Karlsruher Institut für Technologie

Bachelor Thesis

Using Per-Cell Data to accelerate Open Addressing Hashing Schemes

Jan Benedikt Schwarz

Submission date: 13.05.2019

Supervisor: Prof. Dr. rer. nat. Peter Sanders,
M. Sc. Tobias Maier

Institute of Theoretical Informatics, Algorithmics II
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, Datum

Zusammenfassung

Hash Tabellen werden fast überall verwendet, wobei von der Such Operation am meisten Gebrauch gemacht wird. Diese Arbeit beschäftigt sich mit der Speicherung von zusätzlichen Daten pro Zelle um die Performance von Open Addressing Hashing Schemes zu verbessern. Es werden Variationen von linear probing, robin hood hashing und hopscotch hashing vorgestellt, welche diese Daten benutzen um ihre Performance zu verbessern. Außerdem wird zwischen den Speichermöglichkeiten dieser Daten verglichen, diese können entweder innerhalb der Hash Tabelle oder außerhalb in einer eigenen Tabelle gespeichert werden. Einige Variationen haben sich in den Tests als sehr erfolgreich gezeigt, so war es zum Beispiel möglich erfolgreiche Suchanfragen, im Vergleich zu linear probing, um bis zu 80% zu verschnellern.

Abstract

Hash tables are used nearly everywhere, often for a find heavy workload. This thesis deals with the use of per-cell data to accelerate the performance of open addressing hashing schemes. It introduces variations of linear probing, robin hood hashing and hopscotch hashing which use this additional data to accelerate their performance. In addition, we compare how the additional data can be stored, which is either inside the hash table between its cells or outside of it in a second array. Some variations have shown to be quite competitive, one of them, for example, had up to 80% faster successful lookup times than linear probing.

Contents

List of Figures

1 Introduction

1.1 Overview

Hash tables have become one of the most used data structures in computer science. Their main use is to store key-value pairs and allow access to the pairs though their keys. To achieve fast lookup times the key is hashed and then used as the index where the element is stored. However a problem occurs once two elements are hashed to the same index, which is called a collision. Hash tables have lookup times of $\Theta(1)$ if they are mostly empty, but the collision resolution has a large impact on the performance the more the table is filled. Therefore it is important to find a trade-off between size and speed.
One of the most common practices for collision resolutions is open addressing, where elements with collisions are stored at another index of the array, which previously had no element. The most common technique used for open addressing, linear probing, stores elements in the first empty cell after the already filled one. Alternative hashing schemes like hopscotch and robin hood hashing often try to reposition other elements in the table for a more favorable position.
While hash tables are used for a variety of applications, the workload of the hash table varies and the used hash table should be adjusted to the workload of its application. Using a hashing schemes that optimizes the operation which an application uses the most can greatly increase the overall performance. For example, a word by word translator mostly has successful lookups, therefore a hash table with optimal successful lookup performance is desired even if the insert, delete or unsuccessful lookup operation had a worse performance. A duplicate detector on the other hand, which only inserts elements until the insertion is unsuccessful because an element is a duplicate and already in the table, should be tuned for fast insertions.
The contribution of this paper is twofold. First, it introduces variations of the known cache efficient algorithms linear probing, robin hood hashing and hopscotch hashing that use per-cell data to e.g. improve negative find performance of linear probing, reduce cache misses of robin hood hashing or reduce the additional space of hopscotch hashing. Second it introduces and compares different ways of storing per-cell data for each collision resolution method. The hashing schemes in this work are easily swappable so one can test which is best for a given application. Last, it will be elaborated where each hashing scheme would fit best.

1.2 Related Works

Linear Probing [8] stores all elements in a continuous array where each cell can store one element. Elements are stored at the first empty cell after the cell with the index of the elements hashed key. Linear probing is the most commonly used technique and using per-cell data can optimize it for specific aspects.

Robin Hood Hashing was introduced by Pedro Celis in 1985 [3] and is a variation of linear probing that stores all elements in a sorted order. This is used to reduce variance and improve the performance of unsuccessful lookups. In this work we will use the sorting to our advantage to allow faster lookups and deletions though the usage of per-cell data.

Coalesced Hashing [12] splits the table into two segments. The first segment is used as addressing region and has the same number of cells as the hash functions range. The second segment called cellar is solely used to store elements that caused collisions. This

hashing scheme already uses per-cell data for pointers to create a linked list for each group of elements with the same hash.

Hopscotch Hashing [6] is a hashing scheme where elements are stored in a constant sized neighborhood around the cell they are hashed to. A per-cell bitmask is used to reduce the number of cells that have to be looked at in the case of a contains operation. In this work we will try to reduce the bits needed for each cell to lower the space used and test different storage methods to improve the cache friendliness.

Chained Hashing [8] stores elements with the same hash in a dynamically allocated linked list. While its theoretical performance is great, it is not cache friendly because of its dynamic allocations and indirections. While we will not explore this method further because it demands dynamic allocation, other works [11] have shown that per-cell data can be used to accelerate chained hashing.

Cuckoo Hashing [9, 7] assigns elements to two cells and an element has to be stored in one of them. If both cells are filled the element is swapped with one of the elements the assigned cells. The element that is removed from its cell is then stored at the other cell it is assigned to. If this cell is also filled, the same occurs again for the element at that cell.

1.3 Outline

Chapter 2 provides an overview of a hash table that uses per-cell data (Section 2.1) and why using such data can be advantageous (Section 2.2). Lastly it describes how this data can be stored by using an additional array or increasing the space of the hash table and placing it between the cells of the table (Section 2.3).
Different hashing schemes that use per-cell data, as well as the hashing scheme they are based on, are discussed in Chapter 3. It starts with a description of open addressing and displacement (Section 3.1). Then it provides general information about linear probing and variations of it. These variations store if an element was hashed to a cell or create an upper bound for the maximum displacement an element can have (Section 3.2). Section 3.3 is about robin hood hashing and its variations which optimize the time it takes to find the first element that is hashed to the same cell as the key that is searched. The final Section 3.4 describes hopscotch hashing and how the per-cell data it needs could be reduced by allowing bits to address multiple cells or using one bit that indicates if there is an element outside the neighborhood.
In Chapter 4 the different hashing schemes are empirically evaluated using two different scenarios. In the first scenario (Section 4.3) we ignore the additional space that is allocated for the per-cell data and fill the hash table up to 60%. In the second scenario (Section 4.2) each hash tables allocated space is fixed, giving hash tables which use less per-cell data a higher capacity. The table is filled up to 90%, to simulate a situation where allocated space is a major concern. Future research targets for hashing schemes with per-cell data are discussed in Chapter 5 and the conclusion of this work is provided in Chapter 6.

2 Acceleration Data

In this thesis additional data is stored for each per cell, called acceleration bits, to improve the performance of hashing schemes. This chapter introduces a notation scheme for this data, possible uses as well as different methods to store it.

2.1 Notation

A hash table contains elements that consist of a key k and a value v. Furthermore, hash tables use a hash function $h(k)$ that maps the key to an integer in range of the hash tables capacity as indexer. If the cell to which an element is hashed to, which called assigned cell, is already filled by an element that was inserted earlier, it gets stored in another nearby cell. We call the index of the cell where an element with key k is stored $X(k)$. In this paper we inspect hash tables that store n_A additional bits for each index. These are called acceleration bits $A[x]$, with $A_i[x]$ being the i-th acceleration bit at index x. A Data Pair $P[i]$ is a pair of containing the element and the acceleration bits at index i. The acceleration table A is treated as an array containing the acceleration bits of all indexes. A hash tables capacity is the maximum number of elements it can store and its physical size is the space needed to store the hash table. A hash table which uses more bits per cell therefore has a lower capacity than another table with the same size using less acceleration bits.

2.2 Purpose

Each index has a set amount of acceleration bits where the hashing method can store additional information about the hash table to boost its performance. Already known applications using acceleration bits are hopscotch hashing where the bits give information about the position of elements that are hashed to the index [6], and deletion flags. Deletion flags improve deletion performance by marking the element of a cell as deleted in order to avoid an expensive backward shift deletion[3]. Other ways to use acceleration bits explored in this work include storing if an element was ever hashed to an index to preemptively stop unsuccessful searches and pointing to the first element that is hashed to a cell.

2.3 Storage

For the overall performance of the table, it is important where and how the acceleration bits are stored. If they are physically close to their corresponding element, accessing acceleration bits and elements together becomes faster. On the other hand sequential access on just one of them is faster if the acceleration data is stored in a separate table. If a storing method does not fully use its allocated space it not only has a lower capacity that is possible for a given physical space, but also loses some cache efficiency as the unused space could be loaded into the cache. Lastly, because some hashing methods access the acceleration bits of multiple cells sequentially, the speed of a lookup should not be ignored even when no cache miss occurs.

2.3.1 Split Tables Storage

These methods store elements and acceleration bits in separate tables. As a result a minimal amount of cash misses when accessing either only elements or only acceleration

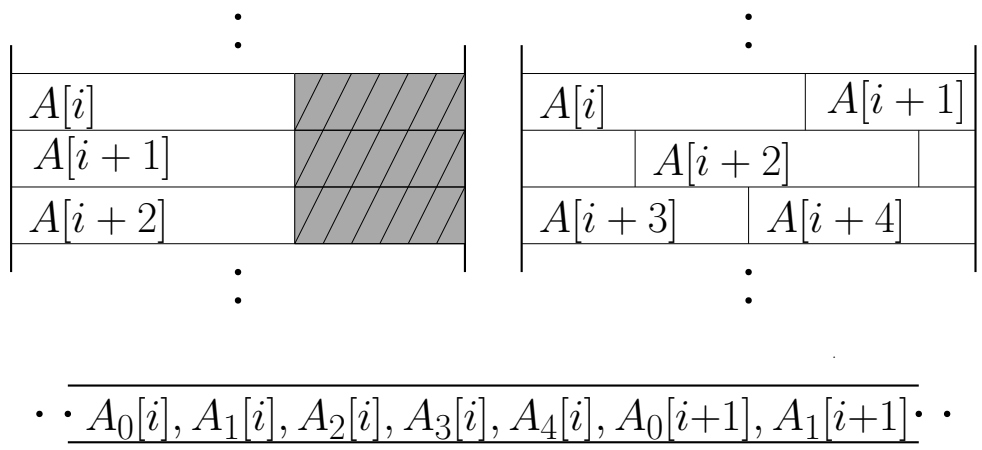

Figure 1: Data layout - split storage: acceleration table storing $A(i)$ in an array of a bigger data type (top-left), a continuous array (top-right) and a bit-by-bit storage (bottom).

