Karlsruher Institut für Technologie

Engineering SlickHash

Master Thesis von

Gerd Augsburg

An der KIT-Fakultät für Informatik
Institut für Theoretische Informatik, Algorithm Engineering

1. Prüfer/Prüferin:	Prof. Dr. rer. nat. Peter Sanders
2. Prüfer/Prüferin:	Prof. Dr. Thomas Bläsius
1. Betreuer/Betreuerin:	Dr. rer. nat. Florian Kurpicz

22. März 2024 – 22. Dezember 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 22.12.2024

. . . . . .

(Gerd Augsburg)

Zusammenfassung

Diese Arbeit behandelt die Implementierung und Optimierung einer neuen Art von Hashtabelle, die eine sehr effiziente Speichernutzung bietet bei gleichzeitig guter Performance. Die Slick-Hash genannte Tabelle speichert ihre Elemente in einer Sliding-Block Datenstruktur. Diese bietet adressierbare Blöcke an, die ihre Größe und Position den Bedürfnissen anpassen können und dennoch ein kontinuierlicher Speicherbereich ist, der effizient durchlaufen werden kann. Es werden verschieden Arten untersucht wie die dafür nötigen Metadaten verwaltet werden können und wie sie auf 9 Bit pro Block reduziert wurden. Da der Anpassbarkeit der Sliding-Blocks Grenzen gesetzt sind werden für die Hashtabelle weitere Speicherebenen verwendet. Es werden Implementierungsvarianten für diese Speicherebenen untersucht und gezeigt, dass auch hier Sliding-Blocks eine gute Wahl ist. Die entstandene Hashtabelle wird mit anderen Hashtabellen-Implementierungen verglichen. Dazu wird eine vereinheitlichte Schnittstelle für die Hashtabellen entworfen und dazu passende Leistungstests. Es wird gezeigt, dass Slick-Hash sehr Speichereffizient ist, bei den Ausführungszeiten seiner Operationen jedoch langsamer ist als die Alternativen. Es werden einige Optimierungen der Ausführungszeiten untersucht, die jedoch keine signifikanten Auswirkungen zeigten.

Inhaltsverzeichnis

1 Einführung

Diese Arbeit beschäftigt sich mit der Vervollständigung und Optimierung einer neuen Art von Hashtabelle, die eine sehr gute Speichernutzung verspricht ohne die Ausführungszeiten der Operationen auf der Hashtabelle zu sehr zu verlangsamen. Für eine erfolgreiche Optimierung müssen Schlüsselmetriken gemessen werden. Diese sollen auch mit anderen Hashtabellenimplementierungen verglichen werden um nicht nur die eigene Verbesserung zu erfassen sondern auch die Vor- und Nachteile gegenüber anderen Tabellenimplementierungen. Daraus ergeben sich drei Themenschwerpunkte. Zunächst muss für verschiedene Hashtabellen ein gemeinsamer Kern gefunden werden den man miteinander vergleichen kann. Für den Verglich müssen Metriken gefunden und erfasst werden die durch verschiedene Aufrufmuster – den $\mathrm{Workloads}$ – auf den Kern der Hashtabellen aussagekräftige Informationen über erwartetes reales Verhalten liefern. Als letzter Schwerpunkt wird, auf Basis dieser Vergleiche, anschließend die neue Hashtabelle optimiert.

1.1 Hashtabellen

Hashtabellen sind eine fundamentale Datenstruktur in der Informatik, die in vielen Anwendungen zum Einsatz kommt. Dadurch ergibt sich ein breiter Anforderungsraum der sich nicht durch eine einzelne Lösung vollständig abdecken lässt. Die bedeutendste Abwägung ist dabei die Geschwindigkeit der verschiedenen Operationen auf der Hashtabelle und ihre Speichernutzung. Schnelle Tabellenimplementierungen benötigen üblicherweise mehr Speicher.

Im Kern ist die Hashtabelle eine Abbildung von einem Schlüssel, dem Key, auf einen Wert, dem Value, zusammen als Eintrag oder Entry bezeichnet. Entsprechend besteht die Tabelle aus einer Menge $S \subseteq E = K \times V$ wobei jeder Key nur einmal vorkommen kann. Dazu wird der Key über eine Funktion auf einen Speicherbereich abgebildet in dem dann der Eintrag hinterlegt ist. Eine grundsätzliche Unterscheidung ist dabei ob die Hashtabelle statisch oder dynamisch ist. Bei einer statischen Hashtabelle sind die Schlüssel alle im Vorfeld bekannt und man kann eine Funktion für die Abbildung wählen, die eindeutig ist. Dadurch ist die Verwaltung des Speichers sehr einfach. Die Herausforderung besteht darin eine solche Abbildung zu finden. Diese Arbeit betrachtet nur dynamische Hashtabellen. Bei dynamischen Hashtabellen sind die Schlüssel nicht bekannt und man muss mit Kollisionen bei der Abbildung rechnen. Die Herausforderung besteht im effizienten Auflösen der Konflikte bei denen beide Einträge erhalten bleiben. Die grundlegenden Operationen einer dynamischen Hashtabelle sind:

- Hinzufügen neuer Einträge
- Testen auf das Vorhandensein von Schlüsseln
- Zugriff auf Werte über den Schlüssel
- Aktualisieren von Werten
- Löschen von Einträgen

In der Praxis wird dabei oft der Test auf das Vorhandensein von Schlüsseln mit den anderen Operationen kombiniert.

Den dynamischen Hashtabellen gemein ist, dass die Abbildungsfunktion von Schlüssel auf Speicherbereich über eine Hashfunktion geschieht. Die Hashfunktion $h(\cdot)$ reduziert den Wertebereich des Schlüssels auf einen definierten Raum $h(K) \rightarrow [0..m]$ [1]. Dabei ist das Ziel für Hashtabellen, dass dieser Raum möglichst gleichmäßig genutzt wird und Ähnlichkeiten im Schlüssel nicht zu ähnlichen Werten im Zielraum führen. Für diese Arbeit wird davon aus gegangen, dass dieser Wert des Zielraums zufällig und gleich verteilt ist. Es ist für praktische Hashfunktionen eine valide Annahme und es ist nicht Ziel Hashfunktionen zu vergleichen. Diese können jedoch einen großen Einfluss auf die Eigenschaften der Hashtabellen haben. Insbesondere wenn entweder die Berechnung der Hashfunktion sehr lange dauert oder die Hashfunktion zu schwach ist und die quasi zufällige Verteilung nicht mehr gegeben ist.

1.2 Bestehende Hashtabellen

Für den Vergleich der Slick-Hash-Implementierung wurden verbreitete und platzeffiziente Hashtabellen zusammen getragen. In der Umsetzung kommen zwei Programmiersprachen zum Einsatz: Rust und C++. Beide Sprachen haben eine Standard Hashtabelle. In C++ ist es $\mathrm{unordered\_map}$, die Rust Standard Hashtabelle heißt $\mathrm{HashMap}$ für eine bessere Unterscheidung im folgenden $\mathrm{StdHashMap}$ genannt. Eine Suche im zentralen Repository von Rust [2] ergab keine weiter relevante Hashtabelle. Fokus anderer Hashtabellen sind zusätzliche Eigenschaften wie Erhalt der Einfügereihenfolge oder Nebenläufigkeit. Implementierungen mit Fokus auf Speichereffizienz konnten nicht gefunden werden. Auf der C++ Seite gibt es kein zentrales Repository das durchsucht werden kann. Eine Sammlung und Vergleich von Hashtabellen wurde von Martin Leitner-Ankerl [6] erstellt. Sie diente als Vorauswahl um gute und platzeffiziente Implementierungen zu finden. Ausgewählt wurden $\mathrm{folly\ F14Map}$ von Facebook, $\mathrm{abseil\ flat\_hash\_map}$ von Google und $\mathrm{tsl\ sparse\_map}$. Zusätzlich wurden noch die platzeffizienten Implementierungen von Linear Probing ($\mathrm{prob\_linear}$ im weiterem $\mathrm{linear\_probing}$ genannt) und Cuckoo (cuckoo und $\mathrm{growing\_cuckoo}$) von DySECT [3] ausgewählt. Die ausgewählten Tabellen unterscheiden sich nicht nur in Implementierungungsdetails sondern auch fundamental in ihrer Strategie zur Auflösung von Konflikten bei der Speicher-Abbildungsfunktion.

Die Tabellen StdHashMap, F14Map, flat_hash_map und linear_probing basieren alle auf dem verbreitetem Prinzip des Linear-Probing [4]. Dabei wird eine erwartete Speicherstelle bestimmt und bei einem Konflikt linear weiter gesucht. Bei der Abfrage von Einträgen wird an der erwarteten Speicherstelle begonnen und die Elemente verglichen, bis das passende Element gefunden wurde oder man auf eine freie Stelle trifft, was bedeuted, dass das gesuchte Element nicht vorhanden ist. Bei einer vollen Tabelle kann eine nicht erfolgreiche Suche daher sehr lange dauern. Die erfolgreiche Suche findet ihr Element jedoch oft nahe der erwarteten Position. Neue Elemente werden beginnend von ihrer erwarteten Speicherstelle in die erste freie geschrieben. Es ist eine simple lineare Suche und eine Umstrukturierung bestehender Elemente ist nicht notwendig. Beim entfernen von Elementen muss jedoch geprüft werden ob nachvolgende Elemente vorgezogen werden müssen, damit die neu entstandene Lücke ein Auffinden nicht verhindert. Beim Linear-Probing ist die erfolglose Suche und das entfernen von Elementen tendenziell langsam und es profitiert sehr von vielen freien Speicherstellen. Daher reservieren die Tabellen StdHashMap, F14Map und flat_hash_map mehr Speicherstellen als angefordert und Fingerprints für die Elemente. Bei StdHashMap sind es $12,5\%$ zusätzliche Speicherstellen jedoch immer eine Potenz von zwei. Dadurch wird meist noch mehr zusäztlicher Speicher reserviert. Dazu speichert die StdHashMap für jedes Element ein Byte Fingerprint um die Suche zu beschleunigen. Der Einstellungsparameter ist die Anzahl der reservierten Speicherstellen wobei nur prob_linear nicht selbstständig zusätzliche Speicherstellen reserviert.

Die Tabelle unordered_map löst Konflikte mit einer verlinkten Liste auf. Bei der Suche wird über die Abbildungsfunktion die passende Liste ausgewählt, die anschließend durchsucht wird. Das sorgt für mindestens einen zusätzlichen Sprung im Speicher und einen weiteren für jedes Element in der Liste das durchlaufen werden muss. Zusätzlich wird Speicher für die Verweise der verlinkten Liste benötigt. Dafür wird kein Speicher für die Elemente unnötig reserviert. Beim Einfügen neuer Elemenet wird der passenden Liste das Element hinzugefügt und beim entfernen der Liste entnommen.

Die sparse_map hat eine Kombination aus Linear-Probing und dynamisch wachsenden Buckets. Der Zugriff erfolgt prinzipiell wie beim Linear-Probing, jedoch wird der Speicher für die Positionen der Abbildungsfunktion nich direkt reserviert. Der Zielraum wird unterteilt und für jeden Teilbereich ein dynamisch wachsender Vektor angelegt. Um von der erwarteten Position auf die tatsächliche Position im Vektor ab zu bilden wird ein Bitvektor verwendet. Dieser hat die Länge des Teilbereichs und enthält eine 1 für jede besetzte Speicherstelle. Die Position im Daten-Vektor entspricht dann der Summe der gesetzten Bits im Bitvektor bis zur angefragten Position. Dadurch benötigen leere Speicherstellen weniger Platz.

Die Tabellen cuckoo und growing_cuckoo sind beides Blocked-Cuckoo-Hashing [8][7] Tabellen. Der Speicher wird dabei in Blöcke unterteilt und es stehen mehrere Hashfunktionen für die Verteilung der Einträge zur Verfügung. Bei der Suche nach Einträgen wird die erste Hashfunktion verwendet und in dem entsprechend gefundenem Block nach dem Eintrag gesucht. Wenn der Eintrag nicht gefunden wird, werden noch der Reihe nach die Blöcke aus den anderen Hashfunktionen durchsucht. Wenn beim Einfügen im Block der ersten Hashfunktion kein Platz mehr ist, wird versucht die Elemente in ihren alternativen Blöcken unter

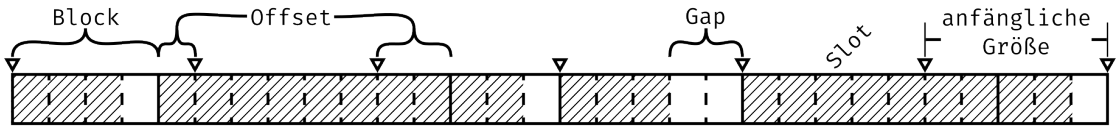

Abbildung 1.1: Bestandteile der Sliding-Blocks

zu bringen. Dabei werden nicht nur die alternative Blöcke des neuen Elements betrachtet sondern auch untersucht ob vorhandene Elemente in alternativen Blöcken untergebracht werden können. Diese Suche ist sehr aufwändig greift viel auf zufällig verteilte Blöcke zu und erfordert rehashing der Elemente um die alternativen Blöcke zu bestimmen. Beim Entfernen von einem Eintrag kann dieser ohne umbauten frei gegeben werden. Als Einstellungsparameter stehen die Blockgröße, die Anzahl der Hashfunktionen und die Gesamtzahl der reservierten Plätze für Einträge zur Verfügung.

