Hardwareunterstützung für nicht-blockierende Synchronisation

Martin Hoffmann
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martin.Hoffmann@e-technik.stud.uni-erlangen.de

ABSTRACT
Mit wachsender Prozessoranzahl und der sich daraus ergebenden Parallelisierung werden Computersysteme immer komplexer. Insbesondere die Synchronisation verteilter Daten zeigt sich dabei als problematisch. Blockierende Synchronisationsverfahren sind nicht immer zufriedenstellend. Blockierende Sperren führen zu einer Serialisierung, die die Leistungsvorteile der Parallelisierung wieder aufheben kann. Eine Lösungsmöglichkeit sind nicht-blockierende Synchronisationsverfahren mit ihren Eigenschaften der Lock- und Wartefreiheit, die die parallele Ausführung von Software nicht einschränken.

In den meisten aktuellen Prozessoren sind Operationen implementiert, die das Design von warte- und lockfreien Datenstrukturen ermöglichen. Dazu gehören z.B. Compare-And-Swap und Load-Linked/Store-Conditional Operationen. Für die Zukunft ist relevant, ob und wie gut sich diese Mechanismen auch in größeren Mehrkernsystemen implementieren lassen.

1. Überblick
In den Abschnitten 2 und 3 wird auf die Notwendigkeit der Synchronisation von verteilten Datenstrukturen eingegangen und die zwei Lösungsmöglichkeiten blockierende und nicht-blockierende Synchronisation aufgezeigt. Hierbei wird näher auf die nicht-blockierende Variante eingegangen und deren Eigenschaften der Warte- und Lockfreiheit vorgestellt. Die Grundlage für nicht-blockierende Synchronisation auf Multiprozessorsystemen sind atomare Synchronisationsbefehle mit einer Read-Modify-Write Semantik. Die wichtigsten und am meisten eingesetzten Vertreter dieser Befehle sind Compare-And-Swap auf CISC Maschinen und Load-Linked/Store-Conditional auf RISC Prozessoren. Die Semantik dieser Befehle wird vorgestellt, wobei auch auf das ABA-Problem des Compare-And-Swap Befehls eingegangen wird. Im Anschluss werden die grundlegenden Architekturen von Multiprozessorsystemen vorgestellt und wie auf diesen Architekturen Synchronisation hergestellt werden kann. Grundlage hierfür ist ein kohärenter Cache-Zugriff, der auch den, eventuell verteilten, Hauptspeicher konsistent hält. Es werden zwei Cache-Kohärenz Protokolle vorgestellt und bewertet: Snoopingbasierte und verzeichnisbasierte Cache-Kohärenz.

2. Blockierende und nicht-blockierende Synchronisation
Verteilt genutzte Datenstrukturen fordern zwingend Synchronisationsverfahren, um die Datenbestände zu jeder Zeit konsistent halten zu können. Warte- und lockfreie Synchronisationsmethoden ermöglichen es parallele Prozesse zu synchronisieren ohne dabei bestimmte Programmabschnitte mit Hilfe von Lockvariablen sperren zu müssen. In den folgenden Abschnitten wird zunächst die Problematik der blockierenden Synchronisation aufgezeigt. Daraufhin werden die zwei nicht-blockierenden Semantiken „wartefrei“ und „lockfrei“ vorgestellt.

2.1 Die Problematik blockierender Techniken
Die klassische blockierende Synchronisation zeigt eine Reihe von Nachteilen. Durch den Einsatz von Schlossvariablen wie Semaphoren oder Mutexen werden kritische Abschnitte definiert, in denen nur ein Prozess zur gleichen Zeit Zugriff auf ein bestimmtes Betriebsmittel erhält. Das damit verbundene Blockieren der beteiligten anderen Prozesse birgt dabei einige Nachteile. [2]

Es kann zu Verklemmungen (Deadlocks) kommen, falls es gegenseitige Abhängigkeiten gibt. Außerdem kann sich ein gewisser Effizienzverlust bei zu grob granularen Sperren zeigen. Eine mögliche Verbesserung durch eine feinere Granularität kann dem entgegenwirken, erhöht allerdings auch die Fehleranfälligkeit. Das Erkennen und Sperren von kritischen Abschnitten ist nicht immer einfach und dadurch fehleranfällig. Außerdem erschwert die blockierende Synchronisation die Wartbarkeit und Erweiterbarkeit der Programmsoftware. Schließlich kann blockierende Synchronisation Prioritätsumkehrung verursachen, wenn hochpriore Prozesse von Niederprioren durch gehaltene Sperren blockiert werden. Dies kann insbesondere bei Echtzeitsystemen ein großes Problem darstellen.

2.2 Nicht-blockierende Synchronisation
Nicht-blockierende Synchronisationstechniken das Sperren kritischen Abschnitten. Zur Modifikation von Datenstrukturen werden ausschließlich atomare Operationen genutzt. Kleine Änderungen, wie das Ändern eines Zählers oder die Manipulation eines Zeigers, können durch Prozessorbefehle wie Compare-And-Swap oder Load-Linked/Store-Conditional vollzogen werden. Bei komplexen Datenstrukturen wird die Änderung zunächst auf einer lokalen Kopie des Objektes durchgeführt. Wurde das ursprüngliche Objekt währenddessen nicht verändert, wird es durch die modifizierte Kopie ersetzt. Im anderen Fall schlägt die Operation fehl und muss wiederholt werden.

Wie die Blockierenden bergen auch die nicht-blockierenden Synchronisationsverfahren gewisse Nachteile. Die Algorithmen sind oft sehr komplex und nur schwer verständlich. Außerdem gibt es nur wenige universell einsetzbare nicht-blockierende Algorithmen für bestimmte einfache Datenstrukturen. Um komplexe Datenstrukturen nicht-blockierend zu synchronisieren muss in vielen Fällen eigener Algorithmus erstellt werden.

Es gibt zwei Arten von nicht-blockierender Synchronisation. Man unterscheidet zwischen „wartefreier“ und „lockfreier“ Synchronisation.

