Seminar: Sprachen für Parallelprogrammierung

Transactional Memory

KIT – Universität des Landes Baden-Württemberg und nationales Großforschungszentrum in der Helmholtz-Gemeinschaft

IPD Snelting, Lehrstuhl Programmierparadigmen

Sven Janko

14. Juli 2010

Inhaltsverzeichnis

1 Einleitung **3**

2 Motivation **3**

3 Transactional Memory **4**

3.1 Der Begriff Transaktion 4

3.2 Konflikte 5

3.3 Granularität 5

3.4 Synchronisationsmechanismen 5

3.5 Konflikterkennung 7

3.5.1 Commit-Time Validation 7

3.5.2 Commit-Time Invalidation 8

3.6 Contention Manager 8

3.7 Offene Fragestellungen und Probleme 9

4 Fazit **10**

1 Einleitung

Multikernprozessoren sind seit einigen Jahren allgegenwärtig, wohingegen es an der Software, die die Prozessoren effizient ausnutzt, noch in einigen Gebieten fehlt. Das liegt nicht zuletzt daran, dass die Entwicklung nebenläufiger Programme sehr schwierig ist. Sie fordert vom Programmierer fundiertes Wissen über die interne Arbeitsweise der Synchronisations- und Sperrmechanismen des Systems. Aufgrund der großen praktischen Bedeutung des Parallelismus befassen sich Wissenschaftler weltweit mit einem neuen Konzept, das es ermöglichen soll solche Programme leichter zu entwickeln.

2 Motivation

Um mehrere Threads am gleichzeitigen Zugriff auf gemeinsam genutzte Speicherbereiche zu hindern ist die Verwendung von Sperren unumgänglich. Der Programmierer muss Sorge dafür tragen, dass ein gegenseitiger Ausschluss gewährleistet wird. Auch wenn die Programmierung mit Locks einigen als trivial erscheinen mag, birgt sie einige Gefahren, die nicht nur unerfahrene Entwickler häufig zu spüren bekommen.

Fünf wichtige Punkte, über die sich der Programmierer bei der Verwendung von Sperren Gedanken machen muss, werden im Folgenden vorgestellt [1]:

Synchronisation und Koordination Um einen korrekten Programmablauf zu garantieren ist es notwendig die verschiedenen Threads zu synchronisieren und deren Kommunikation zu koordinieren. Hierfür stehen spezielle Sprachkonstrukte zur Verfügung, deren Funktionsweise der Entwickler genauestens studieren muss, um sie richtig anwenden zu können.

Granularität Die Granularität ist eine wichtige Entwurfsentscheidung. Werden wenige grob-granulare Sperren verwendet, so sind die Programme zwar leichter verständlich und auch für Dritte gut nachzuvollziehen, aber die Geschwindigkeit ist unter Umständen nicht optimal. Es müssen also Einbußen in der Skalierbarkeit hingenommen werden.

Bei Verwendung von vielen fein-granularen Sperren wird der Quellcode schnell unübersichtlich und ebenso fehleranfällig, womöglich lässt sich dadurch aber die Performanz steigern.

Deadlocks Programmierfehler, die zum Stillstand des Programms führen, sind inakzeptabel. Mehrere Threads befinden sich in einem Deadlock, wenn jeder dieser Threads auf ein Ereignis (z.B. die Freigabe einer Resource) wartet, das nur ein anderer Thread verursachen kann.

Die Fehlerursache lässt sich aufgrund der nicht deterministischen Ausführung paralleler Programme nur schwer finden.

Livelock Ein Livelock ist eine Variante des Deadlocks. Im Gegensatz zum Deadlock verharren die beteiligten Threads beim Livelock nicht, sondern wechseln ständig

zwischen Zuständen hin und her, sodass kein Fortschritt im Programmablauf erreicht wird.