1.3 Slick-Hash

Die neue Hashtabellenvariante Slick-Hash ist in „Sliding Block Hashing (Slick) – Basic Algorithmic Ideas“[5] beschrieben. Slick-Hash verwaltet die Einträge in Blöcken mit der Besonderheit, dass diese Blöcke ihre Größe ändern und verschoben werden können (**sli**ding blo**ck**). Die Implementierung folgt anfangs dem ursprünglichem Entwurf und versucht darauf aufbauend Optimierungen vor zu nehmen. Der ursprüngliche Entwurf sieht zwei Speicherbereiche vor. Der Hauptbereich der aus den Sliding-Blocks besteht und einen sekundären Speicherbereich der $\mathrm{Backyard}$ genannt wird. Auf die Implementierung des $\mathrm{Backyard}$ wird in der ursprünglichen Beschreibung nicht näher eingegangen. Es wird jedoch gezeigt, dass eine Implementierung mit einer beliebigen Hashtabelle als $\mathrm{Backyard}$ funktioniert. Die Sliding-Blocks des Hauptbereichs ist in Abbildung 1.1 zu sehen. Sie besteht besteht aus Slots S aufgeteilt in Blöcken der anfänglichen Größe B. Jeder Slot enthält eine Kombination aus Key und Value. Zu jedem Block gibt es zusätzliche Metadaten M bestehend aus einem Offset o_i, der Menge der ungenutzten Slots (Gap) g_i und einem $\mathrm{Threshold}$ t_i. Aus dem Schlüssel eines Elements e werden durch Hashfunktionen zwei Werte abgeleitet. Zum einen ein Blockindex i des Hauptbereichs und zum anderen ein $\mathrm{Threshold}$ t_e.

Für die Suche nach einem Eintrag, wird aus dem Schlüssel der Blockindex und der $\mathrm{Threshold}$ gebildet. Wenn $t_e < t_i$ ist der Eintrag im $\mathrm{Backyard}$ ansonsten im Hauptbereich. Initial ist der Block i gegeben durch $b_i = S[iB,\ iB+B)$[1]. Über die Metadaten werden Abweichungen von diesem Startzustand nachverfolgt. Der Block der den Eintrag enthält ist entsprechend $b_i = S[iB + o_i,\ (i+1)B + o_{i+1} - g_i)$. Diese Slots werden durchsucht um den Eintrag mit passendem Schlüssel zu finden. Da der Block klar begrenzt ist kann im vergleich zum $\mathrm{Linear\text{-}Probing}$ die erfolglose Suche auch bei einer sehr vollen Tabelle schnell beendet werden.

Beim Einfügen und Löschen werden die Invarianten aufrecht erhalten. Beim Einfügen wird der Blockindex und $\mathrm{Threshold}$ des Elements bestimmt. Ist $t_e < t_i$ wird das Element ans

[1] Zu beachten ist hier die exklusive Range $[a, b)$ von a (inklusive) bis b (exklusiv)

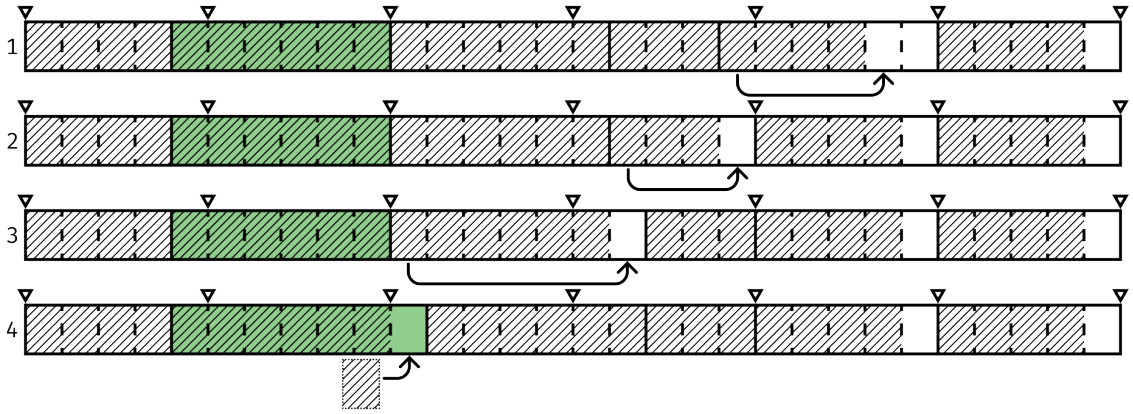

Abbildung 1.2: Sliding-Blocks Einfügen mit Verschiebung

Backyard übergeben. Ansonsten wird versucht es in den Block ein zu fügen. Gibt es noch freie Slots im Block, also wenn $g_i > 0$, kann das Element in den Slot $S[(i+1)B + o_{i+1} - g_i]$ geschrieben werden mit anschließender Reduktion des Gap $g_i \leftarrow g_i - 1$. Ist im Block kein Slot mehr frei wird in der Nachbarschaft nach einem freien Slot gesucht. Dieser freie Slot wird zum Zielblock verschoben dargestellt in Abbildung 1.2. In jedem Schritt wird in einem Block auf dem Weg ein Element verschoben und der Offset angepasst. Die Suche findet ausschließlich auf den Metadaten statt. Aus den Metadaten ist auch bekannt welcher Slot frei ist. Im Vergleich zu Cuckoo Tabellen muss dieser nicht im Block gesucht werden. Zusätzlich durchläuft die Suche nach einer freien Stelle die Metadaten linear und vorhersehbar. Auch die Verschiebung der Einträge findet in der Nachbarschaft des Zielblocks statt. Dadurch werden weniger Cache-Misses erwartet als bei Cuckoo Tabellen.

Die Suche sowie die Offsets und Blockgröße sind begrenzt. Dadurch kann es sein, dass kein freier Slot gefunden wird. In dem Fall muss mindestens ein Element ins Backyard verschoben werden. Es wird der minimale Threshold t_{min} von den Elementen im Block und dem neuen Element gesucht. Alle Elemente mit diesem Threshold werden ins Backyard verschoben und der Threshold vom Block wird zu $t_i \leftarrow t_{\text{min}} + 1$. Für die übrigen Elemente ist jetzt genug Platz im Block.

Beim Löschen von Elementen wird wieder am Threshold unterschieden ob das Element im Hauptblock ist. Dort kann es durch das letzte Element im Block ersetzt werden mit anschließender Erhöhung des Gap $g_i \leftarrow g_i + 1$. Für das Zurückführen von Elementen im Backyard in den Hauptblock und das zurücksetzen des Threshold ist ein gebündeltes Vorgehen vorgeschlagen. Wenn im Hauptbereich genügen Platz ist werden alle Thresholds zurück gesetzt und die Elemente des Backyard eingefügt. Dabei können auch wieder Elemente im Backyard landen.

1.4 Implementierung

Die Implementierung fand in der Programmiersprache Rust statt. Im Vergleich zu den Hauptalternativen C und C++ ist Rust eine sehr junge Sprache. Für C++ spricht, dass die

meisten effizienten Hashtabellen bisher mit dieser Sprache entwickelt wurden. Diese können jedoch auch in Rust verwendet werden und beim Autor besteht mehr Erfahrung mit der Sprache Rust. Für Rust spricht auch, dass existierende Bibliotheken einem Rust-Projekt einfacher hinzugefügt werden können, was die Implementierung der Auswertung stark vereinfacht. Ein weiterer Anreiz ist der bisherige Mangel an platzeffizient Hashtabellen in Rust. So kann auch eine aufs wesentliche reduzierte platzeffiziente Tabelle hier einen Beitrag zum Rust Ökosystem leisten.

Bei der Implementierung kommen verschiedene ganzzahlige Typen zum Einsatz. In Rust sind diese direkt mit ihrem Vorzeichen und Bitlänge bezeichnet. So sind u8, u16, u32, u64 und u128 vorzeichenlose Ganzzahlen mit 8, 16, etc. Bit Länge. Die Version mit Vorzeichen ist i8, i16, etc. Eine Besonderheit sind usize und isize, die jeweils die Länge der Adressierung der Maschine haben. Es kamen während der Entwicklung jedoch nur 64 Bit Maschinen zum Einsatz. Zusammengesetzte Typen werden in Rust struct genannt. Pointer existieren in Rust, bevorzugt werden jedoch Referenzen. Dabei ist &T eine Referenz zu einem struct vom Typ T. Diese Referenzen bieten nur Lesezugriff, dafür können beliebig viele gleichzeitig lesen. Um Veränderungen am struct hinter der Referenz vornehmen zu können muss diese mutable sein. Von den veränderbaren Referenzen &mut T darf jedoch immer nur eine existieren. Auch keine weiteren unveränderlichen. Ein slice beschreibt in Rust eine Referenz zu einem kontinuierlichem Speicherbereich. Dabei ist &[T] oder &mut [T] bestimmt durch einen Startpunkt und einer Länge. Kontinuierlicher Speicher dynamischer Größe für Elemente vom Typ E werden in Rust üblicherweise in einem Vec<E> verwaltet. Dabei kann auf den Vektor v auch mit einer Range zugegriffen werden. Mit v[a..b] bekommt man einen slice von a bis exklusive b. Für ein Zugriff inklusive b wird v[a..=b] verwendet.

2 Einheitliche Schnittstelle

Es sollen verschiedenen Hashtabellen aus C++ und Rust miteinander verglichen werden. Um das zu ermöglichen wurden die Operationen der Hashtabellen untersucht und unter einer gemeinsamen Schnittstelle vereinheitlicht zu sehen in Listing 1. Die Zeilen 1–9 beschreiben dabei die Hashtabelle und die Zeilen 10–14 den Rückgabewert der Einfügeoperation. Damit können verschiedene Rückgabewerte einheitlich behandelt werden ohne sie in ein anderes Datenformat umwandeln zu müssen solange der tatsächliche Rückgabewert den trait aus den Zeilen 10–14 implementiert. Die Zeilen 2 und 3 wenden diesen trait für die Hashtabelle an. Neue Tabellen können durch with_capacity aus Zeile 4 mit einer erwarteten Anzahl an Einträgen erstellt werden. Eventuelle Überallokation ist als Parameter den einzelnen Tabellen überlassen.

Die grundlegenden Operationen aus Kapitel 1.1 werden über die Methoden in den Zeilen 5–8 erfüllt. Die Methoden get und get_mut aus Zeile 5 und 6 erfüllen den Test auf das Vorhandensein von Schlüsseln und den Zugriff auf die Werte. Sie unterscheiden sich nur in der Eigenschaft ob der Wert verändert werden darf. Rust besitzt keinen allgemeinen Wert für das Fehlen eines Wertes, bekannt als null. Für einen optionalen Wert wird in Rust dieser in einem Option Typ verpackt. Ein Option<&Value> hat die die Varianten Some(&Value) für das Vorhandensein eines Wertes wenn der Schlüssel gefunden wurde und None für das Fehlen eines Wertes wenn der Schlüssel nicht vorhanden ist. In C++ bekommt man einen Verweis auf den Wert, den man entweder zunächst noch überprüfen muss ob er valide ist oder man bekommt zusätzlich einen Boolean mit zurück der anzeigt ob der Verweis gültig ist. Beides lässt sich gut in ein Option übertragen.

```
1  pub trait HashTable<Key: Hash + Eq, Value> {
2      type Insertion<'v>: Insertion<Value>
3      where Self: 'v;
4      fn with_capacity(capacity: impl Capacity) -> Self;
5      fn get(&self, key: &Key) -> Option<&Value>;
6      fn get_mut(&mut self, key: &Key) -> Option<&mut Value>;
7      fn try_insert(&mut self, key: Key, value: Value) -> Self::Insertion<'_>;
8      fn drop_entry(&mut self, key: &Key) -> bool;
9  }
10 pub trait Insertion<Value> {
11     fn is_inserted(&self) -> bool;
12     fn as_ref(&self) -> &Value;
13     fn as_mut(&mut self) -> &mut Value;
14 }
```

Listing 1: Vereinheitlichte Hashtabellen Schnittstelle

Das entfernen von Einträgen gibt in Rust üblicher weise den entfernten Eintrag als Option zurück, wenn dieser gefunden werden konnte. In den C++ Implementierungen ist für das entfernen jedoch nur ein Boolean üblich, der anzeigt ob ein Eintrag entfernt wurde. Daher gibt drop_entry aus Zeile 8 den Eintrag direkt frei und ein Marker zurück, ob die Tabelle verändert wurde.

Die größte Varianz in der Umsetzung gibt es beim Einfügen von Einträgen. Sie unterscheiden sich im Umgang mit vorhandenen Schlüsseln. Bei allen Varianten wird auch beim Einfügen zunächst geprüft ob der Schlüssel bereits vorhanden ist. Wenn schon ein Eintrag mit passendem Schlüssel vorhanden ist gibt es grundsätzlich die beiden Varianten, das Einfügen ab zu brechen oder den vorhandenen Wert zu aktualisieren. In der Vereinheitlichten Version wurde sich dafür entschieden das Einfügen ab zu brechen. Als Ergebnis von try_insert aus Zeile 7 bekommt man ein Objekt, dass anzeigt ob der Wert eingefügt wurde und Zugriff auf entweder den neu eingefügten oder den vorhandenen Wert gibt. Dieses Zugriffsmuster ließ sich bei allen Hashtabellen abbilden.

3 Hashtabellen Benchmark Implementierung

Dieses Kapitel betrachtet die Implementierung der Hashtabellen Benchmarks. Die Herausforderung dabei ist die zuverlässige Erfassung von Metriken und die Erfassung der richtigen Metriken. Die einheitliche Schnittstelle aus Kapitel 2 erlaubt Benchmarks unabhängig von der Tabellenimplementierung sodass bei der Implementierung keine weiteren Annahmen über die konkrete Hashtabelle gemacht werden.

3.1 Metriken

Um den Erfolg der verschiedenen Optimierungen an der Slick-Hash Implementierung zu verifizieren wurden umfangreiche Leistungsmessungen durchgeführt. Die beiden wichtigsten Metriken sind dabei die Ausführungszeit der Operationen und die Speichernutzung der Hashtabelle. Die Ausführungszeit lässt sich relativ leicht über die Systemzeit messen. Die Schwierigkeit besteht darin, dass der Zustand des Systems großen Einfluss auf die Ausführungszeit hat. Insbesondere leere Caches und runter getaktete CPUs erhöhen signifikant die Ausführungszeit. Um dem entgegen zu wirken werden die gleichen Operationen die gemessen werden sollen vor der Messung einige Sekunden lang ausgeführt. Zufällige Kontextwechsel des Betriebssystems können die Ausführungszeit auch erhöhen. Daher werden möglichst viele Messungen durchgeführt und darüber gemittelt. Die Operationen auf Hashtabellen haben zufällige Komponenten, sodass die Ausführungszeit nicht immer gleich ist. Das macht die Bildung eines Mittels zusätzlich notwendig. Verwendet wird das arithmetische Mittel.

$$\mathsf{avg}(o) = \frac{\sum_{i=0}^{n} t(o)_i}{n}$$

Mit der Operation o, der Ausführungszeit für die Operation $t(o)$ und der Anzahl der Operationen n. Bei ausreichend großem n kann man mit dem Wert gute Vorhersagen zur Ausführungszeit von Programmen treffen bei denen die Anzahl der Operationen bekannt ist.

