major topic throughout this dissertation.

scaling to problem size is an inherent strength

concurrent data structures for scalable algorithms

The performance of one hash table operation is actually not that dependent on the CPU frequency. It depends more on the speed of a memory accesses. As such hash tables have to be designed to minimize the amount of necessary memory accesses. One important technique to achieve this goal is to take advantage of memory prefetching (loading memory into the cache before it is actually accessed, see Section 2.5.1) opportunities as much as possible—both automatic prefetching and manual prefetching. Consequently, hash tables can only scale well if the memory connection can support the amount of data that is being transmitted. Overall, we see that scalable hash table implementations depend on scaling both CPU and memory resources to be able to scale to larger problems.

memory access is more important than CPU speed

Another kind of scalability that one could take a look at is scaling well to the requirements of different application, i.e., having flexible solutions that adapt well to any given set of problems. This can either mean creating one data structure that is a *jack of all trades* and can solve any problem. Or it can mean having a *toolbox* full of different specialized tools and techniques that can be combined to fit all needs.

jack of all trades vs. toolbox implementations

Hash tables are ubiquitous in programming, they are used in all kinds of different applications from personal projects to important industry projects. Consequently, the requirements are as diverse as the application domains. Possible requirements could be, dynamic table size (i.e., growing), memory efficiency, small tables, large tables, or even specific workload patterns, and many more, combining and juggling all of these potentially contradicting requirements leaves a lot of potential performance on the table. Therefore, true scalability can only be achieved by a set of specialized solutions that can take advantage of an applications specifications.

for true scalability fit hash tables to specific application

1.1 Overview and Contributions

Throughout this dissertation we describe a set of hash-based data structures, split into 3 libraries according to three different areas of use. Together, these hash tables can fit into all kinds of different applications. All described hash tables follow the same general interface—simplifying drop in replacement without unnecessarily changing the context of table accesses. This is of course especially true for all implementations within one library but we also paid special attention that inter-library-compatibility is possible.

Space Efficient Hash Tables In Chapter 3 we describe the domain of space efficient hashing. Prior to our work in this area, there was no notion of *dynamic space efficiency*, i.e., holding tight memory bounds even in cases where the table has to be migrated to fit all elements (Section 3.3). Dynamic space efficiency is an important concept because it is the only way to guarantee space efficiency in cases where the final number of elements is not known at the construction time of the table.

dynamic space efficiency

In addition to the theoretical groundwork we also show two approaches for implementing dynamic space efficient hash tables. The first approach is a blueprint for adapting existing hash table architectures to make them dynamically space efficient. The idea is to grow the table in-place without using any additional memory (see Section 3.4). We used this technique to implement 4 different (dynamically) space efficient hash tables. The second approach called *DySECT* (**Dy**namic **S**pace **E**fficient **C**uckoo **T**able; Section 3.5) is specifically designed for this use case. It

adapting existing hash tables

DySECT

remains space efficient by growing small subsections of the table and using a rebalancing mechanism that is based on cuckoo hashing (a popular technique for space efficient hash tables Section 2.4.2, page 37) to make use of the created slots.

In our experiments (Section 3.8) both approaches succeed at offering well scaling solutions, even for very large fill degrees of 97 % and above.

Concurrent Hash Tables In Chapter 4 we look into concurrent hash tables. *folkloreHT* The central theme of this chapter is folkloreHT a fast concurrent hash table implementation that was not developed by us but seems to be "folklore". However, folkloreHT, is very specialized and thus is not usable in many workloads, it cannot grow, supports only word sized keys and values, and does not support deletions.

lifting restrictions Throughout Section 4.4 we develop scalable techniques to lift these restrictions. In particular we design a table migration that allows the table to be grown cooperatively by all threads operating on the table. This migration is notable because *scalable migration algorithm* during the migration there is little to no interaction between threads which makes it very efficient in practice. These techniques (for lifting restrictions) can be combined freely, thus leading to a flexible solution that can be adapted individually to specific use cases.

extensive experiments with up to 17 hash tables In Section 4.6 we executed an extensive experimental evaluation of folkloreHT and its extensions. For these tests we used multiple input types—uniformly random, skewed inputs, and real world string data from the Gutenberg project—as well as different access patterns—insert, find, insert-or-update—on two machines to compare 17 different hash table implementations from 7 different libraries. In all of these tests, our implementations either folkloreHT or its generalizations were among the top performing data structures. They commonly outperformed all competitor tables by at least a factor of two.

Concurrent Quotient Filters In Chapter 5 we take a look at quotient filters. A quotient filter is an *Approximate Membership Query* data structure (AMQ filter) that works similar to a hash table but stores fingerprints instead of actual elements. The base quotient filter described by Bender et al. [8] uses 3 status bits per slot. We have designed a number of different data structures changing or adding to the *proposed 6 variants* use of status bits. Overall we propose 6 variants: sequential 2BQ-filter and sequential LPQ-filter in Section 5.4.2, the concurrent LPQ-filter, the concurrent locally locked quotient filter and the concurrent 2BQ-filter in Section 5.5 (infeasible from practical point of view), and the fully expandable quotient filter in Section 5.6.

Noteable among these are: (1) The *concurrent LPQ-filter* because it is a lock free data structure which is uncommon for concurrent quotient filters (see page 151). This is also the variant that performs best in our experiments. (2) The *locally locked* quotient filter because its locking technique allows us to avoid many instances of locking, and because we have implemented a version of this filter that allows for limited growing. This limited growing variant is also a building block for the concurrent fully expandable quotient filter. And (3) the *fully expandable quotient filter* because it can grow arbitrarily while holding a bounded false positive rate.

noteable:
- concurrent LPQ-filter
- locally locked Q-filter
- fully expandable Q-filter

In our tests (Section 5.7) we use two machines to compare these variants against 4 other AMQ-filters. During these tests, both the LPQ-filter and the locally locked quotient filter outperform most other variants especially on lower fill degrees and on successful finds. They both scale well with the number of threads, even in multi-socket NUMA scenarios. We also test two variants of the fully expandable quotient filter against the locally locked quotient filter with the limited growing technique growing up to two orders of magnitude.

2 Hashing Fundamentals

In this section we present our definition of hash-based data structures and we establish the necessary notation that is used throughout this dissertation. Additionally, we explain some concepts and interfaces that are inherent to the presented data structures.

Definition

We define a *hash-based data structure* to be any data structure that stores information associated with keys within a table, where the position of this data is at least in part dictated by a pseudo-random mapping from keys to the slots of the table.

hash-based data structure

The goal of hash-based data structures is usually to achieve operation times that are independent of the number of maintained elements (at least in expectation). Instead the running times usually depend on the fill degree of the table.

To describe the objects that are stored within a hash-based data structure we use the two words *element* and *key*. The subtle difference is that the key is the part of the object that is used to derive the position in the table. One could say the key describes the elements distinguishing features. Two objects that have the same key are considered to be the same object for the purpose of a hash-based data structure. Additionally, a lot of hash-based data structures can only contain one object with the same key. An element can be more than just its key. It usually contains the key, but it can also contain other data. An easy example where elements and keys are different from one another would be during an aggregation like word count. The key would be the word itself. The element that is stored in a hash table would be the word together with its occurrence-counter.

element and key

Throughout this dissertation, we use n to denote *numbers of elements*. Usually this is the number of elements inserted into a data structure. But, depending on

number of elements n

the context n could also be used as the cardinality of other sets of elements, e.g., number of elements in one batch of operations or in one subsection of the data structure. Whenever necessary we use indices to specify the exact meaning of the variable. As described in the definition above, a hash-based data structure stores its data within a table. In this context we use *table* to denote any basic data structure that supports $O(1)$ access time to a set of m numbered *slots* $(a_0 \ldots a_{m-1})$ that each hold the same amount of data, like an array or an unbounded array. Depending on the hashing technique different table architectures are possible, e.g., a table consisting of multiple subtables or a table that stores elements indirectly using pointers to the real data.

table has m slots

The position where data is stored has to be retrieved when looking for the element. Thus it is very common that each element is associated with one slot of the table. We call the associated slot its *canonical slot*. Section 2.3 describes how to compute the canonical slot given the key of an element. Similarly, each operation also has a canonical slot—the canonical slot of the key that is being queried or inserted. If there is another element already stored in the canonical slot, we call this a *collision*. Different hashing schemes have different ways to resolve collisions (see Section 2.4). The running time of each operation is usually dependent on the number of collisions. Usually operations are at their fastest when their canonical slot is empty. Therefore, the *fill degree* $\delta = n/m$ of the table is important. For all hash-based data structures where data is stored directly in the table, the fill degree indicates the probability of a random slot being empty $P[\textit{slot empty}] = 1 - \delta$, i.e., the canonical slot of an element that has not yet been inserted.

canonical slot

fill degree $\delta = n/m$

2.1 Common Hash-Based Data Structures and Their Interfaces

Given the above definition, there are three common hash-based data structures *hash tables*, *hash sets*, and *AMQ-filter*. In this section, we describe their functionality, and their interfaces. We also show what they might have in common or what differentiates them, i.e., where common techniques can be reused or where specializations are necessary.

2.1.1 Hash Table

Hash tables are probably the most well known hash-based data structure. They solve the *"dictionary problem"* and are therefore also called dictionary data structures. A dictionary is a set of data elements, where each element consists of a key (a word) and its value (its definition). The goal of a dictionary is to offer a fast look up, i.e., to retrieve elements given their key. Thus, dictionaries implement the following functionality (*insert, find, remove*): given an element *insert* adds this element to the dictionary; given a key *find* retrieves the element/value associated with the key—if such an element was inserted previously; given a key *remove* erases elements with the provided key. Find operations are often also called *queries*. Queries can be either successful, i.e., the element is present and was found or unsuccessful, i.e., the element is not present and thus was not found, sometimes especially when talking about our experiments we also say positive and negative queries.

dictionary problem

insert, find, and remove

query

Hash tables closely follow the characteristic scheme for hash-based data structures described above. Elements are stored in a large table. Whenever possible, we store an element in its canonical slot. The canonical slot is computed using a hash function (see Section 2.3). If an element is inserted and its canonical slot is already filled then a *collision resolution technique* is used to find another place to store the element (see Section 2.4). Find and remove operations get the key of the queried element and use the hash function to compute the same canonical slot. Then they look for the element in its canonical slot (and eventual alternative slots depending on the collision resolution technique).

canonical slot

collision resolution

Hash tables are usually constructed with an *initial capacity* (i.e., table size). Once the table is filled, operations become increasingly slow until the table is completely filled and new elements cannot be inserted. Therefore, it is common to include a mechanism to increase the table's capacity. The procedure is similar to growing an unbounded array, a new larger table is allocated and all elements from the original table are moved to the new table. We say the elements are *migrated* to the new table. This migration usually takes linear time in the size of the original table. Thus it can easily be amortized (if the capacity is doubled $O(m)$ time for $O(m)$ new slots). This migration is ordinarily hidden from the user of the hash table (similar to unbounded arrays). Most implementation contain really simple growing algo-

initial capacity

migration

rithms like this. However, throughout this dissertation we will see that there is a lot of potential for optimized migration techniques.

Hash tables are essential to many algorithms like aggregations, map-reduce type reductions, lazy dynamic programming, and many operations on collections of elements (i.e., unification, occurrence counting, intersection). Additionally, they are also part of many common data structures like dictionaries, indexes, and set representations. Because hash tables are this important, there are hash table implementations in many basic programming libraries, e.g., `std::unordered_map` in the Standard Template Library (STL) for C++, the `HashMap` from the rust standard library, and the `HashMap` in java's utils library. There are even programming languages—like python, ruby, and perl—that have integrated support for hash tables. All these hash tables have different implementations and also different interfaces.

representing elements

One choice that stands in the beginning of the interface design is the representation of elements. All major implementations of hash tables represent their elements as *key-value-pairs* $x = \langle k, v \rangle$ where k is x's key and v is its value. But in the literature there is also another common notation. In this notation, we use a *key-extractor function* $k = key(x)$ to obtain the key of a given element. Here the element can be viewed as a black box without any structure, as long as the key can be extracted efficiently. Throughout this dissertation, we use this "key-extractor notation" because it is more in line with the other presented hash-based data structures (i.e., hash sets and AMQ-filter) and because it allows for more flexibility. Whenever it is unambiguous we use x and $key(x)$ interchangeably, e.g., $h(x) \Leftrightarrow h(key(x))$ when using the hash function to map the element to the table.

key-value-pair

key-extractor function

element access interfaces

The second major difference between hash table interfaces is the way they allow access to the elements, e.g., after a query. There are three main concepts: return by value, by reference, or by iterator . Each method comes with its own strengths and weaknesses.

return by value, by reference, or by iterator

Return by *value*: When returning by value, the query does not give any access to the stored element. Instead, the value is copied from the element and returned. When the table uses return by value there is often an update operation that queries the table and changes the value that is stored with a certain key. This interface is often used by scientific codes (because it is simple to implement and is less prone to errors) or by concurrent libraries, e.g., libcuckoo [41]. However, the necessary copies can cost time and the necessity of an update function is cumbersome.

Return by *reference*: A reference is a pointer like "object" that allows accessing the underlying element. When a query returns a reference to the stored element, users can freely access and change the stored element. Any changes become visible to future queries of the same element. Reference based interfaces are used within java's utils library both `HashMap` and `ConcurrentHashMap` return by reference for non-trivial value types. Java in general, only allows passing complex data types per reference.

Return by *iterator*: An iterator is similar to a reference in that it gives access to the stored elements. In addition to the element itself, iterators also give access to other stored elements. Other elements can be accessed by "iterating" through the table (i.e., incrementing the iterator). Iterator-like interfaces are common in C++ for example in `std::unordered_set`, which is the hash table in the C++ standard template library, and in `concurrent_unordered_map`, which is part of the Threading Building Block (TBB) library developed by Intel.

Whenever references and or iterators are passed out of the hash table, *referential integrity* becomes an important topic. Referential integrity specifies what actions invalidate previously returned references and or iterators. A reference returned by a query can be viewed as a pointer to the slot that holds the queried element. Therefore, any operation that removes the elements from its slot may change the target of the pointer such that it does not point to the queried element. The most obvious example is deleting the queried element from the hash table. In this case, the deletion invalidates all references to the deleted element. However, there might be less obvious operations that invalidate references like moving an element within the table. This can be problematic when using certain collision resolution techniques like cuckoo and Robin Hood hashing that rely on moving elements within the table. In such tables references could be invalidated when new elements are inserted.

referential integrity

Usually hash tables operate as set-like data structures, meaning that each key can only be inserted one time. If a key that is already stored in the hash table is inserted for a second time, the insertion fails. Many hash table implementations even return a reference (or iterator) to the previously inserted element. A hash table that accepts multiple elements with the same key is called a *multi-hash table*. Multi-hash tables

keys are unique

multi-hash table

can often be implemented similar to normal hash tables, especially if the number of repeated keys is small, but some collision resolution techniques have inherent problems with repeated keys.

The interface of deletions is more or less the same between different libraries, but some underlying hashing implementations have problems implementing deletions. Thus, there exist implementations that do not allow any deletions (for an overview of different hashing techniques and their capabilities see Table 2.1). Using a reduced hash table like this could still be beneficial, since a lot of applications do not actually use delete operations.

Many hash tables, even ones that do not return by iterator offer a method to iterate over the table, i.e., executing a loop that visits each contained element. This could for example be used to insert the elements into another data structure, for example at the end of an algorithm. Or to do something with them while they remain in the table. Usually, iterating over all elements takes $O(m)$ time (Note depending on fill degree of the table m might be significantly larger than n).

Overall, all different hash table interfaces have different advantages and disadvantages that can be used to implement algorithms in different ways. As we will see throughout this dissertation, different hash table interfaces also have an influence on the underlying hash table implementation and its performance. Some implementations (i.e., collision resolution techniques) lend themselves better to certain interfaces. Some information on the inherent advantages and disadvantages of collision resolution techniques concerning interfaces can be found in Table 2.1 (see Section 2.4).

2.1.2 Hash Set

In a hash table elements consist of their key which is their unique identifier and some associated data that can be retrieved when querying said key. However, a very common use case for hash-based data structures is representing a set of elements. Here, the additional data per element is often unnecessary—it is only important whether a certain element is present (e.g., for unification). Thus, an element should be represented by its key alone. In applications like this *hash sets* can be the data structure of choice. A hash set is basically a hash table that only stores keys without

element = key

any associated data. The overall construction of the data structure remains the same. Elements are stored within a large table in their canonical slot or in a slot that is retrievable from their canonical slot using a collision resolution technique.

Even though the construction remains similar to that of a hash table, there are different requirements for the underlying implementation of a hash set. Keys cannot be changed after being inserted, therefore, returning by reference or iterator is usually not necessary. Instead, queries usually only return **true** or **false** indicating whether the queried key was inserted or not, thus removing the necessity of worrying about referential integrity. We call this form of query a *contains query*. A hash set still stores all inserted keys. Thus iterating through inserted elements remains a possibility.

contains query

iterating through hash set

Elements in a hash set are usually much smaller then they would be in a hash table (since they consist only of their key), thus, hash table implementations with additional per-element-data (e.g., pointers for chaining) have larger relative overheads. Throughout this dissertation, we mainly focus on the efficient implementation of the generalized common hash tables (with additional data). However, since none of our techniques use any unnecessary overheads, they should be well suited to implement hash sets or other similar data structures.

2.1.3 AMQ-filter

Approximate Membership Query Data Structures (abbreviated with AMQ filter) are similar to hash sets, in that they represent sets that do not store any additional data per element. Thus, AMQ-filters also use contains queries—sometimes also called membership queries. The main interface difference compared to hash sets is that AMQ-filter queries return approximate results (i.e., their result can be wrong). An *approximate contains query*, can return a *false positive*, but not a *false negative*. Meaning that an approximate contains operation may return **true** even if the element was not inserted (false positive), however, it cannot return **false** when querying an element that was inserted (no false negative). Each AMQ-filter has a certain *false positive rate* p^+ that describes the probability of returning **true** when querying a "random" uncontained element. It should be noted that the answer of an AMQ-

approximate
$\Rightarrow$false positives
$\not\Rightarrow$ false negatives

false positive rate

filter is usually deterministic, thus, repeated queries of the same element always return the same false positive.

Allowing false positive queries can reduce the necessary memory to a few bits per element. While a hash set has to store the whole (potentially large) key, AMQ-filters get away with storing only small sketches, sometimes called *fingerprints*. Because of their small size, AMQ-filters are often used as summary data structures for slower more complex data structures like databases or web based services—accelerating the overall performance, by shortcutting negative queries. A good example would be safe browsing. Safe browsing is a service that displays a warning when accessing a potentially dangerous phishing website. To do this the browser regularly downloads an AMQ-filter from a trusted source (e.g., Google) whenever a website is accessed, its address is first checked with the AMQ-filter. Positive queries are then rechecked using an online blacklist. Using the AMQ-filter, most queries can be answered offline and without downloading the whole blacklist.

fingerprint

summary data structure

Both the capacity and the false positive rate p^+ of a quotient filter are parameters of the quotient filter construction. Because the keys of inserted elements are not stored explicitly, it is usually impossible to decrease the false positive rate after the filter is constructed. This can also be a problem when trying to increase the capacity of an AMQ-filter. Traditional reallocation and migration techniques are usually impossible, because the original keys cannot be reconstructed. The false positive rate of the AMQ-filter increases with every inserted element. It should reach the given (initialized) false positive rate when the given capacity is reached.

2.2 Concurrent Data Structures

Throughout this dissertation, we also look at some concurrent data structures. *Concurrent data structures* are data structures that are meant to be accessed by many threads at the same time. Operations on the data structures should be as independent as possible—forcing no unnecessary serialization.

concurrent data structures

Throughout this dissertation we use p to denote the *number of threads* operating on a data structure. We assume that each application thread has its own designated hardware thread or processing core. For the purpose of algorithm analysis, we assume that n is significantly larger than p^2—this allows us to simplify algorithm

number of threads p

we assume $n \gg p^2$

complexities by hiding $O(p)$ terms that are independent of n and m in the overall cost. We feel that this is safe to assume for most problems where hash-based data structures constitute a significant part of the running time.

Correctness

The most important requirement for concurrent data structures is that the data structure should always be in a *consistent state*, e.g., there should never be the possibility of a thread accessing an element in an unfinished state or accessing an element that was previously deleted. It should also be guaranteed that once an operation is finished its effects should be visible to all threads, e.g., after inserting x with thread a, x should be found by all future queries. One way to guarantee this, is to prove the *linearizability* of a data structure. A data structure is linearizable if each possible sequence of concurrent operations can be reordered in a way such that executing them sequentially in that order will result in the same outputs—without reordering two operations of the same thread. This can be achieved by guaranteeing that all changes caused by an operation are executed atomically at one point in time between their invocation and their return. This moment is called the *linearization point* of an operation.

consistent state

linearizability

Progress Guarantees

Outside of these consistency guarantees, concurrent data structures can also offer *progress guarantees*. For example, we say a data structure is *non-blocking* if blocking a thread that is accessing the data structure can not prevent other threads from making progress. A data structure is *lock-free* if it is non-blocking and guarantees global progress, i.e., there must always be at least one thread finishing its operation in a finite number of steps. It could be possible that one progressing thread repeatedly prevents another thread from doing so—e.g., through a compare-and-swap loop. The strongest progress guarantee is *wait-freeness*. A data structure is wait free when each operation on said data structure can always be finished in a finite number of steps, independent of all other threads.

non-blocking

lock-free

wait-free

Problems

ABA-problem
concurrent deallocation

There are two problems that are inherent to concurrent data structures (especially lock-free data structures): the *ABA problem* and the problem of *concurrent deallocation*. Both of these problems are encountered multiple times throughout this dissertation.

The *ABA problem* can be summarized as follows. It is hard to read a consistent state of multiple data entries if those entries are changed concurrently. For example, let us assume we have two memory slots s_1 and s_2 and one processor p supervising both slots. The processors task is to wait until s_1 and s_2 contain the same value. Thus p repeatedly loads s_1 and s_2 into its memory m_1 and m_2 but since s_1 and s_2 cannot be read at the same time there is time that passes between acquiring m_1 and m_2. Thus, even if m_1 and m_2 are equal p cannot be sure that s_1 was not changed in the meantime. Rechecking s_1 might also not work because both elements could have been flip flopped repeatedly in the meantime.

rechecking is not the solution

Solving the ABA Problem usually involves careful arguments how to ensure that slots have not been changed (e.g., locks, monotony arguments, epoch counter) or arguments why knowing a fully consistent state is not actually necessary.

The *concurrent deallocation problem* is related to the ABA problem. The idea here is that if there is a pointer to some element we cannot deallocate that element unless no other thread is using it. However, it is hard to ensure that this is the case. Common techniques to solve this problem are locks, counting pointer, and hazard pointer [57]. Although only hazard pointers are actually lock-free. Another way to enable safe deallocations is called *quiescent state based reclamation* (qsbr). To use qsbr threads that are not in any critical section have to call a function (from time to time). We can safely deallocate a pointer if this pointer is not publicly available and every thread has called the qsbr function at least once (since the pointer was last publicly available).

2.3 Hash Functions

hashing elements to slots

In this section, we describe how elements are mapped to their canonical slot. The mapping uses a hash function, therefore, we also say elements are *hashed* to their canonical slot. A hash function is a mapping $h : U \mapsto H$ from a universe U of potentially variable sized keys to a fixed size result space H. A common assumption is that $H = [0..m-1]$ such that a direct mapping to table slots is possible. However, for the purpose of this dissertation it is sometimes necessary to think of hashing as a two step process. First the key k is hashed to a number in the *intermediate hash space* $H_{int} = [0..2^\ell - 1]$ using the *preliminary hash function* $h_{pre} : U \mapsto H_{int}$. Here $|H_{int}| = 2^\ell$ is significantly larger than the table size—usually $\ell = 64$ or 32. Then the resulting number is mapped to the range of the table using the *table mapping* $r_{map} : H_{int} \mapsto H$. This two step process is important whenever the table size is adapted, because, when a table is using the same preliminary hash function h_{pre} and a similar mapping r'_{map} we can deduce the approximate position of an element in the new table by its position in the old table.

$h(k) = r_{map} \circ h_{pre}(x)$

The hash function only operates on keys not necessarily on full elements, i.e., $key(x)$ not x. However, it is usually unambiguous to use x and $key(x)$ interchangeably, e.g., $h(x) \Leftrightarrow h(key(x))$ when using the hash function to map the element to the table. This allows us to simplify some of the notation.

2.3.1 Preliminary Hash Function

The goal of the preliminary hash function is to distribute elements around the table evenly, without creating artificial clusters. There are different types hash functions that are commonly used in applications. From the identity function—reinterpreting the bits of the key as a number[1]—to cryptographic hash functions like SHA-1 or MD5, different tasks require different hash functions. Simple hash functions like the identity could lead to artificial clusters when input keys are not sufficiently random (i.e., similar elements map to the same area of a table). Stronger hash functions on the other hand can be slow to compute and often have unnecessarily large result spaces. SHA-1 and MD5 for example are well known cryptographic hash functions. They map to a large result space of 160 bits and 128 bits respectively because their main task is to make it difficult to find hash collisions (two inputs that map onto the same output). All this is not necessary when mapping to the table of a hash-based data structure (assuming non-malicious users). The tables are usually smaller and

between identity and SHA-1

[1]only possible if the key space is small

can be addressed using 32 bits or 64 bits. Additionally, they are not usually filled by adversaries looking for collisions. Instead it is only important that the number of collisions remains reasonably small when inserting elements that contain at least some "randomness" (i.e., from an appropriate distribution). There are still some requirements that a hash function has to fulfill to be usable in a hash-based data structure.

requirements:
- uniformity
- speed

The result of the preliminary hash function should use the whole result space H_{int} evenly, i.e., the hash function should "randomize" all bits of the output independent of the input space, i.e., even when hashing small numbers it should be possible to get a large hash value. This is important because depending on the mapping function (covered later) either the most significant or the least significant bits are used for the mapping to the table. Furthermore, it is important that the hash function is fast to evaluate. The hash function is evaluated at least once per operation on the table. A slow hash function could significantly impact the performance of the data structure.

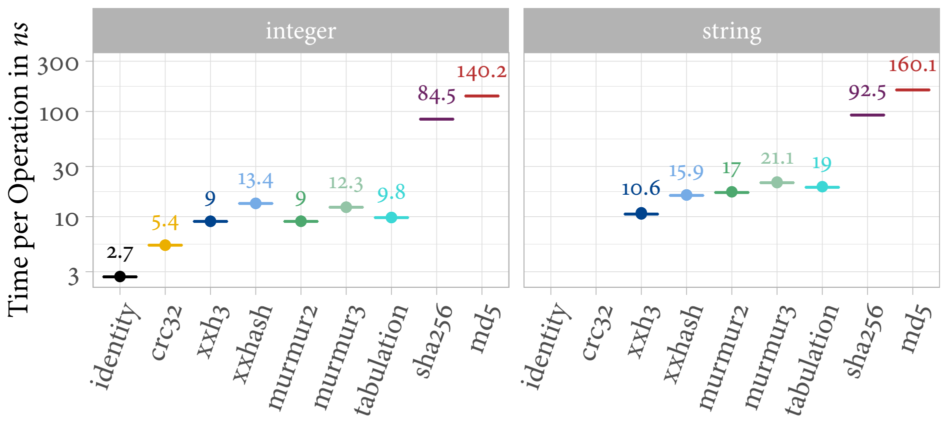

Figure 2.1: Comparing the performance of different hash functions. Measured over two kinds of inputs: *(left)* 64 bit integer data, *(right)* string data (The King James Bible).

pre-hash benchmark

To evaluate the speed of different hash functions, we set up the following small benchmark—Figure 2.1. First we construct a set of keys, either integer keys (i.e., 10^7 random 64bit numbers) or using strings (we used 5 copies of the "King James Bible"

$\approx 4M$ words). Then we iterate over the keys, hashing each key (the full benchmark is repeated 10 times averaging the results). To get a reliable measurement for the integer benchmark, we use some code that prevents optimizations, like compile time computations or pipelining[2]. The overhead of these anti-optimization-measures was quite small (below 3 ns see identity). Such tricks did not seem to be necessary for the string benchmark.

With this setup we test 9 different hash functions. Specifically for integer keys, we use the identity function and crc32—using the hardware optimized compiler builtin (to extend crc32 to compute 64 bit hash values we call it twice using different seeds once for the 32 most significant and the 32 least significant bits). The following hash functions work both for kinds of keys. Both xxh3 and xxhash are part of the xxHash library [15]. Murmur2, murmur3, and the tested implementation of tabulation hashing are all part of the well known smhasher library [4], however, since this has not been updated recently we use an active fork from user Reini Urban [86] (the original library does not contain tabulation hashing); for MD5 and SHA256 we use implementations that are part of the openssl package (installed via the default package manager).

The fastest measured hash function was *crc32*, it is hardware supported and significantly faster than the other hash functions, nearly on par with the *identity function*. However, the builtin function can only hash integer types thus the string test was not possible. The hash functions *xxh3*, *xxHash*, *MurmurHash2*, MurmurHash3/ and *tabulation* are designed for the use in hash tables. They also have very similar performance, i.e., about $\approx 2\times$ worse than *crc32*, but still by far faster than the two cryptographic hash functions *sha256* and *MD5* that we used for comparison. All non-cryptographic hash functions were significantly faster than a main memory lookup that would take around 100 ns. Thus, they should not impose a significant slow down over the faster more specialized functions.

[2]This was done by storing the result in an atomic variable and by using the previous result as part of the input for the next call of the hash function

2.3.2 Mapping to the Table Size

The preliminary hash function outputs pseudorandom integers that are far greater than the number of slots m in our table. Therefore, there has to be another step of mapping the hashed keys to the table size—using the mapping r_{map}. The only requirement for the table mapping is that the intermediate hash space H_{int} is mapped evenly to the final hash space H. Given this requirement there are many possible mappings, but in practice there are two main approaches—a circular mapping or a linear mapping (see Figure 2.2). Both are functions that are fast to compute.

circular or linear

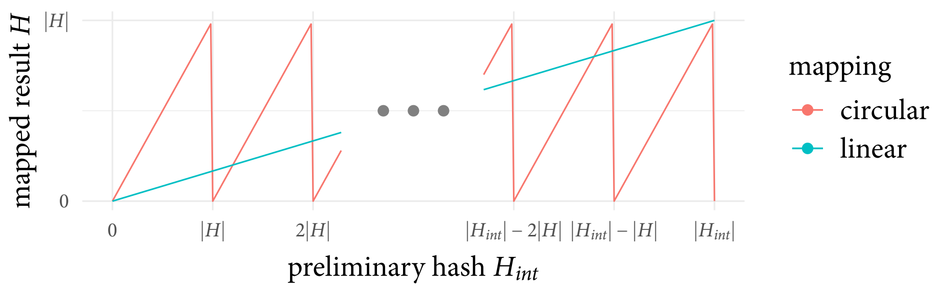

Figure 2.2: Schematic representation of mapping approaches.

table size $m = 2^j$

We first handle the case of hashing to a table with power of two slots, i.e., $m = 2^j$. This case is interesting because it leads to a better performance and because it gives some interesting insights into the two mapping techniques. It is possible to force a table size that is a power of 2 albeit at a memory overhead of a factor of 2. Mapping a hashed value to $m = 2^j$ slots can be done by selecting j bits from the binary representation of the hashed value. Thus, both circular and linear mappings can be computed in just a few cpu cycles using simple bitwise operations—making them very fast. The kind of mapping only depends on the positions of the bits we select. The circular mapping uses the least significant bits. They can be selected with a bitwise-and operation—using a bitmask to obtain the j least significant bits (see Algorithm 2.1). A linear mapping uses the most significant bits. They can be selected by shifting the hashed value to the right until only j bits remain.

arbitrary table size

Both a circular mapping and a linear mapping are still possible on tables with arbitrary sizes (not power of 2). But computing them is somewhat more difficult. The circular mapping can be computed using a simple modulo operation $r_{map}(x) = x \bmod m$. But, in figure 2.3 we see that the modulo computation takes about 4.3 times longer than the bitwise operation. There are two options of computing the linear mapping. (1) The first option is to use floating point multiplication. When constructing the table, we compute the scale factor $s = m/|H_{int}|$ (double precision) floating point number. Using this constant, we can map a hashed value by casting the hashed value to a floating point number, multiplying it with the scale factor, and re-casting it to an integer. (2) The second option uses an integer multiplication in combination with some bit shifting. This implementation of a linear mapping is sometimes called *fastrange* and was popularized by Lemire [39]. First the hashed value is multiplied with the capacity m, and then the result is shifted to the right by ℓ bits (where $H_{int} = [0..2^{\ell} - 1]$), basically computing the same multiplication as the floating point variant, but using only primitive operations. However, this only works if the first multiplication cannot cause any overflows. Therefore, it has to be implemented using double word operations, thus, if ℓ is 64 we need 128bit multiplications (i.e., fastrange64). Alternatively, one could use only the lower 32 bits of a hashed value to compute the mapping (using a 64bit multiplication), this would lead to a mixture of a cyclical and a linear mapping. However, our experiment below shows that using a 128bit multiplication does not lead to any performance penalties.

fastrange

An interesting fact about hashing is that hashing is similar to sorting elements by their mapped table position [62]. When a linear mapping is used this is similar to rounding the intermediate hash values and sorting the elements by their rounded values. If two hash tables use the same preliminary hash function then their keys are sorted similarly this can be beneficial for some algorithms like migration and join.

mapping benchmark

To benchmark the performance of the presented mapping approaches (results in Figure 2.3), we use a similar test as before (in Section 2.3.1) when measuring the performance of preliminary hash functions (with the same tricks to prevent optimization). This time, we map 10^8 random integers to a range of either $\{0, .., 2^{20}-1\}$ (power of two) or to a range of $\{0, .., 10^{11}-1\}$ (arbitrary). The mappings specialized for the power of two case have next to no performance penalty (compared to the identity function benchmarked in Figure 2.1). However, all methods that are usable

Algorithm 2.1 Pseudocode representation of different mapping variants.

```
$map\text{-}circular\text{-}pow2(prehash \in [0..2^\ell - 1]) \mapsto [0..m = 2^j]$
    return $prehash$ & $m - 1$ // bitwise and (keeps $j$ least significant bits)

$map\text{-}linear\text{-}pow2(prehash \in [0..2^\ell - 1]) \mapsto [0..m = 2^j]$
    return $prehash$ >> $(\ell - j)$ // bitwise shift (keeps $j$ most significant bits)

$map\text{-}circular\text{-}arbitrary(prehash \in [0..2^\ell - 1]) \mapsto [0..m]$
    return $prehash$ % $m$ // integer modulo computation

$map\text{-}linear\text{-}arbitrary(prehash \in [0..2^\ell - 1]) \mapsto [0..m]$
    // precompute the floating point number $scaling = m/2^\ell$
    return $\lfloor prehash \cdot scaling \rfloor$ // floating point multiplication

$map\text{-}linear\text{-}fastrange\text{-}arbitrary(prehash \in [0..2^\ell - 1]) \mapsto [0..m]$
    return $(prehash \cdot m)$ >> $\ell$ // $\cdot$ = double length integer multiplication
    // i.e.,64-bit-multiplication with $\ell$ =32 or 128-bit-multiplication with $\ell$ =64
```

with arbitrary table sizes have significant performance impacts. Both fastrange implementations keep their promise (3.7× faster). Both original arbitrary mappings take about as long, as a single evaluation of an optimized hash function, indicating that the table mapping can be a significant factor for the overall performance.

2.3.3 Double Hashing

Some hashing algorithms use multiple hash functions. This is obviously the case for multi hashing (see Section 2.4.2, page 29) and cuckoo hashing (see Section 2.4.2, page 37). But other hashing methods like linear probing can also profit from multiple hash functions, for example, when the current hash function turns out to be particularly bad at distributing elements throughout the table. In these instances, it can be advantageous to change the hash function, and redistribute elements within the table.

seeding a hash function

Most implementations of hash function use a seed. Therefore, creating a new hash function is as simple as using a different seed. This is especially useful, when only one or two hash functions are necessary. Whenever many (i.e., more than

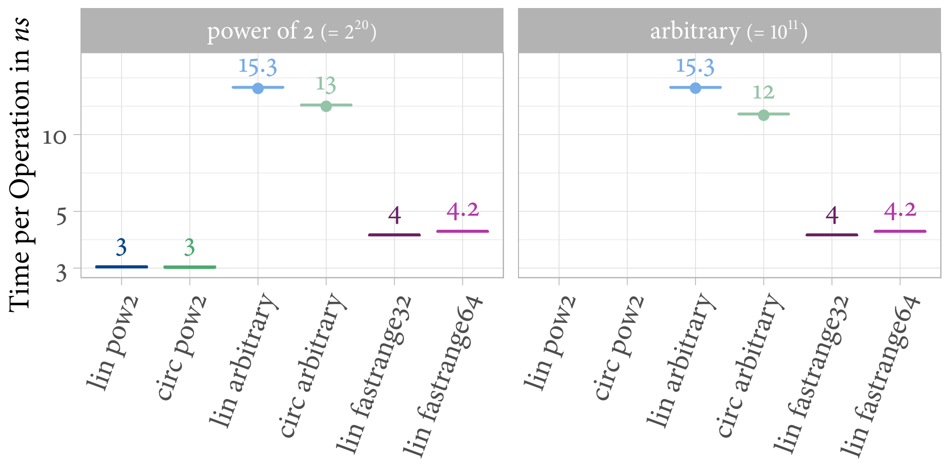

Figure 2.3: Comparing the performance of different map functions. *Left*: table size is a power of two; *right*: arbitrary table size

two) hash functions are necessary at the same time—e.g., within a cuckoo hash table—executing many hash functions per element can become a significant overhead. Double hashing can be a solution for these instances. The idea is to compute only two preliminary hash functions, but to create many hash values from these two preliminary results. The hash values $h'(x)$ and $h''(x)$ are computed using two distinct preliminary hash functions (potentially different seeds). Then we can compute the hash functions $h_0(x), \dots, h_k(x)$ using linear combinations of $h'(x)$ and $h''(x)$.

double hashing

$$h_i(x) = h'(x) + i \cdot h''(x) \mod 2^\ell$$

One requirement for this to work in theory is that $h''(x)$ and 2^ℓ have to be coprime, i.e., $h''(x)$ must not be even. This can easily be enforced (by fixing its last bit to one). In some sense, double hashing reduces the "randomness" of each hash function. Nevertheless, it has been proven to work well in many real world scenarios like computing the different hash functions of a cuckoo hash table [60].

Another way to reduce the number of computed hash functions is to increase the number of used bits from each computed hash function. Throughout this dissertation, we use hash functions primarily to address tables. Therefore, we only ever use $\approx \log(m)$ bits to fairly address the table (especially if the table size is a power of two). Most hash functions return a 64 bit hash value—enough to compute two addresses for tables below 2^{32} slots. One using the upper 32 bits and one using the lower 32 bits. This can even be combined with double hashing, i.e., using the lower 32 bits of $h(x)$ as $h'(x)$ and the higher 32 bits as $h''(x)$, thus generating an arbitrary number of mapped slots from only one preliminary hash function.

splitting hashed values

2.3.4 Some Words about Theory

The theoretical properties and qualities of hash functions have been studied extensively. Largely, there are two kinds of quality metrics. Single function metrics like *uniformity* that are analyzed for one specific hash function or *universality* and *k-independence* which are properties inhibited by a family of hash functions rather than one instance of a hash function.

single function metrics vs. hash-function-family metrics

Single Funciton Metrics

Single function metrics are quality measures or properties that one hash function can hold on its own (i.e., one function with one seed). The most common quality in this category is *uniformity*, i.e., how close does the hash function resemble a uniform distribution (every slot has the same probability). There are multiple projects that evaluate the uniformity of hash functions [4, 86, 15] in an experimental analysis. All (preliminary) hash functions presented in Section 2.3.1 perform really well in these tests (appart from the identity and crc32). However, it should be noted, that uniformity alone, does not enforce any type of randomization. It also does not enforce scattering "similar" elements throughout the whole result space which we stated as a goal for the preliminary hash functions.

uniformity

The property that similar input elements lead to wildly different results is sometimes called the *avalanche effect*. A hash function fulfills the *strong avalanche criterion* if one bit flip in the input element, will cause all output bits to change with a probability of 50 % (e.g., tabulation hashing). However, the property is most commonly mentioned in the context of cryptographic hash functions, as such there are very few tests or proofs for other hash functions that are used for data structures.

avalanche effect

Hash-Function-Family Metrics

In 1979, Carter and Wegman [11] have established a different way of analyzing families of hash functions. When looking at a single hash function there is always the possibility that any given set of input elements constitutes a bad input, i.e., produces above average collisions. This possibility can be quantified if the hash function is chosen at random from a set of hash functions (family of hash functions). Given a random hash function from this set, we can analyze the probability for a hash function that produces above average collisions. The original metric proposed by Carter and Wegman [11] was *universality*. A family of hash functions is universal if given two elements, the probability of randomly picking a hash function where both elements collide is $1/m$. This same universality concept has been generalized [88] to multiple elements and multiple slots, i.e., a family of hash functions is *k-independent* if for k elements $x_1, \ldots, x_k$ and k associated slots $s_1, \ldots, s_k$ the probability of $P[\forall_i\ h(x_i) = s_i] = m^{-k}$.

why analyze families of hash functions

universality

k-independent

The generalized concept of k-independence can be used to show the requirements of different hashing methods towards their hash functions. For example in the case of linear probing (described in Section 2.4.2, page 31), it can be shown that the expected constant running times for insertions can only be guaranteed with a hash function randomly chosen from a family of 5-independent hash functions [67, 74] (there are families of 4-independent hash functions that lead to logarithmic insertion times). Many families of provably k-independent hash functions like high degree polynomials are slow to evaluate. Patrascu and Thorup [73] show that simple tabulation hashing is strong enough to guarantee constant insertion times, even though it is only 3-independent [73] (but not 4-independent or higher).

linear probing needs 5-independence

Most other commonly used hash functions like xxh3 and MurmurHash can be seeded and thus also form families of hash functions. However, the resulting families are usually not proven to be universal or k-independent. In practice we have had very good results with xxh3 [15], thus we used xxh3 in our experiments presented in the following chapters.

experiments use xxh3

2.4 Collision Resolution

(hash-)collisions

open and closed addressing

Whenever an element is inserted into a non-empty table—using a random position determined by a hash function—there is a possibility that the position already holds an element. We call this a *(hash-)collision*. The probability of a new element causing a collision grows with the fill ratio δ of the table. The element that caused the collision still has to be stored in the table. Depending on the hashing technique there are different ways to resolve collisions like this. There are two major categories of collision resolution techniques *open addressing* and *closed addressing*. Open addressing means that elements can be stored in other slots—not just their canonical slot. This includes many common hashing techniques like *linear probing* and *cuckoo hashing* (both described later throughout this Section). In a closed addressing table each element must be stored in its canonical slot. Thus, slots have to be able to store a varying number of elements (usually in a linked list or a similar unbounded data structure). The most common examples of closed hashing are variants of *hashing with chaining*. Due to some potential for confusion with the terms *open* and *closed hashing* (Note below)—that work contrary to open and closed addressing—we instead use the terms hashing with *in-table-displacement* for open addressing and hashing with *separate chaining* for closed addressing.

in-table-displacement

separate chaining

Notation

open addressing ≠ open hashing

There is some misleading vocabulary surrounding the terms open and closed addressing versus the terms open and closed hashing. The most confusing part is that open addressing and open hashing refer to contrary concepts. As described above, open and closed addressing refers to the slot where an element is stored. Open and closed hashing refers to whether the elements can be stored outside the table (i.e., addressed by a pointer). Therefore, hashing with chaining uses closed addressing (all elements are stored in their canonical slot) but open hashing (they are stored outside the table within a queue). Similarly, linear probing is a classic example for open addressing, but usually conforms to closed hashing because all elements are stored within the table. To avoid any type of confusion we have chosen to use the terms *in-table-displacement* and *separate* *chaining* throughout this dissertation. We feel they are more descriptive of the actual data structures.