Priority inversion Das Problem der Prioritätsumkehr ist nicht zu vernachlässigen. Es kann auftreten, wenn ein Prozess N niederer Priorität eine Sperre auf eine gemeinsam genutzte Resource anfordert und zugeteilt bekommt, anschließend ein Prozess H hoher Priorität startet und die gleiche Resource verwenden möchte, die N bereits gesperrt hat. Im schlechtesten Fall beginnt nun ein Prozess M mittlerer Priorität zu laufen und hindert N daran mit der Bearbeitung und Freigabe der Resource fortzufahren. Die Konsequenz ist, dass auch H blockiert wird.

Wünschenswert ist, dass sich der Programmierer um die eben genannten Probleme keine Gedanken machen muss und sich alleine darauf konzentrieren kann, welche Quellcodeabschnitte vor einem gleichzeitigen Zugriff verschiedener Threads geschützt werden sollen.

3 Transactional Memory

Abhilfe soll hier Transactional Memory schaffen. Die Grundidee ist, dass mehrere Programminstruktionen zu einem atomaren Block zusammengefasst werden können. Das unterliegende System trägt dann die Verantwortung dafür, dass keine anderen Instruktionen während der Ausführung dieses Blocks auf dieselben Speicherstellen zugreifen. So sollen die oben genannten Gefahren vom Entwickler ferngehalten werden. [5]

Aus der Datenbankanwendung wurde der Begriff *Transaktion* übernommen und für den gemeinsam genutzten Hauptspeicher adaptiert.

3.1 Der Begriff Transaktion

Eine Transaktion fasst durch ein neues Sprachkonstrukt, etwa *atomic{...}*, mehrere Befehle zu einer Sequenz zusammen [5]. Um die Bedeutung, wie sie aus Datenbanksystemen bekannt ist, beizubehalten werden die folgenden drei Eigenschaften gefordert [7]:

Atomarität Die Transaktion wird entweder komplett ausgeführt oder gar nicht.

Kontinuität Das System befindet sich zu jeder Zeit in einem konsistenten Zustand, auch wenn Konflikte (siehe Abschnitt 3.2) zwischen unterschiedlichen Transaktionen auftreten.

Isolation Jede Transaktion läuft korrekt ab, unabhängig davon, ob noch andere Transaktionen parallel ausgeführt werden.

Die vierte Eigenschaft des *ACID*-Prinzips - die Dauerhaftigkeit - ist nicht gegeben, da der Hauptspeicher flüchtig ist.

3.2 Konflikte

Ein Konflikt zwischen verschiedenen Transaktionen tritt dann auf, wenn mehrere Transaktionen auf die gleiche Speicherstelle zugreifen und mindestens eine Transaktion abgebrochen werden muss, um die Konsistenz des Systems zu gewährleisten.

Es wird zwischen drei unterschiedlichen Konflikten unterschieden [2]:

W-W Zwei oder mehr Transaktionen greifen schreibend auf dieselbe Speicherzelle zu.

W-R Eine Transaktion greift schreibend und - kurze Zeit danach - eine andere lesend auf dieselbe Speicherzelle zu.

R-W Eine Transaktion greift lesend und - kurze Zeit danach - eine andere schreibend auf dieselbe Speicherzelle zu.

Der erste Konflikt (W-W) zählt zu den wahren Konflikten (*true conflicts*), da zwingend mindestens eine der konkurrierenden Transaktionen abgebrochen werden muss, um die Konsistenz des Systems weiterhin bewahren zu können. Die beiden anderen (W-R und R-W) hingegen sind so genannte falsche Konflikte (*false conflicts*), da es in bestimmten Konstellationen möglich ist, die Abschlüsse der Transaktionen zeitlich umzuordnen, sodass eine korrekte Ausführung aller betroffenen Transaktionen gewährleistet werden kann, ohne dass es zu Abbrüchen und der erneuten Ausführung von Transaktionen kommt.