Die Speichernutzung variiert kaum über den Systemzustand, ist dafür aber schwerer zu messen. Es gibt zwei Ansätze für die Messung der Speichernutzung. Zum einen kann das Programm selber zählen wie viel Speicher es beim Betriebssystem anfragt oder man ruft die Menge des zugeordneten Speichers beim Betriebssystem ab. Die Zählung über das Programm wäre zu bevorzugen, da es die bessere Auflösung verspricht und ohne Zeitverzug

aktualisiert wird. Da Rust und C++ jedoch eigene vorgehen bei der Speicherverwaltung haben konnte kein zuverlässiger Weg gefunden werden beides zu zählen. Daher blieb nur der Weg über das Betriebssystem den zugeordneten Speicher ab zu fragen. Der Nachteil ist, dass das Betriebssystem den Speicher in Pages verwaltet und immer nur ganze Pages zuordnet. Das Entspricht auf den verwendeten Systemen einer Auflösung von 4 kiB. Dazu kommt, dass die Speicherverwaltung versucht den Programmablauf zu beschleunigen und Speicher der nur kurzzeitig nicht mehr gebraucht wird nicht frei gibt. Bei ausreichend großen Tabellen ist die verringerte Auflösung kein Problem. Die verzögerte Freigabe führt jedoch dazu, dass eine kurzzeitig verringerte Speichernutzung nicht zuverlässig gemessen werden kann. Für die Gestaltung der Benchmarks führt das dazu, dass möglichst nur eine steigende Nutzung statt findet. Dadurch müssen wiederholte Durchläufe je in einem neuen Prozess gestartet werden.

Für den Vergleich über mehrere Größenordnungen sind absolute Werte bei der Speichernutzung nicht sinnvoll. Um die Unterschiede der Tabellen stärker hervor zu heben wird auch der notwendige Speicher für die Einträge vom gesamten Verbrauch abgezogen. Als Wert für die Gesamtkapazität wird die bei der Erstellung der Tabelle angefragte Kapazität verwendet. Einige Tabellen reservieren mehr und könnten noch mehr Elemente ohne weitere Speichernutzung aufnehmen. Das zu beachten würde den Vergleich aber sehr erschweren. Es ergibt sich die Speichernutzung als Bits pro Eintrag $m = (m_g - m_p - n * m_e)/n$ mit m_g der Gesamtnutzung der vom Betriebssystem erfragt wird, m_p der Speichernutzung vom Prozess entspricht der Gesamtnutzung bevor begonnen wird die Hashtabelle zu erstellen, n die Anzahl der Einträge und m_e die Bits für einen Eintrag. Auch für die Bestimmung des Füllgrads wird die angefragte Kapazität verwendet und nicht die tatsächliche.

3.2 Operationen

Je nach Anwendungsfeld der Hashtabelle sind andere Operationen wichtig. Daher wird bei den Messungen nach den Operationen unterschieden, sodass man passend für das Anwendungsfeld die Gewichtung wählen kann. Die Methodennamen aus Kapitel 2 richten sich nach den Konventionen im Rust Umfeld. Für die Operationen losgelöst von der Implementierung sind jedoch andere Namen üblich. Die get Methoden werden find oder lookup genannt. Bei try_insert ist schlicht insert üblich. Die Methode drop_entry ist sehr Rust spezifisch, da drop in Rust ein Objekt frei gibt und den Destruktor auf ruft. Die üblichen Namen für die Operation sind erase, remove und delete. Alle die Operationen find, insert und delete können erfolgreich oder nicht erfolgreich sein. Bei allen getesteten Hashtabellen entsprach das nicht erfolgreiche insert einem erfolgreichem find und das nicht erfolgreiche delete einem nicht erfolgreichem find. Für eine Bewertung der Hashtabelle reicht es entsprechd aus, wenn man sich auf vier Operationen konzentriert:

- find
- unseccessful_find
- insert
- delete

3.3 Workload

Bei den Eingaben für die Leistungstests ist es die Herausforderung kontrolliert Schlüssel zu generieren, die je nach Operation in der Tabelle bereits vorhanden oder nicht vorhanden sind. Ein einfacher Pseudozufallsgenerator funktioniert dabei nur bei relativ kleinen Eingabegrößen. Die Idee ist, ausgehend von einem seed zufällige Schlüssel zu generieren. Für neue Schlüssel generiert man den nächsten Wert des Pseudozufallsgenerators und für bestehende beginnt man mit dem gleichen seed von vorne. Zum einen werden dabei die Schlüssel immer in der gleichen Reihenfolge durchlaufen, was unerwünscht ist, zum anderen braucht man einen sehr großen Schlüssel Raum um einen Kollisionsfreien seed zu finden. Geht man beim Pseudozufallsgenerator davon aus, dass neue Werte Zufällig sind, ist man bei einer Version des Geburtstagsparadoxons. Es reichen 23 Personen aus um die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, größer werden zu lassen als 50%. Genauso reichen auch verhältnismäßig wenige Schlüssel aus um eine Kollision wahrscheinlich werden zu lassen. Die Wahrscheinlichkeit für n Konfliktfreie Schlüssel aus einem Universum U mit $u = |U|$ ist bei einer gleichmäßigen zufälligen Verteilung:

$$p(\mathsf{konfliktfrei}) = \prod_{i=1}^{n} 1 - \frac{i}{u}$$

Als Schlüssel bei allen Leistungstest wurden $\mathrm{u64}$ gewählt. Schon bei $n = 2^{33}$ ist die Wahrscheinlichkeit eine Konfliktfreie Verteilung zu bekommen kleiner als 50%. Da Eingabegrößen von $n = 2^{35}$ vorkommen musste ein anderer Weg gefunden werden um zuverlässig neue Werte zu erzeugen. Die einfachste Variante ist das Hochzählen. Etwas mehr Varianz kann man bekommen indem man wiederholt eine Zahl überlaufend addiert, die coprime zum Schlüsselraum ist. Beides erzeugt jedoch klare Muster. Um nicht die Qualität der Hashfunktionen zu testen, sondern die Effizienz der Hashtabellen, ist es Ziel möglichst zufällig wirkende Schlüssel zu verwenden. Eine Bibliothek, die das sehr gut erreicht ist $\mathrm{rand\text{-}unique}$[9]. Sie erzeugt aus einem seed eine Sequenz ohne Wiederholungen, bei der jedes Element das noch nicht vor kam in etwa die gleiche Wahrscheinlichkeit hat als nächstes zu kommen. Sie ist zudem indizierbar, sodass der Aufruf bereits erzeugter Schlüssel über einen permutierten Index erfolgen kann.

Für das alternierende hinzufügen und entfernen von Einträgen wird eine Datenstruktur benötigt die verfolgt welche Elemente hinzugefügt oder entfernt wurden. Die Speicherung der validen Schlüssel in einer separaten Liste ist dabei ungeeignet. Tests haben gezeigt, dass der Zugriff auf die Liste die späteren Operationen auf der Hashtabelle negativ beeinflusst. Vermutlich verdrängt der Zugriff auf die Liste andere Teile aus den Caches. Der erfolgreiche find Zugriff auf eine Hashtabelle war langsamer als der erfolglose, wenn der Schlüssel aus einer Liste ausgewählt wurde. Wenn der Schlüssel für ein erfolgreiches find auf einem anderen Weg erzeugt wird ist der erfolgreiche Zugriff schneller als der erfolglose, was auch den Erwartungen entspricht, da der erfolglose alle vorhandenen Einträge prüft während der erfolgreiche bei einem Treffer nicht weiter suchen muss.

Neben der Optimierung der Slick-Hash Implementierung dienen die Vergleichstest dazu das Anwendungsfeld für Slick-Hash zu ermitteln. Dazu wurden die Implementierungen früh

mit anderen Hashtabellen, unter Berücksichtigung verschiedener Füllgrade und Anzahl der Elemente verglichen. Einen starken Einfluss auf die Ausführungszeiten hat der Zugriff auf Daten im Speicher. Slick-Hash hat hier mit seinem zusätzlichem Zugriff auf Metainformationen einen Nachteil. Dafür wurde der Speicher effizienter genutzt und verhalten bei sehr vollen Tabellen war viel versprechend. Daher konzentriert sich die Entwicklung erst mal auf diesen Vorteil.

Der erste Benchmark um einen Eindruck von den Eigenschaften der Hashtabellen zu bekommen ist die vollständige Befüllung. Bei der Datenerfassung wird jedoch nicht nur das Endergebnis aus gegeben sondern auch Zwischenwerte. Einfügezeit und Speichernutzung sollte abhängig vom Füllgrad betrachtet werden können. Als Auflösung ist etwa $1‰$ vorgesehen. Dazu werden die Ergebnisse in Blöcken zusammengefasst. Diese sollten jedoch nicht zu klein sein um die Varianz bei den Operationen aus zu gleichen und wurden auch nach oben beschränkt, was bei großen Tabellen eine noch höhere Auflösung erlaubt. Dazu werden die Einfügeoperationen durch gezählt und bei einem Vielfachen von entsprechend gewähltem s die Messergebnisse raus geschrieben. Jedoch soll der Prozess nicht von Modulo-Berechnungen verlangsamt werden. Daher wurde eine Bitmaske gewählt mit $b_s = \max(\min(32, 2^{\lceil \log_2(n/2000) \rceil}), 2^{14}) - 1$. Immer wenn $n \& b_s = b_s$ wird die Summe der Ausführungszeiten, die Anzahl und die Speichernutzung gespeichert.

Einmaliges Befüllen von Hashtabellen, Abfragen darauf und anschließendes Löschen haben den Vorteil, dass man leicht von einem definierten Zustand aus starten kann, der leeren Tabelle. Ein offener Designpunkt bei Slick-Hash war jedoch wie die Tabelle Löschungen so behandelt, dass nicht schlussendlich alle Einträge im Sekundärspeicher sind. Dafür wurde ein Test entworfen, der die Tabelle mehrfach befüllt und leert und währenddessen das Zeitverhalten und die Speichernutzung verfolgt. Der $\mathrm{Timeline}$ genannte Test hat vier Richtungswechselpunkte bei 25%, 50%, 75% und 100%. Zunächst wird die Tabelle bis 75% gefüllt, dann einige male bis 25% geleert und wieder bis 75% befüllt. Anschließend wird sie bis 100% befüllt und einige male bis 50% geleert und anschließend wieder befüllt. Nach jeder Änderung der Tabelle werden in zufälliger Reihenfolge fünf erfolgreiche und nicht erfolgreiche find aufrufe durchgeführt. Nach Änderungen die etwa $1‰$ der Gesamtkapazität entspricht oder bei Richtungswechseln werden die ermittelten Messungen gesichert. Wobei ein Richtungswechsel der Übergang von Befüllen zu Leeren oder umgekehrt ist. Die Herausforderung bestand darin nach zu verfolgen welche Schlüssel bereits verwendet wurden. Es kam die $\mathrm{rand\text{-}unique}$ Sequenz zum Einsatz. Da die Sequenz indizierbar ist kann der maximal verwendete Index nachverfolgt werden. Bei einem neuen Eintrag wird dieser hochgezählt. Um einen nicht vorhandenen Schlüssel zu erhalten kann ein beliebiger Index verwendet werden, der größer als das bisherige Maximum ist. Für das Löschen von Einträgen wird der Sequenzindex bei jedem Übergang von Füllen zu Leeren unterteilt. Für jeden Teil kann anschließend die Sequenz noch einmal durchlaufen werden. Für eine zusätzliche Varianz geschieht auch das in permutierter Form da die Grenzen der im Teil enthaltenen Indices bekannt ist. Um ein Element zu löschen wird ein zufälliger Teil ausgewählt und dort der nächste wiederholte Wert. Einen vorhandenen Schlüssel für ein erfolgreiches find erhält man aus einem zufälligem Teil und dort einem zufälligem noch nicht gelöschtem Wert.

Es zeigte sich, dass Slick-Hash besonders bei hohen Füllgraden viel versprechend ist. Zur genaueren Untersuchung dieses Bereichs und für die Optimierungen wurde ein weiterer Test entworfen. Dieser konzentriert sich auf volle Tabellen. Dabei wird die Tabelle zunächst auf 98% ihrer Kapazität gefüllt. Die letzten 2% der insert bis zur vollen Tabelle werden gemessen. Anschließend werden eine Reihe von erfolgreichen und nicht erfolgreichen find durchgeführt. Zuletzt werden noch wieder 2% entfernt und die delete gemessen.

4 Slick-Hash Implementierung

In diesem Kapitel wird auf die Implementierung von Slick-Hash näher ein gegangen. Beschrieben wird die Zerlegung in Teilprobleme die Anschließend optimiert werden können. Verschiedene getestete Varianten werden beschrieben so wie die angewendeten Optimierungen. Implementiert wurde für moderne x86_64 Prozessoren.