2.4.1 Separate Chaining

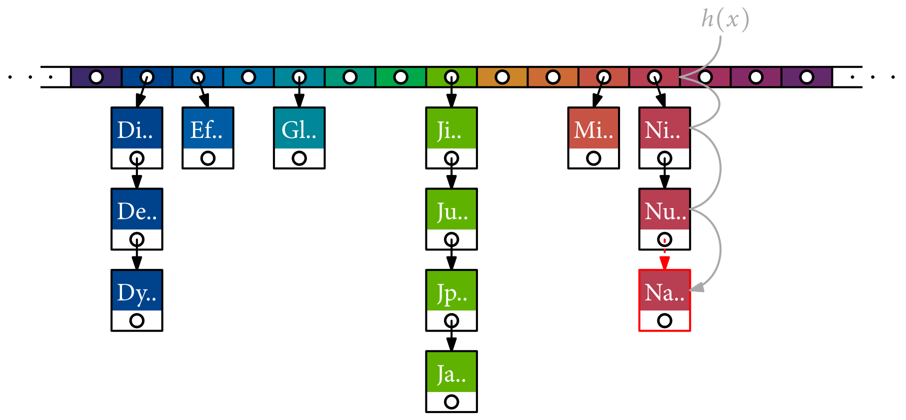

Figure 2.4: Insert operation into a table using chaining.

Hashing with chaining is a very common hashing technique. Instead of storing elements directly in the table each slot holds a queue of elements (usually a singly linked list). An insertion creates a new queue item holding the inserted element ($O(1)$). Queries search the queue of their canonical slot ($O(|queue|)$). Some performance metrics of hash-based data structures are particularly impacted by chaining-based collision resolution. A chaining-based architecture has to store one pointer per slot and per element. Therefore, the memory usage is $m \cdot s_p + n \cdot (s_p + s_e)$ where s_p and s_e are the sizes of pointers and elements respectively. Hence, the memory efficiency depends primarily on the relative size of elements to pointers. The speed of queries is also impacted by using pointers. To find an element queries first have to iterate over the queue of the canonical slot. However, iterating through a queue that is implemented as a linked list is significantly slower than searching through an array because each link leads to a new (main) memory access that has to be resolved before comparing two elements, and before the next link (see Section 2.5.1).

memory usage

There are different variants of hashing with chaining that address some of these problems. One common technique is to store the list heads in the table, thus the performance penalty of queries is reduced, e.g., when querying the first queue element or when the queue contains only one element (which is often the case for realistic workloads $E[|queue|] = \delta$). Another technique that improves the cache efficiency of queries (at the cost of memory) is to group queue elements and to store these groups in larger blocks. This improves query times because blocks are faster to scan through than linked lists. However, non-full blocks can increase the necessary memory. There is a large design space of moving elements to reduce the memory consumption. E.g., if insertions are sufficiently rare the queue can be stored within a perfect sized array, which could actually save memory because less pointers are necessary.

list heads in the table

block queues

Overall it would be fair to say that hashing with chaining offers many versatile solutions that are usable in a variety of situations. But throughout this dissertation we are more interested in specialized solutions that achieve better performance than general purpose chaining hash tables. Thus we put our focus on different variants of in-table-displacement techniques.

versatile but never the optimum

2.4.2 In-Table-Displacement

When all elements are stored directly in the table, hash collisions have to be resolved by finding alternative slots for elements. One common way to do this, is to have an ordered list of alternative slots for each element, i.e., when inserting element x, we look at the slots $s_0(x), s_1(x), \ldots$ ($s_0(x)$ is the canonical slot). Each element has to be stored in one of its slots $s_i(x)$. We say the *displacement* of an element is the number i of the slot it is stored in. Small displacements are beneficial because query operations find their elements faster when they probe slots in ascending order.

displacement

Different hashing techniques with in-table-displacement differentiate themselves in the way alternate slots are chosen, the way queries iterate through the alternate slots, and in how they spot that an element is not in the table. Additionally, there are hashing techniques with in-table-displacement that move previously inserted elements when a new element is inserted or an old one is deleted. This can be used to move displaced elements after a deletion or to reduce the displacement of an element. However, moving elements has impact on the *referential integrity* of the table (see Section 2.1.1). Every pointer, iterator, or reference that used to point to an element before it was moved now points to an empty slot or even to another element that replaced it. Depending on how the hash table is used, referential integrity might be important (e.g., in parallel applications). Thus it should always be clear which operations can invalidate references.

referential integrity

Deletions are often a difficult to handle for hash tables with in-table-displacement. A lot of implementations stop queries once an empty slot is encountered (i.e., once an alternative slot is empty). In these tables, deleted elements cannot be fully removed from their slots because elements that were displaced previously would not be found after the element was cleared. Sometimes this can be fixed by moving other inserted elements within the table to fill the empty slot (e.g., linear probing). However, there are techniques where finding and moving an appropriate element is infeasible (e.g., multi hashing). In those cases marking the element as being removed might be the only option to implement deletions. The removed element is replaced with a dummy called a *tombstone*. Future queries will treat the dummy as if it was any other element, new insertions can replace the tombstone.

tombstone

In the following we explain some of the most common hashing techniques that use in-table-displacement.

Multi Hashing

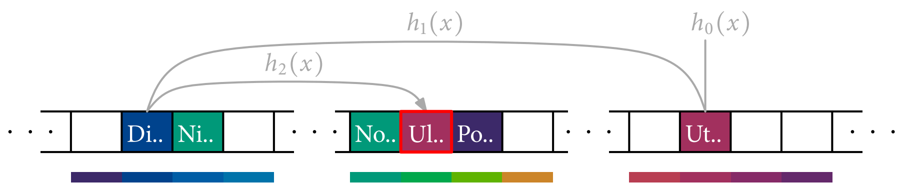

Figure 2.5: Insert operation into a table using multi hashing.

Multi hashing is more of theoretical technique meant to understand in-table-displacement. It is based on the idea that each alternative slot can be found the same way as the canonical slot, i.e., with a new hash function $h_0, h_1, \ldots$. Every probe $s_i = h_i(x)$ has the same probability of finding a free slot independent of the number of unsuccessful probes. This makes multi hashing significantly easier to analyze (expected running times for operations). When inserting x, whenever a collision occurs x is rehashed using another hash function until we find an empty slot (in theory this needs an arbitrarily large number of hash functions). During a query, we probe alternative slots until the queried element or an empty slot is found. An empty slot indicates that the queried element was not previously inserted.

The expected running time of operations within a table using multi hashing depend on the expected probe length (how many slots are accessed before the element or an empty slot is found). There are three main operations: inserting a new element, querying an inserted element, and querying an element that was not inserted (a note on deletions is explained later in this section). Both insertions and negative queries probe through alternative slots until an empty slot is found (both have the same running time). Each probed slot is independently drawn using a uniform hash function. Therefore, each time a slot is probed it is empty with a probability of $p_{empty} = \frac{m-n}{m} = 1 - \delta$ and thus the expected number of probed slots is $E[Probes_n] = (1-\delta)^{-1}$. The number of probes when looking for an inserted element is equal to the number of probes when said element was originally inserted, thus it depends on the fill degree when said element was inserted. Looking for the first element always succeeds on the first probe while looking for the last inserted element could take a lot longer. Probing an element takes the same number of probes as inserting it, therefore, querying a (uniformly-)random contained element leads to the following expected running time:

$$\begin{aligned}
E[Probes_{average\ query}] &= \frac{1}{n}\sum_{i=0}^{n-1} E[Probes_i] \\
&= \frac{1}{n}\cdot\left(\frac{m}{m} + \frac{m}{m-1} + \ldots + \frac{m}{m-(n-1)}\right) \\
&= \frac{m}{n}\cdot\left(\frac{1}{m-n+1} + \frac{1}{m-n+2} + \ldots + \frac{1}{m}\right) \\
&= \delta^{-1}(H_m - H_{m-n})
\end{aligned}$$

Deletions are especially hard to handle for tables using multi hashing. When removing an element, its slot cannot be cleared, because the element could have caused other elements to be displaced. Furthermore, it is very hard to find elements that have been displaced because of the removed element, since the displaced elements could be stored anywhere within the table. Therefore tombstones are the only feasible method for implementing deletions in a multi hashing table. The running time of a deletion is similar to the running time of the query necessary to find the element that should be deleted. However, future queries do not become faster after deleting the element since the slot is still filled (insertions could reuse the slot). The running times described above always use the actual number of filled slots.

Linear Probing

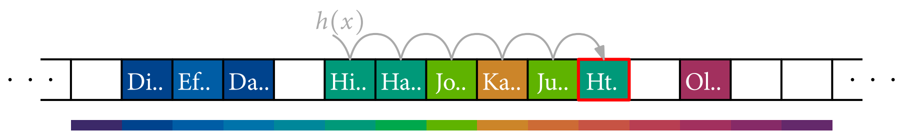

Figure 2.6: Insert operation into a table using linear probing.

Linear Probing is probably the most common example for hashing with in-table-displacement. During an insertion, whenever a collision occurs the inserted element is stored in the first free slot after its canonical slot (i.e., $s_i(x) = h(x) + i \mod m$ circular order). When an element is queried, all slots after its canonical slot are *scanned* (i.e., searched linearly) until either the element or an empty slot is found. This method has significant performance advantages on actual machines because sequential scans take better advantage of modern memory hierarchies than other access patterns(i.e., cache usage and prefetching mechanisms, see Section 2.5.1).

$s_i(x) = h(x) + i \mod m$

scan

Because of its performance in practice and its relative simplicity, linear probing is explained in most beginner level algorithm textbooks. Analyzing its expected running time, however, is significantly more challenging compared to the multi hashing example above. The problem is that elements start to naturally form clusters in the table. A *cluster* is a continuous range of filled slots. New elements have a higher chance of being stored at the end of an already existing cluster (i.e., elements hashed anywhere within a cluster are stored at the end of said cluster). These clusters increase the average probing distance. Especially for negative queries and insertions.

cluster

The following running times have famously been proven by Donald Knuth in an unpublished memorandum in 1963 [36] (also in their book [37]). Knuth's proof is sometimes considered to be the origin of modern average case analysis [67]. Assuming a fully random hash function:

$$E[\textit{probes until empty}] = \frac{1}{2}(1 + \frac{1}{(1-\delta^2)}) \tag{2.1}$$

$$E[\textit{probes query}] = \frac{1}{2}(1 + \frac{1}{1-\delta}) \tag{2.2}$$

$$E[\textit{probes fill}] = n \cdot E[\textit{probes query}] = \frac{n}{2}(1 + \frac{1}{1-\delta}) \tag{2.3}$$

The number of probes to find an empty slot (e.g., during an insertion or negative find) is $E[\textit{probes until empty}]$ (Equation 2.1). Querying an inserted element needs on average of $E[\textit{probes query}]$ (Equation 2.2). Filling a table incrementally with n insertions has a linear cost $E[\textit{probes fill}]$ similar to n queries on the final table (Equation 2.3). These bounds also hold for hash functions that are (uniformly) randomly chosen from a 5-independent family of hash functions [67] or when using tabulation hashing [73] (see Section 2.3.4)

deletions In contrast to multi hashing and many other probing variants, linear probing supports deletions without the use of tombstones. When an element is deleted and its slot is cleared, we can find all other inserted elements that were displaced because of the current element since they are still stored in the consecutive slots of the same cluster. After clearing the slot we scan through the following elements. If we find an element whose canonical slot is before the cleared slot, then we move the element into the cleared slot, and recurse on its slot. However,This implementation violates the referential integrity that is otherwise maintained by all operations of a linear probing hash table, i.e., if there existed any references to the original position of the element (e.g., a pointer), then these references would be invalidated by moving the element. Another method for deletions that maintains referential integrity was recently explored by Sanders [80]. Their method is based on tombstones, but it only inserts a tombstone if it is necessary to find another element—removing old tombstones if they become unnecessary. They show with practical experiments that this method leads to an equilibrium of constant sized clusters, after a period of mixed deletions and insertions.

Implementation details

circular wrapping

Linear probing hash tables are often implemented such that a probe that hits the end of the table wraps around to the front of the table (Pac-Man style). We call this method *circular wrapping*. This implementation has the advantage that the table is topologically uniform—all slots are equal. The problem is that wrapping the table necessitates some computational overhead (either a bitwise AND or conditional jump). A somewhat faster method is to provide a number of slots at the end of the table. We call these the *overflow buffer*. The slots of the overflow buffer are only reachable through probing from the actual table, i.e., no hash value is mapped to the slots of the overflow buffer. The size of the overflow buffer should be large enough to hold the longest expected cluster ($\approx O(\log n)$ for reasonable δ). Additionally, the overflow buffer can be used to simplify the detection of a full table using a *sentinel slot*. To do this, we ensure that the last slot of the overflow buffer can never be filled, i.e., we report that the table is full whenever the last slot would be filled by an insertion (usually triggering a capacity increase). This ensures that each probe is always able to find another empty slot before it reaches the end of the table making checking for the boundary of the table unnecessary.

overflow buffer

sentinel slot

Other Probing Schemes (Quadratic, Random, ...)

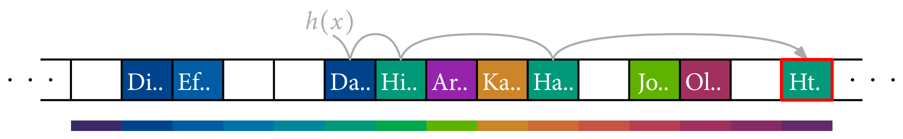

Figure 2.7: Insert operation into a table using quadratic probing.

All of the following techniques aim to reduce the clustering of a table. Thus, achieving the expected probing distances of multi hashing while retaining some of the memory efficiency and performance of linear probing.

Quadratic Probing [37] $s_i(x) = h(x) + i^2 \mod m$ here the distance to the canonical slot grows quadratically in the number of probes necessary to find an element. For most elements, the search is still very local and for larger displacements, this method aims to reduce the clustering problem.

Double Hashing [37] $s_i(x) = h'(x) + i \cdot h''(x) \mod m$ this technique could be described by using double hashing (see Section 2.3.3) to compute the many hash functions used for multi hashing. The distance between two probed slots is computed using a second hash function ($h''(x)$) thus, average cluster size is drastically reduced but the data locality remains low on average.

Mixed Hashing schemes we can easily imagine mixed hashing schemes that use linear probing for a few slots, before switching to one of the other methods. This way the memory efficiency could be similar to linear probing while also shortening clusters.

Circular Cache Line Probing (CCLP) one such mixed technique that we believe might be interesting is CCLP (it has not been mentioned scientifically). It combines some of the advantages of linear probing and random probing. The idea is to split the table into blocks and to scan the whole block, whenever one part of the block has to be loaded (e.g., one block = one cache line). First the whole block containing the canonical slot is probed in a circular fashion. If all slots within the "canonical block" are filled, we use another probing technique (e.g., double hashing) to compute an alternative slot within a new alternative block that is again probed in a circular fashion.

Robin Hood Hashing

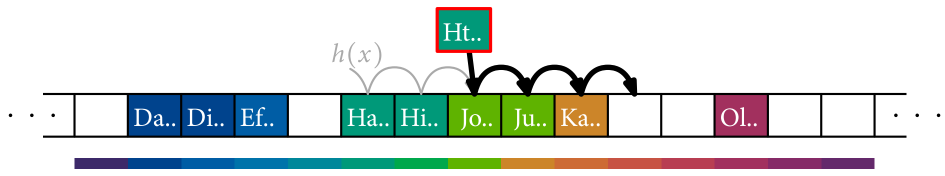

Figure 2.8: Inserting an element into a hash table using Robin Hood hashing.

The motivation behind Robin Hood [12] hashing is to minimize the variance in query running times by moving elements closer to their canonical slot. The resulting hash table is a linear probing hash table where all elements within a cluster are sorted by their hashed key. All elements that belong to the same canonical slot are stored in a continuous range called a *run* or *canonical run*. This order of elements minimizes the maximum displacement while still representing a legal linear probing hash table (i.e., same resulting table as a linear probing table with a different order of insertions).

sorted by their hashed key

canonical run

The insertion algorithm is the main difference between linear probing and Robin Hood hashing. If there is a collision, the element has to be inserted into the correct slot within its canonical cluster (the filled range of slots that contains its canonical slot). The correct slot can be found by hashing the keys of elements that are scanned while searching for a free slot. Once an element is found whose key has a larger position we insert into its slot and this element and all following elements of the cluster are moved one slot to the right. If there is no such element in the cluster, the new element is inserted in the first empty slot after the cluster (similar to linear probing).

The query algorithm is the same as in linear probing. From the canonical slot we scan to the right and compare the keys of stored elements to the queried key until the correct element or an empty slot is found. However, there are techniques that can improve the query performance. For example it is possible to stop scanning the cluster, once an element is found, whose canonical slot is larger than the queried element's canonical slot. Alternatively, it is possible to store the maximum displacement length throughout the whole table. Thus, queries can be aborted after they have looked at enough slots. This method is interesting for Robin Hood hashing in particular, because it minimizes the maximum displacement (it is also less prone to variance). Both of these early rejection methods decrease the number of probed slots at the cost of some additional comparisons. Therefore, they are both only beneficial if the average probing length's are large, i.e., if the table is densely filled.

Robin Hood hashing also allows for some very intuitive acceleration techniques that depend on per-slot *auxiliary data*, i.e., some additional information that is stored in each slot (in addition to the key value pair). These information can be

using auxiliary data

used to avoid executing the hash function unnecessarily or to find the canonical run that belongs to a slot more quickly.

- each slot could store 3 auxiliary bits that encode cluster start, run start, and occupied comparable to quotient filters (see Section 5.4)
- each slot could store the displacement of its run (offset of the first element hashed to it) since displacements are usually small 8 bits per slot should be enough to store these offsets.

Hopscotch Hashing

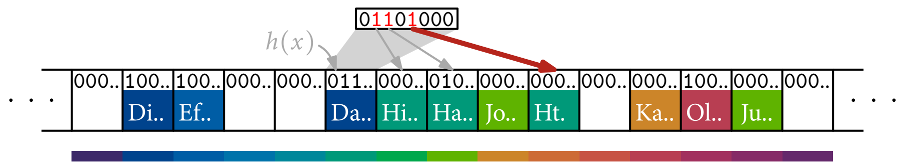

Figure 2.9: Querying a hopscotch hash table.

Hopscotch hashing [33] is another hashing technique that rearranges elements in a linear probing based hash table. However, the main goal is not to minimize displacements but to improve the performance via *auxiliary per-slot data*. Each slot s stores a bitmask $(b_{s,0}, \ldots, b_{s,j})$ of size j representing its neighborhood (the canonical slot and $j-1$ slots after it). Elements can only be stored in the neighborhood of their canonical slot. Each $b_{s,i} = 1$ iff slot $s + i$ stores an element that was originally hashed to slot s. Therefore, queries only need to compare one key per 1-bit in the canonical slots neighborhood bitmask.

bitmask representing the neighborhood

The insertion works similar to linear probing. The slots after the canonical slot are scanned until an empty slot is found. If the empty slot is inside of the canonical slots neighborhood, the element is inserted and the appropriate bit is updated. Otherwise, elements from other slots are moved to the right in an effort to create an empty slot closer to the canonical slot. In some instances this might not be possible (without having to move elements out of their canonical neighborhood), then the current element cannot be inserted. The maximum displacement in the table has to be smaller than the neighborhood size.

Queries and deletions are the main strengths of hopscotch hashing. Through the help of the neighborhood data it is possible to reduce the number of compared elements to the number of elements that are true collisions (i.e., elements that have the same canonical slot). Negative queries can be particularly fast, especially when the canonical slot has an empty neighborhood bitmask. One large advantage is that elements can be removed easily by removing the corresponding 1-bit and clearing their slot, without moving any elements (this works because queries do not access or stop at empty slots, instead they always look at all slots that have a 1-bit in the neighborhood data.

Cuckoo Hashing

Cuckoo hashing [19, 25, 59, 68] is a technique where each element only has a constant number of alternative slots (i.e., slots it can be stored in). Thus, the running time of a query is guaranteed to be constant. This even holds, for densely filled tables, where linear probing would slow down due to large clusters. Cuckoo hash tables are able to operate under higher fill degrees than other tables with in-table-displacement because they allow rearranging elements between their alternative slots. This opens a large search space of new potentially empty slots. A more detailed view at cuckoo hashing can be found in Section 3.5 where we use techniques based on cuckoo hash tables to grow a table incrementally in small steps.

guaranteed constant look-ups

In the following we give a short introduction to H-ary B-bucket cuckoo hashing (from now on (B, H)-cuckoo hashing). There are many other variants of cuckoo hashing, however most of them are adaptations of this base technique. The table is split into m/B buckets with B slots each. Additionally, we use H different hash functions $h_0, ..., h_H$. Instead of hashing to a slot directly, each hash function hashes an element to one of these buckets ($h_i : U \mapsto [0..m/B - 1]$).

(B, H)-cuckoo hashing

When an element is inserted, it is hashed once with all H hash functions obtaining different "canonical buckets". If at least one bucket is not full, the element is inserted into the least full of its buckets. This technique uses the *power of two choices* [58] concept to automatically balance elements within the table. Even when all its buckets are full, the element can usually be inserted by moving one of the other elements into one of its other buckets. This form of displacements can lead

power of two choices

to long chains of elements being moved until a free slot is found (i.e., recursive moves if those buckets are also full).

finding moves by exploring the induced graph

To find these moves, we traverse elements in a graph like manner. Each node represents a bucket and each element contained in the bucket creates an edge from the node to all other nodes that represent the alternative buckets of the element. In this graph representation, each full bucket has an *outgoing degree* of $B \cdot (H-1)$. To find a way of inserting a new element, we have to find a simple path from a node representing one of its canonical buckets to a node representing a bucket that is not full. Once such a way is found, we can move elements along this path freeing up a slot in the original bucket. In general, there are two ways of finding a path like this—breadth first search and random walk 3.5.4).

outgoing degree $B \cdot (H-1)$

breadth first search

random walk

Both queries and deletions are very simple, all canonical buckets are scanned looking for the element. Deletions can clear the slot of found elements, without any additional work.

Coalesced Hashing (in-table-chaining)

in-table chaining

An older technique that we want to point out because it is an interesting combination of in-table-displacement and chaining techniques is coalesced hashing. Each slot of the table stores a pointer in addition to its element. These pointers are used to create chains that connect each slot with all elements that were hashed to said slot. Elements that are not stored in their canonical slot are stored in alternative slots somewhere in the table. Their alternative slot is added to a queue starting in its canonical slot (linked through the additional pointers). This technique is somewhat similar to hopscotch hashing, in that additional per slot data (here pointers) is used to improve queries and other operations. Using the chains created by pointers, we can reduce the number of comparisons and thus reduce the effects of cluster generation.

When an element is inserted, we first check its canonical slot. There are 3 possibilities:

1. the slot is empty: We store the element in its slot.
2. the slot holds another element that was also hashed there: We iterate through the chain given by the pointers ($s_0 = h(x)$ and $s_i = \mathit{pointer}(s_{i-1})$) to find

out, weather the element is already in the table. At the end we find a new alternative slot. There we store the element and update the pointer of the last element in the chain.

3. the slot holds an element that was hashed to another slot: In the classical implementation, we proceed like in case 2. Thus, both chains start to interleave (the queue contains elements with two different canonical slots). This can be avoided at the cost of referential integrity (see box below).

When querying a key, we only have to check all slots within its "canonical chain". When deleting an element we update the pointer of the previous chain element and clear the slot. If the element was actually the first element of its chain, we can move the last element of its chain into the original slot, removing the last pointer of the chain. Merged chains might need some special attention during the deletion algorithm (because slots within one chain could be first chain link of another chain).

There is some interesting design space in how to find alternative slots in a hash table like this. In theory, it would be possible to use any probing mechanism presented previously, but without any changes to the algorithm (see box below) this would lead to many interleaved lists. The original variants of coalesced hashing published in the early 80s [87] solve this problem by using a *cellar* of unreachable slots at the end of the table. The idea is to have m slots in the table, but to address only m' slots using the hash function (we describe a similar technique for linear probing page 33). The last $m - m'$ slots are only used as alternative slots. Using these un-addressable slots as alternative slots prevents chains from merging. Alternative slots can be found using a simple pointer to the last free slot in the table. Whenever an alternative slot is needed, the last empty slot is used, and the pointer is decremented. The pointer automatically starts to use slots from the addressed region, once the cellar is full. The appropriate ratio between m' and m depends on the targeted fill degree, and also on the ratio of successful to unsuccessful lookups [83].

cellar

Updating coalesced hashing

Coalesced hashing is a really old technique that predates the importance of cache efficiency onto the running time of algorithms. But given a few updates, in-table-chaining could be competitive with other in-table-displacement techniques.

The goal is to get the combined advantages of linear probing (cache efficiency) and hashing with chaining (less comparisons, easy deletions) at the cost of some memory (≈ 1 byte per slot not a full pointer). To get a similar cache efficiency to linear probing, we choose alternative slots with one of the probing techniques mentioned in the previous sections, e.g., linear probing, or quadratic hashing. To prevent interleaved chains, we change the third case of the insertion algorithm to:

3. We find an alternative slot for the stored element and move the element to this slot. Then we fix the chain of the moved element. Now the original element can be stored in its canonical slot.

Using linear or quadratic probing to find alternative slots also helps to reduce the memory overhead of in-table-chaining. Since, alternative slots are likely reasonably close to each other, we can store pointers in the form of small offsets to the next queue item. A 1 byte offset should be more than enough to store the relative position of two chain links. If one offset poses a problem, we can move already stored elements to create a free slot closer to the last queue item (similar to hopscotch hashing).

Table 2.1: Overview of collision resolution schemes strengths and weaknesses.

Hashing Method	in-table	cache	duplicates	deletions	ref. invalidation
chaining	×	–	✓	✓	never
chaining with heads	×	0	✓	✓	deletion, mig.
multi hashing	✓	–	slow	tombstone	migration
linear probing	✓	++	slow	✓	deletion, mig.
random probing	✓	–	slow	tombstone	migration
quadratic probing	✓	+	slow	tombstone	migration
Robin Hood hashing	✓	++	slow	✓	non-lookups
hopscotch hashing	✓	++	can fail	✓	insertion, mig.
coalesced hashing	✓	0	slow	✓	migration
cuckoo hashing	✓	0	can fail	✓	insertion, mig.

2.5 Some Facts about Memory

2.5.1 Cache

A common bottleneck in many algorithms is the time spent on waiting for memory accesses. This is also true for most hash table operations. In Section 2.3 we measured the time to compute the hashed value for a random key with 64 bits at about 13 ns (9 ns xxh3 + 4 ns fastrange64). Each main memory access however takes between 50 and 100 ns. This can be reduced if the accessed memory region is stored in the *cache hierarchy* consisting of L1, L2, and L3 caches. However, one main memory access per operation seems to be unavoidable if the table is significantly larger than cache (multiple accesses to the same key would be improved by caching).

cache hierarchy

There are a couple of facts that are necessary to understand the memory performance of hash tables. Once a piece of memory was accessed, it is stored in the cache. As long as it remains cached, future accesses are considerably faster (≈ 1 ns for an access to the L1 cache). The smallest unit of memory that is loaded into the cache is one *cache line* (usually 64 bytes). Thus, accessing two memory slots that are physically close to each other can be considerably faster. Furthermore, modern architectures use a *cache prefetcher* which loads cache lines that were not already accessed, further increasing the effectiveness of accessing physically close memory positions. This is especially apparent when *scanning* through memory entries, i.e., linearly looking at consecutive memory entries. When scanning through consecutive memory entries every piece of the cache line that was loaded is used (high cache reuse rate), additionally, the access pattern is really easy to detect for the prefetcher making it highly likely that consecutive cache lines are already loaded when they are accessed for the first time.

cache line

cache prefetcher

scan = linear search

When accessing memory there are two performance measures *latency*, i.e., the turnaround time from requesting a cache line to it being available to the CPU and *bandwidth*, i.e., the amount of memory per time that can be loaded. The bandwidth is typically high enough, to load multiple cache lines at once. Therefore, there can be some *pipelining*, i.e., multiple cache lines being in transit at the same time, e.g., when they were accessed consecutively. This is only possible, however, if the second memory access does not depend on the first one (i.e., the address of the second

latency and bandwidth

pipelining

memory access does not depend on the value of the first access). We call this a *dependent memory access*. Dependent memory accesses should be avoided because they decrease the possibilities for pipelining, each dependent memory access that is not cached adds its latency to the overall running time of the operation.

dependent memory access

manual prefetching

Cache lines can also be *prefetched* manually. This can increase performance if the hashing scheme dictates some alternative memory positions that are very likely to be accessed, but that are not obvious to the automatic prefetcher. One such example is during cuckoo hashing (page 37). When querying an element we can prefetch all canonical buckets before beginning to access the first bucket, this will cause all buckets to be loaded at the same time. Thus, accessing the second and third buckets will be significantly faster after scanning the first.

Cache Performance of Different hashing Techniques

impact of latency, pipelining, and prefetching

To visualize the impact of cache effects—latency, pipelining, and prefetching—on different hashing algorithms, we conduct the following experiment. A hash table with 50 M slots is filled to around 70 %. After the insertion, we check the displacement of each element and store elements grouped by their displacement (number of probed slots until the element is found). In a second step, we measure the performance of look ups from each group (the whole experiment is repeated 5 times using different key sequences; on an AMD Epyc 7551P CPU with 256 MB L3 Cache). The results can be seen in Figure 2.10: the plot on *top* shows the average time $t_{avg}(i)$ for queries within group i, the *bottom* plot shows the size $s(i)$ of the group (logarithmic scale), the *right* plot overall average query time of each hashing algorithm (i.e $\sum_i t_{avg}(i) \cdot s(i)$).

The most obvious result is, longer displacements lead to longer query times. However the correlation changes between different hashing techniques[3]. All plots seem to start at around 40 ns per operation, which is lower than the common wisdom would indicate for one main memory access but we should also consider that around 1/3 of the hash table fits into the L3 cache of our machine (256 MB). Most

≈ 40 ns for the first memory access

[3]Robin Hood hashing profits from the structure of the benchmark. Elements that are stored in neighboring slots often have the same displacement, this is especially noticeable when groups become small.

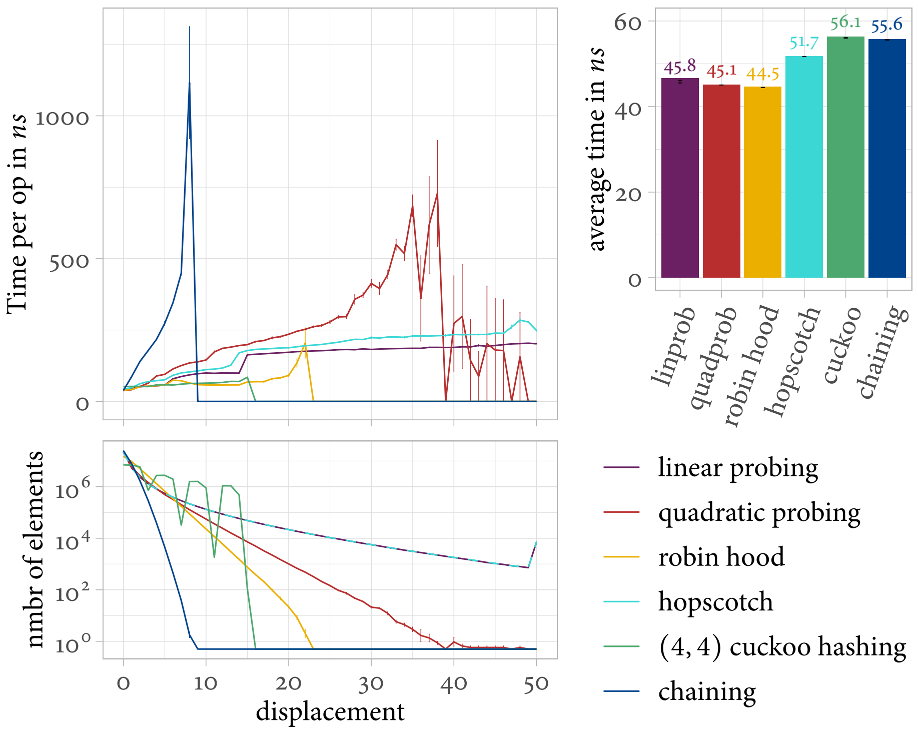

Figure 2.10: Influence of displacement on cache performance. *Top*: shows the average time for queries with a certain displacement distance. *Bottom*: displays the number of elements with a certain displacement. *Right*: average query time overall.

techniques have displacements-distribution that are heavy on elements with low displacements.

Linear probing in particular has relatively long displacements, however, the query time per displacement is faster than other methods, with clearly distinguished levels (plateaus) loosely corresponding to (pairs of) cache lines (cache lines contain four 16 byte elements and are loaded in pairs). *Quadratic probing* achieves shorter displacements due to reducing the size of clusters, however, elements with larger displacements take significantly longer to find due to the low usage rate of loaded cache lines (only one comparison per cache line). *Hopscotch hashing* does not seem to have any benefits compared to linear probing, the displacement-distribution is

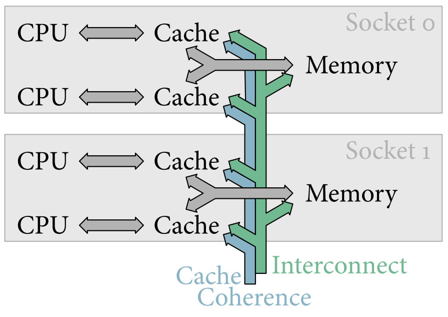

Figure 2.11: Schematic representation of the memory system in a multi socket shared memory machine.

exactly the same as with linear probing and performance does not seem to benefit from the additional neighborhood bitmasks.

Robin Hood hashing shortens the displacements, however, it also reduces the number of elements with smaller displacements, i.e., moving more elements towards the average displacement. Thus, more elements fall into the second level of performance (i.e., displacement > 5).

Looking at the performance metrics of *cuckoo hashing* we can clearly see the bucket structure. At 70 % fill degree there are only very few buckets that have an 8th element. Because of the balancing insertions, there is a higher average displacement. However, the prefetching nearly eliminates the correlation between displacement and query time.

prefetching buckets

The effects of *chaining* are somewhat contrary to cuckoo hashing. The average displacement length is significantly reduced, but the dependence on the running time is significantly larger. Each additional displacement adds about 40 ns to the query time (about the same time as the first access).

Overall, the experiments show that the displacement distances are an indicator for running times, but an efficient access pattern can compensate these larger displacements.

2.5.2 Shared Memory

Concurrent hash tables that run on multi core processors have additional memory characteristics that have to be considered. Each processor has its own cache hierarchy that might also be shared with other processors. Additionally, there are hidden costs behind communications and synchronizations between processors.

cache coherence

One such hidden cost is the *cache coherence protocol*. Whenever a cache line is accessed by a processor, it is loaded into the corresponding cache. When one processor changes a value within the cache line, then existing copies in other caches need to be invalidated (to avoid accessing old data). This process is called *cache invalidation*. This is done automatically through the cache coherence protocol. It holds the state of each cache line that is stored in the cache. The states themselves are usually based on the MESI coherency protocol, i.e., (M)odified, (E)xclusive, (S)hared, (I)nvalidated. However, changes on the same chip are usually handled independently on the L3 cache level. Intel has recently changed the inner workings of their internal cache coherence protocol (for their processor generation codenamed skylake released 2017). For the purpose of this dissertation, the exact states are mostly irrelevant. It is important to know that writing to a cache line invalidates copies of said cache line on other processors, and that changing the state of a cache line incurs some overhead due to updates to other cached copies. Therefore, situations where multiple processors repeatedly write to the same cache line should be avoided. Sometimes this even happens unnecessarily, when multiple processors repeatedly write to different variables that are stored in the same cache line. We call this effect *false sharing*. False sharing can often be avoided by isolating relevant variables.

cache invalidation

false sharing

Whenever multiple processors access the same variables, it is important that all of them observe a consistent state of the variable and that no updates to the variable have the potential to get lost or become visible in an unwanted order (i.e., if one processor changes two variables the second change should not be visible before the first). The correct behavior can be ensured, by using *atomic variables* and *operations*. An atomic operation transforms the initial state of a variable to its resulting state without any intermediate states being observable. A variable that is only accessed through atomic operations is called an atomic variable. Many atomic operations follow the read-modify-write pattern. They couple read and write actions into one atomic transaction. This ensures that the state right before the operation is observed. Common atomic operations (see 2.2) include load, store, ex-

atomic operations/variables

Table 2.2: Common atomic operations and their effects.

name	C++ equivalent[4]	function
load	`load()`	loads an atomic variable
store	`store(x)`	stores the new value
exchange	`exchange(x)`	stores the new value, returning the previous value
fetch-and-add	`fetch_add(x)`	increments by `x`, returning the previous value
		(similar operations exist for: sub, and, or, xor)
compare-and-swap	`compare...`	1. loads the variable
	`... _exchange...`	2. compares its current value with `y`
	`... _strong(y, x)`	3a. if they are the same, store `x`
		3b. otherwise update `y` to current value

change, compare-and-swap (CAS), and fetch and add (other fetch and X operations).

Non-Uniform Memory Access

(NUMA)-node

Larger shared memory machines—especially multi-socket server hardware—have *Non-Uniform Memory Access (NUMA)*. There, processors are split into groups called *(NUMA-)nodes*, each group has its own "local" main memory which it has fast access to. All groups can also access each others memory, but accessing the local memory is faster than other group's memory. Most data structures like most hash-based data structures cannot avoid accessing other sockets. To make a concurrent data structure scalable on NUMA hardware it is necessary to reduce these accesses to a minimum. I.e., in a hash table there might be a slower access because the section of the table is stored on a certain node, however, there should not be a slow access because of some global metadata that could be duplicated on each node. Special problems may arise if the whole data structure is stored on the same node. In those cases, the bandwidth of this nodes data connection (i.e., QPI or UPI links[5], Infinity Fabric) might become a bottleneck.

2.5.3 Memory Mapping

(1) allocating virtual memory pages

(2) mapping to physical memory frames

(3) initializing memory

When allocating memory, there are three steps that usually happen before it can be used. (1) A section from the (virtual-)address space is allocated by calling `malloc`, which only reserves memory (usually constant time). (2) The virtual memory is

[4] all C++ functions take an optional parameter specifying the memory order. Using the correct memory order can be necessary for optimal performance.

[5] QPI and UPI links are trademarked by intel

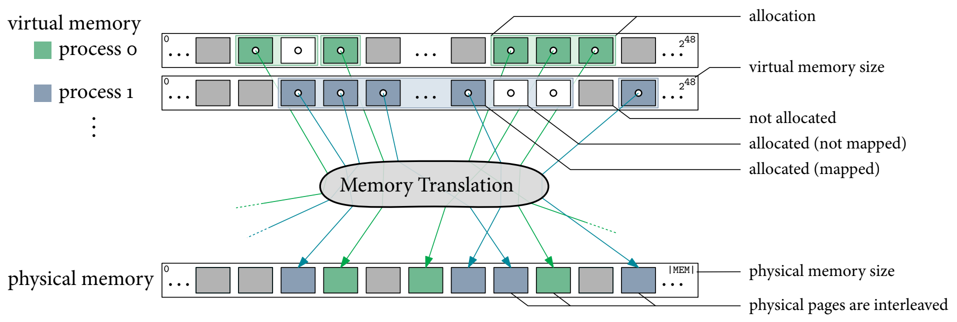

Figure 2.12: Schematic representation of the virtual to physical memory mapping.

organized in pages, before using a memory page, it has to be mapped to a *physical memory frame*. (3) Once the memory is mapped, it usually has to be initialized before it is used. The initialization actually triggers the mapping from virtual to physical memory. Therefore, steps (2) and (3) are connected—no physical memory frame is used before it is accessed and written to.

The mapping happens per virtual memory page (= physical memory frame). Therefore, the amount of physically used memory can be small even if there is a large amount of allocated virtual memory, as long as only a small subset of this memory has ever been accessed (see Figure 2.12). Usually, the virtual memory space is significantly larger, than the amount physically available memory. Thus, the operating system allows *overallocation*—allocating more virtual memory then there is physical memory[6]. Overallocation can be used to create a memory allocation that can grow over time for example to create a growing hash table that does not need any reallocations. This can be done by allocating enough memory to hold the maximum number of elements (i.e., ≈ physical memory size), then we only using the first $m \cdot s_e$ bytes (should be a multiple of the virtual memory page size), where s_e is the size of an element. Additional memory can be made available by simply accessing (i.e., initializing it for the first time). The true memory footprint (amount of physically used memory) is only the number of used bytes in addition to the waste of the last physical memory page.

overallocation

growing memory allocation

The mapping process itself is mostly controlled by the operating system. However, we observed that this mapping can be a bottleneck when called in parallel

[6]This feature can be turned off on some systems.

by many processors at once, e.g., when many processors initialize a large array in parallel.

address translation

Once the memory is mapped, each memory access leads to an *address translation* from the virtual to the physical address space. We only give a short description of the address translation, since it is invisible to the user and it cannot be influenced by programmers. Each access starts with a lookup in the *translation lookaside buffer (TLB)* it stores the frames for all recently accessed virtual memory pages. The TLB is a hardware cache that is a part of the memory management unit (MMU) of the CPU. The operating system is only involved in the translation, i.e., when there is a miss in the TLB. This is either the case, because the page has not been mapped yet (see above) or if the relevant page has not been accessed in a while (thus it was evicted from the TLB). In this case, the operating system translates the address using the *page table*. Depending on the operating system, the table can be implemented in different ways (usually in Linux systems it is implemented as a broad tree).

Translation Lookaside Buffer

3 Space Efficient Hash Tables

There are two commonly analyzed performance metrics that are analyzed for data structures: time per operation and memory consumption. Hash-based data structures in particular have an interesting tradeoff between performance and memory. Less space per element usually leads to slower operations. However there are architectures that are specifically designed for space efficient hashing.

One aspect that we explore throughout this section is the relation between dynamic table size and space efficiency. To build a space efficient hash table it is necessary, to initialize the table with the correct final size. This is only possible if there is a tight bound to the number of elements which is not the case for many practical workloads. Hash tables that can grow in a space efficient manner can guarantee space efficiency even for workloads where the final size is unknown prior to the execution. We show multiple ways to implement hash tables that grow in a space efficient manner. These variants are space efficient throughout their whole lifetime—not just once the capacity is reached and the table is densely filled. All implementations developed throughout this chapter are available publicly in our DySECT [42] library on github.

One common aspect of hash tables that has been studied a lot is their space efficiency [18, 22, 19, 25, 59, 68]. Thus, when researching a set of specialized but interchangeable hash table implementations, one should also pay some attention to hash tables focusing on space efficiency.

state of the art: space efficient only with static size

While current state of the art implementations work well even when filled to 95 % and more, they can only reach these fill degrees if they are initialized with the correct (final) capacity, thereby, requiring programmers to know tight bounds on the maximum number of inserted elements. This can often be unrealistic. Many typical hash table applications like aggregating data elements by their key, often have no guarantees on how many of their elements are unique. Whenever the exact number of unique keys is not known a priori, we have to overestimate the initial capacity to guarantee good performance. In these cases, we need dynamic space efficient data structures to guarantee both good performance and low memory overhead independent of the final number of elements.

