Cache Kohärenz

von
Rainer Leisen

Vorwort

Diese Seminararbeit Cache Kohärenz wurde parallel zu der Veranstaltung „Verteilte und Parallele Systeme II der Fachhochschule Bonn-Rhein-Sieg“ erstellt. In diesem Dokument wird auf die Problematik der Cache Kohärenz und der verwendeten Protokolle zur Handhabung der Caches bei Mehrprozessor- bzw. Mehrrechnersystemen eingegangen. Dieses Dokument erfordert ein allgemeines Wissen über Rechner- und Cachearchitekturen.

Inhaltsverzeichnis

	Seite
1. Einleitung	2
2. Central Directory	3
3. Snooping Caches	4
4. Cache Kohärenzprotokoll: MESI	5
5. Cache Kohärenzprotokoll: MOESI	8
6. Directory Based	9
7. Zusammenfassung	10
8. Literatur	10

1. Einleitung

Parallelrechnerarchitekturen werden in zwei Kommunikationsklassen aufgetrennt:
Die Mehrprozessorsysteme, bei denen sich mehre Prozessoren einen gemeinsamen Speicher teilen und die Mehrrechnersysteme, bei dem jeder Rechner einen eigenen Speicher besitzt. Ein Mehrrechnersystem benutzt ein Verbindungsnetz, um Nachrichten zwischen den einzelnen Rechnern zu versenden.

Eigentlich sollte die Verwaltung eines gemeinsamen Speichers nicht besonders problematisch sein, da man mit Hilfe einer FIFO-Queue vor dem Speichern einen sequenziellen Zugriff und damit eine zuverlässige Kontrolle über die einzelnen Adressblöcke gewährleisten könnte. Dieses würde aber zu einem immensen Flaschenhals zwischen CPU und Speicher führen, der die Leistung der Prozessoren stark ausbremsen würde. In der ersten Iteration bekamen die Prozessoren einen eigenen kleinen Speicher (Cache) zugewiesen auf den sie den alleinigen Zugriff hatten. Moderne Systeme besitzen heutzutage zusätzlich noch einen zweiten Cache. In den folgenden Kapiteln wird näher darauf eingegangen. Diese Architektur, mit einem eigenen Speicher zu jedem Prozessor wiederspricht eigentlich der Definition der Mehrprozessorsysteme, da zu dem gemeinsamen Hauptspeicher nun auch kleine verteilte Speicher hinzu kommen.
Da die Caches meist nur einen Bruchteil der Größe eines Hauptspeichers besitzen, muss eine intelligente Verwaltung zwischen die beiden Speichermedien geschaltet werden, um einen zuverlässigen Abgleich der beiden Komponenten gewährleisten zu können.

Verbotene Zustände zwischen den Speichermedien wären zum Beispiel:

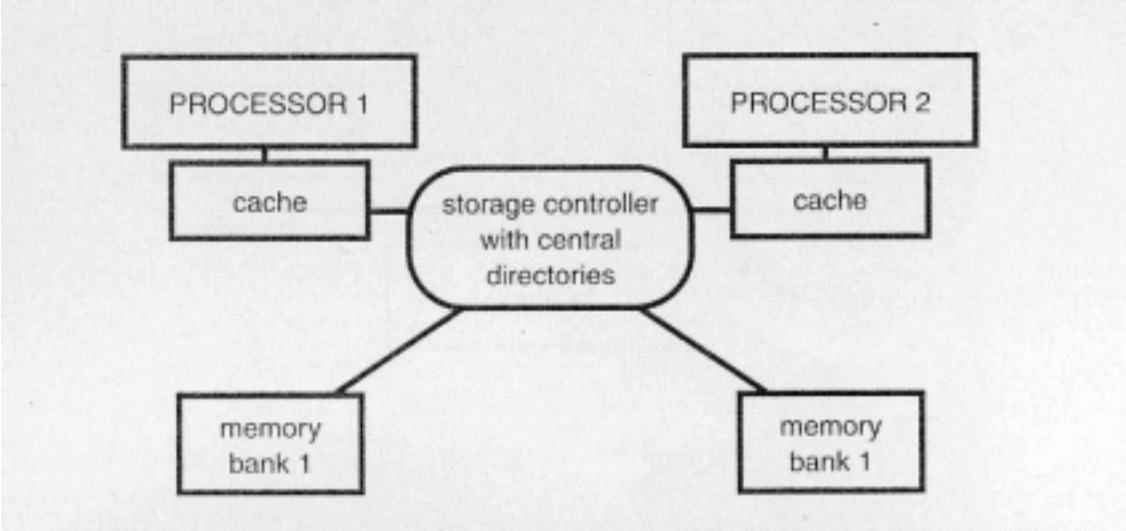
1. Eine CPU „0“ möchte die Adresse ff0 auslesen und findet diese auch in ihrem Cache mit dem Wert A. In der Zwischenzeit hat CPU1 dieselbe Zeile in ihrem Cache gespeichert und diese Adresse mit dem Wert B überschrieben. Die CPU0 hat damit die Änderung der Speicheradresse nicht mitbekommen und den falschen Wert ausgelesen.