3.3 Granularität

Bei der Implementierung eines Transactional Memory Systems muss festgelegt werden, mit welcher Granularität gearbeitet wird. Bei früheren Vorschlägen für hard- und softwarebasierte Systeme wurde mit Wortbreite gearbeitet, d.h. jede Transaktion bestand aus Zugriffen auf ein oder mehrere Worte. Daraus resultierte allerdings ein hoher Verwaltungsaufwand und auch ein hoher Speicherbedarf. In modernen hardwarebasierten Systemen werden Blöcke fester Größe verwendet, insbesondere Seiten oder Cache-Zeilen. Je nach Fähigkeit der Hardware sind auch andere Blockgrößen denkbar. Bei softwarebasierten Systemen benutzt man hingegen Objekte, um den Aufwand für die Zugriffskontrolle zu minimieren. [7]

3.4 Synchronisationsmechanismen

Anders als bei dem bisher verwendeten Lock-basierten Modell sollen die internen Synchronisationsmechanismen bei Transactional Memory dafür sorgen, dass der parallele Zugriff auf Datenstrukturen nicht mehr blockiert. Es wird zwischen den folgenden Eigenschaften unterschieden [1]:

obstruction-free Der Fortschritt eines Threads, der eine Operation ausführt, wird nur garantiert, solange keine Konflikte mit anderen Threads auftreten. Sobald ein zweiter Thread auf Speicherstellen zugreift, die vom ersten Thread verwendet werden,

ist die schwächste der drei Eigenschaften, obstruction-freedom, nicht mehr gegeben. Wird alleine diese Eigenschaft gefordert, besteht ohne weitere Kontrollmechanismen die Gefahr von Livelocks.

lock-free *Lock-freedom* garantiert, dass das System als Ganzes Fortschritte macht, sogar wenn Konflikte zwischen mehreren Transaktionen auftreten. Algorithmen mit dieser Eigenschaft lassen sich in einigen Fällen aus Algorithmen herleiten, die das erstgenannte Merkmal besitzen. Auch wenn lock-freedom ausschließen kann, dass Livelocks auftreten, ist es bei dieser Art von Algorithmen möglich, dass einzelne Threads verhungern, also sehr lange - oder im schlimmsten Fall für den Rest des Programmablaufs - blockiert werden.

wait-free Die stärkste der drei Eigenschaften, *wait-freedom*, sorgt dafür, dass jeder Thread Fortschritte in seiner Ausführung macht, auch wenn er mit anderen um Speicherbereiche konkurriert. Es ist nur selten möglich Algorithmen mit dieser Eigenheit von Grund auf neu zu entwickeln, die auch den Performanzansprüchen in der Praxis genügen. Hier sollen Transformationen weiterhelfen, die aus Rechenvorschriften mit der schwächsten Eigenschaft solche generieren, die wait-free sind.

An einem Beispiel lässt sich der wesentliche Unterschied zwischen dem herkömmlichen und dem neuen Konzept leicht erklären. Betrachtet wird ein Auszahlvorgang an einem Bankautomat.

Listing 1 beschreibt das Vorgehen unter Zuhilfenahme der Locks. In Zeile 1 wird zunächst eine Sperre angefordert, um sicher sein zu können, dass kein zweiter Thread durch gleichzeitige Zugriffe einen inkonsistenten Zustand erzeugt. Falls dem Thread diese Sperre zugeteilt werden kann, darf er mit der Bearbeitung des kritischen Abschnitts (Zeilen 2 - 5) fortfahren. In Zeile 6 muss die Sperre schließlich wieder freigegeben werden.

Ohne Sperren kommt hingegen der Ansatz aus, der in Listing 2 gezeigt wird. Hier genügt es, dass der Programmierer die Anweisungen, die einer kritischen Sektion angehören, durch einen atomic-Block schützt. Er braucht sich nicht um Deadlocks oder Wettlaufsituationen zu kümmern, das erledigt das Transactional Memory-Konzept für ihn. Das zugrundeliegende System erkennt, wenn eine zweite Transaktion zeitgleich auf die gleichen Speicherstellen zugreift und kann dann eine der beiden Transaktionen zum Neustart veranlassen. Der wesentliche Unterschied ist also, dass in Listing 2 der Code ohne vorherige Prüfung auf gleichzeitige Zugriffe ausgeführt wird, wohingegen in Listing 1 immer nur ein Thread zur gleichen Zeit die geschützten Befehle ausführt und eventuell warten muss, bis er an der Reihe ist.