bits sequentially is achieved as only relevant data is loaded into the cache. On the other hand it creates at least two cache misses when a hashing method needs a data pair, because the element and the acceleration bits are stored physically far away from each other. The simplest form of split table storage is to store the acceleration bits of each cell in a data type that uses the same or a higher amount of bits and then store these data objects in an array. The main drawback of using this method is that n_A should be equal to the size of a basic data type. Otherwise, allocated space is wasted as the remaining bits of the data type will not be used. Because of this, n_A is often bigger than needed ,e.g., using hopscotch with a bit-mask of 20 bits provides little benefit over using 32 bit as both would need the same space. To allow more flexible sizes without wasting space we use two other methods. The first one is bit-by-bit storage which splits the acceleration bits into single bits and the second one continuous storage, which stores the acceleration bits directly next to each other with no space between them. Figure 1 shows how these storage methods would look like.

bit-by-bit Storage Each bit is stored individually, e.g. using std::bitset or std::vector<bool> in c++, where the bits from $i * n_A$ to $(i+1) * n_A - 1$ are the acceleration bits for index i. This is especially useful if n_A is very small or if each individual bit plays a different role ,e.g., when the bits are used as flags. However the cost of combining the bits is linear in n_A, making this method inefficient if a large amount of acceleration bits per cell is used. The main advantages of bit-by-bit storage are that no space is wasted and any amount of bits per cell is allowed. To reduce the performance dependency on n_A it is also possible to use byte-by-byte storage if n_A is a multiple of eight, which uses bytes instead of bits.

Continuous The acceleration bits are directly next to each other in one array. To access the acceleration bits of a specific index, the byte containing the first bit of the acceleration bits that are currently demanded is interpreted as eight byte data type. Then the data object is shifted to the left so the bit is at the start of the data object. Afterwards a bit-mask is used to remove the suffix created by acceleration bits of later elements. This method has a constant runtime as it does not depend on the amount of acceleration bits each element has. The only wasted space of less than 64 bits is at the very end of the array and insignificant. If a data type with eight bytes is used it has to be guaranteed

that the acceleration bits are spread over no more than eight bytes, restricting n_A to be less than 59, equal to 60, or 64. In a 64 bit operating system using a data type bigger than 8 bytes would result in a more expensive bit shift and is therefore not recommended. This drawback is minor in the context of this work as all hashing methods use 32 or less acceleration bits per cell. Nevertheless, this restriction but plays a bigger role if a 32 bit operating system is used.

2.3.2 In Table Storage

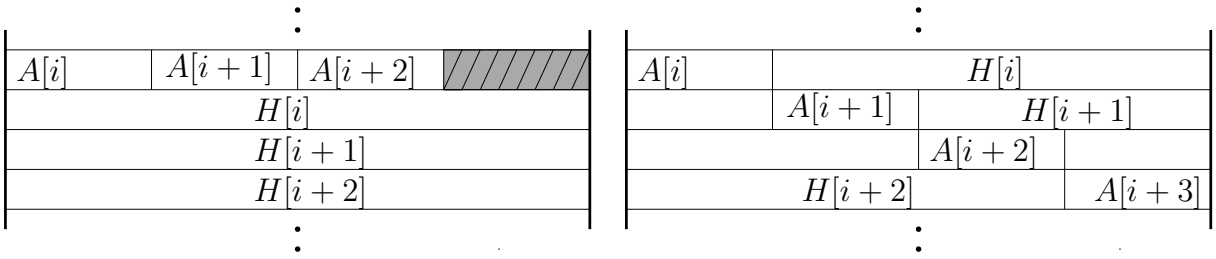

Figure 2: Data Layout - in table storage: Content of a 512 bit cache line with 128 bit elements and 32 bit acceleration data per cell for grouped storage (left) and alternating storage (right).

In tables storage methods store the elements and acceleration bits in the same table. If their index is the same they are stored physically close to each other, often allowing data pairs to be accessed with only one cache miss. On the other hand sequential access on either elements or acceleration data creates more cache misses than needed as both are loaded into the cache while only one of them is needed. For in table storage this thesis introduces two different storing methods. First is alternating Storage which has acceleration bits and elements of the same index directly next to each. The second method is grouped storage which stores groups of elements and acceleration bits. How these two storage methods would look in practice like is shown Figure 2.

Alternating Storage Acceleration bits and elements are alternating between each other in the array, with either the element following its corresponding acceleration bits (which will be assumed in this work) or vice versa. Because bit shifting the elements would be too expensive, the size of the acceleration data per cell is padded to a multiple of eight bit. This padding guarantees that the first bit of the acceleration data as well as the first bit of the element are always at the start of a byte. In this work we used a hash table with 128 bit elements, resulting in the acceleration data for index i being stored at byte $(\lceil n_A/8 \rceil + 16) \cdot i$ of the hash table and the element $\lceil n_A/8 \rceil$ bytes later.

Grouped Storage Data pairs are grouped together, such that the content of one group take as much or less space than one cache line. If the size of a group is smaller it has to be padded. Each group is then stored by first listing all acceleration bits, second the padded space and lastly the elements. By guaranteeing that a group is exactly as large as a cache line, groups do not lap over multiple cache lines and all content of a group can be accessed with one cache miss. In the context of this work, an element has a size of 128 bits and a cache line is 512 bits long. Therefore a group first contains three 32 bit blocks of acceleration data, then 32 bits of free space and finally the three blocks of 128 bits for elements. While it is possible to use a n_A of up to 42 with this setup, it would

require more operations to access the data and decrease performance. The main problems with this method are the wasted space that is used for the padding and that the amount of acceleration bits per cell is dependent on cache line and element size, forcing many hashing schemes to use more bits than they would normally.

3 Hashing Schemes

Hashing schemes specify how and where elements are stored in a hash table how collisions are handled. This chapter discusses open addressing and three hashing schemes that use it: linear probing, robin hood hashing and hopscotch hashing. For each hashing scheme multiple variations are introduced that use per-cell data to accelerate their performance.

3.1 Open addressing

Open addressing is a method for handling collisions in hash tables where all elements are stored in a single array with a fixed size. An elements key is hashed and the hash is then used as the index of the cell where the element will be stored. If this cell is already taken it is placed in another, often nearby, cell of the array which is determined by the coalition resolution. The main advantage of open addressing is its cache friendliness as all elements are stored in the same array, often in the vicinity of the index they are hashed to. The drawback is that it can only store a fixed amount of elements before resizing is needed and its performance varies depending on the amount of elements in the table relative to its size (load factor α).

3.1.1 Displacement

An elements displacement $D(k) = X(k) - h(k)$ is the distance between its keys hashed index and the index of the cell where it actually stored. Elements with high displacements often take more time to find, as most find-algorithms start at the keys hashed index which can result in additional element accesses and cache misses. Hash tables with a high load factor have increased displacement rates as more elements share their hashes and have to be placed somewhere else.

Note: The following sections will contain experiments for different configurations of each variation and only a few will later be used in the evaluation. We think this is necessary to avoid overfilling plots in the evaluation with configurations that are not as competitive as others. Often each variation can use up to five different storage methods in addition to having its own variations. For all experiments we used a fixed absolute size of 10 million 128 bit elements and xxHash [4] as hashing function. Each configuration is tested 50 times. The legend has the format (hashing scheme)_(variation)_(storage method)_(configuration), where storage methods can be *bit* (bit-by-bit storage) *char* (byte by byte storage), *con* (continuous storage using an array containing doubles), *alt* (alternating storage) and *group* (grouped storage where each group contains three data pairs).

3.2 Linear Probing

Linear probing (lp_simple) is an old algorithm for hash tables that was first analyzed in 1963 by Donald Knuth. An element that has to be inserted is stored in its assigned cell if it is empty. Otherwise, the algorithm linearly probes cells, starting at the elements

assigned cell, until an empty cell is found [8]. For lookups the algorithm probes the cells in the same linear order as the insert operation, starting at the searched elements assigned cell, until the element or a empty cell is found. For low load factors the number of cache misses for each operation is low as elements can be stored close to their assigned cells in most cases. However at higher load factors bigger clusters appear and the performance degrades more quickly.

3.2.1 Insertion

To insert an element its key is first hashed. The algorithm then checks the content of the elements assigned cell and stores the element there if it is empty. If the cell is not empty because of an earlier insertion, the algorithm steps one step further in the array and checks the content of the newly reached cell. This is repeated until an empty cell is found. The element is then inserted and the algorithm returns.
Because no elements are swapped, linear probing can create large displacements for elements that are inserted at the start of a large cluster. On the other hand it is also very fast as other hashing schemes, e.g, edit acceleration bits and swap or hash other elements inside the hash table, which reduces their speed.
The estimated amount of element accesses the insertion needs to find an empty cell is $O(1 + 1/(1 - \alpha)^2)$ [8]. For high load factors clusters appear in the hash table which have to be traversed, increasing the amount of element accesses drastically if the load factor comes close to 100%. The algorithm is rather cache friendly as it accesses the cells in linear order, which is one of its main advantages.

3.2.2 Lookup

To find an element the linear probing algorithm looks at each cell in the same order as the linear probing algorithm does for insertions. The element in each passed cell is compared with the key that is searched. It stops once the searched element or an empty cell is found, which takes an average of $O(1 + 1/(1 - \alpha))$ [8]. Because lookups traverse the table in the same manner as the insert operation and unsuccessful ones only stop at empty cells, which is where the insertion algorithm would also stop, the amount of element accesses is the same for unsuccessful lookups and insertions.
For both operations, lookups and insertions, the amount of element accesses a that single operation performs has a high variance. This is a result of element clusters forming inside the hash table. Therefore, elements that have their assigned cell at the very start of a cluster could be stored at the very end of it.

3.2.3 Deletion

Because the elements are in no particular order and an element can only be found if there is no empty cell between the element and its assigned cell, one cannot simply remove an element. Doing so would cause later lookups to stop at the cell that contained the deleted element, even if an element that is stored further ahead could be the searched element. For this reason it is necessary to search for elements that are stored further ahead in the hash table, but could also be placed into the deleted elements cell. To do so it is necessary to hash all elements between the deleted one and the next empty cell. Starting at the cell of the deleted element, each cell is linearly passed to find an element that can be placed into the deleted elements cell. If the hash of a passed cells element is lower or equal to the index of the deleted elements cell the element is moved into it. Afterwards the algorithm continues by searching for an element that can be placed into the now empty cell where the just moved element got taken from. This is repeated until an empty cell is reached.