4.1 Sliding Blocks

Der Kern von Slick-Hash besteht aus den Sliding-Blocks. Die Sliding-Blocks Datenstruktur ist ein kontinuierlicher Speicherbereich, der in Blöcke unterteilt ist. Diese Blöcke können ihre Größe ändern und innerhalb des kontinuierlichen Speichers verschoben werden. Um die variablen Blockanfänge und Größen zu verfolgen werden für jeden Block Metadaten benötigt. Die Blöcke bestehen aus Slots, Speicherbereiche in die ein Eintrag hinterlegt werden kann. Jeder Block hat eine initiale Größe B, $\mathrm{Blocksize}$ genannt, daraus ergibt sich mit dem Index des Blocks i der erwartete Anfang des Blocks $bs_i = iB$. Die Abweichung von dem erwartetem Anfang ist der Offset o_i. Da die Blöcke nicht alle voll sind muss noch nachverfolgt werden wie viele Slots frei sind. Der freie Bereich von einem Block ist der Gap g_i. Zusätzlich sieht die Implementierung von Slick-Hash noch ein weiteres Metadatum für jeden Block vor. Der $\mathrm{Threshold}$ t_i zeigt an ob ein Eintrag im Block sein kann, hat für die Sliding-Blocks jedoch keine Bedeutung. Die $\mathrm{Blocksize}$ ist implizit verwendet und nimmt keine extra Speicher pro Block ein. Dargestellt wird sie in der Implementierung durch einen konstanten Wert vom Typ usize. Sinnvolle Werte liegen jedoch zwischen 4 und 512 mit 32 als Wert, der sich mit sehr guten Balance zwischen den Anforderungen bewährt hat. Die reale Blockgröße B' kann sich durch Verschiebungen ändern, ist jedoch durch die Art der Darstellung der Metadaten beschränkt. Technisch ist maximale Blockgröße $\hat{B}$ gegeben durch $\hat{B} = \min\left(|o_{\min}| + B + o_{\max}, g_{\max}\right)$. Die Blockgröße wird daher auch noch durch die darstellbaren Werte vom Gap beschränkt.

Für die Implementierung von Slick-Hash werden von den Sliding-Blocks drei Operationen benötigt, die möglichst effizient gestaltet werden. Für das find muss über die Einträge im Block iteriert werden können. Elemente müssen für das insert in einen Block eingefügt und dafür falls notwendig andere Blöcke verschoben werden können. Beim delete müssen Elemente aus einem Block entfernt werden.

Die Iteration über einen Block funktioniert sehr einfach. Die Elemente vom Typ E liegen in einem kontinuierlichem Speicherbereichs s, dessen Grenzen über die Metadaten bekannt sind. Die Blöcke werden von vorne befüllt, sodass der befüllte Block vom Startpunkt des Blocks

Block b_{free} mit $free < i$

$g_{free} \leftarrow g_{free} - 1$
$n \leftarrow free + 1$
$o_n \leftarrow o_n - 1$
$free\_slot\_ref \leftarrow \&s[n * B + o_n]$
while $n < i$ **do**
  $n \leftarrow n + 1$
  $o_n \leftarrow o_n - 1$
  $src\_slot\_ref \leftarrow \&s[n * B + o_n]$
  $*free\_slot\_ref \leftarrow *src\_slot\_ref$
  $free\_slot\_ref \leftarrow src\_slot\_ref$
end while
Free Slot: $free\_slot\_ref$

Block b_{free} mit $free > i$

$free\_slot\_ref \leftarrow \&s[(free + 1) * B + o_{free+1} - g_{free}]$
$g_{free} \leftarrow g_{free} - 1$
$n \leftarrow free$
while $n > i$ **do**
  $src\_slot\_ref \leftarrow s[n * B + o_n]$
  $*free\_slot\_ref \leftarrow *src\_slot\_ref$
  $free\_slot\_ref \leftarrow src\_slot\_ref$
  $o_n \leftarrow o_n + 1$
  $n \leftarrow n - 1$
end while
Free Slot: $free\_slot\_ref$

Listing 2: Verschiebevorgang der Slots für einen Freien Slot in b_{free}

bis zum Startpunkt des nächsten Blocks geht, abzüglich des Gap. Der Zugriff funktioniert dadurch wie im ursprünglichen Entwurf aus Kapitel 1.3. Mit $s[(iB+o_i)..((i+1)B+o_{i+1}-g_i)]$ bekommt man einen slice der genau die belegten Slots abdeckt. Über slices kann man anschießend sehr einfach iterieren.

Beim Einfügen in einen Block, kann man drei Fälle unterscheiden. Entweder ist im Zielblock ein Slot frei, in der Nachbarschaft lässt sich ein freier Slot finden oder es kann kein Slot gefunden werden. Wenn im Zielblock noch freie Slots sind ist das Einfügen ein schreiben auf in den ersten freien Slot $s[(i+1)B + o_{i+1} - g_i]$ und eine Verringerung des Gap $g_i = g_i - 1$. Sollte im Zielblock kein Slot mehr frei sein wird in der Nachbarschaft nach einem freien Slot gesucht. Für diese Suche werden nur die Metadaten benötigt. Der Gap von jedem Block zeigt ob dort noch ein freier Slot ist. Gesucht wird $g_s > 0$ bei dem $|i - s|$ minimal ist. Eine weitere Einschränkung ist, dass die bevorstehende Verschiebung die Grenzen des Offsets von jedem betroffenen Block nicht überschreiten darf. Anschließend muss pro Block ein Element verschoben und die Offsets angepasst werden, abhängig davon ob der freie Slot bei einem geringerem Index oder bei einem höheren gefunden wurde. Ausgehend von dem Block mit freiem Slot kann man ausnutzen, dass alle anderen betrachteten Blöcke einen Gap von 0 haben. Den Verschiebevorgang in beide Richtungen kann man im Listing 2 sehen. Sie unterscheiden sich in der Reihenfolge wann die Daten verschoben werden und wann die Metadaten angepasst werden. Beide Varianten kommen jedoch ohne Zwischenspeicher aus und jeder Schritt benötigt nur genau eine Verschiebung. Bei der Implementierung muss noch beachtet werden, dass es durch Verschiebungen möglich ist, dass Quelle (src) und Ziel (free) der gleiche Slot sein können. Entweder muss das vorher getestet werden oder man muss eine Kopieroperation verwenden, die überlappende Daten erlaubt.

Beim Entfernen von Einträgen wird der Eintrag durch den Letzten belegten Slot im Block ersetzt, wenn er nicht selber der letzte ist und der Gap vom Zielblock erhöht. Es besteht hier die Option Verschiebungen rückgängig zu machen und den entstehenden freien Slot so zu verschieben, dass der Betrag der Offsets reduziert wird. Da beim Einfügen jedoch

in beide Richtungen gesucht wird glichen sich die Verschiebungen in den durchgeführten Tests ausreichend gegenseitig aus.

Bei der Organisation der Metadaten gibt es einige Variationen die getestet wurden. Anfänglich wurden alle Metadaten zusammen für jeden Block hinterlegt. Dazu wurde ein struct erstellt mit einem $\mathrm{i8}$ für den Offset, einem $\mathrm{u8}$ für den Gap und für den $\mathrm{Threshold}$ ein $\mathrm{u16}$. Bei einer Anzahl der Blöcke von n_B hat der Vec für die Metadaten eine Länge von $n_M = n_B + 1$. Beim letzten Matadateneintrag sind Offset und Gap 0, sodass bei der Bestimmung der Blockgrenzen keine Sonderbehandlung für das Ende notwendig ist. Beim Zugriff auf die Metadaten zeigte sich, dass nur auf den $\mathrm{Threshold}$ des Zielblocks zugegriffen wird. Daher wurde der $\mathrm{Threshold}$ in einen eigenen Vec verlegt um den Zugriff auf Gap und Offset zu vereinfachen und Lokalität beim iterieren zu erhöhen. Performance Tests zeigten keinen Unterschied, der separate $\mathrm{Threshold}$ bot jedoch andere Vorteile, sodass die Version beibehalten wurde.

Die Speichernutzung konnte reduziert werden, da für den Gap nur ein Bit in den Metadaten benötigt werden. Wenn bekannt ist, dass in einem Block noch mindestens ein Slot frei ist, dann ist für Block i der Slot $s[(i+1) * B + o_{i+1} - 1]$ frei. Dort kann dann hinterlegt werden wie viele Slots frei sind. Für alle Test sind die Einträge in den Sliding-Blocks zwei $\mathrm{u64}$. Daher wurde für den Gap der Datentyp auf ein usize erweitert. Im Slot ist dafür ausreichend Platz und es hebt die technische Beschränkung für die maximale Blockgröße auf. Beim Einfügen mit Verschiebung muss dann beachtet werden diese Information passend zu verschieben. Das betrifft jedoch nur den Start der Verschiebeoperation und hat keinen Einfluss auf die Schleife. Performance Messungen zeigten keinen Nachteil beim Zeitverhalten.

Das Bit für den Gap wurde mit dem Offset zusammen in ein Byte abgelegt. Das erforderte jedoch viele Bitmanipulationen beim Zugriff und Update der Werte. In einem weiteren Schritt wurde der Gap Bit in einen Bitvektor bestehend aus 64 Bit Zahlen hinterlegt. Beim Zugriff zeigte sich kein Nachteil. Bei der Suche können dadurch viele Blöcke über Bitoperationen gleichzeitig getestet werden. Ein vorhandener Gap wird mit einer 1 hinterlegt. Dadurch können im Gap Vector 64 Einträge auf einmal überprüft werden indem geprüft wird ob die Zahl größer 0 ist. Auch der Zugriff auf die $\mathrm{Offsets}$ vereinfachte sich, da der Gap nicht mehr raus gefiltert werden muss. Messungen Zeigten einen kleinen Vorteil. Überwiegend sind die Suchtiefen sehr gering sodass der Vorteil der parallelen Verarbeitung keine große Auswirkung hat.

4.2 Slick-Hash Datenzuordnung

Aufbauend auf den Funktionen der Sliding-Blocks ist Slick-Hash implementiert. Zunächst wird der Umgang mit dem Hauptspeicherbereich erläutert anschließend der Sekundärspeicher und das Zusammenwirken der beiden. Für die Implementierung von Slick-Hash wird als Eintrag für die Sliding-Blocks eine Kombination aus Schlüssel und Wert gewählt. Für den Schlüssel gilt dabei, das er vergleichbar sein muss, man einen Hash von ihm bilden kann und er muss verschiebbar sein. Für den Wert der Tabelle gibt es nur die Einschränkung,

dass er verschiebbar sein muss. Wobei verschiebbar bedeutet, dass die verwendeten Objekte unabhängig von ihrer Speicherposition sein müssen. Das ist in Rust jedoch der Normalfall. In den Tests waren beides stets u64 Werte.

Für jede Operation wird von dem gegebenen Schlüssel über eine Standard Hashfunktion ein u64 Hash gebildet. In der Implementierung kommt ahash zum Einsatz, es funktionieren jedoch auch andere Hashing Algorithmen. Von dem u64 Hash werden die ersten zwei Byte als Threshold verwendet und aus dem Rest der Blockindex abgeleitet. Auf den u16 Threshold wird noch ein saturating_add(1) angewendet. Es erhöht den Wert um 1, läuft jedoch nicht über sondern verbleibt in dem Fall beim maximal darstellbaren Wert. Dadurch ist der Threshold eines Elements immer größer als 0. Der Blockindex wird über ein Modulo durch die Anzahl der Blöcke gebildet.

Davon ausgehend, dass ein Eintrag im Hauptspeicherbereich ist oder dort hin gehört lassen sich die Operationen der Hashtabelle direkt auf die der Sliding-Blocks abbilden. Beim find von Slick-Hash wird für einen gegebenen Schlüssel der Block bestimmt in dem der Eintrag sein kann. Der Blockzugriff auf Sliding-Blocks gibt einen slice zurück, der alle Elemente des Blocks enthält und nicht mehr. Die Einträge in dem Block werden geprüft ob sie den gleichen Schlüssel haben und entweder kann ein passender Eintrag gefunden werden oder der Schlüssel ist nicht vorhanden. Für das insert von Slick-Hash wird zunächst wie beim find geprüft ob der Schlüssel bereits vorhanden ist. Wenn dem nicht so ist erfolgt ein insert in dem ausgewähltem Block aus den Sliding-Blocks. Ein ähnliches Vorgehen ist beim delete. Wie beim find wird geprüft ob der Eintrag vorhanden ist und wenn er gefunden wurde wir dieser durch die Sliding-Blocks entfernt.

Beim insert in einen Block kann nicht immer ein Platz gefunden werden. Daher ist es notwendig diese Einträge extra zu behandeln und find, insert und delete von Slick-Hash entsprechend zu erweitern. Der ursprüngliche Entwurf sieht dafür den Backyard vor. Wenn kein Platz mehr im Block des Hauptbereichs gefunden werden kann wird das Element mit dem kleinsten Threshold gesucht und dies in den Backyard verschoben. Der Threshold des Blocks wird anschließend auf diesen Wert gesetzt. Das ist eine Abweichung zum ursprünglichem Entwurf. Es ist möglich, dass mehrere Elemente den gleichen kleinsten Threshold haben. In dem Fall sieht der ursprüngliche Entwurf vor alle diese Elemente ins Backyard zu verschieben und den Block-Threshold auf den nächst höheren Wert zu setzen. Kollisionen sind jedoch sehr selten. Test haben gezeigt, dass es günstiger ist in den Fällen sowohl im Backyard als auch im Hauptbereich zu suchen als die potentiell mehrfachen Kollisionen sowohl beim insert als auch beim delete zu beachten. Bei einem Schlüssel Threshold der gleich dem Block Threshold ist wird zunächst im Backyard gesucht und wenn das Element nicht gefunden werden kann auch noch im Hauptbereich. Ansonsten ist das Verhalten wie im ursprünglichem Entwurf.

Beim entfernen von Einträgen soll die Hashtabelle nach und nach möglichst wieder in ihren Ausgangszustand zurück versetzt werden. Die Offsets werden nach Bedarf erhöht und reduziert und passen sich damit auch ohne explizites zurücksetzen an. Für den Threshold gibt es die beiden Varianten ihn nach ausreichend vielen delete tabellenweit zurück zu setzen und alle Einträge aus dem Backyard in den Hauptbereich einzufügen oder die Variante bei jedem delete kontinuierlich Elemente aus dem Backyard wieder in den Hauptbereich

zu verschieben. Das tabellenweite Zurücksetzen ist konzeptionell sehr einfach, da für das Backyard eine beliebige Hashtabelle verwendet werden kann, die über ihre Elemente iterieren kann. Bevorzugt ist jedoch das kontinuierliche Zurücksetzen. Das verteilt die Kosten für das delete besser und die Auswahl des Zeitpunkts für das Zurücksetzen führt zu keinen Problemen. Das führt zu der zusätzlichen Anforderung an das Backyard dass man effizient die beiden Elemente mit dem höchsten Threshold finden können muss, die in einen ausgewählten Block des Hauptbereichs kommen. Beim entfernen von einem Eintrag aus dem Hauptbereich wird dieser durch das Element mit dem höchsten Threshold aus dem Backyard ersetzt, dass in diesen Block gehört. Der Threshold des Blocks wird auf den Wert des zweithöchsten Threshold der Elemente im Backyard gesetzt oder auf 0 wenn kein weiteres im Backyard existiert.