2. Die CPU0 und CPU1 haben beide die Adresse der globalen Variabel "i" in ihrem Cache gespeichert. Jede CPU erhöht den Wert auf einen verschiedenen Wert. Danach will jede CPU den Wert in den Speicher schreiben. Frage: Ist der Wert im Speicher jetzt korrekt?

Diese Probleme werden auch als Cache Kohärenz bezeichnet und mit den sogenannten Cache Kohärenzprotokollen verhindert. Diese Zustände können beim Schreiben in den gemeinsamen Speicher, bei der Migration von verschiedenen Prozessen und bei der Ein- und Ausgabe von Daten auftreten. Einige dieser Protokolle werden in den nachfolgenden Kapiteln behandelt.

2. Central Directory

Die nun folgende Methode, um die Cache Kohärenz zu Verwalten, ist veraltet und wird in aktuellen Systemen nicht mehr eingesetzt. Daher folgt nur eine kurze Funktionsbeschreibung, die aber die Problematik der Cache Kohärenz schon etwas vertiefen soll.
Bei dem „Central Directory“ Ansatz wird mit einer zentralen Liste gearbeitet, die alle Einträge der verschiedenen Caches beinhaltet. Eine vereinfachte Darstellung finden Sie in folgender Abbildung:



In dieser Liste sind alle möglichen Tupel von n * Caches zu der momentan sich im Besitz befindenden Zeile abgebildet. Da diese Liste von jedem Prozessor ständig abgefragt wird, muss sie eine extrem kurze Antwortzeit garantieren, da sie sonst zum Flaschenhals zwischen Prozessor und Speicher führen würde. Die hardwaremäßige Umsetzung solcher Verzeichnisstrukturen ist sehr kostspielig.
Zur Realisierung muss ein zusätzliches Verbindungsnetz zwischen den einzelnen Caches und dem Verzeichnis geschaltet werden. Dies ist nicht auf die Datenmenge, sondern auf Geschwindigkeit optimiert, da nur nach Referenzadressen in den Tabellen gesucht wird, um die Lokalität zu berechnen.
Vereinfachtes Beispiel:
Speicherbereich A ist in der Tabelle von Prozessor 1 gespeichert. Fordert der Prozessor 2 den Speicherbereich A an, so wird der Inhalt in den Cache von Prozessor 2 geschoben und die Tabellen von Prozessor 1 und 2 im Verzeichnis aktualisiert.
Folgende Fälle können bei steigender Last zu Problemen führen:
Treten bei den Prozessoren am Anfang viele MISS-Aktionen (Speicherbereich konnte nicht im eigenen Cache gefunden werden) gleichzeitig auf, so werden viele Anfragen gleichzeitig an das Verzeichnis gestellt.
Wandert eine Cachezeile von Prozessor 1 nach Prozessor 2, so muss der Verzeichnisdienst erst die Zeile von Cache 1 anfordern, warten bis Cache 1 diese Anforderung bedient hat, um sie dann an Cache 2 zu senden. Zudem müssen anschließend noch die internen Listen gepflegt werden.
Später wurde diese Technik mit der Verwendung von Kopien und deren Verwaltung zur Optimierung erweitert. Mehr über diese Technik erfahren sie in dem Kapitel 6, in dem eine erweiterte Technologie beschrieben wird.

3. Snooping Caches

Heutzutage wir anstelle des gemeinsamen Verzeichnisdienstes eine Hardware pro Cache eingesetzt, die den Bus nach Cacheanfragen von anderen Prozessoren „beschnüffelt“ (Snooping).

