technische universität
dortmund

Bachelorarbeit

Transactional Memory in systemnahen Datenstrukturen

Steffen Nießing
August 2020

Gutachter:
Prof. Dr. Jens Teubner
Jan Mühlig

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

Inhaltsverzeichnis

Kapitel 1

Einleitung

Datenbanken gewinnen in heutigen Systemen in der Informatik eine immer zentralere Rolle. Sie stehen dafür, große Datenmengen sehr effizient speichern und verarbeiten zu können. Damit sie aber auch weiterhin auf dem aktuellen Stand der Technik bleiben, müssen sie sich ständig weiterentwickeln und sich an aktuelle Hardware anpassen.

Gerade die Forschung an Prozessoren ist in den letzten Jahren sehr stark vorangeschritten. Prozessoren werden von den Herstellern mit immer mehr Prozessorkernen ausgestattet, damit gleichzeitig mehrere Aufgaben bearbeitet werden können. Aus diesem Grund ist es heutzutage wichtig, sein System auf Mehrkernprozessoren zu optimieren. Gerade in Servern werden oft Systeme aus mehreren Prozessoren, sogenannte Multiprozessorsysteme, eingesetzt. Damit verfügt ein aktuelles System nicht selten über 32 und mehr Prozessorkerne.

Damit eine Datenbank viele Daten gleichzeitig verarbeiten kann, muss sie Gebrauch von mehreren Prozessorkernen machen. Um aber gleichzeitig mehrere Kerne betreiben zu können, die auch auf gleichem Speicher operieren können, muss der Zugriff auf die vorliegenden Daten unter ihnen synchronisiert werden. Diese Synchronisation sichert zwar die Integrität der Daten, macht das System aber in erster Linie langsamer, weil Konflikte beim Schreiben detektiert und vermieden werden müssen. Um also eine skalierbare und schnelle Anwendung zu erhalten, muss das Synchronisationskonzept der Anwendung sehr effizient gestaltet werden.

Die meisten Anwendungen nutzen zur Synchronisation zwischen verschiedenen Kernen sogenannte Sperren (engl. Latches). Diese können im Prozessor durch verschiedene Programmiertechniken realisiert werden. Sperren können aber durch undurchdachte Programmierung sehr schnell zu Leistungseinbußen führen, weshalb man sie vermeiden möchte.

Das Projekt des `MxKernel` [21] versucht, diesem Problem mit einer anderen Kontrollflussabstraktion aus dem Weg zu gehen. Anstatt klassischer Threads werden sogenannte `Tasks` eingesetzt, die kleine atomare Einheiten darstellen. Diese werden im `MxTasking`-Projekt aber durch eine Abstraktionsebene wiederum auf Thread-Basis realisiert, um nicht

direkt auf einer *bare-metal*-Umgebung arbeiten zu müssen. Somit kann als Unterbau für dieses Projekt ein Linux-System genutzt werden, um die Messungen durchzuführen.

1.1 Ziele der Arbeit

Im `MxTasking` werden bisher atomare CPU-Instruktionen eingesetzt, um den Zugriff auf eine Prozessorwarteschlange zu synchronisieren. Da in eine solche Warteschlange relativ häufig geschrieben wird, wird getestet, ob die Synchronisation durch andere Techniken verbessert werden kann.

Konkret wird das Konzept transaktionalen Speichers eingeführt und evaluiert, in welcher Relation die Leistung einer Implementierung mit transaktionalem Speicher zu der einer Implementierung mit atomaren Instruktionen steht. Die Tasks auf dem `MxTasking` werden dabei im Datenbankkontext betrachtet.

Transaktionaler Speicher blockiert im Gegensatz zu atomaren CPU-Instruktionen die CPU nicht, sondern schreibt den Speicher in einer Transaktion direkt neu und detektiert am Ende einer Transaktion, ob dieses Vorgehen Konflikte verursacht hat. Bei Konflikten müssen dann die zuvor geschriebenen Änderungen rückgängig gemacht werden.

Als Realisierung von transaktionalem Speicher wird die Hardware-Implementierung von Intel® (Intel® TSX-NI) genutzt, da sie bereits in vielen Prozessoren verfügbar ist.

1.2 Aufbau der Arbeit

Im nächsten Kapitel wird zuerst mit einem Grundlagenteil ein grundsätzliches Verständnis der verwendeten Begriffe über Caching, Warteschlangen, transaktionalen Speicher und die `MxKernel`-Plattform geschaffen. In Kapitel 3 werden dann Algorithmen mit unterschiedlichen Synchronisationsverfahren auf einer Warteschlange vorgestellt, die dann in Kapitel 4 evaluiert werden. In Kapitel 5 wird schließlich ein Fazit über die hier vorgestellten Techniken gezogen. Zum Schluss gibt Kapitel 6 einen Ausblick darüber, inwiefern die hier vorgestellten Techniken im Bezug zum `MxKernel` stehen.

Kapitel 2

Grundlagen

In diesem Kapitel werden einleitend die in dieser Arbeit genutzten Begriffe und Techniken erklärt. Zunächst wird dafür Caching in der CPU betrachtet. Weiterhin werden Warteschlangen, Synchronisation von Threads, transaktionaler Speicher und die `MxKernel`-Plattform beleuchtet.

2.1 CPU Caching und Kohärenz

Essentiell für eine schnelle Verarbeitung von Daten ist nicht nur der Prozessor, sondern auch der Speicher. Heutige Prozessoren haben meistens deutlich höhere Taktraten als ihr Arbeitsspeicher[1], darum wird Speicher für die Gesamtleistung eines Systems schnell zum Flaschenhals. Denn beim Arbeitsspeicher wird häufig auf DRAM-Chips gesetzt, da diese im Gegensatz zu SRAM-Chips geringere Energiekosten haben [12]. Mit der Wahl geringerer Kosten wird dafür aber höhere Speicherlatenz in Kauf genommen. Um dem Flaschenhals des Speichers aus dem Weg zu gehen, werden intern in CPUs oftmals zusätzlich SRAM-Speicher, die so genannten Caches, verbaut.

Caches sollen die Speicherlatenzen des Arbeitsspeichers verstecken. Sie sind (im Vergleich zum Arbeitsspeicher) sehr kleine und schnelle Speicher, die Ergebnisse speichern können, bis diese in den Arbeitsspeicher zurückgeschrieben worden sind. Ein Zugriff auf einen direkt angebundenen Cache kostet eine CPU z. B. in einem Prozessor der Baureihe Intel® Pentium® M nur ungefähr 3 CPU-Zyklen, während ein Zugriff auf den Arbeitsspeicher ungefähr 240 Zyklen dauert [12].

Da die Latenz wächst, wenn Speicher größer wird [12], werden möglichst kleine Caches eingesetzt. Um aber nicht direkt wieder auf den Arbeitsspeicher zurückzugreifen, müssen Caches auch hinreichend groß sein. Darum wird häufig eine Hierarchie von mehreren Cache-Leveln aufgebaut. Dabei wird je ein sehr kleiner Cache direkt an einen Prozessorkern

[1] Bei DDR4-RAM ist laut Spezifikation von JEDEC ein Takt bis zu 2400MHz vorgesehen [10], während heutige CPUs schon Basistaktraten bis zu 4GHz und zusätzlich noch Turbo-Boost haben [1].

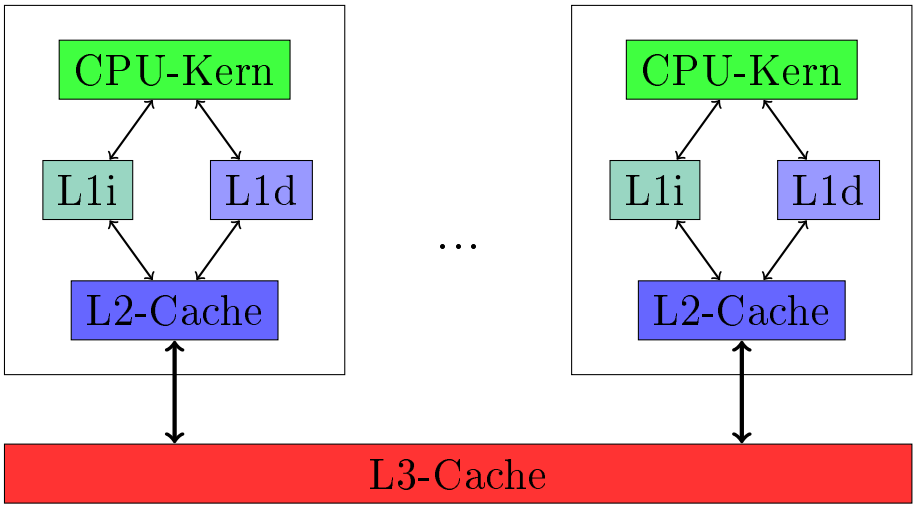

Abbildung 2.1: Beispielhafte Cache-Hierarchie einer CPU mit getrenntem L1d- und L1i-Cache, einem L2-Cache pro Kern und geteiltem L3-Cache

angebunden und jeweils größere Caches werden dann als höhere Level genutzt, um kleinere Caches miteinander zu verbinden.

Intel® z. B. nutzt in seinen Prozessoren großteils einen Level 1 Cache (L1-Cache) pro Prozessorkern, einen L2-Cache pro L1-Cache und einen L3-Cache übergreifend über alle L2-Caches eines Prozessors [13]. Dabei sind die L1-Caches jeweils noch einmal aufgeteilt in L1d (L1-Data-Cache) und L1i (L1-Instruction-Cache). Diese Hierarchie ist beispielhaft in Abbildung 2.1 dargestellt.

Daten in Caches werden in Cache-Lines verwaltet [12]. Eine Cache-Line enthält mehrere Bytes an Daten, die als eine funktionelle Einheit betrachtet werden. Eine Cache-Line ist dabei üblicherweise 64 Bytes groß [12]. Wenn ein CPU-Kern Daten aus dem Arbeitsspeicher anfordert, dann werden diese zunächst in den Cache geladen. Mit dem angeforderten Datum werden aber auch die folgenden Bytes in der Größe einer Cache-Line abgerufen, um die Zugriffe an den Arbeitsspeicher zu reduzieren. Damit werden in Caches immer ganze Cache-Lines geschrieben. Wenn ein Kern z. B. ein `int`-Array aus dem Arbeitsspeicher durchlaufen möchte und in dieser Architektur die Größe eines `int` 4 Byte und die einer Cache-Line 64 Byte ist, dann werden mit dem Abrufen eines einzelnen `int` die folgenden 15 ebenfalls abgerufen.

Eine Cache-Line bekommt zur Verwaltung außerdem noch einige Parameter zugeordnet. Diese sind der Tag, der Index und das Offset. Mit Hilfe dieser Parameter werden Zuordnungen vorgenommen, sodass eine Cache-Line einem Speicherbereich im Arbeitsspeicher zugeordnet werden kann und ein Wert innerhalb eines Caches gefunden werden kann.

Cache Kohärenz Näher betrachtet werden soll als Nächstes besonders der L1-Cache. Dieser ist durch seine direkte Kernanbindung der schnellste Cache, wirft gerade darum aber auch Probleme auf. Denn in einem Mehrkernprozessor mit geteiltem Speicher können alle Prozessorkerne auf die gleichen Daten zugreifen. Wenn mehrere Kerne die gleichen Speicheradressen beschreiben wollen, dann werden die Ergebnisse dieser Schreibvorgänge bis zum Zurückschreiben in den Arbeitsspeicher erst einmal in die verschiedenen L1-Caches geschrieben. Allerdings stehen in den Caches dann möglicherweise verschiedene Ergebnisse und der Wert für die Speicherzelle im Arbeitsspeicher ist unbestimmt.

Um diesem Problem aus dem Weg zu gehen, existiert die Technik der Cache-Kohärenz. Kohärenz ist dafür verantwortlich, dass die Daten in den verschiedenen Caches miteinander vereinbar sind. Das bekannteste Protokoll zur Herstellung von Cache-Kohärenz ist das MESI-Protokoll [12]. Darin werden 4 Status definiert, in denen sich ein Cache-Eintrag befinden kann:

- Modified:
  Wenn ein CPU-Kern eine Cache-Line geschrieben hat, so ist diese *modified*, bis sie in den Arbeitsspeicher zurückgeschrieben worden ist. Damit ist sie auch die einzige gültige Kopie dieses Speichers.

- Exclusive:
  Eine Cache-Line ist *exclusive*, wenn sie nur von einem Kern lesend angefordert wurde.

- Shared:
  Shared wird als Status gesetzt, wenn die Cache-Line von mehreren Caches geteilt wird.

- Invalid:
  Wenn ein Kern eine geteilte Cache-Line beschreibt, so werden alle anderen Cache-Lines, die auf den gleichen Speicher zeigen, auf *invalid* gesetzt. *Invalid* ist darum außerdem der Ausgangszustand einer nicht genutzten oder noch nicht geschriebenen Cache-Line.

Wenn ein CPU-Kern eine Cache-Line beschreibt, kann und muss diese also nur in den Arbeitsspeicher zurückgeschrieben werden, wenn sie den Status *modified* besitzt. Wenn ein anderer CPU-Kern die gleiche Adresse geschrieben hätte, wäre sie sonst als *invalid* markiert worden.

2.2 Programmierparadigmen von Warteschlangen

Um maximale Leistung in einer Anwendung zu erzielen, ist es also wichtig, Daten so zu organisieren, dass sie ein gutes Caching-Verhalten haben, damit die Latenz des Speichers

versteckt wird. Gutes Caching-Verhalten wird vor allem von zusammenhängenden Datenstrukturen wie Arrays erzielt. Allerdings sind Arrays nicht für alle Anwendungsfälle gut geeignet, da sie die Semantik vieler Probleme nicht widerspiegeln und außerdem keine Flexibilität in ihrer Größe bieten. Darum gibt es viele andere Datenstrukturen, die die Semantik eines Problems besser ausdrücken können, dafür aber möglicherweise schlechteres Caching-Verhalten besitzen.

Eine häufig verwendete dieser Datenstrukturen ist eine Warteschlange (Queue)[2]. Warteschlangen werden in vielen Bereichen, wie z. B. im Scheduling, einsetzt [21]. Sie zeichnen sich dadurch aus, dass alle Elemente (engl. Items) nach dem Zeitpunkt ihres Einfügens abgearbeitet werden (FIFO - first in, first out). Dabei gibt es einen Erzeuger, der Elemente während des Programms in die Warteschlange einfügt (Producer), und einen Verbraucher, der die eingefügten Elemente verarbeitet (Consumer).

Allerdings ist dies nur die Basisversion einer Warteschlange. Sie ist nur wenig für Mehrkernsysteme geeignet, da sie in diesem Fall nur von zwei CPU-Kernen gleichzeitig Gebrauch machen kann. Mit einer Erweiterung des Producer-Consumer-Paradigmas kann diese Datenstruktur aber mit mehreren Kernen genutzt werden (vgl. dazu auch [25]). Dabei sind verschiedene Fälle zu betrachten:

SPSC: Single producer, single consumer
Diese Variante ist die oben beschriebene Fassung mit einem Erzeuger und einem Verbraucher.

SPMC: Single producer, multiple consumer
Bei dieser Variante gibt es einen Erzeuger, der Items produziert, und mehrere Verbraucher, die diese Items verarbeiten.

MPSC: Multiple producer, single consumer
In diesem Fall verarbeitet ein Verbraucher Items von mehreren Erzeugern.

MPMC: Multiple producer, multiple consumer
In diesem Fall produzieren mehrere Erzeuger Items für mehrere Verbraucher.

Unter dem Aspekt der Synchronisation von verschiedenen Threads sind von diesen vier Fällen nur die letzten drei interessant, da im ersten Fall keine Synchronisation nötig ist. Aber sobald bei einer Aktionsseite (Erzeuger oder Verbraucher) mehr als ein Thread vorhanden ist, muss sichergestellt werden, dass die Integrität der gelesenen und geschriebenen Daten erhalten bleibt.

[2] Warteschlangen müssen trotzdem nicht zwangsläufig langsam sein. Zum Beispiel in der Haswell Architektur werden sie sogar Prozessor-intern genutzt [20], wo hohe Leistungsansprüche an sie gestellt werden.

2.2.1 Implementierung einer MPSC-Queue

Im Folgenden wird vor allem die MPSC-Queue interessant werden, da diese derzeit im `MxKernel` [21] eingesetzt wird. Bei diesem Paradigma muss die Verbraucherseite nicht synchronisiert werden, wohl aber die Erzeugerseite. Konkret seien die Methoden zum Zugriff auf die Queue als `pop_front` für den Zugriff auf das erste Element („head“) der Queue sowie `push_back` für das Anhängen an das Ende der Queue („tail“) benannt.

`pop_front` kann also direkt auf den *head* der Queue zugreifen, bzw., wenn dieser leer ist, einen `null`-Wert zurückliefern. Nach dem Algorithmus von Vyukov [24] wird außerdem, falls der *head* einen Nachfolger hat, das nächste verlinkte Element als neuer *head* der Queue gesetzt. Die Bezeichnungen von Vyukov von *head* und *tail* weichen allerdings von den hier genutzten Bezeichnungen invertiert ab.

Interessanter als die `pop_front`-Methode ist aber die `push_back`-Methode, da bei dieser eine Synchronisation der Threads nötig ist. Im Algorithmus von Vyukov wird die Hauptarbeit dieser Methode von einem Makro `XCHG` verrichtet. Die Funktion hiervon wird im nächsten Abschnitt erläutert.

2.2.2 Synchronisation von gleichzeitig genutzten Datenstrukturen

Bei den Paradigmen im Kapitel 2.2 muss sichergestellt werden, dass in den letzten drei Fällen der Zugriff auf die Speicheradressen für jeden Thread einer Implementierung eines solches Paradigmas exklusiv ist. Denn ansonsten könnten mehrere Threads gleichzeitig verschiedene Daten in den gleichen Speicher schreiben und diesen damit ungültig werden lassen. Da aber zu jedem Zeitpunkt gesichert sein soll, dass die Daten valide sind, ist dieser Fall auszuschließen.

Diesem Problem kann man mit Sperren (engl. Latches) aus dem Weg gehen [18]. Jeder Thread, der eine Speicheradresse beschreiben möchte, fordert dazu zuerst eine Sperre an und schreibt erst dann, wenn er diese alleine besitzt. Alle anderen Threads warten in dieser Zeit, bis die Sperre wieder freigegeben ist. Wenn sie eine freie Sperre feststellen, versuchen sie selbst, diese anzufordern, um erst dann selbst zu schreiben. Eine Sperre kann dabei immer nur von einem Thread besessen werden, da nur so ein exklusives Verhalten abgebildet werden kann. Damit ist auch sichergestellt, dass kein Thread „verhungert“, indem er darauf wartet, dass die anderen Threads ihm die Sperre alleinig überlassen.

Dieses Vorgehen bedeutet für die Applikation aber, dass der Leistungsgewinn, der von mehreren Kernen erwartet wird, direkt wieder verloren geht oder sogar weniger Leistung erzielt wird. Denn, da in jedem Fall nur ein Thread schreiben kann, ist dieses Verfahren nicht effizienter, als wenn direkt nur ein Thread allein in die Warteschlange schreiben würde.

Je nach Umsetzung kann ein Producer-Consumer-System mit mehr als zwei Kernen trotzdem effizienter sein als seine Basisversion. Denn, während ein Thread darauf wartet, in

den gemeinsamen Speicher zu schreiben, können trotzdem schon wieder neue Items erzeugt werden. Es ist also wichtig, den Zugriff auf den Codeabschnitt, der den gemeinsamen Speicherbereich beschreibt (kritischer Abschnitt), so kurz wie möglich zu halten, damit die Kosten des Wartens gering gehalten werden können und nicht viele Threads blockieren müssen. Also sollte entweder eine Sperre genutzt werden, die schnell wieder freigegeben wird, oder eine andere Möglichkeit zur Synchronisation genutzt werden, die die CPU nicht lange oder gar nicht blockieren lässt.