Der Algorithmus in Listing 2 ist ein Beispiel für *lock-freedom*, er ist aber nicht *wait-free*, weil es sein kann, dass nach Zeile 2 und vor Zeile 5 eine andere Transaktion den Kontostand verändert und somit die Transaktion aus unserem Beispiel zum Neustart zwingt. Dieser Ablauf kann sich theoretisch endlos wiederholen und verhindert unseren Auszahlvorgang. *Lock-free* ist der Algorithmus, da andere Transaktionen erfolgreich abschließen und das System somit als Ganzes weiterhin Fortschritte macht.

Listing 1: Mit Sperren

```
1 fordereSperreAn();
2 leseKontostand();
3 pruefeBonitaet();
4 // breche eventuell ab
5 setzeNeuenKontostand();
6 gebeSperreFrei();
```

Listing 2: Mit Transaktionen

```
1 atomic{
2   leseKontostand();
3   pruefeBonitaet();
4   // breche eventuell ab
5   setzeNeuenKontostand();
6 }
```

3.5 Konflikterkennung

Wie eben bereits beschrieben, kommt Transactional Memory ohne Sperren aus. Grundsätzlich wird bei der Ausführung der Transaktionen ein optimistisches Grundprinzip (*optimistic concurrency*) verfolgt. Das bedeutet, dass eine Transaktion gestartet wird, ohne vorher zu überprüfen, ob es zeitgleich eine andere Transaktion gibt, die genau die gleichen Speicherbereiche verwendet. Konflikte werden im Nachhinein detektiert und behoben. Dieses Vorgehen erfordert aber eine sukzessive Protokollierung sowohl der Schreib- als auch der Lesezugriffe innerhalb einer Transaktion. Dies geschieht in den so genannten *write-* bzw. *read-sets*. Zusätzlich bekommt jeder Speicherberich eine Versionsnummer, über die leicht festzustellen ist, ob seit dem letzten Zugriff eine Veränderung stattgefunden hat. [6]

Im Grunde wird zwei Strategien zur Konflikterkennung nachgegangen [2],[6]:

eager Konflikte werden schon zu Beginn einer Transaktion erkannt. Diese Vorgehensweise hat den Vorteil, dass nur wenige Operationen ausgeführt werden, die später gar keine Auswirkung auf den Hauptspeicher haben. Es wird also nur wenig Rechenzeit verschwendet. Als Nachteil ist hier zu nennen, dass Transaktionen auch unnötigerweise abgebrochen werden können, z.B. dann, wenn die störende Transaktion später selbst (aufgrund Konflikte mit anderen Transaktionen) abbricht.

lazy Es wird erst zum Zeitpunkt des Abschlusses der Transaktion überprüft, ob Konflikte entstanden sind. Hier sind die Vor- und Nachteile genau entgegengesetzt zur zuerst genannten Praktik. Diese Methode reagiert nicht auf Konflikte, die sich später gar nicht auswirken, lässt aber dafür zum Scheitern verurteilte Transaktionen lange laufen.

Für die *lazy*-Variante der Konflikterkennung existieren aktuell zwei Ansätze in der Forschung. *Commit-Time Validation* und *Commit-Time Invalidation*.

3.5.1 Commit-Time Validation

Nahezu alle erforschten Transactional Memory-Systeme verwenden zur Konfliktdetektion die *Commit-Time Validation*-Methode. Hierzu werden die oben bereits erwähnten Versionsnummern verwendet, um Konflikte zwischen Transaktionen festzustellen. Die

Versionsnummern der *read-* und *write-sets* einer Transaktion werden mit den Nummern derselben globalen Speicherstellen verglichen. Wird nun eine Übereinstimmung festgestellt, kann die Transaktion erfolgreich beendet werden, die Konsistenz ist weiterhin gewährleistet. Stimmen die verglichenen Versionsnummern allerdings nicht überein, muss die prüfende Transaktion abgebrochen und von neuem gestartet werden. Obwohl dieses Verfahren für Szenarien mit wenigen Konflikten sehr effizient ist, verringert es den Transaktionsdurchsatz bei vielen Konflikten. Es gibt also weniger Transaktionen, die pro Zeiteinheit erfolgreich beendet werden können.

