technische universität
dortmund

Bachelorarbeit

Skalierbare Hashtabellen für moderne Multi-Core-Systeme

Rico Ahlbäumer
7. März 2018

Gutachter:
Prof. Dr. Jens Teubner
MSc. Jan Mühlig

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 6
http://dbis.cs.tu-dortmund.de/cms/en/home/

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die zunehmende Komplexität von Daten in der heutigen Zeit erzeugt einen gewaltigen Druck auf Datenbanken und Betriebssysteme, um nach wie vor gute Performance zu bieten. Hinzu kommt ein großes Verlangen nach hoher Skalierbarkeit. Dieses Verlangen entsteht aus der modernen Hardware, denn heutzutage besitzt fast jeder Computer mehr als nur einen Prozessorkern, teilweise sogar mehrere Prozessoren. Die Hardware ist hierdurch technisch in der Lage eine Skalierung zu bieten, doch diese kann nur dann erreicht werden, wenn die darauf ausgeführte Software diese Umstände ausreichend ausnutzen kann. Um dies zu erreichen muss ein Mehrkernsystem effektiv mehrere Aufgaben gleichzeitig ausführen können. Sobald diese Aufgaben auf Datenstrukturen zugreifen, die für gleichzeitige Zugriffe nicht ausgelegt sind, kann es zu unerwartetem Verhalten kommen, was beispielsweise zum Absturz der Anwendung oder des ganzen Systems führen kann. Aus diesem Grund ist eine Synchronisation für solche Anwendungen unumgänglich. Um dies zu erreichen können beispielsweise Semaphore verwendet werden. Diese führen aber zur Verlangsamung der Anwendung.

Um die Kosten durch solche Sperrverfahren so gering wie möglich zu halten ist ein sehr feingranulares sperren auf der Datenstruktur vorteilhaft. Hierfür eignen sich Datenstrukturen besser, bei denen schon vor Beginn der Aufgabe bekannt ist, welche Stelle betroffen ist. Eine solche Datenstruktur ist durch eine Hashtabelle gegeben, da durch ihre Hashfunktion bereits bekannt ist, welche Stelle betrachtet wird.

Da auch Hashtabellen durch Sperren ausgebremst werden stellt sich die Frage ob eine solche Datenstruktur auch komplett ohne Sperrverfahren auskommen kann. Einen Ansatz, um die Kosten durch Sperren zu verringern, stellt der MxKernel [16] dar. Der MxKernel ist ein leichtgewichtiger Kernel, welcher vom Thread-Modell abstrahiert und die sogenannten MxTasks einsetzt. Dies sind kleine Einheiten Arbeit, die eine atomare Ausführung garantieren. Sie werden also, anders als beim Thread-Modell vollständig ausgeführt und nicht verdrängt. So können zeitraubende Sperren umgangen werden, da gleichzeitigen Zugriffe auf die gleichen Ressourcen komplett umgangen werden können. Der MxKernel

wird als so genannte *bare-metal*-Laufzeitumgebung direkt auf der Hardware angesetzt und erlaubt Anwendungen damit eine bessere Kontrolle über die benutzte Hardware.

1.1 Ziele der Arbeit

Es konnte bereits gezeigt werden, dass einige Anwendungen auf dem MxKernel durch Verwendung von MxTasks auf Mehrprozessorsystemen besser skalieren als Threads. Unbekannt ist bisher noch, welche Auswirkungen Hashtabellen auf die Skalierung haben. Das Ziel dieser Arbeit besteht darin solche Hashtabellen zu entwickeln und herauszustellen, welche konzeptionellen Unterschiede hierfür zwischen den MxTasks und dem herkömmlichen Thread-Modell bestehen. Weiterhin soll geprüft werden, wie stark diese Hashtabellen auf einem Mehrprozessorsystem skalieren und einen Vergleich dieser mit einigen Thread-basierten Entwicklungen anstellen.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit stellt in Kapitel 2 zunächst die Grundlagen des Hashings, Grundlagen für Multiprozessor- und Mehrkernsysteme, sowie das Konzept des MxKernel und ein darauf entwickeltes Betriebssystem mit Threads auf Task-Basis vor. In Kapitel 3 wird die Implementierung der Hashtabellen behandelt und vergleicht die Implementierungen im Hinblick auf das Thread- und Task-Modell. Die entwickelten Hashtabellen werden evaluiert und in Kapitel 4 werden hierzu Ergebnisse vorgestellt. Kapitel 5 fasst die Arbeit zusammen und gibt einen Ausblick auf zukünftige Forschungen.

Kapitel 2

Grundlagen

Um effizient auf Teile einer Datenbank zuzugreifen, insbesondere solche, die oft genutzt werden, ist es meist sinnvoll Indexstrukturen dafür zu entwerfen. Eine Indexstruktur auf ausgewählten Daten weist diesen einen Index zu und möchte Zugriffszeiten in Größenordnung $o(n)$, bei n Einträgen in der Struktur, erreichen. Diese Zugriffszeit ist wichtig, da auch eine Indexstruktur sehr schnell sehr groß werden kann und somit eine Operation in linearer Zeit zu langsam wird. Dies bedarf einer Datenstruktur, die diesen Zugriffszeiten auch gerecht wird. Zu den bekanntesten Strukturen hierfür zählen:

- sortiertes Array (mit binärer Suche: $O(\log n)$)
- Binärbaum (Baumsuche in asymptotisch $O(1)$)
- B-Baum (Worst-Case Suche in $O(\log n)$)
- Hashtabelle (Suche in asymptotisch $O(1)$)

Diese Datenstrukturen besitzen noch einen weiteren bedeutenden Vorteil gegenüber beispielsweise einer einfachen Liste. Die gesuchten Werte befinden sich in einem erwarteten Bereich, der sich durch ein paar Berechnungen bereits recht früh erkennen lässt. Dies wird später in dieser Arbeit ausgenutzt, um eine einfache Möglichkeit zu bieten mehrere Prozessoren zu unterstützen. Im Rahmen dieser Arbeit wird das Hashing betrachtet.

Im folgenden Kapitel wird ein Überblick über die benutzten Methoden der Arbeit, sowie ein Einblick in den zugrundeliegenden Kernel MxKernel und das darauf entwickelte MxOS System gegeben. Zudem wird ein kurzer Einblick in die moderne Hardware gegeben und weshalb für diese Hashing gut geeignet ist.

2.1 Hashing

Hashing beschreibt eine Technik die bestimmten Daten, beispielsweise einem Tabelleneintrag in einer Datenbank, einen arithmetischen Wert zuordnet, welcher danach in eine

Hashtabelle gespeichert wird [11]. Eine Hashtabelle ermöglicht eine amortisiert erwartete Laufzeit für eine Datenbankoperation (einfügen, suchen, löschen) von $O(1)$. Dies ergibt sich daraus, dass ein berechneter Wert für einen Eintrag bis auf einige Ausnahmen eindeutig einer Stelle in der Hashtabelle zuzuordnen ist. Um diesen Wert zu berechnen benutzt man eine Hashfuntion h:

2.1.1 Definition. Hashfunktion: Sei $h : D \to H$, mit D Menge der zu hashenden Daten und H Menge der möglichen Hashwerte. Man nennt h Hashfunktion, wenn $|D| \geq |H|$ gilt.

Üblicherweise wird H als Teilmenge der natürlichen Zahlen gewählt. Ein Wert aus D wird für diese Arbeit mit Key bezeichnet.

h kann also mehrere Daten auf die gleiche Stelle hashen. So ein Fall wird als Kollision bezeichnet, welche in realer Anwendung jedoch nicht übermäßig häufig auftritt, da nur eine Teilmenge $D' \subseteq D$ mit h abgebildet wird. Sollte dies dennoch passieren, so muss die Kollision behandelt werden. Meist wird dabei eine andere Hashfunktion $h' : D \to H'$, mit H' eine Menge möglicher Hashwerte, benutzt. In den später vorgestellten Hashingvarianten werden Lösungen für das Problem der Kollision gezeigt.

Wenn zuvor bekannt ist wie viele Keys ungefähr in die Tabelle gehasht werden, so kann die Tabelle groß genug gewählt, um eine ausreichend schnelle Kollisionsbehandlung zu garantieren. Sollte die Größe zuvor nicht bekannt sein, beziehungsweise die Tabelle wird mit mehr Werten gefüllt als erwartet, und es ist nicht mehr möglich einen weiteren Wert einzuhashen, so muss die Tabelle vergrößert und neu aufgebaut werden. Dieses Verfahren wird Rehashing genannt und benötigt amortisiert $O(n)$, wenn die Tabelle aus n Einträgen besteht. Nach diesem Verfahren wird auch die Hashfunktion h auf die neue Tabelle angepasst.

Im folgenden werden zwei Varianten vorgestellt, wie eine Hashtabelle aufgebaut werden kann, wie das Problem der Kollision im einzelnen gelöst wird und es werden Vor- und Nachteile dieser dargelegt.

2.1.1 Hashing mit Verkettung

Für das Hashing mit Verkettung [4] entspricht der berechnete Wert der Hashfunktion h einem Bucket, in den der Key gespeichert wird. Ein Bucket bezeichnet hierbei eine dynamische Datenstruktur, wie beispielsweise einen Baum oder eine Liste. Dies ist zugleich auch die Lösung für das Problem der Kollision, da jeder Bucket viele Werte aufnehmen kann.

Auf den ersten Blick erscheint diese Lösung nicht sinnvoll, da die beiden genannten Datenstrukturen keine erwartete konstante Laufzeit $O(1)$ garantieren können. Das Problem tritt jedoch nur dann auf, wenn sehr viele Keys in den gleichen Bucket gehasht werden, was entweder daran liegen könnte, dass die Hashfunktion nicht gut gewählt wurde, oder dass

die Tabelle aus zu wenig Buckets besteht. Das Problem kann also gut umgangen werden, wenn bereits zuvor bekannt ist, wie viele Werte ungefähr in die Tabelle eingehasht werden, indem ausreichend viele Buckets erstellt werden. Wenn dadurch ein Bucket bedeutend weniger Keys enthält, als sich Buckets in der Hashtabelle befinden, so erhält man nach wie vor eine erwartete konstante Laufzeit für eine Operation.

Suchen und Entfernen sind hier sehr einfach gestaltet. Sobald die Hashfunktion h den Bucket berechnet hat, muss dieser nur noch nach dem jeweiligen Key durchsucht werden und sollte dieser dort nicht auffindbar sein, so ist er auch in keinem anderen Bucket der Hashtabelle vorhanden.

2.1.2 Hashing mit offener Adressierung

Das Hashing mit offener Adressierung [4] nutzt wie schon zuvor das Hashing mit Verkettung ebenfalls Buckets, welche sich durch die Hashfunktion h ergeben. Jedoch ist hierbei pro Bucket nur eine gewisse Anzahl an Keys erlaubt, meist sogar nur ein einziger. Diese Variante verhindert, dass Buckets in ihrer Größe extrem ansteigen, benötigt aber eine Möglichkeit mit Kollisionen umzugehen. Dies wird gelöst, indem eine neue Hashfunktion h' auf den Key angewandt wird. Um diese Hashfunktion h' zu finden kann man unter anderem die lineare Sondierung benutzen, welche, sollte der Kollisionsfall auftreten, einfach versucht den Key x in den nächsten Bucket in der Hashtabelle zu hashen, konkret heißt dies: $h'(x) = h(x) + l$, wobei l die Anzahl an benutzten Sondierungsschritten bezeichnet.

Suche und Entfernen sind bei dieser Variante mit dem gleichen Mehraufwand verbunden, wie auch schon einfügen. Sollte in einem Bucket der Key nicht gefunden werden, so besteht nach wie vor die Möglichkeit, dass er sich an einer durch Sondierung berechneten Stelle befindet. Es muss also unter Umständen die ganze Tabelle durchsucht werden. Es kann jedoch kann aufgehört werden zu suchen, sobald man auf ein Nullelement, also einen noch nicht belegten Bucket, trifft, da sonst zuvor die Einfüge-Operation den Key hier eingehasht hätte. Daraus folgt ein Problem beim Entfernen eines Eintrages, wenn dieser einfach auf das Nullelement gesetzt wird. Es kann so die Information verloren gegangen werden, dass nach dem Nullelement noch ein Key existiert, welcher durch Sondieren an diese Stelle gelangt ist und nun nicht mehr auffindbar ist, da eine Suche zuvor abbricht. Für diesen Fall wird beim Entfernen nicht das Nullelement gesetzt, sondern ein Verweis darauf, dass an dieser Stelle ein Key stand. Dies erlaubt diese Stelle neu zu beschreiben und löst das Problem mit der Suche.

Sollte auch mit Sondierung keine Stelle mehr gefunden werden, an dem ein Key eingehasht werden kann oder es ist eine gewisse Belegung - meist 80% - überschritten, so muss ein Rehashing durchgeführt werden.

