Analyse von Synchronisation auf heutigen Prozessoren

Michael Panzlaff
Universität Erlangen-Nürnberg
michael.panzlaff@fau.de

ÜBERBLICK

Vielkernprozessoren spielen heutzutage eine immer bedeutender werdene Rolle in Computern. Die Entwicklung schneller einzelner Prozessorkerne ist zwar nicht komplett zu Ende, aber dennoch werden sie früher oder später an die physikalischen Grenzen stoßen. Die Migration auf moderne Vielkernarchitekturen ist daher unausweichlich um weiterhin den steigenden Leistungsansprüchen zu genügen. Diese Ausarbeitung, die sich insbesondere auf „Everything You Always Wanted To Know About Snychronization but Were Afraid to Ask“ [4] bezieht, soll daher einen kleinen Überblick über mehrere Synchronisationsarten und Architekturen verschaffen um skalierbare Software für Vielkernprozessoren zu schreiben.

1. EINFÜHRUNG

Um die volle Leistung heutiger Prozessoren nutzen zu können, ist die Parallelisierung von Programmcode unumgänglich. Es ist erforderlich, dass künftige Software auch im besten Fall mit einer großer Anzahl an Kernen skaliert. Die Herrausforderung darin besteht also nicht nur darin den sequentiellen Code zu parallelisieren und wettlaufsicher zu gestalten, sondern auch daraus dafür zu sorgen, dass der parallele Anteil möglichst groß ist (*Amdahl's Gesetz*) [8]. Amdahl's Gesetz sagt, dass für einen möglichst günstigen Geschwindigkeitsgewinn der parallele Anteil steigen muss, je mehr Prozessorkerne einem zu Verfügung stehen. Dies stellt eine gewisse Schwierigkeit dar, da zum Beispiel ein gewöhnliches Lock den sequentiellen Anteil der Laufzeit erhöht.

Um bestmögliche Leistung auf gegebener Hardware zu erzielen, ist es daher interessant für den gewünschten Anwendungszweck die am besten geeignete Synchronisationsart zu wählen. Als Überblick soll Tabelle 1 dienen.

Table 1: Synchronisationsebenen [4, S. 33]

Ebene	Beispiele
Nebenläufige Software	Streutabelle, STM
Primitive	Locks, Message Passing
Atomare Operationen	CAS, FAI, TAS, SWAP
Cache Kohärenz	Laden, Speichern

Auf tiefster Ebene findet sich die Cache Kohärenz. Diese ist relevant, weil beim konventionellen nebenläufigen Programmieren Daten über den Speicher ausgetauscht werden. Da jedoch bei modernen Prozessoren zwischen dem Speicher nahezu immer über einen Cache gepuffert wird, muss unter den Caches der jeweiligen Prozessorkerne für Kosistenz und Aktualität gesorgt werden, damit die ausgetauschten Daten auch ankommen und erfolgreiche Synchronisation möglich wird. Daher gibt es u. a. das Kohärenz-Protokoll MESI [3]. Dadurch ergibt sich zwangsweise Latenz, wenn mehrere Prozessorkerne versuchen auf gleichen Speicheraddressen zu lesen und zu schreiben. Die Auswirkungen davon werden später in der Analyse gezeigt.

Damit Anwendungen nicht nur korrekt ihre Daten austauschen, sondern auch wettlauffrei sind, gibt es auf heutigen Prozessoren Instruktionen, die atomare Operationen auf dem Speicher ermöglichen. Verbreitete bekannte Beispiele davon sind „Compare and Swap“ (*CAS*), „Fetch and Increment“ (*FAI*), „TAS“ (*Test and Set*) und SWAP (*Atomic Swap*). Mit diesen relativ einfachen Synchronisationsmechanismen lassen sich somit Programmierprimitive bauen, die man dann z. B. in C als Lock verwenden kann.