Abbildung 1 verdeutlicht genau dieses Szenario: Transaktion T_1 schreibt an die Stelle X, kurze Zeit später lesen mehrere Transaktionen T_2 bis T_n - die sogenannten *in-flight transactions* - den noch alten (von T_1 unveränderten) Wert aus X. Schließt T_1 jetzt erfolgreich ab, müssen alle *in-flight*-Transaktionen abgebrochen werden.

3.5.2 Commit-Time Invalidation

Das Abbrechen mehrerer Transaktionen aufgrund des Abschlusses einer einzelnen Transaktion kann mit der *Commit-Time Invalidation*-Strategie verhindert werden. Abbildung 2 zeigt den zeitlichen Ablauf für diese Variante. Zur Konflikterkennung vergleicht man jetzt den Speicher von T_1 mit dem der *in-flight*-Transaktionen. Werden hier Inkonsistenen aufgedeckt, kann T_1 abgebrochen werden und ermöglicht so T_2 bis T_n das erfolgreiche Abschließen.

Trotz dieser Erhöhung des Durchsatzes an abgeschlossenen Transaktionen wurde in praxisnahen Tests festgestellt, dass das zuerst vorgestellte Verfahren, die *Commit-Time Validation*, effizienter ist. [6]

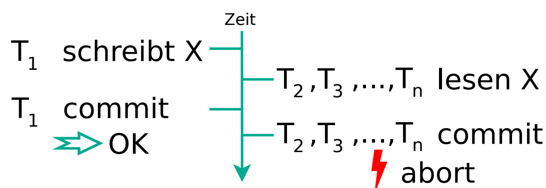

Abbildung 1: Commit-Time Validation

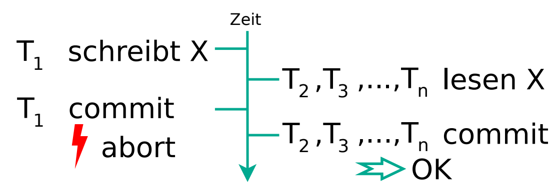

Abbildung 2: Commit-Time Invalidation

3.6 Contention Manager

Um das Verhungern (engl.: *starvation*) einzelner Transaktionen zu verhindern, kann ein *Contention Manager* verwendet werden. Er sorgt bei softwarebasierten Transactional Memory-Systemen dafür, dass Transaktionen Fortschritte machen, dass sie also nicht fortlaufend durch einen Konflikt mit anderen Transaktionen abgebrochen und neu gestartet werden. Der Programmierer soll hierbei die Möglichkeit haben, einen Maximalwert anzugeben, der bestimmt, wie oft eine Transaktion höchstens abgebrochen werden darf.

Nach jeder Feststellung eines Konfliktes wird der *Contention Manager* aufgerufen, dieser entscheidet dann anhand des eingegebenen Wertes, ob die Transaktion abgebrochen werden darf, oder nicht. [4], [6]

3.7 Offene Fragestellungen und Probleme

Semantikprobleme

Trotz schon langer und intensiver Forschung im Gebiet des transaktionalen Speichers sind noch nicht alle Probleme vollends geklärt. Uneinig ist man sich z.B. über die Lösung der folgenden Semantikprobleme [3]:

I/O-Operationen Da Transaktionen abgebrochen werden können müssen, dürfen innerhalb einer Transaktion nur solche Instruktionen ausgeführt werden, die rückgängig gemacht werden können. Bei I/O-Befehlen ist das nur schwer beziehungsweise gar nicht möglich.

Ausgaben auf einer Konsole mögen noch mit angemessenem Aufwand zwischengespeichert und rückgängig gemacht werden können. Für Festplattenzugriffe wäre das Prozedere schon deutlich aufwändiger. Einzelne Aktionen von anderen Ein- und Ausgabegeräten können unter Umständen gar nicht ungeschehen gemacht werden.

weak vs. strong atomicity Weiterhin ist zu entscheiden, ob man Zugriffe auf Speicherbereiche, die von einer Transaktion verwendet werden, zulassen möchte, wenn diese Zugriffe selbst nicht innerhalb einer Transaktion gestartet werden. Entschließt man sich, diese Zugriffe zuzulassen, so spricht man von *weak atomicity*, andernfalls von *strong atomicity*.