2.2 Multiprozessor- und Mehrkernsysteme

Moderne Systeme benutzen nicht mehr, wie traditionell üblich, einen Prozessor mit nur einem Kern. Stattdessen wird die Last hierfür oft auf mehrere Kerne in einer CPU oder, zumeist auf Servern oder Hochleistungsrechnern, sogar auf mehrere CPUs verteilt, welche ebenfalls jeweils mehr als einen Kern besitzen. Ein Grund hierfür sind die immens steigenden Kosten, um die Taktfrequenz eines einzelnen Prozessors zu erhöhen, welche diese Art der Prozessoren nicht mehr sinnvoll machen [17]. Kosten sind aber nicht der einzige Grund, weshalb sich von Einkernprozessoren abgewandt wurde. In einem Mehrkernsystem kann jeder Kern für sich selbst gleichzeitig Prozesse durchführen und ist bis auf kleine Ausnahmen, wie zum Beispiel ein Teil der Caches, ein vollwertiger Prozessor. Multiprozessorsysteme sind ein Verbund von Prozessoren, bei denen jeder Prozessor eigenen Speicher und oft auch mehrere Kerne besitzt [6].

Solche Architekturen sollen dazu führen, dass die Rechenleistung zur Anzahl der Kerne skaliert. Skalierung bedeutet in diesem Fall, dass die Leistung eines Systems proportional zur Anzahl der Kerne oder Prozessoren steigt. Dies ist äußerst wünschenswert, wird inzwischen aber durch den Flaschenhals des Hauptspeichers verhindert, welcher mit der Geschwindigkeit der Mehrkernprozessoren nicht mithalten kann. Dies führt dazu, dass Prozessorcaches größer und leistungsfähiger sein müssen, um dies auszugleichen [2].

Vom Programmierer wird in diesem Zusammenhang gefordert die gestiegene Anzahl an Kernen sinnvoll einzusetzen, da eine Anwendung nicht natürlich wissen kann, wie sie dies ausnutzt. Im folgenden werden die Eigenschaften von Caches und zwei Speicherarchitekturen, welche auf Multiprozessor- und Mehrkernsysteme eingesetzt werden, erklärt und weshalb es für Hashing eine sinnvolle Verbesserung ist auf mehrere Kerne und Prozessoren zugreifen zu können.

2.2.1 Caches

Wenn Daten durchweg über den Hauptspeicher auf den Prozessor geladen werden führt dies zu einem Flaschenhals, da der Weg zwischen Prozessor und Hauptspeicher überlastet wird. Dies ist insbesondere in Mehrprozessorsystemen problematisch, in denen verschiedene Prozessoren über einen Bus mit dem Hauptspeicher verbunden sind. Zudem müssen die Daten für jeden Zugriff neu aus dem Hauptspeicher geladen werden. Um dies zu verhindern besitzen Prozessoren und Kerne Caches. Dies sind schnelle Puffer-Speicher, welche sehr viel kleiner als der Hauptspeicher sind [5]. Der Cache eines Prozessors besteht aus verschiedenen Cache Leveln, die mit aufsteigendem Level langsamer, aber dafür größer werden. Als Beispiel dient hierbei der Intel® Core™i3 Prozessor, welcher von Abbildung 2.1 dargestellt wird. Dieser besitzt auf dem untersten Level, dem sogenannten L1-Cache, eine Größe von 32 Kilobyte, darauf folgend im L2-Cache eine Größe von 256 Kilobyte und auf dem höchsten Level, dem L3-Cache eine Größe von 3 Megabyte. Die Steigerung des

Speichers erfolgt auf Kosten der Zugriffszeiten. Bei modernen Mehrkernsystemen, wie dem oben genannten, wird der L3-Cache von allen Kernen geteilt [5]. Dies erlaubt, dass auf die gleichen Daten von verschiedenen Kernen schnell zugegriffen werden kann.

Jeder Cache ist in Blöcken von sogenannten *Cache-Lines* unterteilt, welche bei oben genanntem Prozessor 64 Byte groß sind. Wenn Daten aus dem Hauptspeicher in den Cache geladen werden, werden diese, auch wenn sie kleiner als die Cache-Line sind, immer in Blöcken mit der Größe der Cache-Line kopiert. Ein Zugriff auf Daten, die sich nicht im Cache befinden, wird Cache-Miss genannt. Eine große Anzahl an Cache-Misses sollte vermieden werden, da dies zu einem Leistungsabfall führt. Sobald ein Level des Caches voll beschrieben ist, wird ein Eintrag nach einer bestimmten Strategie in das nächsthöchste Level verschoben. Wird durch einen Kern ein Cache Eintrag verändert, so wird dieser nicht direkt im Hauptspeicher geändert, sondern nach Cache-Kohärenz-Protokoll [8, S.205f] erst nach erneutem Lesen. Der Zugriff von einem neuen Kern kann also erst erfolgen, sobald der Speicherbereich wieder kohärent ist. Damit dies auch angezeigt werden kann, kann je nach Protokoll eine Cache-Line einen Status annehmen. Als Beispiel sei hier das MESI Protokoll genannt [8, S.213]:

- *Modified*: Daten nur in diesem Cache vorhanden, lokal verändert, Hauptspeicherkopie ungültig
- *Exclusive*: Daten nur in diesem Cache vorhanden, lokal unverändert, Hauptspeicherkopie gültig
- *Shared*: Daten in mehreren Caches vorhanden, lokal unverändert, Hauptspeicherkopie gültig
- *Invalid*: Daten ungültig, anderer Cache enthält diese Daten verändert

Sollte eine Cache-Line *Invalid* gekennzeichnet sein, so muss diese, wenn darauf zugegriffen wird, erst warten, dass die *Modified* Cache-Line in den Hauptspeicher zurückgeschrieben wird. Wenn von zwei Kernen unterschiedliche Daten bearbeitet werden, die sich im Speicher jedoch eine Cache-Line teilen, muss diese zwischen den Kernen immer wieder kohärent gehalten werden. Dieses Problem wird *False Sharing* genannt und kann verhindert werden indem Speicher geschickt alloziert wird, indem dieser zum Beispiel an eine Cache-Line ausgerichtet wird.

2.2.2 Non-Uniform Memory Access

In Multiprozessorsystemen existieren unterschiedliche Arten wie Hauptspeicher an die einzelnen Prozessoren angeschlossen ist. Zum einen Systeme, in denen alle Prozessoren auf einen oder mehrere Speicher zugreifen können, und Systeme in denen jeder Prozessor über

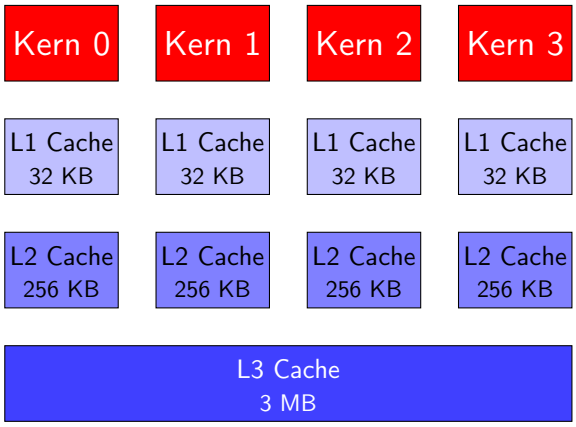

Abbildung 2.1: Diese Abbildung zeigt den Aufbau des Caches eines Intel® Core™i3 Prozessors.

seinen eigenen Hauptspeicher verfügt, auf welchen andere Prozessoren keinen Zugriff erhalten. Wenn aber ein Hauptspeicher von allen Prozessoren geteilt wird, kann ein *Uniform Memory Access* [14] eingesetzt werden. Hierbei werden Prozessoren so an den Speicher angebunden, dass für jeden die gleiche Zugriffszeit ermöglicht wird.

Im Gegensatz hierzu steht das Konzept des *Non-Uniform Memory Access* [12], in dem jeder Prozessor eigenen Speicher besitzt, auf den andere Prozessoren aber auch Zugriff erhalten, siehe hierzu Abbildung 2.2. Ein einzelner Speicher zusammen mit seinen Kernen wird auch NUMA-Knoten genannt. Zugriffszeiten in diesem System bestimmen sich darüber in welchem Speicher sich die gesuchte Adresse befindet. Wenn eine Adresse im gleichen NUMA-Knoten ist, wie der Kern, der sie anfragt, so ist der Zugriff hierauf schneller als auf einem UMA-System. Wenn das Gegenteil eintrifft, also eine Adresse in einem anderen NUMA-Knoten vorliegt, so wird die Zugriffszeit und damit auch das komplette System deutlich verlangsamt [14]. Um NUMA auszunutzen sollte bei der Entwicklung von Software darauf geachtet werden, dass Zugriffe zwischen verschiedenen NUMA-Knoten vermieden werden.

2.2.3 Hashing auf mehreren Kernen

Hashing benötigt, wie viele andere Aufgaben auch, einige Anpassungen, wenn mehrere Kerne gleichzeitig die Tabelle bearbeiten sollen. Eine Tabelle muss abgesichert werden, um kein unerwartetes Verhalten hervorzurufen. Nun stellt sich die Frage, wie fein diese Absicherung sein kann, damit die Tabelle noch korrekte Funktionalität besitzt, jedoch nicht unnötig viele Sperren aufgebaut werden. Hierbei kann eine wichtige Eigenschaft der Hashtabelle ausgenutzt werden: Ein Key befindet sich, durch Hashfunktion und je nach Hashingvariante, in einem vergleichsweise kleinen Bereich der Tabelle. Dies erlaubt sehr feine Sperren. Bei Hashing mit Verkettung (siehe Abschnitt 2.1.1) sogar so fein, dass jeder Bucket eine Sperre erhält, da sicher ist, dass eine Operation in diesem Bucket durchgeführt wird. Dies erzeugt kaum zusätzliche Kosten für Sperren. Bei Hashing mit offener Adres-

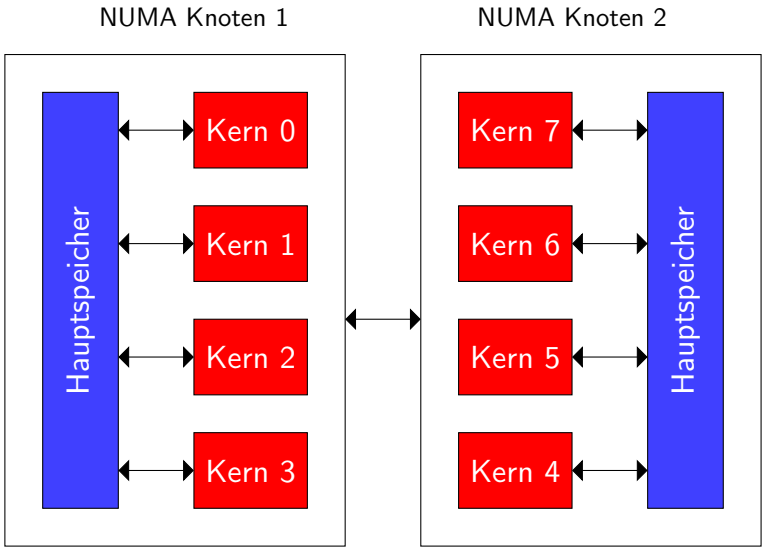

Abbildung 2.2: Diese Abbildung zeigt die Speicheraufteilung eines NUMA-Systems.

sierung (siehe Abschnitt 2.1.2) sollte eine Sperre etwas gröber gehalten werden, da unter Umständen die ganze Tabelle nach einem freien Platz durchsucht werden muss. Hierbei kann es also sinnvoll sein die Tabelle in Bereiche zu unterteilen und jeden Bereich einzeln sperrbar zu machen. So werden selbst bei Sondierung über die Bereichsgrenzen hinaus vergleichsweise wenig Sperren für eine Operation benutzt.

Eine andere Frage ist, wie verteilt die Operationen auf der Hashtabelle sind. Wenn sehr oft der gleiche Bucket oder Bereich abgebildet wird und somit oft Kerne darauf warten müssen, dass ein Bucket oder ein Bereich frei wird, so erübrigt sich der Vorteil, den ein Mehrkernsystem bringt und die Hashfunktion sollte für diese Daten angepasst werden. Wenn sich die Operationen aber gut verteilt auf der ganzen Tabelle befinden, so erhält man, da sehr viel seltener auf eine freiwerdende Sperre gewartet wird, einen deutlich besseren Durchsatz auf dieser Hashtabelle.

Hashing ist also sinnvoll auf mehrere Kerne übertragbar, da durch die große Anzahl an Buckets sehr selten die gleichen Kerne auf dem gleichen Bucket arbeiten. Eine letzte Möglichkeit, um die Effizienz einer Hashtabelle auf mehreren Prozessoren noch weiter zu steigern ist es einen Bucket einem Kern fest zuzuordnen. Dies sollte die Anzahl an Cache Misses in beiden Methoden verringern, da möglicherweise der ganze Bucket oder ein Teil des durchsuchten Bereichs noch im Cache gespeichert ist, sobald dieser wieder durchsucht wird. Durch Sondierung muss natürlich zudem beachtet werden, dass eine Operation möglicherweise über mehrere Kerne springen muss, bis sie komplett durchgeführt wurde.