Als letztes gibt es noch die Anwendung. Diese legt fest, nach welchem Muster die Primitive genutzt werden um threadsichere Kommunikation zu ermöglichen. Ein Beispiel dafür ist z. B. eine Streutabelle, die u. a. in der Hauptreferenzarbeit für Messungen verwendet wurde.

Möchte man eine Anwendung bestmöglich optimieren, ist es natürlich nicht trivial zu ermitteln auf welcher Ebene sich der Flaschenhals befindet. gebnissen zu Verfügung, welche vorher in der Regel nur für spezielle Bedingungen zusammengfasst wurde (z. B. Linux Kernel [1]).

2. TESTSYSTEME

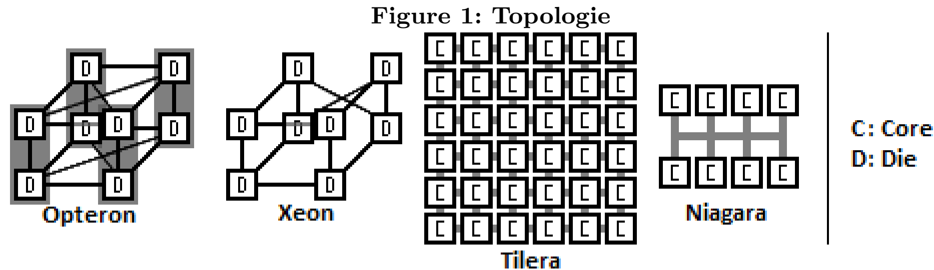
Beim Analysieren sind natürlich die verwendeten Testsysteme sehr wichtig, um die verschiedenen Prozessoren und Architekturen sinnvoll vergleichen zu können. In der Hauptreferenzarbeit wurden daher 4 veschiedene Testsysteme verwendet, die sich in ihrer Topologie unterscheiden (siehe Tabelle 2, vereinfacht).

Sowohl das AMD Opteron als auch das Intel Xeon System zeichnt sich dadurch aus, dass es mehrere Prozessoren in einem System zu einem Verbund zusammenschließt. Wie auch in der Hauptreferenzarbeit wird hier zur Vereinfachung davon ausgegangen, dass sich die 2 *CPU-Dies* auf dem Opteron wie Prozessoren in 2 Sockeln verhalten. Sowohl beim Opteron als auch beim Xeon System besitzen die Prozessoren unabhängige Endstufen Caches (*LLC*) sowie deren angebunden Speicher. Speicherzugriff auf einen anderen Prozessor erfolgt sowohl beim Opteron als auch beim Xeon über höchstens 2 Knoten bei den jeweils benchbarten Prozessoren in der Topologie (s. 1). Hier sollte man beachten, dass mehrere Sockel unter Umständen ein Problem sind. Verbunde über mehrere Prozessoren wirken sich nämlich negativ auf

Table 2: Testsysteme [4, S. 35]

Name	Opteron	Xeon	Niagara	Tilera
Prozessoren	4x AMD Opteron 6172	8x Intel Xeon E7-8867L	SUN UltraSPARC-T2	TILE-Gx36 CPU
Anzahl Kerne	48	80 (kein Hyper-Threading)	8 (64 Threads)	36
Takt	2.1 GHz	2.13 GHz	1.2 GHz	1.2 GHz
L1 Cache (I/D)	64/64 KiB	32/32 KiB	16/8 KiB	32/32 KiB
L2 Cache	512 KiB	256 KiB	-	256 KiB
Last Level Cache	2x6 MiB (geteilt)	20 MiB (geteilt)	4 MiB (geteilt)	9 MiB (verteilt)
Speicher (Knoten)	128 GiB (8)	192 GiB (8)	32 GiB (1)	16 GiB (2)