2.3 Lockfreiheit
Lockfreie Synchronisation ermöglicht Zugriffe auf gemeinsam genutzte Ressourcen ohne diese durch Schlossvariablen zu schüt-

zen. Dies geschieht indem Prozesse das Berechnungsergebnis vorbereiten und dann versuchen das Ergebnis in einem atomaren Schritt zu übertragen. Der atomare Befehl muss gleichzeitig prüfen, ob die Datenstruktur inzwischen verändert wurde. Ist die geschehen, muss die Berechnung erneut erfolgen.

Lockfreie Synchronisation verhindert zwar die Verklemmung, jedoch kann sie zum Verhungern von Prozessen führen (*Starvation*). Wird eine komplexe Änderung, die in einem Prozess vorgenommen wird, stets von kürzeren Modifikationen ungültig gemacht, kann dieser Prozess verhungern.[9]

2.4 Wartefreiheit

Man nennt ein Synchronisationsverfahren wartefrei, wenn die Ausführungszeit einer Operation immer eine feste obere Grenze besitzt. Für einen wartefreien Algorithmus ließe sich somit eine harte obere Schranke für dessen Ausführungszeit bestimmen, die unabhängig von den parallel laufenden Prozessen im System ist. Wartefreiheit ist die stärkste Garantie für Nicht-Blockieren. zulässt. Es wird eine Fortschrittsgarantie gegeben, die das systemweite Voranschreiten aller Threads garantiert und dabei Verhungern (Starvation) nicht zulässt. Der Aufwand für die Implementierung eines wartefreien Algorithmus ist sehr hoch, außerdem steigt der Speicher- und Zeitbedarf mit der Anzahl der beteiligten Prozesse.[2]

Ein entsprechend mächtiger atomarer Synchronisationsbefehl ist essentiell für eine effektive lockfreie Datenstruktur. Im folgenden Abschnitt wird die Semantik und Mächtigkeit zweier atomarer Maschinenbefehle vorgestellt die lockfreie Synchronisation ermöglichen.

3. Atomare Synchronisationsbefehle

Die Grundlage für warte- und lockfreie Synchronisationsverfahren sind atomare Operationen mit einer „Read-Modify-Write“ – Semantik. Das bedeutet einen Datenwert aus dem Speicher laden (Read), ihn durch Berechnung verändern (Modify) und den Wert wieder im Speicher ablegen (Write). Dies muss atomar geschehen, es muss also sichergestellt sein, dass die Speicherstelle während der Modifikation nicht von einem anderen parallel laufenden Prozess verändert wurde. Der bekannteste Vertreter solcher Befehle ist „Compare-And-Swap“ (CAS). PowerPC und DEC Alpha bieten das theoretisch mächtigere Paar Load-Linked/Store-Conditional (LL/SC).

3.1 CAS – Compare And Swap

Der Befehl Compare-And-Swap (CAS) vergleicht eine Speicherstelle mit einem beliebigen Wert und überschreibt diese bei Gleichheit mit einem neuen Wert. Dies geschieht in einem atomaren Maschinenbefehl. Dieser benötigt drei Operanden: Die Speicheradresse, den Vergleichswert und den zu setzenden Wert. Liegt keine Gleichheit vor, wird die Speicherstelle nicht überschrieben und ein Rückgabewert wird gesetzt, der den Abbruch meldet[3]. Folgende Beispielimplementierung verdeutlicht die Semantik. Die Funktion wird atomar ausgeführt.

```
1 bool CAS(int *speicherstelle, int alt, int neu){
2       if (*speicherstelle != alt) return false;
3       *speicherstelle = neu;
4       return true;
5 }
```

Listing 1: Beispielimplementierung einer CAS Operation

Für den atomaren Ablauf muss die CAS Operation in Hardware implementiert werden. Die Architektur sichert dabei den atomaren Ablauf der Instruktion durch eine feingranulare Sperre des Prozessorbusses. Der Gebrauch eines solchen Hardwarelocks verletzt dabei nicht die Defintion von Lockfreiheit, da die Dauer der Sperre auf die Ausführung des Befehls begrenzt ist. In speichergekoppelten Mehrprozessorsystemen muss außerdem ein Verfahren implementiert sein, das die Kohärenz des Speichers und der einzelnen CPU Caches über Prozessorgrenzen hinweg gewährleistet. Da jeder Prozessor einen eigenen lokalen Cache besitzt, kann es sein dass mehrere Hauptspeicherkopien der gleichen Speicherstelle vorliegen. Es muss sichergestellt werden, dass der CAS Befehl immer den aktuell gültigen Wert zum Vergleich hinzuzieht.

Mit dem CAS Befehl lässt sich eine lockfreie Datenstruktur folgendermaßen entwerfen. Der Wert der zu ändernden Speicherstelle wird lokal gesichert und ein neuer Wert berechnet. In einer atomaren CAS Operation wird der lokal gesicherte Ursprungswert mit der Speicherstelle verglichen. Stimmen die Werte noch überein geht man davon aus, dass der Vorgang nicht unterbrochen wurde und tauscht den alten mit dem neuen Wert aus. Schlägt die CAS Operation fehl, wird sie von neuem begonnen. Dies ist ein optimistischer Ansatz, bei dem davon ausgegangen wird, dass eine Intervention zwischen dem Auslesen des Wertes und der CAS Operation eine geringe Wahrscheinlichkeit aufweist.

3.2 ABA - Problem

Der CAS Befehl kann jedoch zu Inkonsistenzen führen. Problematisch ist nämlich, dass der Befehl nur die Gleichheit zweier Werte prüft, aber nicht erkennen kann, ob die Speicherstelle seit dem Laden des Ursprungswertes verändert wurde.[3] In einer nebenläufigen Umgebung kann folgendes Problem auftreten:

Es sei folgender (naiver) lockfreier Stack betrachtet, Fehlerbehandlungen, z.B. bei leerem Stack, wurden zur besseren Übersicht weggelassen:

```
1 class Stack {
2       //die Spitze des Stack
3       //(gemeinsame, zu schützende Ressource)
4       Obj* top;
5  Obj* Pop(){
6       while(1){
7               Obj* ret = top;
8               Obj* next = ret->next;
9       if(CAS(top, ret, next))
10              return ret;
11      }
12 }
13 void Push(Obj* obj){
14      while(1){
15              Obj* next = top;
16              obj->next = next;
17      if(CAS(top, next, obj))
18              return;
19      }
20 }
21 };
```