2.3 MxKernel

Durch die zunehmende Komplexität an Daten, unter anderem aus dem Bereich des *Internet of Things*, wird ein gewaltiger Druck auf Datenbanken und Systeme erzeugt, um nach wie vor gute Performance zu liefern. Diese Daten sollen skalierbar und in Echtzeit verarbeitbar sein. Moderne Multiprozessor- und Mehrkernsysteme sind weitestgehend in der Lage diese Ansprüche auch zu erfüllen, doch ist dies ohne ausreichend starke und hierauf angepasste Software und Betriebssysteme nicht effizient nutzbar. Linux unterstützt Mehrkernsysteme schon seit längerem und skaliert durchaus gut [10]. Doch ein so generalisiertes Betriebssystem ist für die komplexen Datenmengen und die Ansprüche, die an sie gelegt werden, meist mehr hinderlich als hilfreich [1] und aus diesem Grund werden Betriebssysteme und Kernel entwickelt, die sich nur diesen Aufgaben stellen.

Das Projekt *MxKernel: A Bare-Metal Runtime System for Database Operations on Heterogeneous Many-Core Hardware* [16], dass an der TU Dortmund entwickelt wird, versucht die oben genannten Aufgaben zu bewältigen. Das Ziel ist es also große Mengen an Daten leistungsfähiger auf modernen Multiprozessor- und Mehrkernsysteme zu verarbeiten. Die *bare-metal* Laufzeitumgebung MxKernel, ein leichtgewichtiger Kernel, stellt die Grundlage hierfür dar und stellt grundlegende Methoden bereit um Aufgaben abzuarbeiten und diese zu dokumentieren, macht jedoch einige Veränderungen zu herkömmlichen Systemen. Zudem unterstützt der MxKernel von Grund auf die NUMA-Architektur, die Implementierung muss dies jedoch beachten und den Speicher danach auslegen.

In herkömmlichen Systemen wird, sobald Nebenläufigkeit eingesetzt werden möchte, auf Threads gesetzt. Threads sind langlebige Teilprozesse einer Anwendung, die unabhängig voneinander Arbeit auf der Anwendung erledigen und einen eigenen Stapelspeicher besitzen. Den Threads wird vom System mittels eines Schedulers ein Kern zugeordnet und nach einer bestimmten Strategie Rechenzeit auf diesem Kern zugewiesen. Nach Ablauf dieser Zeit wird der laufende Thread verdrängt, so dass ein anderer Thread auf diesem Kern seine Arbeit erledigen kann. Im Rahmen von Multiprozessor- und Mehrkernsystemen werden Threads eingesetzt um mehrere Kerne mit Arbeit zu versorgen. Dies erzeugt jedoch einen großen Nachteil, sobald mehr als ein Thread auf beispielsweise die gleiche Stelle einer Tabelle zugreifen möchten um diese zu verändern. Die Tabelle muss abgesichert werden, damit durch gleichzeitige Operationen keine Ergebnisse entstehen, die nicht gewünscht sind. Dies führt, je nach Granularität der Sperren, dazu, dass das System nur noch sehr schlecht skaliert, da sehr viele Threads darauf warten müssen, dass eine Sperre freigegeben wird und so während ihrer Rechenzeit nur aktiv warten können.

MxKernel MxKernel verfolgt einen anderen Ansatz, um einen Kern mit Arbeit zu befüllen. Die sogenannten MxTasks abstrahieren vom Thread-Modell und sind abgeschlossene, atomare - es findet also keine Verdrängung statt - Einheiten Arbeit, die nur einen

kleinen Teil bewältigen, beispielsweise eine Operation aus einfügen, suchen und löschen. Diese Tasks können einem Kern zugewiesen werden und können diesen während der Laufzeit ändern. Dies ist zum Beispiel auf einer Hashtabelle ausnutzbar, wenn die Tabelle nach Kernen aufgeteilt ist. Durch die atomare Ausführung der Tasks ergibt sich ein weiterer nützlicher Vorteil gegenüber dem Thread-Modell. Ein Task benötigt keinen eigenen Stapelspeicher, für beispielsweise Rücksprungadressen und kann alle notwendigen Speicherstellen im Cache des Kerns vorhalten.

Um die MxTasks abzuarbeiten besitzt im MxKernel Modell jeder Kern eine Warteschlange, in dem Tasks eingereiht und nach dem FIFO Verfahren entnommen werden, bis keine Aufgabe mehr vorhanden ist. Durch diese Warteschlange und der atomaren Ausführung eines Tasks kann komplett auf Sperrverfahren verzichtet werden, wenn die Arbeit so verteilt ist, dass sich unterschiedliche Kerne nicht die selbe Ressource teilen. Ob dies eine bessere Skalierbarkeit und bessere Laufzeiten gegenüber anderen Modellen insbesondere Modellen mit einem Sperrverfahren erzeugt soll im Rahmen dieser Arbeit anhand von Hashing erörtert werden.

MxOS Aufbauend auf dem MxKernel wurde im Rahmen einer Masterarbeit [15] das Betriebssystem MxOS entwickelt, welches die MxTasks zu regulären Threads erweitert. Diesen Threads wird wie normalen Threads Rechenzeit von einem Scheduler zugewiesen und werden nach dieser Rechenzeit verdrängt. Die Rechenzeit, die ein einzelner Thread erhält lässt sich vom Benutzer selbst bestimmen. Sie nutzen, wie die MxTasks, eine Warteschlange nach dem FIFO Verfahren, um den nächsten Thread zu ermitteln, welcher Rechenzeit erhält. Zudem muss deswegen ein Sperrverfahren benutzt werden, um die Ressourcen vor gegenseitigem Zugriff zu schützen. Dieses Sperrverfahren kann durch Semaphore dargestellt werden.

Kapitel 3

Implementierung

Der zuvor vorgestellte MxKernel abstrahiert vom üblichen Thread-Modell auf die MxTasks. Um dieser Umstellung gerecht zu werden, muss bei der Implementierung hierauf gesondert Acht genommen werden. Ein Task kann nicht zum Lastausgleich einem Kern zugeordnet werden, welcher schlecht ausgelastet ist. Dennoch kann es in diesem Modell dazu kommen, dass ein weiterer Task auf einem anderen Kern gleichzeitig die selben Ressourcen nutzen möchte. Es muss in diesem Fall darauf geachtet werden, dass die gleichen Ressourcen immer vom gleichen Kern behandelt werden. Beim Thread-Modell ist dies kein Muss, da dort ohnehin mit Sperrverfahren gearbeitet wird. Auf einer Hashtabelle ist die Zuteilung eines Bereichs zu einem bestimmten Kern besonders einfach, da die Hashfunktion bereits vorgibt, welcher Teil der Hashtabelle durch die auszuführende Aufgabe betrachtet wird.

Zunächst werden in diesem Kapitel die beiden Hashingvarianten genauer erläutert, hierbei wird auf Speicherallokation, allgemeine Funktionalität und Änderungen, die sich durch eine Mehrkernunterstützung ergeben, eingegangen. Im weiteren Verlauf wird behandelt, welche konzeptionellen Unterschiede sich durch diese Hashingvarianten für das Task-Modell des MxKernels und das Thread-Modell ergeben und insbesondere, wie mit den Prinzipien des MxKernel eine gute Prozessorverwaltung erreicht werden kann. Diese werden im Kapitel 4 mittels eines Benchmarks evaluiert. Zum Schluss werden Komplikationen vorgestellt, die während der Implementierung dazu führten, dass diese Idee nicht weiter verfolgt wurde und es wird ein Ansatz zur Lösung dieses Problems gegeben.

3.1 Hashingvarianten

Wie bereits in den Grundlagen in Abschnitt 2.1 erwähnt, werden im Rahmen dieser Arbeit das Hashing mit Verkettung und das Hashing mit offener Adressierung betrachtet. Beide Verfahren benötigen auf der untersten Schicht, also im Bereich wie eine Operation auf der Hashtabelle ausgeführt wird, kaum eine Anpassung an die geänderten Bedingungen, dass diese auf Mehrkernprozessoren funktionieren sollen.

Für beide Methoden wird die gleiche Hashfunktion verwendet, welche den Key per modulo auf die Anzahl der Buckets abbildet. Dies bedeutet, dass für den Key 42 und einer Bucketanzahl von 32 der Hashwert 10 wäre. Es ist keine kompliziertere Hashfunktion nötig, da die später bearbeiteten Werte zufällig erzeugt werden und deshalb eine bestimmte Verteilung auf der Hashtabelle nicht vorhersehbar ist.

Um Hashing für Mehrkernsysteme effizient einzusetzen ist, wie auch bei jeder anderen Struktur, notwendig, dass unterschiedliche Kerne nicht auf die gleichen Ressourcen zugreifen. Dies kann bei einer Hashtabelle sehr einfach erreicht werden, indem die Tabelle in Bereiche unterteilt wird, welche einem Kern zugewiesen werden können. Der ausführende Kern kann dann über den Key berechnet werden, welcher abgearbeitet wird.

Im folgenden werden nun die Implementierungen der beiden Hashingvarianten genauer vorgestellt, auf Unterschiede der Varianten eingegangen und worauf im Hinblick auf die Implementierung für Multikernsysteme geachtet werden muss.

3.1.1 Hashing mit Verkettung

Wie bereits in den Grundlagen in Abschnitt 2.1.1 erklärt, benötigt Hashing mit Verkettung an jeder Stelle der Hashtabelle einen Bucket, welcher eine Datenstruktur wie zum Beispiel eine Liste oder einen Baum enthält. Im Falle dieser Arbeit wurde sich für eine einfach verkettete Liste entschieden, welche aus *Node* Objekten besteht, die einen Key *key* und einen Zeiger *next* auf das nächste Node Objekt in der Liste haben. Die Tabelle selbst besteht aus einem Feld, genannt *buckets*, von Zeigern auf das erste Objekt eines Buckets, sowie einem Wert *bucketCount*, welcher die Anzahl der Buckets speichert. Die Hashtabelle hält zudem ein extra Node Objekt pro Bucket vor, welches das Ende dieser Liste markiert und einen Key von 0 gespeichert hat.

Möchte man nun ein Objekt in die Hashtabelle einfügen, muss der Bucket mittels der Hashfunktion hierfür berechnet werden und es muss überprüft werden, ob dieses Objekt bereits im Bucket vorhanden ist. Diese Methoden werden in den Algorithmen 3.1 (Suche) und 3.2 (Einfügen) gezeigt. Entfernen funktioniert ähnlich wie Suche, nur wird das Node Objekt D, welches zu dem zu Entfernenden Key gehört, noch aus der Liste entfernt. Dies geschieht wie üblich in einer Liste durch Umsetzen des Zeigers, sodass D nicht mehr erreichbar ist.

Auffällig bei einer Einfüge-Operation ist, dass bei der Eingabe anstatt eines Key Objekts ein Node Objekt mitgegeben wird. Dies wurde deshalb so gewählt, da beim Einfügen ein neues Listenobjekt erstellt werden muss, welches eigenen Speicher benötigt. Dies ist problematisch, da die Speicherallokation ein Sperrverfahren benötigen würde, sodass nicht unterschiedliche Kerne auf den gleichen Speicher zugreifen. Da jedoch Sperrverfahren zusätzliche Zeit kosten und damit den Durchsatz der Hashtabelle senken, werden einzufü-

search(k,b,T)
Eingabe: Key k, bucketCount b, buckets T
Ausgabe: **true**, wenn k gefunden, **false** sonst

```
1: bucket ← k mod b
2: if T[bucket].key = 0 then
3:    return false
4: end if
5: for all entries ∈ T[bucket] do
6:    if entries.key = k then
7:       return true
8:    end if
9: end for
10: return false
```

Algorithmus 3.1: Der Algorithmus zeigt eine Such Operation eines Keys für Hashing mit Verkettung.

insertEntry(n,b,T)
Eingabe: Node n, bucketCount b, buckets T
Ausgabe: **true**, wenn n noch nicht eingefügt wurde, **false** sonst

```
1: bucket ← n.key mod b
2: if search(n.key, b, T) then
3:    return false
4: else
5:    tmp ← T[bucket]
6:    T[bucket] ← n
7:    T[bucket].next ← tmp
8:    return true
9: end if
```

Algorithmus 3.2: Der Algorithmus zeigt eine Einfüge Operation einer Node für Hashing mit Verkettung.

gende Nodes bereits zu Beginn erstellt, sodass diese zur Laufzeit einfach eingefügt werden können.

Bei Mehrkernsystemen kann beim Hashing mit Verkettung, sofern Sperrverfahren zulässig sind, darauf verzichtet werden jedem Bucket einem bestimmten Kern zuzuordnen. Da ein Wert immer in genau dem Bucket vorhanden seien wird, welcher auch berechnet wurde, können die einzelnen Buckets jeweils mit einer Sperre abgesichert werden.

3.1.2 Hashing mit offener Adressierung

Aufbauend auf den Grundlagen in Abschnitt 2.1.2 wird bei Hashing mit offener Adressierung wieder auf Buckets zurückgegriffen, welche aber in ihrer Größe beschränkt sind. Für die Implementierung dieser Hashtabelle wurde sich der Einfachheit halber entschieden, dass in jeden Bucket nur ein Key gehasht werden kann. Sondierung wird durchgeführt indem das nächste Element der Tabelle überprüft wird, es wird also lineare Sondierung durchgeführt. Die Tabelle selbst besteht aus einem Feld *entries* von *Hash* Objekten und besitzt einen Integer *size*, welcher die Größe der Hashtabelle speichert. Ein Hash Objekt besitzt einen Key *key*, sowie eine boolesche Variable *deleted*, die angibt, ob an dieser Stelle ein Key entfernt wurde.