To visualize this, assume the following scenario. During a word count benchmark (counting the number of duplicates by storing and incrementing counters in a hash table), we know an upper bound n_{max} on the number of unique words. Therefore, we construct a hash table with at least n_{max} slots. If an instance only contains $0.7 \cdot n_{max}$ unique words, no static hash table can have fill ratios greater than 70 %. Sometimes, dynamic data structures are necessary even if the final number of elements is known, i.e., when distributing a known number of elements onto a number of hash tables. Unless we know the distribution to be balanced, all tables must be initialized for the maximum possible load. In both cases, dynamic space efficient hash tables are required to achieve guaranteed near-optimal memory usage. In scenarios where the final size is not known, the hash table has to grow closely with the actual number of elements. This cannot be achieved efficiently with any of the current techniques used for hashing and migration.

no other library supports space efficient growing

Many libraries—even ones that implement space efficient hash tables – offer some kind of growing mechanism. However, all existing implementations either lose their space efficiency or suffer from performance breakdowns once the table grows above its original capacity.

the results in this section have been peer reviewed

The results presented in this chapter have been collected from our previous conference [45] and journal [49] publications. In this dissertation I use some of the texts verbatim or with fairly small changes. I was the main author for both of these publications. Thus, this text (aside from minor editing through my co-authors) was originally written by me. The theoretical analysis of maximum load bounds presented in [49] was originally written by *Stefan Walzer*. As such, I do not repeat the full analysis. Instead, I present some of the interesting threshold behavior that we noticed and also show some reasons why this happens—without reproducing the full theoretical analysis.

3.1 Contributions

We consider space efficient hash tables that can grow (and shrink) dynamically and are always highly space efficient, i.e., their space consumption is always close to the lower bound even during migrations and when taking into account storage that is only needed temporarily. None of the traditionally used hash tables (i.e., growing techniques) have this property. We show how known approaches like linear probing and bucket cuckoo hashing can be adapted to this scenario by subdividing them into many subtables or using virtual memory overcommitting. However, these rather straightforward solutions suffer from slow amortized insertion times due to frequent reallocation in small increments.

blueprint to adapt known hashing methods

Our main result is *DySECT* (**Dy**namic **S**pace **E**fficient **C**uckoo **T**able) which avoids these problems. DySECT consists of many subtables that grow efficiently by doubling their size. The resulting inhomogeneity in subtable sizes is counterbalanced by the flexibility available in bucket cuckoo hashing where each element can go to several buckets each of which contains several slots. Experiments indicate that DySECT works well with loads up to 98 %. With up to 35 % better performance than the next best solution.

DySECT Dynamic Space Efficient Cuckoo Table

In addition to our data structure designs, we also present an extensive experimental evaluation. Furthermore, we including some analysis of the interesting periodic behavior of DySECTs maximum load bound and previously unpublished experiments like an analysis of insert-delete workload.

3.2 Related Work

Over the last one and a half decades, the field of space efficient hash-based data structures—especially the design space of space efficient hash tables—has regained attention, both from a theoretical and practical point of view. The initial innovation that sparked this attention was the idea that storing an element in the less filled of two "random" spots (often called buckets) leads to very well balanced loads. This phenomenon is known as "*the power of two choices*" [58].

power of two choices

This concept led to the development of cuckoo hashing [68]. Cuckoo hashing extends the power of two choices paradigm by allowing insertions to move elements within the table to create space for new elements (see Section 2.4.2, page 37 for a more elaborated explanation). Cuckoo hashing revitalized research on space efficient hash tables. Probabilistic bounds on the maximum load [18, 22] and expected displacement distances [26, 27] are often highly non-trivial.

power of two choices + moving elements = cuckoo hashing

Cuckoo hashing can be naturally generalized in two directions to increase space efficiency: allowing $H \geq 2$ choices [25] or extending slots in the table to *buckets* that can store $B \geq 1$ elements [19]. We summarize this under the term (B, H)-*cuckoo hashing*.

more hash functions and larger buckets

Further adaptations of cuckoo hashing include: multiple concurrent implementations either powered by bucket locking [41], transactional memory [41], or fully lock-less [63]; a de-amortization technique that provides provable worst case guarantees for insertions [5, 35]; and a variant that minimizes page-loads in a paged memory scenario [23].

Some non-cuckoo space efficient hash tables continue to use linear probing variants. *Quadratic Probing* (see page 33) was already analyzed by Knuth [37] and performs better on densely filled tables. *Robin hood hashing* (see page 34), for example, is a technique that was originally introduced in 1985 [12]. The idea behind robin hood hashing is to move already stored elements during insertions in a way that minimizes the longest possible search distance. Robin Hood hashing has regained some popularity in recent years, mainly for its interesting theoretical properties and the possibility to reduce the inherent variance of linear probing.

Robin Hood hashing

All these publications show that there is a clear interest in developing hash tables that can be more and more densely filled. Dynamic hash tables on the other hand have not received much attention. The few papers we found (e.g. [20]) which take on the problem of dynamic hash tables predate cuckoo hashing, as well as much of the newfound attention for space efficient hashing. All memory bounds presented are given without tight constant factors. The lack of implementations and theory

significantly less interest in growing hash tables about dense dynamic hash tables is where we pick up and offer a fast hash table implementation that supports dynamic growing with tight space bounds.

3.3 Defining Space Efficiency

Tables can usually only operate efficiently up to a certain maximum load factor. Above that, operations get slower or have a possibility to fail. When implementing a hash table, one has to decide between storing elements directly in the table—*in-table storing* (sometimes called *closed hashing*)—or storing pointers to elements—*out-of-table storing* (sometimes called *open hashing*). This has an immediate impact on the amount of memory required—in-table storage uses $m \cdot s_e$ and out-of-table uses at least $m \cdot s_p + n \cdot s_e$ (chaining needs an additional $n \cdot s_p$) where s_e is the size of an element and s_p is the size of a pointer.

in-table vs. out-of-table

For large elements (i.e., much larger then the size of a pointer), one can use a non-space efficient hash table with out-of-table storing to reduce the relevant memory overhead. Therefore, we inspect the common and more interesting case of elements whose size is close to that of a pointer. For our experiments we use 128 bit elements (64 bit keys and 64 bit values). In this case, out-of-table storing introduces a significant memory overhead (at least 50 % for 64 bit pointers). For this reason, we only consider in-table storing hash tables. Their memory efficiency is directly dependent on the table's load. To reach high loads with *in-table storing*, we have to employ *in-table displacement* techniques (see Section 2.4.2). This means that elements are not stored in predetermined slots, but can be stored in one of several possible places (e.g., linear probing, or cuckoo hashing).

large elements → out-of-table

small elements → in-table

3.3.1 α-Space Efficient Hash Tables

Static

We call a hashing technique α-space efficient when it can work efficiently using at most $\alpha \cdot n \cdot s_e + O(1)$ memory (where s_e is the size of an element). An intuition for what we mean by working efficiently is having find performance similar to an unfilled table; and having insertion times close to $\approx (1-\delta)^{-1}$. This is the expected number of fully random probes needed to hit an empty slot, and therefore, a natural estimation for insertion times (i.e., insertion time in a table with multi hashing—see Section 2.4.2, page 29). This definition is in some ways intentionally vague because we want to include machine specific benefits (like cache line usage and prefetching, see Section 2.5.1) of techniques like linear probing that are hard to translate into models.

α-space efficient $\delta \geq \alpha^{-1}$

When a table uses in-table storing (e.g. linear probing, cuckoo hashing) slots usually have the same size as elements. Therefore, being α-space efficient is the same as operating with a load of $\delta \geq \alpha^{-1}$. Because of this, we will mostly talk about the load of a table instead of its memory usage.

Dynamic

The definition of a space efficient hashing technique given above is specifically targeted for statically sized hash tables. We call an implementation *dynamically α-space efficient* if an instantiated table can grow arbitrarily large over its original capacity while remaining smaller than $\alpha \cdot n_{\max} \cdot s_e + O(1)$ at all times.

full table migrations are not space efficient

dynamic α-space efficiency $\delta_{\min} \geq \alpha^{-1}$

One problem for many implementations of space efficient hash tables is the migration. During a normal full table migration, both the original table and the new table are allocated. This requires $m_{new} + m_{old}$ slots. Therefore, a normal full table migration is never more than 2-space efficient. The only option for performing a full table migration with less memory is to increase the memory in-place (see Section 3.7). Similar to static α-space efficiency, we will mostly talk about the *minimum load factor* $\delta_{\min} \geq \alpha^{-}1$.

3.3.2 Ideas for Space Efficient Growing

classic growing mechanisms do not work

Growing is commonly implemented either by creating additional hash tables—decreasing performance especially for lookups (thus, this approach is only feasible in a very limited capacity)—or by migrating to a new table—causing a temporary drop of space efficiency to at best 50 %. The reason being that at least for the time of the migration two tables exist. Thus even when the original table was filled to 100 % and the migration does not change the size (by more than a constant) there are at least two slots per element.

The classical way to circumvent the memory overhead of full table migrations is to split the table into subtables that can be migrated individually. If there are T subtables, growing one subtable individually reduces the necessary memory overhead to m/T (if subtables are balanced), because only m/T slots are duplicated. We use this method both for our updated hashing methods (in Section 3.4) and also within DySECT (Section 3.5).

In Section 3.4 we also present another technique that avoids the memory overhead of full table migrations. We propose an in-place growing technique that uses memory overallocation (see Section 2.5.3) to increase the tables capacity in-place. By "creating" new memory at the end of the table. Afterwards we can use a novel in-place migration technique to reorder the elements to fit the new table size. The proposed in-place migration is flexible enough to be adapted to many existing hashing schemes. with only minor differences.

3.4 A Blueprint for Dynamic α-Space Efficiency

In this section we describe a blueprint that can be used to construct dynamic space efficient variants from many static hashing methods. This blueprint consists of two aspects: speed up the table migration to counteract frequent small growing steps and prevent the memory overhead during migrations.

- speedup migration
- prevent migration overhead

We use the following commonly known hashing methods: *linear probing*, *quadratic probing*, *Robin Hood hashing*, and *cuckoo hashing*. All these methods can fill static tables to close to 100 % (albeit losing performance). Both hashing with chaining, and hopscotch hashing are not statically space efficient, because they store additional per-slot-data (for values of α that are interesting to us; the queue pointer and neighborhood bitmaps respectively). We tested hopscotch hashing in some preliminary experiments, but needed neighborhoods larger than 64 to reach our targeted loads (90 % and more), therefore, creating the same kinds of overheads as chaining.

Growing a hash table usually means allocating a new table, and migrating all elements from the current *"source"* table to the new *"target"* table. Even when neglecting the memory constraint during the migration itself, being space efficient means that the table has to be filled very densely before growing. Additionally, the table can only grow in small steps because it still has to be filled more than $\delta_{\min}$ after growing. This leads to many small growing steps (each one growing only by small factors) which can be very inefficient if migrations are small.

when to grow and by how much

To achieve a minimum load of $\delta_{\min}$ (up to $\alpha = \delta_{\min}^{-1}$-dynamic space efficient), we start growing once the table has a load of $\delta_{trigger} = \frac{\delta_{\min}+1}{2}$ and we grow the table such that it fulfills the minimum load ($n \cdot \delta_{\min}^{-1}$). This method, of keeping the table's load between $\delta_{\min}$ and $\delta_{trigger}$ leads to many repeated full table migrations. As such, it becomes even more important that each migration is as fast as possible.

3.4.1 Amortizing Frequent Growing

Even the frequent migrations that are necessary to grow the table with the number of elements can be amortized if the table is statically $\delta_{trigger}^{-1}$-space efficient (i.e., if its operations remain efficient until $\delta_{\min}$ is reached). We outline a quick proof using the token method. We define $\delta_{\Delta} = \delta_{trigger} - \delta_{\min} = (1 - \delta_{\min})/2$. Between two migrations, i.e., from m to m' slots, we insert $\delta_{\Delta} m$ elements. If each insertion pays δ_{Δ}^{-1} tokens (constant number), then we have m tokens when the table is supposed to grow, allowing us to pay for the expected running time of the migration $O(m)$. From the number of tokens necessary for each element, we see that the cost of frequent migrations scale inversely with $\delta_{\min}$.

3.4.2 Fast Table Migration

needs linear mapping

To implement fast table migration, we need to use the correct mapping from preliminary hash function to the table position. As described in Section 2.3.2 there are two natural ways to map a hash value $h(e)$ (large binary number) to a slot in the table (index $0..s-1$). The circular mapping using a modulo operation ($h(e) \mod s$), or a linear mapping using one of the scaling methods described in Section 2.3.2. We also see that the linear mapping (using fastrange) is actually significantly faster, than the circular approach on arbitrary table sizes (*Note:* Arbitrary table sizes are necessary for space efficiency).

elements quasi sorted by preliminary hash function

Aside from being the faster of the two mapping opportunities, this mapping leads to the elements in the table being quasi sorted by their preliminary hash value (in the absence of collisions they would be fully sorted). This in turn helps us make

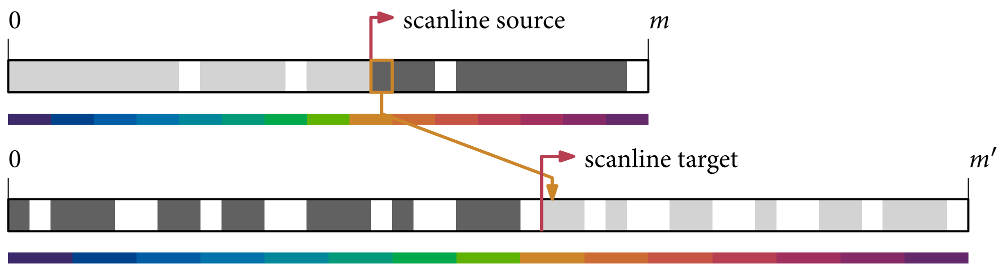

Figure 3.1: Schematic representation of our fast migration algorithm.

the migration cache efficient. We use the same preliminary hash function and mapping technique in the new table, therefore, after the migration the elements are still quasi sorted by the same preliminary hash values.

Thus, the order of elements in the target table is close to the order of elements in the source table. Therefore, scanning through the source table from beginning to end, reinserting each element into the target table is similar to scanning through the target table (as shown in Figure 3.1). This parallel scan of both tables is significantly more cache efficient and thus faster than other migration techniques (i.e., ones that do not use a linear mapping) that have random access patterns. This method works similar for many of the most common hashing methods.

scan through both tables

- *linear probing*: elements within one cluster (consecutive filled slots) are not sorted by their preliminary hash. However, clusters are (usually) small relative to the grow amount, making the migration highly efficient on average.
- *quadratic probing*: in general, quadratic probing offers very short displacements, therefore, many elements will benefit from the cache efficiency.
- *Robin Hood hashing*: this method can be adapted such that elements are purely sorted by their preliminary hash value. Therefore, making the migration as efficient as possible.
- *cuckoo hashing*: cuckoo hashing does not map directly to this parallel scan approach. To still profit from the same effect, we use the following approach. When migrating an element, we first check which of its H hash functions was used to store the element. If possible that hash function is also used to

store the element in the new table. However, due to the linear mapping it is possible that buckets in the target table overfill. We insert these elements using the normal insertion algorithm.

3.4.3 Preventing Overhead During the Migration

As we mentioned before (in Section 3.3.1) there is one problem with classical full table migrations. During the migration there are two tables (both $\geq n$), the source table, and the target table. Bounding the effective load to below 0.5, thus the table can only be dynamic $\alpha = 2 + \varepsilon$-space efficient. We propose two solutions to this problem, both have their own strengths and weaknesses.

In-Place Migration

virtual memory overcommitting

Using *virtual memory overcommitting*, it is possible to increase the size of an allocation in-place (see Section 2.5.3). The idea is that the operating system only maps virtual memory pages to physical memory frames once the virtual memory pages are accessed for the first time. Thus, for the purpose of space efficiency the memory is not yet used. When we construct the hash table at the beginning of a large memory allocation initializing more memory is similar to increasing the tables effective memory size. However, after increasing the capacity of a table, previously inserted elements are out of place and have to be reordered into their new positions. Luckily, the methods described in Section 3.4.2 allow us to implement a cache efficient in-place migration algorithm.

migrate back to front

The main idea of our in-place migration techniques is to again use the fact that the order of elements does not actually change much, because the linear mapping leads to the elements being sorted by their preliminary hash value. Thus, elements that are in the back of the source table will also be towards the back of the target table. The plan is to scan the source table from back to front, migrating each element when it is encountered by the scan (compare Figure 3.2). When migrating the table in this way most migrated elements are hashed to the area behind the scan line. Thus, as long as their probing does not cross the scan line they cannot interfere with the rest of the migration (i.e., linear probing). Elements that are hashed in front of the scan line are stored in a separate overflow buffer and are reinserted after

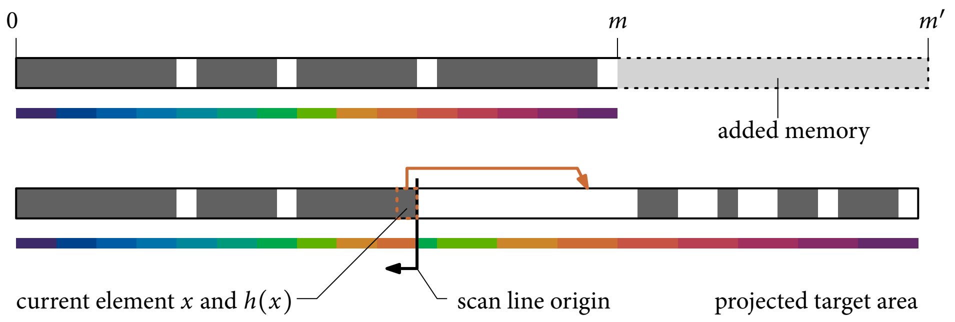

Figure 3.2: Schematic view of the in-place growing algorithm. *Top*: original table with added memory. *Bottom*: Ongoing algorithm showing both scan lines.

the scan is finished. For some hashing techniques, the overflow buffer may contain a lot of elements. In these cases we can store the overflow elements in the empty slots at the end of the table (interleaved with elements that are already migrated). At the end of the migration we can rehash them to find their final slots. Similar to the technique described in Section 3.4.2 this migration accesses both the source and the target table in a cache efficient access pattern (here in reverse order).

This in-place table migration using virtual memory achieves good performance. But, there are some problems with the technique of virtual memory overcommitting. Memory can only be increased by at least a memory page at a time (memory pages can be large on some systems). Additionally, these memory pages cannot be unmapped after they were accessed making shrinking the data structure after deletions impossible (without deallocating the whole memory). The amount of virtual memory per application can be limited by the operating system making it impossible to have many of these overcommiting memory regions at once or virtual memory overcommitting could be turned off entirely. Therefore, virtual memory overcommitting is sometimes considered bad practice, especially in codes that are considered to be portable. Therefore, we report numbers for both scenarios (with and without virtual memory overcommitting).

problems with overcommitting
- memory page size
- OS configuration
- no shrinking
- portability

Multitables

For this method, we split the table into T subtables (T is constant). On the top level, we hash each element to one of the subtables, i.e., each element has one *canonical*

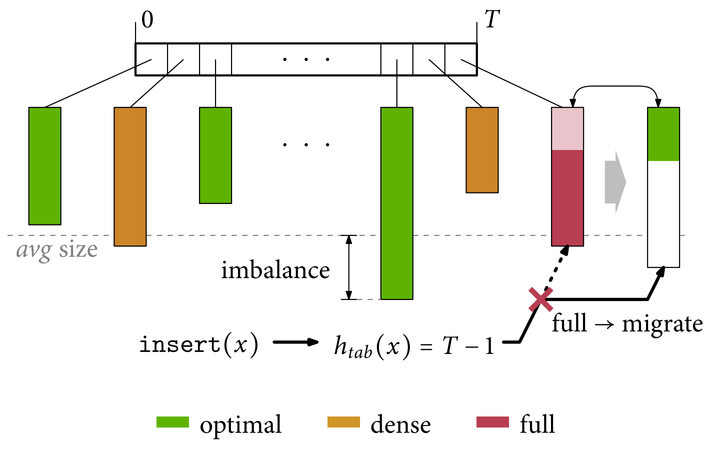

Figure 3.3: Multi-table approach. Currently inserting element x into the table. The insertion triggers a migration which is also shown.

canonical table *table*. After that we use one of the classic hashing schemes to store elements within the subtables. The benefit of this technique is that each subtable can be grown independently. Subtables grow their capacity in the classical manner of allocating a new table and doing a full table migration (using the fast migration techniques Section 3.4.2).

Figure 3.3 shows a table being grown. The overhead of having both the source and the target (sub)table allocated at the same time is reduced to a small constant factor of the overall memory $m_{grow} \approx \alpha n \cdot \frac{T+1}{T}$. This could be worse for instances with bad element imbalances, i.e., when there is one subtable that is significantly larger than the rest. However, if the top level hash function is sufficiently random, the distribution behaves like the well known balls into bins problem. Raab and Steger [79] show that number of elements in the largest subtable is smaller than $n/T + \Theta\left(\sqrt{\frac{n \log T}{T}}\right)$ with high probability—assuming $n > T \log T$. But even small differences can be bad for the overall size, e.g., when the size estimation at the time of construction was correct, or the final size is known, then we would preferably initialize each table with $\alpha \cdot n/T$ slots. Thus, even small imbalances could lead to a few grown subtables and other subtables being less densely filled. Leading to a worse than expected overall fill ratio. Overall, the final capacity could be larger than $\alpha \cdot n$. In our tests most multi-level variants turned out to be slower on average than the tables using in-place techniques.

3.5 DySECT (Dynamic Space Efficient Cuckoo Table)

doubling is efficient

A commonly used growing technique is to double the size of a hash table by migrating all its elements into a table with twice its capacity. This is of course not memory efficient. The idea behind our dynamic hashing scheme is to double only parts of the overall data structure. This increases the space in part of our data structure without changing the rest. But due to our usage of cuckoo displacement techniques this indirectly relieves pressure from other parts of the hash table as well.

3.5.1 Overview

parameters
H number of hash functions
B number of slots in a bucket
T number of subtables

Our DySECT hash table consists of T subtables (shown in Figure 3.4) that in turn consist of buckets, which can store B elements each. Each element has H associated buckets—similar to cuckoo hashing – which can be in the same or in different subtables. T, B, and H are constants, which cannot change during the lifetime of the table. Additionally, we pick a minimum fill ratio $\delta_{\text{min}} \in (0,1)$. The table will never exceed $\delta_{\text{min}}^{-1} \cdot n$ slots once it begins to grow over its initial size (appropriate choices for δ_{min} are discussed later).

mapping elements to buckets

To find the buckets associated with an element e, we compute e's preliminary hash values $h_i(e)$ using appropriate hash functions $h_i, (1 \le i \le H)$. These hash values have to be mapped to buckets. This mapping works in two steps (1) first each hash value is mapped to a table, (2) then the value is mapped to a bucket within said table. The construction of DySECT allows us to use powers of two for both the number of subtables T and for the number of buckets per subtable. Therefore, both mappings can be done with simple bit operations. The preliminary hash value is split into two parts $h_{tab\ i}$ and $h_{in\ i}$. (1) The first $\log T$ bits (i.e., $h_{tab\ i}$) are used to address the table using a direct mapping. (2) The other bits (i.e., $h_{in\ i}$)are used to map to a bucket within said table (using a linear mapping). Since the subtable size is also a power of two, the linear mapping can be implemented using simple bitwise operations.

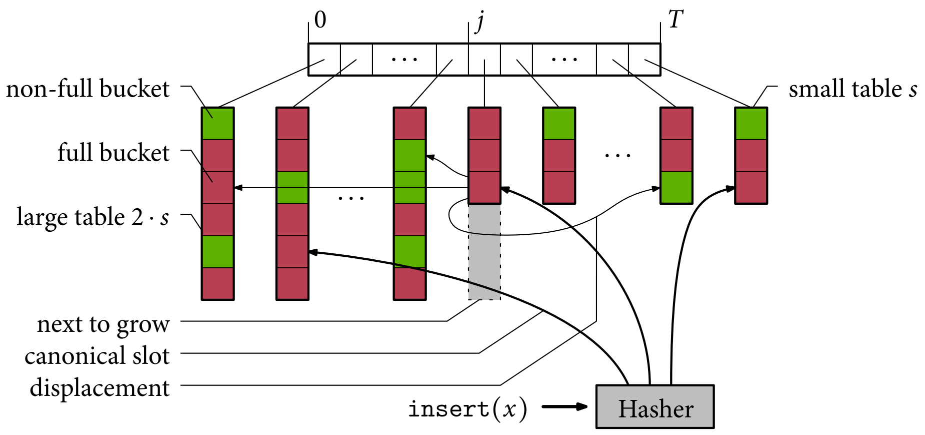

Figure 3.4: Schematic representation of a DySECT table. During the insertion of an element x, showing possible displacements.

displacement paths connect tables

Aside from the bucket computations, insertions work the same way they do in other cuckoo tables by either inserting into the emptiest associated bucket or by finding a displacement path. The interesting idea behind DySECT is that displacements allow us to balance the subtable loads and utilize memory in one subtable to make room in another (denser) subtable and thus enable insertions even if the canonical slots are all filled.

3.5.2 Growing

greedy growing

As soon as the (overall) table contains enough elements such that the memory constraint can be kept during a subtable migration, we grow one subtable by migrating it into a table twice its size. We migrate subtables in order from first to last. This ensures that no subtable can be more than twice as large as any other.

no bucket overflows no displacements necessary

Doubling the size of one subtable is very easy and cache efficient. The new subtable again uses the same preliminary hash function as before, thus, the mapping uses one additional bit. This leads to each source bucket being split into two target buckets. Thus, there cannot be any overfilled buckets in the target table and also no necessary displacements that could be triggered. Similar to all methods described in Section 3.4.2 the migration can again be implemented as a cache efficient simultaneous scan through source and target tables.

Using the implicit graph model of the cuckoo table (described in Section 2.4.2, page 37), we can explain how growing one subtable increases insertion opportunities into the rest of the table. Here, growing a subtable is equivalent to splitting each node that represents a bucket within that subtable. The resulting subgraph becomes more sparse, since the edges (elements) are not doubled. This is true for both outgoing edges—elements that are stored within the grown subtable—and incoming edges—elements that are stored elsewhere but that have an associated bucket within the table. many of these incoming edges now point to half-filled buckets, and therefore, offer new displacement opportunities in the rest of the table.

new insertion opportunities for the whole table

Now lets look at some numbers connected to growing the table. When the data structure contains j large subtables (with $2s$ slots) then there are $m = (T + j) \cdot s$ slots. When $\delta_{\min}^{-1} \cdot n > m + 2s$ we can grow the first subtable while obeying the size constraint (the newly allocated table has $2s$ slots). Doubling the size of a subtable increases the global number of slots from $m_{old} = (T + j) \cdot s$ to $m_{new} = m_{old} + s = (T + j + 1) \cdot s$ (growth factor $\frac{T+j+1}{T+j}$). Note that all subsequent growing operations migrate one of the smaller tables until all tables have the same size. Therefore, each grow until then increases the overall capacity by the same absolute amount (smaller relative to the current size). The cost of growing a subtable is amortized over all insertions since the last subtable migration. There are $\delta_{\min} \cdot s = \Omega(s)$ insertions between two migrations. One migration takes $\Theta(s)$ time.

growing trigger

growth factor

3.5.3 Shrinking

In most workloads, shrinking is not actually necessary. Usually the goal with a space efficient data structures is to remain small enough to fit into the main memory (or into another similar memory bound). Thus, once you have used a certain amount of memory there is often no use in reducing the memory. This assumption is also present in the STL-implementation of many data structures where there is no automated shrinking. Instead STL-data structures usually have a function called *shrink to fit* that shrinks the data structure to a size relative to the current elements. There are workloads where automated shrinking is preferable. This is usually the case when elements are moved between different kinds of data structures (e.g., many different space efficient tables) and the overall size has to be bounded.

shrink to fit

If shrinking is necessary it can work similarly to growing. We replace a subtable with a smaller one by migrating elements from one to the other. During this migration we join elements from two buckets into one. Therefore it is possible for a bucket to overflow. We store these elements at the front of the source table and reinsert them at the end of the migration. Obviously, this can only affect at most half the migrated elements. A shrink to fit operation can be implemented in a similar manner, potentially shrinking multiple subtables.

buckets can overflow

When automatically triggering the size reduction, one has to make sure that the migration cost is amortized. Therefore, a grow operation cannot immediately follow a shrink operation. When shrinking is enabled we propose to shrink one subtable when $\delta_{\min}^{-1} \cdot n < m - s'$ elements (s' size of a large table, $m_{new} = m_{old} - s'/2$).

shrinking trigger

Avoiding memory overhead while shrinking

The current implementation of shrinking actually uses more memory temporarily during the migration to a smaller table. In theory it would be possible to avoid this additional memory by shrinking two large tables A and B at the same time. This can be done by first reducing the used memory of A in-place. By combining the appropriate buckets within A to the front half. Reinserting any overflowed elements. Then all elements in B are migrated into the second half of table A, again reinserting overflow elements. Thus table B can be deallocated. Lastly we allocate two new smaller tables copying the first and second half of table A (afterwards deallocating A). Alternatively we could keep table A and change the pointer that used to point to B such that it points to the second half of table A.

3.5.4 Implementation

Each DySECT-table is implemented as an array of T pointers to its subtables. Whenever a subtable is accessed its pointer is accessed, and the offset of the accessed bucket is added to this pointer. The performance impact of this pointer lookup is fairly low, since all subtable pointers will be cached—at least if the hash table is a performance bottleneck.

Default Parameter Choices

To choose the default parameters, we made a few preliminary experiments using a varying number of hash functions and bucket sizes. We tested both insertions and finds on a statically sized table as well as constructing a table space efficiently by growing from a small initial size.

Given the results shown in Figures 3.5, we use 4 hash functions $H = 4$ and a bucket size of $B = 4$ as default parameters. These values have outperformed other options in terms of *insert* performance (especially into a growing table) and they deliver a particularly good tradeoff between find performance (consistently faster than $B = 8$ $H = 3$) and maximum load (see Section 3.6). The default number of subtables T is set to 256. It has very little impact on the performance, but it impacts the minimum size of the table $m_{\text{min}} = T \times B$ and the minimum growing factor which changes the space efficiency (see Section 3.5.2). If not mentioned otherwise, these are the parameters we use throughout our evaluation of DySECT, e.g., for our experimental evaluation in Section 3.8.

default parameters
$H = 4,\ B = 4,\ T = 256$

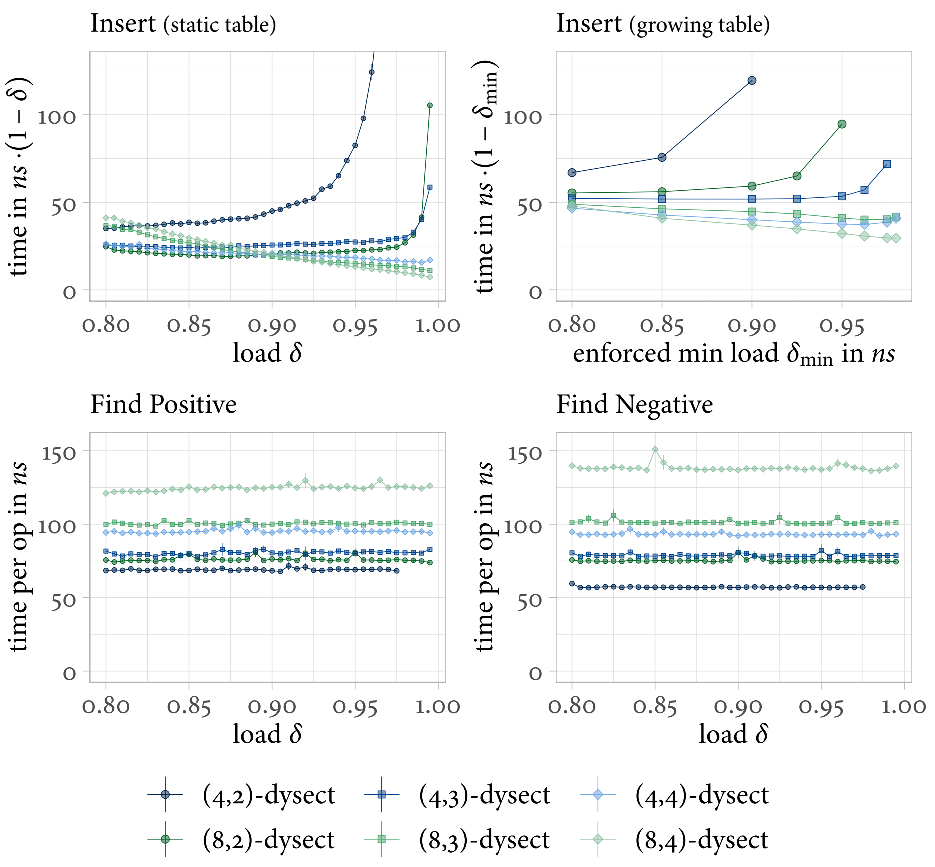

Figure 3.5: Benchmarking operations on dysect tables with different parameters. *Topleft*: insertions with a static table size of 2^{24} slots measuring the performance at different points (1000 operations sample size; performance normalized with $1/\delta$). *Topright*: average insertion time inserting into a dynamic space efficient table averaged over 10 000 000 insertions into a table starting at 50 000 slots. *Bottomleft* and *Bottomright*: performance of positive and negative find queries respectively (experimental setup like in *Topleft*)

Displacement Algorithm

In addition to the parameters described above there remains the choice of a displacement algorithm. This algorithm is executed if all buckets associated with an element are full. It finds a sequence of moves that results in creating an empty slot in one of the associated buckets. As we mentioned in Section 2.4.2 (page 37) there are two general ways, of exploring the table for displacement opportunities. A *breadth first search* of the implicit graph, or a *random walk* (i.e., depth first search) through the graph. Both work on the implicitly defined graph, where buckets are nodes and elements are edges from the node they are stored in to their other associated nodes.

finds moves to create an empty slot

breadth first search
random walk

Bounded BFS The BFS algorithm explores the associated buckets of the inserted element. For each element encountered in these buckets, we insert their potential buckets into a queue (remembering the appropriate element). Then we repeat this procedure with every bucket in this queue, until we find a bucket that has empty slots remaining. Afterwards the chain of elements that lead to this bucket is reconstructed and the corresponding moves are performed.

Contrary to classic BFS implementations, we do not check whether any bucket has been accessed by the BFS before. This makes us look through parts of the hash table multiple times (if one bucket is reached by more than one paths), but it improves the overall speed and memory footprint. Our hope is that usually BFS searches are short and thus the unnecessary work remains small. Additionally, we only use a bounded BFS because without checking bucket repetitions, the search space would become infinite otherwise. If not stated otherwise we use a bound of 8192 visited buckets (this number is chosen arbitrarily).

rechecking the same bucket

(Optimistic) Random Walk The random walk technique corresponds loosely to a depth first search in the graph view of the insertion problem. In the sense that at each node, we explore only one of the outgoing edges (backtracking does not happen since that would mean we found an empty bucket). When inserting an element x if all buckets that are associated with x are full, then we move one random element x' from one of the explored buckets to one of its other associated buckets—

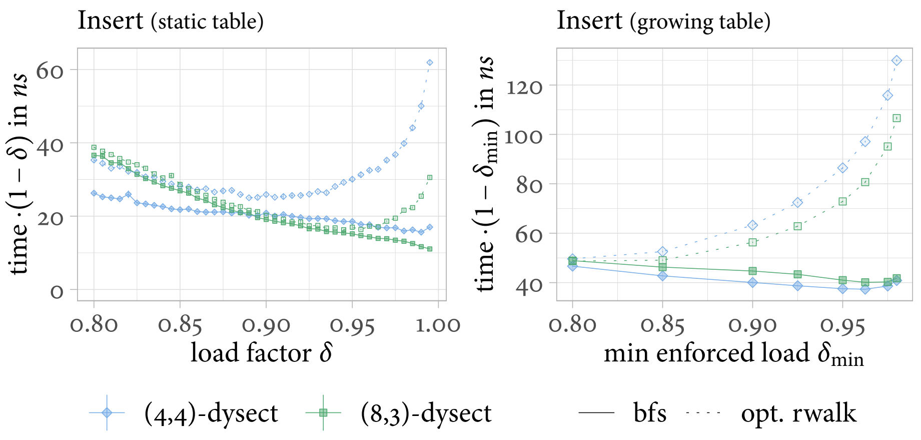

Figure 3.6: Benchmarking operations on DySECT tables with different displacement algorithms. *Topleft*: insertions with a static table size of 2^{24} slots measuring the performance at different points (1000 operations sample size; performance normalized with $1/\delta$). *Topright*: average insertion time inserting into a dynamic space efficient table starting averaged over 10 000 000 insertions into a table starting at 50 000 slots.

freeing up its slot to insert x. Then this procedure is repeated now inserting x' until an element can be stored within a non-full bucket.

optimistic random walk

There are two kinds of random walk. (1) The *optimistic random walk* stores only the currently replaced element. Thus the memory footprint is independent of the number of displacement steps. This variant is useful when the memory footprint of the displacement technique has to be small. The main problem with this variant happens when no displacement path is found. Then, the last replaced element cannot be reinserted and we cannot rollback the insertion. This is a problem because applications expect elements to remain in the table after they have been inserted once. (2) the *pessimistic random walk* only looks for a possible replacement path, without actually doing any replacements. Only once a path is found do we execute all replacements at once.

pessimistic random walk

Default Choice We repeated the same test we used to choose the default parameters, this time, with different displacement algorithms, see Figure 3.6. In our preliminary tests, BFS displacement outperformed random walk displacement. It has less variance and a better cache line utilization, since all elements within one bucket are used to find displacement opportunities. Therefore, we use the (bounded size) breadth first search (BFS) as our default displacement technique.

Reducing the Number of Computed Hash Functions

double hashing

Our default implementations use the double hashing techniques described in Section 2.3.3. This allows us to construct an arbitrary number of 32 bits hash values by computing only one 64 bit hash function. The resulting preliminary hash is then split into two parts that are combined linearly, to obtain a series of hash values.

Virtual Memory Overcommitting

In Section 3.4.3 we describe how virtual memory overcommiting (see Section 2.5.3) can be used to implement growing hash tables that grow in-place. DySECT does not need tricks like memory overcommitting to remain memory efficient since subtable migrations do not present the same memory overhead. But, the DySECT implementation can also use these same techniques. In our experiments (Section 3.8) we show results with and without using this overcommitting technique. It turns out that the traditional variants are even faster then the ones using overcommitting.

removing pointer lookup

Accessing a DySECT subtable usually takes one indirection. The pointer to the subtable has to be read from an array of pointers before accessing the actual subtable. Instead of using an array of pointers, we can implement the subtables as sections within one large allocation (size u). We choose u larger than the actual main memory, to allow all possible table sizes. This has the advantage that the offset for each table can be computed quickly ($t_i = \frac{u}{T} \cdot i$), without looking it up from a table.

in-place migration of subtables

The advantage is that we can grow subtables in-place. To increase the size of a subtable, it is enough to initialize a consecutive section of the table (following the original subtable). Once this is done, we have to redistribute the table's elements. This allows us to grow a subtable without the space overhead of reallocation. Therefore, we can grow earlier, staying closer to the minimum load factor $\delta_{\min}$. The in-place growing mechanism is even easier within DySECT, since the subtable size is doubled and there is no possibility for bucket overflows.

3.6 Analysis of Possible Loads

Usually, cuckoo hash tables cannot be filled up to the last slot. Instead at some point there is an insertion that fails because there is no path in the implicit graph representation that leads to a non-full bucket, i.e., all paths in the implicit graph representation (described in Section 2.4.2, page 37) end in circles.

feasibility threshold δ^*

Most cuckoo hashing schemes exhibit what can be described as threshold behavior, i.e., each hashing scheme has a specific *feasibility threshold* δ^* where loads that are smaller than the feasibility threshold (i.e., $\delta < \delta^* - \varepsilon$) are feasible with high probability[1] (i.e., all elements can be inserted) and loads above that threshold (i.e., $\delta > \delta^* + \varepsilon$) are not feasible with high probability[1]. Examples for such load thresholds are 97.8 % for $(4, 2)$-cuckoo hashing or 99.98 % for $(8, 3)$-cuckoo hashing.

feasibility vs. practicability

Throughout this section, we analyze the feasibility bounds of DySECT. These bounds can be reached by exploring the whole search space for displacements. Reaching these bounds is generally not efficient in the sense of having reasonable insertion times (which we defined as approximately $O\big((1-\delta)^{-1}\big)$). But knowing these bounds is integral for choosing appropriate $\delta_{\min}$ bounds for our growing tables. Additionally, none of our displacement algorithms are truly equipped to explore the full graph (BFS are intentionally bounded and random walks could take an arbitrarily long time).

same as cuckoo when subtables are identical

imbalance between buckets

For (B, H)-DySECT we expect load thresholds that are comparable with (B, H)-cuckoo hashing. Especially when all subtables have the same size. When all tables have the same size there is actually no difference between DySECT and classic cuckoo hash tables. But due to the uniform mapping from elements to subtables there are imbalances between buckets in larger and smaller tables. These imbalances can lead to different feasibility thresholds depending on the ratio between large and small tables. The table mapping $h_{tab\ i}$ is (see Section 3.5.1) independent of the subtable size. Therefore, each table has the same expected number of elements that are associated with it. Since large tables have more buckets there are less elements associated with each individual bucket (than with individual buckets in small tables)—in expectation.

[1]probability $p > 1 - o(1)$

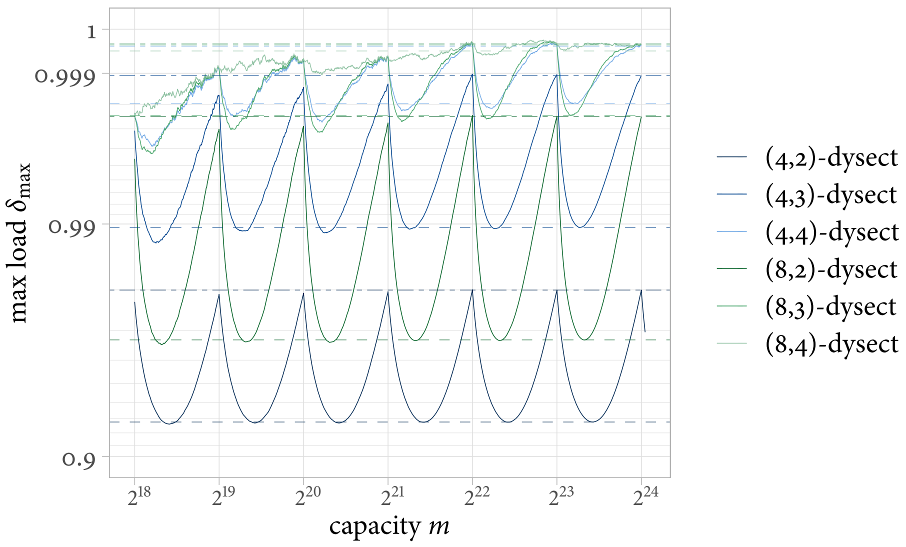

Figure 3.7: Experimentally determined periodic maximum load bounds. Elements are inserted until an insertion fails (500 000 probed buckets), then one subtable is grown.

To prove some more formal bounds and feasiblity thresholds we cooperated with Stefan Walzer. The resulting theoretical work can be found in our joined publication [49]. In this dissertation, we present a shorter intuition into the problem, and an experimental analysis. *outlook on formal bounds*

3.6.1 Experimental Load Bounds

Table 3.1: Experimental load bounds. Extracted from the measurement presented in Figure 3.7 (using the last period $n > 2^{23}$) together with theoretical bounds [49]

B	H	exp. min	theo. min	exp. max	theo. max
4	2	0.9278	0.9566	0.9800	0.9855
4	3	0.9896	0.9984	0.9997	1.0000
4	4	0.9980	0.9980	0.9997	1.0000
8	2	0.9671	0.9915	0.9989	0.9993
8	3	0.9976	0.9766	0.9975	0.9980
8	4	0.9996	0.9999	0.9997	1.0000

To measure the maximum load bound experimentally, we initialize a table with ≈ 50 000 slots (256 subtables). Then we insert elements until an insertion fails, i.e., does not find an appropriate displacement (within 500 000 probed buckets, intentionally high to reduce variance). Once an insertion is unsuccessful, we measure the current load and grow one subtable. We repeat this procedure until 1 000 000 elements have been inserted. We repeat the same experiment with 5 different sets of keys (1 000 000 keys each) to reduce the probability of randomly picking a bad input (averaging the load of each level between the five runs).

grow subtable once insertion fails

The results of the experiment are shown in Figure 3.7. As expected, the table behaves the best when the size is a power of two (i.e., when all subtables have the same size). Between these maxima, the load bound exhibits an interesting periodic behavior. The maximum load threshold depends on the ratio between large and small subtables (i.e., ratio is 0 at powers of two), not on the absolute number of slots or any other factor.