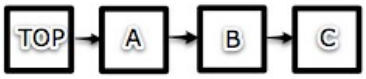
Listing 2: Ein einfacher lockfreier Stack

Die Pop-Methode holt das oberste Objekt aus dem Stack. Der Ablauf ist typisch für eine lockfreie Datenstruktur und deren Read-Modify-Write Semantik. Zunächst wird die Adresse des obersten Elements das zurückgegeben werden soll, lokal gesichert

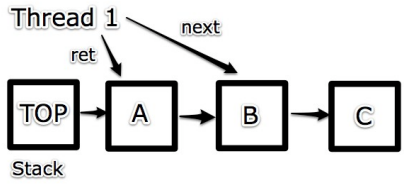
(Read, Zeile 7), sowie die Adresse des nachfolgenden Elements, das zum neuen „Top“-Element wird (Modify, Zeile 8). In einer atomaren CAS Operation wird versucht den globalen Top-Zeiger mit der Speicheradresse des Folgeelements zu ersetzen (Write, Zeile 9). Falls der in Zeile 7 geladenen Ursprungswert noch mit dem aktuellen Wert übereinstimmt, wird die Speicherstelle überschrieben. Man geht dann davon aus, dass der Stack zwischen dem Auslesen in Zeile 7 und der CAS Operation nicht verändert wurde. Falls sich die Werte unterscheiden, wurde die Operation unterbrochen und der Stack verändert. In diesem Fall schlägt die CAS Operation fehl und der Vorgang wird durch die while-Schleife neu begonnen. Die Push-Methode läuft entsprechend analog ab.

Folgender Ablauf kann dabei allerdings zu Problemen führen.

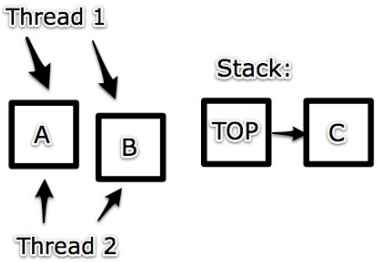
Der initiale Stackzustand sei:



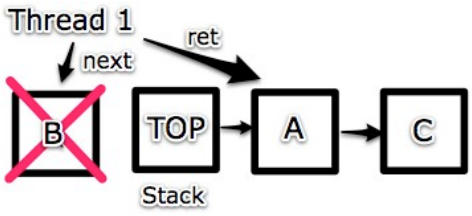
Thread 1 beginnt eine pop-Operation. Seine lokalen Variablen bekommen dabei die Werte A (siehe Zeile 7 in Listing 2) und B (Zeile 8).



Noch vor der CAS Operation wird der Thread unterbrochen und ein anderer Prozess (Thread 2) führt zwei pop-Operationen erfolgreich aus. Folgende Abbildung zeigt das Resultat: Auf dem Stack liegt nur noch Objekt C, der unterbrochene Thread 1 hält die Zeiger auf die Objekte A und B.



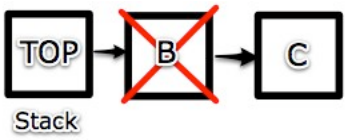
Nun löscht Thread 2 Objekt B und schiebt Objekt A zurück auf den Stack:



.

Nun wird Thread 1 wieder eingelastet und dessen CAS Operation ausgeführt.

Diese wird erfolgreich sein, da der Top-Zeiger tatsächlich auf das Objekt A zeigt (top == ret). Der Rückgabewert der CAS Operation ist gültig und korrekt, allerdings wurde Objekt B von Thread 2 inzwischen gelöscht. Der Top Zeiger zeigt also fortan auf eine freigegebene Speicherstelle:



Man kann sich bei Nutzung des CAS – Befehls somit nicht immer darauf verlassen, dass ein Speicherinhalt seinen Wert zwischen dem Auslesen und neu Beschreiben nicht schon verändert hat. Manche Architekturen bieten neben dem Compare-And-Swap Befehl für eine Speicherstelle (Single-Word Compare-And-Swap) auch einen CAS Befehl für doppelwortbreite Speicherstellen: Double-Word-Compare-And-Swap. Die zweite Speicherstelle kann dann für einen Zähler genutzt werden, der Zugriffe auf die erste Speicherstelle mitverfolgt und dadurch auch mit sehr großer Wahrscheinlichkeit den ABA Fall erkennen kann. Dennoch ist dies keine Lösung des Problems: Wird der Thread nach dem Einlesen des Zählerwertes genau 2^32 unterbrochen wird der Zählerwert durch den Überlauf wieder übereinstimmen. Die Wahrscheinlichkeit dieses Falls ist zwar extrem klein, er ist aber nicht ausgeschlossen.

Der Load-Linked/Store-Conditional Befehl, der im folgenden Abschnitt vorgestellt wird umschifft das ABA-Problem.

3.3 LL/SC – Load-Linked/Store-Conditional

Die Semantik der Befehlspaare Load-Linked (Link oder auch Locked) und Store-Conditional lässt das ABA-Problem nicht zu. LL benötigt als Argument die zu ladende Speicherstelle und gibt diese zurück. Dabei wird diese intern markiert. Die zugehörige SC Operation hat zwei Argumente: Zum einen die zu beschreibende Speicherstelle und den zugehörigen Wert. SC soll dann, und nur dann, erfolgreich ausgeführt werden, wenn die Speicherstelle seit dem zugehörigen Aufruf von LL nicht von einem anderen Thread beziehungsweise Prozessor verändert wurde. Dies ähnelt stark der CAS-Semantik, wobei hier, durch die Markierung der Speicherstelle bei der LL Operation sichergestellt werden kann, dass diese inzwischen nicht verändert wurde. Jeder andere schreibende Zugriff würde die Markierung invalidieren, und den SC Befehl scheitern lassen. Das ABA-Problem kann somit nicht auftreten.