Figure 1: Topologie



[4, S. 36] [10] [13]

die Cache-Latenz und auch auf die Kosten atomarer Operationen aus[4, S. 34]. Bei der Auswertung der Messergebnisse wird noch genauer darauf eingegangen, in welchen Szenarien die Verzögerung besonders zum Problem wird. Ähnlich zu dieser Topologie ist das Tilera System, welches zwar nicht mit mehreren Prozessoren über Sockel verteilt ist, aber bei dem jeder Kern einen Teil des LLCs besitzt und somit andere Kerne auch ggf. Umwege über andere Kerne machen müssen um Cache Kohärenz und Atomarität zu gewährleisten (*nicht-uniform*). Das Niagara System ist in diesem Aspekt etwas anders, da es ledeglich ein Sockel hat und zudem alle Kerne sich den gleichen LLC teilen (*uniform*). Somit fällt beim Niagara zumindestens ein Teil der durch Cache Kohärenz verursachten Latenz weg. So ist beispielsweise die Leistung für dichte Zugriffsmuster bei atomaren Operationen ungefähr 1.7 mal schneller [4, S. 34].

3. LEISTUNGSMESSUNG

Sämtliche Tests wurden mit der SSYNC Synchronisations Suite durchgeführt. Diese ermöglicht das Leistungsmessen der vorgestellten Synchronisationsebenen 1.

3.1 Cache - ccbench

„ccbench“ ist ein Tool, welches einem ermöglicht die Zugriffskosten auf den Cache in Abhängigkeit der Cachezeilen-Zustände zu messen [4, S. 34]. Es unterstützt 30 verschiedene MESI-Zustandskombinationen.

3.2 Locks - libslock

„libslock“ bietet eine Schnittstelle an um 9 verschiedene Lock-Arten zu verwenden. Es untersützt folgende:

- **TAS**: Implementiert mithilfe einer *TAS* Instruktion des Prozessors.
- **TTAS**: Ähnlich zu *TAS*, nur dass hier erst auf die Verfügbarkeit des Locks ohne atomare Operation gewartet wird.
- **Ticket Lock**: Beachtet die Reihenfolge (*FIFO*).
- **MCS Lock**: Benannt nach seinen Erfindern [9]. Variante eines Ticket Locks mit Warteschleife.
- **CLH Lock**: Weiteres Lock, welches mit einer Warteschleife implementiert ist. Die Hauptreferenzarbeit verweist auf [2], Namensgebung ist unklar.
- ***Array* basiertes Lock**: Wird mithilfe eines Feldes implementiert. Details in [7, S. 150].
- **Hierarchisches CLH Lock**: Erweiterung des CLH Locks um Hierarchie. Solch ein Lock kann nur dann aquiriert werden, wenn es in der Hierarchie niedriger liegt, als alle die bereits von einem bestimmten Aktor aquiriert worden sind.
- **Hierarchisches Ticket Lock**: Ähnlich zum oberem nur mit einem Ticket Lock.
- **Mutex**: Ein Lock, welches mithilfe eines *pthread-Mutex* implementiert ist.

3.3 Nachrichten - libssmp

„libssmp“ ermöglicht nachrichtenbasierte Kommunikation in Software zu realisieren. Die Nachrichten werden dabei wie bei Barrelfish [11] an die Größe der einzelnen Cache Zeilen angepasst und können somit zwischen den Prozessorkernen ausgetauscht werden. Die Zeilen werden mit Blöcken gefüllt, die aus Daten und u. a. einem Flag bestehen, welches sagt ob es sich um Nachricht oder keine (eine leere) handelt [4, S. 37].

3.4 Anwendung - ssht

„ssht“ ist eine Streutabelle, die mit möglichst großer Effizienz die Daten im Speicher für größte Cache Effizienz ablegt. Ebenso kann *ssht* konfiguriert werden, ob es mithilfe von *libslock* oder *libssmp* arbeiten soll. Die Bibliothek bietet die Standardoperationen „set“, „get“ und „remove“ an [4, S. 37].