It is also possible to use a lazy deletion strategy and mark the cell of a deleted element using a deletion flag. This allows the search algorithm to skip over it. The drawback here is that, for each cell with a deletion flag, unsuccessful lookups will on average take longer, because empty cells without a set deletion flag are rarer. For this reason none of the tests in this work will use deletion flags.

3.2.4 Variations

One of the main disadvantages of linear probing is, that unsuccessful lookups have to iterate over each cell until they find an empty one to return. To improve this we present two methods (is-Empty bit and bloom filter) that allow some of these lookups to return without accessing any elements by using the per-cell data to sometimes guarantee that an lookup will be unsuccessful. Another property of linear probing, which can be a problem, is the high variance in the amount of elements a lookup has to access. To reduce this variance we present a variation of linear probing (maximum distance) where all lookups can return after a set amount of element accesses. Our variation of coalesced hashing also fixes this issue by allowing lookups and insertion to skip most elements that do not have the same hash as the currently searched or inserted element.

Is-Empty Bit To increase the performance of unsuccessful lookups, each cell has a single bit (lp_ebit), which is set if there is an element in the hash table which hash is the index of the cell. Now the find algorithm first checks this bit and only continues if it is set. If the bit is not set it returns as it already knows that the search will be unsuccessful.

$$A_0[k] = 1 \iff \exists i \in \mathbb{N} : h(P[i].key) = k$$

For low load factors the find algorithm can often return due to this bit not being set. Nonetheless, the performance increase is only small, since a simple linear probing algorithm would not continue for long anyways as most cells are empty. On the other hand, if the hash table has a high load factor, most of the is-Empty bits will be set and therefore lose their effectiveness as fewer lookups can preemptively return.
This method also has a worse deletion performance as it has to check if there is another element mapped to the same cell as the deleted element. To do this, up to all keys between the assigned cell of the deleted element and the first empty cell have to be hashed. This is very inefficient, however elements after the cell of the deleted element already have to be hashed for simple linear probing and its performance is therefore not that much worse than simple linear probings. Because deletion flags are preferred for low amounts of deletions or low load factors, not updating the is-Empty bit if deletion flags are also used could be advantageous. At the same time the performance increase will probably be low for low load factors, since only a few elements have to be hashed as clusters sparse and small for low load factors.

Bloom Filter For high load factors is-Empty bits are inefficient as most bits are set. With the majority of cells marked because another element is mapped to them, the algorithm has to continue until it finds an empty cell. To achieve a better negative lookup performance even at high load factors a bloom filter [2, 11] can be used instead. It allows up to 92% of unsuccessful lookups to return without accessing a single element. Bloom filters are probabilistic data structures that can tell if an element is definitely not in a set or if it might be in it. They use a bit array and hash elements multiple times, with each hash being used as an index for the bit array where the bit with the index is set. To find

out if an element was inserted the same procedure is used, this time checking if the bits are set instead of setting them. If one of the bits is not set the bloom filter can guarantee that an element was not inserted.
A bloom filter can be used as a replacement for is-Empty bits by adding the key of an inserted element to the bloom filter at the elements assigned cell. In this thesis only the last bits of the hash function are used to determine an elements assigned cell. This allows for the remaining bits to be used as the hashes of the bloom filter, improving the performance as no additional hash functions are needed.
The amount of negative searches that pass the bloom filter is dependent on the size of the bit array and the number of bits set for each element. While increasing these values increases the accuracy, it also results in more allocated space or worsens the time the algorithm needs for passing the bloom filter as it has to inspect more bits.
According to our tests (Figure 3) for low load factors (<50%) it is best to use eight bits for the bloom with two hashes per element and a 16 bit sized filter with two hashes per element for medium load factors (<85%). If the table size is fixed and the load factor gets too high the 16 bits variations cannot insert all elements. In the case that the 16 bits variation cannot insert all elements an eight bit sized variation with two hashes per element should be used again.
Simple bloom filters do not support the removal of inserted elements, therefore a new bloom has to be created every time an element is deleted from the table. This forces the algorithm to not only access all elements between the deleted elements assigned cell and the cell it is stored in, which is needed for linear probing, but to also hash them.
Creating a new bloom filter on every deletion can be avoided by using a counting bloom filter [5, 11] instead. Each bit of a simple bloom is replaced with a counter which increases whenever the simple bloom would set the bit. Deleting an element can now be done by simply decreasing the counters. There are two major reasons why we decided not to use this method. First, the paper of H. Song [11] used this method with separate chaining as collision resolution, which does not require to hash other elements when an element is deleted while linear probing does. Second, the space needed to store a counting bloom filter. The number of acceleration bits per cell has to be at least tripled, because using less than 3 bits per counter could quickly lead to an overflow.

Maximum Distance This method tries to minimize the maximum displacement d_{max} in the hash table. The algorithm for swapping is a variation of the one used for hopscotch hashing, described farther ahead in Section 3.4.1. First an element is inserted using linear probing. Afterwards it is swapped with other elements so that all elements have a displacement lower than the current maximum displacement. If that is impossible d_{max} is incremented by one and the algorithm tries again.

$$\forall k \in Keys : X(k) - h(k) \leq d_{max} \wedge d_{max}\ minimal$$

Now the find algorithm can stop if the maximum displacement is reached, as the searched element cannot be in farther ahead, and does not need to continue until an empty cell is reached. For backward shift deletions one can either ignore that the maximum displacement could go down or use an additional data structure described in Section 3.3.2. This would also affect the insertion time. As this method does not lower the average displacement and forces the find algorithm to check an additional condition for every cell it passes, the average time of successful lookups is higher. Unsuccessful lookups, on the other hand, are only faster if the load factor is very high. This is shown in Figure 3. One

advantage of maximum distance linear probing is that the variance of a single lookups time is far lower.

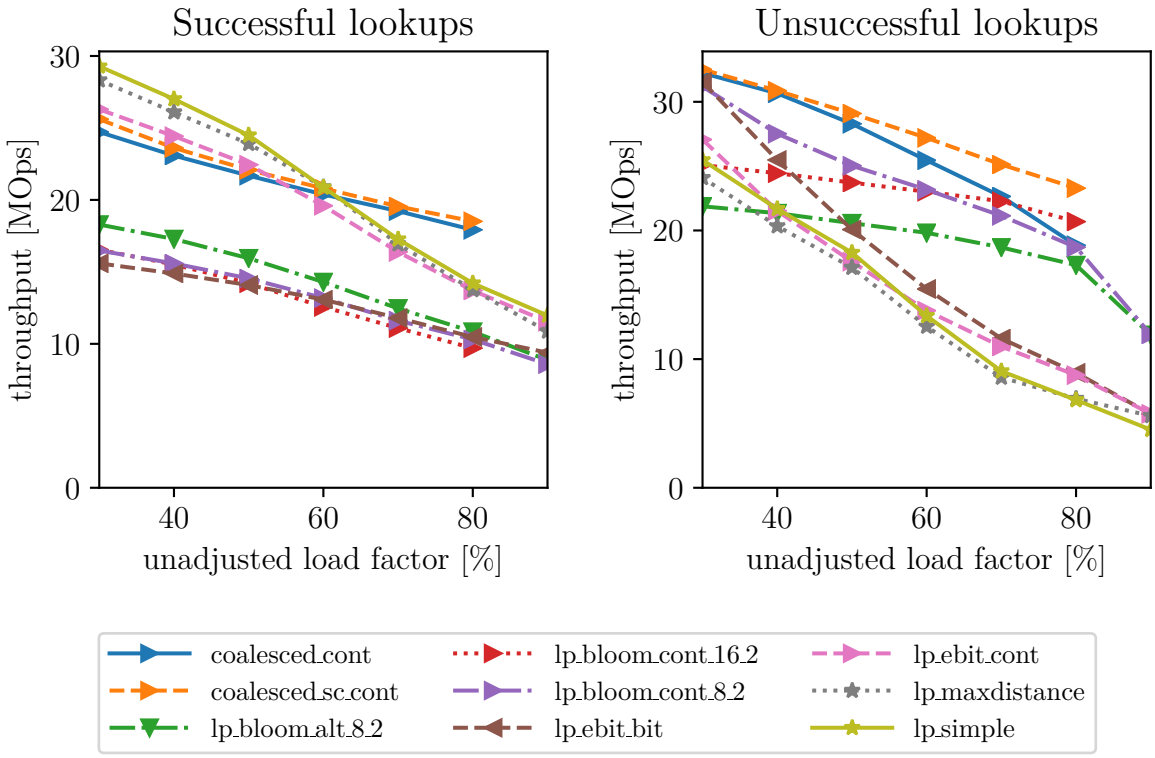

Figure 3: Linear probing variations throughput for successful (left) and unsuccessful (right) lookups.

Standard Coalesced Hashing One idea is to enable the skipping of elements with a different assigned cell than the element one is looking for by creating linked lists for elements with the same assigned cell inside the hash table. Each cell stores a pointer to the next element in the list inside its acceleration bits. If an element is inserted, the algorithm follows these pointers, with the first pointer being at the cell the element is assigned to, until it reaches the last element of the list. It then searches an empty cell, inserts the element there and sets the pointer of the last element of the list to the previously empty cell. To find an element one can now simply iterate the linked list that starts at an elements assigned cell.
This method is a specific case of Coalesced Hashing without a cellar called standard coalesced hashing [12], however there are two differences to the implementation in the original paper. The first difference is the place where elements are stored when the assigned cell is already filled. The original paper stores them at the very end of the hash table while we try to store them near their assigned cell. Secondly we reduced the number of acceleration bits needed for each cell by storing the amount of cells between the lists elements, instead of their actual index, and use these distances as pointer by adding the index of the current element to it. An example for our implementation of standard coalesced hashing is shown in Figure 4.
Because lists can store their elements in the cells where other lists would start, it is possible for an element to be hashed to a cell where an element of another list is. This