Diese Operation erfordert mehr Logik. Ein CAS Befehl benötigt keine weiteren Spezialregister, die Semantik erfüllt ein normaler Maschinenbefehl. Im Gegensatz dazu benötigt das LL/SC Befehlspaar zusätzlichen Speicher um die Reservierung und die zugehörige Adresse zu sichern. Der PowerPC implementiert beispielsweise die LL/SC Semantik folgendermaßen.[5]

Der Prozessor lädt die übergebene Adresse und setzt gleichzeitig eine Reservierung auf diese. Der PowerPC Prozessor kann dabei immer nur eine, nicht mehrere, Reservierungen gleichzeitig halten.

Das Speichern wird nur dann durchgeführt, wenn die Reservierung zur zugehörigen Adresse noch gültig ist. Ist dies der Fall, wird die Reservierung gelöscht, das Datum an der Adresse gespeichert und ein Kontrollregister auf wahr gesetzt.

Die Reservierung wird bei jedem Ereignis zurückgesetzt werden, das die per LL geladene Speicherstelle verändern kann, bevor das zugehörige SC ausgeführt wurde. Ein solches Ereignis ist beispielsweise die Invalidierung der zugehörigen Cachezeile durch den Cache Controller. In diesem Fall konnte ein anderer Prozessor die Speicherstelle erfolgreich beschreiben. Des Weiteren muss die Reservierung bei jedem Kontextwechsel gelöscht werden, damit

kein anderer Thread des gleichen Prozessors ein unzulässiges SC ausführen kann. Dadurch dass eine Cachezeile meist erheblich größer ist, als nur ein Wort (z.B.: PowerPC620: 64 Bytes), kann es auch zu ungewollten Invalidierungen kommen. Der Prozessor muss nicht unbedingt genau die reservierte Speicherstelle verändern, die Reservierung wird auch aufgehoben, wenn eine andere Stelle in der zugehörigen Cachezeile verändert wird („False-Sharing“).

In der LL/SC Umsetzung des DEC Alpha Prozessors invalidiert sogar jede Exception implizit die Reservierung.

Diese ungewollten („spurious“) Invalidierungen lassen sich in realen Implementierungen nicht ganz verhindern und können im Bezug auf lockfreie Algorithmen problematisch sein.

Probleme können dann auftreten, wenn eine große Anzahl an Threads/Prozessoren auf nahegelegen Datenbereichen gleichzeitig arbeiten.

Außerdem muss bei der Implementierung des Algorithmus darauf geachtet werden, dass dieser zwischen dem LL und dem SC nicht selbst einen Befehl ausführt, der die Reservierung aufhebt. (Prozessorabhängig!)

Dennoch ist der LL/SC auf vielen RISC Architekturen die einzige Möglichkeit für eine atomare Read-Modify-Write Operation.

3.4 Hierarchie nach Herlihy

Herlihy führte eine Unmöglichkeits- und Universalitätshierarchie ein, die atomare Operationen bezüglich ihrer jeweiligen Mächtigkeit einordnet [1][4]. Diese Hierarchie basiert auf den Konzepten der Warte- und Lockfreiheit. Laut dieser Hierarchie kann eine atomare Operation einer niedrigen Ebene keine lockfreie Implementierung einer atomaren Operation einer höheren Ebene bieten. Die niedrigste Ebene der Hierarchie ist:

- ∞ CAS und LL/SC
- 2 fetch-and-store, fetch-and-add, test-and-set
- 1. atomares Laden und Speichern.

Herlihy nennt den CAS-Befehl eine universelle Primitive und setzt ihn auf die Ebene unendlich, ebenso wie LL/SC. Dies gilt aber nur, wenn die SC Operation nur dann fehlschlägt, wenn die zugehörige Speicherstelle tatsächlich geändert wurde. In realen LL/SC Implementierungen ist dies, wie im vorherigen Abschnitt dargestellt jedoch nicht gegeben.

Mit Hilfe von LL/SC oder CAS Operationen können somit laut Herlihy fetch-and-store oder test-and-set Befehle lockfrei nachgebildet werden. Umgekehrt ist dies nicht möglich. Ebenso können sich Primitiven auf Ebene 2 nicht gegenseitig nachbilden. Allerdings lässt mit dem LL/SC Befehl eine CAS Operation nachbilden. Der umgekehrte Fall ist aufgrund der ABA-Problematik nicht möglich.

Keine der universellen Primitiven Load-Linked/Store-Conditional und Compare-And-Swap zeigt sich somit als unproblematisch.

CAS leidet unter dem ABA-Problem, das nur mit linearem Aufwand behoben werden kann. Dennoch ist es in der CISC Welt sehr weit verbreitet und wird in der aktuellen Entwicklung von lockfreien Datenstrukturen oft herangezogen.

Load-Linked/Store-Conditional wäre die optimale Read-Modify-Write Primitive, wenn es eine starke Implementierung geben würde. In realen Prozessoren kann nur die schwache Variante umgesetzt werden, die ungewollte Invalidierungen der Reservierung zulässt.

4. Multiprozessorsysteme

Der aktuelle Verlauf der Prozessorentwicklung geht immer mehr in Richtung Mehrkernprozessoren. Dies sind Mikroprozessoren, die mehr als einen vollständigen Hauptprozessor in einem Chip integrieren. Aktuelle Ausprägungen sind Doppelkernprozessoren (Dual-Core: Intel Core 2 Duo, AMD Athlon X2), Vierkernprozessoren (Quad-Core: Intel Core 2 Quad, AMD Phenom X4) und sogar Achtkernprozessoren (Sun UltraSPARC T2). Hinzu kommt die Möglichkeit mehrere Mehrkernprozessoren zu Mehrprozessorsystemen zusammen zu führen. Aus diesem Grund ist es wichtig die Hardwareunterstützung für lock- und wartefreie Synchronisation auch im Bezug auf aktuelle Mehrkernprozessoren und Mehrprozessorsysteme zu betrachten.

4.1 Nachrichtengekoppelte Prozessorsysteme

Multiprozessorsysteme lassen sich im Hinblick auf die Kopplung der einzelnen Kerne unterscheiden. Zum einen in speichergekoppelte Systeme und zum anderen in nachrichtengekoppelte Systeme. Nachrichtengekoppelte Systeme sind im allgemeinen Rechencluster die aus mehreren physikalisch verteilten Prozessorknoten bestehen, welche jeweils einen eigenen lokalen Adressraum aufweisen. Die Kommunikation findet dabei durch Nachrichtenaustausch statt. Solche Systeme arbeiten sehr effizient, wenn das Rechenproblem gut auf Prozessorebene parallelisierbar ist und somit wenig Kommunikation zwischen den einzelnen Knoten erforderlich ist.