Table 3: Cache und Speicher Latenzen [4, S. 35]

	Opteron	Xeon	Niagara	Tilera
L1	3	5	3	2
L2	15	11	-	11
LLC	40	44	24	45
RAM	136	355	176	118

3.5 Anwendung - TM²C

„TM²C“ ist eine Implementierung eines transaktionalen Speichers in Software, der auf Nachrichtenaustausch basiert. Es arbeitet standardmäßig mit *libssmp*, kann aber auch mit *libslock* verwendet werden [4, S. 37]. Unterstütze Operationen werden nicht genannt.

4. MESSERGEBNISSE

4.1 Speicherzugriffe

Tabelle 3 zeigt die Zugriffszeiten auf die einzelnen Cache Ebenen im besten Fall. Die Anzahl der Takt-Zyklen für die Cache Zugriffe sind bei fast allen Prozessoren relativ ähnlich. Der Niagara ist im Vergleich zu den anderen bei LLC Zugriffen relativ schnell, besitzt allerdings auch nur 2 Cache Ebenen und im Gegensatz zum Tilera nur einen zentralen LLC. Beachten sollte man hier noch, dass die Prozessoren unterschiedlich getaktet sind und somit die Latenzen nicht direkt ablesbar sind. In der Hauptreferenzarbeit wird leider nichts zu den RAM Zugriffszeiten erwähnt. In anbetracht dessen, dass es bei atomaren Operationen primär um die Kohärenz der Caches geht, ist das auch nicht weiter tragisch.

Tabelle 2 [4, S. 38] zeigt die Messergebnisse der Latenzen von den vier wichtigen (CAS, TAS, FAI, SWAP) atomaren Operationen sowie auch zu regulären Lese- und Schreibvorgängen auf Cachezeilen in ihren jeweiligen Zuständen. Hierbei wird ein guter Überblick und Vergleich über die die Zugriffszeiten geschaffen. Es wird u. a. sehr gut gezeigt, welchen Einfluss die zwei *CPU-Dies* vom Opteron auf die Speicherzugriffe haben. Die Latenzen sind fast genau so hoch, wie von einem Sockel zum Nächsten. Ähnlich verhält sich auch der Xeon bei einem zusätzlichen Knoten. Insofern hat man nur sehr geringfügige Vorteile beim Opteron 12 Kerne auf einem Chip zu haben. Für einen zweiten Knoten[1] beim Speicherzugriff wird dann mit der ungefähr dreifachen Zufgriffszeit bezahlt.

Absolut herrausragen tut nur der Niagara Prozessor, der hier mit seinen uniformen Speicherzugriffen punkten kann und außer bei ungültigen Cachezeilen sehr schnell ist. Der Tilera ist im Vergleich zum Opteron und Xeon auch schneller. Hier bremst allerdings der verteilte LLC die Zugriffe in allen Cachezeilen-Zuständen aus. Beobachten kann man hier dennoch, dass alle Einheiten in Zyklen angegeben sind und der Niagara und Tilera fast nur halb so schnell takten wie der Opteron und Xeon. Interessant wäre zu wissen, ob die Latenzen der beiden noch genau so gut bei höheren Taktraten sind und somit ein wesentlich schnellerer Zugriff als auf dem Opteron und Xeon möglich wäre.

Überraschend ist, dass die atomaren Operationen auf dem Opteron und Xeon nur geringfügig langsamer sind wie reguläre Speicherzugriffe. Hier ist die Lücke noch deutlich größer beim Tilera und Niagara. Letztendlich werden zu den Messergebnissen viele Schlüsse gezogen, wie z. B. dass ein Schreibvorgang im Cache (für *owned* und *shared* Zeilen) generell lange dauert, da immer erst alle anderen Caches invalidiert werden müssen. warum diverse Operationen unter bestimmen Bedingungen langsam sind. Andeutungen für die effiziente Nutzungen auf höheren Ebenen werden noch keine gemacht.