4.3 Backyard

Für die erste Backyard Variante, die kontinuierliches delete unterstützt wurde die Hypothese aufgestellt, dass die Elemente im Backyard aus wenigen Blöcken stammen, die dafür deutlich überlaufen. Als Implementierung wurde eine Hashtabelle gewählt die für jeden Block der überläuft eine dynamische Liste enthält. Bei wenigen Blöcken die deutlich überlaufen wäre der zusätzliche Speicher für Hashtabelle und Listen vernachlässigbar gewesen. Experimente zeigten jedoch, dass die überwiegende Mehrheit aus ein bis zwei Elementen besteht, die pro Block überlaufen. Diese werden in der finalen Implementierung über eine Abbildungsfunktion zusammengefasst und wieder in Sliding-Blocks unter gebracht. Hier besteht die Möglichkeit, dass Elemente nicht eingefügt werden können. Wenn das der Fall ist, wird das Backyard vergrößert. Trotz vergrößertem Backyard besteht die Gefahr, dass nicht alle Elemente eingefügt werden können. Für den Fall wird noch eine dritte Speicherebene benötigt. Diese wurde als einfache Liste umgesetzt, die bei allen Operationen mit beachtet werden muss. In den Test war diese Liste jedoch stets leer und hatte damit keinen erkennbaren Einfluss auf die Ausführungszeiten. Um aus dem Backyard ein Element wieder in den Hauptblock zu verschieben wird über den Index des Hauptblocks und der Abbildungsfunktion der passende Backyard Block ausgewählt. Die enthaltenen Elemente werden gefiltert, sodass nur Elemente beachtet werden die zum Hauptblock gehören. Von denen wird das bestimmt, dass den größten Threshold hat und der zweitgrößte Threshold. Beides kann in einem Durchlauf bestimmt und anschließend an den Hauptbereich übergeben werden.

4.4 Optimierungen

Nach der funktionierenden Implementierung und algorithmischen Optimierung gab es noch den Versuch die Operationen durch spezialisierte Prozessorinstruktionen zu beschleunigen. SIMD Instruktionen konnten bei der Suche nach einem freien Slot verwendet werden. Bei der Prüfung ob ein Verschieben möglich ist müssen alle Offsets geprüft werden ob sie

```rust
// Nur wenn AVX2 verfügbar ist
#[cfg(all(target_arch = "x86_64", target_feature = "avx2"))]
pub fn can_slide_up<Idx>(&self, range: Idx) -> bool
where
    Idx: SliceIndex<[i8], Output = [i8]>,
{
    use core::arch::x86_64::*; // Macht SIMD Instruktionen verfügbar
    // Erstellt Vergleichsvektor mit maximalen Werten
    let all_max: __m256i = unsafe { _mm256_set1_epi8(i8::MAX) };
    self.0.get(range).map_or(false, |v| { // Alle betroffenen Offsets falls sie existieren
        let mut wide_iter = v.chunks_exact(32); // Unterteilung in slices der Länge 32
        for chunk in wide_iter.by_ref() { // Für alle slices
            // kann dieser in einen SIMD Vektor geladen werden
            let wide = unsafe { _mm256_loadu_si256(chunk.as_ptr() as *const __m256i) };
            // der wird mit den Maximalwerten verglichen wird
            let cmp = unsafe { _mm256_cmpgt_epi8(all_max, wide) };
            // Ergebnis bits werden in ein i32 übertragen
            // bei diesem müssen alle bits gesetzt sein, sonst kann der Offset nicht erhöht werden
            if unsafe { _mm256_movemask_epi8(cmp) } != !0 {
                return false;
            }
        }
        // Restliche Werte die nicht in den SIMD Vektor passen
        wide_iter.remainder().iter().all(|&v| v < i8::MAX)
    })
}
```

Listing 3: SIMD optimierte Version der Offsetprüfung

ihre Grenzen noch nicht erreicht haben. Per SIMD werden mehrere gleichzeitig mit ihren Grenzen verglichen und anschließend die Ergebnisse zusammengefasst. In Listing 3 wird geprüft ob die Offsets erhöht werden können. Die Prüfung ob die Offsets verringert werden können funktioniert analog.

Auch die Vergleiche der Thresholds bei der Suche nach dem kleinsten konnte über SIMD parallelisiert werden. Jedoch steht dafür nur ein 128 Bit Vektor zur Verfügung. Jeweils 8 Thresholds werden ermittelt und in einen Vektor geladen. Mit der SIMD-Funktion _mm_minpos_epu16 wird das Minimum und dessen Index bestimmt. Diese können mit _mm_extract_epi16 ausgelesen werden. Beide Optimierungen zeigten jedoch keine erkennbare Verbesserung der Ausführungszeit.

Ein Profiling zeigte, dass mit Abstand die meiste Zeit durch warten auf die Daten verbracht wird. Es wurde versucht die Zeit durch Prefetching zu optimieren. Noch bevor die exakten Grenzen eines Blocks über die Metadaten bestimmt wird kann man schon in etwa sagen wo die Daten liegen. Die CPU wurde angewiesen diese Speicherbereiche schon mal in den Cache zu laden. Auch das zeigte keine Verbesserung. In der letzten Variante wurde die Tatsache genutzt, dass die überragende Mehrheit der Daten im Hauptbereich ist und das Backyard geladen werden kann, während man im Hauptbereich sucht. Die Unterscheidung über den Threshold wird komplett weg gelassen. Wenn beim insert ein Element nicht im Hauptbereich unter gebracht werden kann wird ein beliebiges ins Backyard verschoben

und beim delete ein beliebiges zurück geholt. Die erfolglose Suche muss dann immer auch noch das Backyard betrachten. Jedoch kann dieses geladen werden während man im Hauptbereich sucht. Bei delete zeigte sich eine deutliche Verbesserung auf Kosten des erfolglosen find.

5 Evaluation

In diesem Kapitel werden einige der durchgeführten Leistungstests betrachtet. Sie zeigen, dass Slick-Hash vergleichbares Zeitverhalten hat bei effizienterer Speichernutzung. Mit aufgeführt sind zwei Varianten von Slick-Hash. Zum einen SlickHash mit allen durchgeführten Optimierungen mit Verwendung des Threshold und zum anderen SlickHashOT das optimierte Slick-Hash ohne Threshold. Für alle Hashtabellen mit einstellbaren Parametern wurden diese so gewählt, dass möglichst wenig Speicher verwendet wird ohne dass die Ausführungszeiten zu stark ansteigen. Für die Slick-Hash Varianten wurde eine initiale Blockgröße von 32 gewählt. Parametervergleiche zeigten für den Wert eine gute Balance zwischen effizienter Speichernutzung und Zeitverhalten. Ein weiterer Einstellungsparameter ist die Tiefe für die Suche nach einem freien Slot. Für SlickHash ist auch die Suchtiefe 32. Da SlickHashOT den Fokus mehr auf Speichereffizienz hat und Elmente im Backyard die Ausführungszeit stärker negativ beeinflusst wurde dort eine Suchtiefe von 64 gewählt. Die cuckoo Tabelle hat als Hauptparameter die Blockgröße, die Anzahl der Hashfunktionen und der zusätzliche Speicher wie in Kapitel 1.2 beschrieben. Eine gute Balance zwischen Speicherverbrauch und Ausführungszeit zeigte sich bei der Kombination aus einer Blockgröße von 8, einer Anzahl von Hashfunktionen von 4 und einer Speicherreservierung die 2% mehr Slots reserviert als angefordert. Die linear_probing Tabelle hat nur zusätzlichen Speicher als Parameter. Bei einem Wert von 8% zusätzlichen Slots zeigte sich ein vergleichbares Zeitverhalten Die Tabellen StdHashMap und sparse_map haben keine manuellen Einstellungsparameter.

Die F14Map Tabelle ist in den finalen Vergleichstest nicht mehr mit dabei. In früheren Test wurde sie mit verglichen und zeigte wie die StdHashMap ein sehr gutes Zeitverhalten, bei viel zusätzlich genutztem Speicher. Die folly Bibiliothek, die diese Hashtabelle enthält, besitzt jedoch sehr viele Abhängigkeiten. Eine Systemumstellung auf den Testsystemen machte eine erneute Bereitstellung dieser Abhängigkeiten notwendig, was im Rahmen der Arbeit nicht mehr gelang.

Die finalen Tests liefen auf zwei Systemen. Diese sind in der Tabelle 5.1 zu sehen. Beides sind x86_64 CPUs und besitzen moderne Instruktionserweiterungen wie AVX2 und SSE2 die in der Implementierung von Slick-Hash zum Einsatz kommen. Die Programmdateien liegen auf einem Netzwerkspeicher und auch die Ergebnisse werden auf dem Netzwerkspeicher abgelegt.

Die Erzeugung der Datensätze ist in Kapitel 3.3 genauer beschrieben. Für den Vergleich kann davon aus gegangen werden, dass es sich um zufällige 64 Bit Werte handelt. Sowohl für den Schlüssel als auch für den Wert. Einziger erkennbare Unterschied zur Zufälligkeit sollte sein, dass bei Schlüsseln die Wiederholungen kontrolliert werden. Bei einer gewollten

Name	backus	dijkstra
Modell	AMD EPYC 7702P	Intel Xeon E5-2683 v4
Kerne/Threads	64/128	32/64
max Takt	2184 MHz	2100 MHz
Hauptspeicher	972 GiB	491 GiB

Tabelle 5.1: Systeme für die finalen Tests

Wiederholung von Schlüsseln, wie bei den Anbfragen oder dem Löschen von Einträgen, kann auch davon aus gegangen werden, dass es Zufällig ist welcher Schlüssel wiederholt wird.

5.1 Volle Tabelle

Dieser Leistungstest betrachtet das Verhalten einer sehr vollen Tabelle. Es hat sich früh gezeigt, dass hier die Stärken von Slick-Hash liegen. Die Optimierungen betrachteten hauptsächlich diese Auswertung. Der Test läuft in vier Phasen ab. In der vorbereitenden ersten Phase wird die Tabelle erstellt und mit Einträgen befüllt. Es werden 98% der angeforderten Gesamtkapazität vorbefüllt. In der zweiten Phase wird die Tabelle auf 100% der angeforderten Gesamtkapazität aufgefüllt. Diese letzten 2% bis zur vollen Tabelle werden gemessen. In der dritten Phase werden erfolgreiche und erfolglose Abfragen auf der vollen Tabelle durchgeführt und gemessen. Die Anzahl der Abfragen entspricht auch 2% der Gesamtkapazität. Die erfolgreichen und erfolglosen Abfragen werden dabei abwechselnd durchgeführt. In der vierten und letzten Phase werden noch 2% der Tabelle wieder entfernt und der Löschvorgang gemessen.

Im Plot 5.1 ist der zusätzlich genutzte Speicher pro Element abgebildet. Zu sehen ist, dass die Tabellen unordered_map, flat_hash_map, StdHashMap und sparse_map signifikant mehr Speicher brauchen. Die Tabellen flat_hash_map und StdHashMap haben dabei die gleiche Speichernutzung. Das liegt daran, dass die StdHashMap nach dem Vorbild der flat_hash_map implementiert wurde. Die StdHashMap bietet jedoch das bessere Zeitverhalten, daher wird sie im weiterem als Repräsentant für die optimierte lineare Suche verwendet. Die unordered_map benötigt nicht nur mehr Speicher, sondern gehört auch in allen anderen Kategorien immer zu den schlechtesten und wird für eine bessere Übersicht im weiteren weg gelassen. Im Plot 5.2 sind die Tabellen mit Effizienter Speichernutzung. Zu erkennen ist, dass Slick-Hash meistens die effizienteste Speichernutzung hat und das Weglassen des Threshold nochmal ein halbes Bit pro Eintrag spart. Die Cuckoo Tabellen kommen bei der Speichereffizienz der Slick-Hash Tabelle am nächsten. Daher wird im weiterem die Leistung besonders mit diesen verglichen. Ein Vorteil von Slick-Hash, der hier jedoch nicht abgebildet ist, ist eine höhere Resilienz gegenüber der angegebenen Anzahl der erwarteten Elemente. Die Cuckoo Tabelle wurde so konfiguriert, dass sie gerade noch funktioniert. Schon geringfügige Mengen zusätzlicher Einträge laufen Gefahr nicht mehr