Für den Synchronisationsansatz mit Sperren sind atomar auszuführende Instruktionen notwendig, um Konflikte zwischen konkurrierenden Threads zu beheben. In Intel® Prozessoren gibt es z. B. in der Assembly das Präfix `lock`, um anderen Prozessorkernen zu signalisieren, dass ein bestimmter Speicherbereich gerade geschrieben wird und der Zugriff auf ihn exklusiv sein muss [5]. Dieses Präfix wird unter anderem von der Instruktion `xchg` (exchange) verwendet, die es immer implizit setzt. Ein `xchg` ist also gleichwertig mit einem `lock xchg`. Mit `xchg` werden zwei Werte atomar durch den jeweils anderen ersetzt [7]. Durch das implizite `lock`-Präfix kann währenddessen kein anderer Thread auf diese Speicheradresse zugreifen. Alle anderen Zugriffe werden durch die Hardwareimplementierung so lange blockiert, bis die `xchg`-Instruktion beendet ist.

Mit dieser Instruktion kann damit eine Sperre realisiert werden. Wenn ein Thread den Wert 1 (Semantik: Sperre belegt) in eine Speicheradresse schreibt und den Wert 0 (Semantik: Sperre frei) in sein Register zurückgeliefert bekommt, dann hat er gerade diese Sperre bekommen. Da diese Instruktion atomar ausgeführt wird, kann immer nur ein Thread eine Sperre besitzen. Denn durch das `lock`-Präfix müssen alle anderen Kerne mit ihrem Schreibvorgang warten und bekommen dadurch den Wert 1 aus der Speicheradresse zurück. Damit ist für sie die Sperre belegt und der Zustand der Sperre ändert sich nicht, da von ihnen weiterhin nur eine 1 in diesen Speicher geschrieben wird. Somit wird bei allen Threads dieses Schreiben als Schleife implementiert, bis sie irgendwann den Wert 0 erhalten, was ihnen signalisiert, dass sie die Sperre gesperrt haben. Diese Technik wird als Spinlock bezeichnet, da die Threads alle in einer Schleife warten (spinnen), bis sie die Sperre erhalten haben.

Algorithmus 2.1 beschreibt eine einfache Sperre. Allerdings wartet der Prozessor hierbei aktiv darauf, dass die Sperre frei wird, und erzeugt währenddessen sehr viele Instruktionen. Außerdem ist hierbei die Gefahr groß, die Freigabe der Sperre zu vergessen und damit alle anderen Threads verhungern zu lassen.

Gegenseitiger Ausschluss verschiedener Threads in einer Warteschlange kann für den Prozessor günstiger realisiert werden. Wenn z. B. das Einfügen am Ende einer Queue synchronisiert werden soll, so kann dies folgendermaßen realisiert werden:

Eine Warteschlange verwaltet einen Zeiger auf ihr letztes Element (*tail*). Mit einer `xchg` Instruktion wird dieser Zeiger durch einen Zeiger auf das einzufügende Element ersetzt und danach der next-Zeiger des vorher letzten Elementes auf das aktuell letzte Element

```
spinlock:
        mov eax, 1
        xchg eax, [lock_variable]
        test eax, eax
        jnz spinlock
success:
        ...
unlock_spinlock:
        mov eax, 0
        mov [lock_variable], eax
```

Algorithmus 2.1: Implementierung eines einfachen Spinlocks in Intel® Assembly

```
push_back(item):
        exchange queue->tail and item
                //now item is the previous tail
        item->next := queue->tail
```

Algorithmus 2.2: Anhängen eines Elementes an das Ende einer Queue mit `xchg`

umgesetzt. Dies wird in Algorithmus 2.2 aufgezeigt. Damit ergibt sich für die `push_back`-Methode einer MPSC-Queue die Implementierung aus dem Algorithmus von Vyukov [24]. Für ein Sperren-basiertes System ist dies die günstigste Implementierung. Zur Synchronisation muss ohnehin gewartet werden und über das `xchg` wird die kürzestmögliche Wartezeit erzielt, da die CPU jeweils nur für diese eine Instruktion blockiert wird.

Ein Spinlock erzeugt in diesem Fall deutlich mehr Instruktionen und lässt den Prozessor außerdem durch die `xchg`-Instruktionen in gleicher Weise warten. Darum ist hier das einfache `xchg` ohne Spinlock zu bevorzugen.

2.3 Transactional Memory

Neben Latch-basierter Synchronisation und der Verwendung atomarer Instruktionen, gibt es noch andere Techniken zur Synchronisation von gleichzeitig genutzten Datenstrukturen. In diesem Abschnitt wird die Technik des transaktionalen Speichers eingeführt. Dazu wird zunächst eine andere Synchronisationsprimitive eingeführt, die dann auf transaktionalen Speicher erweitert werden kann.

2.3.1 Load-linked store-conditional Architektur (LL/SC)

Eine weitere Primitive zur Synchronisation von verschiedenen Schreibvorgängen ist die *load-linked store-conditional* Architektur [16]. Hierbei können verschiedene Threads jeweils die gleiche Adresse lesen, schreiben kann allerdings nur der erste Thread. Der Grundgedanke ist folgender: Beim Lesen einer Adresse wird festgehalten, dass dieses Lesen „*linked*“ war. Wenn dann ein Thread schreiben möchte, wird zuerst geprüft, ob der aktuelle Wert der Speicheradresse noch der ursprünglich gelesene Wert ist. Wenn ja, so ist dieser Thread der erste Schreiber und der Schreibvorgang wird ausgeführt. Wenn sich der Wert aber zwischenzeitlich geändert hat, so wird der Schreibvorgang nicht ausgeführt, da damit ein inkonsistenter Zustand auftreten würde.

Auch wenn diese Primitive in einer Intel® Architektur nicht direkt realisiert wird, so gibt es trotzdem einen Weg, um ein ähnliches Verhalten zu erzeugen. Die Intel® Assembly stellt hierfür den Befehl `cmpxchg` (compare and exchange) bereit. Dieser vergleicht zuerst den Wert des Registers `eax` mit dem Zieloperanden des `cmpxchg` und schreibt nur bei Gleichheit den Wert des Quelloperanden in den Zieloperanden hinein. Ansonsten wird der aktuelle Wert des Zieloperanden in `eax` zurückgeschrieben [4]. Durch diesen Befehl wird der Gedanke von *store-conditional* deutlich. Außerdem impliziert dieses Verhalten eine weitere Eigenschaft von LL/SC: wenn ein Thread einen Schreibvorgang abbrechen muss, so muss der vorher gelesene Wert zuerst neu eingelesen werden.

2.3.2 Transaktionen und Erweiterung von LL/SC

Auf dem Weg von LL/SC zu transaktionalem Speicher soll zunächst der Begriff einer Transaktion definiert werden, da dieser zentral für das weitere Vorgehen ist. Dabei wird die Definition nach Herlihy und Moss [15] benutzt:

2.3.1 Definition (Transaktion). Eine Transaktion ist eine endliche Folge von Befehlen eines einzelnen Prozesses mit den folgenden Eigenschaften:

- *Serialisierbarkeit*: Eine Transaktion ist serialisierbar, wenn sie nicht mit einer anderen Transaktion „verzahnt“ [15] ist, also alle Befehle einer Transaktion vor allen Befehlen einer anderen Transaktion ausgeführt werden. Erfolgreiche Transaktionen erscheinen immer in der gleichen Reihenfolge.
- *Atomarität*: Eine Transaktion wird entweder ganz ausgeführt („committed“) oder vollständig abgebrochen („aborted“). Bei einem Commit werden alle Änderungen der Transaktion gleichzeitig sichtbar und keine Änderung kann vorher von anderen Transaktionen gelesen werden.

Nach dieser Definition wird die Einführung von LL/SC-Primitiven deutlich: Jede von einer Transaktion gelesene Adresse muss *linked* gelesen werden und darf nur geschrieben werden,

wenn die ganze Transaktion committed. Ein Abort einer Transaktion muss immer dann stattfinden, wenn das *store-conditional* fehlschlägt.

Dies beschreibt allerdings nur die Validierung eines Schreibvorganges. Bei einem Lesevorgang muss gesichert werden, dass keine andere Transaktion diese Adresse in der Zwischenzeit geschrieben hat. Dies kann allerdings dadurch umgesetzt werden, dass ein Lesevorgang zu einem Schreibvorgang umgebaut wird. Wenn das Lesen als *linked* implementiert wird, so sollte das erneute Schreiben des gleichen Wertes keinen Unterschied bedeuten. Wenn beim Schreiben des vorher gelesenen Wertes in den Speicher dann jedoch ein Konflikt auftritt, so ist der Wert von einem anderen Kern geschrieben worden und der vorher gelesene Wert ist nicht mehr gültig.

Nach dieser Interpretation ist eine Transaktion eine Erweiterung der LL/SC-Primitive auf mehrere Befehle. Transaktionen zeichnen sich vor allem dadurch aus, dass Synchronisation durch sie besonders einfach wird, da Synchronisation auch der Kerngedanke hinter LL/SC ist. Nähere Einblicke in das Thema der Synchronisation mit Transaktionen gibt Abschnitt 2.3.5.

Dies ist allerdings nur ein abstrakter Ansatz, wie eine Implementierung aussehen könnte. Durch die Realisierung der Konsistenz eines Lesevorgangs als Schreibvorgang würden unnötige Instruktionen ausgeführt. Tatsächlich gibt es schon seit längerem einen anderen Ansatz zu transaktionalem Speicher, der im Folgenden erklärt wird.

Ein Ansatz für transaktionalen Speicher Einen Ansatz für transaktionalen Speicher haben Maurice Herlihy und J. Eliot B. Moss schon 1993 vorgeschlagen [15]. Bei ihnen gibt es als Primitiven gegenüber LL/SC die Operationen:

- *Load-transactional* (LT): Hiermit wird der Wert aus einer Speicheradresse in ein Register gelesen.

- *Load-transactional-exclusive* (LTX): Hier wird dasselbe wie bei `LT` ausgeführt, allerdings mit dem Hinweis, dass dieses Datum sehr wahrscheinlich geschrieben wird.

- *Store-transactional* (ST): Hiermit wird der Wert eines Registers wieder in den Speicher geschrieben. Allerdings ist dieser Wert für andere Prozesse noch nicht sichtbar, bis die Transaktion committed.

Diese Implementierung ist sehr ähnlich zum oben ausgeführten Gedanken. Eine Transaktion ist also im Grunde genommen nur eine Sammlung von mehreren LL- oder LL/SC-Befehlen, die entweder einen gemeinsamen *store* bei einem Commit der Transaktion ausführen, oder die allesamt invalidiert und damit nicht persistiert werden.

2.3.3 Transactional Memory in Hardware

Bei Intel® s Implementierung von Hardware Transactional Memory (im Folgenden kurz als HTM, oder RTM: „Restricted Transactional Memory“, bezeichnet) wird genau der Ansatz von Kapitel 2.3.2 verfolgt. Wenn eine Adresse genutzt wird, wird festgehalten, dass der Zugriff innerhalb einer Transaktion erfolgt ist. Bei einem Lese- oder Schreibvorgang wird dann festgestellt, ob die Adresse von einem anderen Kern geschrieben wurde, und in diesem Fall werden alle transaktional gesetzten Änderungen verworfen. Die Grenzen einer Transaktion werden dabei durch zwei Instruktionen markiert: `xbegin` beginnt eine Transaktion und `xend` beendet sie [22]. Gleichzeitig gibt es mit den Intel® Transactional Synchronisation Extensions (kurz: TSX-NI), mit denen der transaktionale Speicher eingeführt wurde, noch zwei weitere Befehle für transaktionalen Speicher. Mit dem Befehl `xabort` kann man eine Transaktion vorzeitig abbrechen und mit `xtest` kann man im Code prüfen, ob der Prozessor sich gerade in transaktionaler Ausführung befindet.

Die Realisierung von transaktionalem Speicher in Intel® s Implementierung erfolgt über den L1-Cache des Prozessors. Daher rührt auch der Name RTM, da die Transaktionen auf die Gegebenheiten des L1-Caches beschränkt sind. Konkret wird eine Transaktion damit auf eine bestimmte Größe von Lese- und Schreibvorgängen beschränkt, da alle transaktionalen Änderungen bis zum Commit im Cache erhalten bleiben müssen.

Implementierung einer Transaktion in einer Intel® Architektur Im Code sieht eine RTM-Transaktion wie in Algorithmus 2.3 aus. Die einzigen notwendigen Befehle für eine Transaktion sind hierbei nur `xbegin` und `xend`. Der andere Code ist nur zur Verdeutlichung der neu eingeführten Befehle angegeben.

Diese Transaktion hat allerdings noch keinen Inhalt und wird darum, bis auf außerordentliche Abbrüche, immer committen. Falls allerdings Code, der Konflikte verursachen kann, innerhalb der Transaktion ausgeführt wird, so müssen diese Konflikte detektiert werden.

Konfliktdetektion Für die Detektion eines Konfliktes innerhalb einer transaktionalen Ausführung kann sehr gut das Cache-Kohärenz-Protokoll genutzt werden (vgl. dazu Kapitel 2.1). Dazu wird in jeder Cache-Line festgehalten, ob sie in einem transaktionalen Kontext gelesen oder geschrieben wurde. Wenn am Ende einer Transaktion alle in einer Transaktion verwendeten Cache-Lines einen Status ungleich *invalid* haben, dann gab es keine Schreibkonflikte mit anderen CPU-Kernen. Auch die Lesevorgänge sind damit valide. Wäre nämlich ein Wert gelesen worden, der nachher in einer anderen Transaktion geschrieben wurde, dann wäre dieser Wert im lokalen Cache invalidiert worden.

Zusätzlich muss aber noch berücksichtigt werden, was passiert, wenn eine Cache-Line von einer anderen überschrieben wird. Wenn diese Cache-Line transaktional genutzt wurde, so muss sie im Cache bleiben, damit beim `xend` Konflikte mit dieser Line detektiert

```
xtest
    ; this evaluates to false,
    ; because the transaction has not started yet

xbegin fallback_handler
    ; begin the transaction here

xtest
    ; this evaluates to true,
    ; because here we are inside a transaction

; call xabort here to quickly abort the transaction

xend
    ; here happens the conflict detection and the possible commit

xtest
    ; here this evaluates to false,
    ; because here the transaction has already ended
```

Algorithmus 2.3: Eine Transaktion in Intel® Assembly

werden können. Die Transaktion muss in diesem Fall also abgebrochen werden. Dadurch tritt gleichzeitig auch eine Beschränkung der HTM-Implementierung von Intel® auf: eine Transaktion darf nicht zu viele Adressen nutzen, da sie sonst auch ohne Konflikte in jedem Fall abgebrochen wird.

Nach Yoo u. a. [26] kann jedoch eine transaktional gelesene Cache-Line noch in eine andere Struktur verschoben werden, wo sie weiter nachverfolgt werden kann, um die Transaktion nicht direkt unnötigerweise abzubrechen. Damit kann eine Transaktion also mehr Cache-Lines lesen, als in ihren L1-Cache passen, aber nicht mehr Cache-Lines schreiben.

2.3.4 Abbrüche von Transaktionen und Fallback-Logik

Wie im letzten Abschnitt schon erwähnt wurde, kann es mehrere Gründe geben, dass eine Transaktion abgebrochen wird. Der einfachste Abbruchgrund ist, dass eine Adresse gleichzeitig von mehreren Kernen beschrieben wird. Aber auch zu viele Schreibvorgänge eines einzelnen Kerns brechen eine Transaktion ab. Intel® listet tatsächlich noch mehr Abbruchgründe auf, z. B. müssen bestimmte Grenzen bei ineinander geschachtelten Transaktionen eingehalten werden und außerdem verursachen einige Befehle, wie z. B. `pause`, innerhalb einer Transaktion immer Abbrüche [23].

Wenn also, aus welchem Grund auch immer, eine Transaktion abgebrochen wird, so muss jeder Speichervorgang innerhalb der Transaktion rückgängig gemacht werden („Roll-

back“). Dies bedeutet, dass, für den aktuellen Zustand des Programms betrachtet, kein einziger Befehl zwischen `xbegin` und `xend` ausgeführt wurde. Dieses Rollback wird komplett durch die Hardware übernommen und muss nicht im Code umgesetzt werden.

Dieses Vorgehen ist zwar insofern richtig, dass das System immer in einem konsistenten Zustand gehalten wird, sichert aber keinen Fortschritt im Programm, denn bei einem Abbruch hat die Transaktion keinen Effekt auf den Zustand des Programms. Damit das Programm aber einen Fortschritt erreichen kann, muss spezifiziert werden, was passieren soll, wenn eine Transaktion abgebrochen wurde [18]. Dafür muss ein sogenannter Fallback-Handler spezifiziert werden (vgl. dazu auch den Artikel von Dementiev [11]). Es handelt sich dabei um einen Codeblock, der immer dann ausgeführt wird, wenn eine Transaktion abgebrochen wurde. Um diesen Handler richtig zu verlinken und zu registrieren, nimmt die `xbegin`-Instruktion ein Argument an, das ein Label spezifiziert, wo ebendieser Fallback-Code abgelegt wird. In Algorithmus 2.3 wird dies schon am Ausdruck `xbegin fallback_handler` sichtbar.

Der Fallback-Handler soll dann so programmiert sein, dass ein Fortschritt für das Programm erzielt wird, den eigentlich auch die Transaktion erreichen soll. Dazu muss aber im schlimmsten Fall wieder auf eine Sperre zurückgegriffen werden, die ja genau durch die Transaktion vermieden werden sollte. Sperren funktionieren, richtig implementiert, zur Synchronisation immer, müssen aber möglicherweise den Prozessor warten lassen.

Ein anderer Ansatz für den Fallback-Handler ist darum, die Transaktion erst mehrere Male neu zu starten. Damit kann die Sperre in vielen Fällen umgangen werden und muss nur dann genutzt werden, wenn alle Transaktionen fehlschlagen. Oftmals wird auch zwischen den Neustarts der Transaktionen eine Wartezeit eingelegt, die immer noch effizienter als das Anfordern der Sperre sein kann.

2.3.5 Skalierbare Synchronisation durch Transactional Memory

HTM bietet für die Programmierung von zu synchronisierendem Code einen großen Vorteil, denn die Hardware kümmert sich um den exklusiven Schreibzugriff. Die eigentliche Kunst besteht dann nur noch darin, den Fallback-Handler richtig zu implementieren. Wenn die Applikation allerdings nicht zeitkritisch ist, so kann einfach oft genug die Transaktion wiederholt werden, bis sie irgendwann erfolgreich geschrieben werden kann. Für diesen Ansatz ist eine zufällige Wartezeit zwischen Transaktionen nützlich, damit diese nicht immer an den gleichen Stellen Konflikte erzeugen. Der einfachste Fallback-Handler ist damit in Algorithmus 2.4 angegeben. Dabei abstrahieren die Funktionen `_xbegin()` und `_xend()` die korrespondierenden Assembly-Instruktionen. Der Fallback-Handler wird automatisch detektiert und auf die Adresse des `else`-Blocks gesetzt. Damit wird dieser sowohl bei einem Konflikt als auch bei nicht erfolgreichem Starten einer Transaktion ausgeführt. Das Makro `_XBEGIN_STARTED` abstrahiert einen Statuscode, den die `xbegin`-Instruktion zurückliefert.

```
void transactional_code() {
    transaction_start:
    if (_xbegin() == _XBEGIN_STARTED) {
        //do transactional code here
        _xend();
    } else {
        std::this_thread::sleep_for(
                std::chrono::milliseconds{rand() % 100});
        goto transaction_start;
    }
}
```

Algorithmus 2.4: Einfachster transaktionaler Fallback-Handler

Dieser Statuscode gibt an, ob die Transaktion erfolgreich gestartet wurde oder ob ein Fehler aufgetreten ist. Im Fehlerfall wird auch ein Grund für den Abbruch angegeben.