Exceptions Die Ausnahmebehandlung ist ebenfalls ein strittiger Punkt. Hier ist noch zu definieren, wie genau sich eine Transaktion verhalten soll, wenn eine Ausnahme auftritt.

Kompatibilität zu bisherigem Code

Natürlich möchte man nicht alle bereits existierenden Programme neu schreiben müssen, um das Transactional Memory-Konzept nutzen zu können. Deswegen sind noch Anstrengungen nötig, um zu ermöglichen, dass Konstrukte des neuen Verfahrens parallel zu den herkömmlichen Sperrmechanismen eingesetzt werden können.

Indeterminismus erschwert Fehlerfindung

Die Entscheidung über den Abbruch einer bestimmten Transaktion hängt stark vom aktuellen Zustand des Systems ab. Der Programmierer kann nicht vorhersehen, welche Transaktion wann mit einer anderen Transaktion um Speicherbereiche konkurrieren wird. Dieses indeterministische Verhalten des Programms kann die Fehlerfindung deutlich erschweren.

Performanz

Bisher ist es zwar schon gelungen einen deutlichen Geschwindigkeitszuwachs im Vergleich zu Lock-basierten Systemen zu erzielen. Die verwendeten Datenstrukturen wiesen bei diesen Tests aber ausgewählte Eigenheiten auf, die in der Praxis nur selten vorkommen. Die meisten Implementierungen kommen bisher noch nicht an die Performanz von feingranularen Sperren heran.

4 Fazit

Alles in allem kann man sagen, dass Transactional Memory ein sehr vielversprechendes Konzept ist, das eine attraktive Alternative zu den bisherigen Sperrkonstrukten darstellt. Es lässt auf ein leichteres Programmiermodell mit mehr Garantien hoffen und verspricht Korrektheit, Skalierbarkeit und Effizienz der parallelen Programme.

Trotz schon zahlreicher Implementierungen in Software wird Transactional Memory aber immernoch fast ausschließlich für die Forschung eingesetzt. Dies dürfte wohl auch so bleiben, bis die Probleme aus Abschnitt 3.7 gelöst wurden. [3]

Literatur

[1] Keir Fraser, Tim Harris: *Concurrent Programming Without Locks*, 2007, `http://portal.acm.org/citation.cfm?id=1233307.1233309&coll=GUIDE&dl=GUIDE&CFID=92043844&CFTOKEN=83703377`

[2] Arrvindh Shriraman, Sandhya Dwarkadas, Michael L. Scott: *Flexible Decoupled Transactional Memory Support*, `http://portal.acm.org/citation.cfm?id=1381306.1382134&coll=GUIDE&dl=GUIDE&CFID=92043844&CFTOKEN=83703377`

[3] Cascaval, Blundell, Michael, Cain, Wu, Chiras, Chatterjee: *Software Transactional Memory: Why Is It Only a Research Toy?*, Sept 08, `http://portal.acm.org/citation.cfm?id=1454456.1454466&coll=GUIDE&dl=GUIDE&CFID=92043844&CFTOKEN=83703377`

[4] Jennifer Mankin, David Kaeli, John Ardini: *Software Transactional Memory for Multicore Embedded Systems*, `http://portal.acm.org/citation.cfm?id=1542465`

[5] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy: *Composable Memory Transactions*, 2005, `http://portal.acm.org/citation.cfm?id=1065952`

[6] Justin E. Gottschlich, Manish Vachharajani, Jeremy G. Siek: *An Efficient Software Transactional Memory Using Commit-Time Invalidation*, 2010, `http://portal.acm.org/citation.cfm?id=1772954.1772970&coll=GUIDE&dl=GUIDE&CFID=92043844&CFTOKEN=83703377`

[7] Victor Pankratius, IPD Tichy: *Vorlesung: Multikern-Rechner und Rechnerbündel*, KIT WS 2009, http://wwwipd.ira.uka.de/Tichy/uploads/folien/166/Cluster05STM.pdf