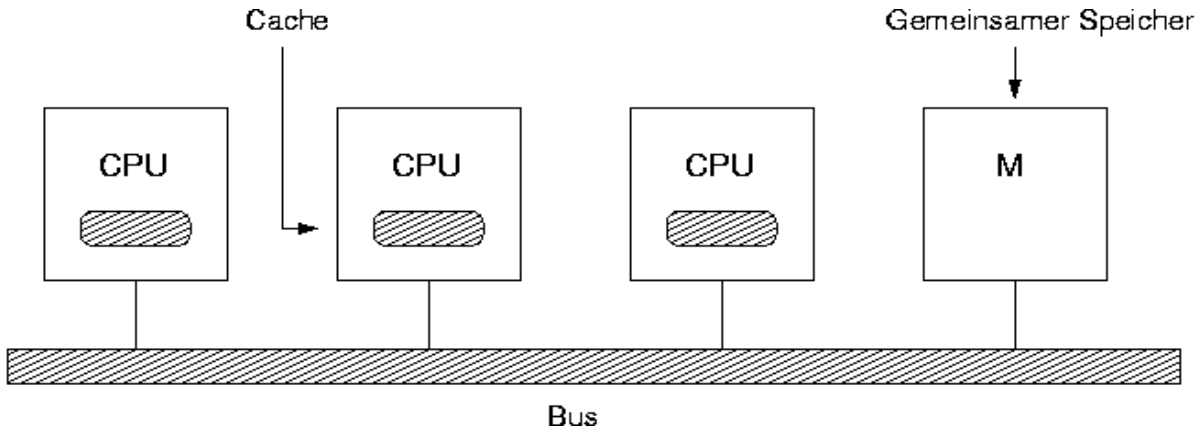


Bild: CPU mit Caches, die über einen gemeinsamen Bus auf den Speicher zugreifen. Der Bus wird von jedem Cache „beschnüffelt“

Diese Informationen werden dann von verschiedenen Protokollen ausgewertet und die Zeilen anschließend im Cache aktualisiert. Folglich wird die Funktion an Hand des „Write-Through Protokolls“ genauer beschrieben, welches zu den einfacheren Protokollen zählt.
Man Unterscheidet bei diesem Protokoll zwischen vier Auslösern- und Aktionstupel:

Auslöser	Aktion	Entfernte Aktion
Lesefehlschlag	Hole Daten aus dem Speicher	
Lesetreffer	Benutze Daten aus dem lokalen Cache	
Schreibfehlschlag	Aktualisiere Daten im Speicher	
Schreibtreffer	Aktualisiere Cache und Speicher	Invalidiere Cache-Eintrag

Ein Lesefehlschlag tritt dann auf, wenn das angeforderte Wort von der CPU noch nicht im Cache ist. Als Aktion wird dann die passende Zeile aus dem Speicher in den Cache geladen. Bei einem erneuten Versuch der CPU auf das Wort zuzugreifen, tritt ein Lesetreffer ein, das heißt die Zeile kann aus dem lokalen Cache gelesen werden. Soll ein Wort geschrieben werden, das nicht im Cache vorhanden ist, so tritt ein Schreibfehlschlag auf und das geänderte Wort wird in den Arbeitsspeicher geschrieben. Die referenzierte Zeile wird bei dieser Aktion nicht automatisch in den Cache geladen. Ist die Zeile aber im Cache vorhanden (Schreibtreffer), so werden Cache und Arbeitsspeicher aktualisiert. Dieses Protokoll gewährleistet, dass Änderungen an einem Wort auch immer im Speicher stehen.
Was müssten zwei Snooping Caches (A und B) gewährleisten, um die Cache Kohärenz zu garantieren?
Bei einem Lesefehlschlag oder Lesetreffer von Cache A muss Cache B nicht reagieren, da nur lesend auf Speicherbereiche zugegriffen wird. Einen Lesetreffer von A bekäme B nicht mit.
Eine Schreiboperation benötigt etwas mehr Aufwand. Egal ob Cache A beim Schreiben eines Wertes einen Schreibfehlschlag oder eine Schreibtreffer ausgelöst hat, Cache B bekommt diese Aktion mit, da es auf jeden Fall über den Speicher geht. Cache B vergleicht diese Zeile mit den entsprechenden Einträgen bei sich. Falls diese Zeile nicht in seinem Cache stand, dann ist für Cache B diese Operation von A bedeutungslos. Sollte diese Zeile aber nun auch im Cache B vorhanden sein, so muss er diese Zeile aus seinem Cache entfernen oder sie als ungültig markieren, da diese nicht mehr dem aktuellen Wert im Speicher entspricht. Cache B steht es frei, ob er den aktuellen Wert sofort aus dem Speicher liest oder erst bei einem Lesefehlschlag in seinem Knoten.