4.2 Atomare Operationen

Graph 4 [4, S. 40] zeigt nun die Auswertung des Stresstests der atomaren Operationen. Getestet wurden die abgearbeitete Menge an Operationen, die parallel auf den selben Daten arbeiten. Interessante Infos über diese erkennt man bei allen Prozessoren:

- **Opteron**: Dieser zeigt eine sehr starke Leistung bei einem Thread. Diese fällt ab, sobald Nebenläufigkeit mit mehreren Threads eine Rolle spielt und bleibt auch konstant, solange alle Threads auf dem gleichen *CPU-Die* laufen. Sobald über mehr als einen Knoten auf den Speicher zugegriffen wird, sinkt die Leistung weiter und nimmt mit zunehmender Threadanzahl ab. Ein Vergleich mit einer anderen Messung [12, S. 9] zeigt etwas abweichende Ergebnisse [2]. Die Systeme sind nicht komplett identisch (hier nur 2x 8 Kern Opterons), die Prozessoren sind allerdings aus der gleichen Generation. Man könnte also ähnliches Verhalten erwarten. Der rapide Abfall mit mehr als einem Thread ist auch dort gut zu sehen, die „Stufe“ allerdings fehlt[3]. Der Abstieg wird mit zunehmender Anzahl an Threads allerdings auch hier bestätigt.

- **Xeon**: Hier verläuft der Graph relativ ähnlich zum Opteron. Sobald mehr Threads eingeplant werden, als auf einen Prozessor passen, macht die „Stufe“ ihren letzten Satz nach unten. Die Implementierung des *FAI* mittels *CAS* ist so wie auch bei allen anderen mit Abstand am langsamsten. Alle anderen atomaren Operationen verhalten sich in etwa gleich schnell. Die Werte dieses Xeons lassen sich leider nicht direkt mit denen aus [12, S. 9] vergleichen, da es sich um eine andere Generation handelt. Ebenso fehlt aber auch dort die „Stufe“ im Graphen.

- **Niagara**: Als Einzel-Sockel-System verhält dieses sich grundlegend anders für eine höhere Anzahl an Threads. Hier ist die Skalierung der Threads mit der Datenmenge wesentlich besser, aber erreicht für keine Threadanzahl die Leistung, die z. B. der Opteron oder der Xeon mit einem Thread erreicht. Hier ist dann ab 8 Threads nahezu das Maximum ausgereizt. Im Gegensatz zum Opteron und Xeon sollte hier auf die Wahl der besten atomaren Operation Wert gelegt werden.

- **Tilera**: Dieser verhält sich im Sinne des Durchsatzes mit Abstand am besten aller Architekturen. Es gibt

[1]Knoten im Sinne von Schritten, die zu benachbarten Prozessoren getätigt werden müssen

[2]Die Anzahl der Operationen ist zwar nicht direkt vergleichbar mit dem Datendurchsatz, sollte aber in etwa proportional dazu liegen.

[3]Allerdings wird in der referenzierten Arbeit auch nicht erwähnt, dass die Threads falls möglich auf einen *CPU-Die* eingeplant werden. Dies erklärt ggf. die fehlende „Stufe“.

zwar ein leichtes Absinken, hat aber schon für sechs Threads den Hochpunkt (eine Kern „Reihe" im Tilera Prozessor). Sowohl beim Tilera als auch beim Niagara findet sich wieder, dass die einzelnen atomaren Operationen unterschiedlich lange brauchen (Vgl. 4.1). Dies ist zumindest in gewisser Hinsicht etwas überraschend, da man bei der verteilten Struktur des Tilera Prozessors eher das Gegenteil erwarten würde. Zu beiden Einzel-Sockel-Systemen lassen sich nicht wirklich Vergleichsmesswerte finden.

5. LOCK LEISTUNG IM VERGLEICH

In [4, S. 42] werden die einzelnen atomaren Operationen für alle neun Lock Typen aus *libslock* getestet. Dies wird mit einer gleichverteilte Menge an Locks durchgeführt (1, 4, 16, 32, 128 und 512 Locks).