Bei diesem Code ist der Fortschritt des Programms zwar nicht gesichert, aber wenn die Transaktion klein genug ist, um nicht von sich aus abgebrochen zu werden, wird dieser Code mit großer Wahrscheinlichkeit nach hinreichend großer Zeit erfolgreich sein.

Effizient ist dieser Handler aber nicht. Für eine erfolgreiche Implementierung in einem zeitkritischen Programm wird in jedem Fall eine Sperre benötigt. Diese kann dann zusätzlich noch genutzt werden, um den Code effizienter zu gestalten. Mit näheren Details zur Implementierung beschäftigen sich die Kapitel 3.2 und 3.3.

2.4 `MxKernel` und `MxTasking`

Im Folgenden wird die Plattform vorgestellt, auf der die Implementierung vorgenommen wird. Sie verwendet einige der vorher vorgestellten Konzepte und soll zur Nutzung von transaktionalem Speicher erweitert werden.

`MxKernel` ist ein neuer Ansatz für das Design von Betriebssystemen zur Verwaltung von heterogener Hardware [21]. Herkömmliche Betriebssysteme sind für sämtliche Arten von Programmen auf CPUs ausgelegt und verfolgen deshalb ein Task-Modell in Form von Threads und Prozessen.

Threads sind allerdings durch ihre CPU-optimierte Struktur nicht unbedingt für jeden Anwendungsfall eine effiziente Struktur. Durch moderne Hardware wird für viele Anwendungsfälle nicht nur Gebrauch von CPUs zur Ausführung von Berechnungen gemacht, sondern es werden auch GPUs oder FPGAs genutzt, um gerade für Datenbanken gut skalieren zu können. Da diese von der Programmierung deutlich anders zu verwalten sind als

CPUs, muss mehr Aufwand in die Implementierung gesetzt werden, um mehrere Techniken parallel nutzen zu können.

Im `MxKernel` wird darum vom normalen Thread-Modell abstrahiert und stattdessen werden sogenannte `Tasks` eingesetzt [21]. Diese sind kleine abgeschlossene Befehlseinheiten, für die garantiert wird, dass sie atomar ausgeführt werden. Dies impliziert direkt eine einfache Form der Synchronisation, da sie für `Tasks` immer implizit vom Betriebssystem übernommen wird.

Im Bezug auf Datenbanken haben `Tasks` auch eine stärkere Semantik als Threads, da `Tasks` an den Charakter einer Datenbanktransaktion erinnern. Datenbanksysteme arbeiten nach dem *ACID*-Prinzip [14], sichern also Atomarität, Konsistenz, Isolation und Haltbarkeit. Da Haltbarkeit eine datenbankspezifische Eigenschaft ist, wird sie hier nicht weiter betrachtet. Die anderen drei Eigenschaften werden aber vom `Task`-System des `MxKernel` gesichert, ebenso wie von transaktionalem Speicher.

Für Datenbankanwendungen auf dem `MxKernel` müssen Datenbanktransaktionen auf `Tasks` abgebildet werden, um sie dann ausführen zu können. Durch die Ähnlichkeit der Modelle ist die Übersetzung vergleichsweise einfach.

Wenn die Datenbanktransaktionen, und damit auch die entsprechenden `Tasks`, klein genug sind, drängt sich der Gedanke auf, sie mit HTM auszuführen, da dieses Konzept direkt auch die geforderten Eigenschaften sichern würde. Eine direkte Ausführung mit HTM wirft jedoch wieder Probleme auf, z. B. beim Fallback-Pfad, denn eine Datenbanktransaktion muss immer ausgeführt werden, wenn sie im Code committed wird. Damit wird sie zu ihrem eigenen Fallback-Pfad und es muss der ineffiziente Ansatz aus Abschnitt 2.3.5 mit etlichen Versuchen genutzt werden. Als Alternative bliebe nur wieder ein Latch-basierter Fallback-Handler, wobei man dann wiederum auf HTM verzichten könnte. Darum ist fraglich, welche Leistung HTM für diesen Anwendungsfall bieten kann. Allerdings kann HTM trotzdem sehr nützlich sein, wenn der Scheduler für die Tasks implementiert wird, da hierbei viel Synchronisationslast erzeugt wird und Latch-basierte Methoden damit Ressourcen blockieren.

`MxTasking` Im `MxTasking` kann HTM darum sehr wohl Einsatz finden. `MxTasking` ist eine Abstraktion des `MxKernel` auf einem Linux-System. Es werden also keine `Tasks` direkt eingesetzt, sondern immer noch herkömmliche Threads. Jeder ausführende Kern bekommt einen Thread, der dann wiederum eine Abstraktion von Tasks abarbeitet. Diese Tasks sind dann wieder abgeschlossene Ausführungseinheiten, die den *bare-metal* `Tasks` des `MxKernel` [21] nahe kommen. In der Implementierung gibt es eine ausgezeichnete Klasse `TaskInterface`, die die Basis für einen Task bereitstellt. Dieser kann damit erzeugt, gelöscht, gescheduled und ausgeführt werden.

Scheduling im `MxTasking` Im `MxTasking` bekommt jeder Thread seine eigene Queue für abzuarbeitende Tasks und ist damit alleiniger Verbraucher dieser Queue. Allerdings können Tasks an verschiedenen Stellen erzeugt werden und damit auf andere Kerne verschoben werden. Damit kommen also MPSC-Queues (vgl. Kapitel 2.2) an dieser Stelle zum Einsatz. Da diese Queues jeweils nur einen Consumer besitzen, braucht die `pop_front`-Methode dieser Queues nicht synchronisiert zu werden. Für die Implementierung ihrer `push_back`-Methoden wird dazu der Algorithmus 2.2 verwendet, damit beim Schreiben keine Inkonsistenzen entstehen. Die konkrete Implementierung dieses Algorithmus wird weiter in Kapitel 3.1 ausgeführt.

Diese MPSC-Queues können nun mit HTM aufgewertet werden. Nach Kapitel 2.3.5 kann HTM sehr gut dazu eingesetzt werden, parallelen Code zu synchronisieren. Dies ist vor allem für die `push_back`-Methoden wichtig. Kapitel 3.2 liefert eine Implementierung einer Queue, die durch HTM synchronisiert wird.

Kapitel 3

Realisierung einer Scheduling-Warteschlange

In diesem Kapitel werden nun die Implementierungen der Warteschlange mit verschiedenen Synchronisationstechniken vorgestellt.

3.1 Warteschlange mit atomaren Instruktionen

In Kapitel 2.4 wurde schon erwähnt, dass das `MxTasking` MPSC-Queues für das Scheduling nutzt, deren `push_back`-Methoden nach dem Algorithmus 2.2 implementiert werden. Die Implementierung der `push_back`-Methoden sieht damit konkret folgendermaßen aus:

```
void push_back(QueueItem *item) {
    item->next(nullptr);
    auto *previous =
            __atomic_exchange_n(&tail,item,__ATOMIC_SEQ_CST);
    previous->next(item);
}
```

Algorithmus 3.1: Implementierung der `push_back`-Methode im `MxTasking`

Hierbei generiert das Makro `__atomic_exchange_n` im Assembly-Code die `xchg`-Instruktion. `__ATOMIC_SEQ_CST` steht für das Memory Ordering und bedeutet, dass diese Befehle alle sequentiell ausgeführt werden müssen. Das Makro `__atomic_exchange_n` tauscht allerdings nicht die beiden Operanden aus, sondern schreibt den Wert des zweiten Operanden in den ersten und gibt den vorherigen Wert des ersten Operanden zurück. Wegen der generierten `xchg`-Instruktion wird dieses Makro atomar ausgeführt.

In Algorithmus 3.1 wird also eine Queue mit atomaren Instruktionen realisiert. Nach Kapitel 2.2.2 wird implizit mit der `xchg`-Instruktion ein `lock`-Präfix gesetzt. Also muss

der schreibende Prozessorkern warten, bis kein anderer Kern mehr die gleiche Adresse schreiben möchte und kann währenddessen keine weitere Arbeit verrichten.

Um das blockieren der CPU zu umgehen, wird im nächsten Abschnitt eine Warteschlange mit HTM-Unterstützung vorgestellt, die dieses Problem zu einem gewissen Grad vermeiden kann.

3.2 Warteschlange mit HTM

Der HTM-Ansatz geht anders an die Synchronisation der Threads heran: Bei diesem Ansatz wird versucht, das Ende der Queue einfach neu zu schreiben, ohne vorher auf konkurrierende Threads zu achten. Danach wird detektiert, ob es einen Schreibkonflikt gab. Wenn dies der Fall ist, wird ein Rollback der Transaktion ausgeführt und die zuvor geschriebenen Änderungen werden rückgängig gemacht. Damit sieht der Pseudocode für diesen Ansatz folgendermaßen aus:

```
push_back(item):
    begin transaction (abort_handler)
        previous ← tail
        tail ← item
        previous.next ← tail
    commit transaction
abort_handler:
    //abort code goes here
```

Algorithmus 3.2: Ansatz zum Umschreiben der `push_back`-Methode mit HTM

In diesem Code sieht man, dass man sich in der transaktionalen Region nicht darum kümmern muss, ob die Threads synchronisiert sind oder nicht. Man muss lediglich programmieren, *was* das Ergebnis sein soll, und nicht, *wie* es erreicht werden soll.

Damit dieser Code allerdings voll funktionsfähig ist, muss noch der Fallback-Handler geschrieben werden. Dieser Code funktioniert bisher nur in dem Fall, wenn gerade kein anderer Thread in den gleichen Speicher schreibt. Bei einem Schreibkonflikt müssen beide Threads ihre Änderungen rückgängig machen und auf eine andere Strategie ausweichen. Diese „andere Strategie“ kann ganz verschieden programmiert sein. Es ist nicht unüblich, dass eine Transaktion erst mehrfach neu gestartet wird, bevor ein anderer Code ausgeführt wird. Damit könnte ein möglicher Fallback-Handler wie in Algorithmus 3.3 programmiert sein. Dieser Fallback-Handler probiert den transaktionalen Ansatz mehrere Male aus (bis zu einer oberen Grenze `max_retries`) und greift danach auf den Algorithmus 3.1 zurück.

Dadurch, dass auf den Algorithmus 3.1 zurückgegriffen wird, wenn der transaktionale Ansatz fehlschlägt, und dieser Algorithmus immer erfolgreich ist, ist für den Algorithmus

```
abort_handler:
  retry_count ← retry_count + 1
  if retry_count ≤ max_retries then
    jump to push_back
  else
    //fall back to algorithm 3.1
  end if
```

Algorithmus 3.3: Naiver fallback handler für die `push_back`-Methode mit HTM

3.3 immer der Fortschritt des Programms gesichert. Damit ist dieser Algorithmus eine gültige Implementierung für die `push_back`-Methode.

Diese Implementierung hat allerdings noch einige Schwächen, die im nächsten Abschnitt behoben werden.

3.3 Implementierung des Schedulings im `MxTasking`

Im `MxTasking` ist momentan der Algorithmus 3.1 als `push_back`-Methode enthalten. Diese wird im Folgenden durch eine transaktionale Implementierung ersetzt:

Als Basis wird der Algorithmus 3.2 herangezogen. Allerdings kann dieser Algorithmus noch nicht als vollständige Grundlage verwendet werden, da damit noch Synchronisationsprobleme zusammenhängen. Das Problem dieses Codes ist, dass bei gleichzeitigem Schreiben einer Transaktion und des Fallback-Handlers die Transaktion einen Fehlschlag detektiert und dann den vermeintlich sicheren Fallback-Pfad ebenfalls rückgängig macht. Denn die Implementierung von Intel® realisiert HTM über das Cache-Kohärenz-Protokoll und transaktionale Änderungen werden nach ihrer Ausführung in den Cache geschrieben, auch wenn die Transaktion nicht committed. Dieses Vorgehen kann dann wieder Konflikte mit nicht transaktional geschriebenen Änderungen auslösen.

Um diesem Problem aus dem Weg zu gehen, muss eine Transaktion abgebrochen werden, wenn gleichzeitig ein Fallback-Pfad aktiv ist. Dies muss über einen Latch realisiert werden. Konkret wird ein Queue-weiter `bool` instantiiert, der angibt, ob gerade ein Fallback-Handler innerhalb einer `push_back`-Methode aktiv ist. Dieser `bool` wird im Folgenden aber nur mit atomaren Instruktionen geschrieben, fungiert also als Latch. Dieser Latch wird im Folgenden `currently_writing` genannt. Wenn also nach einer gestarteten Transaktion festgestellt wird, dass dieser Latch gesetzt ist, dann muss die Transaktion abgebrochen werden. Wenn der Latch hingegen frei ist, so kann die Transaktion fortgeführt werden. Dadurch, dass der Latch innerhalb der Transaktion gelesen wurde, wird sie auch abgebrochen, wenn ein Fallback-Handler aktiv wird, nachdem ihr ausführender Thread den Latch geprüft hat. Innerhalb der Transaktion muss der Latch nicht mehr gesperrt werden, da die Synchronisation durch die Hardware Konflikte bei den Adressen feststellt und dadurch

```
1  //queue-wide latch currently_writing
2  void push_back(QueueItem *item) {
3      item->next(nullptr);
4      if (_xbegin() == _XBEGIN_STARTED) {
5          if (currently_writing) _xabort(0xFF);
6          auto previous = tail;
7          tail = item;
8          _xend();
9          previous->next(item); //this is thread-safe anyway
10     } else {
11         while (__atomic_exchange_n(&currently_writing, true,
12                                    __ATOMIC_SEQ_CST)) {
13             __asm__ volatile ("pause");
14         }
15         auto previous = tail;
16         tail = item;
17         currently_writing = false;
18         previous->next(item);
19     }
20 }
```

Algorithmus 3.4: Basisimplementierung der transaktionalen `push_back`-Methode

implizite Synchronisation erfolgt. Es muss nur der Fall ausgeschlossen werden, dass eine Transaktion gleichzeitig mit einem Fallback-Handler aktiv ist.

Als Fallback-Handler kommt an dieser Stelle dann wieder eine Implementierung wie aus Algorithmus 3.1 zum Tragen. Hierbei muss explizit der Latch `currently_writing` angefordert werden, um anderen Threads Konflikte signalisieren zu können. Innerhalb des kritischen Abschnitts zwischen Anforderung und Freigabe des Latches kann dann eine Latch-freie Implementierung der `push_back`-Methode genutzt werden, da durch den Latch `currently_writing` gesichert ist, dass immer nur ein Thread aktiv in den gemeinsamen Speicherbereich schreiben kann. Die Anforderung des Latches `currently_writing` muss aber wieder synchronisiert werden, da mehrere Threads gleichzeitig auf einen Fallback-Handler zurückgreifen müssen, wenn ihre Transaktionen gegenseitig immer Abbrüche ausgelöst haben. Darum wird die Anforderung des Latches durch einen Spinlock realisiert (vgl. Abschnitt 2.2.2).

Damit ergibt sich der C++ Code aus Algorithmus 3.4 für die Basisimplementierung der transaktionalen `push_back`-Methode. Diese Implementierung verwendet immer nur maxi-

mal eine Transaktion und greift beim Fehlschlag auf den Fallback-Handler zurück. Auffallend ist die Zeile

```
__asm__ volatile (“pause”);
```

Sie generiert die Assembly-Instruktion `pause`, die dem Prozessor signalisiert, dass er sich gerade in einem Spinlock befindet. Damit wird die nächste Instruktion nicht direkt verarbeitet, um nicht zu viel Energie zu verbrauchen und die Speicherordnung nicht zu verletzen [6]. Statt der Konstruktion mit der Inline-Assembly-Instruktion wäre auch die Nutzung des Makros `__mm_pause();` möglich.

3.4 Entwicklung der Warteschlange

Im Rahmen der Implementierung des Transactional Benchmark (siehe Abschnitt 4.1.2) hat sich die Queue von einer HTM-Basisimplementierung weiterentwickelt und wurde über einige Stufen optimiert. Die Messungen in Kapitel 4 werden jeweils nur für die finale Version der Queue durchgeführt.

Trotzdem ist es für die Implementierung einer HTM-Anwendung hilfreich, erst mit einer HTM-Basisimplementierung zu beginnen, bevor dieser Code optimiert wird. Denn Optimierung kann immer durch weitere Parameter in der Implementierung beeinflusst werden.

In den folgenden Abschnitten wird gezeigt, wie die Queue aus Algorithmus 3.4 effizienter gestaltet werden kann.

3.4.1 Nutzung des Latches zum schnellen Abbruch einer Transaktion

In Algorithmus 3.4 wird neben einer Transaktion immer noch ein Latch benötigt, um den Fallback-Handler richtig implementieren zu können. Damit kann man die `push_back`-Methode aber unter Nutzung des Latches effizienter gestalten. Denn der Latch hat die Aussage, dass gerade im Fallback-Handler eine Einfügeoperation in die Queue ausgeführt wird. Also wird eine gerade startende Transaktion wahrscheinlich fehlschlagen, da ein anderer Thread gleichzeitig schreibt. Weil das Öffnen und Committen einer Transaktion zeitlich relativ teuer ist, kann es in diesem Fall effizienter sein, die Transaktion nicht zu öffnen.

Die Frage nach Alternativcode für den Fall, dass der Latch vor dem Beginn einer Transaktion belegt ist, ist relativ leicht zu beantworten. Da die Devise ist, eine Transaktion nur zu öffnen, wenn der Latch frei ist, muss darauf gewartet werden, dass dieser Zustand eintritt. Auch wenn der transaktionale Ansatz nicht noch einmal versucht werden soll, muss der Latch dennoch frei werden, damit er für den Latch-basierten Schreibvorgang des Fallback-Handlers angefordert werden kann. Damit kann die Bedingung in Zeile 4 von Algorithmus 3.4 geändert werden zu:

```
if (!currently_writing && (_xbegin() == _XBEGIN_STARTED)) {
```

Algorithmus 3.5: Optimierung zu Latch-basiertem Ansatz bei aktivem Fallback-Handler

Durch die Short-Circuit-Evaluation des &&-Operators wird die Transaktion nur gestartet, wenn der Latch den Wert **false** hat, also frei ist.

3.4.2 Anpassung der Wartestrategien zwischen Transaktionen

Im Algorithmus 3.4 wird davon ausgegangen, dass nur genau einmal versucht werden soll, eine Transaktion zu öffnen. In Abschnitt 2.3.4 wird aber schon erklärt, dass eine Transaktion auch ohne Datenkonflikt abbrechen kann. Wenn immer nur ein Versuch unternommen wird, eine Transaktion zu starten, wird ziemlich schnell der Fallback-Pfad genutzt und der Latch belegt, obwohl der transaktionale Weg funktionieren könnte. Darum kann es deutlich effizienter sein, die Transaktion mehrfach zu starten.

Da ein Abbruch einer Transaktion aber immer noch von einem Datenkonflikt verursacht werden kann, ist es nicht sinnvoll, die Transaktion unmittelbar neu zu starten, da dann ein konkurrierender Thread gleichzeitig wieder einen Konflikt auslösen würde. Darum ist es sinnvoll, zwischen zwei Transaktionen zu warten.