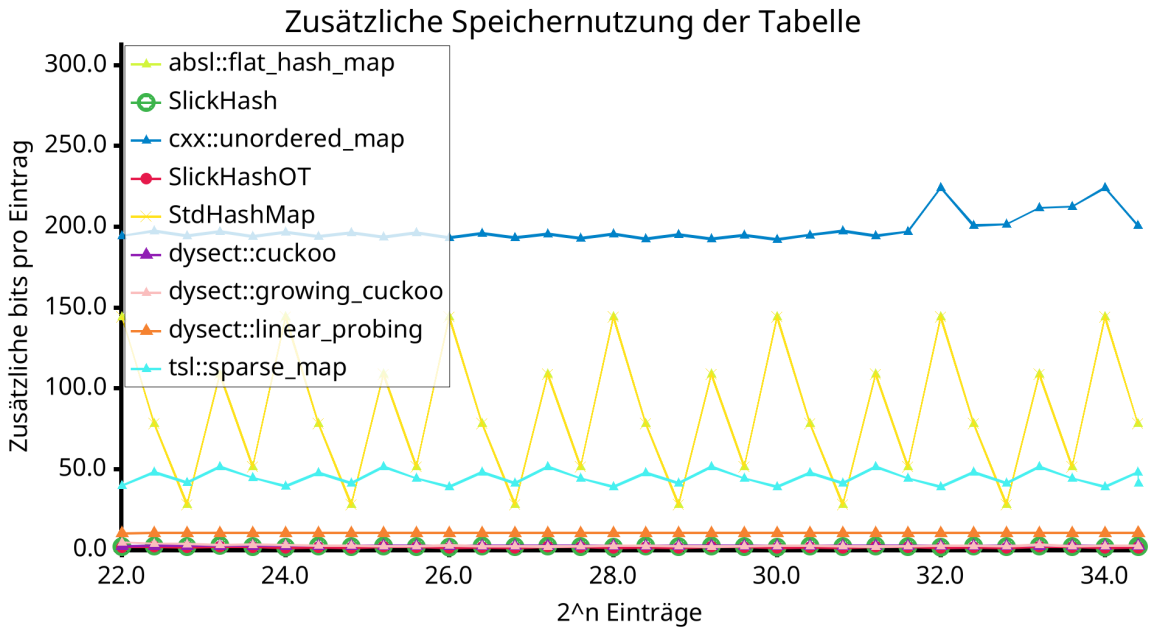

Abbildung 5.1: Speichernutzung der Tabellen, durchgeführt auf backus

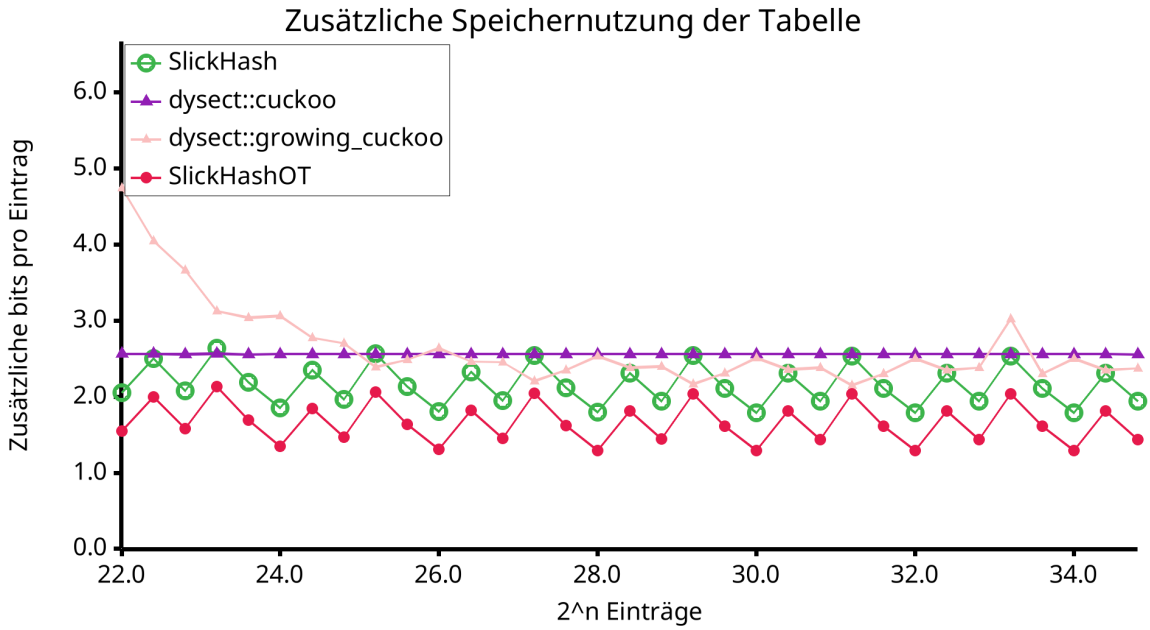

Abbildung 5.2: Speichernutzung der Sepeichereffizienten Tabellen, durchgeführt auf backus

unter gebracht werden zu können, wohingegen Slick-Hash damit keine Probleme hat und nur etwas langsamer wird.

Im Plot 5.3 ist die erfolgreiche Suche auf den Tabellen zu sehen. Hier verhalten sich die beiden Slick-Hash Varianten sehr ähnlich, da ein Großteil der Elemente im Hauptblock zu finden sind. Die Variante ohne Threshold spart in dem Fall die Abfrage des Threshold

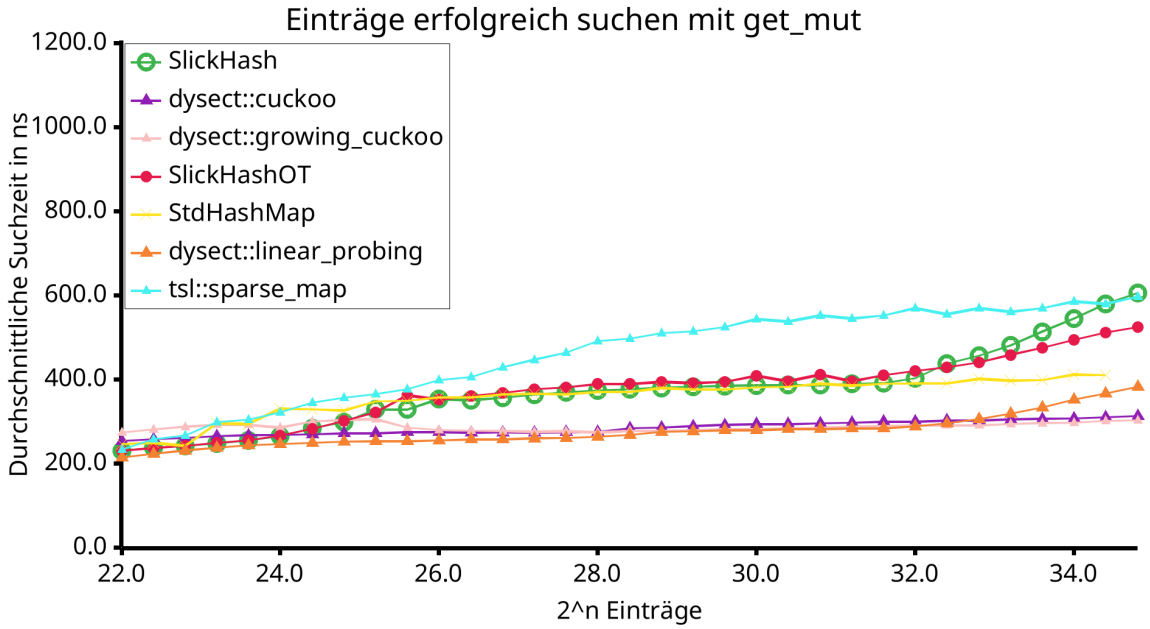

Abbildung 5.3: Durchschnittliche Zeit zu Suche vorhandener Einträge, durchgeführt auf backus

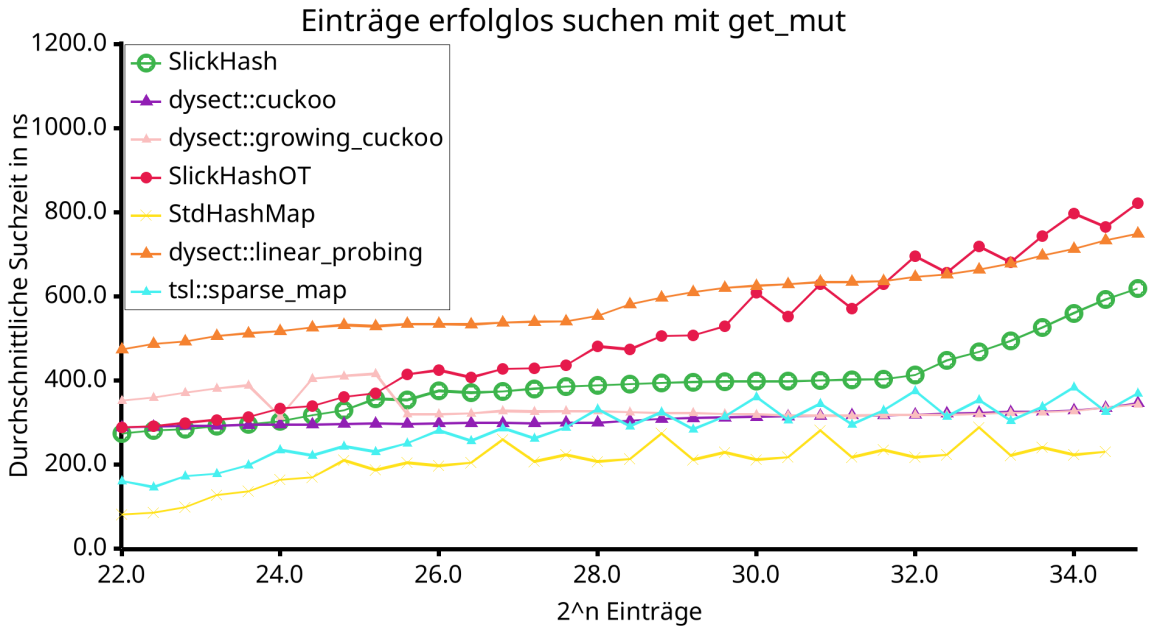

Abbildung 5.4: Durchschnittliche Zeit bis erfolglose Suche nach Einträgen beendet wird, durchgeführt auf backus

muss dafür jedoch für Elemente im Backyard erst den Hauptblock durchsuchen und dann erst das Backyard. In der Summe ergibt sich zunächst eine geringfügig erhöhte Zugriffszeit bei fehlendem Threshold. Beide Slick-Hash Varianten sind dabei auf einer Stufe mit StdHashMap. Ab einer Größenordnung von 2^{32} Elementen scheinen die Kosten für den Speicherzugriff zu zu nehmen. Vermutlich beginnt die CPU Speicher nicht mehr vorab zu laden. Dadurch steigen die Kosten für den Metadatenzugriff und die Slick-Hash Varianten werden langsamer. Der zusätzliche Zugriff von SlickHash gegenüber SlickHashOT auf den Threshold sorgt dann auch dafür, dass SlickHash langsamer als SlickHashOT wird. Die Cuckoo Varianten profitieren bei der Suche von ihren kleineren Blockgrößen was die zu erwartenden Vergleiche der Schlüssel reduziert. Die linar_probing Tabelle profitiert hier von ihrer sehr simplen Struktur und ausreichend Speicher. Die StdHashMap verwendet zwar auch eine lineare Suche und noch mehr Speicher, jedoch wird dieser auch dafür genutzt um Zusatzinformationen für einen beschleunigten Ausschluss von Schlüsseln zu speichern. Da der Vergleich von u64 Schlüsseln sehr schnell ist verlangsamt der Umweg die StdHashMap gegenüber dem linear_probing beim erfolgreichem Zugriff. Dafür ist sie sehr schnell in der erfolglosen Suche wie im Plot 5.4 zu sehen ist. Die linear_probing Tabelle muss hier sehr lange suchen, bis sicher ist, dass ein Schlüssel nicht vorhanden ist. Bei den Slick-Hash Varianten zeigt sich hier der Vorteil des Threshold. Nur in sehr seltenen Fällen muss in mehr als ein Block geschaut werden und dieser Block ist meistens der Hauptblock. Ohne den Threshold werden fast immer Hauptblock und Backyard durchsucht. Während der Suche im Hauptblock kann und soll das Backyard bereits geladen werden jedoch scheint es als würde die CPU ab 2^{28} Einträgen anfangen das Vorabladen beim SlickHashOT zu ignorieren. Dadurch kommt zu den zusätzlichen Vergleichen auch noch das Laden des Backyard Blocks. Ab 2^{32} Einträgen beginnt auch SlickHash wieder langsamer zu werden. Das passt zu der Vermutung, dass bei so großen Speicherbereichen das Vorabladen nicht mehr richtig funktioniert. Gegen die Vermutung spricht, dass die Cuckoo Tabellen sowie StdHashMap und sparse_map davon nicht erkennbar betroffen sind. Die genaue Ursache für die Verschlechterung der Ausführungszeit ist unbekannt. Die Cuckoo Tabellen profitieren bei der erfolglosen Suche wieder von der geringeren Anzahl der möglichen Positionen. Da diese jedoch über den Speicher verteilt sind wurde hier nicht damit gerechnet, dass sie schneller sind als Slick-Hash. Das Vorabladen der alternativen Positionen scheint besser zu funktionieren und das auch über die Speichergröße von 2^{32} Einträgen hinaus.