5.1 Konfliktsituation um ein Lock

Im Messgraph sieht man für den Opteron und Xeon wieder die „Stufe" sobald mehr Threads als Prozessorkerne auf einem *CPU-Die* versuchen ein Lock zu aquirieren. Ähnlich sinkt der Durchsatz der verschiedenen Lock Implementierungen mit steigender Threadanzahl. Interessant ist hier, dass für die Locks die unterschiedlichen Varianten jetzt stark variiernde Leistung relativ zueinander zeigen. So ist zum Beispiel beim Opteron und beim Xeon das *hticket*-Lock bei starkem Konflikt bis zu mehr als 3 mal so schnell ist wie ein *TAS*-Lock, von dem man zugrund seiner Einfachheit her erwarten würde, dass es nicht den letzten Platz belegt.

Auf den Einzel-Sockel-Systemen ist der Durchsatz wesentlich über denen der Multi-Sockel-Systemen. Sowohl beim Niagara als auch beim Tilera dominiert das CLH-Lock. Etwas Überraschend ist hier, dass zuvor *TAS* als mit eine der schnellsten Operationen beim Niagara und beim Tilera gemessen worden sind und dessen Lock Implementierungen nun zu den schlechtesten gehört.

5.2 Konfliktsituation um 512 Locks

Eine Messung auf eine breite Verteilung an Locks soll dazu dienen, eine etwas breitere Messung zu erhalten für typischere Konfliktszenarien. Unabhängig des Locktyps werden bei geringem Konflikt die Ergebnisse wesentlich konsistenter. Beim Opteron findet sich diesmal keine „Stufe" wieder und die Leistung stagniert interessanterweise erst ab eine Anzahl von rund 24 Threads.

Bei Konflikten auf einem einzigem Prozessor, zeichnet sich der Xeon mit doppelter Leistung bei fast allen Lock-Arten ab, sinkt aber wieder, sobald die Speicherzugriffe wieder über mehrere Knoten (s. Fig. 1) erfolgt.

Überraschenderweise sind die zwei Einzel-Sockel-Systeme bei weniger Konflikt langsamer als wenn alle Threads versuchen das gleiche Lock zu aquirieren. Bei beiden steigt der Durchsatz erst mit steigender Anzahl an Threads auf die große Menge an Locks.

5.3 Konfliktsituation im Mittel

Darstellung 8 [4, Fig. 8] zeigt zusammenfassend in etwas gröber noch für dazwischenliegende Anzahlen an Locks die Leistung der getesteten Prozessorsysteme. Der Verlauf ist wie zu erwarten „etwas dazwischen"[4].

[4]Zwischen der Messung mit einem Lock und 512 Locks

Leider ergibt sich bei allen Messungen und auch laut der Autoren keine Lock-Implementierung, die in allen Fällen am besten geeignet ist. Die Warteschleifen Locks profitieren von hohem Konflikt, jedoch glänzen die einfachen Locks wenn über mehrere Sockel hinweg synchronisiert wird [4, S. 42].

6. NACHRICHTENBASIERTE KOMMUNIKATION

In Abschnitt 6.2. in der Hauptreferenzarbeit [4, S. 43] wird nun die nachrichtenbasierte Kommunikation (*message passing*) mithilfe von *SSYNC* gemessen. Nachrichten umfassen die Größe einer Cache Zeile (64 Byte). Der Nachrichtenaustausch wird mithilfe der Cache Kohärenz realisiert. Im folgenden werden *Punkt-zu-Punkt* Kommunikation analyisiert und das klassische *Client-Server-Modell*. Für beide wird jeweils Einweg als auch Hin- und Zurück-Kommunikation betrachtet.