Wie dieses Warten implementiert wird, kann einen großen Einfluss auf die Gesamtleistung eines Programms haben. Wenn zu lange gewartet wird, wird künstlich Leistung verschenkt. Wenn zu kurz gewartet wird, ist die Wahrscheinlichkeit für einen Konflikt wieder sehr hoch. Gleiches Warten in jedem Thread führt ebenfalls wieder zu Konflikten, weswegen es nicht die beste Strategie ist, die Ausführung mit `nop`-Instruktionen hinauszuzögern.

Um dem Problem des gleichen Wartens in jedem Thread aus dem Weg zu gehen, ist eine Schleife sinnvoll, da damit eine Variable Anzahl an Wartezyklen generiert werden kann. In einem empirischen Test in einem Microbenchmark mit der bisherigen Queue stellt sich heraus, dass es am effizientesten ist, in einer Schleife darauf zu warten, dass der Latch frei wird. Damit ergibt sich für die Wartestrategie folgender Codeschnipsel:

```
while (__atomic_load_n(&currently_writing, __ATOMIC_SEQ_CST)) {
    __asm__ volatile ("pause");
}
```

Algorithmus 3.6: Optimierte Wartestrategie zwischen dem Start zweier Transaktionen

Das Makro `__atomic_load_n` hat hierbei keine weitere Bewandtnis. Es symbolisiert ein atomares Lesen auf der Variable, damit alle Vorgänge auf ihr atomar und sequentiell ausgeführt werden. Im disassemblierten Code wird sichtbar, dass dieses Makro mit einem `movzx`-Befehl umgesetzt wird, es sollte darum keinen weiteren Einfluss auf die Leistung des Programms haben.

Das Makro zu nutzen, kann dennoch sinnvoll sein. Wenn es nicht genutzt wird, könnte nach der Optimierung vom letzten Abschnitt ein Optimierer hier die Variable `currently_writing` als konstant **true** annehmen, auch wenn sie sich zwischenzeitlich ändert. Wenn hingegen das Makro genutzt wird, so wird dieser Codeabschnitt nicht optimiert und bleibt so, wie er vom Programmierer vorgesehen wurde.

Das gerade angesprochene Verhalten ist auch der Grund dafür, dass in Zeile 5 von Algorithmus 3.4 ein `__atomic_load_n` genutzt werden sollte. Der Compiler g++ optimiert ab der Optimierungsstufe -O2 dieses **if** komplett weg[1]. Da dieses **if**-Statement aber essentiell für die korrekte Funktionsweise der Transaktion ist, sollte die Bedingung atomar ausgeführt werden, um vor der Optimierung geschützt zu sein.

Das Warteverhalten des Fallback-Handlers auf den Latch wird momentan durch ein Spinlock wie in Abschnitt 2.2.2 beschrieben. Wie dort schon erwähnt, blockiert der Prozessor bei einer `xchg`-Instruktion. Es kann effizienter sein, erst einmal ohne `xchg` zu prüfen, ob der Latch gerade frei ist. Wenn dies nicht der Fall ist, muss die CPU kein `xchg` ausführen und nicht blockieren, da sowieso weiter gewartet werden muss. Erst wenn der Latch vorher frei ist, kann das `xchg` den Latch erfolgreich sperren. Damit kann in der Bedingung der **while**-Schleife zur Optimierung vor dem `__atomic_exchange_n` ein `__atomic_load_n` auf die Latch-Variable ergänzt werden, um die `xchg`-Instruktionen zu vermeiden.

Somit kann in Algorithmus 3.7 die Endimplementierung der Queue mit HTM angegeben werden. Diese Queue verwendet eine Variable

```
constexpr static std::uint8_t transactional_iterations;
```

die die maximale Anzahl an Versuchen für den transaktionalen Ansatz angibt. Damit kann in Zeile 22 durch konditionale Compilierung bei einem Wert von 0 transaktionalen Versuchen auf die Queue mit atomaren Instruktionen zurückgegriffen werden. Diese ist effizienter, als sich künstlichen Overhead durch einen nicht notwendigen Latch zu schaffen (vgl. dazu auch Abschnitt 2.2.2).

In dieser Implementierung wird außerdem das Makro `__atomic_store_n` verwendet. In der Disassembly des Codes scheint es, wie auch `__atomic_exchange_n`, die `xchg`-Instruktion zu generieren. In diesem Fall hätte der Code

```
currently_writing = false;
```

den gleichen Effekt wie die Nutzung des Makros. Durch die Optimierung in der **while**-Schleife mit dem `__atomic_load_n` wird für das Prüfen der Latch-Variable nur ein `lock`-Präfix generiert, wenn die Latch-Variable vorher frei war. Somit erzeugt das implizite `lock`-Präfix dieser `xchg`-Instruktion in diesem Fall kein unerwünschtes Blockieren der CPU. Wahrscheinlich kann ein einfaches `mov` in der CPU effizienter realisiert werden als ein

[1] Dieses Verhalten ist empirisch durch das Durchsehen der Disassembly des Programms aufgefallen. Disassembliert wurde mit dem Reverse Engineering Tool Cutter [9].

```
1  void MPSCQueue::push_back(QueueItem *item) noexcept {
2      item->next(nullptr);
3      for (std::uint8_t iteration_count = 0u;
4           iteration_count < transactional_iterations; ++iteration_count) {
5          if (!currently_writing && (_xbegin() == _XBEGIN_STARTED)) {
6              if (__atomic_load_n(&currently_writing, __ATOMIC_SEQ_CST)) {
7                  _xabort(0xFF);
8              }
9              QueueItem *prev = tail;
10             tail = item;
11             _xend();
12             prev->next(item);
13             return;
14         }
15         else {
16             while (__atomic_load_n(&currently_writing, __ATOMIC_SEQ_CST)) {
17                 __asm__ volatile ("pause");
18             }
19         }
20     }
21
22     if constexpr (transactional_iterations > 0) {
23         while (__atomic_load_n(&currently_writing, __ATOMIC_SEQ_CST) ||
24                __atomic_exchange_n(&currently_writing, true, __ATOMIC_SEQ_CST))
25         {
26             __asm__ volatile ("pause");
27         }
28         QueueItem *prev = tail;
29         tail = item;
30         __atomic_store_n(&currently_writing, false, __ATOMIC_SEQ_CST);
31         prev->next(item);
32     } else {
33         QueueItem *prev = __atomic_exchange_n(&tail, item, __ATOMIC_RELAXED);
34         prev->next(item);
35         return;
36     }
37 }
```

Algorithmus 3.7: Implementierung der MPSC-Queue mit HTM

`xchg`, aber durch diese Implementierung wird der atomare Charakter der Latch-Variablen deutlicher. Wenn volle Leistung der Queue eine Anforderung an das Programm ist, dann wird die Implementierung ohne das `__atomic_store_n` verwendet.

Kapitel 4

Messungen

In diesem Kapitel wird die Leistung der verschiedenen Queue-Implementierungen verglichen. Dazu werden im nächsten Abschnitt zuerst zwei Benchmarks vorgestellt, die dann im Folgenden die Messdaten liefern.

4.1 Benchmarking vom `MxTasking`

Um die Leistung der Scheduling-Queue beurteilen zu können, muss sie mit einem oder mehreren Benchmarks gemessen werden. In dieser Arbeit werden zwei verschiedene Benchmarks für diese Queue genutzt. Der erste Benchmark arbeitet mit einer Abwandlung von B-Bäumen und der zweite ist ein synthetischer Benchmark, der so gut wie keine Arbeit verrichtet und damit hohe Last auf die Queues erzeugen kann. Diese Benchmarks werden in den nächsten Abschnitten vorgestellt.

4.1.1 B^{link}-tree Benchmark

Der B^{link}-tree Benchmark ist ein „Real-World-Benchmark“, der dafür gedacht ist, die Leistung des `MxTasking` im generellen Sinn zu messen. Er misst die Ausführungszeit seiner Tasks für variable Anzahlen von CPU-Kernen und arbeitet in mehreren Phasen.

Gemessen wird jeweils für eine Benchmark-Iteration. Eine Benchmark-Iteration hat dabei immer eine feste Anzahl an CPU-Kernen, die in dieser Iteration genutzt werden können. Jeder Kern bekommt seinen eigenen Worker-Thread, der Tasks abarbeitet. Für jede Iteration gibt es zwei Phasen. Die erste Phase ist die *fill*-Phase, in der ein sogenannter B^{link}-tree [17] aufgebaut wird. In der zweiten Benchmark-Phase, der *mixed*-Phase, werden verschiedene Operationen wie z.B. Lookups oder Updates auf dem vorher erstellten Baum ausgeführt.

Um Daten für den Baum und darauf auszuführende Operationen zu generieren, wird der Yahoo! Cloud Serving Benchmark (YCSB) [8] genutzt. Dieser generiert in der Standardkonfiguration zwei Dateien: eine *fill*-Datei und eine *mixed*-Datei. In der *fill*-Datei werden

nur INSERT-Operationen generiert, in der *mixed*-Datei hingegen werden Operationen wie READ oder UPDATE generiert. Zusätzlich zu der Operation wird immer ein Zahlenwert erzeugt, mit dem dann ein korrespondierender Knoten erstellt oder abgerufen werden kann.

Aus den von YCSB generierten Dateien werden dann im Benchmark Tasks generiert. Das Einlesen der YCSB-Dateien passiert beim Start der Applikation genau einmal, die Generierung der Tasks erfolgt in jeder Iteration zur Laufzeit, um realistischer an einer möglichen Anwendung zu bleiben. Beim Einlesen der YCSB-Dateien werden dann für den Benchmark 3-Tupel aufgebaut. Diese beinhalten die von YCSB generierte Operation, den von YCSB generierten Zahlenwert als Key sowie einen zusätzlichen Wert als Value. Der Value ist dabei, je nach generierter Operation, ein zufälliger Wert oder 0. In diesem Benchmark werden immer 50 Tupel vom globalen „Workload“ abgefragt und danach die Tasks erzeugt, die nach ihrer Erzeugung direkt gescheduled werden. Dieses Vorgehen wird wiederholt bis die Workload leer ist. Wenn alle Worker-Threads ihre letzten Tasks bearbeitet haben, benachrichtigen sie den Benchmark, der dann beim letzten beendeten Worker-Thread die Ausführung stoppt und die Zeit misst, die seit dem Start des Benchmarks vergangen ist. Neben der Zeit können mit Hilfe von Intel® Performance Countern noch einige andere Daten im Zusammenhang mit der Ausführung erhoben werden. Diese werden in Abschnitt 4.3 weiter erläutert.

Zusätzlich bietet der B^{link}-tree Benchmark noch einige zusätzliche CLI-Optionen. In dieser Arbeit sind dabei nur die Optionen *--exclusive*, mit der das Scheduling-Verhalten geändert werden kann, und *-pd <distance>*, mit der das Prefetching-Verhalten gesteuert werden kann, interessant. Über die *exclusive*-Option werden Tasks immer auf den Kern gescheduled, auf dem ihr angeforderter Knoten im Baum allokiert wurde. Mit der Option *pd* kann eingestellt werden, wie viele Tasks im Voraus geprefetched werden sollen.

4.1.2 Transactional Benchmark

Der Transactional Benchmark ist ein im Rahmen dieser Arbeit entstandener Benchmark, um die Last auf die Queues im Gegensatz zum B^{link}-tree Benchmark deutlich zu vergrößern. Er verfolgt einen ähnlichen Ansatz wie der B^{link}-tree Benchmark, da er auf der gleichen Codebasis arbeitet, allerdings werden in diesem Benchmark einige Konzepte anders realisiert.

Die Generierung der Workload für diesen Benchmark ist identisch zum B^{link}-tree Benchmark. Auch hier wird der YCSB [8] benutzt, allerdings ist für diesen Benchmark nicht relevant, welche Operationen der YCSB generiert. Auch die numerischen Werte werden in den meisten Fällen ignoriert, denn das Hauptziel dieses Benchmarks ist es, möglichst viel Last auf die einzelnen Queues zu simulieren. Um dies zu erreichen, müssen die Tasks so wenig Arbeit wie möglich verrichten, darum werden hier hauptsächlich Tasks benutzt, die buchstäblich keine Arbeit verrichten. Die einzige Aufgabe, die ein Task immer ausführen

muss, ist, sich selbst zu deallokieren, damit der Speicher richtig freigegeben wird. Somit können Tasks sehr schnell abgeschlossen werden und damit neue Tasks aus der Queue herausgenommen bzw. in sie eingefügt werden.

Die Allokation der Tasks erfolgt vor der Messung des Benchmarks. Zuerst wird die Workload eingelesen und dann wird für jedes Tupel der Workload ein Task generiert und auch direkt gescheduled. Das Scheduling erfolgt hier in Form einer Gleichverteilung auf alle genutzten CPU-Kerne. Damit ist gesichert, dass jeder Kern zu Anfang gleich viel Arbeit verrichten muss und nicht direkt leer läuft. Erst nach dem Scheduling des letzten Tasks wird die Messung gestartet.

Mit diesem Konzept wird allerdings noch keine Last auf die Queues erzeugt, die Synchronisation erfordern würde. Da im `MxTasking` MPSC-Queues eingesetzt werden und die Tasks alle allokiert und gescheduled sind, können die Tasks einfach ohne Synchronisation aus den Queues herausgenommen werden. Damit Synchronisation erforderlich wird, wird neben den Tasks, die keine Arbeit verrichten (im Folgenden als `DoNothingTasks` bezeichnet), für jeden Kern ein ausgezeichneter Transfer-Task (`TaskTransfererTask`) erstellt. Dieser Transfer-Task kann auf die Queue seines eigenen Kerns zugreifen und aus dieser Tasks herausnehmen. Diese herausgenommenen Tasks werden von ihm dann über verschiedene Transfer-Strategien auf andere Kerne verteilt und dort gescheduled. Auf einem anderen Kern kann ein verschobener Task nicht noch einmal verschoben werden, er muss also auf diesem Kern ausgeführt werden. Durch die Verwaltung der Transfer-Tasks wird ein verschobener Task vor der nächsten Ausführung des dortigen Transfer-Tasks ausgeführt.

Durch dieses Vorgehen kann Last auf die `push_back`-Methoden simuliert werden und dadurch wird Synchronisation der Worker-Threads erforderlich. Damit greifen also die Implementierungen aus Kapitel 3.

Transfer-Strategien Wie viel Last auf eine Queue generiert wird, hängt von der Transfer-Strategie ab. In diesem Benchmark sind verschiedene Strategien implementiert worden, um das Verhalten der Queues in verschiedenen Situationen beurteilen zu können. Die Haupt-Transfer-Strategien sind:

- *Self:* Bei dieser Strategie wird ein zu verschiebender Task wieder auf den Kern gescheduled, von dem er verschoben wird. Mit dieser Strategie wird keine Synchronisation erforderlich und darum kann der Overhead einer einfachen `push_back`-Operation gemessen werden.

- *Random:* Mit dieser Strategie wird ein Task auf einen zufällig ausgewählten Kern verschoben. Durch den Zufallsgenerator der `Random`-Klasse des `MxTasking`-Projektes wird damit im Endeffekt eine Gleichverteilung von Tasks zu Kernen erzeugt, allerdings ist bei dieser Strategie Synchronisation der `push_back`-Operationen notwendig.

- *AllToCoreZero:* Diese Strategie ist eine künstliche Konstruktion, bei der alle Tasks schließlich auf nur einem Kern ausgeführt werden. Dies kommt jeweils einer Messung mit einem Kern nahe, erzeugt aber ungewöhnlich große Last auf die Queue dieses Kerns.

Die meisten Synchronisationskosten entstehen also bei der Strategie *AllToCoreZero*. Dieses Beispiel ist rein synthetisch, da damit in einem Mehrkernsystem nur ein Kern effektiv genutzt wird. Bei dieser Strategie ist sehr gut sichtbar, wie viel Zeit mit der Synchronisation der Threads verbracht wird. Wenn eine Iteration mit nur einem Kern durchgeführt wird, muss keine Synchronisation erfolgen. Wenn aber *AllToCoreZero* mit mehr als einem Kern durchgeführt wird, ist die Zeit, die mehr als mit nur einem Kern benötigt wird, reine Synchronisationszeit. Damit kann dann evaluiert werden, wie transaktionaler Speicher im Gegensatz zu atomaren Instruktionen skaliert[1].

Load-Strategien Neben den Transfer-Strategien bietet der Transactional Benchmark auch verschiedene Load-Strategien. Load meint damit die Arbeit, die ein Task verrichtet. Wie oben schon erwähnt, ist die primäre Strategie für diesen Benchmark der `DoNothingTask`. Um aber etwas praxisnaher zu werden, gibt es auch Strategien, die geringfügig Arbeit verrichten. Implementiert sind zum Beispiel folgende Task-Arten:

- `SumArrayTask`: Bei dieser Task-Art wird zu jedem Task ein uninitialisiertes Array alloziert, das dann vom Task aufaddiert wird.

- `RandomAddTask`: Diese Task-Art rechnet eine temporäre Variable ± den von YCSB generierten numerischen Wert. Wenn der temporäre Wert 0 ist, wird der YCSB-Wert addiert, ansonsten subtrahiert. Die Anzahl der Wiederholungen wird zufällig von std::rand() bestimmt und in der Workload gespeichert, sodass der Wert für einen Benchmark-Aufruf jeweils konstant bleibt.

  Bei dieser Task-Art werden auch die zwei Phasen des Benchmarks ausgenutzt. Der zufällige Wert, wie oft die Addition stattfinden soll, ist nur in der *fill*-Phase interessant, in der *mixed*-Phase wird dieser Wert immer auf 0 gesetzt.

Durch diese Implementierung können sehr schnell viele Ergebnisse gemessen werden, um die beiden Queue-Implementierungen mit und ohne HTM zu vergleichen. Im Folgenden werden nicht alle Ergebnisse aller Kombinationen vorgestellt, da dies den Rahmen dieser Arbeit sprengen würde. Trotzdem sind die Ergebnisse verschiedener Kombinationen nützlich, um

[1] Diese Aussage ist nicht ganz korrekt, kann aber in diesem Fall so angenommen werden. In Wirklichkeit entsteht noch zusätzliche Zeit, die beim Zugriff auf den Speicher entsteht. Da aber in den Fällen mit und ohne transaktionalen Speicher diese Zeit, durch die gleiche Allokation bedingt, genau gleich sein sollte, rechnet sich dieser Anteil wieder heraus.

Anwendungsfälle bestimmen zu können, in denen HTM besonders gut bzw. schlecht skaliert. Darum werden im Folgenden verschiedene Strategien evaluiert, um Rückschlüsse auf Real-World-Applications ziehen zu können.

4.2 Rahmenbedingungen der Benchmarks

Die Ergebnisse der folgenden Abschnitte werden für beide Benchmarks immer auf dem gleichen Server gemessen. Dieser hat zwei Intel® Xeon® Gold 6226 Prozessoren und 192GB DDR4-RAM verbaut. Im BIOS ist Hyperthreading aktiviert, sodass in der gesamten Konfiguration 48 logische Kerne zur Verfügung stehen. Das Betriebssystem ist Ubuntu 20.04 LTS.

Wegen aktiviertem Hyperthreading und dem „Pinning“ der Threads (s. u.) auf verschiedene logische CPU-Kerne ist die Sortierung der Kerne entscheidend. Sie sieht wie folgt aus:

- Kerne 1-12: Reelle Kerne des ersten CPU-Sockets
- Kerne 13-24: Hyperthreading-Kerne des ersten CPU-Sockets
- Kerne 25-36: Reelle Kerne des zweiten CPU-Sockets
- Kerne 37-48: Hyperthreading-Kerne des zweiten CPU-Sockets

Über das C++-Makro `CPU_SET` können die Threads auf dem Linux-System auf verschiedene CPU-Kerne verteilt werden, sodass gesichert sein kann, dass sich nicht zwei Worker-Threads den gleichen Kern teilen. Damit werden die Worker-Threads fest auf diese Kerne „gepinnt“ und sollten vom Betriebssystem dann nicht mehr auf andere Kerne verschoben werden.