Comparing the different parameter choices, we see that variants with fewer hash functions exhibit a stronger dependence towards the ratio of large vs. small tables than larger H.

3.6.2 Simple Explanation Using Local Effects

To remove some of the complexities of analyzing maximum load bounds we try to look only at local effects that are independent of moving elements, i.e., without using the displacement graph. Elements can only be stored in one of their canonical buckets, therefore, if a table or a bucket has fewer associated elements (elements that are hashed to this table or bucket) than slots, then this table or bucket can never be 100 % filled.

argument over per bucket distribution

Table Imbalance By growing subtables individually we introduce a size imbalance between subtables. Large subtables contain more buckets but the number of elements hashed to a large subtable is not larger than the number of elements that are hashed to a small subtable. This makes it difficult to spread elements evenly among buckets. Assume there are n elements in a hash table with T subtables, j of which have size $2s$ the others have size s. If elements would be spread evenly among

the number of elements hashed to a table does not depend on the tables size buckets then all small tables store around $n/(T+j)$ elements, and the bigger tables store $2n/(T+j)$ elements. For each table we expect about Hn/T elements that have an associated bucket within that table. This also shows that having more hash functions can lead to a better balance.

For example, in a table using two hash functions ($H = 2$) and only one grown table ($j = 1$) this means that $\approx 2n/(T+1)$ elements should be stored in the first table to achieve a balanced bucket distribution. Therefore, nearly all elements associated with a bucket in the first table ($\approx 2n/T$) have to be stored there. This might be one reason why $H = 2$ does not work well in practice.

there are not enough elements hashed to a table

Individual Buckets This same style of argument can be formalized on a per bucket level. Let us assume a set E of $n = m$ elements—not all of them can be inserted thus the following calculations derive optimistic estimates. We say the *degree* of b is $deg(b) = |\{e \in E | \exists h_i \;\; with \;\; h_i(e) = b\}|$, i.e., the number of elements that are hashed to b with at least one hash function. This number can be estimated using the balls into bins model. Each element casts H balls, thus overall mH balls are being cast (note: $n = m = (j+T) \cdot s \cdot B$). However, not all bins have the same probability. The probability of hitting a specific bucket in a small table is $p_{small} = T^{-1}s^{-1}$, therefore, deg_{small} is binomially distributed with $Bin(Hm, p_{small})$, this in turn can be approximated with a Poisson distribution $Po(Hm \cdot p_{small}) = Po(H(j/T+1) \cdot B)$. The probability of hitting a specific bucket in a large table is $p_{large} = p_{small}/2$, thus the expected number of elements is also halved $deg_{large} = Po(Hm \cdot p_{large}) = Po(Hm \cdot p_{small}/2)$.

bucket degree

No bucket b can have more than $deg(b)$ elements. We say b has a *bucket defect* of $defect(b) = \min(0, B - deg(b))$, i.e., b has at least $defect(b)$ free slots. The bucket defect is a simple random variable and its expectation is easy to calculate. Therefore, $m - \sum_b defect(b)$ is an upper bound to the number of elements that can be inserted, before an insertion fails (from our original set of m elements). The influence of bucket defects can be seen in Figure 3.8. We see that this simple concept already explains a lot of the periodic behavior of the load bound, without actually assigning any elements to buckets.

bucket defect

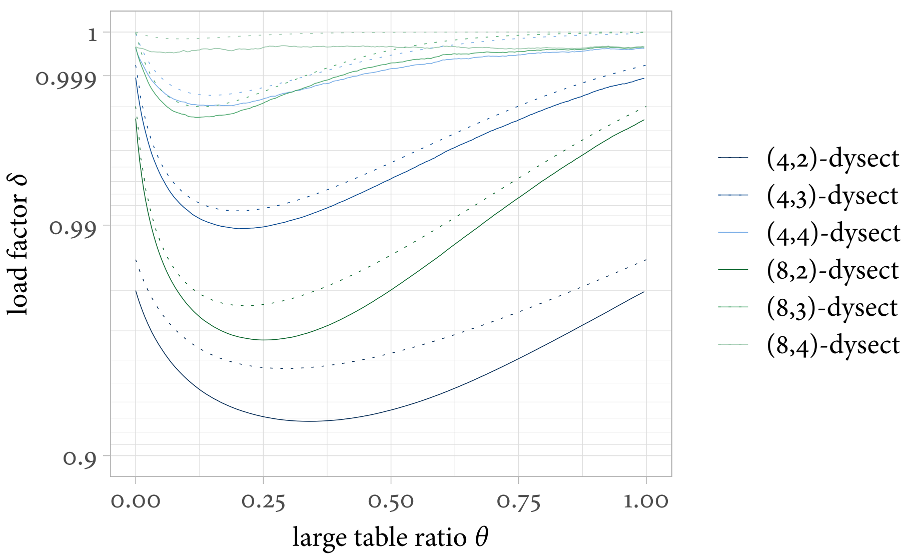

Figure 3.8: Comparing the expected maximum fill degree due to the number of bucket defects (dashed lines) to the measured load bound between 2^{23} and 2^{24} (solid lines).

3.6.3 Dynamic Load Threshold

It is clear that a newly grown subtable is not filled as densely as other subtables. In fact, it has a load below 50 % (subtable size is doubled). This load imbalance between subtables does not actually change the theoretical load bound. But, it does lead to long insertion times. Cuckoo hashing with a static table size profits from the fact that power of two choices hashing creates a very well balanced table, i.e., there are very few buckets with more than one free slot. Most free slots are distributed evenly throughout the table (making them easy to find). Growing subtable A creates a lot of buckets with very few elements, thus, insertions have to find a displacement path to one of these buckets (fewer buckets with empty slots).

subtable migrations cause imbalance

imbalance causes long insertions

To visualize the described problem assume the global table is filled close to 100 % before A is grown. Now all free slots in the global table are positioned in A. New elements that are not hashed to A automatically trigger the displacement algorithm. Each edge examined during the displacement algorithm has some probability p_A of hitting A, i.e., $\approx 1/T$. Therefore, we have to look at approximately T edges of the implicit graph. However, whenever we find a path, and move the appropriate element into the table there is one less element that helps future insertions (i.e., element outside of A that can be moved into A). This process benefits more from increasing the number of hash functions than it does from larger buckets, because the number of elements that is expected to be hashed to a table increases with H. Notice that repeated `insert` and `erase` operations help to equalize this imbalance because elements are more likely inserted into the sparser areas, and more likely to be deleted from denser areas.

3.7 Greedy Growing

The goal of our implementations is to achieve the best possible performance while keeping the size constraint. Thus, it is beneficial for our hash tables to be as close as possible to the memory bound. This is achieved through greedy growing, i.e., the table grows as soon as the number of elements increases enough to fit a larger table (and the migration is amortized by inserted elements). The resulting progress is displayed in Figure 3.9.

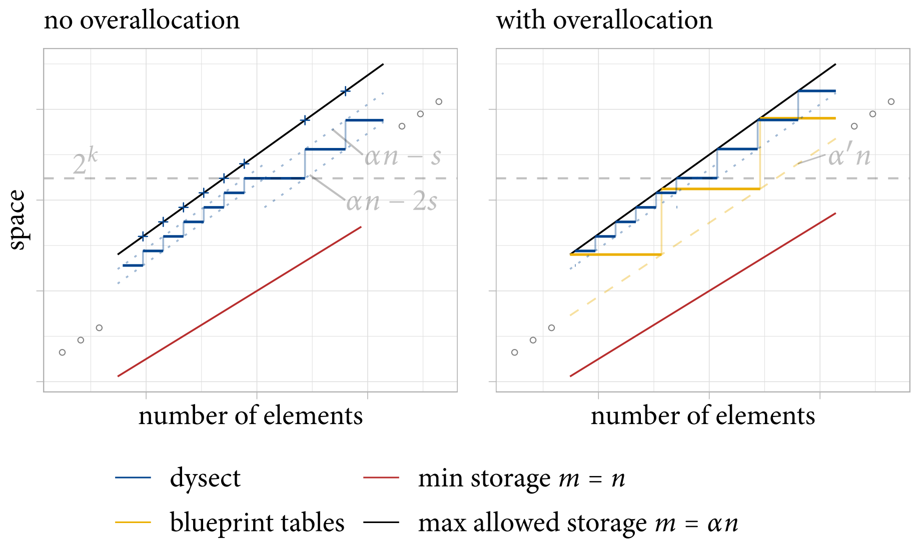

Figure 3.9: Memory usage of our different α-space efficient variants while the table is growing.

DySECT (no overallocation) The table grows when αn is larger than $m + 2s$ where s is the size of a small table. This is the moment where a new subtable can be allocated without breaching the size constraints (see markers on the αn line). The small original table is deallocated after the migration causing the overall memory to drop. The used memory fluctuates between $\alpha n - s$ and $\alpha n - 2s$ (right after and before a migration respectively). We see that the amount by which a migration changes the table size increases when the capacity reaches a power of two, i.e., when all tables have the same size (T and B are powers of two).

DySECT (with overallocation) When using overallocation and in-place table migration, capacity increases can be triggered earlier (at $\alpha n = m + s$), because the subtable is migrated in-place (i.e., its old memory is reused). This also means that we can fully use the allotted memory—as the used memory fluctuates between αn and $\alpha n - s$. The amount the table grows by changes when m reaches a power of two similar to the variant without overallocation.

Blueprint (with overallocation) All variants presented in Section have the same memory consumption. They grow when $m = \alpha' n$ where $\alpha' = (\alpha + 1)/2$. At this point a migration can be amortized (see Section 3.4.1). Since the memory grows in-place, there is no need for temporary memory.

Blueprint (no overallocation; not shown) This variant uses multiple tables to reduce the overhead of having to allocate a new table once a single table is growing. However, since the subtable sizes are independent of each other, i.e., each subtable grows depending on the number of elements inserted into it, imbalances in the table size can happen. Since the memory m_t of each subtable t ranges between $\alpha' n_t$ and αn_t the overall memory is also between $\alpha' n$ and αn. However this does not include the moment when a subtable is growing. During the process of growing, a subtables memory is $(\alpha + \alpha') \cdot n_t$. Therefore, the overall memory bound could be broken if a subtable grows while the overall memory is close to αn (this is unlikely, but it might happen if all subtables grow at approximately the same time).

Conclusion Overall we see, that both DySECT variants grow closer with the allotted memory of αn than the variants using our blueprint for dynamic α-space efficiency. Especially since adapting the number of subtables has little impact on the overall performance and increasing the number of subtables increases the number growing steps needed to double the table size (i.e., number of growing steps before s is increased).

3.8 Performance Experiments

We have shown through our construction that our tables can be α-space efficient. And that the frequent growing can be amortized. However, there are many factors that impact hash table performance. To show that our ideas actually perform well in practice, we use benchmarks with both synthetic and real world data sets.

All reported numbers are averaged by running each experiment five times. The experiments were executed on a one-socket AMD EPYC 7702P with 64 cores each running at 2.0 GHz (3.35 GHz Turbo Frequency), 256 MB L3 cache size and 1 TB of main memory[2]. It uses a Ubuntu 20.04.2 operating system and all tests were compiled using gcc 9.3.0 with `-march=native` and `-O3` flags. All experiments in this chapter of the dissertation are sequential (in contrast to the other chapters), therefore, the whole cache is available for processor running the benchmark. This could have some influence on the measured running times.

AMD Epyc 2GHz with 256 MB L3 Cache

We tested 5 different hash table architectures DySECT ◐ (see Section 3.5), cuckoo hashing ◧, linear probing ◆, quadratic probing ▲, and Robin Hood probing ▽ (see Section 3.4). For each of these architectures we have both an implementation that uses a multitable approach (without memory overcommitting; plots use dashed lines and markers are less opaque) and one that uses memory overcommitting (with in-place growing; plots use solid lines and higher opacity markers). Some workloads/tests are not feasible with the multitable variants created by our blueprint (both implementations of GrowT are always feasible). In these cases we show only feasible variants. Both DySECT and Cuckoo use the parameters described in Section 3.5.4, i.e., $B = 4$, $H = 4$, and $T = 256$. All multitable variants also use $T = 256$

5 different architectures each with in-place and multitable variants

[2] Experiments on different machines and architectures including a Desktop machine yield qualitatively similar results

3.8.1 Influence of Fill Ratio (Static Table Size)

query random elements

The following test was performed by initializing a table with a static table sizes (non-growing). Then we fill the table with random keys. At different fill degrees, we measure the running time of new insert (see Figure 3.10), and find (see Figure 3.11) operations. To measure the performance of an operation, we execute a sample of 1000 operations, averaging the running time. Positive finds are measured using randomly selected elements from within the table. Thus, representing an average query. Querying predominantly early insertions could lead to faster query times (especially on linear probing and quadratic probing), querying predominantly later elements could lead to slower query times. Negative queries query random elements from the whole key space.

no multitable variants

We omit testing the non-DySECT multi table variants. They are not suitable for this test because using a static table size (without growing) they cannot compensate potential imbalances between subtables. In a growing scenario multitables can be densely filled because subtables with more elements will just be grown more often. This is not possible in the static scenario. Each subtable would have to be constructed with m/T slots. Any kind of imbalance would likely cause one table to overflow. This is not the case for DySECT where subtable imbalances can be alleviated. In this test DySECT without memory overcommitting ○ or in-place migration actually performs better, than the version with memory overcommitting ●.

We repeated this test for four different table sizes ($\ell := 2^{24}, 1.25\ell, 1.5\ell, 1.75\ell$). This way, we can demonstrate the influence of the ratio of large subtables θ ($0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}$) on insertion times (the maximum load depends on θ see Section 3.6). Figure 3.10 (*left*) shows the average running time between the four measurements. The impact of θ on the performance of DySECT ○ insert operations can be seen in Figure 3.10 (*right*).

insert times normalized with $\frac{1}{1-\delta}$

As to be expected, the insertion performance depends highly on the load of the table. Therefore, we show it normalized with $\frac{1}{1-\delta}$ which is the expected number of fully random probes to find a free slot and thus a natural estimate for the running time of insertions. We see that—up to a certain point—the insertion time behaves proportional to $\frac{1}{1-\delta}$ for all tables. Close to the capacity limit of the table, the insertion time increases sharply. Depending on the growing phase θ, DySect ◐

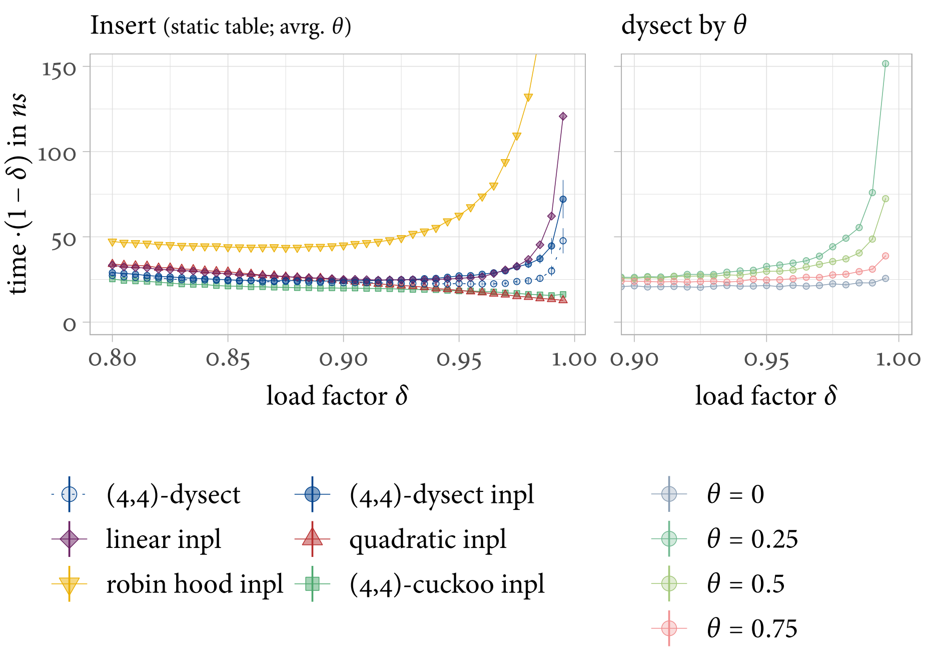

Figure 3.10: Insertions into a static table showing the influence from the load factor, on the performance of insertions. *Left*: average over four table sizes; *right* DySECT with different ratios of large subtables (θ). To make insertion time more readable, we normalize it with *time per operation* $\cdot (1 - \delta)$.

has smaller maximum load bounds than cuckoo 🟩 (see Section 3.6). This is also clearly visible in the performance (*right* side) when the table is very full. For many realistic loads (<97 %), however the impact remains small. Quadratic probing ▲ is exceptionally fast, even for the highest fill degrees.

Figure 3.11 shows the performance of find operations. Linear ◆ and quadratic ▲ both perform really well on successful find operations, up to a load of around 90 %. The reason for this is that most elements are inserted before the table becomes too full, therefore, many elements have small displacements—improving the average find performance. However, cuckoo hashing ▲ in particular and also DySECT ◐ perform much better on decently filled tables and on negative find queries, especially because their performance is completely independent of the load of the table. For all other tables the performance of negative find queries is much more related

cuckoo based approaches better on find

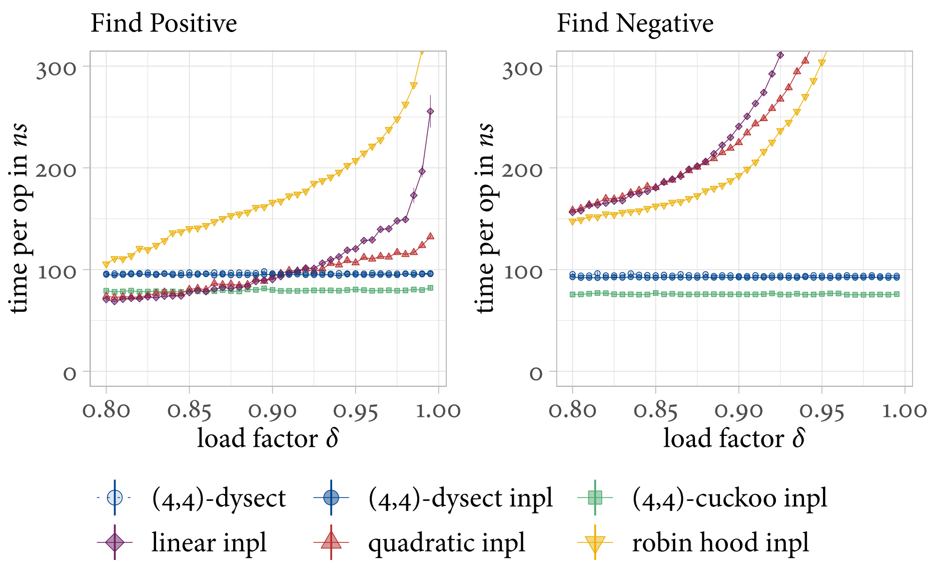

Figure 3.11: Performance of successful (*left*) and unsuccessful (*right*) finds. DySECT's find performance is independent from the load factor, and the operations success.

to the load of the table. There, traditional "probing-based" tables perform really bad. Robin Hood hashing performs somewhat better than linear probing. It worsens the successful find performance by moving previously inserted elements from their original position, in order to achieve better average displacements (which likely backfires since the probing performance is non-linear). However the negative find performance is somewhat improved compared to linear probing.

3.8.2 Influence of Fill Ratio (Dynamic Table Size)

minimum load factor $\delta_{\min}$
normalization with min load

In this test 10 M elements are inserted into an initially empty table. The table is initialized expecting 50 000 elements, thus growing is necessary to fit all elements. The tables are configured to guarantee a load factor of at least $\delta_{\min}$ at all times. Figure 3.12 shows the performance in relation to the minimum load factor. Insertion times are computed as average of all 10 M insertions. They are normalized similar to Figure 3.10 (divided by $\frac{1}{1-\delta_{\min}}$).

We see that both DySECT ◐ variants perform the better than the competitors on all fill degrees, even on lower load factors around 80 %. Here we achieve a 22 %

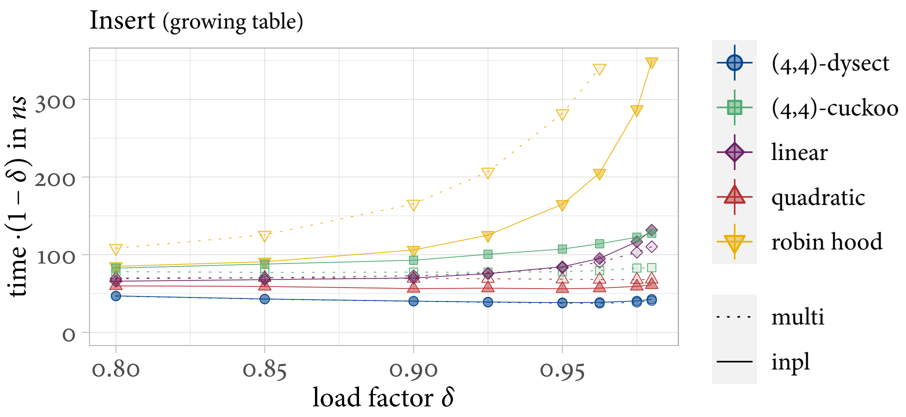

Figure 3.12: Insertions into a dynamic growing table enforcing a minimum load factor δ_{min}.

lower running time next best solution (233 % ns vs. quadratic probing ▲ 299 ns). On denser instances with 97.5 % load, we can increases the distance to 35 % (1545 % ns vs. quadratic probing ▲ 2368 ns). With growing load, we see the insertion times of linear probing ◆ and robin ▼ hood hashing degrade due to the combination of long insertion times, and frequent table migrations. However both cuckoo □ and quadratic ▲ remain close to DySECT ◐. This demonstrates the cache efficiency of the migration algorithms. DySECT ◐, quadratic probing ▲ and cuckoo □ all remain close to $O(\frac{1}{1-\delta_{\text{min}}})$ albeit at different constant factors due to the frequent migrations and high amortization overheads. DySECT ◐ has much smaller overheads because only very few elements are touched by each subtable migration ($\approx \frac{n}{T}$).

cuckoo and quadratic constant → amortization works

dysect dominates due to smaller overheads

We also measured the performance of find operations on the created tables, they are similar to the performance on the static table in Figure 3.11, therefore, we omit displaying them.

3.8.3 Word Count—a Practical use Case

Word count and other aggregation algorithms are some of the most common use cases for hash tables. Data is aggregated according to its key. This is a common application, in which static hash tables can never be space efficient, since the final size of the hash table is usually unknown. Here we use two different real world

gutenberg (literature)
common crawl (largely html)

data sets: first we use parts of the *Gutenberg* library (3 different parts, each with 2.2 GB, i.e., 1/10 overall size the experiment is repeated 3 times per block), second we use blocks of the *CommonCrawl* data set (`commoncrawl.org/the-data/get-started` from Apr. 2021, again we use 3 blocks at 5.4 GB and we repeat the experiment 3 times).

For each of these blocks, we count the occurences of each contained word. The gutenberg blocks have on average around 342 M words with around 7.6 M uniques, and the common crawl blocks have on average 229 M words with around 34.8 M uniques. Thus gutenberg blocks contain more words but have less unique words (on average 45 repetitions per word), which makes sense because they mostly contain natural language, and common crawl blocks contain fewer but more unique words (on average 7 repetitions per word).

For the word count benchmark, we hash each word to a 64 bit key and insert this key together with a counter. Subsequent accesses to the same key increase this counter. Note that hash collisions while computing the key are improbable, but they could lead to incorrect counts. However, this implementation is significantly faster and more space efficient than any variant storing strings within the table. Similar to the growing benchmark, we start with an empty table initialized for 50 000 elements.

The performance results can be seen in Figure 3.13. We do not use any normalization because most operations actually behave more like successful find operations instead of insertions. When using DySECT ◐ tables, the running time seems to be nearly independent from the table's enforced load. We experience little to no slowdown until around 92.5 %. Our probing based hash tables perform really well on the gutenberg data set the reason is likely, that the heavy hitting elements that appear many times ("the", "a" ...) are constantly cached, and are probably stored close to their canonical slot. The common crawl data set has less repetitions, there DySECT ◐ performs best among all tables above a fill degree of 95 %. On these high fill degrees, the performance degenerates similar to the insertion benchmark.

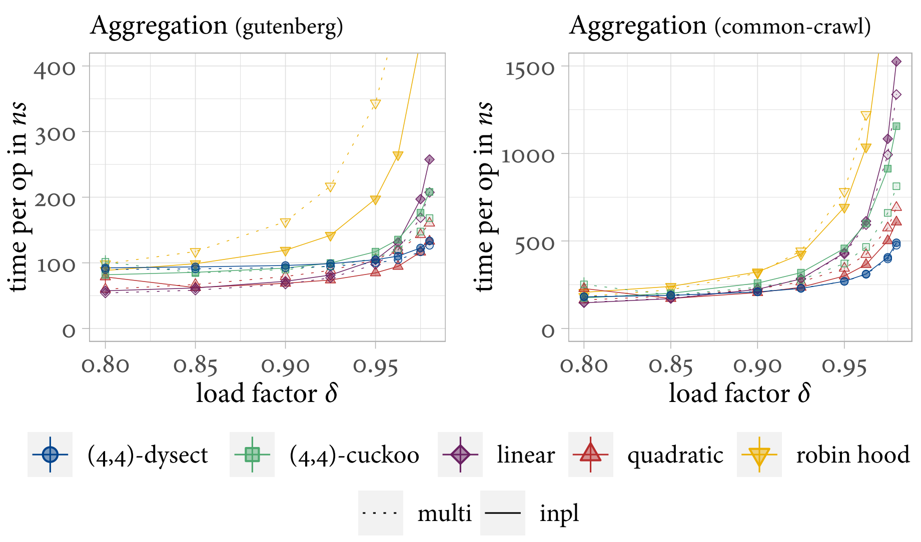

Figure 3.13: Word count benchmark. Behaves like a mix of insert and find operations. DySECT's performance only weakly depends of the load factor.

3.9 Conclusion

We have shown that dynamically growing hash tables can be implemented to always consume space close to the lower bound. We find it surprising that even our simple solutions based on common probing schemes seem to be new. They already offer a great performance considering the amount of migrations that happen. This also shows that our simple migration algorithm is highly efficient. It is interesting that quadratic probing performs best for insert heavy workloads even with very high fill degrees, yet is mentioned much less often than more popular techniques like cuckoo hashing.

DySECT is a sophisticated solution that exploits the flexibility offered by bucket cuckoo hashing to significantly decrease the number of element migrations over the structurally simpler in-place migration approach. When very high space efficiency is desired, it is up to 35 % faster than the simple solutions (quadratic probing which has significantly worse find performance) and 53 % faster than the structurally similar cuckoo hashing with multi tables.

4 Concurrent Hash Tables

In the previous chapter we have taken a look at memory scaling. However, the term scalability is more commonly associated with scaling to computational resources, i.e., scaling to multiprocessor environments. This has never been more important than today. Multi processor environments have become omnipresent and processor numbers are growing continuously. The same is true for industry and research applications that need to take advantage of these computational resources in order to make their problems solvable.

Hash tables are at the core of many algorithms, especially ones dealing with big data problems—organizing large quantities of data and making them searchable. Concurrent hash tables can also be used as an intuitive and asynchronous way to share data between threads. Thus communicating without the need for active synchronizations and contention.

This chapter is an update into our ongoing research in the field of concurrent hash table data structures. The findings of this research are integrated into our concurrent hash table library growt [43].

Concurrent hash tables are an area where specialized hash tables seem to be even more important than in the sequential case. In some preliminary experiments we found that a specialized table using linear probing and changing slot contents with simple compare-and-swap operations was significantly faster than many generalized implementations like TBB's `concurrent_unordered_map`. However, this implementation is limited to word sized key-value types and does not support deletions or dynamic size adaptation. Throughout this section, we explain how to lift each of these limitations in a provably scalable way and demonstrate that dynamic growing has a comparable overhead compared to dynamically growing a sequential table.

few interactions between threads

our goal is scalability

Concurrent hash tables are used in numerous applications. In many applications, hash tables are at the center of tight loops, as such they can be crucial for good running times. Moreover, well scaling implementations are necessary for these applications to work efficiently and to achieve speedups even in highly concurrent scenarios. Fortunately, hash tables offer some great opportunities for the design of concurrent data structures. Two simultaneous operations often have no influence on each other (e.g., when they are operating on different keys). Unfortunately, currently available concurrent hashing libraries do not offer the kind of scalability that would be necessary, especially when adaptively sized tables are necessary or contention on some elements occurs. Our approach of interchangeable implementations with different capabilities and performance characteristics is aimed at preserving the best possible performance for each task and offering scalable solutions for a wide range of possible applications.

We perform extensive experiments comparing the performance of our implementations with six of the most widely used concurrent hash tables. Each of our specializations is considerably faster than the best alternative algorithms with similar restrictions. Specialized implementations can be an order of magnitude faster than the best more general tables. In some extreme cases, the difference even approaches four orders of magnitude. All of our implementations as well as the code used for our benchmarks can be found on github [43]. The results presented in this chapter have in part been previously published in our conference [47] and journal [48] publications. These publications were written in cooperation with Peter Sanders and Roman Dementiev, with me being the main author. For this dissertation, I have rewritten and substantiated some of the design ideas concentrating on sections previously contributed by Sanders—specifically concerning arbitrary key and value types. Other sections of the previous publications are reused verbatim.

4.1 Motivation

To show the ubiquity of hash tables we give a short list of example applications: a very simple use case is storing sparse sets of precomputed solutions (e.g., [66, 7]). A more complicated scenario for hash tables is at the heart of many database like applications for example in the case of aggregations `SELECT FROM...COUNT...GROUP BY` x (e.g., [62]). Such a query selects rows from one or more relations and counts for every key x how many rows have been found (similar queries work with `SUM`, `MIN`, or `MAX`). Hashing can also be used for a database join [13]. Another group of examples is the exploration of a large combinatorial search space where a hash table is used to remember the already explored elements (e.g., in dynamic programming [84], itemset mining [72], a chess program, or when exploring an implicitly defined graph in model checking [85]). Similarly, a hash table can maintain a set of cached objects to save I/O-operations or repeated computations [64]. Further examples are duplicate removal, storing the edge set of a sparse graph in order to support edge queries [55], maintaining the set of non-empty slots in a grid-data structure used in geometry processing (e.g., [21]), or maintaining the children in tree data structures such as van Emde-Boas search trees [17] or suffix trees [53].

Outside of implementing specific parallel algorithms, concurrent hash tables can also be used to exchange information between different threads in an efficient and flexible manner, i.e., without contentious access to shared memory. Contrary to other concurrent data structures there is often little or no interaction between concurrent operations making scalability achievable. Therefore, concurrent hash tables are one of the most important concurrent data structures.

concurrent data structures allow asynchronous comunication between threads

Many of these applications and use cases have in common that—even in the sequential version of the program—hash table accesses constitute a significant fraction of the running time. Thus, it is essential to have highly scalable concurrent hash tables that actually deliver significant speedups in order to parallelize these applications. Unfortunately, currently available general purpose concurrent hash tables do not offer the needed scalability (see Section 4.6 for precise numbers). On the other hand, it seems to be folklore that a lock-free linear probing hash table can be constructed using atomic *compare-and-swap* (CAS) of key-value-pairs. We call our implementation of this simple concept folkloreHT [84]. But this simple ap-

hash tables are often in inner loops

folklore solution

restrictions:
- key- and value-size
- no capacity changes
- no deletions

proach has several restrictions which keep it from becoming a standard for parallel hash tables: keys-value pairs have to be small enough to enable the necessary CAS operation (i.e., 128 bit for double-word atomic CAS), the table cannot grow over its preallocated size, and it does not support true deletion (in a way that reclaims memory).

In this implementation, find operations can proceed naively and without any changes to the table. This is very important to us since there are scenarios where different hashing techniques can have a large impact on the concurrent performance. For example, consider a situation with mostly read only access to the table and heavy contention for a small number of elements that are accessed again and again by all threads. folkloreHT actually profits from this situation because the contended elements are likely to be replicated into local caches. On the other hand, any implementation that uses local locks or does any memory changes on the shared table during find operations, would become much slower than the sequential variant on current machines. The purpose of our research in this area is to document and explain performance differences, and, more importantly, to explore to what extent we can make folkloreHT more general with an acceptable deterioration in performance.

contention on a small number of elements

4.2 Related Work

updates on our continuing research

In this dissertation we follow up on our continuing research and findings about generalizing fast concurrent hash tables [46, 47, 48]. In addition to describing the previously presented generalizations we go into further detail on arbitrary key- and value-types—including experiments—and we show new measurements on current more relevant machines comparing to the most up to date libraries.

Often in textbooks—but:
- no focus on scalability
- surprising lack of detail

There has been extensive previous work on concurrent hashing. The widely used textbook "The Art of Multiprocessor Programming" [32] devotes an entire chapter to concurrent hashing and gives an overview over previous work. However, it seems to us that a lot of previous work focuses more on concepts and correctness but surprisingly little on scalability. For example, most of the discussed growing mechanisms in [32] assume that the number of elements in the table is known exactly without a discussion that this introduces a performance bottleneck limiting the speedup to a constant. In practice, the actual migration is often done sequentially.

Stivala et al. [84] describe a bounded concurrent linear probing hash table that is very similar to folkloreHT but is specialized for dynamic programming. It only supports insert and find operations. An interesting point is that they need only word size compare-and-swap instructions at the price of reserving a special empty value. This technique could also be adapted to port our code to machines without 128-bit-CAS. Kim and Kim [34] compare this table with a cache-optimized lock-free implementation of hashing with chaining and with hopscotch hashing [33]. The experiments use only uniformly distributed keys, which induce little contention. Both linear probing and hashing with chaining perform well in that case. The evaluation of find-performance is a bit inconclusive: chaining performs faster in their experiments, but uses more space than linear probing. Moreover it is not specified whether this is for successful (search for present keys) or mostly unsuccessful (search for non-present keys) queries. We suspect that varying these parameters could reverse the result.

Gao et al. [28] present a theoretical dynamically resizeable lock-free dynamic linear probing hash table. The main contribution is a formal correctness proof. Not all details of the algorithm let alone an implementation is given. Specifically, the act of moving the table is only reduced to the asynchronous write-all problem [30, 51]. Thus, there is also no analysis of the complexity of the growing procedure.

Shun and Blelloch [81] propose *phase concurrent hash tables* which are allowed to use only a single operation within a globally synchronized phase. They show how phase concurrency helps to implement some operations more efficiently and even deterministically in a linear probing context. For example, deletions can adapt the approach from Knuth [37] and rearrange elements. This is not possible in a general hash table since this might cause find operations to report false negatives. They also outline an elegant growing mechanism albeit without implementing it and without filling in all the details like how to initialize newly allocated tables. They propose to trigger a growing operation when any operation has to scan more than $k \log n$ elements where k is a tuning parameter. Shun and Blelloch make extensive experiments including applications from the problem based benchmark suite [82].

Li et al. [41] use the bucket cuckoo-hashing method by Dietzfelbinger and Weidling [19] and develop a concurrent implementation. They use fine grained per bucket locks which can sometimes be avoided using transactional memory, e.g., Intel® Transactional Synchronization Extensions TSX (Intel TSX). To further reduce the number of acquired locks per insertions they use a BFS-based insertion algorithm to find a minimal displacement path. As a result of their work, they implemented the open source library libcuckoo, which does not use Intel TSX (see Section 4.6 for an experimental evaluation). This approach has the potential to achieve very good space efficiency. However, our measurements indicate that the performance penalty is high especially when there is contention on a subset of keys.

implementations from outside the scientific comunity

The practical importance of concurrent hash tables also leads to new and innovative implementations outside of the scientific community. A good example of this is the Junction library [78] that was published by Preshing in the beginning of 2016, shortly after our initial publication [46]. Additionally, well tested and reliable concurrent hashing implementations can be found in multiple libraries provided by the biggest software companies like intel's TBB library [76] or in facebook's folly library [64].

4.3 Concurrent Hash Table Interface and Folklore Implementation

concurrent interface needs attention to detail

Although it seems quite clear what a hash table is (see Section 2.1.1) and how this generalizes to concurrent hash tables, there is a surprising number of details to consider. Therefore, we quickly go over some of our interface decisions and detail how this interface can be implemented in a simple, fast, lock-free concurrent linear probing hash table.

folkloreHT

In this section we describe a hash table that we call *folkloreHT*. Variations of folkloreHT are used in many publications and it is not clear to us by whom it was first published. It is the basis for all other hash table variants presented in this publication. folkloreHT has a bounded capacity m that has to be specified when the table is constructed and elements fit into two machine words (typically 128 bit).

The most important requirement for concurrent data structures is that they should be *linearizable* (see Section 2.2), i.e., it must be possible to order the operations in some sequence—without reordering two operations of the same thread—so that executing them sequentially in that order yields the same results as the concurrent processing. For a hash table data structure this basically means that all operations should be executed atomically at one point in time between their invocation and their return. For example, a `find` should never return an inconsistent state, e.g., a half-updated data field that was never actually stored at the corresponding key.

linearizability

Our variant of the folkloreHT ensures the atomicity of operations using *double-word atomic compare-and-swap* operations for all changes to the table. As long as the key and the value each only use one machine word, we can use double-word CAS operations to atomically manipulate a stored key together with the corresponding value. There are other variants that avoid needing double-word CAS operations, but they often need a designated empty value (see [78]). Since double-word-CAS instructions are widely available on modern hardware, using them should not be a problem. If the target architecture does not support these instructions, the implementation could in theory be switched to use a variant of concurrent linear probing without double-word CAS. As it can easily be deduced by the context, we will usually omit the prefix "double-word" and use the abbreviation CAS for both single and double word CAS operations.

double-word atomic CAS

To map keys to their canonical table slots we use a common preliminary hash function like xxHash [15] and a linear table mapping (see Section 2.3.2). We use table sizes which are powers of two to simplify the mapping (which consists of a single shift operation) and to improve the performance of the cyclic table via fast modulo operations (one bitmask).

table mapping

Initialization Given a number of expected elements n_{exp}, the constructor first defines m to be the smallest power of two that is still at least twice as large as n_{exp} (i.e., $2^{j-1} < 2n_{exp} \leq m = 2^j$). Then it allocates an array of size m consisting of 128-bit-aligned slots whose key is initialized to the *empty key* (here 0). This empty key cannot be used for stored key-value-pairs—see Section 4.4.6 for how to get around this limitation.

empty key

return by:
- value
- reference
- iterator

Accessing Elements In Section 2.1.1 we explain three different implementations of ways to access a hash table—return by value, return by reference, and return by iterator. In concurrent scenarios the method of giving access to the table is even more important. Both references and iterators to elements in a data structure can be invalidated by operations on said data structure. Usually this is the case when elements are moved within the data structure, e.g., iterators and references to elements in an unbounded array (e.g., `std::vector`) are invalidated when the array relocates. Accessing elements via invalidated iterators leads to unforeseen errors like segmentation faults. This is a problem for concurrent data structures because the invalidating operation could be triggered from another thread while one thread is currently accessing an element through an iterator—basically invalidating the iterator while it is being used. This is not necessarily a problem for folkloreHT because elements will not be moved within the table, thus iterators would not be invalidated, but we need to design an interface that is future proof for the different extensions we want to design and being able to move elements is essential for most extensions (e.g., deletions, growing, shrinking).

one thread may invalidate another thread's iterator

Another problem with a concurrent hash table interface is that all changes to elements in the table have to be atomic. Additionally, updates should also work even if elements are moved through the table. To ensure this we need a syntax that allows arbitrary changes without relying on the user to ensure atomicity. At the same time there should be an interface for experienced programmers to implement fast atomic updates that can take advantage of atomic operations like fetch and add, exchange, or compare-and-swap.

interface ensures atomicity

Modifications We categorize all operations that change the hash table's content into one of the three functions: *insert*, *update* and *insertOrUpdate* (excluding deletions). All of these functions are variations of the same general concept exemplified by *insertOrUpdate*.

In Algorithm 4.1 we show the pseudocode of the *insertOrUpdate* function. The operation computes the hash value of the key and proceeds to look for an element with the appropriate key (beginning at the corresponding position). If no element matching the key is found (when an empty space is encountered; case I), the new element has to be inserted. This is done using a compare-and-swap operation (case

Ia). A failed swap can only be caused by another insertion into the same slot (case Ib). In this case, we have to revisit the same slot to check if the inserted element matches the current key. If a slot storing the same key is found, it is updated using the *atomicUpdate* function (case II). This function is usually implemented by evaluating the passed update function (*up*) and using a compare-and-swap operation to change the slot. In the case of multiple concurrent updates, at least one update is successful.

failed CAS lead to retries

Algorithm 4.1 Concurrent *insertOrUpdate* operation.

```
input:
element $e = \langle k, d \rangle \in E = K \times V$,
update function $up: E \times P \to V$, additional update parameters $par \in P$
output:
true if $k$ was not present (insert), false if $k$ was present (update occurred)
$i = h(k)$
while true
    $current = table[i]$
    if $current.key == empty\text{-}key$ then
        // case I: k is not present yet...
        if $table[i].CAS(current, e)$ then
            return true // case Ia: successfully inserted
        else
            continue // case Ib: retry at the same position
    else if $current.key == k$ then
        // case II: key is already present...
        if $table[i].atomicUpdate(current, up(\cdot,\cdot), pars)$ then
            // e.g., $table[i].CAS(current,\ up(\langle k, current.data \rangle,\ pars))$
            return false// case IIa: updated present value
        else
            continue // case IIb: retry at same position
    $i$++ // if the table is circular $i = i \mod c$
```

insert(*e*): returns **true** if the insert succeeds, **false** otherwise (i.e, if an element with the specific key is already present). Only one operation should succeed if multiple threads are inserting the same key at the same time. The implementation is similar to Algorithm 4.1, but returns **false** if case II is reached (instead of any updates).

update$(k, up(\cdot,\cdot), pars)$: returns **false** if there is no value stored with the specified key, otherwise this function atomically updates the stored value to *new value* $= up(e, pars)$. Notice that the resulting value can be dependent on both the current value and the additional input parameters *pars*. The implementation is similar to Algorithm 4.1 but returns **true** in case IIa and **false** if case I is reached.

insertOrUpdate$(e, up(\cdot,\cdot), pars)$: this operation updates the current value if one is present, otherwise the given data element is inserted. The function returns **true** if *insertOrUpdate* performed an *insert* (key was not present) and **false** if an *update* was executed.

inserting thread can be identified

We choose this interface for two main reasons. It allows applications to quickly differentiate between inserting and changing an element. Additionally, even in contentious cases where multiple threads try to insert the same element at the same time there is exactly one thread that succeeds with its insertion. With the interface described above, arbitrary changes are automatically executed atomicallyAs described above, we made sure that using our interface arbitrary changes would be automatically executed in an atomic fashion (see default implementation of *atomicUpdate* in the comment). Experienced users can customize atomic changes by overloading the *atomicUpdate* function, e.g., with a simple overwrite (using single word store) or increment (using fetch and add).

trivial lookups

Lookup For the performance of many hash table workloads lookup operations are even more important than table modifications. FolkloreHT, provides lookups that proceed without any memory changes. *find*(k) returns an iterator like object that acts as an accessor to the element with key k. If there is no such element then the accessor is a dummy indicating that no such element exists.