means that the cell where a list of elements with the same assigned cells starts can be in the middle of another list. Lists can therefore contain elements with different assigned cells and the search operation still looks at them.
We propose another insert algorithm to guarantee that all members of a list are assigned to the same cell (coalesced_16_sc). This is achieved by comparing the assigned cell of the element that has to be inserted with the assigned cell of the element that the cell contains. If they are the same, the algorithm can proceed like normal. If they are not the same the algorithm swaps the two elements and corrects the pointer to the cell: Either to point to the next element of the list or to point nowhere if there is no such element. Afterwards, the element that has been removed from the hash table is inserted again. Overall this results in lists being shorter and therefore finding an element in a list is generally faster.
Deletions with standard coalesced hashing have to be handled carefully if their value is the distance to the next elements. They only have a limited range and, after deletions occurred, it is possible for empty cells to exist between the elements of a list. Pointers in a hash table with standard coalesced hashing in most cases do not shrink and their value only decreases if they are removed after all later elements of their list have been deleted. As a result it is far more likely for a pointer to exceed its maximum range if deletions occur and no precautions have been taken.
To improve the longevity of the hash table using coalesced_16 we remove all elements that are stored after the removed element and that are part of the same list, and reinsert them. This guarantees that no empty cell exist between all removed elements and the last element in the list that was not removed. Therefore the distance between the lists elements remains small even after many deletion/insertion cycles.
For coalesced_16_sc on the other hand we utilize that the elements of each list all have the same assigned cell. It allows the remaining elements of the list to all swap with the element before them or the last element of the list to be placed in the cell of the deleted element. As a result deletions do not increase the distance between elements, however they also do not reduce them which should make this method worse for the hash tables longevity. However, it is also far faster than the previous method as no additional elements are hashed and fewer elements are accessed.
We had two other ideas for deletions that we did not implement. The first is very fast but unsustainable: deletions simply remove an element and update the pointer pointing to the cell that contained the element. As a result lookups have to continue until a cell without a pointer is found, even if it passes an empty cell on the way, resulting in worse lookup performances and a worse longevity. The second idea is for insertions to not follow the cells pointers but instead use the same insert as simple linear probing. The pointers can be updated after an empty cell is found. As a result it does not matter if there are empty cells between the elements of a list and deletions do not have to worry about creating them. If the same deletion algorithm as coalesced_16 is used the longevity of the hash table should be guaranteed.

3.3 Robin Hood

Robin hood hashing (rh_simple) was first introduced in 1985 by P. Celis and P. Larson and J. I. Munro. The idea behind it, is to move elements based on their displacement to have a lower variance in the overall displacement of elements. While for linear probing elements that were inserted early generally have lower displacement than the elements that are inserted later, and are never moved unless a deletion occurs, in robin hood hashing the time when an element has been inserted matters little as it tries to give each element the

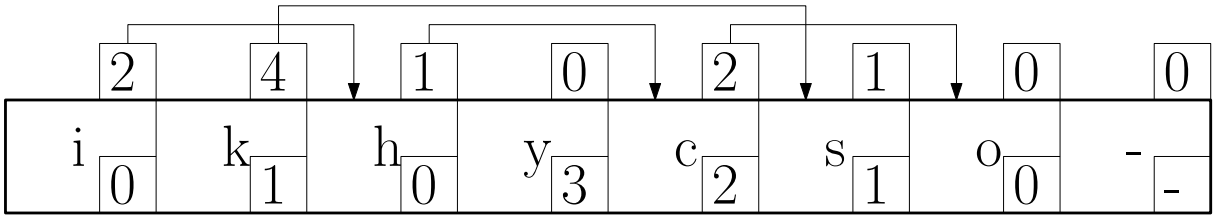

Figure 4: Example of a hash table using standard coalesces hashing (coalesced_16). The letters represent the elements, the number at the bottom right is their hash and the number above the distance to the next element of the list.

same displacement. Elements that have a low displacement are placed farther away from their assigned cell to allow other elements that have a high displacement to be placed closer to their assigned cell.

3.3.1 Insertion

To achieve that elements have a normalized displacement, the insertion algorithm swaps elements during the insertion depending on their displacement. It goes linearly over each cell, starting at the inserted elements assigned cell and hashes every passing element. The displacement of each element that is passed is compared with the displacement of the element that is currently inserted had it been in the cell instead. If the passed element has the lower displacement it will be placed into the cell. To make space for it, the current element in that cell and all following ones until the next empty cell are moved by one cell. The element is then inserted into the now empty cell.
The algorithm also stores the maximum displacement that is currently in the table, allowing unsuccessful find operations to stop if the maximum displacement is reached. While the maximum displacement is needed to allow unsuccessful lookups to return early, every time an element is moved inside the hash table its new displacement has to be calculated. Therefore the find algorithm, which often shifts many elements by one to make space for the currently inserted element, has to hash far more elements than it would otherwise have to.
One property of robin hood hashing is that the elements of the table are sorted by their hash and all elements with the same assigned cell are directly next to each other, which will be important for the variation this thesis introduces.

3.3.2 Deletion

Each element after the deleted cell is hashed and, if the hash is lower than the index of the cell they are in, moved one step back. If the hash is not lower it means that the element is already in its assigned cell and cannot be shifted back, making the algorithm stop. Otherwise, it continues until an empty cell is reached. For the maximum displacement counter, the first method is to simply not update it and ignore that the actual maximum displacement could be lower than the counter. This can be done, because it would only force unsuccessful find operations to do more cell lookups than necessary. The second method is to store an array of integers which tell how many elements for each displacement exist. Therefore, the value at index zero proclaims how many elements have a displacement of zero, the value at index one how many elements have a displacement of one, etc. The array does not need a lot of space as robin hood keeps the maximum displacement at its minimum. The drawback of this method is that the array has to be updated on every swap which also includes the insert operation, therefore reducing its performance. In this

work we choose to use neither of these methods and simply not support deletions if one of them would be needed to not influence insertion or lookup times.

3.3.3 Variations

While robin hood hashing has a far lower variance than linear probing, the additional performance loss of the insertion operation is often too high to justify using it. The variations we present try to solve this by using the sorting that robin hood hashing creates to improve the performance of lookups. Our first variation (Startpointer) will use a pointer to skip directly to the first element that is assigned to the same cell as the element that is searched for or that is getting deleted. The second variation (3Bit) will scan over the acceleration bits of multiple cells to archive the same.

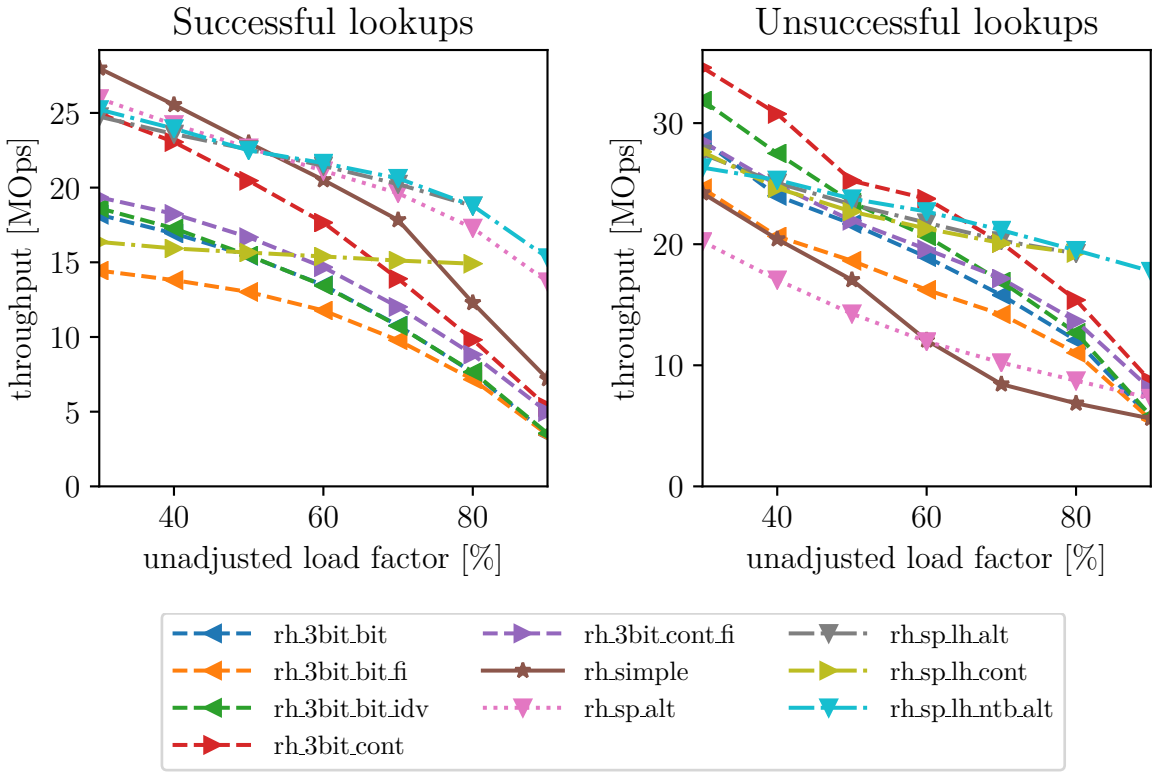

Figure 5: Robin hood hashing variations throughput for successful and unsuccessful lookups.

Startpointer Each cell gets one byte of acceleration data, called startpointer (rh_sp), which, added to the cells index, is the index of first element that is mapped to the cell. Because hash tables that use robin hood hashing are sorted, elements with the same hash form a chain. The startpointer can be used to determine where the chain of the elements with the same assigned cell starts. As a result the find algorithm only has to go over this chain to find an element.
One implementation for robin hood hashing with startpointers initiates all acceleration data with a bytes maximum value to identify cells that have no element hashed to it. An example of it is shown in Figure 6. Because the find algorithm only knows where a chain starts, but not where it ends, unsuccessful lookups have to continue until the maximum distance or an empty cell is reached. Other stop conditions, like hashing each passed

element to see if the chain ended or searching for the startpointer that tells where the next chain would start, are also possible. It is interesting to note that the insert algorithm in our tests was slower if it used the startpointer to find the start of the chain. Most likely this is because of branching issues as it has to check if the startpointer has the maximum value.
A second implementation (rh_sp_lh_ntb) has each cell, that contains an element but has no element hashed to it, point to the same position as the next cell that has an element hashed to it. Now the end of a chain is simply the startpointer of the cell that is next to the one indicating where the chain starts. Insertions can also be improved as an element can always be inserted at either the startpointer of the initial cell, if it previously had no element hashed to it, or at the end of the chain. To increase the performance of unsuccessful finds it is also possible to only use seven bits as startpointer and the last bit as is-Empty bit (rh_sp_lh). On the other hand using only seven bits for the startpointers also increases the probability that an element has a displacement larger than what is addressable with seven bits and thus fail the insert. An unsuccessful insert because the element was placed outside the addressable range is far less likely if eight bits are used instead of seven.
The efficiency of the startpointer variations comes from the low amount of cells that are accessed and the fact that these cells are all directly next to each other in the table, which allowed for up to 100% faster lookups than simple robin hood hashing. A find operation therefor only accesses the startpointer and all elements that are mapped to the cell, plus a maximum of one, if its unsuccessful, creating a very low amount of cache misses.
The deletion algorithm is similar to the one used in simple robin hood hashing. The only difference is the updating of startpointers for shifted elements.