Um Unterschiede im Scheduling des Betriebssystems auszugleichen, werden immer mehrere Werte gemessen. Gerade bei voller Kernkonfiguration wird das Betriebssystem zwischendurch auf einem oder mehreren Kernen von Worker-Threads interagieren. Um diese Effekte herauszurechnen, wird jeder Wert für den Transactional Benchmark 5 Mal gemessen und davon das arithmetische Mittel gebildet. Damit werden diese Effekte geglättet. Beim B^{link}-tree Benchmark wird auch jeweils mit 5 Iterationen gemessen. In einigen Fällen ist es jedoch nicht sinnvoll oder praktikabel, mit vielen Iterationen zu messen, daher wird dort dieser Parameter herabgesetzt.

In der *Random*-Konfiguration des Transactional Benchmark wird kein echter Zufallsgenerator genutzt, da die Werte dieses Benchmarks dann sehr schlecht zu vergleichen wären. Stattdessen wird auf jedem Kern eine Instanz der gleichen Zufallsgeneratorklasse genutzt, die aber für jeden Kern einen anderen Anfangswert (Seed) hat. Dieser Seed ist jeweils die ID des ausführenden CPU-Kerns. Damit erzielen die Worker-Threads untereinander unterschiedliches Verhalten, um eine Art Zufall zu simulieren, sind aber trotzdem in verschiedenen Benchmark-Iterationen vergleichbar.

Die Workload beträgt jeweils 20.000.000 Elemente, der Transactional Benchmark verwendet aber im Folgenden die Befehlszeilenoperation "`-tr 3`", womit aus jedem YCSB-Tupel jeweils 3 Tasks erstellt werden. Dieses Vorgehen weitet die Zeit des Benchmarks auf und schafft damit größere Abstände in den gemessenen Zeiten bei verschiedenen Kernkonfigurationen. Beim Transactional Benchmark werden also 60.000.000 Tasks plus die `TaskTransfererTasks` ausgeführt.

Bei den transaktionalen Messungen werden im Folgenden immer maximal 10 Transaktionen verwendet. Dies wird in den Plots abkürzend auch mit HTM(10) bezeichnet.

Durchsatz im Transactional Benchmark Die Durchsatzberechnung erfolgt aus den gemessenen Werten für die ausgeführten Tasks und die dafür benötigte Zeit. Dabei schließt die Zahl der ausgeführten Tasks im Transactional Benchmark auch die `TaskTransfererTasks` mit ein. Der Durchsatz ist also nicht der Nettodurchsatz der vom Nutzer gewählten Tasks, sondern der Durchsatz aller Tasks, die der Worker-Thread abarbeitet. Dies sorgt dafür, dass der Wert des Durchsatzes dem nachher reellen Wert im `MxKernel` ohne Verwendung von `TaskTransfererTasks` nahe kommt.

4.3 Vergleichsgrößen für Benchmarks - Intel® Architectural Performance Monitoring

Um bessere Vergleiche zwischen verschiedenen Benchmark-Konfigurationen ziehen zu können als nur über den Durchsatz, werden die Intel® Architectural Performance Monitoring Events [2] in den hier vorgestellten Benchmarks genutzt. Das sind CPU-integrierte Zähler, die Metadaten über die Ausführung bereitstellen können. Intel® listet sieben vordefinierte Zähler, die z. B. Aufschluss über CPU-Zyklen, ausgeführte Instruktionen oder auch Cache-Misses des Last-Level-Cache (LLC) geben.

Tatsächlich gibt es aber je nach Hardware-Support noch deutlich mehr solcher Zähler. In dieser Arbeit werden hauptsächlich vier dieser Zähler mit jeder Benchmark-Iteration mit gemessen:

- `UnHalted Core Cycles`: Dies sind die ausgeführten CPU-Zyklen während einer Benchmark-Iteration
- `Instructions Retired`: Dies sind die ausgeführten Instruktionen in einer Iteration
- `CYCLE_ACTIVITY.STALLS_MEM_ANY`: Hiermit werden die CPU-Zyklen gemessen, in denen auf Speicheroperationen gewartet wird
- `TX_MEM.ABORT_CONFLICT`: Hiermit werden die Konflikte zwischen verschiedenen HTM-Transaktionen gemessen

Diese Performance-Events geben Aufschluss über die Gründe, warum sich der Durchsatz in der gemessenen Form ergibt. Für die weiteren Betrachtungen sind die CPU-Zyklen und die Instruktionen nicht von sehr großer Bedeutung, die Memory Stalls und die HTM-Konflikte sind deutlich aussagekräftiger.

4.4 Ergebnisse der Benchmarks

In den folgenden Abschnitten werden die Ergebnisse der einzelnen Benchmarks beschrieben und verglichen. Für die nicht angegebenen Konfigurationsparameter gelten jeweils die im Abschnitt 4.2 angegebenen Werte.

4.4.1 Transactional Benchmark mit und ohne Transaktionen

Die folgenden Teilabschnitte stellen die verschiedenen Ergebnisse des Transactional Benchmark vor. Dafür wird zuerst eine Basislinie festgelegt, die aufzeigt, wie viel Verwaltungsaufwand eine einzelne Transaktion im Gesamtbild verursacht. Danach wird dann evaluiert, inwiefern die HTM-Implementierung anders skaliert als die Queue-Implementierung mit atomaren Instruktionen.

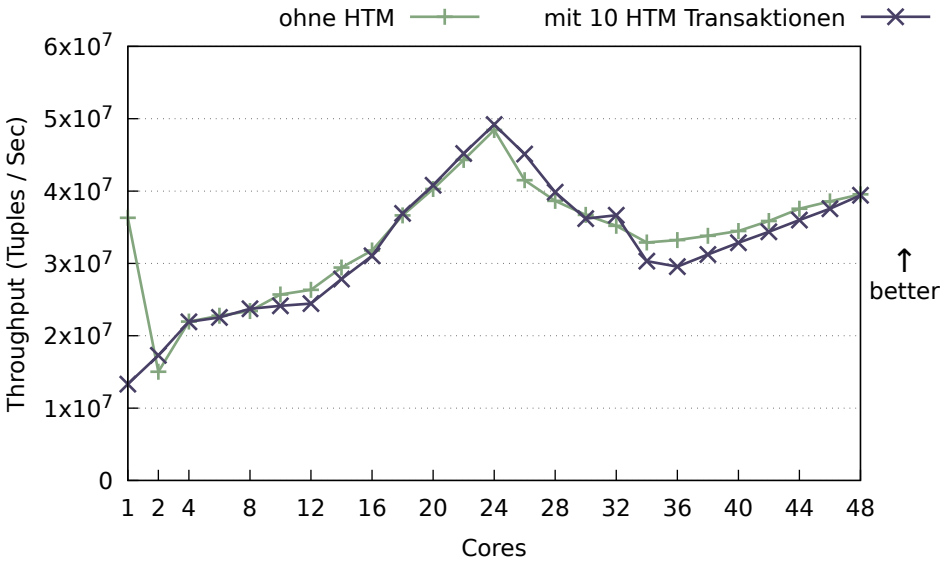

Abbildung 4.1: Durchsatz des Transactional Benchmark in der Konfiguration `DoNothingTask` / *Self* mit und ohne HTM

Konfiguration DoNothingTask/Self Um eine Vorstellung davon zu bekommen, wie viel Verwaltungsaufwand eine einzelne Transaktion verursacht, ist zuerst in der Konfiguration mit dem `DoNothingTask` in dem Transfer-Modus *Self* gemessen worden. Damit werden also alle Tasks wieder auf den eigenen Kern gescheduled und der Vergleich der Messungen

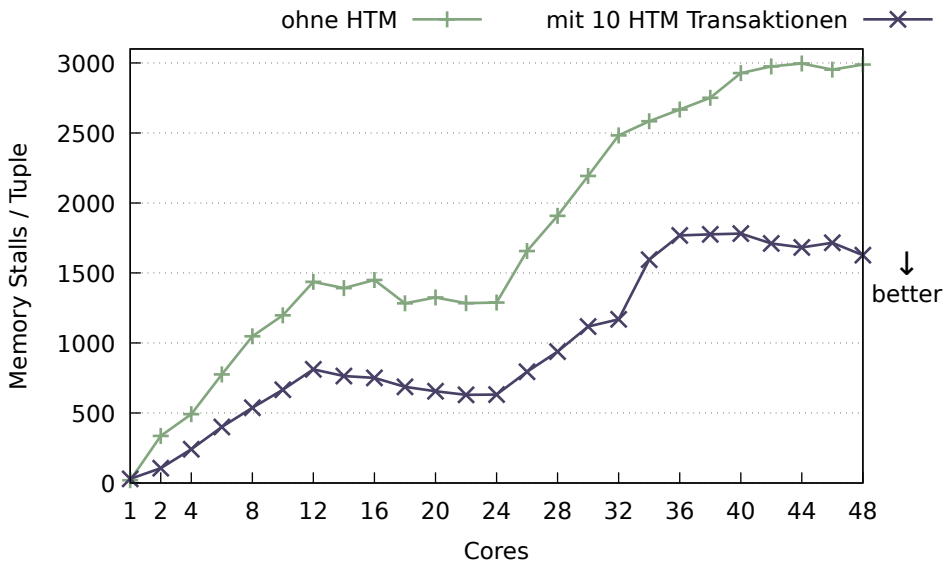

Abbildung 4.2: Vergleich der Memory-Stalls des Transactional Benchmark in der Konfiguration `DoNothingTask` / *Self*

mit und ohne HTM zeigt nur den Verwaltungsoverhead der Queues, da der restliche Code des Benchmarks jeweils identisch ist. Das Ergebnis ist in Abbildung 4.1 angegeben. Diese Abbildung ist allerdings nur eine Momentaufnahme. Durch unterschiedliche Belegung von Speicherbänken zwischen verschiedenen Benchmark-Ausführungen kommt es gerade in dieser Konfiguration sehr schnell zu instabilen Ergebnissen. Bei der Messung dieses Benchmarks an mehreren verschiedenen Tagen ergibt sich bei 48 Kernen teilweise ein Unterschied im Durchsatz von 10.000.000 Tupeln pro Sekunde. Dieses Verhalten tritt aber in beiden Implementierungen gleichsam auf. Darum wird hier eine Momentaufnahme angegeben, in der der Durchsatz in beiden Implementierungen zufällig relativ ähnlich aussieht. Trotzdem sei noch erwähnt, dass die Ergebnisse bei bis zu 24 Kernen in verschiedenen Läufen ziemlich stabil bleiben. Dadurch ist dieser Teil des Plots sehr wohl vergleichbar.

Bei dieser Benchmarkkonfiguration fällt auf, dass die Transaktionen gerade auf nur einem Kern einen deutlichen Verwaltungsoverhead erzeugen. Wenn Transaktionen also bei Code eingesetzt werden, der nicht synchronisiert werden muss, so erzeugen sie nur hohe Kosten und bringen keinen Mehrwert. Ab 2 Kernen halten die Transaktionen bis zu 24 Kernen aber ein ähnliches Niveau zur Implementierung ohne HTM.

Die Implementierungen scheinen beide gut mit Hyperthreading zu skalieren, denn in beiden Hyperthreading-Regionen (von 13-24 und von 37-48 Kernen) können jeweils Leistungssteigerungen gemessen werden.

Der Grund dafür, warum gerade in der Hyperthreading-Region eine große Leistungssteigerung zu beobachten ist, und vorher nicht, liegt an den Speicherzugriffen. Aus Abbildung 4.2 kann man entnehmen, dass die Memory-Stalls in diesen Kernregionen jeweils

auf gleichem Niveau bleiben oder aber deutlich schwächer anwachsen. Das ist darauf zurückzuführen, dass die Tasks jeweils zwei globale Objekte anfragen müssen und dies mit Hyperthreading effizienter realisiert werden kann. Während ein Thread auf einem CPU-Kern auf Speicher wartet, kann der andere bereits weiterarbeiten.

Bei der Konstruktion eines Tasks wird ihm ein `benchmark`-Objekt mitgegeben, um verschiedene Parameter des Benchmarks verfügbar zu machen. Weiterhin wird die so genannte `runtime`-Klasse benötigt, da darin eine Methode zur Deallokation eines Tasks vorhanden ist. Wie in Abschnitt 4.1.2 schon erwähnt wurde, muss auch ein `DoNothingTask` immer dafür sorgen, dass sein eigener Speicher freigegeben wird. Darum muss er das `runtime`-Objekt kennen und darüber den Allokator aufrufen, um seinen Speicher korrekt freigeben zu lassen. In der Hyperthreading-Region ist dieses Objekt durch den anderen Thread auf diesem Kern mit großer Wahrscheinlichkeit schon im Cache, sodass darum dieses Objekt nicht noch einmal angefragt werden muss, denn nach Marr und Binns [19] wird der Cache von den zwei Threads auf dem gleichen CPU-Kern geteilt.

Somit ist auch gut zu erklären, warum der Durchsatz ab 24 Kernen in beiden Implementierungen deutlich einbricht. Ab dann werden Kerne des zweiten CPU-Sockets genutzt. Hier müssen die globalen Objekte aus dem Hauptspeicher abgefragt werden, da sich die CPUs keinen Cache mehr teilen. Bei bis zu 24 Kernen können die globalen Objekte über den L3-Cache abgerufen werden.

Weiterhin fällt bei den Memory-Stalls auf, dass die transaktionale Implementierung in jedem Fall deutlich kürzer auf Speicher wartet als die atomare Implementierung. Da der Durchsatz aber ungefähr gleich verläuft, wird damit der Overhead der transaktionalen Konfliktdetektion durch die Hardware deutlich.

Konfiguration DoNothingTask/Random Wenn jetzt allerdings Synchronisationslast auf die Queues gegeben wird, ergibt sich der Durchsatz aus Abbildung 4.3. Bei zwei Kernen entstehen hohe Synchronisationskosten, da dabei die beiden Queues das höchste Konfliktpotenzial haben. Danach werden die Konflikte weniger und durch die größere Anzahl von parallel arbeitenden Kernen steigt der Durchsatz stetig bis zu 24 Kernen an. Ab dann wird die Region des ersten CPU-Socket verlassen und die Tasks werden auch auf dem zweiten Socket gescheduled. Durch die Synchronisation zwischen den zwei CPUs und das Verlassen des L3-Caches der ersten CPU müssen ab dann mehr Tasks Hauptspeicherzugriffe absetzen, was wiederum viele CPU-Zyklen bedeutet, in denen auf Speicher gewartet wird.

Die HTM-Transaktionen scheinen gerade über die Socket-Grenze hinaus gut zu skalieren, denn es wird bei 48 Kernen der Durchsatz von 24 Kernen sogar überschritten. Dahingegen scheint die Implementierung ohne HTM ihr Leistungsmaximum bei 24 Kernen zu erreichen. Die HTM-Implementierung scheint darüber hinaus durch das Cache-Kohärenz-Protokoll und andere Lastverteilung auf dem Speicher gut mit dem zweiten Socket interagieren zu können, ohne wirklich viel Leistung zu verlieren. Auch hierbei, wie schon bei

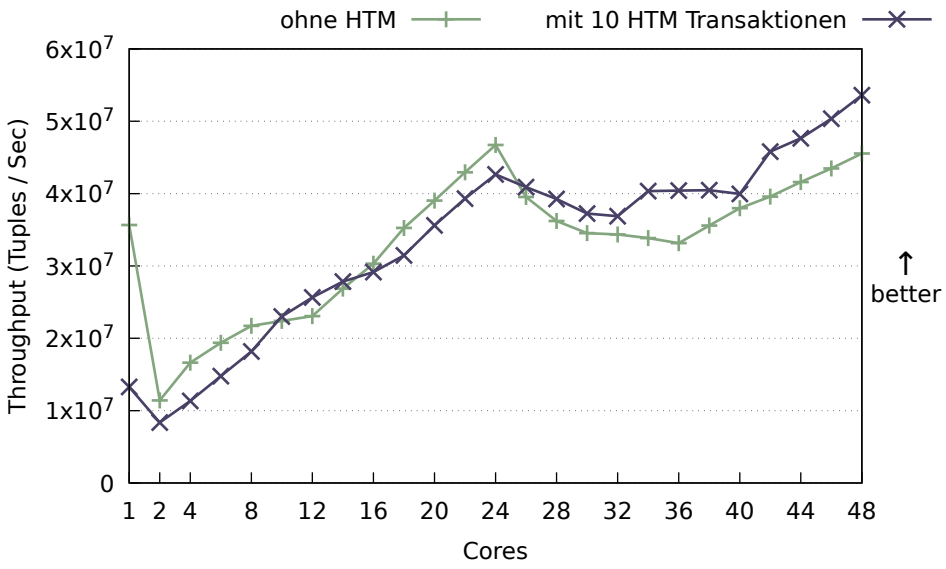

Abbildung 4.3: Durchsatz des Transactional Benchmark in der Konfiguration `DoNothingTask` / *Random* mit und ohne HTM

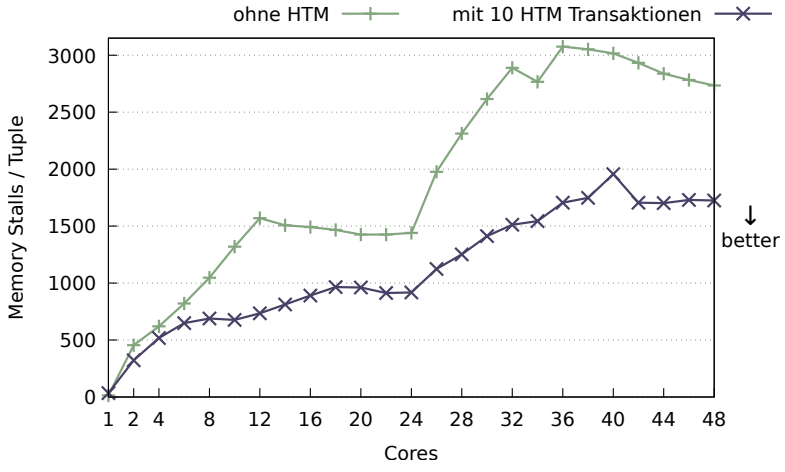

Abbildung 4.4: Vergleich der Memory-Stalls des Transactional Benchmark in der Konfiguration `DoNothingTask` / *Random*

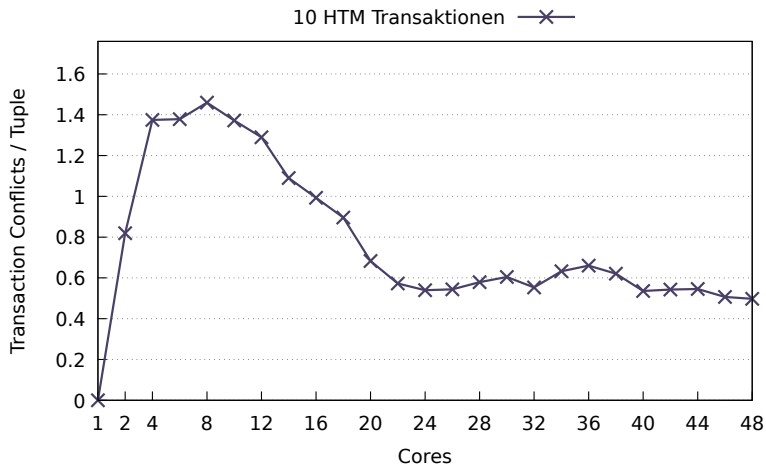

Abbildung 4.5: Abbrüche von HTM Transaktionen pro Task in der Konfiguration `DoNothingTask` / *Random*

der Konfiguration `DoNothingTask` / *Self*, machen die günstigeren Speicherzugriffe dieses Verhalten möglich. Dies ist in Abbildung 4.4 dargestellt. Da in der Implementierung ohne HTM teilweise fast doppelt so lang auf Speicher gewartet wird, wird damit die zusätzliche Zeit der Konfliktdetektion überschritten und die HTM-Implementierung hat somit einen höheren Durchsatz.