Im Hinblick auf die Implementierung für Mehrkernsysteme muss beachtet werden, dass wiederum nicht mehrere Kerne gleichzeitig in den selben Bucket schreiben können. Bei dieser Variante kommt nun noch das Problem der Sondierung hinzu. Für ein Modell mit Sperrverfahren wäre eine Sperre für einen einzelnen Bucket also äußerst kostenintensiv. Deshalb wird die Tabelle auch für Modelle mit Sperren in zusammenhängende Bereiche unterteilt, die jeweils durch eine Sperre abgesichert werden können.

Um einen Key initial einzuhashen benötigt die Tabelle nun den Bereich, auf dem sich der Key befindet. Dann kann der Key an der richtigen Stelle eingehasht werden, es wird ausgegeben, dass der Key bereits in der Tabelle vorhanden ist oder es wird ausgegeben, dass dieser Key nicht in diesem Bereich eingehasht werden kann. Wenn letzteres zurückgegeben wird, dann wird dieser Key auf den nächsten Bereich übergeben. Genaueres hierzu im Abschnitt 3.2. Sollte ein Key übergeben werden, der mit der Hashfunktion diesem Bereich zugeordnet ist, so wird die Methode an der Stelle der Hashfunktion beginnen einen freien Platz zu suchen. Die anderen beiden Operationen funktionieren ebenfalls so wie in Abschnitt 2.1.2 beschrieben. Algorithmus 3.3 zeigt zur Veranschaulichung eine Einfüge-Operation bei dieser Methode.

Wenn nun ein Key auf keinem Bereich der Hashtabelle mehr eingefügt werden kann, dann muss ein Rehashing ausgeführt werden. Ein Rehashing extrahiert alle Werte aus der alten Tabelle, welche nach wie vor vorhanden sind, und hasht diese in eine neue Tabelle ein, welche doppelt so viele Einträge einhashen kann, wie die alte Tabelle. Da für die neue Tabelle extra Speicher benötigt wird, kann diese, um Sperrverfahren zu vermeiden, nur auf

insertEntry(s,e,k,sz,T)

Eingabe: Start s, Ende e, Key k, size sz, entries T

Ausgabe: 1, wenn Key noch nicht eingefügt wurde. -1, wenn Key bereits eingefügt wurde. 0, wenn Ende des Bereichs erreicht ist.

```
1:  hashedKey ← (k mod sz)
2:  if (hashedKey > s)and(hashedKey ≤ e) then
3:     count ← hashedKey
4:  else
5:     count ← s
6:  end if
7:  while count ≤ e do
8:     if T[count].key = k then
9:        return −1
10:    end if
11:    if (T[count].key = 0)or(T[count].deleted = true) then
12:       T[count].key ← k
13:       T[count].deleted ← false
14:       return 1
15:    end if
16:    count ← count + 1
17: end while
18: return 0
```

Algorithmus 3.3: Der Algorithmus zeigt eine Einfüge Operation eines Keys für Hashing mit offener Adressierung.

einem Kern erstellt werden. Wenn ein Sperrverfahren eingesetzt werden möchte, so kann ein Rehashing abgesichert werden, indem sichergestellt wird, dass während des Rehashings keine weiteren Aufgaben abgearbeitet werden. Hierfür kann beispielsweise der ausführende Kern alle Sperren der einzelnen Bereiche halten. Für ein Rehashing ohne Sperrverfahren muss ein Algorithmus entwickelt werden, sodass die Tabelle durch ein Rehashing nicht verfälscht wird. Die Entwicklung eines solchen Algorithmus ist zumeist sehr aufwendig und ein Beweis zur Richtigkeit ist schwer zu erbringen. Eine Idee, wie dies umgesetzt werden kann, wird in Abschnitt 3.3 betrachtet.

3.2 Prozessorverwaltung

Um zu verhindern, dass verschiedene Kerne im MxKernel die gleichen Bereiche der Hashtabelle bearbeiten, müssen Tasks auf dem richtigen Kern gestartet werden. Dies wird insbesondere dann zu einer wichtigen Aufgabe, wenn Operationen auf der Hashtabelle initial nicht dem Kern zugeordnet werden, welcher den benötigten Bereich bearbeitet, sondern zufällig sind. Dies kann im Task-Modell sehr einfach gelöst werden, da ein Task nicht an einen bestimmten Kern gebunden ist und zur Laufzeit verändert werden kann. Dadurch besteht nicht nur die Möglichkeit auf Sperrverfahren zu verzichten, sondern es wird auch im Cache eines Kerns nur sehr selten ein Eintrag gehalten, welcher nie auf diesem Kern bearbeitet wurde. Im Thread-Modell wird sich die Zuordnung eines Threads zu einem Kern während der Laufzeit nicht verändern. Doch aufgrund des vorhandenen Sperrverfahren wird in diesem Modell der Zuordnung einer Aufgabe zu einem bestimmten Kern, wie im Task-Modell angewandt, keine größere Bedeutung zugeschrieben.

Dieser Abschnitt wird aufzeigen, wie eine Operation im Task- und Thread-Modell abgearbeitet wird und welche Vorteile sich durch die verschiedenen Modelle ergeben. Die Anweisungen werden von MxTasks oder den Threads des MxOS Systems aus Abschnitt 2.3 ausgeführt. Dabei wird sich in diesem Abschnitt auf das Hashing mit Verkettung beschränkt.

3.2.1 Threads

Threads sind langlebige Prozesse, welche zur Laufzeit ihre Zugehörigkeit zu einem Kern nicht verändern werden. Diese feste Zuordnung zu einem Kern kann durch die Aufteilung der Hashtabelle in Bereiche ausgenutzt werden, wenn auf einem Thread nur Aufgaben ausgeführt werden, welche dem Bereich seines Kerns zugeordnet sind. Dies erübrigt jedoch nicht die Notwendigkeit eines Sperrverfahrens, da ein Thread zugunsten eines anderen Threads auf diesem Kern verdrängt werden kann, welcher unter Umständen den gleichen Bucket behandelt. Um dies zu verhindern wird für das Thread-Modell ein Sperrverfahren mittels Semaphoren eingesetzt, indem jedem Bucket eine Semaphore zugewiesen wird,

welche so lange von einem Thread gehalten wird, wie Arbeit auf diesem Bucket erledigt wird.

Wenn Anweisungen zufällig an die Kerne verteilt werden, ist von vornherein nicht klar, zu welchem Bucket diese gehören. Dies produziert durch das Cache-Kohärenz Protokoll Overhead, sobald ein Thread auf eine Cache-Line zugreift, welche durch einen anderen Kern bearbeitet wurde und damit erst auf die bearbeitete Cache-Line warten muss. Dies wird insbesondere dann zu einem großen Problem, wenn diese Threads sich auf unterschiedlichen NUMA-Knoten befinden.

Wie ein Thread seine Arbeit auf der Hashtabelle ausführt wird in Algorithmus 3.4 dargestellt. Dabei enthält eine Aufgabe einen Key k und eine Operation o. Aufgrund des vorhandenen Sperrverfahrens kann dieser Algorithmus sehr einfach gehalten werden. Sobald in einem Thread eine Semaphore einen Bereich sperrt oder freigibt, wird vom MxOS System diese Semaphore in Form eines sogenannten Syscall-Task an den Kernel übergeben und der derzeit laufende Thread gibt seine Kontrolle an den Kernel ab. Dieser bearbeitet die Semaphore und teilt, falls diese frei war, dem Thread wieder Rechenzeit zu. Sollte diese nicht frei gewesen sein, wird dem nächsten lauffähigen Thread in der Warteschlange des Kerns Rechenzeit zugewiesen.

```
1: while Aufgabenliste ≠ ∅ do
2:     erhalte Aufgabe a
3:     b ← Bucket von Key a.k berechnen
4:     Bucket b sperren
5:     Operation a.o auf Hashtabelle mit Key a.k ausführen
6:     Bucket b freigeben
7: end while
```

Algorithmus 3.4: Dieser Algorithmus zeigt die Hauptschleife für einen Thread im Thread-Modells für Hashing mit Verkettung.

3.2.2 Tasks

Im Task-Modell können einige Vorteile der MxTasks ausgenutzt werden. Da ein einzelner Task nicht verdrängt und immer komplett ausgeführt wird, können sich Tasks auf dem gleichen Kern einen Speicher teilen, da für einen einzelnen Task beispielsweise keine Rücksprungadresse hinterlegt werden muss. Indem die Hashtabelle nun in gleichgroße Teile aufgeteilt und jeweils einem Kern zugeordnet wird, kann eine Eigenschaft der Hashtabelle ausgenutzt werden, um Sperrverfahren zu umgehen. Sobald der Key bekannt ist, ist auch bekannt in welchem Bucket und damit insbesondere auf welchem Kern diese Anweisung bearbeitet wird. Um nun eine Anweisung abzuarbeiten wird diese in zwei Phasen unterteilt. In der initialen Phase holt sich der Task auf dem Kern, auf dem er zuerst gestartet wurde,

eine Aufgabe und bereitet nun für die zweite Phase vor. Hierbei wird die Methode gesetzt, die auf der Hashtabelle ausgeführt werden soll, also einfügen, löschen oder suchen, es wird anhand des Keys der Bucket und damit der Kern berechnet und in die Warteschlange des berechneten Kerns eingefügt. Sobald dieser Task aus der Warteschlange entnommen wird, kann seine Anweisung durchgeführt werden. Sobald diese abgeschlossen ist, ändert der Task seinen Kern wieder auf den Kern, auf dem die initiale Phase bearbeitet wurde und reiht sich wieder in diesem Kern ein. Danach startet er, sobald dieser aus der Warteschlange entnommen wird, wieder die initiale Phase. Siehe hierzu Algorithmus 3.5, wobei eine Aufgabe aus einer Operation und einem Key besteht. Es zeigt sich für diese beiden ausgeführten Tasks die gleiche Syntax. Zuerst wird ihre bestimmte Aufgabe ausgeführt und danach die neue Aufgabe vorbereitet, der Kern für diese bestimmt und auf eben diesem Kern eingereiht. Man sieht wie beim Thread-Modell, dass zwischen der Vorbereitung einer Aufgabe und ihrer Ausführung möglicherweise eine andere Aufgabe vorbereitet oder bearbeitet wird. Im Thread-Modell geschieht dies durch die Ausführung einer Semaphore, im Task-Modell geschieht dies, da die auf der Hashtabelle ausgeführte Operation einen eigenen Task darstellt.

Initiale Phase:

1: erhalte Aufgabe a
2: setze für diesen Task t Operation $a.o$ auf der Hashtabelle mit Key $a.k$
3: $cpu \leftarrow$ Kern für $a.k$ bestimmen
4: t in Warteschlange von Kern cpu einfügen

Ausführung der Operation auf Hashtabelle:

1: $a.o$ mit $a.k$ auf Hashtabelle ausführen
2: setze für diesen Task t initiale Operation (Aufgabe a erhalten)
3: $cpu \leftarrow$ initialen Kern bestimmen
4: t in Warteschlange von Kern cpu einfügen

Algorithmus 3.5: Diese Algorithmen zeigen die beiden Phasen bei Abarbeitung eines Tasks für das Hashing mit Verkettung.

Die Aufteilung in diese zwei Phasen führt dazu, dass ein Kern in seinem Cache die Buckets behalten kann, welche ihm auch zugewiesen sind und nur sehr wenige Einträge aus anderen Buckets enthält, die zu Problemen im Cache-Kohärenz Protokoll führen können. Wenn eine Anweisung in der initialen Phase auf einen Kern geladen wird, auf dem sie nicht ausgeführt wird, sind die Zusatzkosten, um diese auf den richtigen Kern zu legen, nur dadurch gegeben, dass diese vom jeweiligen Speicher des NUMA-Knoten geladen werden müssen. Dies kommt daher, da in der ersten Phase keine Änderung an der Anweisung und

insbesondere an der zugehörigen Node gemacht wird, so wird auch die Cache-Line nicht verändert. Ein Ändern der Kerne kann jedoch auch einen Nachteil mit sich bringen, falls auf allen Kernen Anweisungen vorbereitet werden, die auf dem gleichen Kern ausgeführt werden. Wenn also zum Beispiel von 4 Kernen Anweisungen vorbereitet werden, die alle auf Kern 0 ausgeführt werden, kann es dazu kommen, dass die Warteschlangen der übrigen Kerne leer laufen. Dies führt zu einer sehr geringen Auslastung und damit schlechterem Durchsatz. Dieses Problem existiert beim Thread-Modell weniger, da die Zuordnung zu einem Kern nicht verändert wird und somit auf jedem Kern zumindest Arbeit läuft, auch wenn diese möglicherweise darauf warten muss, dass ein Bucket frei wird. Ob eine Erhöhung der Anzahl der Tasks pro Kern wirklich eine Verbesserung erbringt wird in Kapitel 4 evaluiert.

3.3 Rehashing auf MxKernel

In Abschnitt 3.1.2 wurde bereits erwähnt, dass die Entwicklung eines Rehashings in einem sperr freien Modell sehr aufwendig sein kann. Weshalb dies aufwendig ist und wie ein möglicher Algorithmus aussehen kann wird in diesem Abschnitt vorgestellt.