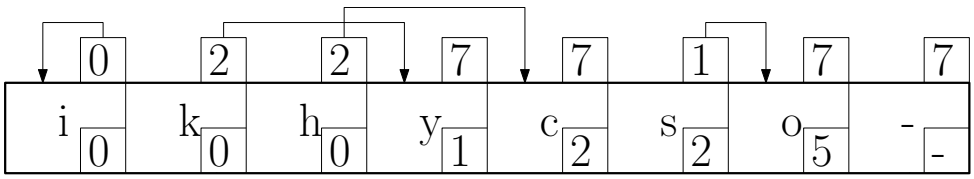

Figure 6: Example of a hash table using robin hood with startpointer (rh_sp) with $n_A = 3$ and is Empty bit combination. The letters represent elements and the number to their bottom right is their keys hash.

3Bit A relative unknown variation of robin hood is based on quotient filters [1] and will be called 3Bit in this thesis. Similar to startpointers, 3Bit robin hood only has to access acceleration bits to find all elements that are mapped to a cell. Like a quotient filter, a cell gets three acceleration bits, each to describe if one of the following statements is true:

- is-occupied bit: The cell has an element mapped to it
  $A_0[i] = 1 \Leftrightarrow \exists j \in \mathbb{N} : h(P[j].key) = i$
- is-continuation bit: The key in the cell and the previous cell have the same hash
  $A_1[i] = 1 \Leftrightarrow h(P[i].key) = h(P[i-1].key)$
- is-shifted bit: The element in this cell has a displacement higher than zero
  $A_2[i] = 1 \Leftrightarrow h(P[i].key) = i$

To find the elements that are mapped to a cell, the algorithm first checks if the cell has an element mapped to it and stops if that is not the case. It then goes backwards until the first element with a displacement equal to zero is found, counting all cells that have an element mapped to them as well as every occurrence of an unset is-continuation bit. Afterwards it goes back to the cell where it originally started, stops counting cells that have an element mapped to them and continues forward, only counting unset is-continuation bits. Once both counters have the same value the first element that is mapped to the cell is reached and the algorithm can start comparing the keys of each cell. Starting with the second element, unsuccessful finds can be identified using the is-continuation bit. If this bit is set, the algorithm has passed all elements that were assigned to the same cell as the element that is searched. Therefore the search was unsuccessful and the algorithm can return.
For insertions, because we do not use a maximum displacement, not every element that is moved has to be hashed, since the acceleration bits can be updated without hashing the elements (rh_3bit). However the same tactic used for the find algorithm can also be used to find the position where an element has to be stored (rh_3bit_fi). This allows us to not hash any element in the table, and we only need to hash the element that has to be inserted. In theory this should not change the time of element lookups, however in our experiments the find operation was slower if 3Bit_fi was used for a not yet solved reason. As 3Bit robin hood hashing accesses the acceleration bits of multiple cells while ignoring their element, it is recommended to use split table storage for the acceleration table. On some occasions the 3Bit algorithm does not need all acceleration bits at once, which can be used to increase performance if the storage method allows access of individual bits (rh_3bit_idv).
Figure 5 shows how bit-by-bit storage is worse than continuous storage, even if only three bits are used. If eight bits are used, the difference between byte by byte storage and continuous storage is negligible. Another important note is how variations that often iterate over the acceleration data are far better of using split table storage. This is best seen by looking at the startpointer variation with one hash per element using continuous and alternating storage, as it iterates little for low load factors and more for higher ones.

3.4 Hopscotch

Hopscotch hashing was introduced in 2008 by M. Herlihy, N. Shavit and M. Tzafrir. The idea behind hopscotch hashing is, similar to many of our variations, that finds only need to access cells that are storing an element that was hashed to the assigned cell. To achieve this, hopscotch hash tables (hop_simple) store all elements inside a set range of the cell they are assigned to, called neighborhood. In addition, they use per-cell data to store at each cell the positions of elements that are assigned to the cell. They do so by using a bit array, the nth bit of the bit array at index i denotes if the element at cell n+i is assigned to index i.

$$A_i[x] = 1 \implies h(P[x+i].key) = x$$

Now the hopscotch algorithm can skip all cells where the corresponding bit is not set as it knows that their element is mapped to another cell. This is done by starting at the assigned cell and looking at the first bit of the bit array, if it is set the element at the cell is accessed. If it is not set or the accessed element was not the one that is searched, the bit array is left shifted by one and the process repeated for the next bit until the element is found or the bit array only has zeros remaining. Because the lookup algorithm always checks all elements in the neighborhood of a cell, regardless if cells inside it are empty,

deletions can be done by simply removing the element and setting the corresponding bit in the bit array back to zero. An example for how a table using hopscotch hashing looks like can be seen in Figure 7.

3.4.1 Insertion

To insert an element an empty cell is first searched by starting at the assigned cell of an element and begins a linear search. If the found empty cell is inside the neighborhood of the elements assigned cell, which means that the bit array has a bit responsible for the empty cell, the bit is set and the element inserted. Otherwise, the algorithm has to create an empty cell inside the neighborhood of the elements assigned cell.
This is achieved by repeatedly finding an element closer to the assigned cell that can be moved into the empty cell without it leaving the neighborhood of its own assigned cell. To find such an element the algorithm starts at the first cell that has the empty cell inside its neighborhood and hashes the element in the cell. If the index of the empty cell minus the hash is smaller than the range of a neighborhood, the element can be stored at the empty cell and is therefore moved. If the difference is not smaller, the same repeats with the cell one step closer to empty cell. Should the algorithm reach the empty cell, it returns with an unsuccessful lookup. Otherwise elements are moved using this method, until an empty cell in the range of the inserted elements bit array range is created. The element is then inserted into this created empty cell and the responsible bit in the bit array is set.

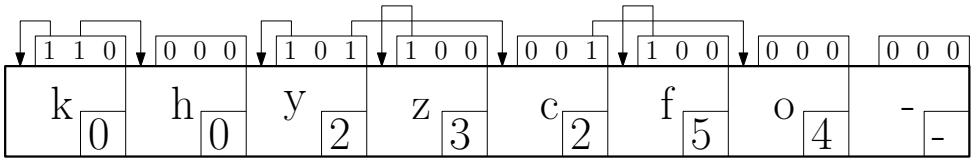

Figure 7: Example of a hash table using hopscotch hashing with $n_A = 3$. The letters are elements and the number to their bottom right is their hashed index.

3.4.2 Variations

For hopscotch hashing to work, the neighborhood has to be relative large - in this work it contained 32 cells. However a hash table with a capacity of 10 million elements could not be filled to 80%. To allow higher load factors without increasing the neighborhoods size, we present hopscotch hashing with out of bounds bit. It uses an additional bit per cell which will be set at a cell if an element was hashed to the cell but could not be placed inside its neighborhood.
Another problem of hopscotch hashing is that often the needed amount of acceleration data is too high. In our tests the acceleration data size was a quarter of the size of the data array and a hash table using linear probing would have had a 25% higher capacity using the same space. To reduce the size of the acceleration data we implemented a variation (Areas) that allows for each bit in the bit array to be responsible for more than one cell. This increased the neighborhoods size and allowed us to reduce the number of bits used per cell.

Out of Bounds Bit To avoid the resizing needed in simple hopscotch hashing, one bit is added to the bit array that indicates that there is an element assigned to the cell which is outside its neighborhood (hop_oob).

$$A_{i_{max}}(x) = 1 \implies \exists j \in \mathbb{N} : H(D(x+j).key) = x \wedge j \geq n_A$$

If an element can not be inserted using the simple hopscotch hashing algorithm, it is stored inside the empty cell at which the algorithm stopped and this bit is set. Because this only happens at large clusters in the table, only a few of these out of bounds bits are set. If the bit is set and the element was not found inside its assigned cells neighborhood, the find algorithm begins a linear search. This search starts at the first cell outside of the neighborhood.
Because now linear probings search is used, deletions have to be done more carefully. In addition to deleting the element, all elements between the first empty cell and the cell of the deleted element have to be checked. If they are out of bounds they are moved to the empty cell if possible, creating a new empty cell where the element was previously and the process is repeated. This is very inefficient as shown in Figure 9. A large bit array is not necessary anymore as its always possible to fill the hash table to its maximum capacity, independent of the neighborhood size. As the large neighborhood of simple hopscotch hashing is only necessary because of a few big clusters, it can be advantageous to, e.g., only use eight bits for the bit array if the out of bounds variation is used as it reduces the space needed for the hash table.

Areas To increase the range of the bit array, in the hopscotch hashing with areas variation a single bit can be responsible for more than one cell. If a bit is set all cells its responsible for have to be checked during the lookup. The following configurations were tested in this work:

- Constant: Each bit is responsible for the same amount of cells (cons).
- Linear: It is linear increased, giving a bit one more cell than the previous bit.
- Exponential: With each bit amount of cells its responsible for doubles (exp).

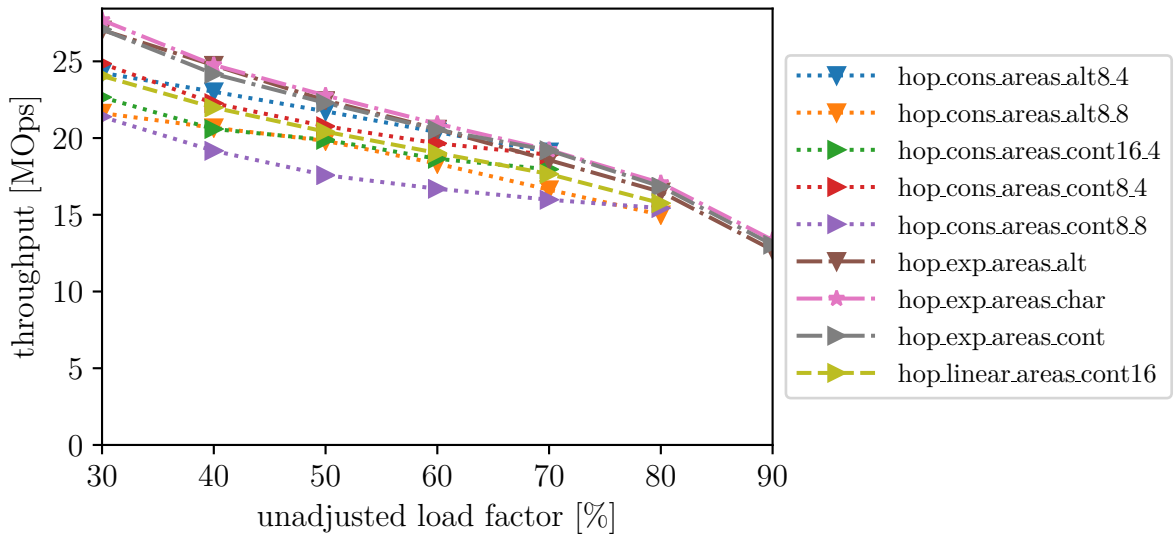

Figure 8: Hopscotch hashing with areas throughput for unsuccessful find operations, the first value at the end of the variations name is the number of n_A (8 for exponential areas) and, for constant areas, the second value is the number cells each bit is responsible for.