To implement a correct find operation one has to be aware that using current hardware it is impossible to atomically read both the key and the value together[1] (as they have 128 bit bits). Therefore—even though any change to a slot is atomic—it is possible for a slot to be changed in between reading its key and its value, this is

[1]The element is not read atomically, because x86 processor specifications do not guarantee atomicity for 128-bit-reads.

called a *torn read*. To argue the correctness of our *find* implementation, we have to make sure that torn reads cannot lead to any inconsistent behavior. There are two kinds of interesting torn reads: first, an empty key is read while the queried key is inserted into the same slot. In this case the element is not found (consistent since it has not been fully inserted). Second, the element is updated between the key being read and the value being read. Since the value is read after the key, only the newer value is read. This is consistent with a finished update (updates cannot change the key). Modifications can also encounter these torn reads, but since all modifications use compare-and-swap instructions, the compare-and-swap happens atomically to the whole 128 bit slot.

torn reads

Deletions FolkloreHT does not support true deletions (deletions that reclaim previously used memory). Simply deleting elements from folkloreHT is not possible because we have to make sure that future lookup operations can still find displaced elements. The same problem arises when rearranging elements. The only way folkloreHT can support deletions is using *tombstones* (see 2.4.2).

deletions use tombstones

delete(k) returns **true** if an element with key k is successfully removed. The key stored in a slot is replaced with *del-key* (this key represents a tombstone). Future operations scan over deleted elements like over any other non-empty entry. This means that the slot cannot be used anymore. Using *tombstones* for handling deleted elements is usually not feasible because the starting capacity has to be set dependent on the number of overall insertions (deletion does not free up any deleted slots). Even worse, *tombstones* fill up the table and slow down find queries. In Section 4.4.4 we show how our generalizations can be used to handle *tombstones* more efficiently.

No inconsistencies can arise from this kind of deletion. In particular, a concurrent find operation with a torn read would return the element before the deletion since the delete-operation leaves the value-slot a untouched. A concurrent insert $\langle x, b\rangle$ might read the key x before it is overwritten by the deletion and return **false** because it concludes that an element with key x is already present. This is consistent with the outcome when the insertion is performed before the deletion in a linearization. Other operations would just continue scanning over the element.

Note that even new inserts cannot reuse a slot with a tombstone, because then torn reads could lead to errors. If a query reads its queried key, but before the

slots with tombstones cannot be reused value is read the element is first removed (keeping its value the same) and then overwritten by a new insertion. Now the first thread reads the updated value that was never stored together with the original key.

counting elements with fetch and add creates contention

Size Keeping track of the number of contained elements deserves special notice here because it turns out to be significantly harder in concurrent hash tables than in sequential hash tables, where it is trivial to count the number of contained elements—using a single counter. This same method is possible in parallel tables using atomic fetch and add operations, but it introduces a massive amount of contention on one single counter creating a performance bottleneck. Because of this we did not include a counting method in folkloreHT. In Section 4.4.2 we show how this can be alleviated using an approximate count.

4.4 Generalizations and Extensions

fast migration can solve many shortcommings of folkloreHT

In this section, we detail how to adapt the concurrent hash table implementation—described in the previous section—to be universally applicable to most hash table workloads. Most of our efforts have gone into a scalable migration method that is used to move all elements stored in one table into another table. It turns out that a fast migration can solve most shortcomings of folkloreHT (especially deletions and adaptable size).

4.4.1 Storing Thread-Local Data

thread-local data is a building block for other extensions

Storing thread specific data connected to a hash table is necessary to efficiently implement some of our other extensions. Per-thread data can be used in many different ways, from counting the number of insertions to caching shared resources.

From a theoretical point of view, it is easy to store thread specific data. The additional space is usually only dependent on the number of threads ($O(p)$ additional space), since the stored data is typically constant sized. Compared to the hash table data this is usually negligible ($p \ll n < m$).

Storing thread specific data is challenging from a software design and performance perspective. Our solution uses explicit handles. Each thread has to create a *handle*, prior to accessing the hash table. All hash table operations are then accessible via the handle, not the hash table object itself (which only supports a *getHandle* function). These handles can store thread specific data since they are not shared between threads. This is not only in line with the RAII idiom (resource acquisition is initialization [56]), but it also protects our implementation from some performance pitfalls like unnecessary indirections and false sharing (see Section 2.5.2). Moreover, the data can easily be deleted once the thread does not use the hash table anymore (delete the handle).

handles

There are alternatives to our solution using handles. Some of our competitors use a *register* function that each thread has to call once before accessing the table the first time. This function serves a similar purpose of creating some memory specific to this thread. Alternatively, one could use the C++ storage class specifier `thread_local` to create per-thread-memory. However, thread local storage duration behaves very similar to static storage duration, i.e., the number of thread local variables has to be known at compile time. And thus it is not possible to work on multiple different hash tables at once. Overall, both implementations have the problem of not being able to delete the data once the thread does not need to access the table anymore.

alternatives:
- register
- `thread_local`

4.4.2 Approximating the Size

Keeping an exact count of the elements stored in the hash table quickly leads to contention on one count variable. Additionally, when the table is used in parallel any size reading can be outdated before it is returned. Therefore, we propose to support only an approximate size operation that can be exact when it is coordinated between threads.

contention on size variable
exact size is outdated quickly

To keep an approximate count of all elements, each thread maintains a local counter of its successful insertions (using the method desribed in Section 4.4.1). The global counter is then frequently updated (e.g., every $\Theta(p)$ insertions) by atomically adding the local counter onto the global insertion counter I and then resetting the local count. Contention at I is provably small if the exact number of local insertions before updating the global counter is randomized, e.g., between 1 and $2p$. I underestimates the size by at most $O(()p^2)$. Since we assume the size

local insertion counter

to be much larger than p^2 this method provides a small relative error. By adding the maximal error, we also get an upper bound for the table size. Additionally, all threads can update the global counter to receive an exact count, e.g., after a phase of insertions is done when the size remains constant.

exact count

global/local deletion counter

If deletions are also allowed, we maintain a global counter D in a similar way. $S = I - D$ is then a good estimate of the total size as long as $S \gg p^2$. In this case, both global counts are updated every $\Theta(p)$ operations (independent of insert or delete), ensuring the same approximation quality as without deletions.

When a table is migrated for growing or shrinking (see Section 4.4.3), each migration thread locally counts the elements it moves. At the end of the migration, local counters are added to create the initial count for I. D is set to 0 because tombstones are removed during the migration.

4.4.3 Table Migration

our migration is not lock-free but fast and scalable

While Gao et al. [28] have shown that lock-free dynamic linear probing hash tables are possible, there is no result on their practical feasibility. Our focus is geared more towards engineering the fastest migration possible, therefore, we allow locking, as long as the practical performance is not impacted by those locks. To do this we want to avoid locks that have to be acquired during every operation. Instead, we design a way that introduces locking, only during table migrations. The maximum number of locks that a thread has to acquire (throughout the existance of the table) is in $O(\textit{number of migrations})$ (i.e., $O(\log n)$).

Moreover, our implementation works lock free, as long as no migration is triggered. Once a migration is triggered, it is not lock-free but happens asynchronously and is hidden from the threads using the hash table. This is helpful for instances where the necessity to grow is improbable, i.e., probabilistic bounds, but since growing is rare it is also improbable to ever be slowed down by a lock.

Eliminating Unnecessary Contention from the Migration

migration uses h_{int} and r_{map} to its advantage

Here we use the idea—introduced in Section 2.3 —that hashing a key to its canonical slot is a two step process. The overall hash function h is split into two parts: the intermediate hash function h_{int} which maps a key to a number in the range of

$1..2^{64}-1$ and the table mapping r_{map} which maps that number to a specific entry. When migrating elements into a new table, we ensure that the intermediate hash function remains the same. We also use the mapping technique to our advantage. As described in Section 4.3 we use a linear table mapping (implemented with a simple bitmask). There are two types of migrations: the table size stays at least the same ($m' \geq m$) or the table shrinks ($m' < m$).

Growing Exploiting the properties of linear probing and the linear mapping from hashed values to the table, there is a surprisingly simple way to migrate the elements from the old table to the new table in parallel which results in exactly the same order as the sequential algorithm and that greatly reduces the synchronization between threads.

To describe the technique, we will first introduce some definitions. Let m be the size of the old table, and m' be the size of the new table, then we call $\gamma = m'/m$ the *growing factor* of the migration. Notice that the specific tables mentioned here are still using sizes that are powers of two, therefore, the growing factor will also be a power of two. But all presented techniques also work with arbitrary table sizes and arbitrary growth factors ($\gamma \geq 1$). Additionally, we use the term *cluster* for a range of filled slots that is enclosed by empty slots (see Figure 4.1). In linear probing hash tables clusters can become large because each element whose canonical slot is within a cluster will be stored at the end of that cluster. Slots in the original table form a circular group $\mathbb{Z}/m\mathbb{Z}$ any slot calculations are computed in the cyclic group, slot calculations in the target table are calculated in $\mathbb{Z}/m'\mathbb{Z}$.

growth factor $\gamma = m'/m$

cluster

Lemma 1. *Consider a cluster $a..b$, i.e., slots $a..b$ are non-empty and slots $a-1$ and $b+1$ are empty. When migrating the table, sequential migration maps the elements stored in that cluster into the range $\lfloor \gamma a \rfloor..\lfloor \gamma(b+1) \rfloor$ in the target table, regardless of the rest of the source array.*

Proof. Let x be an element stored in the cluster $a..b$ at position $p(x) = h(x)+d(x)$, where $h(x) = \lfloor h_{pre}(x)\frac{m}{|H_{int}|} \rfloor$ is x's preliminary hash value $h_{pre}(x)$ linearly mapped to a slot using the mapping factor of $\frac{m}{|H_{int}|}$. We say $d(x)$ is the *displacement* of x.

displacement

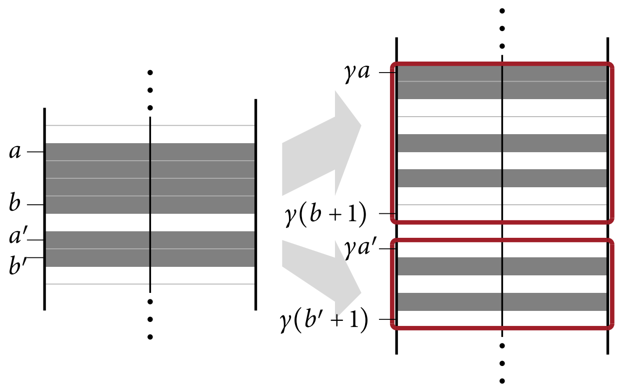

Figure 4.1: Schematic representation of two neighboring clusters and their non-overlapping target areas growing factor $\gamma = 2$.

Then $h(x)$ has to be in the cluster $a..b$ because linear probing does not displace elements over empty slots ($h(x) = \lfloor h_{pre}(x)\frac{m}{|H_{int}|}\rfloor \geq a$), and therefore, $h_{pre}(x)\frac{m'}{|H_{int}|} \geq a\frac{m'}{m} = \gamma a$.

Similarly, $\lfloor h_{pre}(x)\frac{m}{|H_{int}|}\rfloor \leq b$ implies that $h_{pre}(x)\frac{m}{|H_{int}|} < b + 1$, and therefore, $h_{pre}(x)\frac{m'}{|H_{int}|} < \gamma(b+1)$. □

Thus, two distinct clusters in the source table cannot overlap in the target table. We can exploit this lemma by assigning entire clusters to only one migrating thread which can then process its assigned cluster completely independently from any other threads. Meaning that as long as other threads work on other clusters, two threads never access the same slots of the target table.

two clusters are independent

Distributing clusters between threads can easily be achieved by first splitting the table into blocks (regardless of the tables contents) which we assign to threads for parallel migration. A thread that is assigned block $d..e$ migrates all clusters that start within this range—implicitly moving the "responsibility borders" to free slots between clusters as seen in Figure 4.2. The blocks are assigned by incrementing an atomic variable by the block size. Since the average cluster length is short and $m \in \Omega(p^2)$, it is sufficient to deal out blocks of size $\Omega(p)$ to reduce the contention on the atomic variable. Additionally, each thread is responsible for initializing all slots in its region of the target table. This is important because sequentially initializing the hash table can quickly become infeasible. It is possible that a cluster

implicitly moving borders to the end of a cluster

table is initialized at the same time

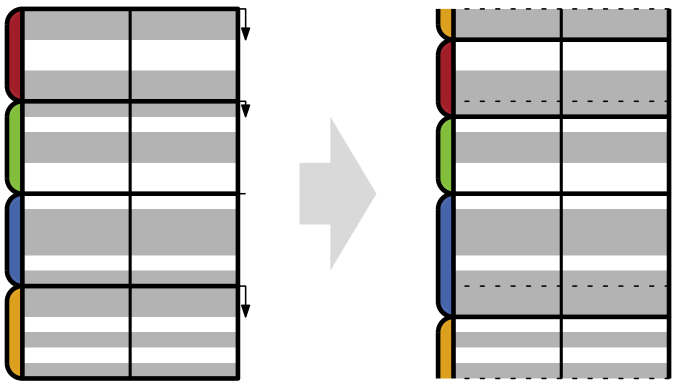

Figure 4.2: *Left*: dynamic block distribution—table split into even blocks. *Right*: resulting cluster distribution (implicit block borders).

covers a complete block. Using our simple scheduling method this would not be a problem, since this cluster fully belongs to the thread that works on the block it starts in. The covered block has no work to be done (clusters to migrate). The overall work balance is still ensured by our simple work balancing mechanism (given that: $\#(\mathit{clusters}) \gg p$).

Note that waiting for the last thread at the end of the migration introduces some waiting (locking). Usually, this does not create significant work imbalance, since the block/cluster migration is very fast and clusters are expected to be short. But if the operating system would interrupt a migrating thread, then all other threads would have to wait until this one thread is rescheduled and finishes its current block.

migration is not lock-free

Shrinking When elements are deleted, shrinking might be necessary to reuse unused memory (see Section 4.4.4). Unfortunately, the nice structural Lemma 1 no longer applies. Because when shrinking the table, separate clusters can collide in the target table. However, we can still parallelize the migration with little synchronization. Once more, we cut the source table into blocks that we assign to threads for migration. The scaling function maps each block $a..b$ in the source table to a block $a'..b'$ in the target table. We have to be careful with rounding issues so that the blocks in the target table are non-overlapping. We can then proceed in two phases. First, a migrating thread migrates those elements that move from $a..b$

separate clusters can collide

two phases:
- in block migration
- overflow

to $a'..b'$. These migrations can be done in a sequential manner (without atomic operations), since target blocks are disjoint. The majority of elements fits into the target block. Then, after a barrier synchronization, all elements that did not fit into their respective target blocks are migrated using concurrent insertion i.e., using atomic operations. This has negligible overhead since elements like this only exist at the boundaries of blocks. The resulting order of elements in the target table is not guaranteed to be the same as for a sequential migration but the data structure invariants of a linear probing hash table are still fulfilled.

Hiding the Migration from the Underlying Application

growing is triggered using the approximate count

To make the concurrent hash table more general and easy to use, we would like to avoid all explicit synchronization. The growing (and shrinking) operations should be performed asynchronously when needed, without involvement of the underlying application. The migration is triggered once the table is filled to a factor of α (e.g., 2/3), this is estimated using the approximate count from Section 4.4.2, and checked whenever the global count is updated. When a growing operation is triggered, the capacity is increased by a factor of $\gamma \geq 1$ (Usually $\gamma = 2$). The difficulty is ensuring that this operation is done in a transparent way without introducing any inconsistent behavior and without incurring undue overheads.

(1) who migrates elements
(2) preventing inconsistencies

To hide the migration process from the user, we have to solve two problems. First, we have to find threads to grow the table (strategies u and p), and second, we have to ensure that changing elements in the source table does not lead to any inconsistent states in the target table (possibly reverting changes made during the migration; strategies a and s). Each of these problems can be solved in multiple ways. We propose two strategies for each of them resulting in four different variants of the hash table (mix and match).

Recruiting User-Threads (u) A simple approach to dynamically find threads that migrate the table, is to recruit threads that try to perform table operations. These threads would otherwise have to wait for the completion of the growing process anyway. Recruiting user threads works well when the table is regularly accessed by all user-threads, thus, all threads cooperate on growing phases. But it can be inefficient in the worst case, e.g., when most threads stop accessing the table for example because they are stuck in a barrier waiting for the completion of a global computation phase. The few threads still accessing the table at this point would need a lot of time for growing (up to $\Omega(n)$) while most threads are waiting for them. One could try to recruit waiting threads but it looks difficult to do this in a sufficiently general and portable way. One way to allow users to add threads to an eventual table migration would be to call a dedicated *yield* function, this function would check, whether there is an ongoing migration and return otherwise. Similarly, one could even implement a specialized barrier, that inserts waiting threads into a thread pool of potential helpers.

Using a Dedicated Thread Pool (p) A provably efficient approach is to maintain a pool of p threads dedicated to growing the table. They are blocked until a growing operation is triggered. This is when they are awoken to collectively perform the migration in time $O(n/p)$ (assuming fair scheduling of migrating threads). Afterwards, they block again until the next migration is triggered. During a migration, application threads might have to sleep until the migration threads are finished. This will increase the CPU time of our migration threads making this method nearly as efficient as the recruiting variant. Using a reasonable computation model (i.e., processor speeds are within a constant of each other and all active threads pinned to one node get fair time slots), one can show that using thread pools for migration balances the cost of each table migration between all cores. We omit the relatively simple proof.

ensuring fair measurements

Note: in our implementation of this technique, we create one growing thread per application thread accessing the hash table. Additionally, we pin each growing thread to the same logical core as the corresponding application thread. Thus we ensure fair measurements because the same number of computational resources are used for all variants (recruiting, thread pool, and competitors).

Marking Moved Elements (a—asynchronous) During the migration it is important that no element can be changed in the old table after it has been copied to the new table. Otherwise, it would be hard to guarantee that changes are correctly applied to the new table. The easiest solution to this problem is to mark each

mark slots before copying

slot before it is copied. Marking each slot can be done using a compare-and-swap operation to set a special marked bit which is stored in the key. In practice this reduces the possible key space. If this reduction is a problem, see Section 4.4.6 on how to circumvent it. To ensure that no copied slot can be changed, it suffices to ensure that no marked slot can be changed. This can easily be done by checking the bit before each writing operation, and by using compare-and-swap operations for each update. However, this prohibits the use of fast atomic operations to change element values.

no updates to marked slots

Whenever a thread t finds a marked element, it is clear that the current table is being migrated. Then t either helps with the migration, or blocks until the thread pool has finished migrating the table (depending on the migration strategy u or p). In principle, find operations could proceed without waiting for the migration. We chose not to enable this in our implementation.

concurrent deallocation

After the migration, the old hash table has to be deallocated. Before deallocating an old table, we have to make sure that no thread is currently using it anymore. This problem can generally be solved by using reference counting. Instead of storing the table with a usual pointer, we use a reference counted pointer (e.g., `std::shared_ptr`) to ensure that the table is eventually freed.

locally buffering the incremented counter

The main disadvantage of counting pointers is that acquiring a counting pointer requires an atomic increment on a shared counter. Therefore, it is not feasible to acquire a counting pointer for each operation. Instead the counter is incremented once, and a reference to the table is stored locally (using the method from Section 4.4.1). At the beginning of each operation, we can use the local version number to make sure that the local table reference still points to the newest table version. If this is not the case, the old counter is decremented and a new pointer is acquired. This happens only once per version of the hash table. The old table is automatically freed by the last thread that updates its local pointer. This solutuion achieves the same worst case memory overhead $O(pm)$ as described by Gao et al. [28] because each thread keeps at most one table alive ($O(n)$ if the table grows with each migration). Note that classic counting pointers cannot be exchanged in a lock-free manner increasing the cost of changing the current table (using an explicit lock). We use a specialized counting implementation that reuses previous counters after their protected table was deleted. This implementation works lock-free with a small memory overhead of $O(p)$ counter objects.

Preventing Concurrent Updates (s—semi-synchronized) We propose a simple protocol inspired by read-copy-update protocols [54]. It uses p local flags $f_{1,..,p}$, and one global growing flag f_G to control that no thread is using the flag while it is being migrated. It should be said that to implement this technique efficiently special attention has to be paid to the memory order of different operations on flags. Whenever a thread t accesses the table, it sets its local busy flag f_t at the start of the operation and unsets it after the operation is completed (before returning the result of the operation). When a thread t_G triggers the growing operation it sets some global growing flag f_G using a compare-and-swap instruction. This global flag is inspected by each thread after setting its local flag (before executing each operation). If the flag is set, the local flag f_t is unset. Then the thread waits for the completion of the growing operation, or helps with migrating the table depending on the current growing strategy. After setting f_G, the growing thread t_G waits until all busy flags have been unset at least once before starting the migration (overhead $O(p)$ is a lower order term compared to the migration $\sim O(m/p)$). When the migration is completed, the growing flag is reset, signaling to the waiting threads that they can safely continue their table operations. Because this protocol ensures that no thread is accessing the previous table after the beginning of the migration, it can be freed without using reference counting.

wait until ongoing updates are finished

We call this method *semi-synchronized* because grow and update operations are disjoint. However, threads that are participating in one growing step (i.e., the u-variant) still arrive asynchronously, e.g. when the parent application called a hash table operation. Compared to the marking based protocol, we save cost during migration by avoiding compare-and-swap operations (i.e., not marking elements). However, this is at the expense of setting the busy flags for *every* operation. Our experiments indicates that overall this is only advantageous for updates using atomic operations like fetch-and-add that cannot coexist with the asynchronous consistency method's marking bit per element.

why semi-synchronized

Overview of the resulting methods In the beginning of this section we identified two orthogonal problems that have to be solved to migrate hash tables: which threads should execute the migration? and how can we make sure that copied elements cannot be changed in the old table? For each of these problems we formulated two strategies. The table can either be migrated by user-threads that execute operations on the table (*u*), or by using a pool of threads which is only responsible for the migration (*p*). To ensure that copied elements cannot be changed, we propose to mark elements before they are copied, thus proceeding fully asynchronously (*a*); and we explain a semi-synchronized protocol which ensures that all running update operations finish before the table is migrated (*s*).

u—user threads
p—thread pool
a—asynchronous(marking)
s—semi-synchronized

All strategies can be combined—creating the following four growing hash table variants: *uaGrow* recruit user threads and asynchronous marking for consistency; *usGrow* also uses user threads for the migration, but ensures consistency by synchronizing updates and growing routines; *paGrow* uses a pool of dedicated migration threads for the migration and asynchronous marking of migrated entries for consistency; and *psGrow* combines the use of a dedicated thread pool for migration with the semi-synchronized exclusion mechanism.

4.4.4 Deletions

In Section 4.3 we describe why the normal folkloreHT table cannot support true deletions. Instead we use *tombstones* to mark deleted elements, without reclaiming the slot for future insertions. This method can be improved by using our migration technique to clean the table once it is filled with too many *tombstones*.

As described in Section 4.3 there are two problems with this kind of deletion. The initial size of the table has to be set according to the number of overall insertions, and the generated *tombstones* clutter the table slowing down future operations. Both of these problems can be solved by migrating all non-tombstone elements into a new table. The decision when to migrate the table should be made solely based on the number of insertions I since the last table migration (i.e., number of non-empty slots). The count of all non-deleted elements $I - D$ is then used to decide whether the table should grow, keep the same size (notice $\gamma = 1$ is a special case for our optimized migration), or shrink. Either way, all tombstones can be

removing tombstones during the migration removed in the course of the element migration. We implemented both a growing migration, and a migration to a table with the same size. We did not implement the shrinking mechanism.

4.4.5 Bulk Operations and Forall

Forall

Many algorithms need to iterate over all elements within a hash table, e.g., outputting all unique elements after one round of duplicate detection. This problem can be solved easily by iterating over all slots of the table and omitting empty slots. This approach is very cache efficient if the table is sufficiently filled (constant ratio of filled slots). The use of dynamically sized hash tables—in particular—enforces a constant fill ratio because growing is only triggered once the table is sufficiently filled.

dynamically sized ⇒ constant fill degree

Forall operations can easily be done in a parallel manner—given the right interface. However, different interfaces have different strengths and weaknesses. Depending on the interface forall operations can be synchronized or asynchronous and the load balancing can be either dynamically or statically balanced. Additionally, it should be possible that only a subset of threads cooperates on the operation. To enable quick implementations supporting all of these different techniques, we decided on an interface that uses *block-iterators*. A block iterator can be constructed using two table indices start ℓ and end r. Once constructed, the iterator iterates over all filled slots between ℓ and r. Thus each element can be handled in parallel if different threads create different block iterators. Given this interface, all kinds of forall implementations can be realized. For example using static load balancing by splitting the table into p equal parts. Dynamic load balancing can be implemented by splitting the table into many smaller blocks using fetch-and-add on a single atomic variable to distribute the work packages (i.e., similar to the load balancing during the migration). The latter approach has the advantage that random element imbalances would likely be evened out.

synchronous or asynchronous

static or dynamic balancing

block-iterator

Synchronicity is achieved using dedicated barriers around the operation. The only restriction to this implementation is that the iterators should not be invalidated while the operation is going on. Thus, no element should be inserted and no element should be deleted during the operation (each thread is allowed to delete slots stored in its range). However, executing concurrent insert and delete operations likely creates problematic interactions with any kind of forall implementation. Similarly, it seems to be impractical to migrate a table while also executing a forall operation.

Bulk Operations

Our experiments in Section 4.6.4 show that contentious updates (highly skewed key sequences) still have a large impact on running times (even when using atomic operations). The main reason for this is probably the number of uncached main memory accesses caused by cache invalidation (see Section 2.5.2) and the necessary sequencing through atomic operations. Bulk operations might be a way to solve this problem in instances where all updates are known a priori. By sorting a number of hash table operations by their hashed key (or their canonical slot), one can partition the table into sections where updates (or even insertions) can run in an embarrassingly parallel fashion.

Construction Building a hash table for n elements passed to the constructor can be parallelized in this fashion. First, we use integer sorting to sort the inserted elements by their hash function value. This works in time $O(n/p)$ regardless of eventual collisions and repeated insertions, i.e., sorting circumvents contention. See the work of Müller et al. [62] for a discussion of this phenomenon in the context of aggregation.

Generalization Generalizing batches of operations to other methods, i.e., deletions, updates, or even mixed batches is straightforward. Processing batches of $m = \Omega(n)$ arbitrary hash table operations in a globally synchronized way can use the following strategy—similar to the strategy we just outlined for the case of bulk insertions. (1) First (integer-)sort all operations by their hash key in expected time $O(m/p)$. (2) Co-partition the sorted insertion array and the hash table into corresponding pieces of size $O(m/p)$. Most of the work can now be done on these pieces in an embarrassingly parallel way (load balancing is possible by choosing

co-partitioning the hash table and the sorted array of operations more than p pieces). Each piece is operated on sequentially by only one thread. (3) When working on each piece, we can minimize the sequence of operations for each key, i.e., any insertions and updates that happen before a deletion can be removed and depending on the update function updates can be grouped. (4) Then "merge" each block into the hash table (the hash table may have to be migrated beforehand to provide space for new elements). We can adapt ideas from parallel merging [31]. For insertions we can start looking for a free slot at position $\max(h(x), i)$ where i is the position of the last element this thread operated on. Atomics only have to be used for the first cluster of one partition, and once insertions enter the partition of another thread. This only happens for the last cluster of one threads partition (constant expected cluster length).

reducing the number of unnecessary operations

4.4.6 Restoring the Full Key Space

Our table uses two special keys, the empty key (*empty-key*) and the deleted key (*del-key*). Elements that actually have these keys cannot be stored in the hash table. To fix this, one could use two additional slots (m_e and m_d) in the global hash table data structure. If the element with the empty key is inserted it is stored in m_e (deleted key is stored in m_d). These two slots are not accessible through linear probing, therefore, they cannot be filled with any other elements. The case distinction needed to access these slots when an insert or lookup uses one of these special keys should have rather low impact on the overall performance.

empty-key and del-key

One of our growing variants (*asynchronous*) uses a marker bit in its key field. This halves the possible key space from 2^{64} to 2^{63}. To regain the lost key space, we can store the lost bit implicitly. Instead of using one hash table that holds all elements, we use the two subtables t_0 and t_1. The subtable t_0 holds all elements whose keys do not have their most significant bit set. While t_1 stores all elements whose keys do have the most significant bit set. Instead of storing the full keys, t_1 only stores the lower 63 bits of the keys. The most significant bit is removed (stored implicitly).

growing-marker-bit

Each element can still be found in constant time because when looking for a certain key, it is immediately obvious which table is responsible for the corresponding element. After choosing the right table, comparing the 63 explicitly stored bits uniquely identifies the correct element. Notice that both empty keys have to be stored distinctly (as described above). The size of both subtables can be chosen independently. In this case, both tables could be grown independently of one another.

4.4.7 Complex Key and Value Types

Using compare-and-swap instructions to change the content of hash table slots makes our data structure fast but limits its use to cases where keys and values fit into machine words. Lifting this restriction is bound to have some impact on performance. In the following, we outline ways to keep this penalty small. The general idea is to replace the key-value-pairs by pointers to the actual data, similar to hashing with chaining (see Section 2.4.1). So far, this is a very common technique and it runs the risk of combining the weaknesses of hashing with chaining (i.e., additional memory for pointers and indirections when accessing elements) and linear probing (i.e., long displacements and slowdown on full tables). But we feel that there are some possible improvements that are often overlooked.

- comparison
- concurrent deallocation
- memory allocation

Open Problems with Complex Elements There are three main problems that pointers introduce to the data structure: (1) each key comparison generally causes one main memory access, (2) memory allocation costs performance, and (3) deallocation in concurrent scenarios causes problems. In the following, we will show solutions to all of these problems that are adapted to our overall data structure to achieve the best possible performance.

Fast Comparison

When using pointers to store elements outside of the table, we have an effect that is similar to hashing with (out-of-table) chaining. Finding an element will always incur one "random" memory access per probed element (each one likely incurring a cache fault). Contrary to chaining, the accessed memory locations can be prefetched because the address of one memory access does not depend on the result of the previous access. But for our fast comparison, we want to minimize the memory accesses outside the table.

minimize memory accesses outside the table

To implement this fast comparison, it is important to know that pointers actually use fewer than 64 bits. On modern hardware, using current operating systems, pointers actually only use the least significant 48 bits[2]. The 16 most significant bits can be used for other data. For example, we can use these bits to mark migrated elements in the asynchronous variant of our migration algorithm (see Section 1).

To implement our fast comparison, we use the remaining 15 bits to store a *fingerprint* of the element. A fingerprint is a small number dependent on the key (here between 0 and $2^{15}-1$) that is independent of the canonical slot. We can easily compute such a number by hashing the key (using only 15 bits of the hashed value). We can even reuse the original hash value if we just use 15 bits that were not used for determining the canonical slot of the element. If we use the linear mapping with m being a power of two for computing the canonical slot, then the $64-\log m$ least significant bits were unused and thus should be independent from the canonical slot.

fingerprint

Thus, while querying for a key, we first compute its canonical slot and its fingerprint. Then we scan from its canonical slot similar to normal linear probing, but instead of comparing keys we would first compare fingerprints. Thus, we only compare the keys if the fingerprint stored in the slot matches the queried fingerprint. This method reduces the number of unnecessary key comparisons by a factor of 2^{-15} in expectation. This is especially interesting because comparing complex keys—like strings—usually takes more time than a simple integer comparison.

saves complex key-comparisons

Note: the same methods would still work with true 64 bit pointers. In this case, we would use the double-word variant of our table and store the fingerprint in the second word. This way, we could even support longer fingerprints, although it seems questionable that the speedup due to the smaller false positive rate would be worth the effort (using an additional hash function to compute fingerprints).

Frequent Allocation

To store elements outside of the table, it is necessary to allocate the *out-of-table memory* (see Section 2.5.3). Repeatedly calling commonly used memory allocation

out-of-table memory needs allocation

[2] Due to alignment reasons, there are even some additional unused bits in the least significant portion of a pointer, but we do not use those here

methods like `malloc` has a large impact on the performance of the data structure, especially in concurrent scenarios. There are specialized allocators that can alleviate some of these problems. However, we believe that parts of our design and our specific use case allow us to improve the performance even further.

The running time of these common allocation methods is usually independent of the element size, thus allocating a lot of memory at once is reasonably efficient. *local-page-allocator* We call the following technique a *local-page-allocator*. Using the per-thread-data technique described in Section 4.4.1, we keep one active memory page per-thread. Whenever a new element is inserted, we append its content to the active memory page and insert a pointer to this address. Once a page is completely filled, we can start a new page (keeping a pointer to the old one). Large elements (≈ page size) can be allocated using one of the common allocation functions. With the described allocation system, small allocations boil down to (non-atomically) incrementing a local variable. This is especially efficient in the absence of deletions, where it is impossible to create gaps in a previously filled memory page.

Concurrent data structures are often used to communicate data between different threads. Nonetheless, it is very common that threads predominately operate *reducing non-local memory access* on their own elements. This design has the advantage that it reduces the number of accesses to other NUMA-nodes in these cases (this works better in conjunction with our fingerprinting technique).

Concurrent Deallocation

Out-of-table storage is significantly more complicated if deletions are necessary. The critical situation that can arise is when one thread A reads a slot s, and the fingerprint matches its queried element. Then another thread B removes the element in slot s and replaces the pointer with a dummy. If B deallocates the element, then thread A will cause a segmentation fault when accessing the old element. This is a problem, since B has no way of knowing of A's intention to access the element. Luckily, there are different solutions to this problem that are specific to our design.

The concurrent deallocation problem is commonly solved with one of two methods, either by using complicated protection methods like *hazard pointer* or by pro- *simple locking method* tecting pointers with some kind of locking mechanism. Both options introduce some overhead. A simple locking mechanism that is easy to implement in our design is to introduce a locking bit in conjunction with the pointer. The locking bit could be implemented as part of the fingerprint—reducing the actual fingerprint size by one bit. After successfully comparing the fingerprint, A acquires the lock before accessing the element in slot s. Then the key can be compared without fear of deallocation. Similarly, thread B can only remove the element after locking its pointer first. Fingerprints can still be compared without locking the pointer, thus, a lock only impedes concurrent queries to other elements in the same cluster if they have a matching fingerprint. However, this method is sensitive to contentious workloads, even if there is only read contention (i.e., multiple threads query the same element, and try to obtain the same lock).

reclamation during table migration

One interesting possibility that is inherent to our hash table design is the fact that the table is repeatedly migrated—either when growing or when it runs full of tombstones. Connected to these migration phases, we have already solved the problem of concurrent deallocation for the old tables (see Section 4.4.3). Whenever an old table is deallocated, it is guaranteed that there is no operation still working with the old table. Thus there is also no operation still using one of the old pointers (that were only stored in this table). Thus, a simple reclamation technique would be to remove elements in conjunction with the table migration. When we are using the local-page-allocator described above, we can even defragment the memory pages by moving elements that remain in the table to free up unnecessary memory pages (this is only possible in the semi-synchronized variants).

reuse memory instead of deallocation

Another interesting idea is to reuse memory that was previously used by deleted elements. This works best when all inserted elements have the same size. This is usually the case in programming languages like C++, where all instances of the same (object-)type have the same size. Reusing the memory can either be done after the migration. Or we can reuse the memory during the lifetime of one table. For this it is necessary that queries which still access the previously stored element—e.g., for comparing their keys—recognize that the element has changed. This could be implemented efficiently by locally storing the addresses of recently removed elements until a new element is inserted. Freed addresses can be exchanged between threads, by adding them to a global data structure. Before doing so, pointers should be grouped (group size $O(p)$) to reduce the contention on the global structure.

This overall technique could be used to bound the number of actual allocations to $O(n_{\max} + p^2)$ where $n_{\max}$ is the maximum number of elements stored at the same time.

number of allocations $\in O(n_{\max} + p^2)$

4.5 Implementation Details

In this section, we describe some choices and details that we made while implementing our data structures.

All of our growing and non-growing hash tables (uaGrow, usGrow, paGrow, psGrow, and folkloreHT) can be instantiated with either specialized 128bit slots using double word compare-and-swap operations or with our methods for complex key and value types described in Section 4.4.7. Slots that can hold complex key and value types are implemented as atomic 64bit pointers, storing the actual element outside the table. Each pointer has a 15 or 16 bit fingerprint for fast comparisons. Changing a slot's content is implemented with an atomic compare exchange operation[3]. Changing an element's value can be done using any atomic operation on the value. Elements are allocated using Intel®Thread Building Blocks's [76] (TBB) scalable allocator.

specialized for 128bit or complex key value types

In our growing variants, a migration is triggered when the table is approximately 2/3 filled. Each migration either doubles the capacity m of the table or the capacity remains the same—depending on the number of non-tombstone elements ($\leq 1/3$). The migration works in blocks of 4096 slots that are assigned by incrementing an atomic number. Blocks are migrated with a minimum amount of atomics by using the cluster migration described in Section 4.4.3, thus moving the block boarders implicitly.

migration at 2/3

migration block size 4096

During our testing we found out that the memory mapping has a significant influence on the test's running time. Therefore, we use a user-space *memory-pool-allocator* implemented in TBB to prevent a slowdown due to the re-mapping of virtual to physical memory (see Section 2.5.3) which is probably protected by a coarse lock in the Linux kernel. This pool-allocator manages a large amount of memory that was already accessed and thus is already mapped from virtual to physical mem-

memory-pool-allocator to reduce the influence of memory mapping

[3]using the `cmpxchg16b` assembler instruction

ory. By allocating memory from this memory pool, we avoid concurrent syscalls to map the allocated memory, thus bypassing the kernel-lock. The performance impact from the allocation itself should be fairly minimal because there is only one allocation per migration level. The use of the memory-pool-allocator improves the performance especially when using more than 24 threads.

4.6 Experimental Evaluation

We performed a large number of experiments to investigate the performance of different concurrent hash tables in a variety of circumstances (an overview over all tested hash tables can be found in Table 4.1). We begin by describing the tested competitors (Section 4.6.1), the test environment (Section 4.6.2), and the test instances (Section 4.6.3). The results of our tests and a discussion thereof can be found in Section 4.6.4.

4.6.1 Competitors

To compare our implementation to the current state of the art we use a broad selection of other concurrent hash table implementations. These *competitors* were chosen on the basis of their popularity in applications and academic publications. We introduce symbols to simplify relating between the text and performance plots. Each implementation is assigned a unique symbol that is used throughout this section and within each graph (different implementations from the same library use the same symbol in different colors).

competitors from research and popular libraries

each hash table has a symbol

Non-Growing Hash Tables

One of the most important subjects of this chapter is to offer generalizations to our simple lock-free hash table implementation folkloreHT. While this makes folkloreHT much more usable in a variety of circumstances, for example when growing is necessary. We also want to show the overhead that is necessary for supporting all of these generalizations and we want to analyze if a dynamically sized table can compete with non-growing hash tables.

measure the overhead we ”pay” for growing

FolkloreHT + Our folkloreHT implementation described in Section 4.3. This hash table is also the core of our growing variants. Therefore, we can immediately determine the overhead that the ability for growing places on this implementation (Overhead for approximate counting and shared pointers).

ShunHash × This hash table implementation by Shun and Blelloch [81]. They offer two implementations, one deterministic variant that is supposed to be used in a phase concurrent manner, i.e. no reads can occur concurrently with writes and a non-deterministic variant which we used for our tests (in our preliminary experiment there was only very little difference between both versions)s.

Hopscotch Hash ⊠ Hopscotch hashing (see Section 2.4.2, page 36) is a variant of hashing with in-table displacement that was originally developed by Herlihy et al. [33] for this concurrent use case. The version we tested was published together with their original publication proposing the technique. Interestingly, the provided implementation only implements the functionality of a hash set (unable to retrieve/update stored data). Therefore, we had to adapt some tests to account for that[4] (`insert` ≅ `put` and `find` ≅ `contains`).

LeaHash ✳ This hash table is designed by Lea [38] as part of Java's Concurrency Package. We have obtained a C++ implementation which was published together with the hopscotch table. It was previously used for experiments by Herlihy et al. [33] and Shun and Blelloch [81]. LeaHash uses hashing with chaining and the implementation that we use has the same hash set interface as hopscotch[4].

Efficiently Growing Hash Tables

The following hash tables are able to grow efficiently from a very small initial size. They are used in our growing benchmarks, where we initialize tables with an initial capacity that is more than 3 orders of magnitude smaller than the number of insertions into the table, thus making efficient growing necessary for good performance. There is one hash table (folly) that can only grow by at most a "constant"

[4]The measured results should be taken with a grain of salt. Hash table implementations should be straightforward to implement, but they might lose some performance over the measured results

factor (approximately 20). To still be able to test folly, we decided to increase their intitial capacity to -th of the target size.

uaGrow ●, usGrow ●, paGrow ●, and psGrow ● These are our generalized implementations (see Section 4.5). To de-clutter the plots by a little bit, we usually only test uaGrow ● and usGrow ●. Additionally we present a specific comparison to the other two implementations in a separated comparison.

Junction Linear ◆, Junction Grampa ◆, and Junction Leapfrog ◆ The junction library consists of three different variants of a dynamic concurrent hash tables. It was published by Jeff Preshing on github [78] after our first publication on the subject ([46]). There are no scientific publications, but in their blog [77] Preshing publishes some insightful posts on their implementation. In theory, junction's hash tables use an approach to growing which is similar to ours. A filled bounded hash table is migrated into a newly allocated bigger table. Although all three variants are constructed from a similar idea their performance seems to differ quite significantly. The junction hash tables use a *quiescent-state based reclamation* (QSBR) protocol for memory reclamation. For the memory reclamation to work, i.e., each thread has to call a designated function in regular intervals, as long as it is not in a critical section. Calling this function indicates that the thread does not hold any references to the table. The table can be deallocated once its pointer has been removed and every thread has called the function at least once (transferring some of the overhead of deallocation to users of the data structure).

Junction tables need to be used with invertible hash functions. Since xxH3 (i.e., the hash function we used for all of our tests) is not invertible, we decided to use the provided hash function called *avalanche* (comes with the junction library). The different hash tables within the library all perform variants of in-table displacement (see blogpost [77] for for individual details).

tbbHM ▲ and tbbUM ▲ Intel®Threading Building Blocks [76] (TBB) library is one of the most widely used libraries for shared memory parallel (and concurrent) programming. It contains two different hash table implementations: (*tbbHM* ▲) concurrent hash map and (*tbbUM* ▲) concurrent unordered map respectively. Both versions have some significant differences especially concerning their interfaces and the way accessed elements are locked. Therefore, they behave very differently, for example under contention.

CUCKOO ■ (`cuckoohash_map`) This hash table using (bucket) cuckoo hashing as its collision resolution method, is part of the small libcuckoo library (Version 1.0). It uses a fine grained locking approach presented by Li et al. [41] to ensure consistency. Cuckoo is mentionable for their interesting interface, which combines easy container style access with an update routine similar to our update interface.

RCU ⊠/RCU QSBR ⊠ This hash table is part of the userspace-rcu library (Version 0.8.7) [75] that brings the read copy update principle to userspace applications. Read copy update is a set of protocols for concurrent programming that are popular in the Linux kernel community [54]. RCU uses the recommended read-copy-update variant (`urcu`). RCU QSBR uses a QSBR based protocol that is comparable to the one used by junction hash tables. It forces the user to repeatedly call a function with each participating thread.

FOLLY ▼ (`FOLLY::ATOMICHASHMAP`) This hash table was developed at facebook as a part of their open source library folly [65, 64] (Version 57:0). It uses restrictions on key and data types similar to folkloreHT. In contrast to our growing procedure, the folly table grows by allocating additional hash tables. This increases the cost of future queries. The implementation has a built-in maximal growing factor of ≈ 20 (over its initial size). Therefore, instead of initializing folly hash tables with the usual initial capacity we use an initial capacity that is $1/4th$ of the target capacity.