Zuvor wird jedoch erst vorgestellt, wie eine Aufgabe für das Hashing mit offener Adressierung im Task-Modell ausgeführt wird. Wie auch schon in Abschnitt beschrieben, sind die Tasks in 2 Phasen aufgeteilt, wobei die erste Phase identisch zum Hashing mit Verkettung ist, siehe hierzu Algorithmus 3.5. Die zweite Phase ist jedoch etwas komplizierter, da durch Sondierung ein Kern gewechselt werden kann. Ein Algorithmus für eine Einfüge-Operation in dieser zweiten Phase ist in Algorithmus 3.6 dargestellt. Sobald durch Sondierung ein Ende des Bereichs erreicht wird, wird die Anweisung in den nächsthöheren Kern eingereiht - bei 16 Kernen erfolgt bei Kern 15 der Wechsel auf Kern 0 - und in dem Bereich ausgeführt, der diesem Kern zugeordnet ist. Dies wird solange durchgeführt, bis der Task wieder in die Warteschlange des Kerns eingereiht werden soll, welcher zu Beginn über den Key bestimmt wurde. Bei den Operationen Suchen und Löschen wird der Task nun beendet, da der Key niemals gefunden werden konnte, und der Task kann auf seinem initialen Kern eine neue Aufgabe erhalten. Bei der Operation Einfügen wird nun, um den Task später doch einfügen zu können, ein Rehashing Task eingereiht.

Ein erster Entwurf des Rehashing Tasks sah vor diesen immer auf Kern 0 durchzuführen und den restlichen Kernen während des Rehashings zu verbieten weitere Operationen auf der Hashtabelle durchzuführen. Dies wäre durch ein Sperrverfahren sehr einfach möglich gewesen, indem die komplette Tabelle gesperrt wird, solange Rehashing die Tabelle neu aufbaut. Um dies auf dem Task-Modell ohne explizites Sperren zu erreichen, wurden die Tasks, die Operationen auf der Hashtabelle durchführen, während der Laufzeit des Rehashings, in die Warteschlange des Kerns eingehangen, der nach dem Rehashing dafür zuständig wäre diesen Task in die Hashtabelle einzuhashen. Diese Art, um das Problem

des Rehashings zu lösen verstößt jedoch im Grunde gegen die Idee des MxKernel, indem ein aktives Warten auf allen Kernen außer Kern 0 erzeugt wird, was zudem zu einer sehr geringen Auslastung führt, und das Rehashing kein kleines Stück Arbeit darstellt, welches von MxTasks gefordert wird, sondern die komplette Tabelle betrifft. Zudem führt dies auf Kern 0 zu gewaltigem Overhead, da alle Veränderungen aus den anderen Kernen betrachtet werden und dementsprechend alle veränderten Cache-Lines, die die Hashtabelle betreffen, aktualisiert werden müssen. Ein weiteres Problem stellen Tasks dar, die bereits laufen, während das Rehashing gerade gestartet wird. Diese werden zumindest die Arbeit auf dem aktuellen Bereich ausführen und somit eventuell einen Key löschen, welcher vom Rehashing dennoch wieder eingefügt wird oder einen Key einfügen, den das Rehashing nicht betrachtet. Also wird nicht nur gegen das Prinzip des MxKernel verstoßen sondern die Tabelle ist zudem noch inkonsistent. So konnte diese Idee verworfen werden, es wurde jedoch im Zeitraum dieser Arbeit keine bessere Lösung implementiert.

```
1: Operation a.o mit Key a.k auf Hashtabelle ausführen
2: if Operation erfolgreich or bereits eingefügt then
3:    setze für diesen Task t initiale Operation (Aufgabe a erhalten)
4:    cpu ← initialen Kern bestimmen
5:    t in Warteschlange von Kern cpu einfügen
6: else if Operation erreicht Ende des Bereichs then
7:    cpu ← derzeitiger Kern +1
8:    if cpu = Kern zu Beginn then
9:       t als Rehashing Task in Warteschlange von Kern 0 einfügen
10:   else
11:      t in Warteschlange von Kern cpu einfügen
12:   end if
13: end if
```

Algorithmus 3.6: Dieser Algorithmus zeigt die zweite Phase bei Abarbeitung einer Aufgabe beim Hashing mit offener Adressierung. Dabei wird sich auf die Einfüge Operation beschränkt.

Eine Idee für eine Lösung dieses Problems wäre eine Umgestaltung der Hashtabelle. Man unterteilt die Hashtabelle wieder in verschieden Bereiche, benutzt dieses mal jedoch zwei Hashfunktionen. Eine, um den Bereich zu ermitteln, welcher sich auch durch Sondierung nicht mehr ändern kann und eine weitere um innerhalb des Bereichs die Position des Keys zu bestimmen. Sollte nun eine Stelle bereits belegt sein, so wird lineare Sondierung durchgeführt. Sobald der gewählte Bereich nicht mehr befüllt werden kann, wird für den einzelnen Bereich ein Rehashing durchgeführt. So kann nach wie vor eine amortisierte Laufzeit von $O(1)$ erreicht werden und es ist kein Sperrverfahren notwendig, da auf einem Bereich niemals mehr als eine Operation durchgeführt wird. Diese Idee erinnert sehr stark an das Hashing mit Verkettung, bei dem nur der Unterschied besteht, dass die Buckets

keine feste Größe haben und in diese einfach eingefügt und nicht eingehasht wird. Aus diesem Grund wurde auch diese Idee nicht weiter verfolgt.

Kapitel 4

Evaluation

Im vorherigen Kapitel wurden die Hashingvarianten anhand der zugrundeliegenden Modelle für Nebenläufigkeit vorgestellt. Durch die folgende Evaluierung wird geprüft, wie gut die Modelle bei gleichen Bedingungen gegeneinander abschneiden und ob diese dem Anspruch gerecht werden skalierbar zu sein. Zudem werden einige Verbesserungen am Benchmark und einigen Einstellungen der Hashtabellen gemacht.

Um diese Methoden zu evaluieren werden Operationen auf der Hashtabelle durchgeführt und in jedem Schritt die Anzahl der Kerne und damit der Grad an Parallelität erhöht. Um den Durchsatz der jeweiligen Benchmarks zu ermitteln wird die Prozessorzeit mittels der **rdtsc**-Instruktion ausgelesen [3] und in Anweisungen pro Sekunde umgerechnet.

Neben dem MxKernel und MxOS System wird dieser Benchmark ebenfalls auf Linux durchgeführt, um einen Vergleichswert zu erhalten und die Performance einschätzen zu können. Zum Schluss wird der Benchmark noch mit der Intel® Threading Building Blocks [9] Bibliothek verglichen, die effiziente Implementierungen zur Nutzung von Mehrkernsystemen anbietet. Diese Bibliothek benutzt zur Nebenläufigkeit das klassische Thread Modell. Im folgenden wird jedoch zuerst der Versuchsaufbau beschrieben.

4.1 Versuchsaufbau

Der Benchmark benutzt für die Evaluierung das Hashing mit Verkettung. Das beste Ergebnis mit dieser Methode wird zudem mit dem Ergebnis, welches Hashing mit offener Adressierung für das Task-Modell lieferte, verglichen. Für diesen Benchmark wurden 24 Millionen Einfüge-Operationen gewählt und keine Such- oder Entfernen-Operationen. Um Threads und Tasks mit Arbeit zu befüllen wir von diesem Benchmark eine Schlange von Operationen, die sogenannte *Instruction Queue* benutzt, die zu Beginn angelegt und von Tasks oder Threads abgearbeitet wird. Die Einträge in dieser Schlange bestehen aus einer Operation (einfügen, löschen, suchen), sowie einem zufälligen Integer Key, welcher für die Operation benutzt wird und größer 0 ist. Eine Lösch- oder Suchoperation wird nur

auf einem Key ausgeführt, welcher bereits irgendwann einmal in der Schlange war, also mit Sicherheit bereits einmal in der Hashtabelle war. Mit genauerer Betrachtung der zufälligen Anweisungen können schon bereits bei Erstellung entschieden werden welchem NUMA-Knoten diese zugeordnet werden, um den Flaschenhals, der sich durch eine feste Verteilung auf einen Knoten ergibt, zu umgehen. Ein weiteres Problem der Instruction Queue, welches ebenfalls aus der NUMA-Architektur und dem Cache-Kohärenz Protokoll entsteht, ist ein Entnehmen der nächsten Anweisung. Diese wird über einen Zeiger geregelt, welcher von jedem Kern betrachtet und verändert wird und damit insbesondere in jedem Cache vorhanden ist. Dies führt zu rapiden Einbußen im Durchsatz der Hashtabelle, sobald der zweite NUMA-Knoten benutzt wird, siehe hierzu Abbildung 4.2.

Dieser Benchmark wird auf einem *Primergy RX200 S7* Server [7] des Herstellers *Fujitsu* durchgeführt. Das System besitzt zwei NUMA-Knoten mit jeweils einem Intel® Xeon® Prozessor, welcher die Prozessornummer E5-2690 trägt. Der Prozessor ist mit einer Rate von $2,90$ GHz getaktet und besitzt jeweils 8 Kerne und einen 20 MB großen L3-Cache, welcher von allen Kernen geteilt wird. Zudem besitzt jeder Kern einen eigenen L2-Cache der Größe 2MB und einen L1-Cache der Größe 512KB.

4.2 Verbesserung

Bereits vor dem Vergleich beider Modelle miteiander lassen sich Verbesserungen am Benchmark durchführen. Diese führten bereits zu deutlichen Zunahmen des Durchsatzes. Diese werden im folgenden vorgestellt und evaluiert.

Größe der Hashtabelle Die Größe einer Hashtabelle ist von entscheidener Wichtigkeit, insbesondere wenn diese nicht dynamisch wachsen kann. Im Falle des Hashings mit offener Adressierung wurde aufgrund der angesprochenen Probleme des Rehashings entschieden die Größe der Hashtabelle auf das doppelte der Anzahl der Operationen zu setzen. Dies führt dazu, dass nie ein Rehashing durchgeführt werden muss, da für einen Key immer ein freier Eintrag gefunden wird.

Für das Hashing mit Verkettung lässt sich die Anzahl der Buckets betrachten. Offensichtlich steigt der Durchsatz des Benchmarks, wenn mehr Buckets eingesetzt werden, aber ebenso steigt auch der Speicherverbrauch. So wurden verschiedene Größen, anteilig zur Anzahl der Operationen auf der Tabelle evaluiert. Die erhaltenen Ergebnisse werden in Abbildung 4.1 dargestellt. Es wurde sich schlussendlich für ein Verhältnis von einem Zehntel Buckets zu Operationen entschieden, da so im Schnitt maximal 10 Einträge pro Bucket vorhanden sind und diese in einer Liste noch schnell abgearbeitet werden können.

Verbesserung der Auslastung Wie bereits im vorherigen Kapitel angemerkt, können die Warteschlangen eines Kerns leerlaufen oder mit nur sehr wenig Aufgaben belegt sein,

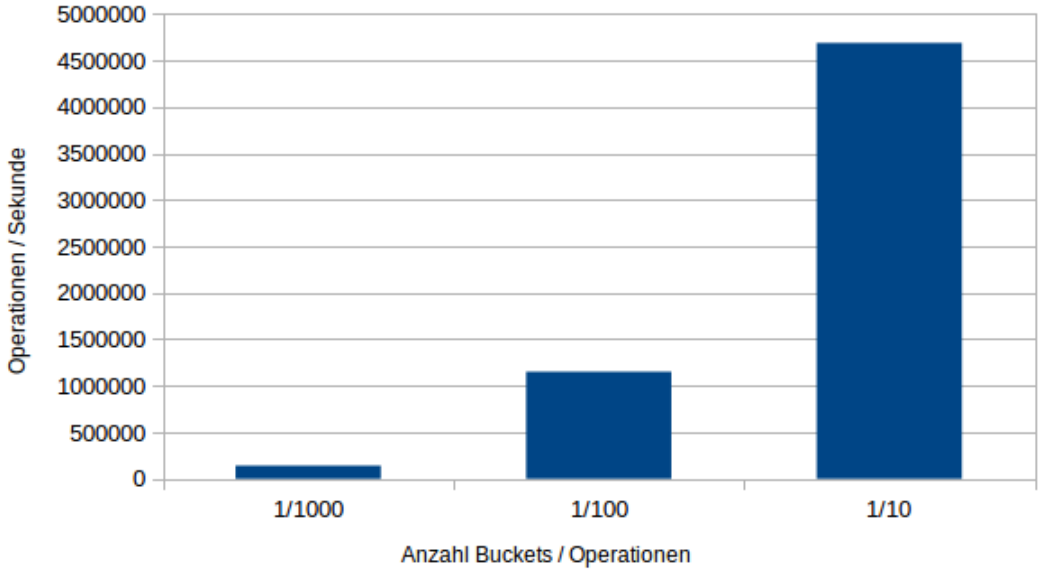

Abbildung 4.1: Diese Abbildung zeigt den Durchsatz, welcher bei Hashing mit Verkettung erreicht wird, um 4,8 Mio Operationen durchzuführen. Der Fokus hierbei liegt auf der Verbesserung, die erreicht wird, wenn die Anzahl an Buckets erhöht wird.

sobald mehrere Tasks für den gleichen Kern vorbereitet werden, und somit die Auslastung der Kerne senken. Indem nun zu Beginn des Benchmarks mehr Tasks pro Kern erstellt werden, also die Warteschlangen eines Kernes zu Beginn deutlich voller sind, sinkt die Wahrscheinlichkeit, dass ein Kern gar keine Arbeit zugewiesen bekommt. Siehe hierzu Abbildung 4.2, diese zeigt eine Verbesserung des Durchsatzes von im Schnitt 15%, wenn anstatt 4 Tasks pro Kern 100 Tasks pro Kern benutzt werden.