- **Punkt-zu-Punkt Kommunikation**: Für den Opteron findet sich hier wieder die Verzögerung der Cache Kohärenz wieder. Bei einem oder bei zwei Schritten verdoppelt sich bzw. verdreifacht sich die Latenz. Für den Xeon sogar ungefähr das drei bzw. vierfache. Herausstechen tut hier besonders der Tilera durch seine vermaschte Struktur. Dieser erreicht selbst bei mehreren Speicherzugriffsschritten eine nur geringfügig höhere Verzögerung als die ohnehin schon Geringe bei lediglich einem Schritt. Der Niagara ist aufgrund seines uniformen Zugriffs auch bei Nachrichten zu benachbarten Kernen schnell. Bei allen verdoppelt sich die Latenz für Hin- und Zurück-Kommunikation ungefähr.
- **Client-Server Kommunikation**: Auch hier ist der Tilera wieder mit Abstand am schnellsten bei Einwegkommunkation, allerdings auch größtenteils bei Hin- und Zurück-Kommunikation. Der Xeon ist lokal auf einem Sockel mit 10 Klienten nochmal ein Stückchen vor dem Tilera, fällt jedoch wieder zurück sobald wieder mehrere Schritte zu anderen Sockeln in der Cache-Kohärenz liegen. Der Opteron und der Niagara nehmen sich in der Leistung nicht viel.

7. ANWENDUNGSEBENE

Auf der Anwendungsebene wird nun getestet wie sich die Leistung der Locks bzw. Nachrichten und deren Dichte auf die Leistung auswirkt. Die genauen Messergebnisse sind in [4, S. 44] und [4, S.45] zu finden.

7.1 Streutabellen Test

Diese wird jeweils mit 512 (geringer Konflikt) und 12 (hoher Konflikt) Behältern (*Buckets*) getestet. Für jeden Behälter gibt es ein eigenes Lock und für jeden Testdurchlauf ist die idealste Lockart gewählt worden. Die Ergebnisse zeigen schon dass unabhängig der Anzahl an Behältern grundsätzlich die Variante mit weniger Einträgen pro Behälter schneller ist. Warum dies so ist, wird in der Hauptreferenzarbeit nicht erklärt. Vermuten könnte man hier den größeren Aufwand zum Verwalten eines Behälters, der mit dessen Größe korrelieren sollte.

- **512 Behälter**: Besonders der Xeon profitiert vom geringen Behälterkonflikt. Mit 36 Threads hat er von allen den größten Durchsatz. Der Opteron bleibt hier ein

ganzes Stück zurück. Sowohl beim Niagara als auch beim Tilera Prozessor gibt es eine ähnlich lineare Skalierung des Durchsatzes mit der Anzahl der Threads.

- **12 Behälter**: Hier liegt der Opteron in etwa gleichauf mit dem Xeon. Hier skaliert der Opteron allerdings etwas besser mit geringerer Threadanzahl als bei der 512er Streutabelle. Die Einzel-Sockel-Systeme liegen hier auch ein Stückchen besser in der Leistung als bei der 512er Variante.

Im Fazit wird erwähnt, dass längere kritische Abschnitte die Leistung bei wenig Konflikt verbessern können. Eine Erklärung dafür liefern die Autoren nicht. Ebenso wird Nachrichtenaustausch als mögliche Leistungsverbesserung vorgeschlagen. In den Graphen finden sich leider nur die best geeigneten Lock Implementierungen und deren Leistung. Messergebnisse aus anderen Arbeiten zu einem vergleichbaren Experiment lassen sich leider nicht auffinden.

7.2 Memcached

Memcached ist ein Cache System auf Anwendungsebene, welches z. B. für Datenbanken verwendet wird. Intern arbeitet eine Streutabelle, welche mehrere Locks für Teile der Tabelle hat und ein globales Lock, falls die Tabelle vergrößert werden muss. Standardmäßig arbeitet *Memcached* mit *pthread-Mutex*. Dies wird hier durch *libslock* ersetzt. Getestet wurden folgende zwei Szenarien:

- **Get-Only**: In diesem Test wurden ledeglich „get“ Anfragen an die Streutabelle gestellt. Das globale Lock muss daher nicht aquiriert werden und folglich ist die Leistungsmessung relativ unabhängig des verwendetet Lock-Typs. Hier liegen die Flaschenhalse primär nicht an der Synchronisation.