Bisher sind immer 10 HTM-Transaktionen maximal erlaubt worden. Bei zu wenig Transaktionen wird zu früh auf den Fallback-Handler zurückgegriffen, der dann wieder in seiner Leistung beschränkt ist, bei zu vielen Transaktionen wird möglicherweise zu viel konfliktbehafteter Code ausgeführt, ohne derweil Fortschritt zu erzielen. Tatsächlich werden diese 10 möglichen Transaktionen aber nie wirklich genutzt. Abbildung 4.5 zeigt, dass die Maximalzahl abgebrochener Transaktionen pro Task bei etwas mehr als 1,4 liegt. Da bei den Tasks auch die `TaskTransfererTasks` mit eingerechnet sind, sollte der tatsächliche Wert aber etwas höher liegen, da die `TaskTransfererTasks` immer auf dem gleichen Kern bleiben und damit von sich aus tendenziell weniger Konflikte auslösen. Das bedeutet, dass die optimale Anzahl maximal erlaubter Transaktionen ungefähr zwischen 2 und 3 liegen sollte. Da diese Werte Durchschnittswerte über jeweils eine ganze Benchmark-Iteration sind, ist die Abbruchrate im Einzelnen wahrscheinlich etwas höher oder niedriger, trotzdem sollte sich das Bild bei maximal 3 erlaubten Transaktionen nicht sonderlich verändern.

Konfiguration RandomAddTask/Random Bei dieser Konfiguration wird, wie in der Beschreibung des Transactional Benchmark schon erwähnt, eine Unterscheidung zwischen *fill*-Phase und *mixed*-Phase vorgenommen. In der *fill*-Phase wird ein Wert zufällig häufig aufaddiert und dann wieder subtrahiert. Somit wird gesichert, dass der temporäre Wert nicht überläuft und ein Speicherfehler entsteht. In der *mixed*-Phase wird dieses Vorgehen komplett übersprungen, da die Anzahl der Iterationen in jedem Fall 0 ist. Somit ist inner-

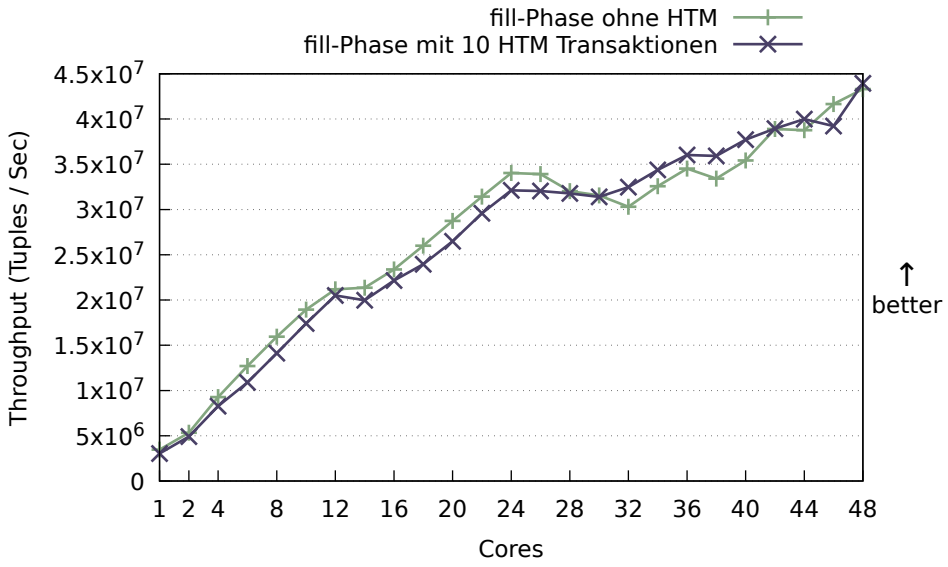

Abbildung 4.6: Vergleich des Durchsatzes der *fill*-Phasen des Transactional Benchmark in der Konfiguration `RandomAddTask` / *Random*

halb dieses Benchmarks ein Vergleich zwischen einer Workload mit und einer Workload ohne Arbeit in den Tasks direkt möglich. Da die Workload ohne Arbeit sehr ähnlich zum `DoNothingTask` ist, sind die Ergebnisse der *mixed*-Phase sehr ähnlich zu den Ergebnissen der Konfiguration `DoNothingTask` / *Random*. Darum wird im Folgenden nur die *fill*-Phase dieser Konfiguration betrachtet. Bei dieser Phase werden zwar zufällige Werte benutzt, um Last in den Tasks zu simulieren, allerdings wird für die Zufallsgeneratoren in jedem Lauf des Benchmarks jeweils der gleiche Seed genutzt. Darum sind die Werte der beiden Implementierungen trotzdem vergleichbar.

Die Werte beider Implementierungen liegen hierbei immer relativ dicht beieinander. Hier scheint die Implementierung ohne HTM innerhalb der Region des ersten Sockets besser zu skalieren, während die HTM-Implementierung beim zweiten Socket besser skaliert. HTM scheint also auch unter (geringer) Last besser über die Socket-Grenze hinweg operieren zu können als Implementierungen mit atomaren Instruktionen. Dies ist wieder durch die Speicherzugriffe zu erklären. Dadurch, dass die HTM-Implementierung keine atomare `lock`-Instruktion mit strikter Speicherordnung benutzt oder die CPU für Instruktionen blockieren muss, kann sie während des Speicher-Fetchings schon weitere Instruktionen ausführen.

Konfiguration DoNothingTask/AllToCoreZero Nun wird eine Queue sehr hoher Last ausgesetzt. Bei dieser Messung werden einige Benchmark-Parameter verändert. Wegen der großen Instabilität werden hierbei nur 3 Iterationen durchgeführt, da die Messungen durch Mittelwertbildung nicht viel stabiler werden. Außerdem wird eine kleinere Workload mit nur 20.000.000 Elementen (im Gegensatz zu den sonst üblichen 60.000.000 Elementen)

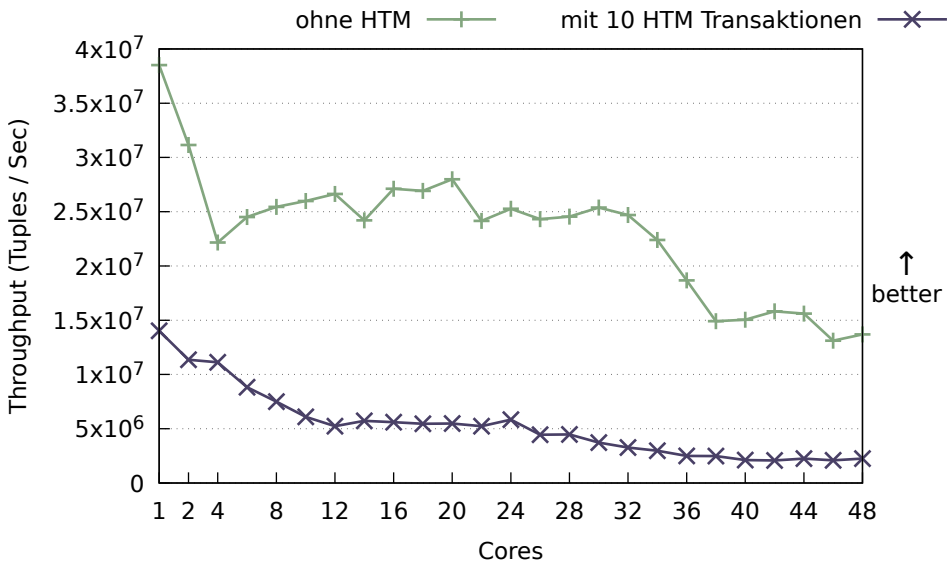

Abbildung 4.7: Durchsatz des Transactional Benchmark in der Konfiguration `DoNothingTask` / *AllToCoreZero* mit und ohne HTM

benutzt, da die Zeit dieser Messung sonst sehr schnell sehr groß wird und die tatsächlichen Zeitwerte nicht wirklich relevant sind. Relevant ist nur das Verhältnis von den Implementierungen mit und ohne HTM und das ist auch in dieser Konfiguration vergleichbar.

In Abbildung 4.7 ist der Durchsatz dieser Konfiguration dargestellt. Da hier sehr hohe Last auf eine einzige Queue erzeugt wird, in die alle Threads gleichzeitig hineinschreiben wollen, so entstehen fast in jeder Situation Konflikte. Darum werden jeweils alle verfügbaren Transaktionen abgebrochen oder gar nicht erst gestartet und es wird nahezu in jedem Fall auf den Fallback-Handler zurückgegriffen. Da dieser fast identisch zur Implementierung ohne HTM ist, kann die HTM-Implementierung nur langsamer sein. Durch die Abbrüche der Transaktionen und das vorherige Warten wird aber zusätzlicher Overhead erzeugt. In der HTM-Implementierung wird der Latch `currently_writing` sehr oft belegt sein und darum wird zwischen den Versuchen, die Transaktionen erneut zu starten, jeweils gewartet, bis dieser Latch wieder frei ist. Wenn er aber während dieses Wartens direkt von einem anderen Fallback-Handler wieder belegt wird, so warten die Threads ziemlich lange, bevor sie selbst auf ihren Fallback-Pfad zurückgreifen.

Was bei der Messung ohne HTM auffällt, ist der Durchsatz bei 2 Kernen. Dieser ist höher als in der Konfiguration `DoNothingTask` / *Random*. Dies wird dadurch bedingt, dass der Kern, auf den die Tasks verschoben werden, gleichzeitig auch die Tasks selbst abarbeiten muss. Darum wird sein `TaskTransfererTask` immer im Wechsel zu anderen auf ihn verschobenen Tasks ausgeführt. Somit ist das Konfliktverhalten in diesem Fall deutlich geringer, als es bei einem zufälligen Scheduling sein kann.

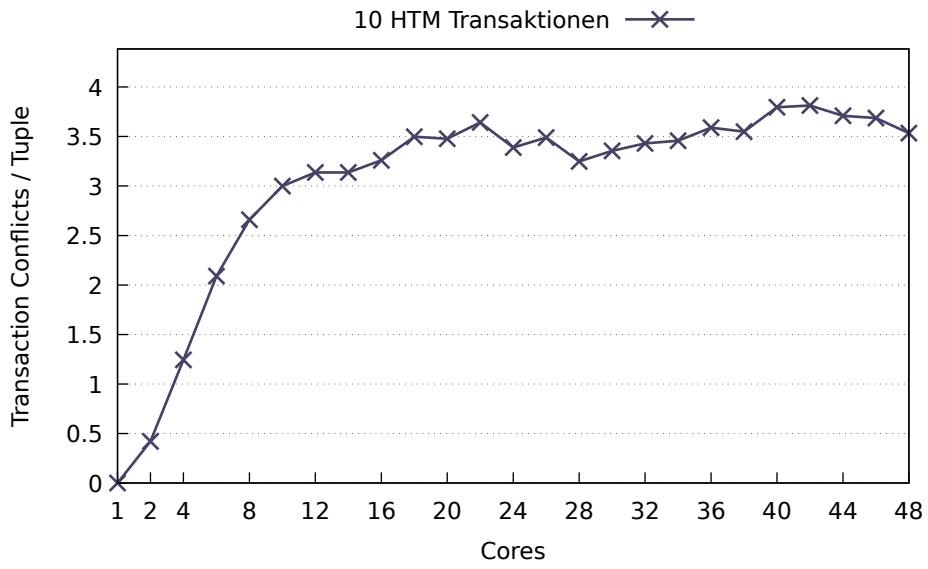

Abbildung 4.8: Abbrüche von HTM Transaktionen pro Task in der Konfiguration `DoNothingTask` / *AllToCoreZero*

In Abbildung 4.8 sind die HTM-Konflikte pro Task in dieser Konfiguration aufgezeigt. Da hierbei wieder die `TaskTransfererTasks` mit eingerechnet sind, ist der tatsächliche Wert der Konflikte der eigentlichen Tasks höher. Hierbei können die `TaskTransfererTasks`, die auf ihre eigenen Kerne neu gescheduled werden, keine Konflikte mit anderen Tasks auslösen. Trotzdem werden damit nicht die möglichen 10 Abbrüche pro Einfügeoperation erreicht. Dies liegt auch daran, dass nach der Optimierung aus Abschnitt 3.4.1 manche Transaktionen gar nicht erst gestartet werden, wenn gerade ein Fallback-Pfad aktiv ist. Darum ist davon auszugehen, dass bei den `DoNothingTasks` kaum eine Einfügeoperation durch Transaktionen erfolgreich verläuft.

Konfiguration SumArrayTask/Random Als letztes wird noch eine Konfiguration des Transactional Benchmark betrachtet, die etwas speicherintensiver ist. Dafür wird jetzt der `SumArrayTask` mit einer Array-Größe von 64 Bytes verwendet. Da die Größe des Arrays identisch mit der Größe einer Cache-Line ist, muss nur eine Speicheroperation ausgeführt werden, um das gesamte Array in den L1-Cache zu laden. Trotzdem sieht man in Abbildung 4.9, dass dies für den Transactional Benchmark einen deutlichen Leistungsverlust bedeutet, denn der transaktionale Speicher skaliert besser mit wenigen Speicheroperationen. Dies wird an den CPU-Zyklen deutlich, in denen auf Speicher gewartet wird. In der Konfiguration `DoNothingTask`/*Random* hat sich herausgestellt, dass transaktionaler Speicher mehr Durchsatz als die Implementierung ohne HTM erzielt, wenn die Memory-Stalls in der Implementierung ohne HTM um den Faktor 1,5 bis 2 höher liegen als in der HTM-Implementierung. Da z. B. bei 48 Kernen die Memory-Stalls nur um einen Faktor 1,25 höher liegen, geht dabei der Leistungsgewinn von HTM verloren.

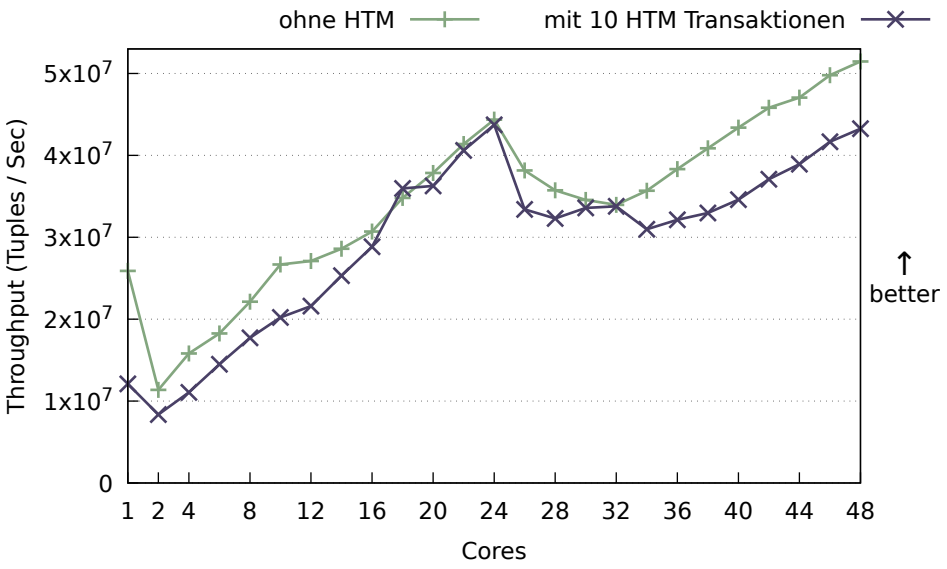

Abbildung 4.9: Durchsatz des Transactional Benchmark in der Konfiguration `SumArrayTask` / *Random*

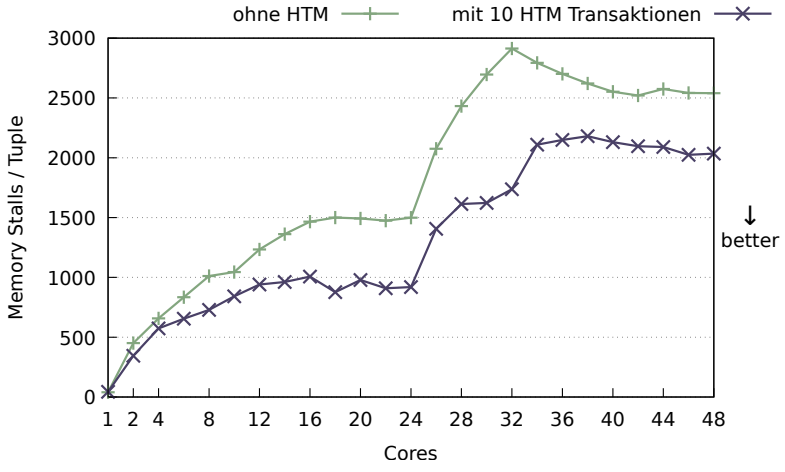

Abbildung 4.10: Vergleich der Memory Stalls im Transactional Benchmark in der Konfiguration `SumArrayTask` / *Random*

4.4.2 B^{link}-tree Benchmark mit und ohne Transaktionen

In den folgenden Teilabschnitten werden nun die Ergebnisse des B^{link}-tree Benchmark angegeben.