NUMA-Effekt der Instruction Queue Wie schon aus Abbildung 4.2 ersichtlich ist, führt eine schlechte Ausnutzung der NUMA-Architektur dazu, dass der zugrundeliegende Benchmark einen deutlich schlechteren Durchsatz erreicht, sobald der zweite NUMA-Knoten Arbeit zugeteilt bekommt. Dieser Effekt entsteht durch den geteilten Zeiger, welcher die nächste Anweisung angibt, der von jedem Kern verändert wird und dementsprechend kohärent auf beiden NUMA-Knoten gehalten werden muss. Um dies zu umgehen kann für jeden Kern ein Zeiger erstellt werden und die Instruction Queue ebenfalls nach Kernen unterteilt werden. Welchen Vorteil diese Veränderung mit sich bringt zeigt sich in Abbildung 4.3. Die nur sehr geringe Verbesserung, die sich bei Hinzunehmen des zweiten NUMA-Knotens gegenüber nur einem einzigen NUMA-Knoten bei der neuen Instruction Queue zeigt, lässt sich auf weitere NUMA-Effekte zurückführen. Diese entstehen möglicherweise, weil die Anweisungen in der Instruction Queue nicht nach der Cache-Line ausgerichtet werden und somit die Möglichkeit des False Sharing besteht. Dieser Effekt kann

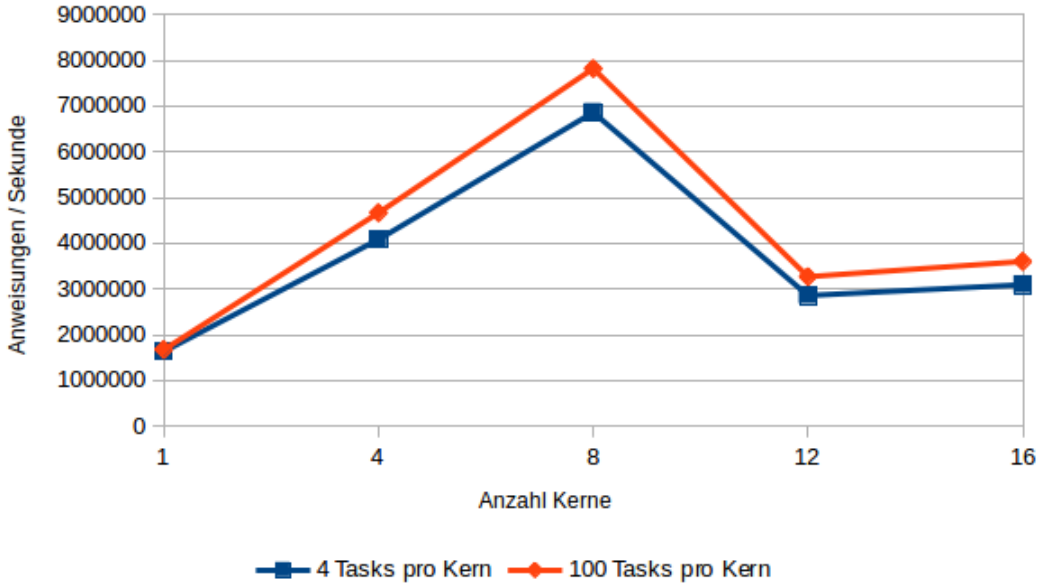

Abbildung 4.2: Diese Abbildung zeigt welche Auswirkungen ein zweiter NUMA-Knoten oder Prozessor auf den Durchsatz des Benchmarks hat, wenn Objekte von beiden Prozessoren verändert werden. Siehe hierzu den Abfall von Operationen pro Sekunde bei 12 und 16 Kernen. Zudem lässt sich die Verbesserung erkennen, die eine Erhöhung der Anzahl der Tasks pro Kern mit sich bringt. Hierbei wurden 24 Millionen Einfüge Operationen auf der Hashtabelle ausgeführt.

auch verbessert werden, indem Daten bei ihrer Erstellung dem passenden NUMA-Knoten zugewiesen werden, dies wurde jedoch nicht weiter betrachtet.

4.3 Vergleiche

Der entwickelte Benchmark benutzt wie zuvor angesprochen die jeweiligen Modelle des MxKernels und des MxOS Systems, welche unterschiedliche Ansätze verfolgen. In diesem Abschnitt soll der Benchmark auf dem MxKernel, dem MxOS System, Linux und den Intel® Threading Building Blocks evaluiert werden.

4.3.1 MxKernel mit MxOS

Um dem Benchmark bei zunehmender Anzahl an Kernen eine Möglichkeit zu bieten seine Leistung im Task-Modell des MxKernel zu steigern wird die Hashtabelle in so viele Teile unterteilt, wie Kerne vorhanden sind. Im Thread-Modell ist dies unkritisch, da nach wie vor nur einzelne Buckets gesperrt werden, somit entsteht hier kein Engpass. Für Hashing mit offener Adressierung ist so eine Aufteilung kritisch, da im schlimmsten Fall jeder Thread außer einer darauf warten muss, dass ein Bereich freigegeben wird.

Erwartungsgemäß wird der Benchmark auf dem MxOS System eine geringere Leistung liefern, da hier viele Einträge kohärent gehalten werden müssen und zudem durch Sema-

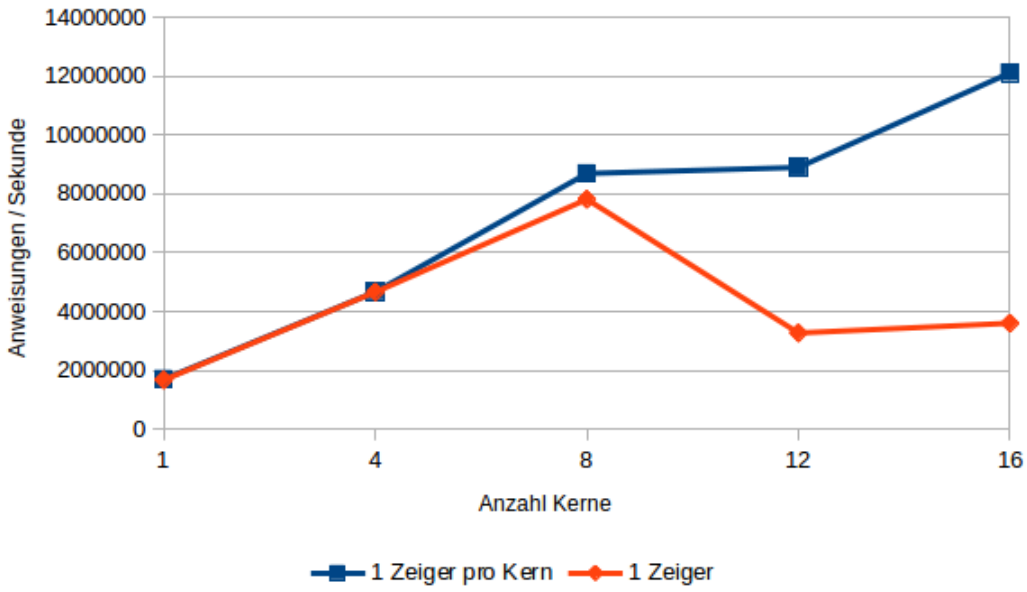

Abbildung 4.3: Diese Abbildung zeigt die Veränderung eines von allen Kernen veränderten Zeigers dazu jeweils einen eigenen Zeiger pro Kern zu nutzen. Diese Benchmarks benutzten jeweils 24 Millionen Einfüge Operation auf der Hashtabelle für Hashing mit Verkettung im Task-Modell mit jeweils 100 Tasks pro Kern.

phore weiterer Overhead entsteht, da Semaphore ebenfalls an die Warteschlange der Kerne in Form von Syscall-Tasks übergeben werden. Sobald ein solcher Syscall-Task übergeben wird, gibt der derzeit laufende Thread seine Kontrolle über den Kern ab und der Kernel bearbeitet die Semaphore. Wenn diese bereits gesperrt wurde kann einem lauffähiger Thread Rechenzeit zugewiesen werden, ansonsten wird dem vorherigen Thread wieder Rechenzeit zugewiesen. Ohne eine Synchronisation wird der Benchmark einen deutlich besseren Durchsatz liefern können, jedoch wird hierdurch die Hashtabelle verfälscht. Abbildung 4.4 zeigt die erwarteten Ergebnisse, wobei das Thread-Modell mit Sperrverfahren sogar nur 50% des Durchsatzes gegenüber des Task-Modells erbrachte, selbst bei der Nutzung von nur einem einzigen Kern. Im Vergleich hierzu erbringt die Variante ohne eine Synchronisation im Thread-Modell einen doppelt so guten Durchsatz wie das Thread-Modell mit Synchronisation. Da jedoch die Hashtabelle hierdurch ungültig wird - es befinden sich nach dem Benchmark bis zu 100 Einträge weniger in der Hashtabelle - kann dies nur unter speziellen Bedingungen so zum Einsatz kommen.

Auffällig ist, dass trotz der geringeren Leistung durch die Semaphoren der Benchmark auf dem MxOS System dennoch skaliert und damit bei Einsatz von mehr Kernen die erwartete Leistungssteigerung erbringt. Dagegen skaliert der MxKernel nur auf dem ersten NUMA-Knoten, also auf den ersten 8 Kernen. Danach erfolgt eine rapide Abnahme der Leistung. Da dieser Effekt im MxOS nicht auftritt, kann angenommen werden, dass im MxKernel Eigenschaften der NUMA Architektur unausgereift ausgenutzt werden. Wenn

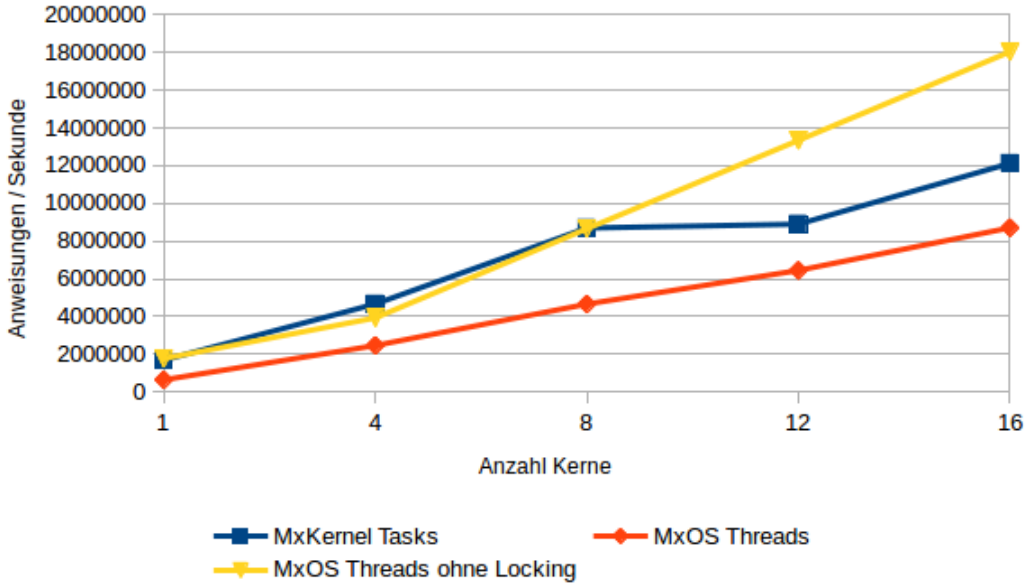

Abbildung 4.4: Diese Abbildung visualisiert die Messergebnisse des Hashings mit Verkettung unter dem MxKernel und dem MxOS mit, sowie ohne Sperrverfahren.

explizit betrachtet wird wie viel Leistung im Schnitt pro Kern geleistet wird, siehe hierzu Abbildung 4.5, so fällt auf, dass das Task-Modell für 12 und 16 Kerne einen weitestgehend gleichen Durchsatz pro Kern erreicht. Dies bedeutet, dass die Hashtabelle, trotz des Einknickens durch NUMA Effekte, skaliert. Wenn die NUMA Effekte des MxKernels außer Acht gelassen werden und nur das Ergebnis auf den ersten 8 Kernen betrachtet wird, so kann man erkennen, dass die Tasks einen ähnlichen Durchsatz erreichen wie die nicht synchronisierten Threads des MxOS. Deswegen kann angenommen werden, dass ein Task ohne großartigen Aufwand seinen Kern wechseln kann und es nicht ausschlaggebend ist, auf welchem Kern der Task gestartet wurde.

Vergleicht man nun das Hashing mit Verkettung mit dem Hashing mit offener Adressierung, welches ohne Rehashing durchgeführt wurde, sieht man, dass offene Adressierung bei nur einem genutzten Kern einen höheren Durchsatz hat, dies aber nur sehr langsam gesteigert wird, sobald mehrere Kerne eingesetzt werden. Siehe hier die Abbildung 4.6. Dies geschieht möglicherweise durch eine unzureichende Auslastung der Kerne, da viele Werte nicht mehr in den Bereich passen, welcher ihnen durch die Hashfunktion zugewiesen wurde, und müssen deshalb auf einem höheren Kern eingehasht werden. Dies führt dazu, dass die Warteschlangen eines Kernes mit Tasks voll laufen und die restlichen Kerne ohne oder mit sehr wenig Arbeit belastet sind.