Die Synchronisation innerhalb eines so gekoppelten Systems wird implizit über Nachrichten vorgenommen. Sie werden deshalb im Folgenden nicht weiter betrachtet.

4.2 Speichergekoppelte Systeme

Im Gegensatz dazu arbeitet ein speichergekoppeltes System mit einem gemeinsamen Adressraum auf den alle beteiligten Prozessoren zugreifen können. Die Kommunikation und Synchronisation verläuft über gemeinsame Variablen.

4.2.1 Distributed Shared Memory Systeme

Speichergekoppelte Systeme lassen sich weiter in symmetrische Multiprozessoren (SMP) und in sogenannte Systeme mit verteiltem gemeinsamen Speicher (Distributed Shared Memory DSM) aufteilen. Letztere besitzen einen logischen gemeinsamen Adressraum trotz physikalisch verteilter Prozessorknoten. Abbildung 1 zeigt schematisch den Aufbau eines DSM Systems mit *m* Knoten. Jeder Knoten besitzt seinen eigenen lokalen Cache und Hauptspeicher. Über ein Verbindungsnetzwerk sind die einzelnen Speicher miteinander verbunden. Beispiele für klassische DSM-Systeme sind DASH, SCI und SGI Origin. (z.B. SGI Origin 3400 am RRZE [6])

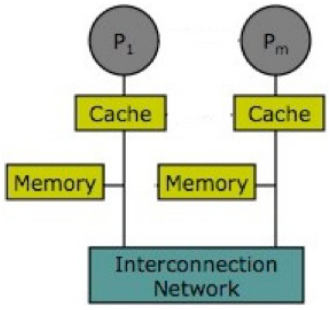


Abbildung 1: Schema eines Distributed Shared Memory Systems

Abbildung 1 zeigt den schematischen Aufbau eines solchen DSM-Systems mit 1 .. m Prozessorknoten. Diese besitzen jeweils einen

eigenen Cache und Hauptspeicher und sind über eine Verbindungsnetzwerk miteinander verbunden

4.2.2 Symmetrische Multiprozessoren

Symmetrische Multiprozessoren haben einen globalen gemeinsamen Speicher, sowohl logisch als auch physikalisch. Die einzelnen identischen Prozessoren sind mit ihren Caches an einem gemeinsamen Bus mit dem Hauptspeicher verbunden. Abbildung 2 soll dies ebenso schematisch für m Prozessoren darstellen. Jeder Prozessor besitzt seinen eigenen Cache. Über einen gemeinsamen Bus können die Knoten untereinander und mit dem gemeinsam genutzten Hauptspeicher kommunizieren,

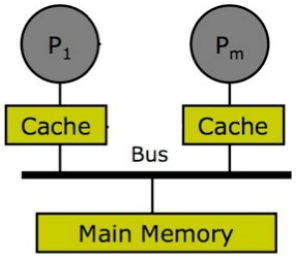


Abbildung 2: Schema eines Symmetrischen Multiprozessor Systems

Können in einem SMP System alle Prozessoren gleichberechtigt mit der gleichen Geschwindigkeit auf alle Hauptspeicherbereiche zugreifen, so nennt man dies Unified Memory Access (UMA). Die ersten Multiprozessorsysteme waren meist SMP Maschinen. Auch der Großteil der aktuellen Desktop-Multicoreprozessoren weisen diese Architektur auf.

Die Prozessorentwicklung im Serverbereich, sowohl bei AMD als auch bei Intel geht momentan in Richtung NUMA-Architekturen, also Non-Unified-Memory-Access. AMD geht mit seinen Opteron/Phenom-Prozessoren diesen Weg, Intel mit dem Core i7. Beide setzen integrierte Speichercontroller und schnelle serielle Verbindungen zwischen den Prozessoren ein. Es lassen sich beispielsweise zwei Opteron Dualcore Prozessoren auf einem Mainboard zu einem NUMA System zusammenschalten, wie in Abbildung 3 dargestellt. Jeder Prozessor besitzt dabei eine lokale Speicherbank. Über den „Hypertransport“ Kanal sind beide CPUs miteinander verbunden. Jeder Prozessor darf gleichberechtigt auf alle Speicherbänke zugreifen, jedoch ist die Zugriffsgeschwindigkeit, im Gegensatz zum UMA, unterschiedlich. Während lokale Zugriffe sehr schnell vonstatten gehen, erhöht ein entfernter Zugriff, der über den Hypertransportkanal laufen muss, die Latenz.

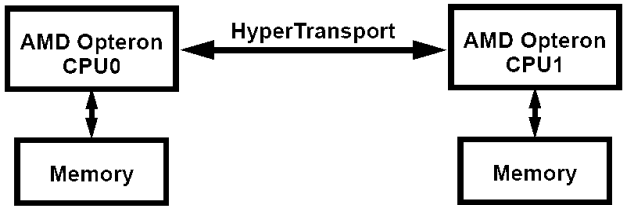


Abbildung 3: AMD Opteron NUMA-Architektur

Setzen AMD und Intel ihre Pläne um, wird man in Zukunft NUMA-Systeme mit mehreren zusammengeschalteten Multicoreprozessoren vorfinden. Beispielsweise zwei Vierkernprozessoren, die zu einem Achtkernsystem vereint werden. Ein solches System besitzt acht verschiedene Caches und zwei lokal getrennte Speicherbänke. Ein großes Problem zeigt sich im Bezug auf die Synchronisation gemeinsamer Daten: Wie lässt sich die Konsistenz der Daten effizient und mit möglichst wenig Leistungseinbußen erhalten?