Die erfolglose Suche hat eine große Gemeinsamkeit mit dem Einfügen neuer Elemente. Das Einfügen beinhaltet immer auch eine Prüfung ob der gegebene Schlüssel nicht schon vorhanden ist. Erst dann kann das gegebene Element als ein neues eingefügt werden. Die resultierenden Einfügezeiten sind im Plot 5.5 zu sehen. Die Tabellen linear_probing und StdHashMap haben als Abbruchbedingung der nicht erfolgreichen Suche eine Speicherstelle an die das neue Element geschrieben werden kann. Daher ist bei den beiden Tabellen die Einfügezeit quasi identisch mit der erfolglosen Suche. Durch die Optimierung der erfolglosen Suche bei der StdHashMap Tabelle ist sie die mit Abstand schnellste beim Einfügen neuer Elemente. Die Cuckoo und Slick-Hash Tabellen bezahlen ihre Speichereffizienz mit hohen Einfügezeiten. Bei der sehr vollen Tabelle ist es wahrscheinlich, dass Slick-Hash im Hauptbereich keinen Platz finden kann. In dem Fall muss von alle enthaltenen Elmenten der Threshold gebildet werden um das Minimum zu finden und es dann ins Backyard zu

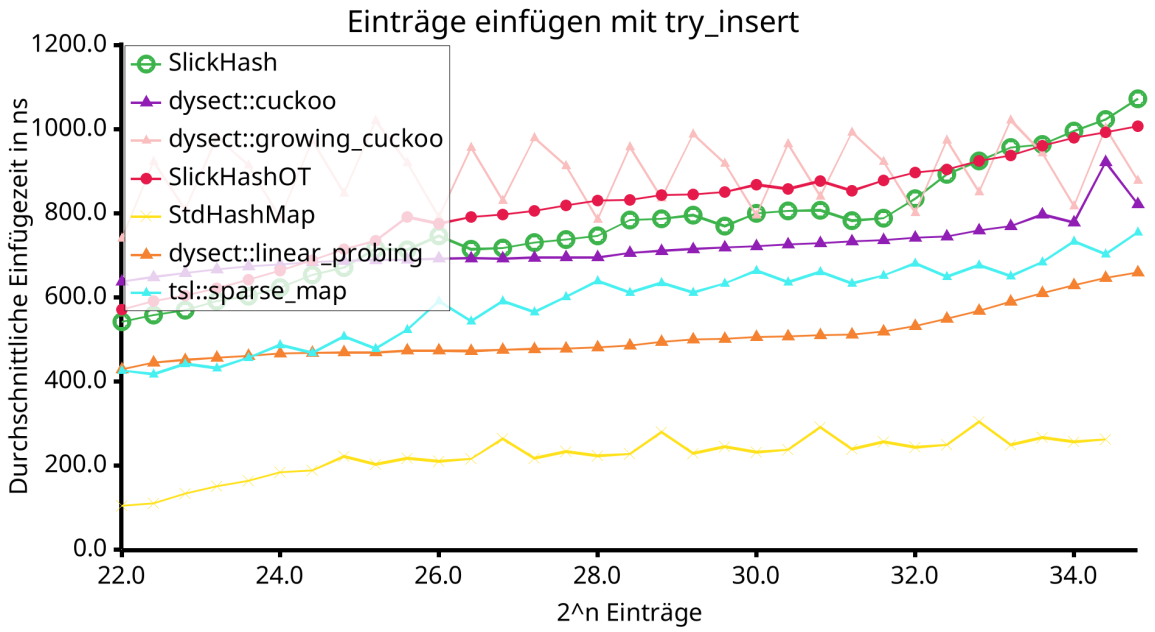

Abbildung 5.5: Durchschnittliche Zeit für das Einfügen neuer Elemente, durchgeführt auf backus

verschieben. Diese Suche wird beim weglassen des Threshold gespart. Dafür ist jedoch die erfolglose Suche aufwändiger. Die verwendete SlickHashOT hat jedoch einen Implementierungsfehler (siehe Kapitel 5.2) und das Einfügen müsste langsamer sein. Es wird nur der Hauptblock durchsucht. Das SlickHash trotzdem schneller ist, spricht dafür, dass viele Einträge direkt ans Backyard verwiesen werden. Dort sind die Blöcke kleiner. Bei einer Größenordnug von 2^{32} Elementen zeigt sich wieder der Anstieg der Ausführungszeit.

Die Einsparungen der fehlenden Suche nach einem maximalem Threshold bei Slick-Hash ohne Threshold kann man Löschen von Elementen beobachten, zu sehen im Plot 5.6. Hier ist die Slick-Hash Variante ohne Thereshold deutlich schneller. Bei den sehr vollen Tabellen ist es wahrscheinlich, dass ein Element aus dem Hauptblock entfernt wird und ein Element aus dem Backyard wieder in den Hauptblock zurück geholt werden muss. Bei Slick-Hash mit Threshold muss dafür das passende Element mit maximalem Threshold gefunden werden. Beim Slick-Hash ohne Threshold reicht hingegen ein beliebiges passendes. Die Tabellen linear_probing und StdHashMap müssen beim Löschen noch sicher stellen, das nachfolgende Elemente weiterhin gefunden werden. Die StdHashMap schafft dies sehr schnell im Vergleich zu linear_probing, verwendet jedoch auch viel mehr Speicher. Keinen zusätzlichen Aufwand haben die Cuckoo Tabellen. Das Löschen eines Eintrags entspricht hier der Suche des Eintrags. Die Ursache für den Ausschlag bei growing_cuckoo und $2^{33.2}$ Elementen ist unklar. In der Speichernutzung in Abbildung 5.2 kann man an dieser Stelle auch eine erhöhte Nutzung erkennen. Der Ausschlag ist jedoch beim Entfernen von Elementen. Beim Einfügen könnte man von einer Neuallokation aus gehen. In etwa zur Zeit der Ausführung des Durchlaufs kam es zu einem konkurierenden Zugriff auf die Programmdateien. Daher ist die Vermutung, dass es dabei zu einer Störung kam.

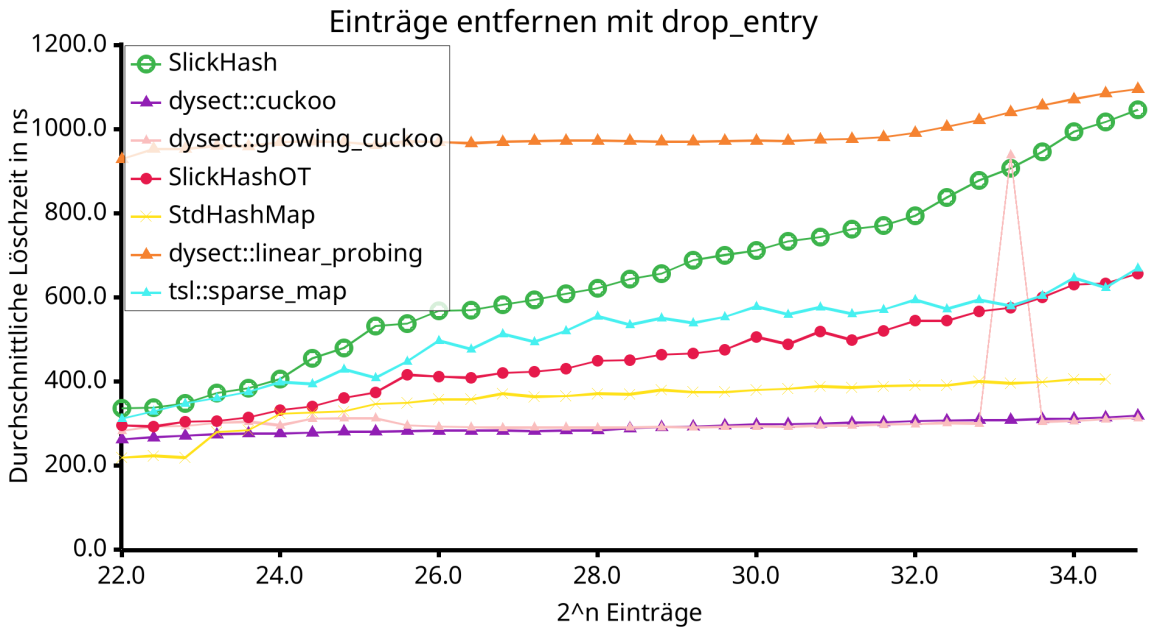

Abbildung 5.6: Durchschnittliche Zeit für das Entfernen von Elementen, durchgeführt auf backus

5.2 Verhalten bei wechselnden Füllständen

Slick-Hash erlaubt es einen linearen Speicherbereich sehr dicht zu befüllen. Das macht sie gut geeignet für sehr volle Tabellen. Jedoch werden auch bei Slick-Hash die Operationen bei sehr hohen Füllgraden langsamer. Zusätzlich muss Slick-Hash aktiv beim wiederholten Einfügen und Löschen ein entarten verhindern. Um die Wirksamkeit der Gegenmaßnahmen zu verifizieren und das Verhalten über verschiedene Füllstände hinweg zu untersuchen, wurde der Timeline Test entworfen. Dabei wird die Tabelle alternierend gefüllt und geleert. Zunächst fünf mal in einem Füllstandsbereich zwichen 25% und 75% der für eine Hashtabelle keine Herausforderung darstellen sollte und anschließend noch fünf mal zwischen 50% und 100%. Dabei werden zwischendurch immer wieder erfolgreiche und erfolglose Suchoperationen durchgeführt. Die Zeiten werden über jeweils etwa 1‰ der Gesamtkapazität gemittelt.

Das Verhalten über verschiedene Füllgrade hinweg ist im Plot 5.7 zu sehen. Dargestellt ist ein zeitlicher Verlauf. Die rote Fläche zeigt den Füllstand der Tabelle an. Zu erkennen ist, dass Slick-Hash sehr lange eine gute Performance aufrecht erhält. Im mittleren Füllbereich zeigt sich ein linearer Zusammenhang zwischen Füllgrad und den Operationszeiten. Das ist gut mit entsprechend gefüllten Blöcken zu erklären. Nicht erklären lässt sich damit, dass bei SlickHashOT ein Einfügen schneller ist als eine erfolglose Suche. Einfügen sollte immer langsamer sein als eine erfolglose Suche, da vor dem Einfügen geprüft werden muss ob das Element nicht schon vorhanden ist, was einer erfolglosen Suche entspricht. Das spricht für einen Implementierungsfehler. Der konnte inzwischen auch gefunden, jedoch nicht mehr rechtzeitig behoben werden. Beim Einfügen prüft SlichHashOT nicht die Elemente im Backyard die eigentlich immer geprüft werden müssten.

Bei den hohen Füllgraden bleibt die Ausführungszeit auch noch eine weile linear. Erst bei den letzten Prozentpunkten des Füllgrades steigt die Ausführungszeit vom Einfügen und Löschen stark an. Dabei ist die Slick-Hash Variante ohne Threshold beim Löschen von Einträgen schneller, dafür benötigt sie bei der erfolglosen Suche mehr Zeit. Das Einfügen kann auf Grund des Implementierungsfehlers nicht fair verglichen werden. Bei einer Korrektur müsste das Einfügen etwas mehr Zeit benötigen als die erfolglose Suche und damit in etwa auf einer Stufe mit SlickHash sein. Die erfolgreiche Suche ist bei beiden sehr ähnlich. Beide behalten auch nach mehreren Zyklen ihre Leistungscharakteristik bei allen Operationen bei. Es findet keine erkennbare Verschlechterung statt. Das zeigt, dass die Gegenmaßnahmen gegen eine Entartung erfolgreich sind.

Verglichen mit den Cuckoo Implementierungen die in Plot 5.8 zu sehen sind ist Slick-Hash durchgehend langsamer in allen Operationen. Die Cuckoo Tabellen erhöhen bei einem sehr hohen Füllgrad nur die Ausführungszeit der Einfüge-Operation signifikant. Wie bei Slick-Hash geschieht es jedoch erst wenn die Tabelle fast voll ist. Das Entfernen von Einträgen benötigt hingegen keine zusätzliche Zeit bei vollen Tabellen. Beide Cuckoo Tabellen sind bei der erfolgreichen Suche erstaunlich stabil. Vermutlich sind die Blöcke klein genug, sodass ein veränderter Füllstand nicht auffällt. Die growing_cuckoo Tabelle hat bei der erfolglosen Suche eine recht hohe Varianz, die cuckoo Variante ist auch hier sehr stabil. Beide Cuckoo Tabellen verändern ihr Verhalten auch nach mehreren Zyklen nicht.

Die StdHashMap, die im Plot 5.9 zu sehen ist, zeigt als einzige eine Veränderung. Die Tabelle wurde zu Anfang mit der erwarteten maximalen Anzahl an Einträgen erstellt. Nach einigen Zyklen von hinzufügen und entfernen führt ein vollständiges Befüllen jedoch zu einer Neuallokation obwohl genügend Platz vorhanden sein sollte. Die Spitze in der Ausführungszeit vom Hinzufügen neuer Einträge verlässt dabei sogar den Plot und geht bis 700 ns. Dafür ist sie zu den übrigen Zeitpunkten die schnellste Tabelle. Das wird sich aber mit einer deutlich erhöhten Speichernutzung erkauft.

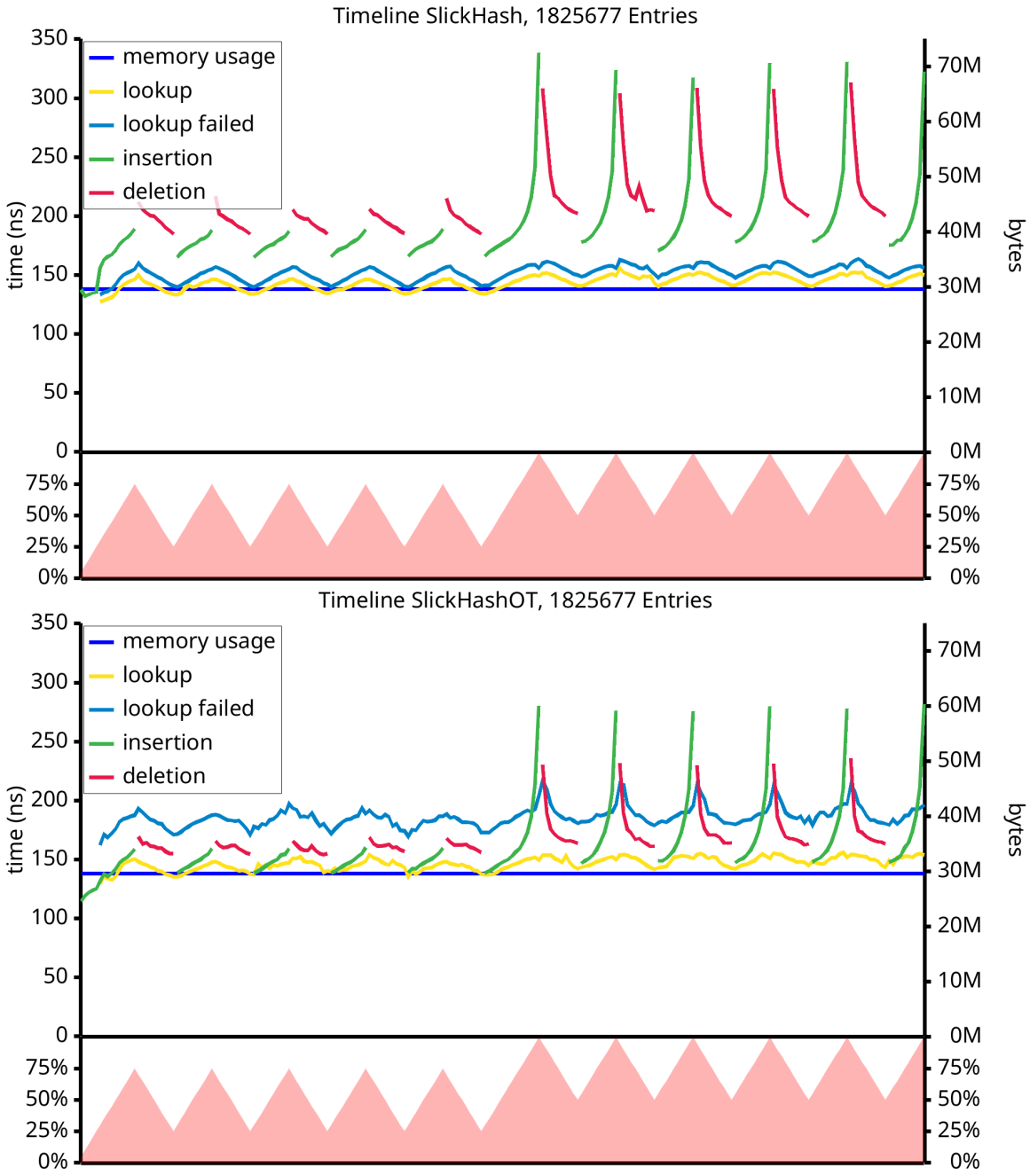

Abbildung 5.7: Zeitverhalten der Operationen auf einer Slick-Hash Tabelle bei verschiedenen Füllständen, durchgeführt auf dijkstra