4.6.2 Hardware Overview

amd-epyc All experiments were executed on a one-socket machine with an AMD EPYC 7702P with 64 cores each running at 2.0 GHz (3.35 GHz Turbo Frequency) with 256 MB of L3 cache size and 1 TB of main memory. We call this machine *amd-epyc*. To show the scalability of different approches, we also execute experiments on a four-socket *4-socket-intel* Intel Xeon Gold 6138 machine with 20 cores per socket, each running at 2.0 GHz

Table 4.1: Overview of concurrent hash table implementations and their functionality.

name	plot	std. interface	growing	atomic updates	deletion	arbitrary types
FolkloreHT	+	✓		✓		
Shunhash	×	sync phases		partially[1]	✓	
Hopscotch	⊠	set interface		set interface	✓	
Lea Hash	✳	set interface		set interface	✓	
xyGrow						
uaGrow	●	using handles	✓	✓	✓	without deletion[2]
usGrow	●	using handles	✓	✓	✓	without deletion[2]
paGrow	●	using handles	✓	✓	✓	without deletion[2]
psGrow	●	using handles	✓	✓	✓	without deletion[2]
Junction						
linear	◆	qsbr function	✓	only overwrite	✓	
grampa	◆	qsbr function	✓	only overwrite	✓	
leapfrog	◆	qsbr function	✓	only overwrite	✓	
TBB						
hash map	▲	accessor obj.	✓	✓	✓	✓
unordered	▲	✓	✓	✓	unsafe	✓
Cuckoo	■	by value	slow	✓	✓	✓
RCU						
urcu	⊠	register thread	✓	✓	✓	✓
qsbr	⊠	qsbr function	✓	✓	✓	✓
Folly	▼	✓	const factor	✓		

[1] There are some specialized operations (chosen at the time of construction)
[2] not implemented in our implementation (possible for future release)

(3.7 GHz Turbo Frequency) with 27.5 MB of L3 cache size (per socket) and 768 GB of main memory (overall). We call this machine *4-socket-intel* (or short *4-intel*). Both machines use a Ubuntu 20.04.2 operating system and all tests were compiled using gcc 9.3.0 with `-march=native` and `-O3` flags.

4.6.3 Test Methodology

Each test measures the time it takes, to execute 10^8 hash table operations (*strong scaling*). Each data point was calculated by taking the average of five separate ex-

strong scaling 10^8 *operations*

ecution times[5]. Different tests use different hash table operations and key distributions. The used keys are pre-computed before the benchmark is started. Each speedup given in this section is computed as the *absolute speedup* over our hand-optimized sequential hash table.

reported speedups are absolute speedups

The work is distributed among threads dynamically. Until all operations have been distributed, threads reserve blocks of 4096 operations to execute (using an atomic counter). This ensures a minimal amount of work imbalance, making the measurements less prone to variance.

dynamic scheduling

All random inputs are precomputed before the execution. Two executions of the same test use the same input keys. Most experiments are performed with uniformly random generated keys (using the Mersenne twister random number generator [52]). Real world inputs usually have recurring elements, thus possibly introducing contention which can potentially lead to performance issues. To test hash table performance under contention, we use Zipf's distribution to create skewed key sequences. In a sequence of keys generated with Zipf's distribution, the probability for any given key k_i is $P(k_i) = 1/(i^s \cdot H_{N,s})$, where $H_{N,s}$ is the N-th generalized harmonic number $\sum_{i=1}^{N} \frac{1}{i^s}$ (normalization factor) and N is the universe size ($N = 10^8$). The skew can be adjusted using the exponent s. We use Zipf's distribution because it closely models some real world inputs like natural language, natural size distributions (e.g. of companies or internet pages), and even user behavior ([6, 9, 1]). Zipf distributed keys are pre-generated similar to our other random key distributions as to not influence the measurements unnecessarily (this is especially necessary since constructing Zipf distributed keys is more costly than uniformly distributed numbers).

uniform random with mersenne twister

skewed inputs with Zipf distribution

The main argument for using randomized keys over real world data is that even when using real world data, the access patterns will be randomized due to the hash function. The only influence that the input sequence has given a good hash function is the repetition of keys, which we simulate using our skewed input sequences. Only for our tests with arbitrary key types we use the real world *Project Gutenberg* data (each word is treated as a key) because generating realistic strings is much

why randomized data?

Project Gutenberg data set

[5]on 4-socket-intel we have experienced some deterministic slowdowns in the first and second iteration, thus, we increase the number of iterations and drop the first two measurements (warmup runs)

more difficult. The Gutenberg collection is a set of books within the public domain. It contains over 60 000 documents in many different languages, for the purpose of our tests however, we limit ourselves to the documents available in English.

xxH3 hash function

In our tests we use the hash function *xxH3 [15]* which is part of the xxHash library. It is a highly optimized hash function implementation that is efficient for both long and short inputs. Additionally, while it is not a cryptographic hash function, it still offers enough entropy to "guarantee" well balanced hash tables.

We use sequential variants of our growing and fixed size tables to measure and report absolute speedups. They do not use any atomic instructions or other overheads necessary for our concurrent data structures. In a number of preliminary experiments our sequential variants outperform popular choices like Google's dense hash map (80 % increased insert throughput), making them a reasonable approximation for the optimal sequential performance.

4.6.4 Experiments

The most basic functionality of each hash table is inserting and finding elements. The performance of many parallel algorithms depends on the scalability of parallel insertions and finds. Therefore, we begin our experiments with a thorough investigation into the scalability of these basic hash table operations.

Insert Performance

10^8 insertions
(1) static table size
(2) growing table

We begin by benchmarking the insert performance 10^8 different uniformly random keys are inserted into a previously empty hash table. We repeated this test two times once inserting into a preinitialized table that has the appropriate capacity to hold all inserted elements and once inserting into a growing table (i.e., initialized with a capacity of ≈ 50 000 causing the table to grow by about 3 orders of magnitude—folly grows by a factor of 4). Figure 4.3 shows results measured on both machines (amd-epyc and the 4-socket-intel server). Due to the number of tested hash tables, these plots can be pretty overwhelming but we want to single out some trends.

top tables use linear probing

On *amd-epyc* we see that insertions into the static table (Figure 4.3; *top*) scale really well for almost all hash table architectures. The top four best absolute performing hash tables are all variants of folkloreHT (i.e., linear probing with compare-

and-swap): shunhash × (38.2; absolute speedup taken at $p = 64$), folkloreHT + (38.1), uaGrow ● (34.1), and usGrow ● (27.4). All other tables have at most half the performance, i.e., cuckoo ■ (14.8) and folly ▼ (12.5). Our growable variants uaGrow ● and usGrow ● are 11 % and

28 % slower than folklore. This is the overhead caused by the ability to grow the table, i.e., due to the overhead of keeping pointers to the table up to date.

bottleneck on 4-socket-intel

On 4-socket-intel (Figure 4.3; *third* plot) we could not reproduce the nice scaling behavior of amd-epyc. Our solutions scale relatively well up to 20 cores; then they scale reasonable up to 40 cores (second socket); afterwards there is a lot of variance indicating that there is some bottleneck impacting the performance and causing congestion (accesses scale better—indicating that the bottleneck might be write access to NUMA-memory). However the relative behavior of the hash tables remains similar. The top four hash tables have not changed their order (cuckoo and folly switched places).

absolute speedup with growing vs. absolute speedup on static table

In our experiments with growing tables (see Figure 4.3) we can see that again our variants based on linear probing are the most efficient. Especially on *amd-epyc* (*second* plot) there uaGrow ● has an absolute speedup of 25.9 (at $p = 64$) and usGrow ● has a speedup of 25.6 (24 % and % less than in the static case). Thus, usGrow has comparable absolute speedups for the case with a growing table compared to the case with a static table, indicating that our migration approach scales about as well as insert operations. Similar to the static case, no other table comes within a factor of two in this test. The best competitors are junction grampa ◆ with a speedup of 9.8 and the two rcu tables ⊠, ⊠ both with around 6.8. On *4-socket-intel* uaGrow ●, usGrow ●, and junction grampa ◆ perform relatively similar, all seemingly converge to the same maximum speed. Comparing the absolute speedups of our solutions between the static table size and the growing table we achieve 9 % (and 17 %) lower speedups than in the static case, indicating again that the migration scales about as well as inserting elements.

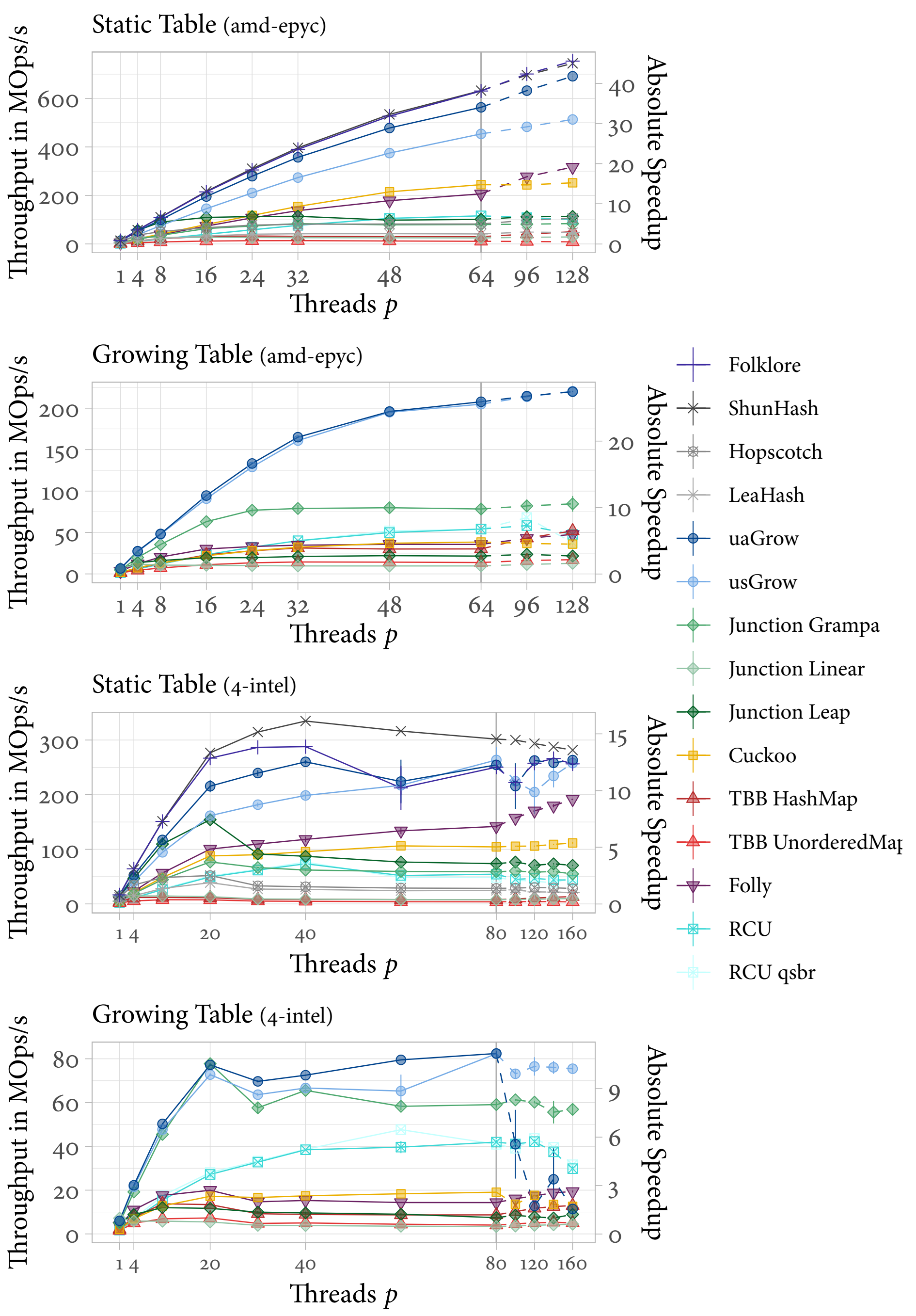

Figure 4.3: Throughput while inserting 10^8 elements into a previously empty table. Performance is measured either with static table size (plots *1* and *3*) or with dynamically growing tables $m_{start} \approx 50\,000$ (plots *2* and *4*). The results are collected both on amd-epyc (plots *1* and *2*) and on 4-socket-intel (plots *3* and *4*).

Find Performance

amd-epyc vs. 4-intel × growing vs. static × positive vs. negative

When looking for a key in a hash table there are two possible outcomes—either it is in the table or it is not. For most hash tables not finding an element takes longer than finding said element. Therefore, we present two distinct measurements for both cases Figure 4.4. We have also experienced the fact that queries on a grown table can also take more or less time to execute. Therefore, we also show measurements querying tables that have previously grown (these benchmarks were executed on the tables constructed during the previous benchmark). We measure the performance of positive queries by looking up all 10^8 elements that were previously inserted into the hash table (average find time). For measuring negative queries 10^8 uniformly random keys are searched in the same hash table.

Since find operations can proceed without changing memory it is fairly obvious that many hash tables can achieve higher throughputs on find heavy workloads, e.g., folkloreHT + has 1.42 times higher throughput on positive find queries vs. insert operations ($p = 64$; 1.14 times higher for negative finds). The hash tables that perform well in this benchmark are again the same four linear probing based hash tables that performed well on the insertion benchmark (shunhash ×, folkloreHT +, uaGrow ●, and usGrow ●). In addition to these hopscotch ⊠ and folly ▼ achieve nearly the same speedup. Hopscotch hashing ⊠ is especially strong for negative find queries where it is the fastest hash table.

queries scale better than insertions

The absolute speedups measured in this benchmark are also significantly higher than the speedups reached during the insertion test suggesting that queries scale better than insertions (folkloreHT + 1.18 × and 1.25 ×, and uaGrow ● 1.13 × and 1.42 ×). Hyperthreading also seems to be quite efficient for find queries.

on 4-intel queries are faster on migrated tables

The results on *4-socket-intel* again suggest that there is a problem with memory accesses to different NUMA-nodes. It is somewhat indicative how folkloreHT + and shunhash × start to collapse when scaling to a second node. Bot uaGrow ● and usGrow ● scale somewhat better on successful find queries but have the same problem on find negative queries. What is interesting however, is that these problems are significantly reduced on the tables that have been migrated (*row 3 and 4*). The reason could be that these tables were initialized concurrently by all processors who executed the migration thus the tables memory would be scattered among

NUMA-nodes in contrast to the static tables that were initialized on one specific node (uaGrow ● 1.44 × and 1.78 × and usGrow ● 1.31 × and 1.92 × higher throughput on migrated tables). Thus on multi-socket hardware it seems to be beneficial to intentionally allocate the table too small, thus forcing the table to grow. Alternatively one could probably implement a parallel table initialization that would split the tables memory among NUMA-nodes. However, this would necessitate a larger change to the tables interface and also some synchronization overhead during the construction of the table.

Overall, we have to say that none of the competitor tables performs well after it has been growing. Most tables that were performing well in these find benchmarks were non-growing tables (i.e., hopscotch ⊠, shunhash ×, and folkloreHT +) other tables have bad performance specifically on grown tables (i.e., folly ▼).

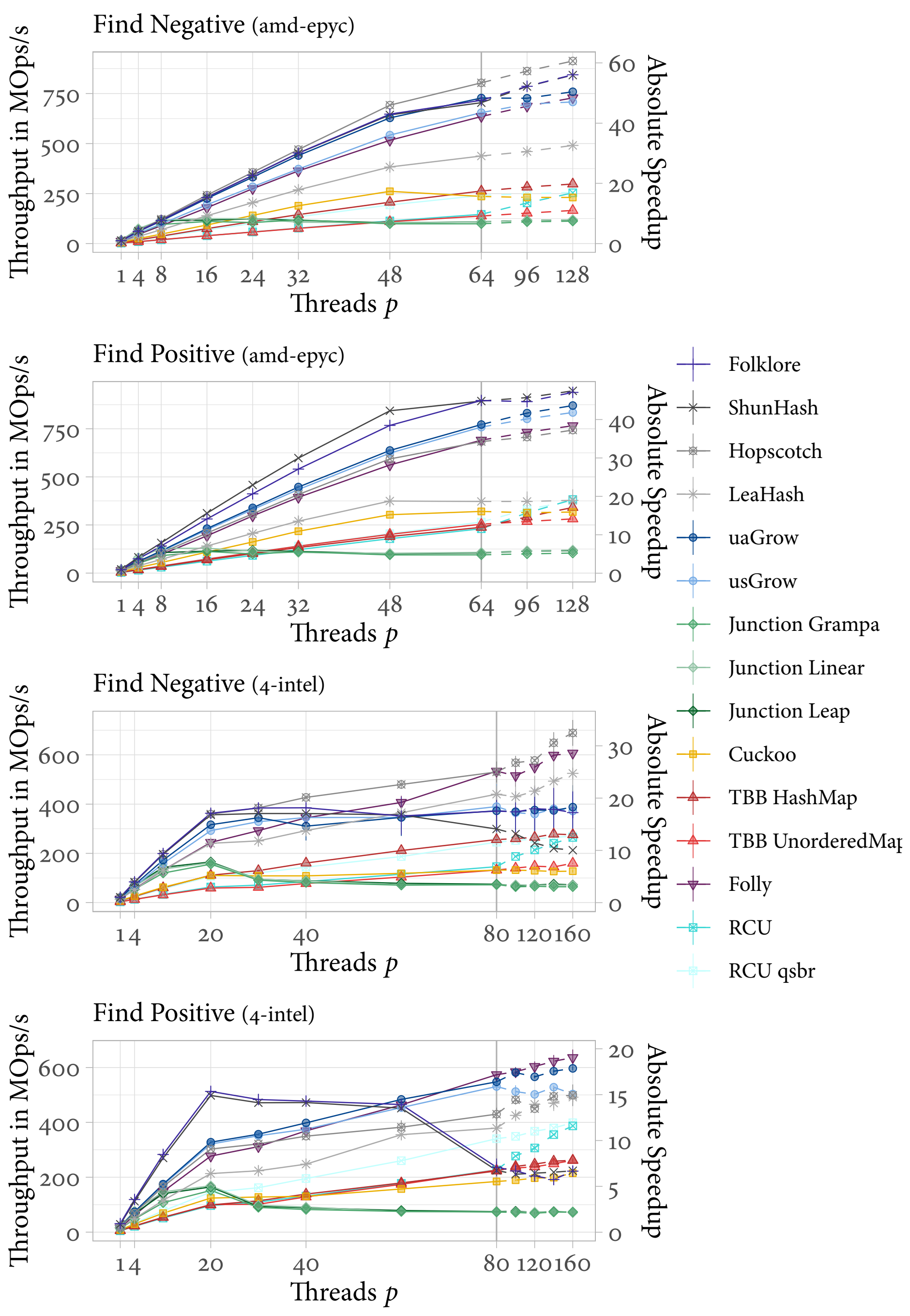

Figure 4.4: Performance and scalability of find operations. We call 10^8 find operations on a table containing 10^8 unique keys. We measure the throuput of successful find operations—keys that are present (plots *1* and *3*) and of unsuccessful find operations—random uninserted keys (plots *2* and *4*). The results are collected both on amd-epyc (plots *1* and *2*) and on 4-socket-intel (plots *3* and *4*).

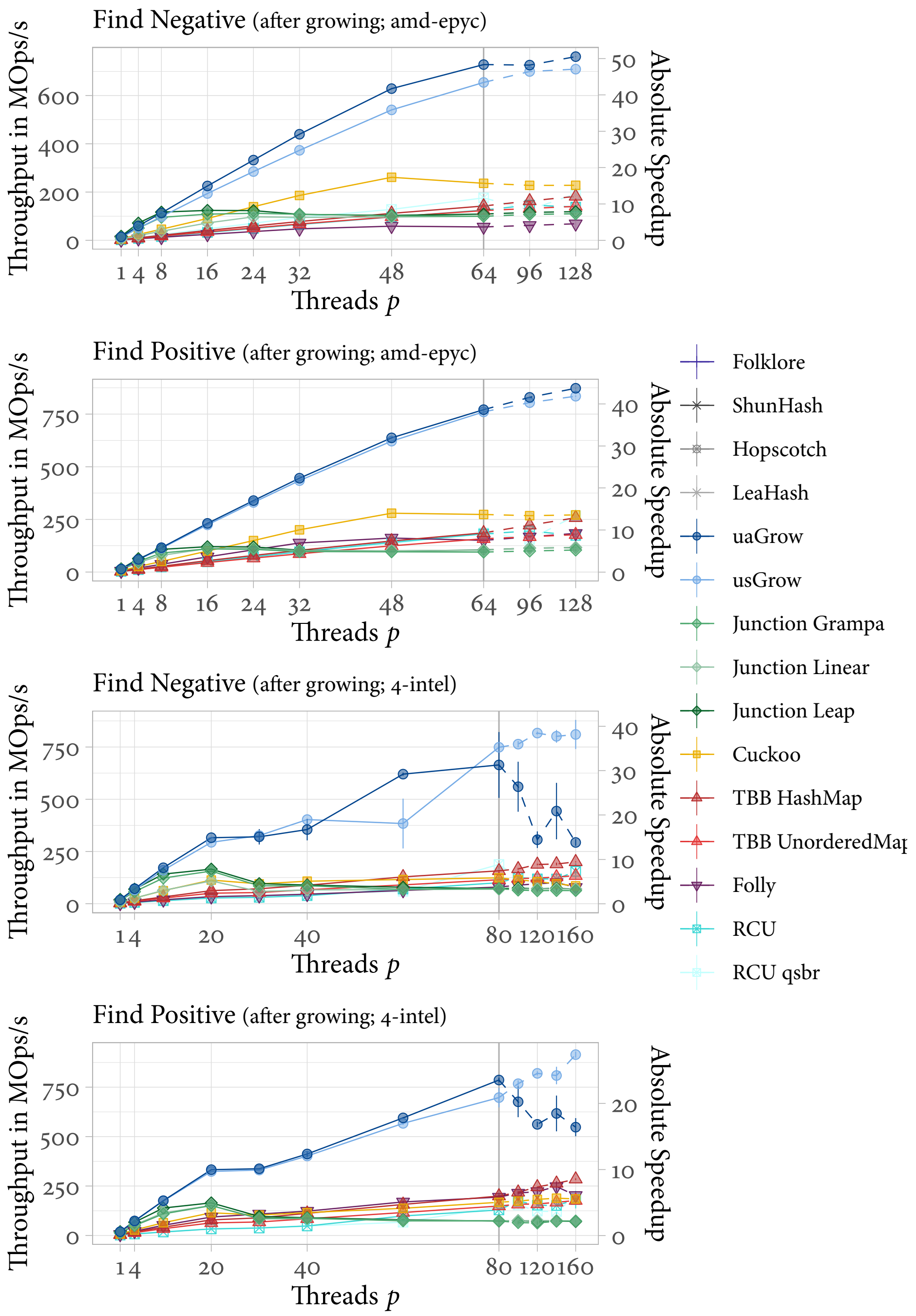

Figure 4.5: Experiment similar to Figure 4.4. However, all experiments (shown here) are concluded on a table that has grown to its current size ($m_{start} \approx 50\,000$). We measure the throuput of successful find operations—keys that are present (plots *1* and *3*) and of unsuccessful find operations—random uninserted keys (plots *2* and *4*). The results are collected both on amd-epyc (plots *1* and *2*) and on 4-socket-intel (plots *3* and *4*).

Performance under Contention

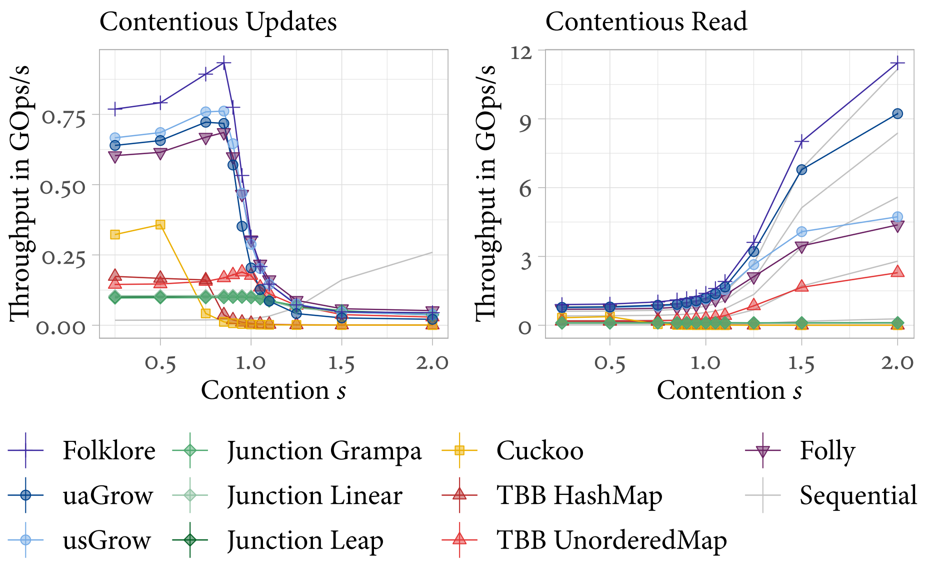

Figure 4.6: Throughput of updates and find operations with a skewed input distribution. Executing 10^8 operations with $p = 64$ tested on *amd-epyc*. *Left*: assign operations (write access); *right*: contentious look ups (no write access; we also plot 10×, ..., 40× the sequential Performance).

all updates/queries are successful

Up to this point all data sets we looked at contained uniformly random keys sampled from the whole key space. This is not necessarily the case in real world data sets where some keys might appear many times. One key might even dominate the input. Access to this key's element can slow down the global progress significantly especially if hash table operations use locking (even fine grained locking) to protect hash table accesses. To benchmark the robustness of the compared hash tables on these degenerate inputs we construct the following test setup. First, we fill the table with all keys from 1 to 10^8, thus all updates and find queries will be successful. Then we compute a sequence of skewed keys using the Zipf distribution described in Section 4.6.3. We generate 10^8 keys from the range $1..10^8$ with a varying exponent s (amount of skew). We reduced the set of tested hash tables for this benchmark: hopscotch ⊠ and leahash ✳ are only hash sets, thus updates are not possible; shunhash × and rcu ⊠, ⊠ can support updates but the interfaces are somewhat unwieldy (additionally shun is very similar to folkloreHT creating unnecessary clutter).

For the first benchmark we execute an update operation for each key of the skewed key sequence overwriting its previously stored element (Figure 4.6 *left*). These update operations create contentious write accesses to the hash table. Note that updates perform simple overwrites, i.e., the resulting value of the element is not dependent on the previous value. The hash table remains at a constant size throughout the execution, making it easy to compare different implementations independent of effects introduced through growing. In the second benchmark we execute find operations instead of updates, thus creating contentious read accesses.

sequential tables profit from contention

For sequential hash tables contention on some elements can have very positive effects. When one slot is visited repeatedly, its contents will be cached and thus future accesses will be faster. The sequential performance shown in our plot rises (grey line). For concurrent hash tables, however, contention has a very different effects. For concurrent hash tables there is a significant difference between contentious write or find operations. The reason is that multiple threads can read the same value simultaneously but only one thread at a time can change a value (on current CPU architecture). Therefore, read accesses can profit from cache effects—much like a sequential hash table—while write accesses are hindered by the contention. This goes so far that for workloads with high contention, no concurrent hash table can achieve the performance of a sequential table.

From the update measurement shown in Figure 4.6 *left* it is clearly visible that the serious impact through contention begins between $s = 0.85$ and 0.95. Up until that point contention has a positive effect even on update operations. For a skew between $s = 0.85$ and 0.95 about 1 % to 3 % of all accesses go to the most common element (key k_1). This is exactly the point where $1/p \approx P(k_1)$, therefore, on average there will always be one thread changing the value of k_1. Contentious update operations are one of the motivations for the semi synchronized scheme employed in usGrow ● . In usGrow ● elements can be updated using a simple 64 bit compare-and-swap operation instead of a 128 bit compare-and-swap operation on the whole slot as it would be necessary for uaGrow ● (to ensure that the marked bit of the element has not been set before the change). This is possible because updates and grow routines cannot overlap in usGrow ● this variant.

upto $s = 0.85$ mostly positive $\rightarrow \approx 1\,\%$ of keys are equal

usGrow advantage on updates

Figure 4.6 (*right*) shows that concurrent hash tables achieve similar performance benefits from contentious read accesses as sequential ones. FolkloreHT + and ua-

Grow ● consistently have absolute speedups greater than 40 (see topmost grey line, i.e., sequential throughput times 40) which is consistent with the negative find performance on non-contentious queries.

Overall, we see that our folkloreHT + implementation consistently outperforms all other competitors. Our two growing variants uaGrow ● and usGrow ● both have their own advantages, i.e., less overhead on find operations and cheaper atomics for updates. However both versions still outperform most competitors at least on lookups. None of the tables can achieve any speedups on highly contentious write accesses these are probably instances that would profit from a special treatment of their most common keys (Note: $s = 0.95 \rightarrow P[k_1] \geq 3\%$).

Aggregation—a common Use Case

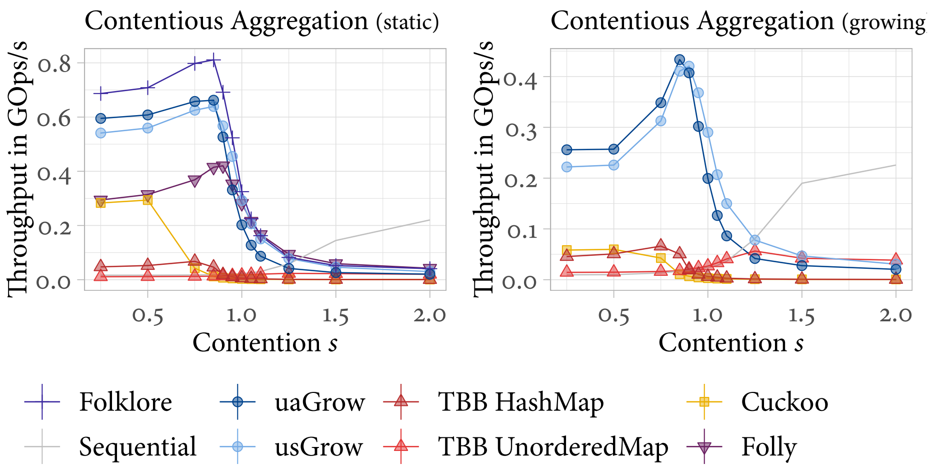

Figure 4.7: Throughput of an aggregation with a skewed input distribution. Executing 10^8 insert-or-increment operations ($p = 64$ threads executed on *amd-epyc*). *Left*: experiment using a static table size; *right*: the same experiment using a growing table.

Hash tables are often used for key aggregation. The idea is that all data elements connected to the same key are aggregated using a commutative and associative function. For our test, we implemented a simple key count program. To implement the key count routine with a concurrent hash table an insert-or-increment function is necessary. Our tables update function can easily be adapted by using specialized

key-count benchmark

update functions (see Section 4.3). We use the same hash tables as in the previous test except for all junction tables ◆, ◆, ◆. In junction's update interface there is no possibility to define an update function where the resulting value (after the update) depends on the previous value (i.e., increment). This was mainly a problem of the used interfaces, therefore, it could probably be solved by reimplementing a more functional interface.We also left folly out of the growing plot because it does not allow arbitrary growing factors and guessing the final table size is not as easy in this benchmark. Like in previous tests, we make two distinct measurements.

The aggregation benchmark uses the same Zipf key distribution as the previous tests with contention, however, this time we do not preinsert any elements. Instead we initialize the empty table either with 10^8 slots (*left*) or with ≈ 50 000 slots (*right*). Then we measure the time it takes to call insert-or-increment on each key. The first thread that calls insert-or-update with a given key inserts that key. All further threads add one to the count. The number of distinct elements in the hash table is dependent on the contention of the key sequence (given by s). This makes growable hash tables even more desirable because the final size can only be guessed before the execution.

10^8 insert-or-update operations

no preinserts (just table initialization)

The results, shown in Figure 4.7, are fairly similar to the contentious overwrite benchmark shown in Figure 4.6 (*left*), i.e., slightly increasing performance until $s = 0.85$ with a sharp decline afterwards. However, changing a value by increment has some slight differences to overwriting it since the updated value of an insert-or-increment is dependent on its previous value. In the best case this increment can be implemented using an atomic fetch-and-add operation (i.e., usGrow ●, folkloreHT +, and folly ▼). However this is not possible for all hash tables. Sometimes dependent updates are implemented using a read-modify-CAS cycle (i.e., uaGrow ●) or fine grained locking (i.e., tbb hash map ▲ or cuckoo ■).

atomic fetch-and-add

Until $s = 0.85$, uaGrow ● seems to be the more efficient of the two growing option since it has an increased writing performance and the update cycle are successful most of the time. From that point on, usGrow ● is clearly more efficient because fetch-and-add behaves better under contention. Folly ▼ can use the same fetch-and-add operation and thus catches up with our specialized implementation for higher skews. On higher loads, both usGrow ● and folly ▼ actually catch up to folkloreHT + which again performs the best out of all implementations. In the growing benchmark we see more or less the same behavior with the main difference being that TBBum makes an appearance among the front runners (for really high skews).

Arbitrary Data Types

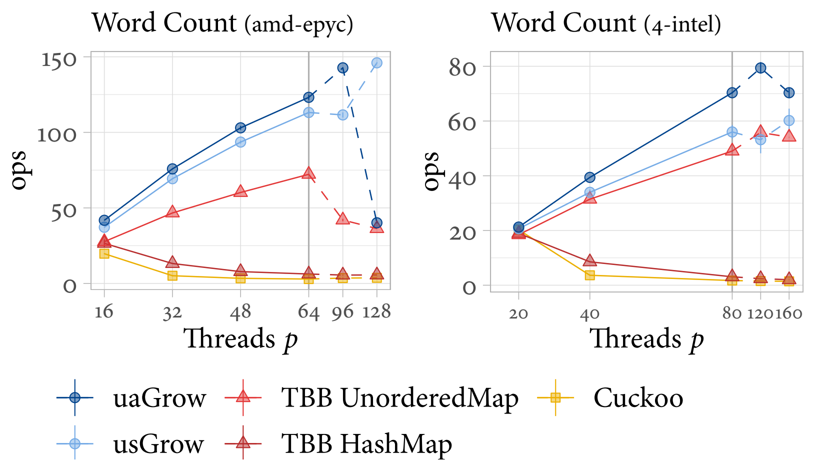

Figure 4.8: Word count benchmark using string-keys taken from the Gutenberg library.

In essence this test is very similar to the previous test. We again use insert-or-increment to count the number of occurrences of individual keys. However, this benchmark is special as it uses both non-integer keys (strings) and a real world data set, i.e., the Gutenberg library containing thousands of books. In Section 4.4.7 we describe how to implement support for arbitrary data types within our hash table architecture based on folkloreHT. The data set we used contains 3 496 038 668 words with 46 493 779 individuals, thus each word appears on average ≈ 75.2 times.

real world Gutenberg data

$3.4 \cdot 10^9$ *words* $4.6 \cdot 10^7$ *individuals*

Some Implementation Details In C++ creating many `std::string` objects repeatedly allocates small amounts of memory. This is quite inefficient. It is possible, however, to use a specialized allocator for these implementations. Thus we specialized the standard string class to use TBB's scalable allocator, reducing the overall overhead.

scalable strings

To prepare the data set, we first put all texts into one large file (22 GB). To distribute operations (i.e., words) among the processors of our machine we proceed similar to our migration algorithm. Each thread continuously reserves one block of this file, moves its file descriptor to the beginning of the block, and adds all words starting in its block to the quotient filter using an insert-or-update operation similar to the previous experiment.

work distribution

In Figure 4.8, we see that our table variants perform the best among all competitors. Both tbbHM ▲ and cuckoo 🟨 have speedups < 1 with increasing processor counts (only achieving 5 % and 2 % of uaGrow ●'s throughput $p = 64$). This is likely caused by the fact that both variants have to use fine grained locking to ensure consistent behavior. TBBum ▲ performs well in this benchmark. It achieves 59 % of uaGrow ●'s throughput. Overall, we achieve throughputs of 123 MOps/s (for uaGrow ●) and 113 MOps/s (for usGrow ●). This is about 59 % of the growing insert performance (with uniform integer keys) and around 16 % of the positive query performance. However, we do not know how much the potential contention has influenced the performance in either way. Overall this seems to be a realistic overhead for arbitrary data types—accentuating why it is important to have specialized solutions for (common) smaller data types.

overhead of arbitrary data types

important to have specialized tables for small data types

Deletion Tests

As described in Section 4.4.4, we use migration not only to implement an efficiently growing hash table but also to clean up the table after deletions. This way all tombstones are removed and thus freed slots are reclaimed. But how does this fare against different ways of removing elements? This is what we investigate with the following benchmark.

First we initialize a table with $\approx n$ slots (i.e., can hold the whole window), thus tables that offer true deletions (like cuckoo 🟨) do not need to use any form of growing. Then we prefill the table with n elements. The test itself consists of 10^8 insertions—each immediately followed by a delete operation. Therefore, the table remains at approximately the same size throughout the test ($\pm p$ elements). All inserted keys are generated before the benchmark using a uniform distribution (similar to our other tests). Each thread has a circular array of its n/p last inserted keys. After each

alternating insert and delete operations

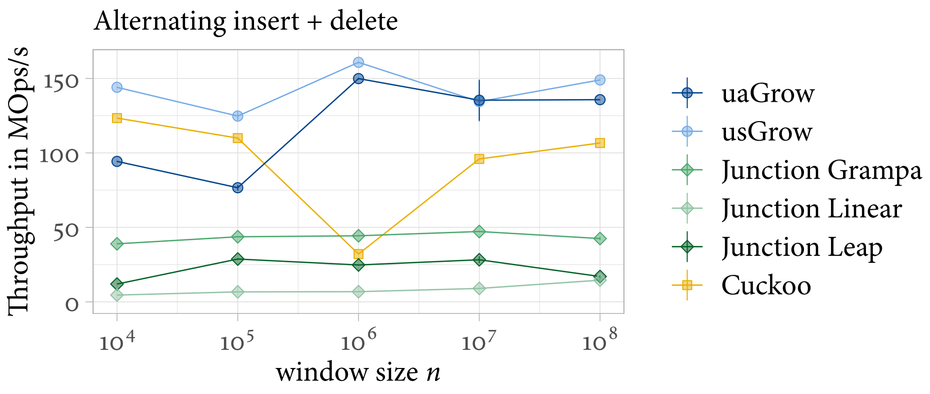

Figure 4.9: Throughput of alternating insert and delete operations, thus keeping the table at a constant size n. The table contains a sliding window of n elements (executed on *amd-epyc* with $p = 64$). The measured throughput combines one iteration of insert+delete = $1Op$.

insertion the new key is put into the array and the deletion is called on the replaced key, thus each deleted key is guaranteed to be in the table.

The results of this experiment (see Figure 4.9) show that our implementation performs better than other available implementations. It also indicate that there is no strong correlation between performance of operations the size of the table (i.e., the size of the window). This is what one would expect because the performance of hash table operations is not dependent on their size. However, this part of the result is not as clean as one would have expected. On smaller window sizes, there are a lot more migrations leading to increased synchronization overheads compared to synchronization free insert operations. The absolute performance of usGrow ● ≈ 150 MOps/s (i.e., around 300 MOps/s single insert or delete operations) is in line with the performance of insertions into a growing table.

Throughputs are in line with growing insertions

Using Dedicated Growing Threads

In Sections 4.4.3 and 4.5 we describe the possibility of using a pool of dedicated migration threads which grow the table cooperatively. To test this variant we use two tests where tables have to be migrated repeatedly, i.e., inserting into a grow-

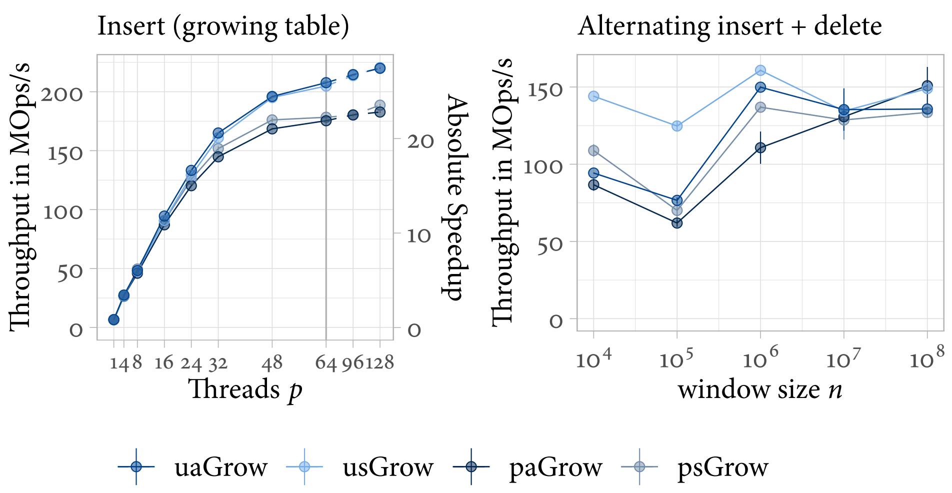

Figure 4.10: Testing the performance of our migration thread pools. *Left*: Insertions into a growing table (compare Figure 4.3); *right*: alternating insert and delete operations (compare Figure 4.9)

ing hash table (Figure 4.10 *left*) and the deletion test which alternates insertions and deletions (*right*). The insertion benchmark grows the table around 3 orders of magnitude, as such it forces around 10 to 11 full table migrations of increasing size. The alternating deletion test causes even more migrations especially on smaller window sizes. The table is repeatedly migrated without size increases, to remove unnecessary tombstones from the data structure. We see that the variants using a thread pool perform similar albeit somewhat slower than the recruitment variants.

benchmarks that caus migrations

The overhead arises because using additional migration threads necessitates some communication with the operating system. The growing threads have to sleep while they are not needed (otherwise normal operations would become slow)—using a conditional wait syscall (i.e., futex). Then the migration threads are awoken and operating threads cannot access the data structure during the migration (they sleep). This all happens once during each migration cycle (scheduling and notification). Originally, we designed the thread-pool-variant to prevent slow table migrations in cases where only few application threads actually work on the table (when using the recruitment variants, this thread would have to do all the

overheads caused by communication (with operating system)

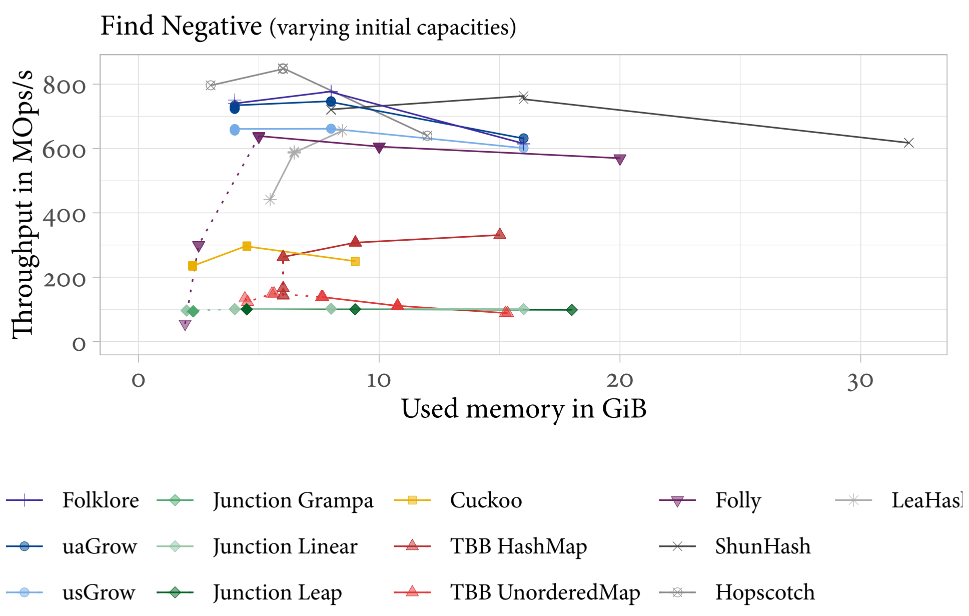

Figure 4.11: Performance of negative find queries relative to the memory footprint. Tables are initialized with different initial capacities (*Note*: contrary all of our other plots, this plot does not show standard deviations).

work). However, this advantage does not come into play in these tests, since all of the application threads actually operate on the table.