4.3 Cache Kohärenz

Ein System, das zu jeder Zeit garantiert, den aktuellen Wert eines Datums zu beschaffen, auch wenn es sich in einem anderen Cache befindet, heißt cache-kohärent. Wieso ist Cache Kohärenz für uns wichtig? CAS und LL-SC basieren auf dem Vergleichen und Verändern von verteilt genutzten Speicherinhalten. Moderne Prozessoren halten Speicherinhalte aus Performancegründen im schnellen lokalen Prozessorcache („Memory Gap“). Dabei muss vor allem in Mehrprozessorsystemen sichergestellt sein, dass die Hauptspeicherkopien in den jeweiligen Caches der Prozessoren stets konsistent sind. Gerade bei lock- und wartefreien Algorithmen, die mit den vorgestellten Read-Modify-Write Befehlen gmeinsam genutzte Daten verarbeiten, beispielsweise bei einer CAS Operation keine inkonsistenter Cacheinhalt herangezogen werden. Die Sicherstellung der Cache Kohärenz ist somit essentiell für das korrekte funktionieren nicht-blockierender Algorithmen und gleichzeitig ein entscheidender Performancefaktor.

Ein Cache-Kohärenz-Protokoll hat also die Aufgabe, den Status eines gecachten Speicherblocks zu verfolgen

Für die Einhaltung der Cache Kohärenz haben sich zwei Verfahren entwickelt. Zum einen das sogenannte snoopingbasierte Verfahren und zum anderen das verzeichnisbasierte Verfahren. In den folgendenden Abschnitten werden beide Verfahren vorgestellt.

4.3.1 Snoopingbasierte Cache Kohärenz

Beim sogenannten „Bus Snooping“ verfolgen alle an einen gemeinsamen Datenbus angeschlossenen Cachecontroller die Lese- und Schreibzugriffe der einzelnen Knoten. Dabei überprüft jeder Cachecontroller, ob eine Cache-Zeile mit der entsprechenden Adresse vorhanden ist und modifiziert deren Zustand oder lädt beziehungsweise speichert die Daten im Hauptspeicher.

In den meisten aktuellen Prozessorsystemen wird dabei das sogenannte MESI-Protokoll (bzw. erweiterte Varianten) eingesetzt, das vier Zustände für jede Cachezeile definiert. Durch lokale und entfernte Zugriffe ausgelöste Zustandsübergänge, werden durch das Snooping festgestellt, da sie alle über einen gemeinsamen Bus übertragen werden. Folgende Abbildung zeigt den Zustandsautomaten des MESI-Protokolls. Die vier grundlegenden Zustände eines Cacheblocks werden im folgenden vorgestellt. Die genutzten Abkürzungen entsprechen denen aus Abbildung 4.

Modified: Eine Cachezeile, die als „modified" markiert ist, ist nur im aktuellen Cache vorhanden und „schmutzig“ („dirty“). Sie stimmt somit nicht mehr mir dem Wert im Hauptspeicher überein. Schreib- und Lesezugriffe des zugehörigen Prozessors können auf dem lokalen Cache ohne Buszugriffe vorgenommen werden (Self-Loop Übergänge RH, WH). Versucht ein anderer Prozessor das veraltete Datum aus dem Hauptspeicher zu lesen, erkennt dies der Cachecontroller durch das Snooping und kann den Zugriff mit einem Retry-Befehl unterbrechen. Er schreibt den modifizierten Wert zurück in den Hauptspeicher (Snoop Push) und gibt die Cachezeile weiter. Ihr Zustand ändert sich dabei auf „Shared“. Versucht ein anderer Prozessor schreibend zuzugreifen (Übergang SHW), wird die Zeile ebenso zurückgeschrieben, und daraufhin als ungültig („Invalid“) markiert.

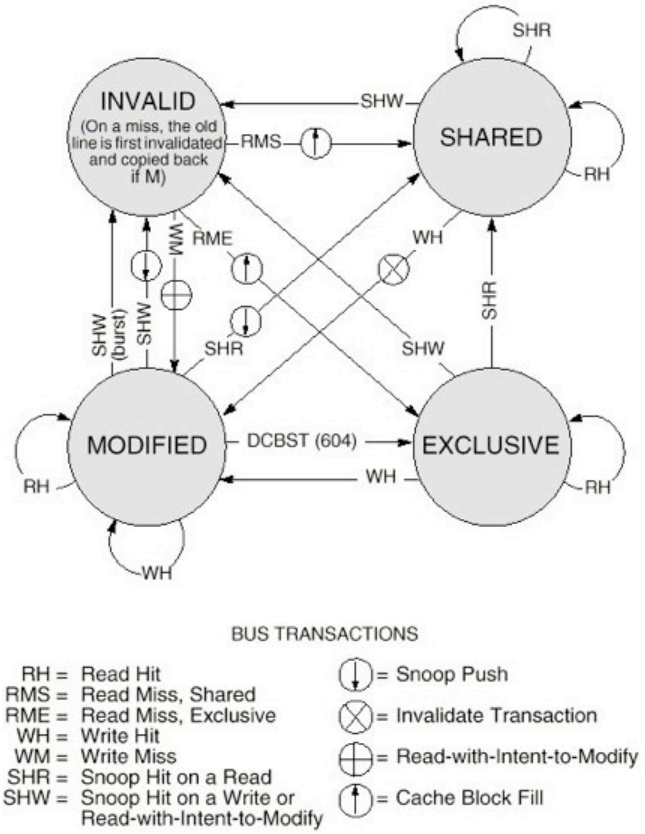


Abbildung 4: Zustandsautomat des MESI Protokolls

Exclusive: Die Cachezeile ist nur im aktuellen Cache vorhanden und nicht „schmutzig“, also noch konsistent mit dem Hauptspeicher. Der Prozessor hält den Block als einziger, exklusiv, in seinem Cache. Ein Lesezugriff (*RH*) kann direkt auf dem Cache vorgenommen werden, ohne auf den Bus zugreifen zu müssen. Bei einem Schreibzugriff ändert sich der Zustand zu „Modified“ (Übergang *WH*). Einen Schreibzugriff durch einen anderen Prozessor auf die zugehörige Hauptspeicheradresse (*SHW*) veranlasst den Cachecontroller die Cachezeile auf „Invalid“ zu setzen. Ein Lesezugriff durch einen anderen Prozessor (*SHR*) hingegen setzt den Zustand auf „Shared“, wobei das Datum direkt zwischen den Caches ausgetauscht werden kann. Es ist dadurch in beiden Caches vorhanden.