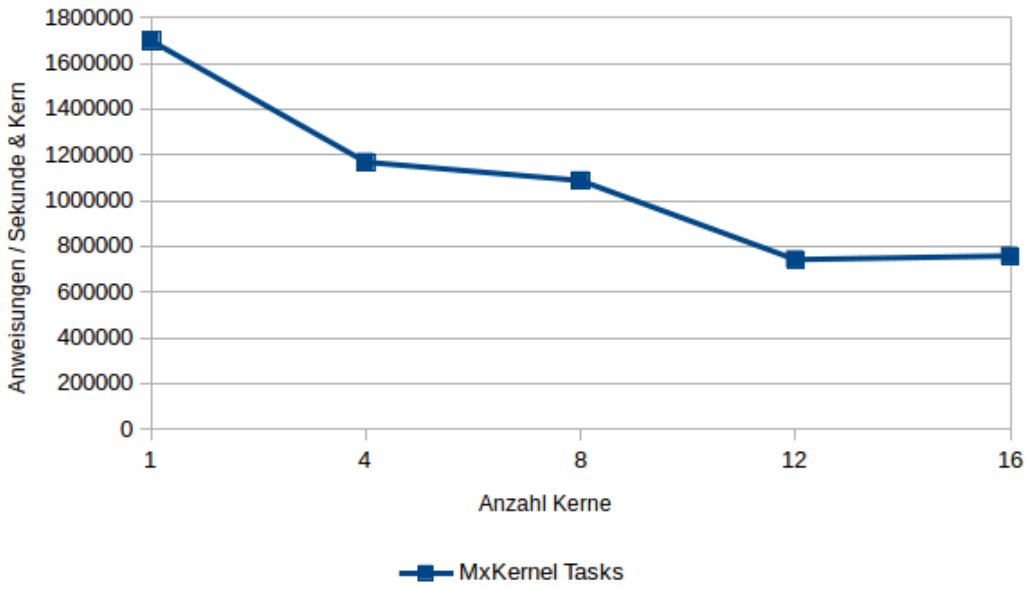

Abbildung 4.5: Diese Abbildung visualisiert die Messergebnisse des Hashings mit Verkettung unter dem MxKernel. Dabei wird der durchschnittliche Durchsatz pro Kern betrachtet.

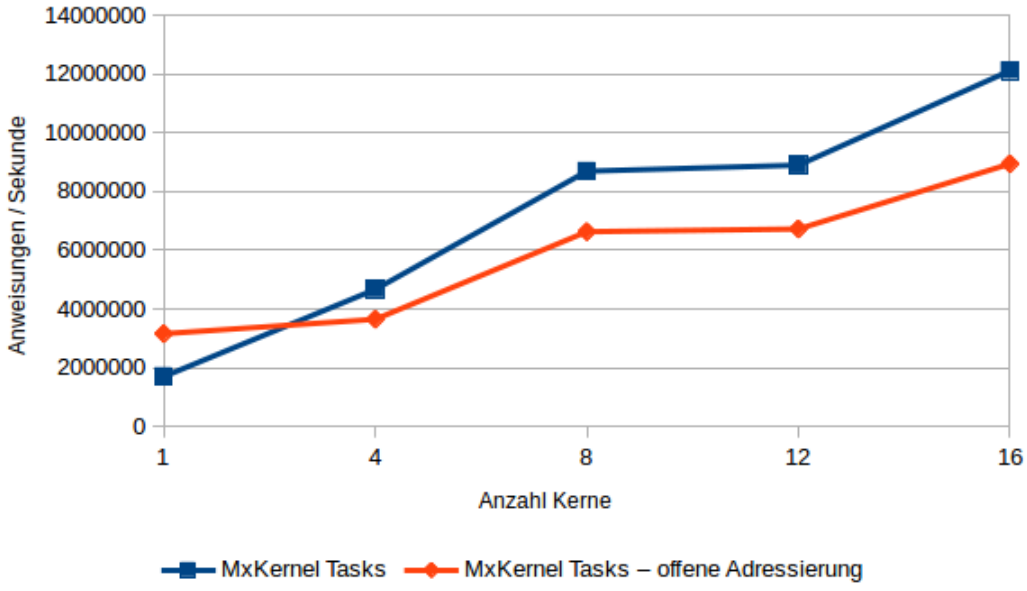

Abbildung 4.6: Diese Abbilung visualisiert die Messergebnisse des Hashings mit Verkettung und des Hashings mit offener Adressierung unter dem MxKernel.

4.3.2 MxKernel mit Linux

Im vorherigen Abschnitt konnte gezeigt werden, dass eine Hashtabelle im Task-Modell des MxKernel einen besseren Durchsatz erbringt als eine auf dem Thread-Modell des MxOS basierende Hashtabelle mit Semaphoren. Interessant dürfte zudem ein Vergleich zu einem aktuellen Betriebssytem sein, um die erhaltenen Ergebnisse vergleichen zu können. Hierfür wird die Linux Distribution *Ubuntu 17.10* verwendet, welche auf dem Linux-Kernel der Version $4.13.0-36$ beruht. Um diesen Vergleich erbringen zu können, können MxKernel und MxOS Anwendungen im Linux-Gastbetrieb nativ unter Linux gestartet werden. Für das Thread-Modell werden alle Systemaufrufe, wie Threads und Semaphore an das Äquivalent des Linux Systems weitergeleitet. Threads werden, wie unter MxOS an einen festen Kern gebunden, jedoch entscheidet das System, auf welchem Kern dieser Thread gestartet und ausgeführt wird. Im Gegensatz dazu wird das Task-Modell des MxKernel an einen Linux Thread pro vorhandenem Kern weitergeleitet, welcher die Tasks abarbeiten kann. Dabei ist der Kern wie auch schon im MxKernel festgelegt.

Die Ergebnisse bestätigen im Vergleich des Thread-Modells zwischen MxOS und Linux die zuvor genannten Probleme der Semaphore des MxOS Systems. Unter Linux erreicht der Benchmark durchweg 40% mehr Durchsatz. Siehe hierzu Abbildung 4.7. Daraus kann gefolgert werden, dass Semaphore im Linux System effizienter implementiert wurden. Generell zeigt sich in diesem Vergleich aber auch, dass die Ergebnisse des MxOS und die Ergebnisse in Linux gleich gut skalieren. Dieser Vergleich ist in Abbildung 4.8 verdeutlicht.

Die Messungen des Task-Modells für Linux offenbaren die gleichen Schwächen, wie zuvor schon angesprochen. Es ist der gleiche Leistungseinbruch wie im nativen MxKernel zu erkennen. Siehe hierzu Abbildung 4.9. Bis zu 8 Kernen bewegen sich die Unterschiede in beiden Systemen im 5% Bereich, wobei auf Linux ein besserer Durchsatz erreicht wird und beide Systeme skalieren. Sobald 12 Kerne eingesetzt werden bricht unter Linux der Durchsatz ein und es wird 15% weniger Durchsatz erreicht als beim MxKernel. Dies zeigt sich ebenfalls für 16 Kerne. Ein Grund hierfür könnte wie oben bereits geschrieben eine unzureichende Ausnutzung der NUMA Architektur des Task-Modells sein, welche unter der Linux Simulation noch stärker zum tragen kommt. Lässt man diesen Effekt außer Acht zeigt sich durchweg ein besserer Durchsatz unter Linux. Dies kann bedeuten, dass unter Linux bereits einige Komponenten effizienter implementiert wurden.

Aus den Ergebnissen ergibt sich, dass die Linux Threads für 16 Kerne den besten Durchsatz bieten und das Thread-Modell allgemein besser skaliert. Dies ist durch die nicht perfekte Ausnutzung der NUMA Architektur im Vergleich zum MxKernel und der Linux Tasksimulation etwas beeinflusst. Führt man den Benchmark nur auf einem NUMA-Knoten durch, so wird durch das Task-Modell ein besserer Durchsatz erreicht, welcher zudem eine bessere Skalierung bietet als das Thread-Modell. Dies ist positiv zu werten, da somit

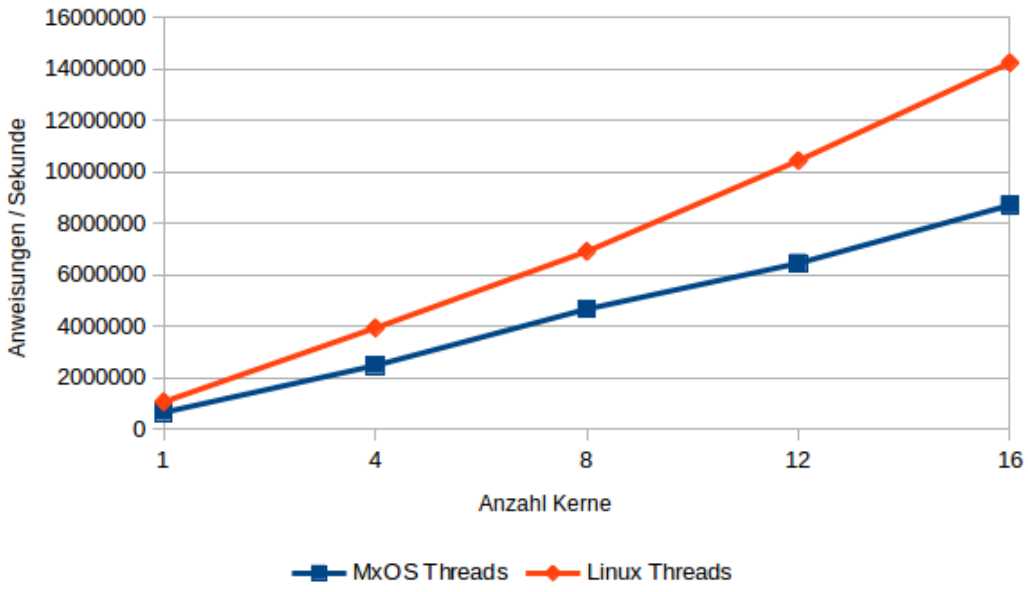

Abbildung 4.7: Diese Abbilung visualisiert die Messergebnisse des Hashings mit Verkettung unter dem MxOS System und Linux.

die grundlegende Idee des MxKernel einen Vorteil in Durchsatz gegenüber des bisher üblichen Thread-Modells bietet.

4.3.3 MxKernel mit Intel Threading Building Blocks

Im letzten Abschnitt wurde bereits ein Vergleich zwischen Linux und dem MxKernel, sowie dem MxOS System hergestellt. Für einen weiteren Vergleich wird in diesem Abschnitt die Intel® Threading Building Blocks [9] Bibliothek, kurz TBB, vorgestellt und gegen den MxKernel evaluiert. Diese bietet eine Reihe an Implementierungen für Mehrkernprozessoren an, um typische Probleme, wie zum Beispiel die Synchronisation, zu lösen. Dabei unterstützt TBB das typische Thread-Modell. TBB bietet für Hashing die sogenannte *concurrent_hash_map* an, welche gleichzeitige Aufrufe verschiedener Threads auf diese Tabelle ermöglicht. Dabei wird für einen Eintrag beim Einfügen ein sogenannter *accessor* erstellt, welcher für diesen Eintrag Sperren, sowie den Zugriff regelt. Sobald dieser Bucket beschrieben werden soll, wird der accessor für diesen Bucket die Sperre freigeben, um eine Einfüge-Operation durchzuführen. Bei erfolgreicher Durchführung wird der neue accessor nun Operationen für diesen Bucket übernehmen.

Um nun TBB mit dem MxKernel zu vergleichen wird für die TBB Implementierung die gleiche zufällig erzeugte Instruction Queue aus Abschnitt 4.2 gewählt, der Durchsatz wird ebenfalls in Prozessorzeit gemessen, welcher in Anweisungen pro Sekunde umgerechnet wird und es werden für jeden benutzten Kern ein Thread erstellt, welcher anteilig Anweisungen aus dieser Schlange erhält. Dieser Benchmark wurde unter Linux durchgeführt und

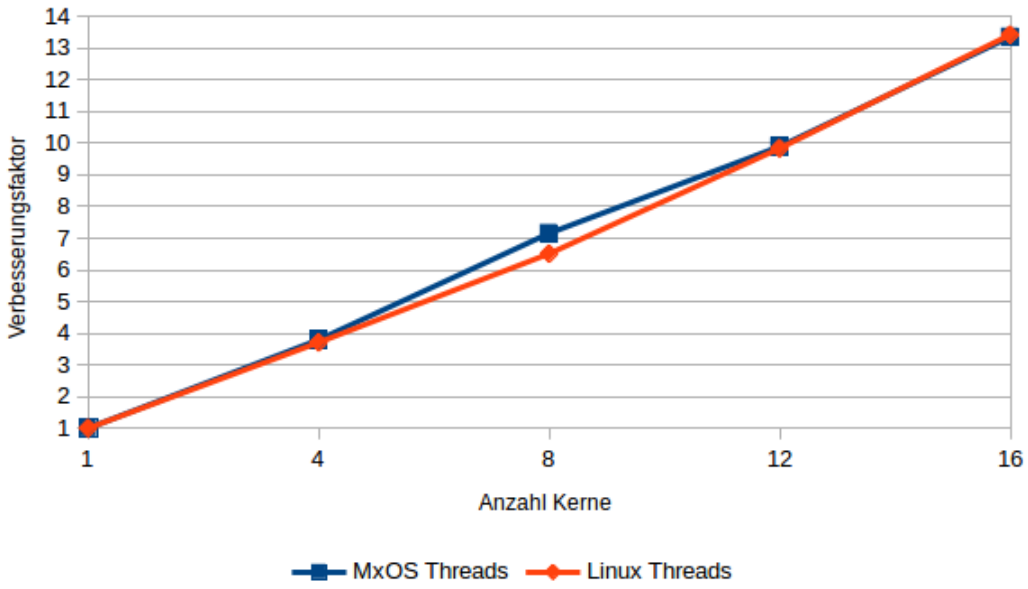

Abbildung 4.8: Diese Abbildung visualisiert den Faktor unter MxOS und Linux, wie sehr sich der Durchsatz der Hashtabelle verbessert, wenn mehr Kerne eingesetzt werden. Die Grundlage hierfür wird durch den Wert für einen Kern gegeben.