In Figure 8 it can be seen that for negative lookups variations where the amount of cells a bit is responsible for increases are faster than the ones each bit is responsible for the same

amount of cells. This is mainly because most elements are stored close to their cell and, if a key is not found, fewer cells have to be looked up. The chance of a later bit being set is low because most elements are stored close to their assigned cell. As a result accessing many element because a bit that is responsible for many cells was set only happens in a few instances. Furthermore, it is important to note that doubling the neighborhood of cells only has a small effect on the maximum possible load factor. Simple hopscotch hashing seems to be the best performance-wise. Nevertheless, at a high load factor it is not usable as elements can not be placed inside their neighborhood because they are already full (Figure 9). However the variations we presented allow a high load factor and can be used instead.

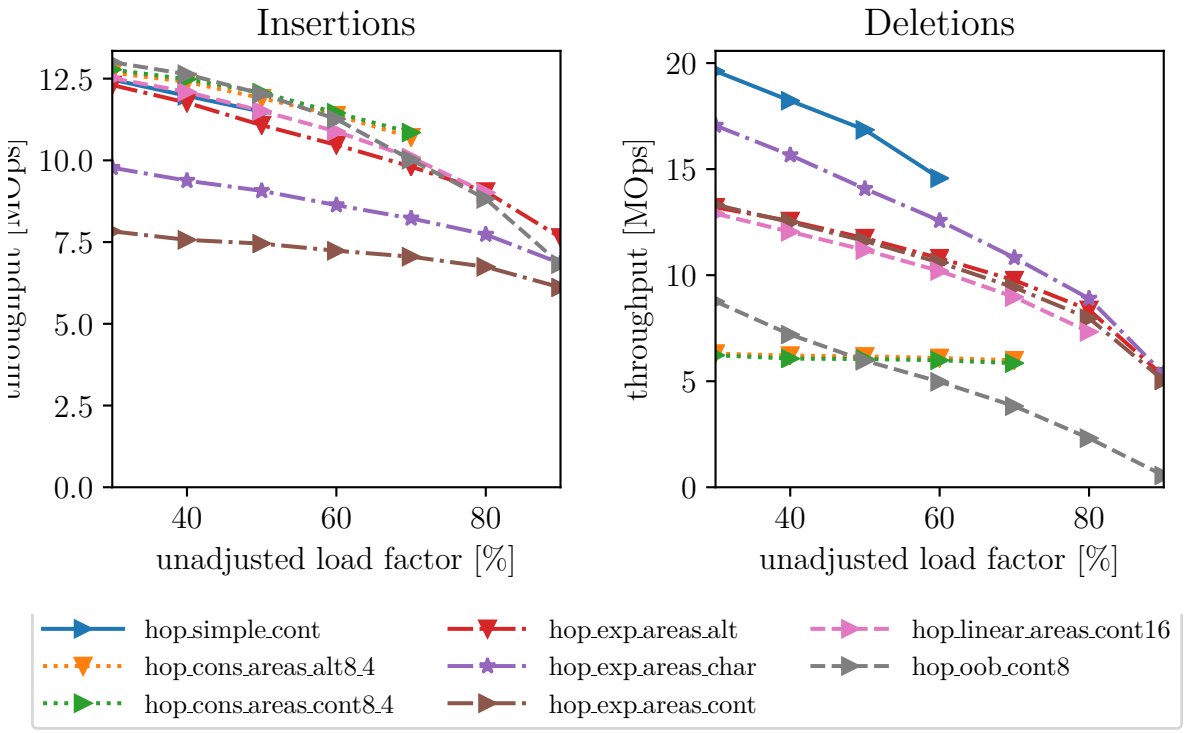

Figure 9: Hopscotch hashing variations throughput for insertions and deletions. The first value at the end is the number of acceleration bits (exponential always uses 8) and, for constant areas, the second value is how many cells each bit is responsible for.

4 Evaluation

Hashing schemes can be evaluated by a variety of factors like element size, hash function and load factor [10]. In this paper a hashing schemes performance was evaluation for each operation separately using different load factors. Hashing function and element size were fixed while, depending on the scenario, either the hash tables capacity or its allocated space was fixed.

4.1 Setup

The machine used for the tests had multiple AMD Ryzen 7 1800X Eight-Core CPUs which ran at 1.9GHz. All implementations are written in C++, only use a single thread and use xxHash as hashing function [4]. The insert operation is tested by measuring the time it takes to fill an empty hash table with random elements to a given load factor. If one of the tests failed because the hash table could not insert an element, we labeled the hash table as not usable for the current load factor. For the performance of positive lookups all inserted elements were searched once in the same order they were inserted. The performance of negative lookups was tested by searching randomly generated elements. To measure the throughput of deletions, the hash table was first filled to a given load factor, then we deleted every tenth element in the order they were inserted. Afterwards the table was filled to the load factor again. This deletion-insertion process was repeated 10 times so that all elements that were present at the start of the deletion process had been removed. The times between the start and end of each deletion process was measured and added together. All hash tables were tested 50 times for each load factor, with different seeds every time. The same seeds were used for each hash table. A compact overview of the most competitive hashing schemes in this thesis can be seen in Table 1.

4.2 Unadjusted Capacity

Our first batch of tests (Figure 11) used an unadjusted capacity, meaning that each hash table has the same maximum capacity, regardless of how much space they would actually occupy. This can be useful in a scenario where speed is the primary concern and allocated space only plays a background role. While other load factors are also shown on the graphs, the evaluation will be focused on the performance of hash tables at a load factor of 50%. Looking at the performance of the insert operation, it is unsurprising that simple linear probing has the best performance. This is mainly because nearly every other hashing scheme does the same as linear probing at some point during the insert, which is iterating each cell to find an empty one, in addition to the other tasks they have to perform. Linear probing is followed by standard coalesced hashing without shortened chains as it, similar to linear probing, iterates over cells until an empty one is found. Its performance is most likely worse than simple linear probing because it can only skip cells in a few instances as the load factor is low. Robin hood hashing is third, most likely because it only has to iterate over a few cells and hash them. The bad performance of robin hood hashing with startpointers and less hashing as well as hopscotch hashing with exponential areas can be explained by the large minimum amount of operations they perform.
While coalesced hashing comes out at the top, most featured hashing tables have the same throughput for successful lookups if the capacity is unadjusted. This can be explained by looking at the variance of elements that have to be looked up before the right element is found. Less than 15% of elements can not be found with the first two element accesses. For

unsuccessful lookups using an is-Empty bit is faster than simple linear probing, however coalesced hashing and 3Bit robin hood hashing still have the best performance. This is most likely because the is-Empty bit is stored in a separate table and lookups that passed it will cause at least one additional cache miss. The standard coalesced hashing variations on the other hand often have the elements they access in the same cache line and accessing them is therefore faster. Simple linear probing and robin hood hashing are at the bottom as they both iterate over multiple elements until they stop. Hopscotch hashing is in the middle as it only has to access a few or no elements, however it needs to shift the bit array multiple times to get the position of the elements that are assigned to a cell.
Hopscotch and coalesced hashing show great deletion performances as they do not have to rearrange multiple elements in the hash table during the deletion process. However in most cases the insertion performance also has to be taken into account when looking at deletions. For every deletion an insertion has to have occurred, therefore coalesced hashing has an advantage. On the other hand, depending on the implementation of coalesced hashing, the longevity of the hash table can be far worse than hopscotch hashing. The only other hashing scheme with a good deletion performance is 3Bit robin hood hashing which suffers the same problem as hopscotch hashing of having a bad insertion performance. The bad deletion performance of hopscotch hashing with out of bounds bit and linear probing with is-Empty bit is caused by hashing all elements up to the next empty cell and updating their acceleration bits.

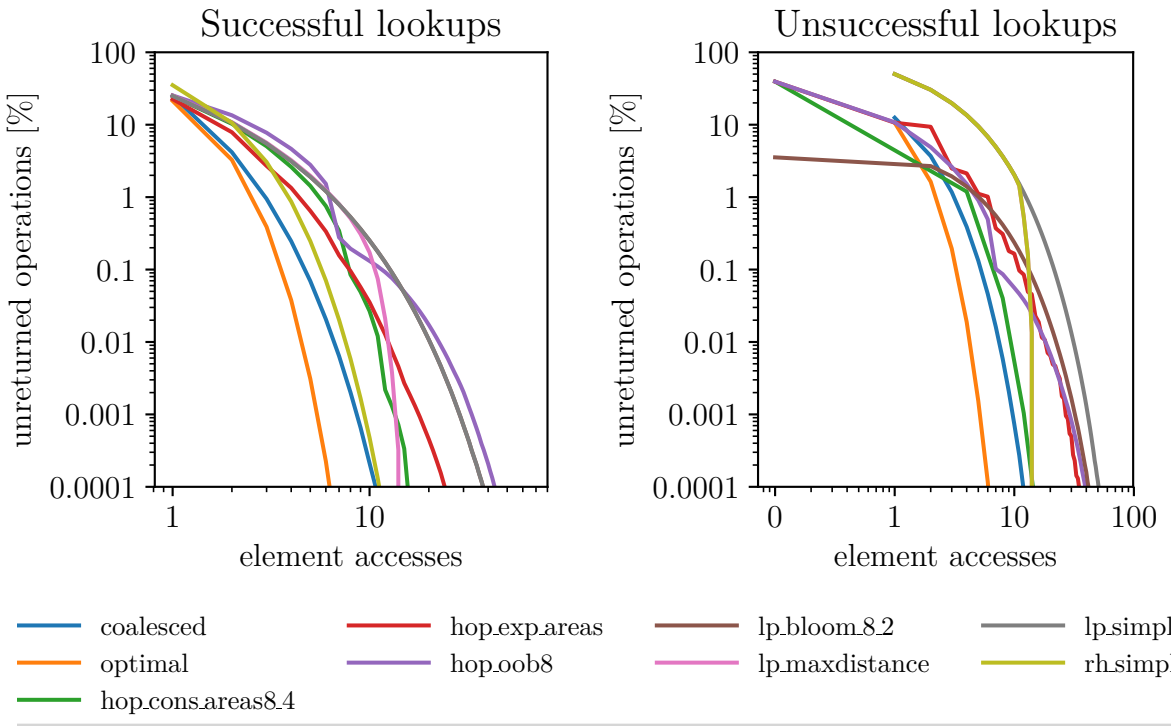

Figure 10: Percentage of lookups that require more than the given amount of element accesses before returning (with unadjusted capacity of 50%) for successful and unsuccessful lookups. *Note:* The following hashing scheme variations are considered optimal (only scan through elements that are hashed to the same cell): hopscotch, rh_3bit, rh_sp_lh_ntb and coalesced_sc. The last two hashing scheme variations access at least one element before they can return.