Shared: Ist ein Speicherblock als „Shared“ markiert, existiert er als unmodifizierte Kopie in der lokalen Cachezeile und in anderen Caches. Entfernte (*SHR*) und lokale (*RH*) Lesezugriffe ändern den Zustand nicht. Ein lokaler Schreibzugriff (*WH*) modifiziert die Zeile und alle anderen Cache-Kopien werden invalidiert. Entsprechend führt ein externer Schreibzugriff (*SHW*) zu einem Zustandwechsel zu „Invalid“.

Invalid: Eine als „Invalid“ markierte Zeile ist ungültig. Lesezugriffe veranlassen das Laden des entsprechenden Speicherblocks in die Cache-Zeile. Dabei zeigen die anderen Cachecontroller an, ob der Block bereits bei ihnen gespeichert ist, und übergeben ihn (*RMS*). Dies führt zu einer „shared“ markierten Zeile. Befindet sich der Speicherblock noch keinem Cache wird er „exclusive“ markiert geladen (*RME*). Der schreibende Zugriff (*WM*) führt dazu, dass die Speicherstelle geladen, entsprechend geändert und als „Modified“ markiert wird.

Das Ziel des MESI Protokolls ist die Minimierung der Buszugriffe auf den gemeinsam genutzten Hauptspeicher. Konkret sind die in der Abbildung die Zustandsübergänge zu sich selbst (Selfloops) Allerdings benötigt das Protokoll selbst auch Kommunikationsbandbreite. Bei wachsender Anzahl von Prozessoren beziehungsweise bei hohen Speicheranforderungen wird der zentrale Kommunikationskanal (z.B. Hypertransport bei AMD, Quick Path Interconnect bei Intel) zum Flaschenhals. Das snoopingbasierte Verfahren skaliert bei steigender Anzahl an Prozessoren eher schlecht.

Vor allem bei symmetrischen Multiprozessoren aber auch kleinen NUMA Architekturen haben sich snoopingbasierte Kohärenzprotokolle dennoch durchgesetzt. Durch die lokale Nähe der einzelnen CPU Kerne auf einem Chip beziehungsweise auf dem Mainboard lassen sich in Verbindung mit einem entsprechend leistungsfähigen Bus sehr schnelle Verbindungen zwischen den Caches und dem Speicher herstellen. Das snoopingbasierte Verfahren kann seine Vorteile nur bei Systemen mit einem zentralen Bus ausspielen, da hier jeder Buszugriff von jedem Teilnehmer gleichermaßen beobachtet werden kann.

Multiprozessorsysteme mit einer größeren Anzahl an Prozessoren, und damit vor allem Distributed Shared Memory Systeme nutzen keinen zentralen Bus, und damit auch ein anderes Kohärenzverfahren.

4.3.2 Verzeichnisbasierte Cache Kohärenz

Ziel eines Distributed Shared Memory Systems ist die Erhöhung der Speicher- und Kommunikationsbandbreite durch Verteilung des Hauptspeichers. Lokaler Speicherverkehr wird vom entfernten Zugriff getrennt was die Bandbreitenanforderungen an das Kommunikationssystems verringert. Jedoch kann ein Kohärenzprotokoll nicht snoopingbasiert umgesetzt werden, da die einzelnen Knoten nicht jeden Speichertransfer mitverfolgen können. Bei jedem Cache-Miss müsste eine Broadcastanfrage versendet werden und jeder angeschlossene Knoten müsste melden, ob er eine Kopie vorhält. Man geht hier einen anderen Weg.

Zu jedem Speicherblock wird in einem zentralen Verzeichnis festgehalten ob und in welchen Caches er vorliegt sowie sein jeweiliger Zustand. Bei einem Cache-Miss wird der zugehörige Eintrag aus dem Verzeichnis ausgewertet und falls nötig mit den Knoten kommuniziert, die eine Kopie des Speicherblocks im Cache halten. Es können dabei gezielt die Knoten angesprochen werden, die wirklich eine Kopie besitzen. Zwar lässt sich das System so sehr leicht skalieren, jedoch stellt der Verzeichniszugriff einen Flaschenhals dar. Der große Vorteil ist die gute Skalierbarkeit und der geringere Bandbreitenbedarf durch Punkt zu Punkt Kommunikation, nachteilig ist die erhöhte Latenz.

Für jeden Speicherblock müssen im Verzeichnis verschiedene Informationen festgehalten werden. Abbildung 5 zeigt den schematischen Aufbau eines Systems mit verzeichnisbasiertem Caching. Mehrere Prozessoren sind über ein Verbindungsnetzwerk an einen gemeinsamen Speicher angeschlossen. Die Struktur ähnelt stark der des SMP Systems aus Abbildung 2. Das Verbindungsnetzwerk ist hier jedoch kein gemeinsamer Bus, der ein Snooping-Verfahren ermöglichen würde, sondern eine andere Variante, die Punkt-zu-Punkt Verbindungen ermöglicht. Neben dem Hauptspeicher liegt das Verzeichnis, das wie der Hauptspeicher über das Verbindungsnetzwerk angesprochen wird.

Für jeden Prozessor im System wird ein Präsenzbit („presence bit“) gespeichert. Dadurch ist festgehalten, welche Prozessoren den Block nutzen, und gegebenenfalls benachrichtigt werden müssen. Außerdem muss pro Speicherblock ein „Dirty Bit“ gesichert werden, das anzeigt ob die Speicherzelle in einem der Caches aktualisiert wurde.