Memory Consumption

memory consumption One aspect of parallel hash tables that we did not talk about throughout this chapter is memory consumption. Overall, a low memory consumption is preferable but having less slots means that there are more hash collisions. This leads to longer running times especially for non-successful find operations. It would be interesting to adapt a hash tables size in order to increase its capacity. Most hash tables do not allow the user to set a specific table size directly. Instead they are initialized using the expected number of elements. We use this mechanism to create tables of different sizes. Using these different hash tables with different sizes, we find out how well any one hash table scales when it is given more memory. This is interesting for applications where the hash table speed is more important than its memory footprint (lookups to a small or medium sized hash table within an application's inner loop).

The values presented in Figure 4.11 are acquired by initializing each hash table with varying initial table capacities. Non-growing tables are created with initial capacities of with $(1, 1.2, 1.5, 1.8, 2, 2.5, 3, 4) \times n$ (where $n = 100\,\mathrm{M}$) and growing hash tables are additionally initalized with 50 000, $(0.2, 0.4, 0.6, 0.8) \times n$ (folly does not use 50 000). Afterwards the table is filled with $n = 10^8$ elements. The plotted measurements show the throughput of executing 10^8 unsuccessful lookups on the preinitialized table (compare Figure 4.4). This throughput is plotted over the amount of allocated memory each hash table used. To measure the memory consumption we use the *resident set size* (number of actual physical memory pages that are loaded in memory) which we query after all elements have been inserted. Measurements are connected with lines (for better visibility), these lines are in no way interpolating between the measurements—arbitrary capacities seem to be impossible for all tested hash tables (i.e., all hash tables snap to capacities that are probably powers of two in addition to some overheads). Measurements that are initialized with an initial capacity smaller than n are marked with dashed lines.

vary initial capacity

resident set size

all hash tables snap to "≈ powers of two"

The minimum size for any hash table should be around $1.53\,\mathrm{GiB} \approx 10^8 \cdot (8\,\mathrm{B} + 8\,\mathrm{B})$ (Key and Value each have 8 B). Our hash table uses a number of slots equal to the smallest power of 2 that is at least two times as large as the expected number of elements. In this case, this means we use $2^{28} \approx 2.7 \cdot 10^8$, therefore, the table is filled to ≈ 37 % and use exactly 4 GiB. We believe that this memory usage is reasonable, especially for heavily accessed tables where the performance is important. This is supported by our measurements as all hash tables that use less memory have bad performance (*Note*: both hopscotch ⊠ and leahash ✳ only have a hash set interface).

minimum table size 1.53 GiB

We see that most tables cannot profit from the increased table capacity. One reason for this is probably the memory performance of our machine *amd-epyc* and speed the at which memory is prefetched. The only table that profits significantly from size increases is leahash. It is a hash table that uses chaining, thus having more chains that are empty leads to more trivial negative find queries.

4.7 Conclusion

large performance gap from specialized folklore to available general purpose

We demonstrate that a bounded linear probing hash table specialized to pairs of machine words has much higher performance than currently available general purpose hash tables like Intel TBB, Cuckoo, or RCU based implementations. This is not surprising from a qualitative point of view given previous publications [84, 34, 81]. However, we found it surprising how big the differences can be in particular in the presence of contention. For example, the fact that a hash table requires a lock for reading can decrease its performance by multiple orders of magnitude.

all restrictions can be overcome

We have also shown, that all restrictions that come with this hashing technique, i.e., static table size, no deletions, and restricted key- and value-types, can be overcome. Our generalizations can be combined to give us a wide variety of different implementations (folkloreHT, uaGrow, usGrow, paGrow, and psGrow each with a specialized version for complex key and value types). Moreover, in dedicated tests our specialized variants continuously outperform any (sufficiently different) competitor table. Usually by at least a factor of two or greater. This is true for growing hash tables both in our insert and find benchmarks; for hash tables with deletions; and also for hash tables with arbitrary data types.

our solutions continuously outperform other options

library implementation

What we have not talked about yet is the fact that we have combined these variants into a library [43] that given a set of simple parameters combines the described extensions (at compile time) to construct a table type that offers all the necessary functionality. Without hand picking combinations of specialized and extended classes and templates. We have constructed the table in a way that is as close to the C++ standard hash-table-interface as possible without sacrificing any performance (e.g., including iterators).

migration technique

One of our main contributions is also the efficient and communication avoiding migration algorithm where blocks are assigned to cores using a simple atomic variable and interaction between cores is avoided by implicitly moving the borders of assigned blocks to free slots. This allows the migration to work without any form of atomic operations in the target table thus reducing the interaction between threads. The only overhead that comes with our deletion technique is that hash tables have to be growable. As long as this is the case, other operations are not impacted by deletions.

Further directions of research could be to look into translating the same low-communication, concurrent migration mechanism to different hashing schemes (i.e., cuckoo hash tables or hopscotch hash tables). One such example could be the space efficient DySECT table introduced in Section 3, there we are already using a similar algorithm for its subtable migrations. However, to implement fast concurrent DySECT tables, we would also have to implement a concurrent displacement algorithm. Similarly, the concurrent quotient filter shown in Section 5 also uses an adapted variant of the same migration technique. Even non-hash-based data structures could benefit from similar element migrations that reduce the comunication (i.e., necessity for atomic operations) by preserving element order and partitioning the target data structure among threads.

applying the migration to other data structures

5 Concurrent Quotient Filters

AMQ data structures are often used to improve the scalability of complex storage solutions like databases and online/cloud storage, for example to reduce the number of unnecessary queries. However AMQ-filters themselves also offer interesting opportunities when analyzed for their scalability. The main idea behind AMQ data structures is to improve both speed and memory performance by relaxing correctness guarantees, i.e., contains queries are allowed to output false positives.

Today, AMQ-filters are used on the largest of inputs—like huge bioinformatics databases. As such it is natural that they are built on high performance hardware that has the capabilities to cope with the amount of data. On these machines, concurrency is a necessity and scalability of data structures to large processor counts is important for many algorithms to achieve satisfactory speedups. The findings of our research throughout this chapter can be found in our library lpqfilter [44].

So far, we have considered a number of dynamic hash tables. *Approximate Membership Query data structures* (AMQ-filter; see Section 2.1.3) are a different type of hash-based data structure that are interesting to reevaluate from a scalability perspective. Similar to hash sets, AMQ-filters are usually used to represent sets, thus, they offer *insert* and a *contains* operations (sometimes also *deletions*). However, in contrast to classical hash sets that more or less function the same as hash tables, AMQ-filters offer an interesting tradeoff. They allow *false positives* on their contains queries, meaning a contains query can return true even if the queried ele-

AMQ-filter

false positives

ment is not part of the set. This allows us to trade the exactness of the data structure to safe memory or improve the performance.

quotient filter

Specifically we take look at different variants of *quotient filters*. A quotient filter works similar to a hash table with in-table-displacement (see Section 2.4.2). But instead of storing full elements, quotient filters only store fingerprints of the inserted elements. Queries check whether the requested fingerprint is stored in the data structure and return true iff that is the case. Given the similarity to in-table-displacement (specifically linear probing and Robin Hood hashing) we can use methods that are similar to the previous sections to open up an interesting design space and develop a variety of new adaptations.

fewer memory accesses than Bloom filters

supports table migration

One of the main advantages of quotient filters is their cache efficiency. Similar to hash tables with linear probing, quotient filters can often insert elements and answer queries by accessing a few consecutive cache lines. What makes quotient filters even more interesting from a scalability perspective is that quotient filters offer the opportunity for table migrations similar to growing a hash table. This is uncommon among AMQ-filters. The table migration uses a more or less traditional migration approach—similar to the linear probing tables analyzed in Section 4.4.3.

5.1 References

The results presented in this chapter are based on a conference paper [50] published jointly, with Peter Sanders and Robert Williger. The paper was mainly written by the author of this dissertation, with Peter Sanders mostly contributing to the ideas and conception (in addition to some editing of the publication) and Robert Williger, mostly contributing to the initial implementation. Similar to previous chapters, some of the original texts are used verbatim or with fairly small changes.

5.2 Introduction

Motivation

AMQ-filter have become an integral part of many complex data structures and database applications. Their small size and fast access times can be used to sketch large, slow data sets. In these cases a fast AMQ-filter is queried before accessing the database to check whether the slow database lookup is actually necessary. Quotient filters have recently been used in network analysis [2] and bioinformatics [71]. Data sets in these two areas are among the largest data sets available today, but given the current big data revolution we expect more and more research areas and industry applications will have a need for the space efficiency and speedup potential of AMQ data structures.

increased necessity due to new big data applications

The most common AMQ data structure in practice is still a Bloom filter. In our experience and in preliminary experiments, however, quotient filters consistently outperform Bloom filters on many workloads (see Section 5.7). We believe that one reason for the continued (perceived) predominance of Bloom filters is likely inertia, but another reason might be that concurrent Bloom filters are easy to implement—even lock-free and scalable implementations. This is important because scalable implementations have become critical to handle growing data sets in today's multi-processor scenarios.

Bloom filters are ubiquitous despite of shortcomings

Contribution

Typically concurrent quotient filters are implemented using an external array of locks—each protecting a region of the table. Accessing this array incurs one additional memory access per operation—negating the advertised cache efficiency. We propose a new fine grained locking scheme that stores locks inside the table and has no memory overhead. Instead of traditional locks we use the content of table slots to represent very localized locks that lock specific clusters in the table. Using this new locking scheme we achieve 1.6 × times higher insertion performance and over 1.8 × higher query performance than with the common external locking scheme.

zero-memory-overhead locking scheme

Additionally, we propose several unique quotient filter variants that aim to reduce the number of status bits (2-status-bit-variant—2BQ-filter) or to simplify concurrent implementations (linear probing quotient filter—LPQ-filter). The linear probing quotient filter variant that we present even leads to a lock-free (see Section 2.2) concurrent filter implementation. What makes this especially interesting is that we can show that any lock-free implementation of other common quotient

newly introduced variants 2BQ-filter and LPQ-filter

filter variants would incur significant overheads in the form of additional data fields or multiple passes over the accessed data.

Quotient filters allow increasing the table capacity through migrating elements into a larger table (similar to hash tables). However, the false positive rate does not change when growing the AMQ-filter. Thus, the false positive rate continues to grow with the number of additional insertions. Hence, this growing method is only useful for a limited number of migrations (before the fp rate becomes too high). We implement this growing technique for our concurrent quotient filters and extend it to allow unbounded growing while maintaining a *bounded false positive rate*. We call the resulting data structure a *fully expandable quotient filter*. Its design is similar to scalable Bloom filters [3], but we exploit some concepts (like the limited growing technique) inherent to quotient filters to improve the space efficiency and the query speed.

limited growing

5.3 Related Work

The concept of quotient filters is based on compact hash tables [14] that store part of their key implicitly (via the table possition). However the first specific mention in the context of AMQ data structures was in 2012 by Bender et al. [8]. Since then, there has been a steady stream of improvements. For example Pandey et al. [70] have shown how to reduce the memory overhead of quotient filters by using rank-select data structures. This also improves the performance when the table becomes full. Additionally, they show an idea that saves memory when insertions are skewed (some elements are inserted many times). They also mention the possibility for concurrent access using an external array of locks (see Section 5.7 for results). Recently, Geil et al. [29] proposed a GPU-based implementation of quotient filters, further indicating that there is a lot of interest in concurrent AMQ-filters even in these highly parallel scenarios.

different variants and use cases of quotient filters

Quotient filters are not the only AMQ data structures that have received attention recently. *Cuckoo filters* [24, 61] and very recently *Morton filters* [10] (based on cuckoo filters) are two other examples of AMQ data structures. Due to their similarity to cuckoo hash tables (see Section 2.4.2, page 37), they do not lend themselves

cuckoo and Morton filters

to easy parallel solutions (insertions can have large effects on the overall data structure).

A *scalable Bloom filter* [3] allows unbounded growing by adding additional levels of Bloom filters once a level becomes full. Each new filter is initialized with a higher capacity and more hash functions than the last. The query time is dependent on the number of times the filter has grown both because more filters have to be checked and because later filters have more hash functions. In Section 5.6, we show a similar technique for fully expandable quotient filters that mitigates many of these problems.

scalable Bloom filter allows capacity increase

5.4 Sequential Quotient Filter

In this section we describe the basic sequential quotient filter as well as some variants to the main data structure. We use the same naming conventions as for hash tables, i.e., m is the number of slots, n the number of elements, and $\delta = n/m$ is the fill degree of a table. In addition to these, we define the *false positive rate* p^+ to denote the probability of a false positive query, i.e., the probability of a query looking for a random non-present element to return true (see Section 2.1.3).

false positive rate p^+

5.4.1 Basic Quotient Filter

Quotient filters are approximate membership query data structures that were first described by Bender et al. [8] and build on an idea for space efficient hashing originally described by Cleary [14]. Quotient filters represent possibly large elements by *fingerprints*. The fingerprint $f(x)$ of an element x is a number in a predefined range $f : x \mapsto \{0, \ldots, 2^k - 1\}$ (binary representation with exactly k digits). We commonly obtain a fingerprint of x by taking the k least significant bits of a hash function value $h(x)$ (i.e., xxHash [15]).

fingerprint $f(\cdot)$

A quotient filter stores the fingerprints of all inserted elements. When executing a query for an element x, the filter returns **true** if the fingerprint $f(x)$ was previously inserted and **false** otherwise. Thus, a query looking for an element that was inserted always returns **true**. A false positive occurs when x was not inserted, but its fingerprint $f(x)$ matches that of a previously inserted element. Given a fully

query for x returns true iff $f(x)$ was inserted random fingerprint function, the probability of two fingerprints being the same is 2^{-k}. Therefore, the probability of a false positive is bounded by $n \cdot 2^{-k}$ where n is the number of stored fingerprints.

$p^{+} = n \cdot 2^{-k}$

To achieve expected constant query times as well as to save memory, fingerprints are stored in a special data structure that is similar to a hash table with in-table-displacement. During this process, the fingerprint of an element x is split into two parts: the q most significant bits called the quotient $quot(x)$ and the r least significant bits called the remainder $rem(x)$ with $q + r = k$. The quotient is used to address a table consisting of $m = 2^q$ memory slots of $r + 3$ bits. A slot can store one remainder and three additional status bits. The quotient of each element is only stored implicitly by the position of the element in the table. The remainder is stored explicitly within one slot of the table. Similar to many hashing techniques, we try to store each element in one designated slot (index $quot(x)$) which we call its *canonical slot* (see Section 2). With the help of the *three status bits* we can reconstruct quotient of each stored element even when it is not placed in its canonical slot.

quotient and remainder

$r + 3$ bits per slot

canonical slot

3 status bits

The main idea for resolving collisions is to find the next free slot—similar to linear probing hash tables. However, we reorder the elements such that they are sorted by their fingerprints (see Figure 5.1). This is similar to Robin Hood hashing (see Section 2.4.2, page 34), thus, we use some of the same notation. Elements with the same quotient (the same canonical slot) are stored in consecutive slots, we call them a *run*. The *canonical run* of an element is the run associated with its canonical slot. The canonical run of an element does not necessarily start in its canonical slot. It can be shifted by other runs. If a run starts in its canonical slot we say that it starts a *cluster* that contains all shifted runs that follow. Multiple contiguous clusters (i.e., clusters that have no free slots between them) form a *supercluster*. We use the 3 status bits that are part of each slot to distinguish between runs, clusters, and empty slots. For this we store the following information about the contents of the slot (further described in Table 5.1): were elements hashed to this slot (is its run non-empty)? Does the element in this slot belong to the same run as the previous entry (used as a run-delimiter signaling where a new run starts)? Does the element in this slot belong to the same cluster as the previous entry (is it shifted)?

sorted by fingerprint

(canonical) run

(super) cluster

assignments of status bits
- was hashed to
- run continuation
- cluster continuation

`1**`	this slot has a run
`000`	empty slot
`100`	cluster start
`*01`	run start
`*11`	continuation of a run
`*10`	– (not used, see 5.5.2)

Table 5.1: Meaning of different status bit combinations.

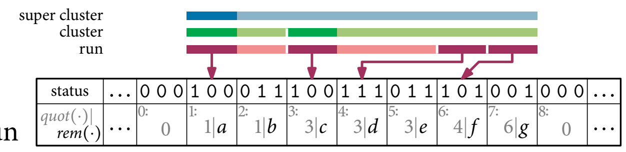

Figure 5.1: Section of the table with highlighted runs, clusters, and superclusters. Runs point to their canonical slot.

During the query for an element x, all remainders stored in x's canonical run have to be compared to $rem(x)$. First we look at the first status bit of the element's canonical slot. If this status bit is not set there is no run for this slot and we can return false. If there is a canonical run, we use the status bits to find it by iterating to the left until the start of a cluster (third status bit = 0). From there, we move to the right counting the number of non-empty runs to the left of the canonical run (slots with first status bit = 1 left of $quot(x)$) and the number of run starts (slots with second status bit = 0). This way, we can easily find the correct run and compare the appropriate remainders.

query compares remainders in canonical run

find run by counting runs from cluster start

An insert operation for element x proceeds similar to a query, until it either finds a free slot or a slot that contains a fingerprint that is $\geq f(x)$. The current slot is replaced with $rem(x)$ shifting the following slots to the right (updating the status bits appropriately).

When the table fills up, operations become slow because the average cluster length increases. When the table is full, no further insertions are possible. In this case, quotient filters can be migrated into a larger table, increasing the overall capacity. To do this, a new table is allocated with twice the size of the original table and one bit less per remainder. Addressing the new table demands an additional quotient bit, since it is twice the size. This issue is solved by moving the uppermost bit from the remainder into the quotient ($q' = q + 1$ and $r' = r - 1$). As fingerprint size and number of elements remain the same, the capacity increase does not affect the false positive rate of the filter. But the false positive rate still increases linearly with the number of insertions ($p^+ = n \cdot 2^{-k}$). Therefore, the false positive rate dou-

migration to increase capacity

bles when the table is filled again. We call this migration technique *limited growing*, because to guarantee reasonable false positive rates this method of growing should only be used a limited number of times.

5.4.2 Variants

During our work with quotient filters we have developed the following variants both varying the use of status bits and fingerprints. The basic idea behind both variants is that if we enforce $rem(x) \neq 0$. We sacrifice some potential fingerprints—little influence on p^+—but we can then differentiate empty from filled slots thus reducing the necessary number of status bits.

Two-Status-Bit Quotient Filter—2BQ-filter

Pandey et al. [70] already proposed a 2-status-bit-variant of their counting quotient filter. Their implementation however is closer to a thought experiment and serves as a motivation for their rank-select based implementation. It has average query and insertion times in $\Theta(n)$. The goal of our 2-status-bit-variant is to achieve running times close to $O(\textit{supercluster length})$. We change the definition of the fingerprint (only for this variant) such that no remainder can be zero, $f' : x \mapsto \{0, ..., 2^q - 1\} \times \{1, ..., 2^r - 1\}$. Obtaining a non-zero remainder can easily be achieved by rehashing an element with different hash functions until the remainder is non-zero[1]. This change to the fingerprint only has a minor impact on the false positive rate of the quotient filter ($n/m \cdot (2^r - 1)^{-1}$ instead of $n/m \cdot 2^{-r}$).

force $rem(x) \neq 0$

Because there is no element with $rem(x) = 0$ we can easily distinguish empty slots from filled ones. Each slot uses two status bits: the *occupied-bit* (first status bit in the description above) and the *new-run-bit* (run-delimiter). Using these status bits we can find a particular run by going to the left until we find a free slot and then counting the number of occupied- and new-run-bits while moving right from there (*Note*: a cluster start within a larger supercluster cannot be recognized when moving left).

find supercluster start via empty slot

[1]slightly worse bounds can be achieved without rehashing—using $rem'(x) = \max\{1, rem(x)\}$

Linear Probing Quotient Filter—LPQ-filter

This quotient filter variant is a hybrid between "pure" linear probing and classic quotient filters. It uses no reordering of stored remainders and no status bits. Similar to the two-status-bit quotient filter above we ensure that no element x has $rem(x) = 0$ by adapting the fingerprint function.

no reordering and no status bits

During the insertion of an element x the remainder $rem(x)$ is stored in the first empty slot after its canonical slot. Without status bits and reordering it is impossible to reconstruct the fingerprint of each inserted element. Therefore, a query looking for an element x compares its remainder $rem(x)$ with every remainder stored between x's canonical slot and the next empty slot. There are potentially more remainders that are compared than during the same operation on a normal quotient filter. To offset that, we add three additional bits to the remainder (leading to a longer fingerprint), while not using more memory than a classic quotient filter. The increased remainder length reduces the chance of a single comparison to lead to a false positive by a factor of 8. Therefore, as long as the average number of compared remainders is less than 8 times higher than before, the false positive rate remains the same or improves. In our tests, we found this to be true for fill degrees up to $\approx 70\,\%$. The number of compared remainders corresponds to the number of slots probed by a linear probing hash table. It should be noted that LPQ-filters cannot support deletions or the limited growing technique.

more remainder bits counteract more comparisons

Lemma 2 (p^+ LPQ-filter)**.** *The false positive rate of a linear probing quotient filter with $\delta = n/m$ (number of comparisons due to Knuth [37], chapter 6.4) is*

$$p^+ = \frac{E[\#comparisons]}{2^{r+3}-1} = \frac{1}{2}\left(1 + \frac{1}{(1-\delta)^2}\right) \cdot \frac{1}{2^{r+3}-1}$$

5.5 Concurrent Quotient Filter

Besides correctness there are two main goals for any concurrent data structure—scalability and overall performance. One major performance advantage of quotient filters over other AMQ data structures is their cache efficiency. Concurrent quotient filters should strive to preserve this advantage, especially on insertions into short or empty clusters and on (unsuccessful) queries with empty canonical runs.

preserving cache efficiency

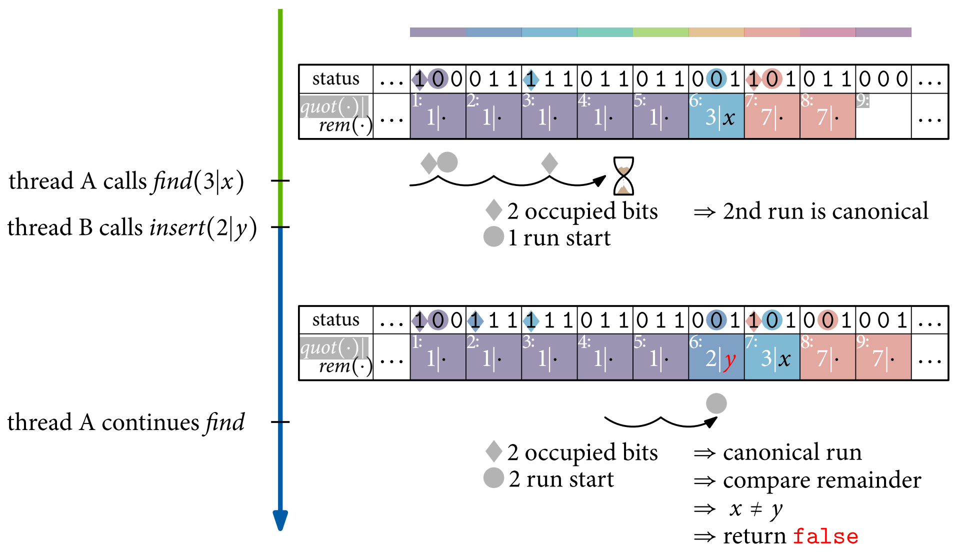

Figure 5.2: Example of a query, where observing two parts of individually consistent states leads to a false negative. Thread A queries a contained element. During the query thread B changes the data structure. This leads to A checking the wrong run and results in a false negative.

These lead to operations that access only one or two cache lines and lead to at most one write operation.

additional array of locks

From a theoretical point of view, using an external array of locks should lead to a good concurrent quotient filter, as long as the number of locks is large enough to reduce contention and the number of slots per lock is large enough to ensure that clusters don't span multiple locking regions. A similar solution was described by Pandey et al. [70]. They also describe a variant that handles contention by first inserting elements into a local filter if acquiring the appropriate lock failed. Nonetheless, having an external array of locks introduces a significant overhead especially on the easy operations mentioned above. This assumption is confirmed in our experiments (see Section 5.7), where we compare against the counting quotient filter by Pandey et al. and against our own implementation of this simple locking technique.

external locking performs badly in practice

The Question of Lock-Free Quotient Filters

One way to achieve theoretically provable scalability is *lock-freeness* as defined in Section 2.2. However, there are some problems inherent to lock-free quotient filters. Every query has to read multiple data entries in a consistent state to succeed. All commonly known variants (except the LPQ-filter) use at least two status bits per slot, i.e., the *occupied-bit* and some kind of *run-delimiter-bit* (the run delimiter might take different forms). The remainders and the *run-delimiter-bit* that belong to one slot cannot reliably be stored close to their slot and its *occupied-bit*. Therefore, the *occupied-bit* and the *run-delimiter-bit* cannot be updated with one atomic operation. For this reason, implementing a lock-free quotient filter that uses status bits would cause significant overheads (i.e., additional memory or multiple passes over the accessed data). The problem is that reading parts of multiple different but individually consistent states of the quotient filter leads to an inconsistent perceived state (a scenario visualizing this can be seen in Figure 5.2). To show this we assume that insertions happen atomically and transform the table from one consistent state directly into the new final consistent state. During a query we have to find the canonical run and scan the remainders within this run. To find the canonical run we have to compute its rank among non-empty runs using the occupied-bits (either locally by iterating over slots or globally using a rank-select data structure) and then find the run with the same rank using the run-delimiter-bits. There is no way to guarantee that the overall state has not changed between finding the rank of the occupied-bit and finding the appropriate run-delimiter. Specifically we might find a new run that was created by an insertion that shifted the actual canonical run. Similar things can happen when accessing the remainders, especially when remainders are not stored interleaved with their corresponding status bits. Some of these issues can be mitigated using multiple passes over the data but ABA (see Section 2.2) problems might arise in particular when deletions are possible. To avoid these problems while maintaining good performance we have two options: either comparing remainders from different canonical slots and removing the need for status bits (i.e., the LPQ-Filter) or by using locks that protect the relevant range of the table.

query needs consistent state of the whole cluster

each run:
- occupied-bit in canonical slot
- run-delimiter-bit at run start

parts of different consistent states can form an inconsistent perceived state

LPQ-filters have no status bits thus they avoid this problem

Concurrent Storage of Arbitrarily Sized Slots

Whenever data is accessed concurrently by multiple threads, we have to think about the atomicity of data manipulation. To reduce the number of cache lines accessed during an operation and to be able to change both status bits and remainder atomically, they need to be stored together in the same atomic data element (i.e, alternating status bits and remainders).

atomic data members with arbitrary sizes

The size of a quotient filter's slots depends on the target false positive rate of the quotient filter, thus, it is common that slots have odd sizes (i.e., non-powers of two). In practice however, atomic operations only work on certain predefined atomic data types (typically integers with 1, 2, 4, or 8 bytes). Additionally, atomic operations become really slow (or depending on the architecture even impossible) unless the data objects are aligned (i.e., address is divisible by object size). When combining both of these facts it becomes clear, that slots cannot be stored consecutively in memory without any form of *memory offcuts* (i.e., some remaining unused bits). We have to take into account the memory that is wasted by memory offcuts.

wasted memory due to offcuts

In the trivial case, i.e., one slot per atomic data object, the offcuts are often infeasibly large. To reduce memory offcuts to a minimum, we use the largest common atomic data type (64 bit) and pack as many slots as possible into one data element—we call this packed construct a *slot group*. This way, the potential waste of multiple slots accumulates to encompass additional slots. Furthermore, using this technique we can atomically read or write multiple slots at once—allowing us to update them all at once and even avoid some locking (if the relevant parts of a cluster fit in one group).

slot group

5.5.1 Concurrent Linear Probing Quotient Filter

LPQ-filters (as described in Section 5.4.2) are a simplification of the general quotient filter concept. Fingerprints are still split into quotient and remainder, the quotient is still used for addressing the table, and the remainder is still stored within the table. But instead of implicitly storing the quotient via status bits and element order we discard the quotient as soon as the remainder is inserted. This change makes concurrent LPQ-filters much easier to implement because inserting an element only ever changes one slot.

Thus, operations on an LPQ-Filter can be executed concurrently similar to operations on a concurrent linear probing hash table (see Section 4). The table consists of grouped atomic slots as described above. Each insertion changes the table atomically using a compare-and-swap instruction on a single slot group. This implementation is even lock-free because at least one competing write operation on any given slot group is successful. Queries find all remainders of elements that were inserted into their canonical slot because all such remainders were stored in the first empty slot after their canonical slot and the contents of a slot never change once something is stored within it.

similar to concurrent linear probing hash table

lock-free

This implementation of concurrent quotient filters has significant advantages when it comes to performance and scalability, e.g., it is lock-free, no writes occur during contains queries and correctness and linearizability follow the same principles as linear probing hash tables. But it also inherits the downsides that sequential LPQ-filters carry, i.e., increased false positive rate over 70 % fill degree, no support for deletions and no support for migration of elements (for growing).

pro: *simple, lock-free, no writes during contains*

contra: *bad when full, no deletions, no migration*

5.5.2 Concurrent Quotient Filter with Local Locking

In this section we introduce an easy way to implement concurrent quotient filters with a simple *local locking protocol*. Our protocol is based on previously unused status bit combinations. Thus, it does not require any additional memory and does not increase the number of slots that are accessed during operations. Our concurrent implementation is based on the basic (3-status-bit) quotient filter. But we show that the same locking idea also applies to the 2BQ-filter variant, thus, a similar protocol is possible albeit practically infeasible because it needs unaligned compare-and-swap operations (details to come later in this section). One of the main advantages of our local locking technique is that under many circumstances locks can be avoided. This is most often the case if operations work on short or empty clusters.

local locking protocol

avoid locking small clusters

Using status bits for local locking

To implement our locking scheme we use two combinations of status bits that are impossible to occur naturally (see Table 5.1)—010 and 110. We use 110 to imple-

write-lock 110 in empty slot locks preceeding supercluster

ment a *write-lock*. It is written into the first free slot after the canonical supercluster at the beginning of each write operation (using a compare-and-swap operation, see Block B in Algorithm 5.1.a). Insertions wait when encountering a write-lock. This ensures that only one insert operation can be active per supercluster. The combination `010` is used as a *read-lock*. All operations (both insertions—Block C in Algorithm 5.1.a—and queries—Block G in Algorithm 5.1.b) acquire a read-lock by replacing the status bits of the first element of their canonical cluster with `010`. To release a read-lock, we have to restore the status bits that were originally stored in its slot. Read-locks are always stored in the first slot of a cluster, thus, they can be unlocked by restoring the status bits to `100` which indicates a cluster start.

read-lock 010 in cluster start locks cluster

insertions need both a read- and a write-lock

Inserting threads can encounter read-locks in later clusters of the same supercluster. They have to wait for each encountered read-lock to be released while moving elements in the table, see Block E in Algorithm 5.1.a. This way insertions cannot interfere with concurrent read operations because changes can only occur to the left of the read lock. After waiting for the read lock, the cluster start (that was read locked) is shifted as part of the insert operation. Thus, it becomes part of the canonical cluster that is protected by the insertion's original read-lock. Contains queries never have to wait for any locks except the read-lock on their canonical cluster.

The only case where a contains query encounters any other lock—except a potential read lock on its canonical cluster start—is if it has reached the end of its cluster and either the cluster is write-locked or the next cluster is read-locked (in the same supercluster). Either way, the query is done looking at fingerprints of the canonical run (i.e., the element has not been found).

deletions

It would also be possible to implement *deletions* in a similar way to insertions (first acquiring a write-lock, then a read-lock). But both queries and insertions are affected by the possibility of holes being created within clusters while acquiring a lock. While we have not implemented deletions in our implementation, a possible deletion algorithm using the same locking scheme can be found later in this section (Algorithm 5.2).

Algorithm 5.1 Concurrent locally locked quotient filter operations.

5.1.a Insertion

```
$(quot, rem) \leftarrow f(key)$
// Block A: try a trivial insertion
$group \leftarrow$ atomically load data around $quot$
if insertion into $group$ is trivial then
    finish insertion with a CAS and return
// Block B: write-lock the supercluster
scan right from $it \leftarrow quot$
    if $it$ is write-locked then
        wait until released and continue
    if $it$ is empty then
        trylock $it$ with write-lock and break
            if lock unsuccessful then
                re-examine $it$
// Block C: read-lock the cluster
scan left from $it \leftarrow quot$
    if $it$ is read-locked then
        wait until released and retry this slot
    if $it$ is cluster start then
        trylock $it$ with read-lock and break
            if lock unsuccessful then
                re-examine $it$
// Block D: find the correct run
$occ = 0; \quad run = 0$
scan right from $it$
    if $it$ is occupied and $it < quot$ then $occ$++
    if $it$ is run start then $run$++
    if $occ = run$ and $it \geq quot$ then break
// Block E: insert into the run and shift
scan right from $it$
    if $it$ is read-locked then
        wait until released
    store $rem$ in correct slot
    shift content of following slots
    (keep groups consistent)
    break after overwriting the write-lock
unlock the read-lock
```

5.1.b Query

```
$(quot, rem) \leftarrow f(key)$
// Block F: try trivial query
$group \leftarrow$ atomically load data around slot $quot$
if answer can be determined from $group$ then
    return this answer
// Block G: read-lock the cluster
scan left from $it \leftarrow quot$
    if $it$ is read-locked then
        wait until released and retry this slot
    if $it$ is cluster start then
        trylock $it$ with read-lock and break
            if lock unsuccessful then
                re-examine $it$
// Block H: find the correct run
$occ = 0; \quad run = 0$
scan right from $it$
    if $it$ is occupied and $it < quot$ then $occ$++
    if $it$ is run start then $run$++
    if $occ = run$ and $it \geq quot$ then break
// Block I: search remainder within the run
scan right from $it$
    if $it = rem$
        unlock the read-lock
        return contained
    if $it$ is not continuation of this run
        unlock the read-lock
        return not contained
```

Avoiding locks

locking is expensive ⇒ avoiding locks pays off

Locking—both internal and external—introduces a significant overhead. Both acquiring and releasing a lock takes a compare and swap operation. Thus, avoiding locks whenever possible is an important feature of our design. Many instances of locking can be avoided, e.g., when the canonical slot for an insertion is empty (the insertion happens within a single compare-and-swap) or when the canonical slot of a query either has no run (first status bit is 0), or stores the sought fingerprint. In addition to these trivial instances of lock elision where the whole operation happens in one slot, we can also profit from our grouped atomic storage scheme. Since we store multiple slots together in one atomic data member, multiple slots can be changed simultaneously. Each operation can act without acquiring a lock if the whole operation can be completed within one slot group. The correctness of the algorithm is still guaranteed because the relevant slots cannot be part of a read-locked cluster (otherwise the operation would have to wait in the cluster start). Thus changing the global consistent state atomically cannot lead to problems with the perceived state, i.e., both the old and the new version form a consistent state with the rest of the table.

avoid locks when whole operation in one slot group

Growing concurrently

The limited growing technique described in Section 5.4.1 can be used to increase the capacity of a concurrent quotient filter similar to that of a sequential quotient filter. In the concurrent setting we have to consider two things: distributing the work of the migration between threads and ensuring that no new elements are inserted into parts of the old table that were already migrated, as otherwise they might be lost.

- distribute work of migration
- prevent insertion in old table

recruiting user threads

To distribute the work of migrating elements, we use the *recruiting user-threads* method from Section 4.4.3 (page 102). Thus, after the migration is triggered, every thread that starts an operation first helps with the migration before executing the operation on the new table. Reducing interactions between threads during the migration is important for performance. Therefore, we migrate the table in blocks. Every thread acquires a block by incrementing a shared atomic variable. The migration of each block happens one supercluster at a time. Each thread migrates all superclusters that begin in its block. This means that a thread does not migrate the first supercluster in its block if it starts in the previous block. It also means that the thread migrates elements from the next block if its last supercluster crosses the boundary to that block. The order of elements does not change during the migration, because they remain ordered by their fingerprint. In general this means that most elements within one block of the original table are moved into one of two blocks in the target table (block i is moved to $2i$ and $2i + 1$). By assigning clusters depending on the starting slot of their supercluster, we enforce that there are no two threads accessing the same slot of the target table. Hence, no atomic operations or locks are necessary in the target table.

As described before, we have to ensure that ongoing insert operations either finish correctly or help with the migration before inserting into the new table. Ongoing queries also have to finish to prevent deadlocks. To prevent other threads from inserting elements during the migration, we write-lock each empty slot and each supercluster before it is migrated. These *migration-write-locks* are never released. To differentiate migration-write-locks from the ones used during normal insertions, we write-lock a slot and store a non-zero remainder (write locks are usually only stored in empty slots). This way, an ongoing insertion recognizes that the write-lock it encountered belongs to a migration. The inserting thread first helps with the migration before restarting the insertion after the table is fully migrated. Queries can happen concurrently with the migration because the migration does not need read-locks.

Concurrent 2BQ-Filter

no unused status-bit-combinations

In the 2-status-bit-variant of the quotient filter there are no unused status-bit-combinations that can be used as read or write-locks. But we can still use the same general technique of representing a lock through slot contents that cannot occur during normal operation. The 2BQ-filter variant enforces that there cannot be an empty remainder ($rem(x) = 0$). At the same time, status bits are only set in slots that contain an element. Thus, a slot with empty remainder and non-zero status bits cannot normally occur. We can use such a slot to represent a lock. To write-lock a supercluster, we store `01` in the status bits of an otherwise empty slot after the supercluster.

use non-zero status bits with remainder 0 *as lock*

write-lock `01`

read-locks protect the whole supercluster

Read-locks are a little more difficult. They cannot work exactly the same as they do in the 3-status-bit-variant described earlier. When operating on a 2BQ-filter, we cannot recognize cluster starts within a larger supercluster—at least not when scanning to the left from the canonical slot (Blocks C and G in Algorithms 5.1.a and 5.1.b). Only supercluster starts can be identified. They are the first non-empty slot (i.e., their neighbor is empty). Therefore, we can only read-lock superclusters, not individual clusters. When scanning to the left, we have to wait at encountered read locks even if they appear within a supercluster. This happens if two supercluster grow together—after the read lock was taken.

read-lock `11`

To read-lock a supercluster we remove the remainder from the table and store it locally until the lock is released—the status bits of the supercluster start are `11` (they are not changed). Even though this variant seems to be feasible theoretically, it is not practical because we have to ensure that the read-locked slot is still a supercluster start after it was locked (the slot to its left remains empty). This is necessary to prevent potential ABA-problems where the locked element does not change its content but is not a cluster start anymore. After successfully locking a slot, it does not have to remain a super cluster start, i.e., two clusters are still allowed to grow adjacent to each other.

double-slot compare-and-swap

To ensure that the locked slot is still the supercluster start we can atomically compare-and-swap both the supercluster start and its empty neighbor at the same time. This however is a problem since both slots might be stored in different atomic slot groups or even in different cache lines. This would lead to unaligned compare-and-swap operations and, thus, poor execution time (we confirmed this hypothesis with some preliminary experiments, but have not finished a full implementation). The variant is still interesting from a theoretical perspective where a compare-and-swap operation changing two neighboring slots is completely reasonable (usually below 64 bits). Indeed this variant might be interesting in an implementation for transactional memory where it would be used if a transaction fails. In this case, unaligned atomic operations could be rare enough to be justified as a fallback.

Concurrent Deletions

The concurrent quotient filter with local locking as it is described above cannot support concurrent deletions. Implementing a deletion would be possible, however, it is at the cost of slowing down other operations, therefore, our implementations do not support deletions. Deletions would however be possible in a phase concurrent manner (compare [81]) meaning, that there could be a deletion phase, where other operations are not possible. Or slower versions of queries and insertions would have to be used in a deletion phase.

impact to other operations

See Algorithm 5.2 for a pseudocode description of the deletion algorihtm. A thread executing a deletion first has to acquire a write-lock on the supercluster then it has to scan to the left (to the canonical slot) to guarantee that there was no element removed while finding the cluster end. Then the read-lock is acquired and the deletion is executed similar to the sequential case. During the deletion it is possible that new cluster starts are created due to elements shifting to the left into their canonical slot. Whenever this happens, the cluster is created with read-locked status bits (`010`). After the deletion all locks are unlocked.

(1) acquire write lock
(2) check if split
(3) read lock
(4) delete

Implementing deletions in this way would impact the other methods in some ways. During a query, after acquiring the read-lock when moving to the right (Block H in Algorithm 5.1.b), it would be possible to hit a new cluster start or even an empty slot before reaching the canonical slot. In this case the previous read-lock is unlocked and the operation is restarted. During an insertion after acquiring the write-lock (before Block C in Algorithm 5.1.a) it is necessary to scan to the left to check that no element was removed after the canonical slot (comparable to Block III Algorithm 5.2).

necessary changes to query and insert

5.6 Fully Expandable QFs

The goal of this fully expandable quotient filter is to offer a resizable quotient filter variant with a bounded false positive rate that works well even if there is no known bound to the number of elements inserted. Adding new fingerprint bits to existing entries is impossible without access to the inserted elements. We adapt a technique that was originally introduced for scalable Bloom filters [3]. Once a quo-

bounded false positive rate for unbounded elements

Algorithm 5.2 Concurrent locally locked quotient filter *delete* operation.

```
$(quot, rem) \leftarrow f(key)$
// Block I: try a trivial deletion (if there is another empty slot in group)
$group \leftarrow$ atomically load data around $quot$
if deletion into $group$ is trivial then
    finish deletion with a CAS and return
// Block II: write-lock the supercluster
scan right from $it \leftarrow quot$
    if $it$ is write-locked then
        wait until released and continue
    if $it$ is empty then
        trylock $it$ with write-lock and break
            if lock unsuccessful re-examine $it$
// Block IIa: scan to the left to ensure there are no holes from other deletions
scan left from $it$ until $quot$
    // ignore read-locks
    if $it$ is empty or write-lock then
        unlock the write-lock
        restart the operation
// later there can be no holes because any deletion waits for our lock
// Block III: read-lock the cluster
scan left from $it \leftarrow pos($write-lock$)$
    if $it$ is read-locked then
        wait until released and retry this slot
    if $it$ is cluster start then
        trylock $it$ with read-lock and break
            if lock unsuccessful re-examine $it$
// Block IV: find the correct run
$occ = 0; \quad run = 0$
scan right from $it$
    if $it$ is occupied and $it < quot$ then $occ$++
    if $it$ is run start then $run$++
    if $occ = run$ and $it \geq quot$ then break
// Block V: delete from this run and shift elements left
scan right from $it$
    if $it = rem$ then break
    if $it$ is cluster start or read-locked or write-locked then
        // not in the table
        unlock both locks
        return false
// Block VI: shift elements to the left until the next cluster start
// new cluster starts are created with read-locks
scan right from $it$
    // this loop can be done atomically to each group (keep groups consistent)
    $next \leftarrow read(it + 1)$
    if $next$ is cluster start or read-lock or write-lock then
        $it \leftarrow$ clear
        break
    $next \leftarrow$ precompute shifted status bits
    if $next$ becomes a cluster start then
        $next \leftarrow$ status bits for read-lock
    $it \leftarrow next$
unlock all the locks // also the newly created ones
return true
```

tient filter is sufficiently full, we allocate a new *level* to the data structure, each new level is an *additional* quotient filter. Each subsequent level increases the fingerprint size. Overall, this ensures a bounded false positive rate. This old idea offers new and interesting possibilities when applied to concurrent quotient filters, such that avoiding locks on lower levels, growing each level using the limited growing technique, higher fill degree through cascading inserts, and early rejection of queries also through cascading inserts.

level = additional filter with longer fingerprints

The fully expandable quotient filter starts out with one quotient filter, but over time, it may contain multiple levels each consisting of one quotient filters. At any point in time, only the newest (highest) level is active. Insertions operate on the active level. The data structure is initialized with two user-defined parameters the *initial capacity c* and the upper *bound for the false positive rate* $\overline{p}^+$. The first level table is initialized with m_0 slots where $m_0 = 2^{q_0}$ is the first power of 2 where $\delta_{grow} \cdot m_0$ is larger than c, where δ_{grow} is the fill ratio where growing is triggered and $\overline{n_i} = \delta_{grow} \cdot m_i$ is the maximum number of elements level i can hold. The number of remainder bits r_0 is chosen such that $\overline{p}^+ > 2\delta_{grow} \cdot 2^{-r_0}$ ($k_0 = q_0 + r_0$ fingerprint bits).

initial capacity c
bounded false positive rate $\overline{p}^+$

at most $\overline{n_i}$ *elements on level i*

Queries have to check each level. Within the lower levels queries do not need any locks because the elements there are finalized. Query performance depends on the number of levels. To keep the number of levels small, we have to increase the capacity of each subsequent level. To also bound the false positive rate, we have to reduce the false positive rate of each subsequent level. We achieve both of these goals by increasing the size of the fingerprint k_i by two for each subsequent level ($k_i = 2 + k_{i-1}$). Using the longer fingerprint, we can ensure that once the new table holds twice as many elements as the old one ($\overline{n_i} = \overline{n_{i+1}}/2$), it still has half the false positive rate ($\overline{p}_i^+ = \overline{n_i} \cdot 2^{-k_i} = 2\overline{p}_{i+1}^+ = 2 \cdot \overline{n_{i+1}} \cdot 2^{-k_{i+1}}$).

no locks on lower levels

two additional bits per level

When a level reaches its maximum capacity $\overline{n_i}$, we allocate a new level. However, instead of allocating the new level to immediately have twice the number of slots as the old level, we allocate it with one 8th of the final size (1/4 of the current level), and use the limited growing algorithm (described in Section 5.4.1) to grow it to its final size (over time with three growing steps). This way, the table has a higher average fill rate (at least $2/3 \cdot \delta_{grow}$ instead of $1/3 \cdot \delta_{grow}$).

new table starts at 1/8th of its final size

Theorem 1 (Bounded p^+ in expandable QF)**.** *The fully expandable quotient filter maintains the false positive probability $\overline{p}^+$ set by the user independently of the number of inserted elements.*

Proof. For the following analysis, we assume that fingerprints can potentially have an arbitrary length. The analysis of the overall false positive rate p^+ is very similar to that of the scalable Bloom filter. A false positive occurs if one of the ℓ levels has a false positive $p^+ = 1 - \prod_i (1 - p_i^+)$. This can be approximated with the Weierstrass inequality $p^+ \leq \sum_{i=1}^{\ell} p_i^+$. When we substitute the shrinking false positive rates per level ($p_{i+1}^+ = p_i^+/2$), we obtain a geometric sum which is bounded by $2p_1^+$: $\sum_{i=0}^{\ell-1} p_i^+ 2^{-i} \leq 2p_1^+ < \overline{p}^+$. □

$O(\log n/c)$ levels

Using this growing scheme the number of filters is in $O(\log n/c)$. Therefore, the bounds for queries are similar to those in a broad tree data structure. However, due to the necessary pointers, tree data structures take significantly more memory. Additionally, they are difficult to implement concurrently without creating contention on the root node.