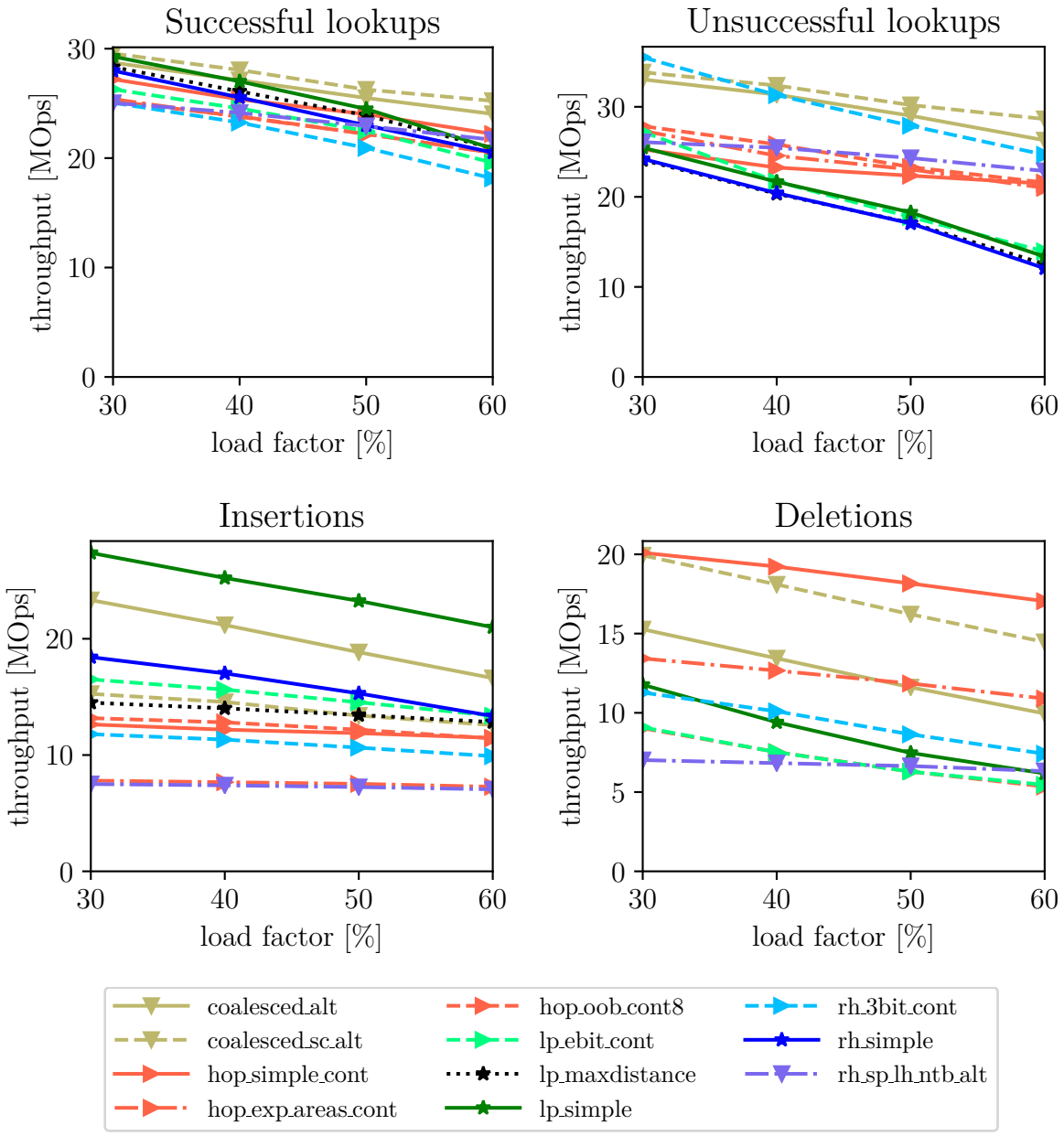

Figure 11: Throughput of different hashing schemes using a low load factor and an unadjusted capacity.

4.3 Adjusted Capacity

Our second batch of tests (Figure 13) adjusted the capacity of each hash table to factor in the space allocated for the per-cell data. This adjustment forces each hash table to occupying the same space as a hash table with the given capacity that uses no per-cell data. As a result hash tables that use a lot of acceleration bits have their capacity reduced by more than 10%. Moreover, it means that some hash tables could not support a high unadjusted load factor, which is the load factor of the hash table if its capacity had not been reduced.
For insertions linear probing has by far the best performance. It is followed by its own variations as they only do a few more operations for each insertion. The bad performance of the variations for robin hood hashing and hopscotch hashing is mostly because they also have to update the acceleration bits.

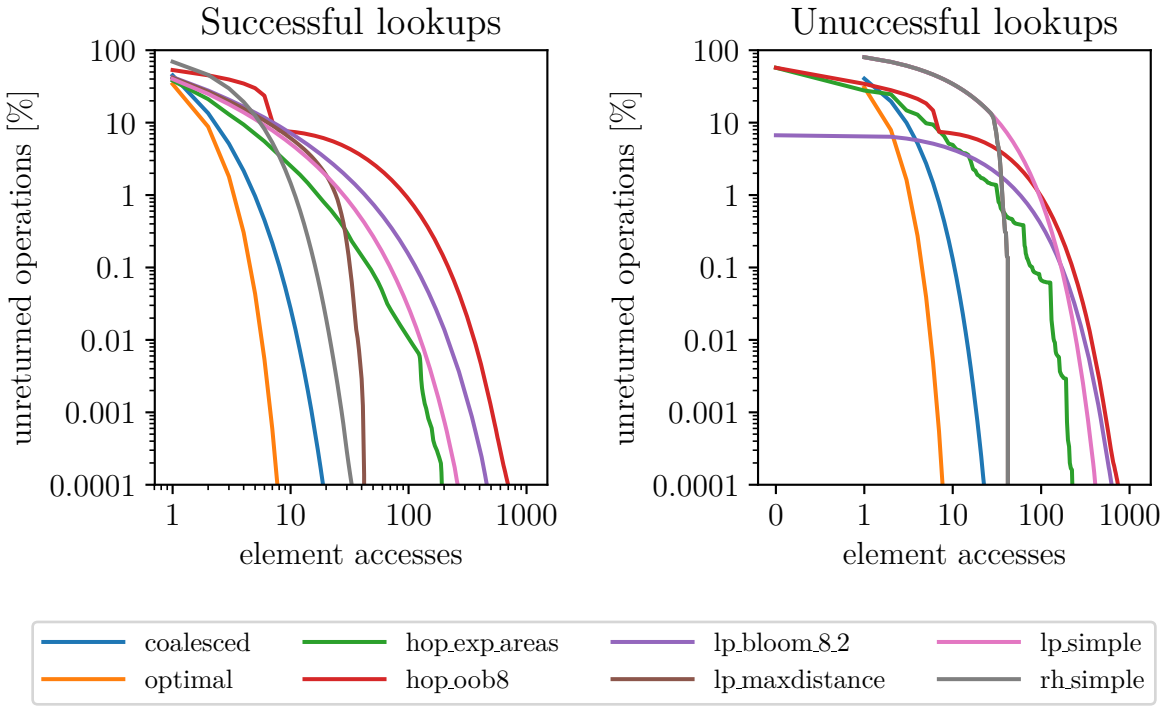

Figure 12: Percentage of lookups that require more than the given amount of element accesses before returning (with adjusted capacity of 80%). *Note:* The following hashing scheme variations are considered optimal (only scan though elements that are hashed to the same cell): hopscotch, rh_3bit, rh_sp_lh_ntb and coalesced_sc. The last two hashing scheme variations access at least one element before they can return.

Both implementations of standard coalesced hashing have the best performance for successful finds. With the variation that shortens chains coming out ahead, being up to 80% faster than simple linear probing, as it looks at fewer elements per lookup. However they both do not support an unadjusted load factor of 90%, making robin hood with startpointers and no additional hashing, which is otherwise slightly worse, the best viable option for this load factor. Our hopscotch variations that support this load factor thanks to using fewer acceleration bits show a similar throughput as linear probing. The robin hood 3Bit variation on the other hand has a worse performance as it accesses a

high amount of acceleration bits because of the high load factor. The bad performance of linear probing with bloom filter is expected, since it is optimized for unsuccessful lookups and has a lower capacity than simple linear probing.
Similar to successful lookups, coalesced hashing with shortened chains and robin hood hashing with startpointers and no additional hashing have the best unsuccessful lookup performance. One of the reasons these two perform so well is that they only access elements that are hashed to the same cell as the key that is looked for. Coalesced hashing without shortened chains is second tier as it has to look at more elements before it can come to a stop (Figure 12). Linear probing with a bloom filter allows up to 92% of lookups to return without accessing any elements (Figure 12), reducing the time of linear probings unsuccessful lookups by up 50%. However because of the cases where a lookup successfully passes the bloom filter, forcing the algorithm to iterate to the next empty cell, robin hood hashing with startpointer still performs better. Hopscotch with exponential areas has a similar time to linear probings bloom filter variation thanks to accessing fewer elements by only looking at specific areas in the hash table. Linear probing with is-Empty bit shows little improvement to regular linear probing compared to the previously mentioned hashing schemes because the amount of unsuccessful searches where the is-Empty bit was not set is low as a result of the high load factor. This forces many lookups to do the same search as if normal linear probing was used.
For deletions coalesced hashing with shortened chains and hopscotch hashing with exponential areas show by far the best performance as they do not rearranging multiple elements in the hash table. They are followed by hopscotch with exponential areas, which only hashes few amounts of elements, as well as coalesced hashing without shortened chains which reinserts the few remaining elements of the deleted elements chain. Hopscotch hashing with out of bounds bit and linear probing with bloom filter have the worst performance as they both hash multiple elements to rearrange the array and to update the acceleration bits.
Linear probing with maximum distance performs similarly to simple linear probing as the only addition is the check if the maximum distance is reached (Figure 3). The variance of lookups however is far better for the maximum distance variation, which can be seen in Figure 10 and 12.