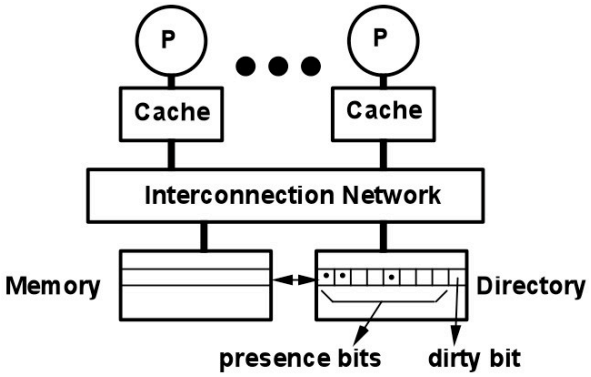


Abbildung 5: Verzeichnisbasierte Cachekohärenz

Die Zustandsinformationen der Cachezeilen selbst sind direkt in den Caches abgelegt, das Verzeichnis verwaltet somit nur die Existenz der Speicherzeile in einem Cache. Die Zustände ähneln denen des MESI-Protokolls, wobei der Zustand Exclusive herausfallen kann, da die Information wie viele Prozessoren einen Cacheblock teilen, schon im Verzeichnis enthalten ist. Die Zustandsübergänge entsprechen größtenteils denen des MESI Protokolls.

Bei einem lesenden Zugriff wird zunächst das „Dirty Bit“ ausgewertet. Ist es nicht gesetzt, wird die Speicherstelle geladen und das Präsenzbit des Prozessors gesetzt. Ist des dagegen gesetzt, wird der Prozessor, der die modifizierte Kopie vorhält, über den Präsenzvektor ermittelt. Die Cachezeile wird zurückgeschrieben, das „Dirty Bit“ kann zurückgesetzt werden und der Präsenzvektor vermerkt den hinzugekommenen Prozessor. Die Cachezeile wird bei den jeweiligen Prozessoren entsprechend als „shared“ markiert. Entsprechend wird beim Schreibzugriff ermittelt, ob und wo die Speicherstelle gecacht ist und Cacheinhalte und Invalidierungsnachrichten gezielt versendet werden.

Für die Kommunikation zwischen den Knoten wird über das Verzeichnis ermittelt, welche Knoten zu benachrichtigen sind. Diese können dann gezielt angesprochen werden, das restlichen Knoten werden nicht unnötig beansprucht. [7]

Das Verzeichnis selbst benötigt dabei nicht unerheblichen Speicherplatz, was ebenso zu einem Skalierungsproblem führen kann. Bei 64 Prozessoren benötigt man 8 Byte zur Speicherung des Präsenzbits. Bei einer Cacheblockgröße von 64 Byte ist dies ein Overhead von 12,5 %. Steigt die Prozessoranzahl auf 256 erhöht sich der Overhead gar auf 50 %. Um dies zu verbessern könnte man die Cacheblockgröße erhöhen. Ein anderer Lösungsansatz ist die Verteilung des Verzeichnisses auf mehrere Knoten, die sich dann gegenseitig mittels Zeigern referenzieren. [8]

5. Zusammenfassung

Entscheidend für lock- und wartefreie Algorithmen sind also effiziente atomare Synchronisationsbefehle wie Compare-And-Swap oder Load-Linked/Store-Conditional. Während diese auf Einprozessorsystemen als einfache atomare Prozessorbefehle implementiert werden können, ist die Umsetzung in komplexen Mehrkern- und Mehrprozessorsystemen erheblich aufwändiger. Das Vorhandensein einzelner lokaler Caches in den Pro zessoren fordert erhöhten Aufwand zur Einhaltung der Kohärenz der jeweiligen Cachezeilen beziehungsweise die Konsistenz der Daten im Hauptspeicher. Dies führt außerdem dazu, dass der Load-Linked/Store-Conditional Befehl nur in einer schwachen Variante umgesetzt werden kann. Es gibt keine reale Implementierung, die eine Reservierung wirklich nur ungültig macht, wenn die Speicherstelle tatsächlich überschrieben wurde. Gäbe es diese Einschränkung nicht, könnte man mit LL/SC die Compare-And-Swap Operation lockfrei nachbilden, und dabei sogar deren ABA-Problem lösen.

Zur Einhaltung der Cachekohärenz haben sich zwei Verfahren herauskristallisiert. Das snoopingbasierte Verfahren und das verzeichnisbasierte Verfahren. Ersteres wird vor allem in verhältnismäßig kleinen Mehrkernprozessoren eingesetzt, die meist eine Art gemeinsamen Datenbus aufweisen (AMD Hypertransport, Intel Quick Path Interconnect, Speicherbus). Das verzeichnisbasierte Verfahren hingegen wird vorzugsweise bei komplexeren Distributed Shared Memory Systemen eingesetzt, da hier meist eine Verbindungsstruktur zwischen den Knoten vorhanden ist, die Bus Snooping nicht ermöglicht. Außerdem skaliert das snoopingbasierte Verfahren mit steigender Prozessoranzahl nur sehr schlecht, weshalb man hier ebenso zur verzeichnisbasierten Lösung greifen muss.

Hingegen ermöglichen Multicore Chips, die mehrere Prozessoren auf einen IC vereinen, schnelle und effiziente snoopingbasierte Verfahren, solange die Bandbreite des zentralen Busses groß genug ist um das erhöhte Nachrichtenaufkommen zu tragen.

LITERATUR

[1] M. Michael and M. Scott, "Implementation of Atomic Primitives on Distributed Shared Memory Multiprocessors", Proceedings of the 1st Annual Symposium on High Performance Computer Architecture, 1995, pp. 221-231.

[2] K. Fraser. „Practical lock-freedom“. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[3] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free Dynamically Resizable Arrays, Texas A&M University

[4] Maurice P. Herlihy. „Impossibility and universality results for wait-free synchronization“ In Proceedings of the 7th ACM Symposium on Principles of distributed computing, 1988

[5] Jonathan Rentzsch, „Save your code from meltdown usding PowerPC atomic instructions“, IBM Technical Report http://www.ibm.com/developerworks/library/pa-atom/

[6] SGI Origin 3400 am RRZE http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/sgi-origin-3400.shtml

[7] A. Jantsch, „System on Chip Architectures – Cache Coherency Protocols“ http://www.ict.kth.se/courses/IL2207/0810/Lectures/L04-CacheCoherency-Oct2008.pdf

[8] Gerndt, Scalable Shared Memory Systems, http://www.lrr.in.tum.de/~gerndt/home/Teaching/scalable_shared_memory_systems/Kapitel6.pdf

[9] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. ACM Transactions on Programming Languages and Systems, 15(5):745-770, November 1993