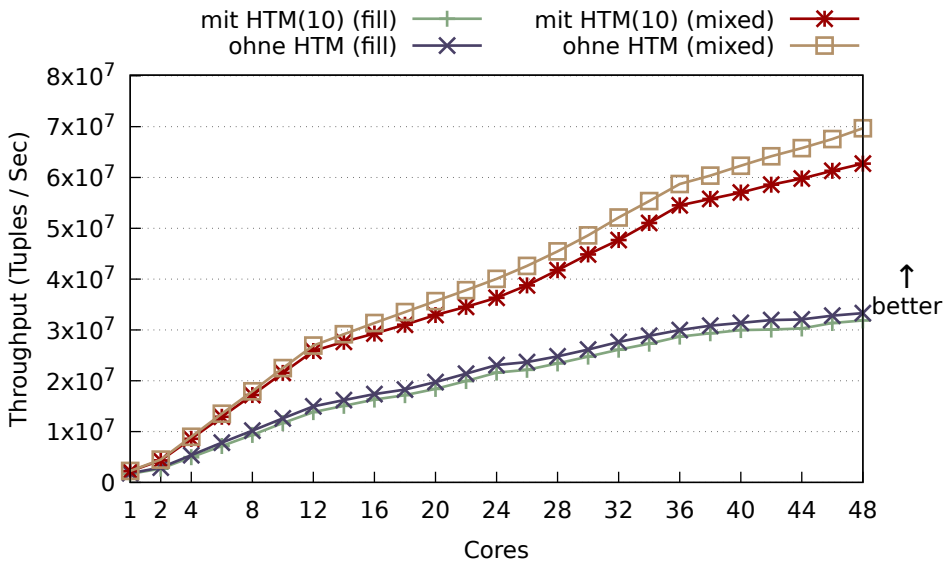

Abbildung 4.11: Durchsatz im B^{link}-tree Benchmark ohne Prefetching und ohne exklusiven Knotenzugriff

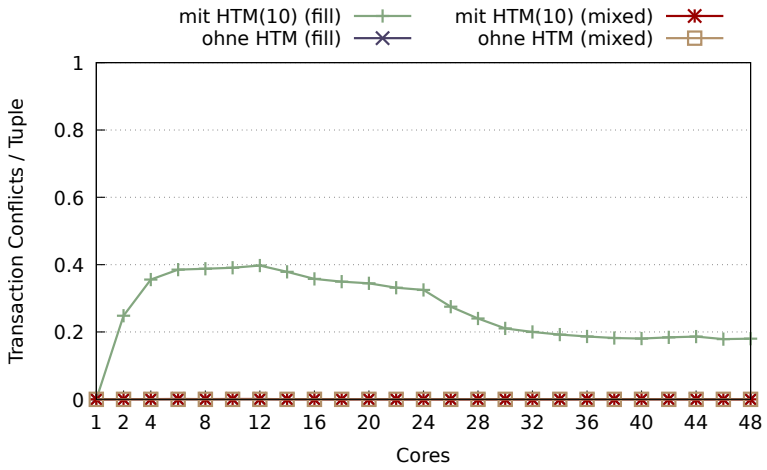

Abbildung 4.12: HTM Konflikte im B^{link}-tree Benchmark ohne Prefetching und ohne exklusiven Knotenzugriff

B^{link}-tree Benchmark ohne Prefetching und exklusiven Knotenzugriff In der Standardkonfiguration erzeugt der B^{link}-tree Benchmark den Durchsatz aus Abbildung 4.11. In dieser Konfiguration verlaufen die *fill*-Phasen beider Implementierungen nahezu

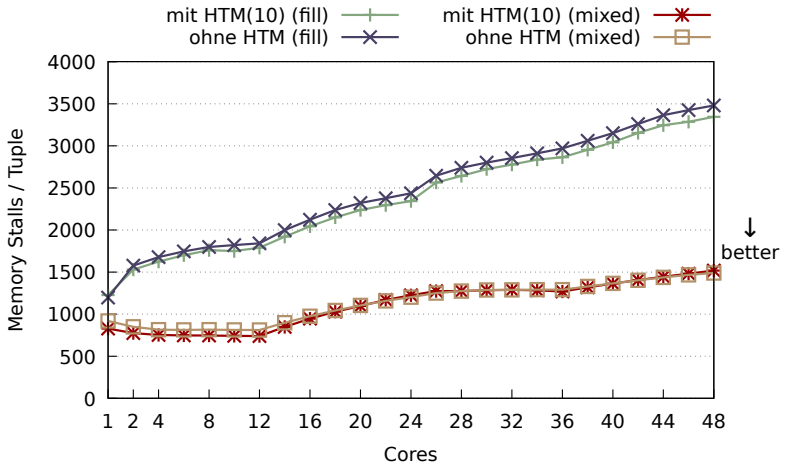

Abbildung 4.13: Memory Stalls im B^{link}-tree Benchmark ohne Prefetching und ohne exklusiven Knotenzugriff

identisch, bei der *mixed*-Phase skaliert die Implementierung ohne HTM allerdings deutlich besser.

Dieses Verhalten rührt daher, dass in der *mixed*-Phase keine wirkliche Synchronisation notwendig ist. Dies wird aus Abbildung 4.12 ersichtlich. In der *mixed*-Phase werden kaum Transaktionen abgebrochen und daher wird wiederum nur der Overhead von Transaktionen gegenüber atomaren Instruktionen gemessen. Da schon aus dem Transactional Benchmark bekannt ist, dass eine einzelne Transaktion deutlich länger dauert als eine atomare Instruktion, wird damit der Unterschied im Durchsatz einfach erklärbar.

In der *fill*-Phase hingegen ist mehr Synchronisation erforderlich. Es werden einige Transaktionen abgebrochen und darum liegt auch mehr Last auf den Queues. Damit wird der Overhead einer Transaktion gegenüber einer atomaren Instruktion geringer. Aus diesem Grund sind die Verläufe der beiden Plots relativ identisch, wenngleich die atomaren Instruktionen immer noch ein kleines Leistungsplus erzielen.

Dass der transaktionale Speicher in diesem Benchmark keine Leistung gegenüber der Implementierung mit atomaren Instruktionen gewinnen kann, liegt auch zu einem großen Teil daran, dass der Benchmark sehr speicherintensiv ist. Dadurch, dass ein B^{link}-tree traversiert wird, entstehen sehr viele Speicherzugriffe, sodass die Speicheroperationen der Queues dabei nicht mehr wirklich ins Gewicht fallen. Durch die vielen Speicherzugriffe benötigen die Tasks gleichzeitig auch wieder mehr CPU-Zyklen und somit wird die Synchronisationslast gesenkt. Dies wird in Abbildung 4.13 deutlich. Dort sieht man wenig Unterschiede in den Memory-Stalls zwischen den beiden Implementierungen. Der transaktionale Speicher kann also seinen Vorteil bei den Memory-Stalls wie im Transactional Benchmark nicht mehr halten und damit sinkt auch sein Leistungsplus.

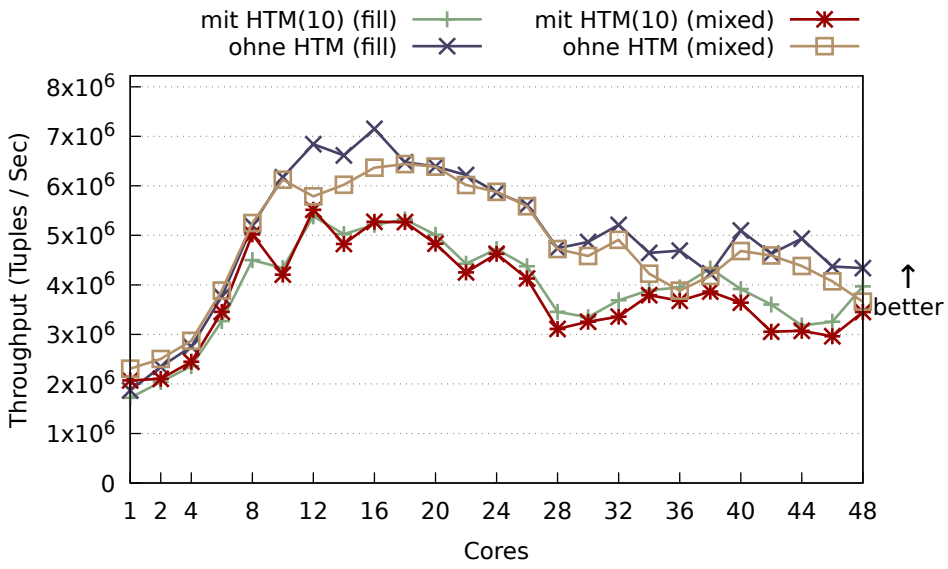

Abbildung 4.14: Durchsatz im BLinktree-Benchmark ohne Prefetching mit exklusivem Knotenzugriff

B$^{\text{link}}$-tree Benchmark ohne Prefetching mit exklusivem Knotenzugriff Wenn allerdings die Tasks immer auf die Kerne gescheduled werden, auf denen ihr Knoten allokiert wurde, so erzeugt dies deutlich höhere Last auf die Queues, da manche Knoten im Baum deutlich häufiger angefragt werden als andere. Gerade der Kern des Wurzelknotens wird dadurch häufiger besucht. Dies sieht man in Abbildung 4.14. Durch die höhere Last auf die Queues wird der Unterschied zwischen den zwei Phasen einer Iteration jeweils deutlich geringer. Dadurch, dass die Tasks in der *mixed*-Phase jeweils auf den gleichen Kern gescheduled werden, auf dem ihre korrespondieren Tasks in der *fill*-Phase ausgeführt wurden, bringt das auch jeweils ähnliche Task-Last auf die Queues.

In diesem Fall skalieren die Transaktionen deutlich schlechter als die atomaren Instruktionen. Aus Abbildung 4.15 ist durch die hohe Zahl abgebrochener Transaktionen ersichtlich, dass auf die Queues nun relativ hohe Last erzeugt wird. Dadurch, dass, wie oben beschrieben, einige Knoten im Baum deutlich häufiger als andere angefragt werden, dürften auf manchen Queues einige transaktionale Implementierungen auf ihre Fallback-Strategien zurückgreifen. Dazu muss jeweils der Queue-weite Latch `currently_writing` angefordert werden, was darum wieder andere Transaktionen abbricht.

Durch den zusätzlichen Speicheroverhead und die Tatsache, dass eine einfache Transaktion langsamer als ein atomares `xchg` ist, wird darum mit der HTM-Implementierung ein deutlich geringerer Durchsatz erzielt als ohne HTM. Durch unglückliches Locking oder durch gleichzeitigen Memory-Stall beider Threads kann auf einem CPU-Kern trotzdem eine Situation entstehen, in der atomare Instruktionen schlechter als HTM skalieren. Dies sieht man insbesondere in Abbildung 4.14 bei 38 Kernen. Dort überschreitet die HTM-

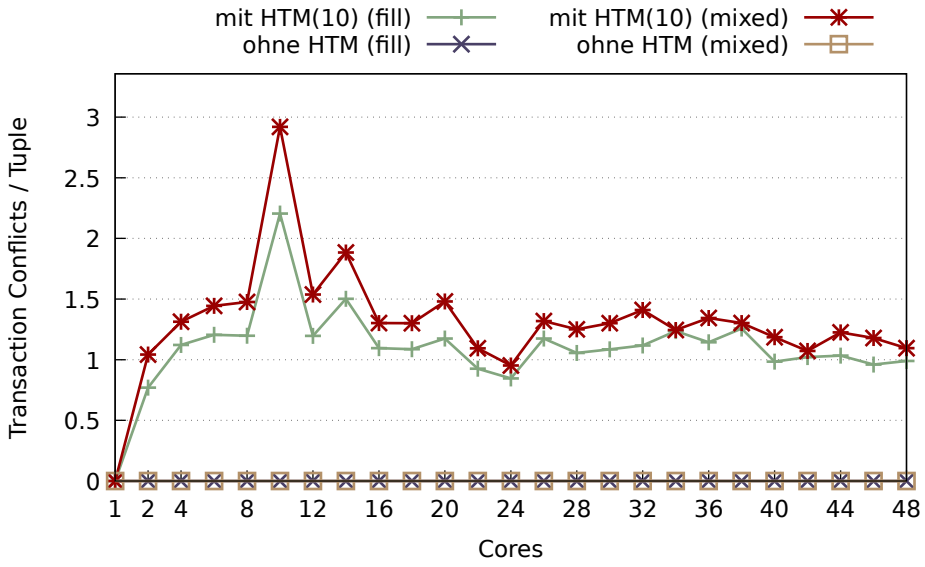

Abbildung 4.15: HTM Konflikte im B^{link}-tree Benchmark ohne Prefetching mit exklusivem Knotenzugriff

Implementierung (zumindest in der *fill*-Phase) den Durchsatz der Implementierung ohne HTM.

B^{link}-tree Benchmark mit Prefetch-Distanz 2 ohne exklusiven Knotenzugriff Nun wird der Fall betrachtet, dass in der Konfiguration ohne exklusiven Knotenzugriff jeweils zwei Tasks im Voraus in die Caches geladen werden. Das Prefetching beschleunigt den gesamten Benchmark, da damit nicht mehr so lange auf Speicher gewartet werden muss, nachdem ein Task abgearbeitet ist.

Die Abbildungen 4.16 und 4.17 ähneln jeweils sehr den Abbildungen 4.11 und 4.12. Durch den schnelleren Durchlauf wird zwar mehr Durchsatz erzielt, aber dadurch, dass der Benchmark in allen Konfigurationen um gleich viel schneller geworden ist, ändert sich das Konfliktverhalten nur wenig bis gar nicht. Die beiden Implementierungen sind darum jeweils wieder ungefähr gleich schnell.

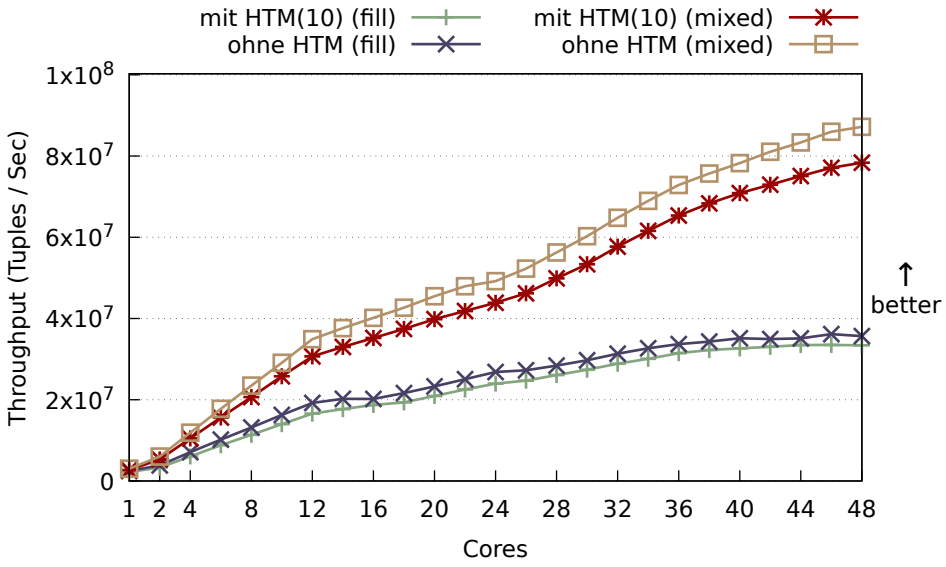

Abbildung 4.16: Durchsatz im B^{link}-tree Benchmark mit Prefetch-Distanz 2 ohne exklusiven Knotenzugriff

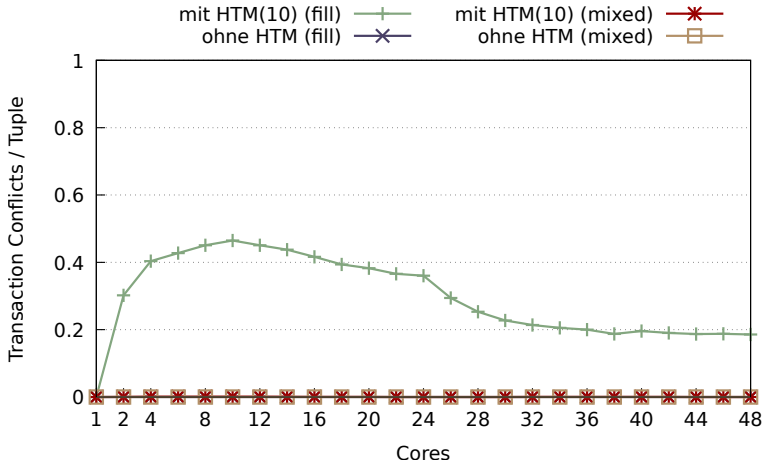

Abbildung 4.17: HTM Konflikte im B^{link}-tree Benchmark mit Prefetch-Distanz 2 ohne exklusiven Knotenzugriff

Kapitel 5

Fazit

Transactional Memory ist neben Latch-basierten Methoden und atomaren Instruktionen eine weitere Technik zur Thread-Synchronisation. Durch die Beschaffenheit der Transaktionen wird Synchronisation von parallelem Code deutlich einfacher, da innerhalb einer Transaktion kein Code zum gegenseitigen Ausschluss von verschiedenen Threads geschrieben werden muss. Trotzdem muss der Programmierer Latch-basierte Methoden kennen und anwenden können, da der Fallback-Handler einer transaktionalen Implementierung oftmals wieder auf Latches zurückgreifen muss.

Aus den Ergebnissen von Kapitel 4 ergibt sich, dass transaktionaler Speicher dabei im Gegensatz zu anderen Synchronisationstechniken auch relativ gut skalieren kann. Gerade bei Problemen mit nicht allzu hoher, aber trotzdem beträchtlicher Synchronisationslast erzielt transaktionaler Speicher einen ziemlich hohen Durchsatz. Auf Multiprozessorsystemen wird er effizienter, da er über das Cache-Kohärenz-Protokoll und nicht über den Speicher synchronisieren muss, denn Speicher wird bei Multiprozessorsystemen oftmals zum Flaschenhals für den Durchsatz der CPUs.

Auch fällt auf, dass transaktionaler Speicher besser skaliert, wenn wenig Arbeit zwischen verschiedenen Synchronisationsvorgängen verrichtet wird. Dadurch, dass der Transactional Benchmark in den Tasks praktisch keine Arbeit verrichtet, bleiben die globalen Objekte und der Queue-weite Latch großteils im L1-Cache jedes ausführenden CPU-Kerns erhalten. Beim B^{link}-tree Benchmark hingegen führen die Tasks teilweise einige aufwändige Speicheroperationen aus, die dann auch wiederum Daten in den Caches verändern. Beim transaktionalen Speicher wird aber sehr häufig der Latch (zumindest) gelesen. Wenn dieser durch viele andere Speicheroperationen aus dem L1-Cache verdrängt wird, schafft dies damit für diese Synchronisationstechnik einen deutlichen Leistungseinbruch.

Damit ist transaktionaler Speicher vor allem dann effizient, wenn sein Verwaltungsaufwand durch Synchronisationskosten versteckt wird, gleichzeitig aber auch nicht zu viel neuer Verwaltungsaufwand durch teure Speicheroperationen erzeugt wird.

Es gibt also Anwendungsfälle, für die transaktionaler Speicher im `MxTasking` effizient wird, allerdings ist gerade bei speicherintensiven Programmen die Implementierung mit atomaren Instruktionen effizienter. Da diese Arbeit im Datenbankkontext betrachtet wird, der damit sehr daten- und speicherintensiv ist, ist darum die Implementierung mit atomaren Instruktionen vorzuziehen.

Kapitel 6

Ausblick

Im B^{link}-tree Benchmark wird aktuell über die Implementierung der "Workload" noch eine Latch-basierte Synchronisation für alle Worker-Threads erforderlich, die keine Tasks mehr in ihrer Queue haben. Darum kann dort auch, durch unglückliches Scheduling bedingt, teilweise nur der Overhead von Latch-basierter Synchronisation gemessen werden. Durch eine transaktionale Synchronisation der Workload könnten damit die Ergebnisse für diesen Benchmark wiederum anders aussehen.

Beim Transactional Benchmark wird momentan ein globales Objekt an die Tasks übergeben, um die Verteilung von Tasks zu CPU-Kernen aufzuzeichnen und ihnen Parameter zu übergeben. Wenn ein Task nun so erstellt wird, dass er auf keine zusätzlichen Objekte mehr zugreifen muss, wird gerade auf einer zweiten CPU deutlich weniger Speicherlatenz zu registrieren sein. Damit würde die Last auf die Scheduling-Queues noch einmal erhöht und damit kann der Overhead der Synchronisation beider Implementierungen besser verglichen werden.

Das Linux-System unterhalb des `MxTasking` erzeugt zwar während der Messungen eher wenig Last auf die CPU-Kerne, allerdings ist zu erwarten, dass durch die Natur des `MxKernel` ohne die Abstraktion des `MxTasking` noch einmal weniger Verwaltungsaufwand entsteht. Somit könnten die Queues insgesamt anders belastet werden. Darum ist fraglich, inwiefern die in dieser Arbeit gemessene Leistung in Relation zur Leistung auf der *bare-metal*-Umgebung steht. Zu erwarten ist, dass zumindest die Proportionen der Werte ungefähr gleich bleiben, allerdings könnte zum Beispiel weniger Last auf die Caches auftreten und dies darum ein Leistungsplus für den transaktionalen Speicher bedeuten.

Transaktionaler Speicher kann also im `MxKernel` für manche Aufgaben sehr gut eingesetzt werden, kann aber durch eine effizientere Implementierung vielleicht noch einige Anwendungsfälle mehr abdecken, in denen durch diese Technik eine Leistungssteigerung erzielt werden kann.

Anhang A

Microbenchmarks

Für kleine Leistungsmessungen wurden im Rahmen dieser Arbeit einige Microbenchmarks durchgeführt. Diese nutzen die Google-Benchmark-Bibliothek [3].