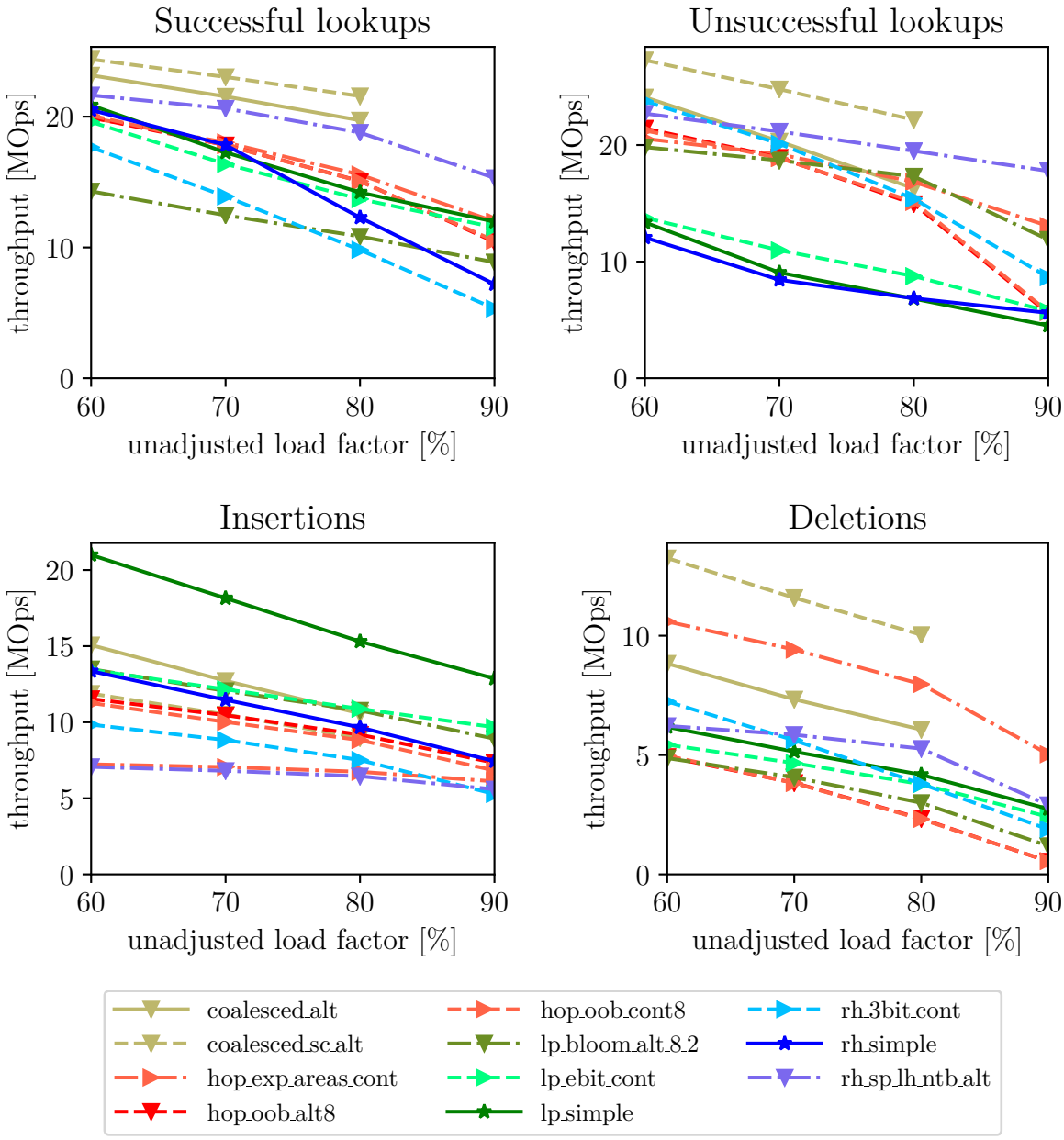

Figure 13: Throughput of different hashing schemes using adjusted capacities and a high unadjusted load factor.

Table 1: Table of advantages and disadvantages comparing all hash table types.

	Low load factor and unadjusted capacity				High load factor and adjusted capacity			
	insert	successful lookups	unsuccessful lookups	erase	insert	successful lookups	unsuccessful lookups	erase
lp_simple	+++	++	+	0	+++	++	-	+
lp_ebit_cont	+++	++	++	0	++	++	0	+
lp_bloom_8_2_alt	++	-	+++	-	++	0	++	0
lp_maxD	++	++	+	[4]	++	++	0	[4]
coalesced_alt	++	++	+++	++[3]	++[1]	+[1]	+[1]	+[1 3]
coalesced_sc_alt	+	++	+++	+++[3]	+[1]	++[1]	+++[1]	+++[1 3]
robin_hood	++	++	+	[4]	+	+	+	[4]
rh_sp_lh_ntb_alt	+	++	++	0	+	+++	+++	+
3bit_cont	+	+	+++	+	+	-	++	0
hop_simple_alt	+	++	++	+++	[2]	[2]	[2]	[2]
hop_oob_char8	+	++	++	0	+	0	+	0
hop_exp_areas_alt	0	++	++	++	0	0	++	+++

[1] does not support an unadjusted load factor of 90%
[2] does not support an unadjusted load factor of 80%
[3] operation reduces the longevity
[4] operation was not implemented

5 Future Work

The hashing schemes presented in this work have shown that the storing method of the acceleration data impacts its performance notably. However multiple variables of hash tables have been fixed in our tests and conclusions may change if these variables are changed. Element size is especially important as larger elements would allow for more per-cell data while affecting the hash tables total size less and accessing multiple elements could become less cache efficient.
Since we tested a wide array of different variations of hashing schemes, focusing on one variation might lead to better performances. For example, while we tested linear probing with a bloom filter for different bloom sizes and hashes per element, we can not guarantee that the used configurations are optimal. Counting bloom filters on the other hand were only mentioned but not implemented and could allow for faster deletions - notably if the size of the acceleration data does not matter.
Parallelization is another factor not included in this work that would be interesting for future works, especially for the performance of our hopscotch variations as hopscotch hashing was originally designed as concurrent data structure.
While the hash tables in this work had a fixed capacity, it is often the case that hash tables are resized during their lifetime. Creating a new hash table with a larger capacity and inserting all elements of the previous table one by one is possible, however it is far from ideal and an efficient growth algorithm could be implemented instead.
While we tried many approaches for per-cell data to improve hashing schemes, there are certainly more approaches possible, notably for other hashing schemes like multi hashing and quadratic or exponential probing. Some ideas for variations and configurations also had to be left for future research due to lack of time, these include:

- Linear probing with each cell storing the maximum distance the find operation has to traverse in order to find all elements that are hashed to the cell. This method could improve the performance of unsuccessful lookups, especially if the table has a high load factor.
- Grouped storage with other group sizes: these were mainly left untouched as we only used one element size resulting in three elements per group, to have each group in

exactly one cache line. Other element sizes or machines with different cache sizes or configurations might improve the performance. It could also be interesting to have groups with a size of two or three cache lines, or a size that allows the accelerations bits of one group to fill exactly one cache line.

Increasing the performance of iterating over the hash table could be another use for acceleration bits. For example, the 3Bit robin hood variation we presented could already be used to accelerate iterations by checking the is-occupied bit before accessing a cell. This would improve the cache efficiency if the load factor is low and the acceleration bits are stored in split storage.

6 Conclusion

In this work we explored the design space of using per-cell data to accelerate hashing schemes. While this idea was already present in hopscotch hashing we expanded it for the use with other hashing schemes. In addition, we presented different ways of storing the acceleration data and compared their performance for different hashing schemes.
Our benchmarks show that for hashing schemes that either access multiple sequential elements or the acceleration bits of multiple sequential cells, but not both, storing acceleration bits and elements in different arrays, which we called split storage, is the better option. While storing the acceleration data bit-by-bit or byte-by-byte can be slightly faster in some scenarios, using a continuous array proved to be an all-rounder for these hashing schemes. On the other hand hashing schemes that access a cells element and its acceleration data at the same time perform the best if alternating storage is used, which stores the two next to each other.
By specializing a hash table to its workload it is possible to increase the overall performance. This specialization can be done by using per-cell data. For example one variation of robin hood (rh_sp_lh_ntb) stores where the relevant elements for a lookup will be and jumps right to them. As a result it beats normal robin hood hashing, linear probing as well as hopscotch hashing in lookup and deletion time for high load factors with the drawback of increased allocated space and worse insert performance. As another example, if a workload mainly consists of negative lookups, it is possible to improve the performance with the help of a bloom filters, which decreases the negative lookup time of linear probing by up to 50%. By storing a bloom filter for each hash, which can guarantee that a searched element is not in the hash table, it is possible for most unsuccessful lookups to return without accessing a single element. To use these bloom filters the hash table sacrifices 6% of its capacity, reducing the performance of other operations.
Space usage can also be important for a hash table, and while hopscotch hashing already uses per-cell data it takes a lot of space. Our variations reduced the needed additional space by up to 75% and also allowed higher load factors than normal hopscotch at the cost of performance.

References

[1] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] P. Celis, P. Larson, and J. I. Munro. Robin hood hashing. pages 281–288, Oct 1985.

[4] Y. Collet. xxhash.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[6] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. pages 350–364, 2008.

[7] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[8] D. E. Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[9] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[10] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, Nov. 2015.

[11] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processing. *SIGCOMM Comput. Commun. Rev.*, 35(4):181–192, Aug. 2005.

[12] J. S. Vitter. Implementations for coalesced hashing. *Commun. ACM*, 25(12):911–926, Dec. 1982.