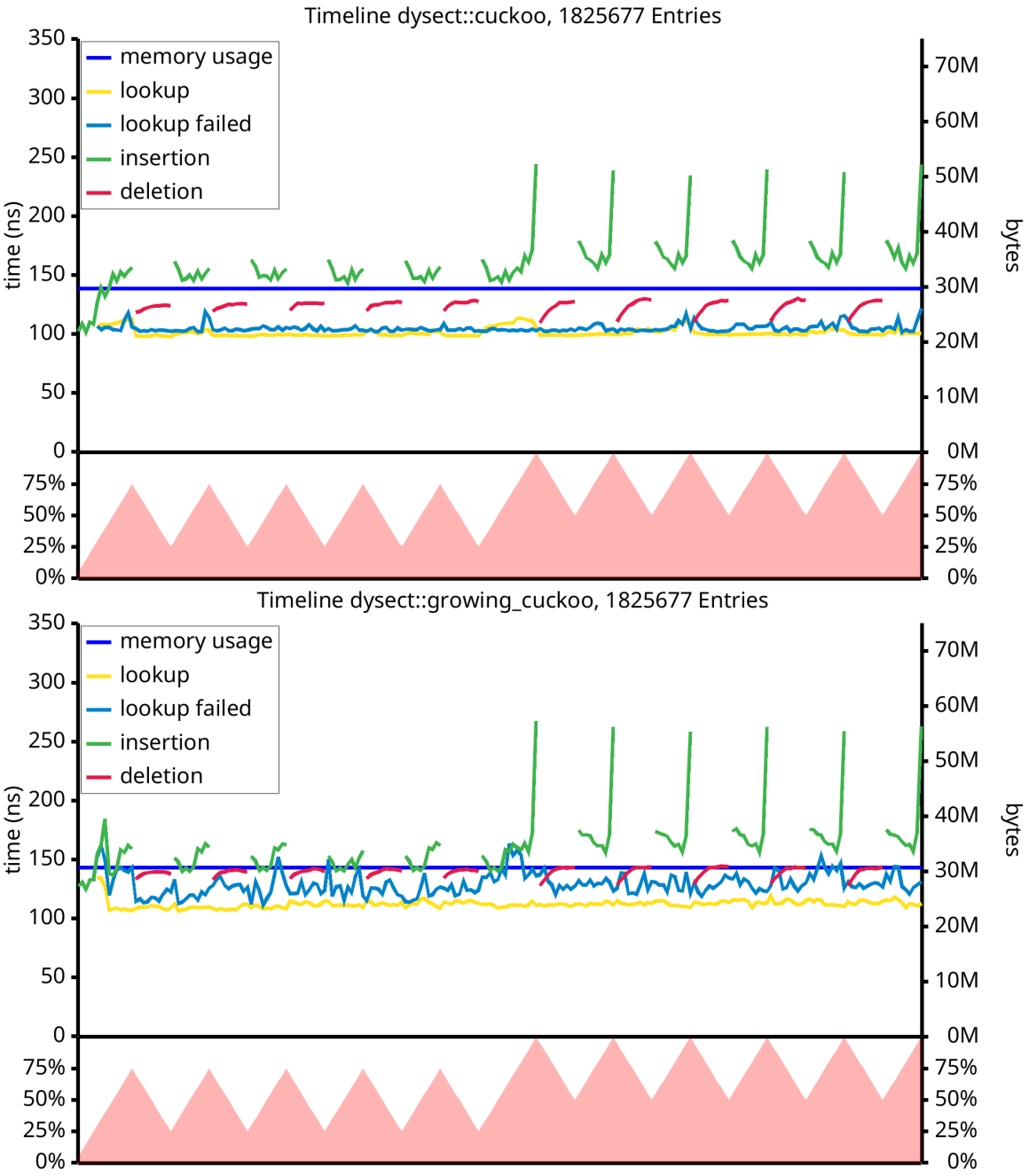

Abbildung 5.8: Zeitverhalten der Operationen auf einer Cuckoo Tabelle bei verschiedenen Füllständen, durchgeführt auf dijkstra

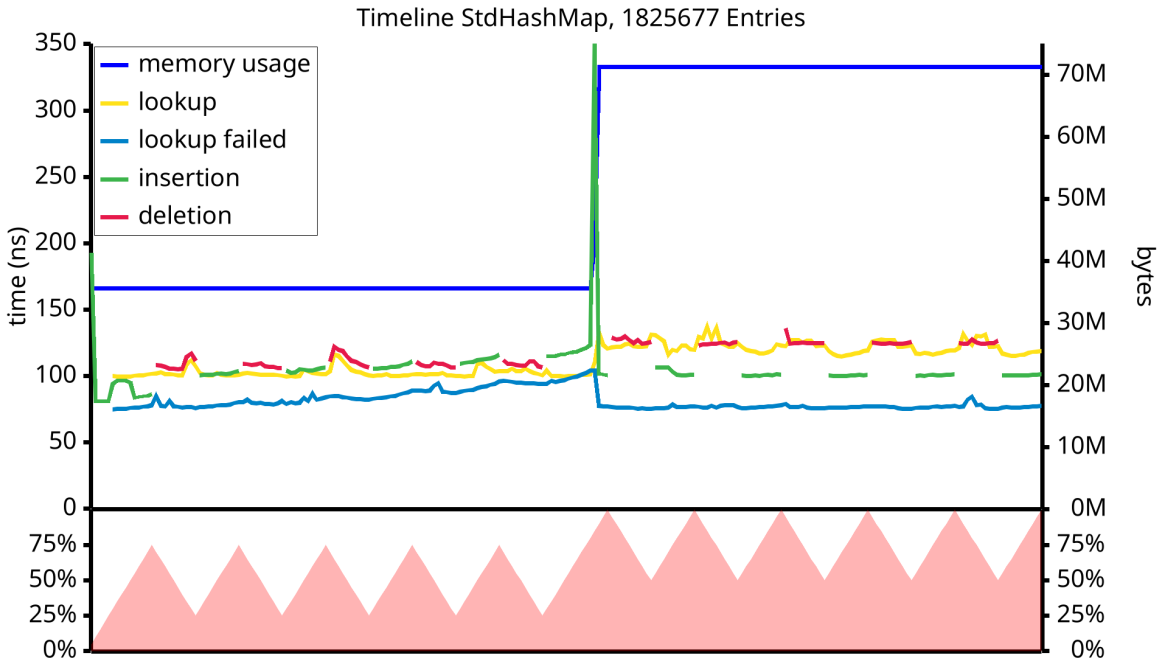

Abbildung 5.9: Zeitverhalten der Operationen auf der StdHashMap Tabelle bei verschiedenen Füllständen, durchgeführt auf dijkstra

6 Fazit

Die Slick-Hash Tabelle ist eine sehr speichereffiziente Hashtabelle. Der Raum der möglichen Optimierungen ist noch nicht ausgeschöpft. Der zusätzliche Aufwand der Metadatenverwaltung beschränkt jedoch die Reduktion der Ausführungszeiten. Slick-Hash hat ihr Anwendungsfeld in Anwendungen, die sehr speichereffizient sein müssen und bei denen die Ausführungszeiten der Operationen nicht im Vordergrund stehen. Auch degeneriert ihr Verhalten nicht so schnell wenn sie über ihre anfängliche Kapazität hinaus befüllt wird. Das erleichtert zusätzlich die effiziente Speichernutzung, da die reservierte Größe konservativ gewählt werden kann. Unabhängig von der Hashtabelle ist die dahinter liegende Datenstruktur der Sliding-Blocks interessant für verschiedene Anwendungsfälle. Eine weitere Untersuchung und Optimierung ist hier lohnend. Darauf aufbauend kann das Konzept von Slick-Hash sehr einfach implementiert werden. Das ermöglicht sehr gut Anpassungen an spezielle Bedürfnisse.

In der Anpassbarkeit liegt auch ein Vorteil von Slick-Hash. Die Gestaltung des Backyard bietet viel Raum für Anpassung. Wenn etwa die Löschungen bekannt und viele am Stück sind kann das Backyard auch am Stück zurück gesetzt werden wodurch man sich die wiederholte Suche nach passenden Einträgen sparen kann. Auch kann man über die Parameter der Blockgröße und der Suchtiefe viel Einfluss auf die Leistung nehmen. Bei kleinen Blöcken steigt jedoch die zusätzliche Speichernutzung pro Element durch die Metadaten. Hier kann eine Implementierung der Sliding-Blocks entgegenwirken, die auf kleine Blöcke optimiert ist. Es bleibt jedoch der zusätzliche Aufwand durch die Metadaten wodurch von einer Cuckoo Implementierung eine besser Leistung erwartet wird. Eine Cuckoo Implementierung muss dafür jedoch sehr voll gemacht werden. Kann man die erwartete Anzahl an Elementen nicht so genau vorher sagen bietet Slick-Hash mehr Spielraum, da zusätzliche Elemente noch immer im Backyard unter gebracht werden können. Solange die Überladung nicht zu groß ist, ist das Backyard verhältnismäßig klein und kann kostengünstig und einfach durch Neuallokation wachsen.

Bei reichlich zusätzlich vorhandenem Speicher sind die Linear-Probing Tabellen sehr schnell. Insbesondere beim Einfügen von neuen Elementen. Die Cuckoo Implementierungen nutzen viel Zeit beim Einfügen um eine deutliche dichtere Nutzung des Speichers zu ermöglichen und bei den anderen Operationen trotzdem noch schnell zu sein. Slick-Hash findet seine Anwendung, wenn man den Speicherverbrauch noch weiter reduzieren will. Damit bleibt die Wahl der passenden Hash-Tabelle für die eigene Anwendung eine Abwägung die stark geprägt ist von Speichernutzung gegen Laufzeit. Mit Slick-Hash kann diese Abwägung jetzt noch weiter in Richtung effiziente Speichernutzung verschoben werden.

Die weitere Entwicklung von Slick-Hash und den Sliding-Blocks sollte sich auf die effiziente Speichernutzung konzentrieren. Die Optimierungen durch parallele Datenverarbeitung haben praktisch keine Wirkung gezeigt. In den gewählten Konfigurationen von Blockgröße und Suchtiefe gab es dafür nicht genügend Operationen. Durch den zusätzlichen Zugriff auf die Metadaten ist es Unwahrscheinlich, dass die Zugriffszeiten besser werden als bei anderen Hashtabellen. Jedoch werden die Zugriffszeiten vom Zugriff auf den Speicher dominiert. Daher lässt sich die Speichernutzung allein schon durch die Erhöhung der Blockgröße und der Suchtiefe verbessern bei nur geringfügig steigenden Ausführungszeiten. Weiterhin ist bei Slick-Hash im Hauptbereich der zugewiesene Blockindex stabil. Dadurch lässt er sich als Informationsquelle für den Eintrag verwenden. Damit lässt sich die Speichernutzung noch weiter reduzieren.

Literatur

[1] Lianhua Chi und Xingquan Zhu. „Hashing Techniques: A Survey and Taxonomy“. In: *ACM Comput. Surv.* 50.1 (Apr. 2017). ISSN: 0360-0300. DOI: 10.1145/3047307. URL: https://doi.org/10.1145/3047307.

[2] *Crates.io*. URL: https://crates.io/.

[3] *DySECT*. URL: https://github.com/TooBiased/DySECT.

[4] Peter Sanders Kurt Mehlhorn. *Algorithms and data structures: The basic toolbox*. Bd. 55. Springer, 2008.

[5] Hans-Peter Lehmann, Peter Sanders und Stefan Walzer. *Sliding Block Hashing (Slick) – Basic Algorithmic Ideas*. 2023. arXiv: 2304.09283 [cs.DS]. URL: https://arxiv.org/abs/2304.09283.

[6] Martin Leitner-Ankerl. *Comprehensive C++ Hashmap Benchmarks 2022*. 2022. URL: https://martin.ankerl.com/2022/08/27/hashmap-bench-01/ (besucht am 15. 07. 2024).

[7] Tobias Maier, Peter Sanders und Stefan Walzer. „Dynamic Space Efficient Hashing“. In: *Algorithmica* 81.8 (Aug. 2019), S. 3162–3185. ISSN: 1432-0541. DOI: 10.1007/s00453-019-00572-x. URL: https://doi.org/10.1007/s00453-019-00572-x.

[8] Rasmus Pagh und Flemming Friche Rodler. „Cuckoo hashing“. In: *Journal of Algorithms* 51.2 (2004), S. 122–144. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2003.12.002. URL: https://www.sciencedirect.com/science/article/pii/S0196677403001925.

[9] *rand-unique*. URL: https://github.com/hoxxep/rand-unique.