Cascading Insert

insert remainder into lower levels if canonical slot is free

Cascading insertions can be used to improve memory usage and query performance of growing quotient filters. The idea is to insert elements on the lowest possible level. If the canonical slot on a lower level is empty, we insert the element into that level. This can be done using a simple compare-and-swap operation (without acquiring a write-lock). Queries on lower levels can still proceed without locking because insertions cannot move existing elements.

cascading inserts improve fill degree of lower levels up to $\delta = 100\,\%$

The main reason to grow the table before it is full is to improve the performance by shortening clusters. The trade-off for this is space utilization. For the optimal space utilization it would be necessary to fill each table to 100 %, i.e., $\delta_{grow} = 1$. Using cascading inserts, this can be achieved while still maintaining a good performance on each level. Queries on lower levels have no significant slowdown due to cascading inserts because the average cluster length remains small (cascading inserts lead to one-element clusters). Additionally, if we use cascading inserts, we can abort queries that encounter an empty canonical slot in one of the lower level tables because this slot would have been filled by an insertion. We call this *early query termination*. Yet cascading inserts also cause some overhead. Each insertion has to check every level whether its canonical slot is empty. This was already mandatory in some applications where elements were not allowed to be inserted multiple times, i.e., a full query on every level was already necessary, for example in applications like approximate element unification to prevent repeated insertions of one element. Such applications often use combined query and insert operations that only insert if the element was not yet in the table. In these cases, cascading inserts do not cause any additional overhead.

early query termination

5.7 Experiments

Throughout this section we compare a number of different AMQ-filters. We test multiple variants described throughout this chapter like our LPQ-filter and the locally locking quotient filter (see Section 5.5) as well as some state of the art data structures like bloom filter and counting quotient filter. Furthermore, we test the set of growing quotient filters described throughout this chapter, however, we did not find any state of the art implementations of other growing AMQ-filters available anywhere else. Each test was repeated 9 times using 3 different sequences of keys—3 runs per sequence. Similar to the experiments in the previous chapter ()

compare our variants with state of the art

3 repetitions × 3 seeds

The Non-Growing Competitors

- ● linear probing quotient filter (LPQ-filter presented in Section 5.5.1). This is our lock free quotient filter variant that does not use any reordering. It does not use any status bits, instead it always uses three additional remainder bits.
- 🟩 locally locked quotient filter (presented in Section 5.5.2). This is the variant that uses status bit combinations to locally lock specific clusters. This quotient filter is also the base implementation used for all of our growing quotient filter variants compared in the growing test.
- ◆ externally locked quotient filter. This variant is based on the same sequential code as our locally locked quotient filter but instead of using the novel in-

table-locking approach we use an external array of locks, i.e., one lock per 4096 slots.

▲ counting quotient filter—implementation by Pandey et al. [70] found at [69]. This implementation has a similar locking approach as our externally locked quotient filter. In addition to our implementation, it does support delete operations and improved duplicate insertions (using variable sized counters) but it does not support growing.

▼ classic bloom filter implementation. This is a bloom filter implementation using simple atomic operations. This simple implementation uses less memory than the comparable quotient filters.

▼ optimized bloom filter. For this variant of the bloom filter we tweaked the parameters for the table size and number of hash functions to offer a fair comparison to the quotient filter data structures. This version uses the same amount of memory as the quotient filters. It also uses fewer hash functions to keep a similar bound on the false positive rate but a reduced running time both for insertions and for find operations (5 hash functions).

two machines:
- 4-socket-intel
- amd-epyc

Hardware All experiments were executed on two machines (1) a four-socket Intel Xeon Gold 6138 machine with 20 cores per socket, each running at 2.0 GHz (3.7 GHz Turbo Frequency) with 27.5 MB of L3 cache size (per socket) and 768 GB of main memory (overall). (2) a one-socket machine with an AMD EPYC 7702P with 64 cores each running at 2.0 GHz (3.35 GHz Turbo Frequency) with 256 MB of L3 cache size and 1 TB of main memory. Both machines use a Ubuntu 20.04.2 operating system and all tests were compiled using gcc 9.3.0 with `-march=native` and `-O3` flags.

Fill Ratio Benchmark

test performance at varying fill degrees constant p

In this benchmark, we test all filter implementations under a varying fill degree. To do this we initialize a table with $2^{26} \approx 67.1$ M ≈ slots and $r = 10$ remainder bits (LPQ-filter ● has 13 remainder bits). This table is filled with uniform random elements concurrently by all cores of the machine ($p_i = 80$ and $p_a = 64$ respectively).

Every 10 % of the fill ratio we execute a performance test. Each such test consists of 100000 positive queries, negative queries, and insertions. These operations are split among all processors and are executed concurrently after a synchronization. Each positive query is looking for a random previously inserted element (not necessarily the last inserted elements). Negative queries query a new random element, these may include false positives. Any necessary random numbers are computed prior to executing the benchmark. The results are shown in Figure 5.3.

increasing fill degree → decreasing performance

As one would expect, the throughput of most quotient filter variants decreases with increasing fill ratio. However, this seems not to be the case for the counting quotient filter and the Bloom filter. Our advanced quotient filter implementations—the LPQ-filter ● and the locally locked quotient filter ■—display their strengths on sparser tables with about × and 2.2 higher insertion throughputs than the similarly implemented externally locked quotient filter ◆ at 30 % fill degree (*amd*: 2.7 and 1.6). At 70 % fill degree the advanced implementation are still 2.9 × and 1.6 × faster than the externally locked implementation (*amd*: 2.2 × and 1 ×).

find performance

Our LPQ-filter ● is by far the fastest AMQ for positive queries. The table likely profits from the same effect we have already seen in the previous chapters (see Section 2.5.1 and Section 3.8), i.e., the fact that average query times are fast when many elements have a small displacement—a few elements with large displacements are a lot less impactful. The other quotient filter variants behave similar to Robin Hood hashing in that many elements have medium to large displacements. Both Bloom filters ▼, ▼ that we used in these experiments are especially good at negative queries because the density of 1s is actually very small (for most fill degrees), therefore, queries can often be aborted after one or two memory accesses.

The insert performance of the counting quotient filter ▲ is at best 1.7 MOps/s on intel (3.3 MOps/s on amd), however, its find performance is at about the same as the externally locked variants ◆—on our machine amd-epyc–and significatly better on the intel machine, e.g., by a factor of 1.8 on positive finds at 70 % fill (factor of 2.1 on negative finds).

fp rate

All quotient filter variants (except for the LPQ-filter ●) have the same false positive rate—which is determined by the fingerprint function. The false positive rate of the LPQ-filter ● and the optimized bloom filter ▼ both start out smaller than the normal quotient filter rate but they become bigger on fill degrees over 70 %. The classic bloom ▼ filter starts out very low but the false positive rate begins to pick up with the fill degree. At 100 % fill degree, it should have the same false positive rate as the quotient filters.

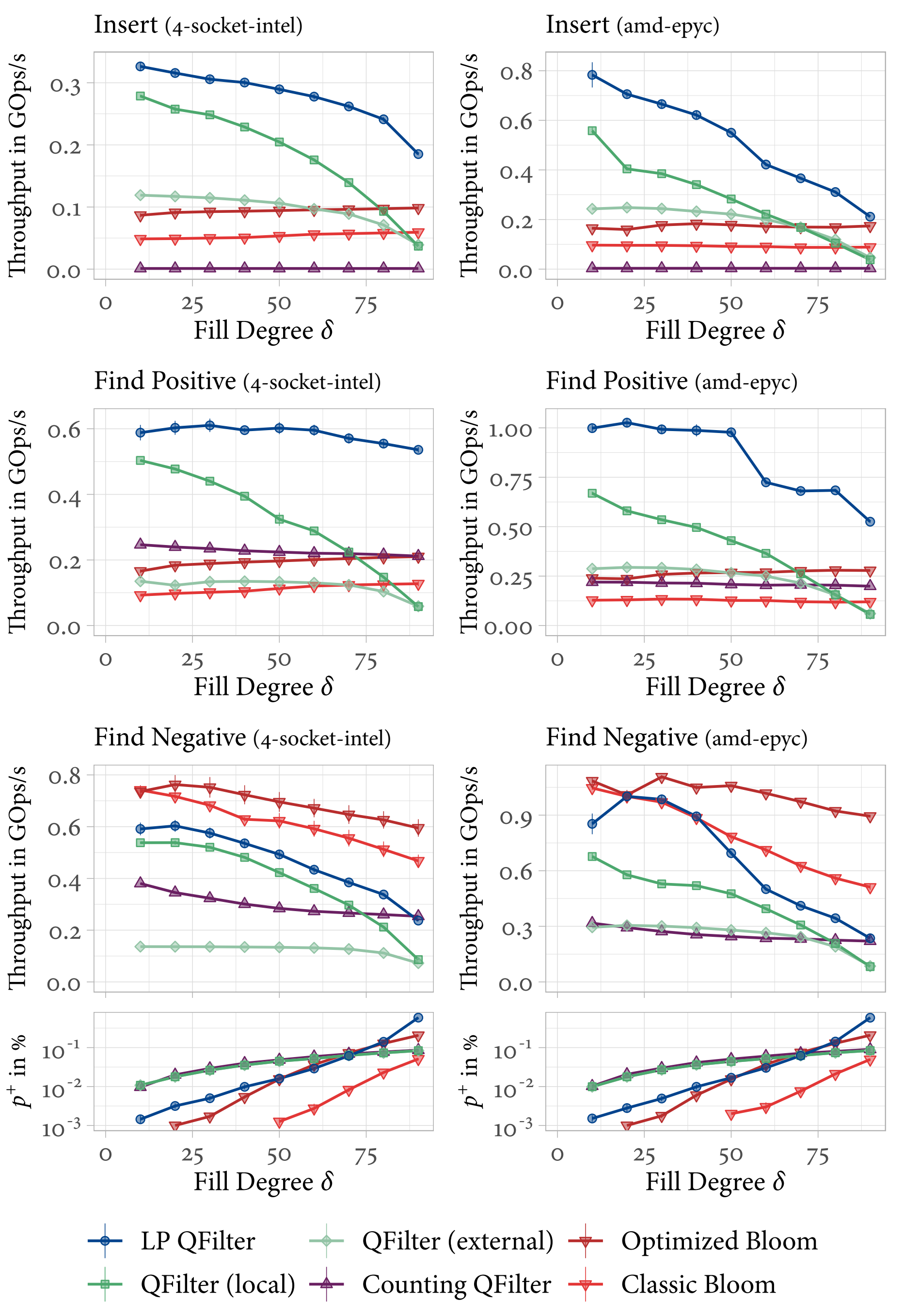

Figure 5.3: Throughput over fill degree. At specific fill degrees we test the performance of the quotient filter operations, i.e., insertions (*top*), positive finds (*scnd*), and negative finds (*thrd*). Additionally we also measured false positives (*bottom*). All measurements were executed on both described machines (i.e., 4-socket-intel and amd-epyc at p = 80 and 64 respectively).

Speedup of Table Construction

construct table—70% fill, varying p

Our first test (presented in Figure 5.4) examines the scaling behavior of all implementations under a varying number of operating threads p. We prepare a table with a capacity m to hold M elements. Then we fill this table with 46 M uniformly random elements through repeated insertions (by all p threads concurrently). Filling the table to about 70 % using $r = 10$ remainder bits (13 for the LPQ-filter ●).

linear scaling

We can see that all data structures scale close to linearly with the number of processors. There is only a small bend (on the 4-socket intel machine) when the number of cores exceeds the first socket ($p = 20$) and once it reaches the number of physical cores (highlighted at $p = 80$ and $p = 64$ respectively). However, the data structures even seem to scale relatively well when using hyperthreading. The absolute performance is quite different between the data structures. LPQ-filter ● has the best throughput.

speedup over the respective sequential versions is comparable

As it was the case in the previous test the LPQ-filter ● has the best performance among all tested filters. It has an absolute speedup of 19.7 (relative to the sequential LPQ-filter with $p = 80$ on *intel*; $p = 64$ on *amd* 38.7). Compared to the sequential quotient filter the speedups are even larger, i.e., 32.2 and 64.1 respectively. Even though the locally locked quotient filter 🟩 uses locking. It has similar speedups when compared to the (non-LP) sequential quotient filter, of 21.9 (with $p = 80$ on *intel*; $p = 64$ on *amd* 39.3).

Overall we see that the speedups and also the overall performance are better on the *amd* machine, even though it has less cores at about the same frequency. However, it has a significantly better memory connection and an about 10 times larger L3 cache size it. Additionally, it does not have any NUMA effects that are negatively impacting the multi-socket intel machine. Compared to our results on concurrent hash tables presented in Section 4.6, we see that quotient filters scale better on 4-socket-intel, i.e., higher speedup. The main reason for this is probably because the actual amount of loaded data is significantly lower due to the packing of multiple slots into one 64 bit read. This points to the fact that memory bandwidth might be an issue at least for the traditional hash table implementations.

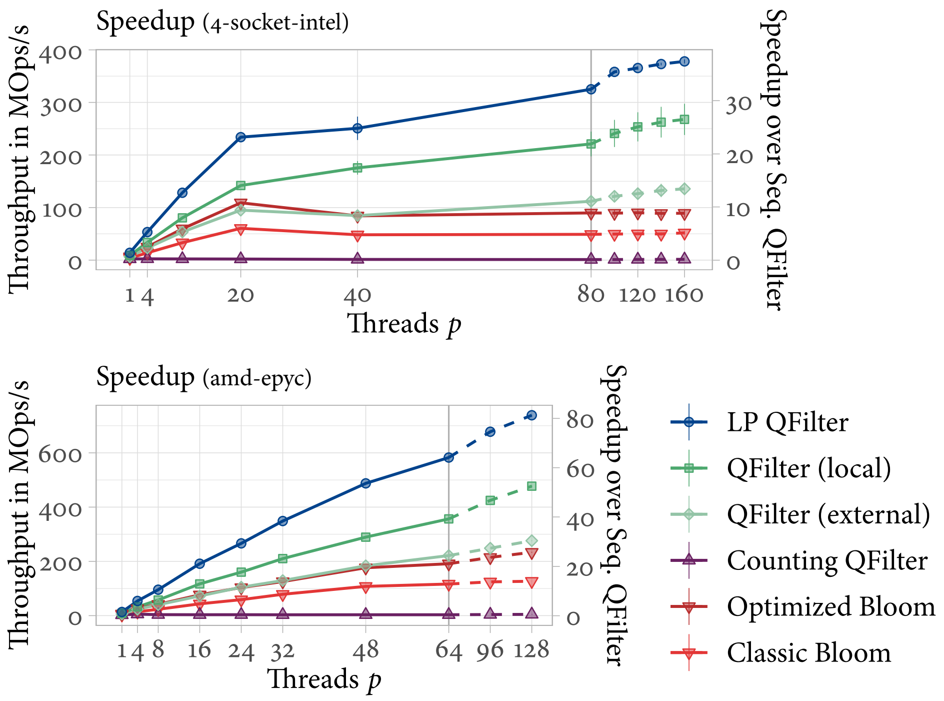

Figure 5.4: Speedup constructing a table. Strong scaling measuremen inserting 47 M elements into a table with $2^{26} \approx 67$ M slots (70 % fill degree). The speedup is measured relative to a sequential quotient filter (non-LP version) (right scale).

Growing Benchmark

In this experiment we compare the performance of the different growable quotient filter implementations, the limited growing quotient filter ● ●, the expandable growing quotient filter ■, and the expandable growing quotient filter with cascading inserts ▲. We show two test series using the limited growing quotient filter, one starting at remainder length 10 ● (i.e., false positive rate of 2^{-10} in the first table) and one starting at remainder length 17 ● (can grow by a factor of ≈ 128 and still have a false positive rate of 2^{-10}). The limited growing quotient filter grows by repeatedly migrating the table—each time reducing the fingerprint by one bit. The expandable quotient filter ■ uses the multi level growing approach we adapted to quotient filters (see Section 5.6). It keeps a bounded false positive rate of 2^{-10}. Each level of the data structure grows 3 times before it reaches its final size. The expandable quotient filter with cascading inserts ▲ keeps the same false positive rate but uses the cascading insert technique described on page 162 to pack additional elements into the lower levels of the data structure.

For this benchmark we insert 50 M uniform random elements into each of our fully expandable quotient filters. Each filter was initialized with a capacity of only $2^{19} \approx 0.5$ M (95 × growing factor) slots and a target false positive rate of $\overline{p}^{+} = 2^{-10}$. The inserted elements are split into 50 segments of M elements. We measure the running time of inserting each segment as well as the query performance after each segment (similar to the fill benchmark). The results of this experiment are shown in Figure 5.5.

As expected, the false positive rates of the quotient filters with limited growing ● ● increase linearly with the number of elements. The query performance only depends on the fill degree of the current table—not the actual number of elements. The table that was initialized with a higher precision ● has a worse overall performance, but has the same "final" false positive rate as the variants with bounded growing (inserting more elements would cause the rate to diverge).

Both fully expandable variants ■ ▲ stay below 2^{-10} false positive rate. But their query performance suffers due to the lower level look ups. Cascading inserts can improve positive query times by 10 % (average over all queries; 10 % on *amd*), however, for unsuccessful queries we do not see the same significant speedups 2 % and 7 % respectively (on *intel* and *amd* respectively). The reason for this is that after some time there are likely very few empty slots in lower level tables, therefore, early query rejection does not do much good (aborting a query after in a lower level table). Finding an element on a lower level seems to be beneficial, especially since elements inserted on a lower level (through a cascading inserto) are always in their canonical slot. As expected cascading inserts are slowing down insertion times significantly. A lot of workloads could still profit from cascading inserts because AMQ-filter data structures are usually queried a lot more often than they are updated.

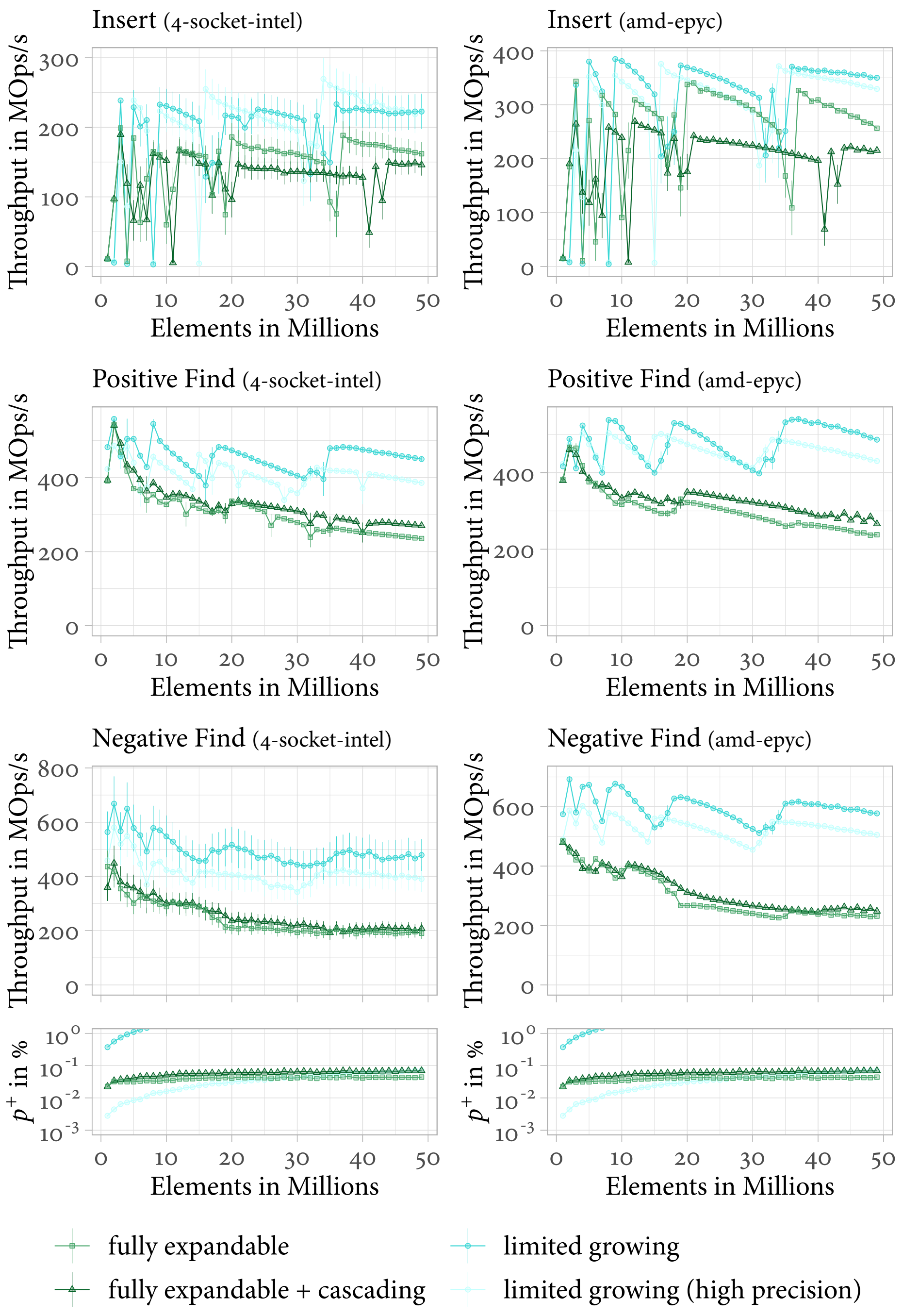

Figure 5.5: Throughput of the growing quotient filter variants (all variants based on the local locking quotient filter). All threads cooperatively insert 50 segments of 20 M elements each into a table initialized with $2^{24} \approx 16.8$ M slots (executed on 4-socket-intel and amd-epyc at $p = 80$ and 64 respectively).

5.8 Chapter Conclusion

- lp quotient filter
- local locking quotient filter
- expandable quotient filter

Throughout this chapter we have described a number of quotient filter variants that add to the varied list of AMQ-filters. Some of our variants are targeted for specific use cases, like the LPQ-filter that is fast to build but only works until certain fill degrees or the expandable growing filter variants that open the design space for unbounded growing quotient filters. Other techniques are targeted at optimizing already available variants like the locally locked quotient filter that improves the performance of concurrent quotient filters and serves as a building block for the expandable variants.

local locking with status bits

We proposed a new technique for concurrent quotient filters that uses the status bits inherent to quotient filters for localized locking. Using this technique, no additional cache lines are accessed (compared to a sequential quotient filter). This variant achieves a 1.6 times increase in insert performance over the external locking scheme ($p = 80$ and 70 % fill degree; 1.8 and 2.3 on queries). Additionally, we proposed a simple linear probing based filter that does not use any status bits and is lock-free. Using the same amount of memory this filter achieves even better false positive rates up to a fill degree of 70 % and also 1.9 times higher insertion speedups (than the local locking variant also at 70 % fill degree).

lock free linear probing quotient filter

expandable
- bounded fill degree
- bounded fp rate

We also designed the fully expandable growing quotient filter which uses the limited growing technique to refine the growing scheme used in scalable Bloom filters. This combination of techniques guarantees that the overall data structure is always at least $2/3 \cdot \delta_{grow}$ filled (where δ_{grow} is the fill degree where the migration is triggered). Using cascading inserts this can even be improved by filling lower level tables even further, while also improving successful query times by around 10 % (on *amd*).

optimal table still depends on workload

Our tests show that there is no optimal AMQ data structure. Which data structure performs best depends on the use case and the expected workload. The linear probing quotient filter is very good as long as the table is not densely filled and as long as positive queries make up a lot of the workload. Similarly, Bloom filters perform well on workloads with a lot of negative queries. The locally locked quotient filter is also efficient on tables below a fill degree of 70 %. But, it is also more flexible for example when the table starts out empty and is filled to above 70 % (i.e., constructing the filter). Our growing implementations work well if the number of inserted elements is not known prior to the table's construction. The counting quotient filter could perform well on very query heavy workloads that operate on densely filled tables.

6 Discussion

Throughout this dissertation we have looked at scaling hash tables with scalable migration algorithms. We have identified three domains where growing hash-based data structures are important and where previous approaches to growing have failed: (1) *Space efficient hash tables* because only growing hash tables can be space efficient if the table size is not known before the construction of the hash table and because there was no previous work showing space efficient growing techniques. (2) *Concurrent hash tables* because data structures that can achieve significant speedups are essential for writing efficient algorithms and because scalability is literally in the name of this dissertation. And (3) *concurrent quotient filters* because quotient filters have the chance to be faster than bloom filters but are hard to implement concurrently and because AMQ-filter with an adaptable capacity are rare and potentially very interesting for many applications.

space efficient hash tables

concurrent hash tables

concurrent quotient filters

Space Efficient Hashing We have defined the concept of dynamic α-space efficient hash tables. The main idea behind the concept is that there always has to be a tight bound to the memory used by the data structure ($\alpha \cdot n \cdot s$ where s is the size of an element). Using this definition, traditional growing techniques (i.e., reallocate and migrate) can be at most 2-space efficient because during the migration at least 2 slots exist for every element.

dynamic α-space efficient hashing

We have developed an in-place growing approach to construct dynamically α-space efficient hash tables and have applied this approach to many common hashing techniques (i.e., linear probing, quadratic probing, Robin Hood hashing, and cuckoo hashing). In-place growing works by allocating additional memory at the end of the existing table and reordering the elements in-place.

in-place growing can be applied to many hashing techniques

We also developed a hash table that is specifically designed to be dynamically α-space efficient—DySECT. DySECT is a hash table architecture that consists of many subtables, thus using a traditional migration algorithm for one subtable does not violate the overall space bound. Through a cuckoo displacement technique, elements can be moved between subtables, thus making newly created memory in one subtable accessible to new elements.

Both techniques perform really well in our experiments with insertions scaling as expected up to loads of 95 % and higher. Among the tables using the in-place growing method, quadratic probing emerged as the strongest competitor, however find operations are slow on all variants that use traditional probing (non-cuckoo methods). DySECT offers a great balance between efficient insert and find operations.

Concurrent Hash Table The motivation behind looking at concurrent hash tables was taking the simple folkloreHT implementation and expanding it in a way that it can be used in a wider variety of circumstances without compromising the raw speed of the simple base implementation. We have implemented a variety of generalizations for folkloreHT that can all be enabled or disabled at compile time. Thus, adapting the functionality of the resulting table without any overheads in cases where a specific extension was not chosen.

Using our generalizations we successfully remove all constraints that we identified in folkloreHT: *dynamic table size*, *deletions*, and *arbitrary key and value types*. Additionally, our generalizations can be freely combined to adapt the resulting hash table to its specific intended work load.

Throughout our extensive experimental evaluation with up to 17 different hash tables from 7 different libraries, our implementations consistently outperformed all state of the art implementations. This is true even for tests where multiple of our extensions were combined, e.g., the word count benchmark on page 132, or the deletion benchmark on page 134.

solutions with and without growing

Concurrent Quotient Filters We describe multiple variants of concurrent AMQ-Filters both non-growable (i.e., LPQ-filter, locally locked quotient filter) and growable (either using the limited growing technique, or fully expandable). We have shown that there cannot be a "simple" lock-free implementation for status-bit-based quotient filter (page 151). This makes our LPQ-filter the only available lock-free solution.

LPQ-filter is lock-free

Both our lock-free and locked solutions scale well with the number of processors and perform especially well on tables with lower fill degrees (page 167). The LPQ-filter has by far the best performance on most work loads and it even has a lower false positive rate for fill degrees under 70 %.

Our growable quotient filters come in two flavors: the limited growing technique that is inherent to quotient filters but has an unbounded false positive rate, and the fully expandable quotient filter that has a bounded false positive rate, albeit at non-constant operation times. In our experiments the fully expandable technique was within a factor of two of the limited growing technique, even when growing by a factor of 100 (page 171).

limited and fully expandable variants

Linear Mapping and Migrations One of the most important takeaways from this dissertation is the influence of the linear table mapping on hash table migrations. Because we use a linear table mapping, elements are stored in an order that is correlated to their preliminary hash value. Throughout this dissertation we use this knowledge in two ways:

because of linear table mapping

First, to construct our space efficient hash tables we use the fact that the order of elements in the source and target tables do not change significantly. Thus scanning through the elements of the source table during the migration and moving them to the target table accesses the slots of the target table in a very predictable order. This is important because it ensures that the elements move behind the scan line during the in-place migration. It is also the reason for the cache efficiency of all our migration algorithms.

order of elements in source table ≈ order in target table

Second, if we know how many elements are hashed into a certain range in the source table (e.g., in a cluster) we know there cannot be more elements hashed into the corresponding range in the target table. Because the target table has at least the same number of elements (concurrent hash tables with deletions), we can guarantee that certain slots remain empty. We use this in our concurrent migrations to reduce interactions between threads (by implicitly moving the border of blocks to free slots).

given a range (source) and its corresponding range (target) elements move from source to target *flexible, interchangeable, top performing*

Concluding In combination the solutions we developed throughout this thesis can be used in a wide variety of situations. Moreover, throughout all experiments against state of the art competitors, our implementations consistently rank among the top performing data structures. And since all of our tables have been developed with interchangeability in mind users can quickly change between different implementations to find the correct data structure for their specific use case and workload.

Bibliography

[1] Lada A. Adamic and Bernardo A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.

[2] Mohammad Al-Hisnawi and Mahmood Ahmadi. Deep packet inspection using quotient filter. *IEEE Commun. Lett.*, 20(11):2217–2220, 2016. doi:10.1109/LCOMM.2016.2601898.

[3] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom Filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007. ISSN 0020-0190. doi:10.1016/j.ipl.2006.10.007.

[4] Austin Appleby. SMHasher. `https://github.com/aappleby/smhasher`. Accessed April 25, 2021.

[5] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Automata, Languages and Programming, 36th International Colloquium (ICALP)*, volume 5555 of *LNCS*, pages 107–118. Springer, 2009. doi:10.1007/978-3-642-02927-1_11.

[6] Robert Axtell. Zipf distribution of u.s. firm sizes. *Science (New York, N.Y.)*, 293:1818–20, 10 2001. doi:10.1126/science.1062081.

[7] Hannah Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007. doi:10.1137/1.9781611972870.5.

[8] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep

Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012. doi:10.14778/2350229.2350275. URL http://vldb.org/pvldb/vol5/p1627_michaelabender_vldb2012.pdf.

[9] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: evidence and implications. In *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 1, pages 126–134 vol.1, Mar 1999. doi:10.1109/INFCOM.1999.749260.

[10] Alex D. Breslow and Nuwan Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *VLDB J.*, 29(2-3):731–754, 2020. doi:10.1007/s00778-019-00561-0.

[11] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.

[12] Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing (preliminary report). In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 281–288. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.48.

[13] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007. doi:10.1145/1272743.1272747.

[14] John G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984. doi:10.1109/TC.1984.1676499.

[15] Yan Collet. xxHash. https://github.com/Cyan4973/xxHash. Accessed April 25, 2021.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

[17] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit key. In *"6th Workshop on Algorithm Engineering & Experiments (ALENEX)"*, pages 142–151. SIAM, 2004.

[18] Luc Devroye and Pat Morin. Cuckoo hashing: Further analysis. *Inf. Process. Lett.*, 86(4):215–219, 2003. doi:10.1016/S0020-0190(02)00500-8.

[19] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.*, 380(1-2): 47–68, 2007. doi:10.1016/j.tcs.2007.02.054.

[20] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

[21] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997. doi:10.1006/jagm.1997.0873.

[22] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Automata, Languages and Programming, 37th International Colloquium (ICALP)*, volume 6198 of *LNCS*, pages 213–225. Springer, 2010. doi:10.1007/978-3-642-14165-2_19.

[23] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In *19th Annual European Symposium on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 615–627. Springer, 2011. doi:10.1007/978-3-642-23719-5_52.

[24] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (CoNEXT)*, pages 75–88. ACM, 2014. doi:10.1145/2674005.2674994.

[25] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/s00224-004-1195-x.

[26] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM J. Comput.*, 42(6):2156–2181, 2013. doi:10.1137/100797503.

[27] Alan M. Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM J. Comput.*, 40(2):291–308, 2011. doi:10.1137/090770928.

[28] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Comput.*, 18(1):21–42, 2005. doi:10.1007/s00446-004-0115-2.

[29] Afton Geil, Martin Farach-Colton, and John D. Owens. Quotient filters: Approximate membership queries on the GPU. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–462. IEEE Computer Society, 2018. doi:10.1109/IPDPS.2018.00055.

[30] Jan Friso Groote, Wim H. Hesselink, Sjouke Mauw, and Rogier Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Comput.*, 14(2):75–81, 2001. doi:10.1007/PL00008930.

[31] Torben Hagerup and Christine Rüb. Optimal merging and sorting on the erew pram. *Inf. Process. Lett.*, 33(4):181–185, 1989. doi:10.1016/0020-0190(89)90138-5.

[32] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012. ISBN 9780123973375.

[33] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Distributed Computing, 22nd International Symposium (DISC)*, volume 5218 of *LNCS*, pages 350–364. Springer, 2008. doi:10.1007/978-3-540-87779-0_24.

[34] Euihyeok Kim and Min-Soo Kim. Performance analysis of cache-conscious hashing techniques for multi-core CPUs. *International Journal of Control & Automation (IJCA)*, 6(2), 2013. ISSN 2005-4297.

[35] Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *45th Annual Allerton Conference on Communication, Control, and Computing*, volume 75, 2007. URL https://www.eecs.harvard.edu/~michaelm/postscripts/aller2007.pdf.

[36] Donald E. Knuth. Notes on "open" addressing. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4899, 1963. Unpublished memorandum.

[37] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. ISBN 0201896850. URL https://www.worldcat.org/oclc/312994415.

[38] Doug Lea. Hash table util. concurrent. concurrenthashmap, revision 1.3.4. *JSR-166, the proposed Java Concurrency Package. http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html*, 2004.

[39] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1), January 2019. ISSN 1049-3301. doi:10.1145/3230636.

[40] Lexico. scalability. https://www.lexico.com/en/definition/scalability, Accessed May 20, 2021.

[41] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Ninth Eurosys Conference, (EuroSys)*, pages 27:1–27:14. ACM, 2014. doi:10.1145/2592798.2592820.

[42] Tobias Maier. Dysect. https://github.com/TooBiased/DySECT, . Accessed May 25, 2021.

[43] Tobias Maier. Growt. https://github.com/TooBiased/growt, . Accessed May 25, 2021.

[44] Tobias Maier. lpqfilter. https://github.com/TooBiased/lpqfilter, . Accessed May 25, 2021.

[45] Tobias Maier and Peter Sanders. Dynamic space efficient hashing. In *25th Annual European Symposium on Algorithms (ESA)*, volume 87 of *LIPIcs*, pages 58:1–58:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.58.

[46] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general?(!). *CoRR*, abs/1601.04017, 2016. URL http://arxiv.org/abs/1601.04017.

[47] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general(?)! In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, PPoPP, pages 34:1–34:2, 2016. ISBN 978-1-4503-4092-2. doi:10.1145/2851141.2851188.

[48] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, February 2019. ISSN 2329-4949. doi:10.1145/3309206.

[49] Tobias Maier, Peter Sanders, and Stefan Walzer. Dynamic space efficient hashing. *Algorithmica*, 81(8):3162–3185, 2019. doi:10.1007/s00453-019-00572-x.

[50] Tobias Maier, Peter Sanders, and Robert Williger. Concurrent expandable AMQs on the basis of quotient filters. In *18th International Symposium on Experimental Algorithms (SEA)*, volume 160 of *LIPIcs*, pages 15:1–15:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SEA.2020.15.

[51] Grzegorz Malewicz. A work-optimal deterministic algorithm for the certified write-all problem with a nontrivial number of asynchronous processors. *SIAM J. Comput.*, 34(4):993–1024, 2005. doi:10.1137/S0097539703428014.

[52] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998. doi:10.1145/272991.272995.

[53] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

[54] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. *Parallel and Distributed Computing and Systems*, pages 509–518, 1998. URL `http://www.rdrop.com/users/paulmck/paper/rclockpdcsproof.pdf`.

[55] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008. ISBN 978-3-540-77977-3. doi:10.1007/978-3-540-77978-0.

[56] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. O'Reilly, 2005. ISBN 0321334876.

[57] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distributed Syst.*, 15(6):491–504, 2004. doi:10.1109/TPDS.2004.8.

[58] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distributed Syst.*, 12(10):1094–1104, 2001. doi:10.1109/71.963420.

[59] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *17th Annual European Symposium on Algorithms (ESA)*, volume 5757 of *LNCS*, pages 1–10. Springer, 2009. doi:10.1007/978-3-642-04128-0_1.

[60] Michael Mitzenmacher, Konstantinos Panagiotou, and Stefan Walzer. Load Thresholds for Cuckoo Hashing with Double Hashing. In *16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, volume 101 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:9,

Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-068-2. doi:10.4230/LIPIcs.SWAT.2018.29. URL `http://drops.dagstuhl.de/opus/volltexte/2018/8855`.

[61] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018. doi:10.1137/1.9781611975055.4.

[62] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1123–1136. ACM, 2015. doi:10.1145/2723372.2747644.

[63] Nhan Nguyen and Philippas Tsigas. Lock-free cuckoo hashing. In *34th International Conference on Distributed Computing Systems Workshops (ICDCS)*, pages 627–636. IEEE Computer Society, 2014. doi:10.1109/ICDCS.2014.70.

[64] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398. USENIX Association, 2013. URL `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala`.

[65] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan Mcelroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani, and Facebook Inc. folly version 57:0. `https://github.com/facebook/folly`, Accessed August 13, 2020.

[66] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology - 23rd Annual International Cryptology Conference (CRYPTO)*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003. doi:10.1007/978-3-540-45146-4_36.

[67] Anna Pagh, Rasmus Pagh, and Milan Ruzic. Linear probing with constant independence. *SIAM J. Comput.*, 39(3):1107–1120, 2009. doi:10.1137/070702278.

[68] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

[69] Prashant Pandey. Counting Quotient Filter. `https://github.com/splatlab/cqf`. Accessed August 07, 2019.

[70] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 775–787. ACM, 2017. doi:10.1145/3035918.3035963.

[71] Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence-search index. In *Research in Computational Molecular Biology - 22nd Annual International Conference (RECOMB)*, volume 10812 of *LNCS*, pages 271–273. Springer, 2018. URL `https://link.springer.com/content/pdf/bbm%3A978-3-319-89929-9%2F1.pdf`.

[72] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186. ACM Press, 1995. doi:10.1145/223784.223813.

[73] Mihai Patrascu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):14:1–14:50, 2012. doi:10.1145/2220357.2220361.

[74] Mihai Patrascu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. *ACM Trans. Algorithms*, 12(1):8:1–8:27, 2016. doi:10.1145/2716317.

[75] Mathieu Desnoyers Paul E. McKenney and Lai Jiangshan. LWN: URCU-protected hash tables. http://lwn.net/Articles/573431/, 2013. URL `http://lwn.net/Articles/573431/`.

[76] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4): 298, April 2008. ISSN 1937-4771.

[77] Jeff Preshing. New concurrent hash maps for c++. `http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/`, 2016.

[78] Jeff Preshing. Junction. `https://github.com/preshing/junction`, Accessed April 27, 2021.

[79] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In Michael Luby, José D. P. Rolim, and Maria J. Serna, editors, *Randomization and Approximation Techniques in Computer Science, Second International Workshop (RANDOM)*, volume 1518 of *LNCS*, pages 159–170. Springer, 1998. doi:10.1007/3-540-49543-6_13.

[80] Peter Sanders. Hashing with linear probing and referential integrity. *CoRR*, abs/1808.04602, 2018. URL `http://arxiv.org/abs/1808.04602`.

[81] Julian Shun and Guy E. Blelloch. Phase-concurrent hash tables for determinism. In *26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107. ACM, 2014. doi:10.1145/2612669.2612687.

[82] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70. ACM, 2012. doi:10.1145/2312005.2312018.

[83] Alan Siegel. On the statistical dependencies of coalesced hashing and their implications for both full and limited independence. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 10–19. ACM/SIAM, 1995. URL `http://dl.acm.org/citation.cfm?id=313651.313657`.

[84] Alex D. Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel V. Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic

programming. *J. Parallel Distributed Comput.*, 70(8):839–848, 2010. doi:10.1016/j.jpdc.2010.01.004.

[85] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *Proceedings of the 33rd Conference on Design Automation*, pages 641–644. ACM Press, 1996. doi:10.1145/240518.240639.

[86] Reini Urban. SMHasher (fork). `https://github.com/rurban/smhasher`. Accessed April 25, 2021.

[87] Jeffrey Scott Vitter. Implementations for Coalesced Hashing. *Communications of the ACM*, 25(12):911–926, Dec 1982. ISSN 0001-0782. doi:10.1145/358728.358745.

[88] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981. doi:10.1016/0022-0000(81)90033-7.