Mit Hilfe dieser Bibliothek wird der Code für einen Benchmark direkt von der Bibliothek generiert. Implementiert werden müssen lediglich die Methoden, die vom Benchmark gemessen werden sollen. Zusätzlich bringt der generierte Benchmark einige Kommandozeilenparameter mit, um zum Beispiel mehrere Iterationen einer Messung durchführen zu können. Für die folgenden Benchmarks wurde zum Beispiel der Parameter `--benchmark_repetitions=5` angegeben. Damit wird jede Messung 5 Mal durchgeführt. Bei einer Iterationszahl größer als 1 berechnet der Benchmark für jede gemessene Methode automatisch den Mittelwert, den Median und die Standardabweichung der gemessenen Werte. Zusätzlich wird hier die Option `--benchmark_min_time=20` genutzt, damit der Benchmark länger läuft und die Ergebnisse damit stabiler und besser vergleichbar werden.

A.1 Test des Overheads einer einfachen Transaktion

In diesem Benchmark wurde zuerst getestet, wie viel Overhead verschiedene Synchronisationstechniken bei einem einfachen Schreibvorgang erzeugen. Dafür wurde die Google-Benchmark-Bibliothek, sowie das Programm auch, in der *DEBUG*-Konfiguration gebaut. In der *RELEASE*-Konfiguration werden einige Werte direkt mit 0 ns angegeben, was keinen Vergleich zwischen den verschiedenen Synchronisationstechniken erlaubt.

Die Benchmark-Methoden sind in Algorithmus A.1 angegeben. In diesen Methoden wird jeweils einer Variablen eine 1 addiert und dann wird sie modulo 2 gerechnet. Dies passiert ohne Synchronisation, mit atomaren Instruktionen, mit einer atomaren Variablen und mit transaktionalem Speicher. Damit die Berechnung in jedem Fall korrekt ausgeführt wird, ist hierbei für transaktionalen Speicher als Fallback-Pfad der gleiche Code wie innerhalb der Transaktion angegeben. Um hier aber nur den Overhead einer Transaktion und nicht

```
static void BM_UInt(benchmark::State &state) {
    std::uint32_t i{0};
    for (auto _ : state) {
        i = (i + 1) % 2;
    }
}
static void BM_Atomic_UInt(benchmark::State &state) {
    std::atomic_uint32_t i{0};
    for (auto _ : state) {
        i = (i + 1) % 2;
    }
}
static void BM_Transactional_UInt(benchmark::State &state) {
    std::uint32_t i{0};
    for (auto _ : state) {
        if (_xbegin() == _XBEGIN_STARTED) {
            i = (i + 1) % 2;
            _xend();
        } else {
            i = (i + 1) % 2;
        }
    }
}
static void BM_Atomic_Exchange_UInt(benchmark::State &state) {
    std::uint32_t i{0};
    for (auto _ : state) {
        __atomic_exchange_n(&i, (i+1)%2, __ATOMIC_SEQ_CST);
    }
}
```

Algorithmus A.1: Microbenchmark-Methoden für den Test, wie viel Verwaltungsoverhead verschiedene Synchronisationstechniken im Gegensatz zu Code ohne Synchronisation generieren

noch eines zusätzlichen Latches zu messen, ist im Fallback-Pfad auf die Anforderung eines Latches verzichtet worden.

Bei den Ergebnissen in Tabelle A.1 wird deutlich, wie viel Verwaltungsaufwand eine einzige Transaktion im Gegensatz zu einer atomaren Operation erzeugt. Die Transaktion benötigt hier mehr als die doppelte Zeit einer atomaren Instruktion.

Benchmark	Time	CPU	Iterations
BM_UInt	2.70 ns	2.70 ns	1000000000
BM_UInt	2.70 ns	2.70 ns	1000000000
BM_UInt	2.70 ns	2.70 ns	1000000000
BM_UInt	2.70 ns	2.70 ns	1000000000
BM_UInt	2.70 ns	2.70 ns	1000000000
BM_UInt_mean	2.70 ns	2.70 ns	5
BM_UInt_median	2.70 ns	2.70 ns	5
BM_UInt_stddev	0.000 ns	0.000 ns	5
BM_Atomic_UInt	16.1 ns	16.1 ns	1000000000
BM_Atomic_UInt	16.1 ns	16.1 ns	1000000000
BM_Atomic_UInt	16.2 ns	16.2 ns	1000000000
BM_Atomic_UInt	16.2 ns	16.2 ns	1000000000
BM_Atomic_UInt	16.2 ns	16.2 ns	1000000000
BM_Atomic_UInt_mean	16.2 ns	16.2 ns	5
BM_Atomic_UInt_median	16.2 ns	16.2 ns	5
BM_Atomic_UInt_stddev	0.015 ns	0.015 ns	5
BM_Transactional_UInt	21.1 ns	21.1 ns	1000000000
BM_Transactional_UInt	21.1 ns	21.1 ns	1000000000
BM_Transactional_UInt	21.1 ns	21.1 ns	1000000000
BM_Transactional_UInt	21.1 ns	21.1 ns	1000000000
BM_Transactional_UInt	21.1 ns	21.1 ns	1000000000
BM_Transactional_UInt_mean	21.1 ns	21.1 ns	5
BM_Transactional_UInt_median	21.1 ns	21.1 ns	5
BM_Transactional_UInt_stddev	0.001 ns	0.001 ns	5

Benchmark	Time	CPU	Iterations
BM_Atomic_Exchange_UInt	9.49 ns	9.49 ns	1000000000
BM_Atomic_Exchange_UInt	9.47 ns	9.47 ns	1000000000
BM_Atomic_Exchange_UInt	9.47 ns	9.47 ns	1000000000
BM_Atomic_Exchange_UInt	9.47 ns	9.47 ns	1000000000
BM_Atomic_Exchange_UInt	9.47 ns	9.47 ns	1000000000
BM_Atomic_Exchange_UInt_mean	9.47 ns	9.47 ns	5
BM_Atomic_Exchange_UInt_median	9.47 ns	9.47 ns	5
BM_Atomic_Exchange_UInt_stddev	0.009 ns	0.009 ns	5

Tabelle A.1: Overhead verschiedener Synchronisationstechniken im Gegensatz zu nicht synchronisiertem Code bei einer einfachen Berechnung

A.2 Test der Wartestrategien zwischen Transaktionen

In diesem Benchmark wird getestet, welche die beste Wartestrategie zwischen dem Abbruch und dem Neustart einer Transaktion ist. Dazu wurden in der Klasse der MPSC-Queue mehrere verschiedene `push_back`-Methoden implementiert, die jeweils eine eigene Queue im Benchmark instantiiert bekommen. Der Code dieser Methoden unterscheidet sich nur im **else**-Block zum **if**, in dem das `_xbegin()` aufgerufen wird.

BM_Queue_x_NOP Bei den Benchmarks `BM_Queue_1_NOP` bis `BM_Queue_3_NOP` wird der folgende Code genutzt:

```
for (auto var = 0u; var < iteration_count; ++var) {
    __asm__ volatile ("nop");
    //additional nops for the other benchmarks here
}
```

BM_Queue_Without_Loop_x_NOP Bei den Benchmarks `BM_Queue_Without_Loop_1_NOP` bis `BM_Queue_Without_Loop_3_NOP` wird beim oben angegebenen Code einfach die Schleife weggelassen:

```
__asm__ volatile ("nop");
//additional nops for other benchmarks here
```

BM_Queue_Spinlock Bei dieser Strategie wird in einer Schleife darauf gewartet, dass der Queue-weite Latch `currently_writing` frei wird:

```
while (currently_writing) {
    __asm__ volatile ("pause");
}
```

BM_Queue_Without_Wait Schließlich wird hier eine Strategie getestet, bei der kein Warten stattfindet.

A.2.1 Benchmark Wrapper-Code

Damit die Queues richtig instantiiert und verwaltet werden, wird der folgende Wrapper-Code für die verschiedenen Benchmarks genutzt:

```
static void Setup_Queue(MPSCQueue **queue) {
    *queue = new MPSCQueue();
}
static void Teardown_Queue(MPSCQueue **queue) {
    delete *queue;
}
static void Do_Benchmark(benchmark::State &state,
          MPSCQueue **queue,
          void (MPSCQueue::*push_back_method)(QueueItem *item)) {
    for (auto _ : state) {
        QueueItem queue_item{};
        (*queue->*push_back_method)(&queue_item);
    }
}
static void Setup_Benchmark(benchmark::State &state,
          MPSCQueue **queue,
          void (MPSCQueue::*push_back_method)(QueueItem *item)) {
    if (state.thread_index == 0) {
        Setup_Queue(queue);
    }
    Do_Benchmark(state, queue, push_back_method);
    if (state.thread_index == 0) {
        Teardown_Queue(queue);
    }
}
```

Algorithmus A.2: Verwaltungscode für den Test der Wartestrategien zwischen zwei Transaktionen

A.2.2 Ergebnisse des Benchmarks

Dieser Benchmark ist mit der Implementierung der ersten lauffähigen transaktionalen Queue evaluiert worden, um damit direkt richtig optimieren zu können und das Ergebnis dieses Tests nicht durch andere Optimierungen zu verfälschen.

Damit sind bei maximal 10 erlaubten Transaktionen die folgenden Werte gemessen worden:

Benchmark	Time	CPU	Iterations
BM_Queue_1_NOP/threads:48	1913 ns	76838 ns	613824
BM_Queue_1_NOP/threads:48	1814 ns	85114 ns	613824
BM_Queue_1_NOP/threads:48	1831 ns	79843 ns	613824
BM_Queue_1_NOP/threads:48	1808 ns	85401 ns	613824
BM_Queue_1_NOP/threads:48	1958 ns	88940 ns	613824
BM_Queue_1_NOP/threads:48_mean	1865 ns	83227 ns	5
BM_Queue_1_NOP/threads:48_median	1831 ns	85114 ns	5
BM_Queue_1_NOP/threads:48_stddev	66.9 ns	4827 ns	5
BM_Queue_2_NOP/threads:48	2115 ns	84971 ns	302832
BM_Queue_2_NOP/threads:48	1837 ns	82052 ns	302832
BM_Queue_2_NOP/threads:48	1674 ns	74568 ns	302832
BM_Queue_2_NOP/threads:48	1782 ns	73182 ns	302832
BM_Queue_2_NOP/threads:48	1969 ns	81974 ns	302832
BM_Queue_2_NOP/threads:48_mean	1875 ns	79349 ns	5
BM_Queue_2_NOP/threads:48_median	1837 ns	81974 ns	5
BM_Queue_2_NOP/threads:48_stddev	171 ns	5165 ns	5
BM_Queue_3_NOP/threads:48	1953 ns	76124 ns	307296
BM_Queue_3_NOP/threads:48	1500 ns	52395 ns	307296
BM_Queue_3_NOP/threads:48	1878 ns	81260 ns	307296
BM_Queue_3_NOP/threads:48	1524 ns	63441 ns	307296
BM_Queue_3_NOP/threads:48	1900 ns	80614 ns	307296
BM_Queue_3_NOP/threads:48_mean	1751 ns	70767 ns	5
BM_Queue_3_NOP/threads:48_median	1878 ns	76124 ns	5
BM_Queue_3_NOP/threads:48_stddev	220 ns	12520 ns	5

Benchmark	Time	CPU	Iterations
BM_Queue_Without_Loop_1_NOP/threads:48	1969 ns	76104 ns	383520
BM_Queue_Without_Loop_1_NOP/threads:48	1751 ns	78088 ns	383520
BM_Queue_Without_Loop_1_NOP/threads:48	1898 ns	87238 ns	383520
BM_Queue_Without_Loop_1_NOP/threads:48	1732 ns	68052 ns	383520
BM_Queue_Without_Loop_1_NOP/threads:48	1845 ns	73226 ns	383520
BM_Queue_Without_Loop_1_NOP/threads:48_mean	1839 ns	76542 ns	5
BM_Queue_Without_Loop_1_NOP/threads:48_median	1845 ns	76104 ns	5
BM_Queue_Without_Loop_1_NOP/threads:48_stddev	99.7 ns	7072 ns	5

BM_Queue_Without_Loop_2_NOP/threads:48	1824 ns	77138 ns	593424
BM_Queue_Without_Loop_2_NOP/threads:48	1759 ns	82291 ns	593424
BM_Queue_Without_Loop_2_NOP/threads:48	1664 ns	75570 ns	593424
BM_Queue_Without_Loop_2_NOP/threads:48	1924 ns	85275 ns	593424
BM_Queue_Without_Loop_2_NOP/threads:48	1763 ns	81557 ns	593424
BM_Queue_Without_Loop_2_NOP/threads:48_mean	1787 ns	80366 ns	5
BM_Queue_Without_Loop_2_NOP/threads:48_median	1763 ns	81557 ns	5
BM_Queue_Without_Loop_2_NOP/threads:48_stddev	95.9 ns	3957 ns	5

BM_Queue_Without_Loop_3_NOP/threads:48	1944 ns	80100 ns	389472
BM_Queue_Without_Loop_3_NOP/threads:48	2305 ns	90361 ns	389472
BM_Queue_Without_Loop_3_NOP/threads:48	1638 ns	66118 ns	389472
BM_Queue_Without_Loop_3_NOP/threads:48	1753 ns	73398 ns	389472
BM_Queue_Without_Loop_3_NOP/threads:48	1801 ns	82232 ns	389472
BM_Queue_Without_Loop_3_NOP/threads:48_mean	1888 ns	78442 ns	5
BM_Queue_Without_Loop_3_NOP/threads:48_median	1801 ns	80100 ns	5
BM_Queue_Without_Loop_3_NOP/threads:48_stddev	258 ns	9172 ns	5

BM_Queue_Spinlock/threads:48	897 ns	39953 ns	913872
BM_Queue_Spinlock/threads:48	868 ns	37589 ns	913872
BM_Queue_Spinlock/threads:48	849 ns	38175 ns	913872
BM_Queue_Spinlock/threads:48	912 ns	39511 ns	913872
BM_Queue_Spinlock/threads:48	1196 ns	50925 ns	913872
BM_Queue_Spinlock/threads:48_mean	945 ns	41231 ns	5
BM_Queue_Spinlock/threads:48_median	897 ns	39511 ns	5
BM_Queue_Spinlock/threads:48_stddev	143 ns	5504 ns	5

Benchmark	Time	CPU	Iterations
BM_Queue_Without_Wait/threads:48	1720 ns	77506 ns	480000
BM_Queue_Without_Wait/threads:48	1870 ns	80747 ns	480000
BM_Queue_Without_Wait/threads:48	1712 ns	81366 ns	480000
BM_Queue_Without_Wait/threads:48	1873 ns	78268 ns	480000
BM_Queue_Without_Wait/threads:48	1639 ns	74903 ns	480000
BM_Queue_Without_Wait/threads:48_mean	1763 ns	78558 ns	5
BM_Queue_Without_Wait/threads:48_median	1720 ns	78268 ns	5
BM_Queue_Without_Wait/threads:48_stddev	104 ns	2609 ns	5

Tabelle A.2: Ergebnisse der verschiedenen Wartestrategien zwischen dem Abbruch und dem Neustart einer Transaktion bei maximal 10 transaktionalen Versuchen

Bei diesen Ergebnissen fällt auf, dass die Implementierung des Benchmarks `BM_Queue_Spinlock` bei Weitem die besten Ergebnisse erzielt. Darum wird in der MPSC-Queue des `MxTasking` diese Implementierung genutzt.

Abbildungsverzeichnis

Algorithmenverzeichnis

Literatur

[1] *10^{TH} GEN INTEL® CORE™ DESKTOP PROCESSORS*. Intel Corporation. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/10th-gen-core-desktop-brief.pdf (besucht am 04. 08. 2020).

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*. Intel Corporation. Mai 2020.

[3] *Benchmark*. Google. 21. Feb. 2020. URL: https://github.com/google/benchmark (besucht am 14. 08. 2020).

[4] Félix Cloutier. *CMPXCHG — Compare and Exchange*. Mai 2019. URL: https://www.felixcloutier.com/x86/cmpxchg (besucht am 18. 07. 2020).

[5] Félix Cloutier. *LOCK — Assert LOCK# Signal Prefix*. Mai 2019. URL: https://www.felixcloutier.com/x86/lock (besucht am 04. 08. 2020).

[6] Félix Cloutier. *PAUSE — Spin Loop Hint*. Mai 2019. URL: https://www.felixcloutier.com/x86/pause (besucht am 05. 08. 2020).

[7] Félix Cloutier. *XCHG — Exchange Register/Memory with Register*. Mai 2019. URL: https://www.felixcloutier.com/x86/xchg (besucht am 07. 08. 2020).

[8] Brian F. Cooper. *Yahoo! Cloud Serving Benchmark (YCSB)*. URL: https://github.com/brianfrankcooper/YCSB/wiki (besucht am 22. 07. 2020).

[9] *Cutter. Free and Open Source RE Platform powered by radare2*. URL: https://cutter.re/ (besucht am 24. 07. 2020).

[10] *DDR4 White Paper*. Corsair®. URL: https://images-eu.ssl-images-amazon.com/images/I/91LwsXv8xUS.pdf (besucht am 04. 08. 2020).

[11] Roman Dementiev. *Exploring Intel® Transactional Synchronization Extensions with Intel® Software Development Emulator*. 6. Nov. 2012. URL: https://software.intel.com/content/www/us/en/develop/blogs/exploring-intel-transactional-synchronization-extensions-with-intel-software.html (besucht am 01. 08. 2020).

[12] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007.

[13] Daniel Hackenberg, Daniel Molka und Wolfgang Nagel. "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems". In: Jan. 2009, S. 413–422. DOI: 10.1145/1669112.1669165.

[14] Theo Haerder und Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Computing Surveys* 15 (1983), S. 287–317.

[15] Maurice Herlihy und J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *SIGARCH Comput. Archit. News* 21.2 (Mai 1993), S. 289–300. ISSN: 0163-5964. DOI: 10.1145/173682.165164. URL: https://doi.org/10.1145/173682.165164.

[16] Maurice Herlihy und Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916.

[17] Philip L. Lehman und S. Bing Yao. "Efficient Locking for Concurrent Operations on B-trees". In: *ACM Transactions on Database Systems (TODS)* 6.4 (1981), S. 650–670. ISSN: 15574644. DOI: 10.1145/319628.319663.

[18] V. Leis, A. Kemper und T. Neumann. "Exploiting hardware transactional memory in main-memory databases". In: *2014 IEEE 30th International Conference on Data Engineering*. März 2014, S. 580–591. DOI: 10.1109/ICDE.2014.6816683.

[19] Deborah T. Marr und Frank Binns. "Hyper-Threading Technology Architecture and Microarchitecture". In: 2002.

[20] D. Molka u. a. "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture". In: *2015 44th International Conference on Parallel Processing*. 2015, S. 739–748.

[21] Michael Mueller. "MxKernel: Rethinking Operating System Architecture for Many-core Hardware". In: 2019.

[22] *Restricted Transactional Memory Overview*. Intel Corporation. URL: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intrinsics-for-restricted-transactional-memory-operations/restricted-transactional-memory-overview.html (besucht am 17. 06. 2020).

[23] *Intel® Transactional Synchronization Extensions (Intel® TSX) Programming Considerations*. URL: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-

tsx/intel-transactional-synchronization-extensions-intel-tsx-programming-considerations.html (besucht am 16. 07. 2020).

[24] Dmitry Vyukov. *Intrusive MPSC node-based queue*. URL: http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue (besucht am 16. 07. 2020).

[25] Dmitry Vyukov. *Producer-Consumer Queues*. URL: http://www.1024cores.net/home/lock-free-algorithms/queues (besucht am 05. 08. 2020).

[26] Richard M. Yoo u. a. "Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503232. URL: https://doi.org/10.1145/2503210.2503232.