- **Set-Only**: Hier wurden nur Einträge in die Streutabelle eingefügt. Dieser Testlauf beansprucht nun die eigentliche Synchronisation. Die Ergebnisse dazu sind in [4, S. 45] dargestellt. Auf dem Opteron ist *Memcached* mit dem *MCS*-Lock am effizientesten, bekommt allerdings wieder Skalierungsprobleme, falls mehr Threads verwendet werden, als auf einen *CPU-Die* passen. Auf dem Xeon verhält es sich ähnlich. Auf dem Niagara und dem Tilera sind die *TAS-Locks* mit am stärksten. Erklären kann man dies u. a. dadurch, dass TAS auch bei den atomaren Operationen mit zu den schnellsten gehört. Werden beim Niagara allerdings mehr Threads verwendet, als auf einen Prozessorkern eingeplant werden können, sinkt der Durchsatz dort auch nochmal.

Hier sieht man nocheinmal gut, dass die Cache Kohärenz eine große Rolle spielt, wenn es dichte Synchronisationskonflikte gibt, da allein der Speicher viel ausbremst und nicht nur die Primitive, welche atomare Instruktionen verwenden.

8. FAZIT

„Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask“ bildet eine solide Wissensbasis von Messergebnissen verschiedenster atomarer Operationen auf allen Ebenen und bietet ein gute Basis zum Nachschlagen für Optimierung bei anspruchsvollen parallelen Szenarien. Für neuere Messungen hat man die Möglichkeit von den gleichen Autoren noch aktuellere Arbeiten zu untersuchen [5] oder sich bei weiteren Autoren zu informieren [6]. Die Messwerte zum Opteron und Xeon lassen sich wahrscheinlich auch auf ihre aktuelle Generation anwenden, da die relevanten Technologien bei AMD und Intel bei den bis vor kurzem aktuellen Generationen (*Broadwell, Bulldozer*) immer noch verwendet werden (*Intel QPI, HyperTransport*). Interessant wird hier auch sein, wie sich die neue Skylake Server Generation mit *Intel UPI* in Multi-Sockel-Szenarien im Vergleich dazu schlagen wird.

Zentrale Aussagen wie „mehre Sockel können stark bremsen“ oder dass hoher Wettbewerb auf einem Lock den Durchsatz sehr stark einschränken kann, sind sehr klar dargestellt.

Die Qualität der Messergebnisse sind durchgehend relativ eindeutig und erklärt. Hier bietet die Hauptreferenzarbeit mehr Details als zum Beispiel auch neuere Arbeiten wie z. B. [12], da bei diesem beispielsweise nicht auf ideales Einplanen geachtet wird und weniger vergleichbare Messungen bei unterschiedlichen Bedinungen vorliegen.

Literatur

[1] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010. 1

[2] T. Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993. 3.2

[3] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach.* Gulf Professional Publishing, 1999. 1

[4] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM. (document), 1, 2, 1, 2, 3.1, 3.3, 3.4, 3, 3.5, 4.1, 4.2, 5, 5.3, 6, 7, 7.2

[5] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 631–644. ACM, 2015. 7.2

[6] V. Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *ACM SIGPLAN Notices*, volume 50, pages 1–10. ACM, 2015. 7.2

[7] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Elsevier, 2012. 3.2

[8] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41:33–38, 2008. 1

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991. 3.2

[10] Oracle. Ultrasparc t2: A highly-threaded, powerefficient, sparc soc, 2007. 1

[11] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe. Early experience with the barrelfish os and the single-chip cloud computer. In *MARC Symposium*, pages 35–39, 2011. 3.3

[12] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, Oct 2015. 4.2, 7.2

[13] M. Technologies. Tile-gx36 processor, 2016. 1