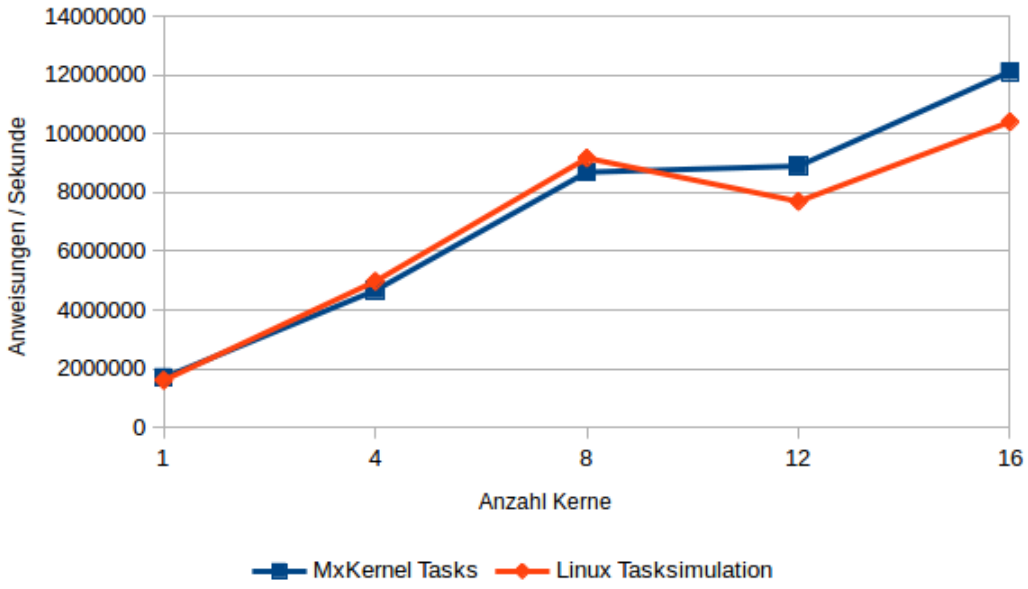

Abbildung 4.9: Diese Abbildung visualisiert die Messergebnisse des Hashings mit Verkettung unter dem MxKernel und der Task-Simulation unter Linux.

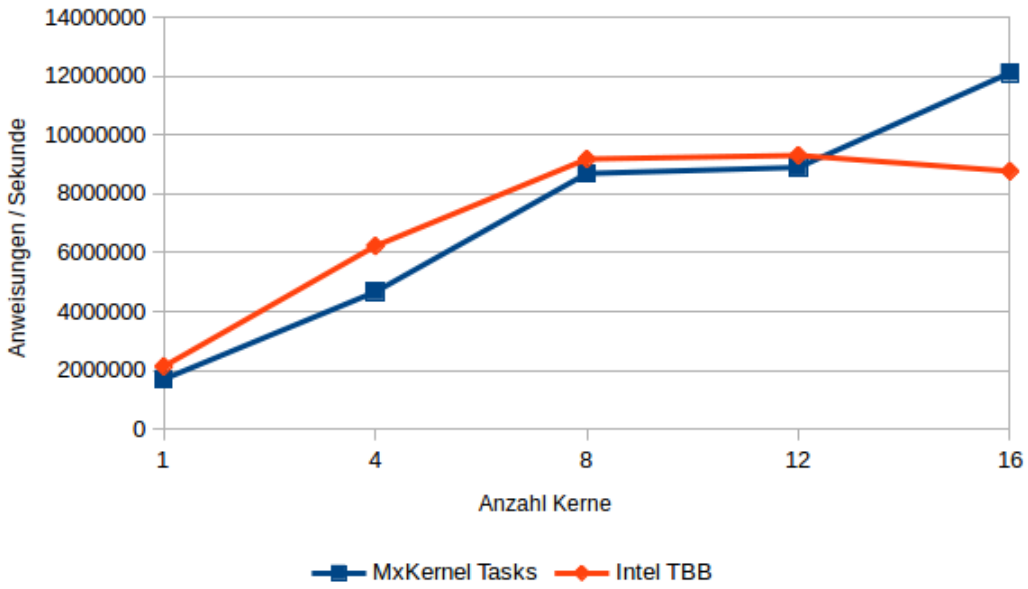

Abbildung 4.10: Diese Abbildung visualisiert die Messergebnisse des Hashings mit Verkettung unter dem MxKernel und der TBB Implementierung der concurrent_hash_map.

benutzt daher auch Linux Threads und Speicherverwaltung. Die Ergebnisse zeigen, dass die Threading Building Blocks Implementierung ebenfalls bis zu 8 Kernen skaliert, beim Wechsel auf 12 und 16 Kerne aber stagniert und sogar etwas an Durchsatz abnimmt. Dies deckt sich mit den Ergebnissen, die Maier et.al.[13] erzielen konnten, während MxKernel bei 16 Kernen einen um 50% höheren Durchsatz erzielt, als TBB. Siehe hierzu Abbildung 4.10. Dieser Vergleich zeigt ebenfalls, dass die Idee des MxKernel einen Vorteil gegenüber Modellen mit Sperren erzielt.

Kapitel 5

Zusammenfassung

Die vorliegende Arbeit beschäftigte sich mit der Frage, wie ein Hashtabelle für verschiedene Hashingvarianten aufgebaut sein muss, damit diese für das Task-Modell des MxKernel fehlerfrei funktioniert. Dafür wurde das Hashing mit Verkettung und das Hashing mit offener Adressierung für das Task-Modell entwickelt und als Vergleich ebenfalls für das Thread-Modell des MxOS Systems.

5.1 Fazit

Die Umsetzung eines Hashings konnte erfolgreich auf den MxKernel übertragen werden. Hierbei mussten die geänderten Umstände des Kernels betrachtet werden, welcher, um unnötigen Overhead zu vermeiden, gänzlich auf Sperrverfahren verzichten möchte. Da ein Task atomar ausgeführt wird, er also seine Arbeit vollständig durchführen wird, kann sehr einfach darauf geachtet werden, dass die gleichen Ressourcen nicht von unterschiedlichen Tasks betrachtet werden. Dafür wurde die Hashtabelle nach Kernen aufgeteilt und so war es immer nur für einen Task pro Kern möglich diesen Bereich zu betreten. Dies führte dazu, dass Tasks ungleichmäßig in die Warteschlangen der Kerne eingetragen wurden, da erst zur Erstellung des Tasks der Kern, auf dem dieser arbeitet, festgelegt wird. Um dies zu lösen wurde der Anzahl der Tasks pro Kern erhöht. Im Gegensatz dazu musste beim Thread-Modell des MxOS ein Sperrverfahren eingesetzt werden. Auch bei einer passenden Verteilung der einzuhashenden Werte auf die einzelnen Kerne mussten diese Bereiche abgesichert werden, da durch eine Verdrängung ein anderer Thread auf diesen Bereich zugreifen könnte.

Die Evaluation überprüfte den Durchsatz und die Skalierbarkeit der Hashingvarianten auf dem MxKernel und MxOS und stellt einen Vergleich zu einem anderen Betriebssystem, sowie einer anderen Implementierung eines mehrkernfähigen Hashings an. Dabei konnte herausgefunden werden, dass das MxOS im Vergleich zur Linux Distribution Ubuntu gleich gut skaliert, es konnten aber Schwächen in den Semaphoren des MxOS im Ver-

gleich zum Linux Äquivalent ausgemacht werden, was dazu führte, dass die Linux Threads einen höheren Durchsatz erzielten. Beim Task-Modell des MxKernel kann dieser Nachteil gänzlich außer Acht gelassen werden, da hierbei auf Semaphore verzichtet werden konnte. Es zeigte sich bis zu 8 Kernen ein schwächerer Durchsatz für die native Umsetzung der Tasks auf blanker Hardware im Vergleich zur Linux Simulation. Für 12 und 16 Kerne fiel der Durchsatz höher aus. Auch skalierten beide Systeme für 12 und 16 Kerne nicht mehr, was sich möglicherweise auf unbehandelte NUMA Effekte zurückführen lies.

Eine Außerachtlassung der NUMA Effekte bestätigt die Idee des MxKernel. Der höchste Durchsatz auf 8 Kernen wird zwar durch die Threading Building Blocks erreicht, es zeigt sich hierfür aber keine weitere Steigerung beim Hinzunehmen weiterer Kerne.

5.2 Ausblick

Im Rahmen dieser Arbeit konnten nur zwei Hashingvarianten effektiv betrachtet werden und für eine wurde sogar ein Teilaspekt außer Acht gelassen. Daher könnte für zukünftige Implementierungen beispielsweise das Cuckoo-Hashing betrachtet werden. Zudem kann ein Modell erstellt werden, welches Rehashing auf einzelnen Bereichen durchführt, wie in Abschnitt 3.3 vorgestellt oder ein sperr freies Rehashing auf der kompletten Tabelle. Dies könnte im Detail den Durchsatz einer Hashtabelle verbessern.

Interessant dürfte zudem eine tiefgründige Analyse der entstandenen Effekte des MxKernel sowie der Linux Task-Simulation auf mehr als 8 Kernen sein. Sobald für dieses Problem eine Lösung entworfen wurde, kann ein Benchmark möglicherweise auch hier skalieren und damit den Durchsatz, welcher durch Linux Threads entstanden ist, übertreffen. Um diesen Effekt zu überprüfen, könnten die Benchmarks zudem auf einer anderen Prozessorarchitektur durchgeführt werden. Weiterhin könnte interessant sein, wie die Hashingvarianten und darauf basierenden Benchmarks sich für mehr als 16 Kerne verhalten und ob diese möglicherweise immer noch skalieren.

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Baccelli, Emmanuel, Oliver Hahm, Mesut Günes, Matthias Wählisch und Thomas Schmidt: *RIOT OS: Towards an OS for the Internet of Things*. In: *The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, April 2013.

[2] Bryant, Ray und John Hawkes: *Linux scalability for large numa systems*. In: *Linux Symposium*, Seite 76, 2003.

[3] Cloutier, Félix: *RDTSC — Read Time-Stamp Counter*. http://www.felixcloutier.com/x86/RDTSC.html. Zugriff am 28.02.2018.

[4] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd Auflage, 2009.

[5] Drepper, Ulrich: *What Every Programmer Should Know About Memory*, 2007.

[6] Ebbers, Mike, Wayne O'Brien und Bill Ogden: *Introduction to the New Mainframe: Z/OS Basics*. Vervante, 2006.

[7] Fujitsu: *FUJITSU PRIMERGY RX200 S7*. http://www.fujitsu.com/tw/Images/ds-py-rx200-S7.pdf. Zugriff am 01.03.2018.

[8] Hennessy, John L. und David A. Patterson: *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[9] Intel: *Intel® Threading Building Blocks*. `https://www.threadingbuildingblocks.org/`. Zugriff am 28.02.2018.

[10] Kleen, Andi: *Linux multi-core scalability*, 2009.

[11] Knuth, Donald E.: *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[12] Lameter, Christoph: *NUMA (Non-Uniform Memory Access): An Overview.* Queue, 11(7):40:40–40:51, Juli 2013.

[13] Maier, Tobias, Peter Sanders und Roman Dementiev: *Concurrent Hash Tables: Fast and General?(!).* SIGPLAN Not., 51(8):34:1–34:2, Februar 2016.

[14] Manchanda, Nakul und Karan Anand: *Non-Uniform Memory Access (NUMA).* New York University, 2010.

[15] Mühlig, Jan: *MxOS: Ein Betriebssystem mit Threads auf Task-Basis.* Diplomarbeit, Technische Universität Dortmund, 2017.

[16] Spinczyk, Olaf und Jens Teubner: *MxKernel: A Bare-Metal Runtime System for Database Operations on Heterogeneous Many-Core Hardware SPP 2037.* https://www.dfg-spp2037.de/sp968-2/. Zugriff am 23.02.2018.

[17] Vajda, Andrs: *Programming Many-Core Chips.* Springer Publishing Company, Incorporated, 1st Auflage, 2011.