Von diesem Basisprotokoll gibt es verschiedene Varianten, die klassifiziert werden in:
Invalidierungsstrategien:
Bei einem Schreibtreffer invalidiert der Cache seinen Eintrag.
Aktualisierungsstrategien:
Der Wert eines Schreibtreffers wird direkt in dem eigenen Cache aktualisiert. Dieses Verfahren benötigt den Transport der Nutzdaten, was größere Anforderungen an die Bandbreite stellt.
Eine weitere Variante ist das „Write-Allocate Policiy Verfahren“, wo bei einem Schreibfehlschlag die geschriebene Zeile zusätzlich in den Cache-Speicher geladen wird.
Diese Lösungsansätze haben aber immer noch das Problem, dass alle Schreiboperationen über den gemeinsamen Bus gehen und somit der Bus zum Engpass des Systems werden kann.
In Kapitel 4 und 5 werden erweiterte Protokolle beschrieben, die diesen Flaschenhals verringern.

4. Cache Kohärenzprotokoll: MESI

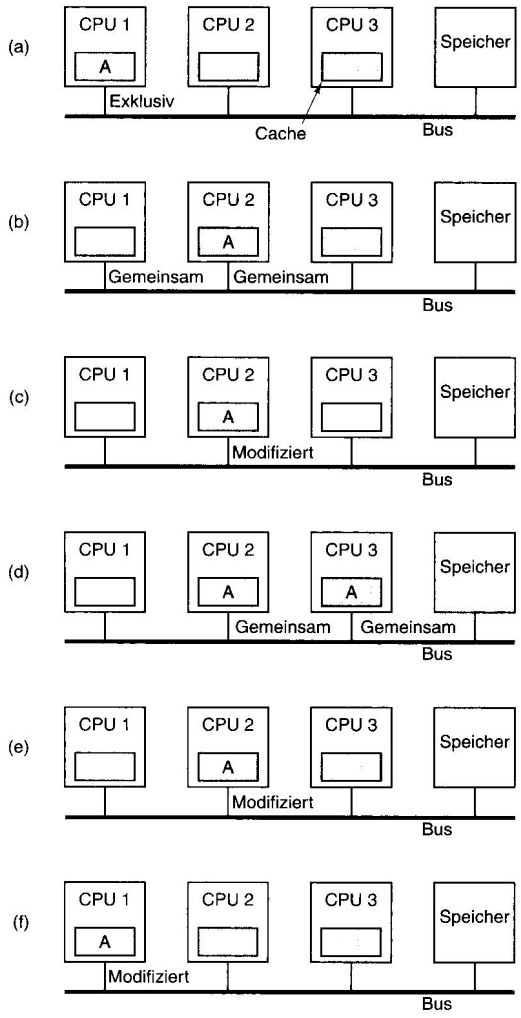
MESI ist ein sogenanntes Write-Back Protokoll, das auf früheren Write-Once Protokollen basiert. Im Gegensatz zu den einfacheren Snooping-Protokollen, wie in Kapitel 3 beschrieben, werden nicht nur zwei Zustände gespeichert, sondern vier:

MESI = modified, exclusive, shared, invalid

Zustand:	Bedeutung :
Modified	gültiger Wert, nur in diesem Cache vorhanden, gegenüber Hauptspeicherwert verändert
exclusive	gültiger Wert, nur in diesem Cache vorhanden
shared	gültiger Wert, in mehreren Caches vorhanden
invalid	Wert ist ungültig (z. B. noch nie geladen)

Dafür werden alle Cache- Speicherstellen mit zwei Statusbits erweitert, die die oben erwähnten Zustände speichern können. Damit alle Zugriffe der anderen Prozessoren ausgewertet werden können, wird auch hierbei die Snooping Technologie eingesetzt, um den Bus abzuhören. Im Gegensatz zu den einfacheren Caches können die Technologien, die das MESI-Protokoll unterstützten, auch Messages senden.

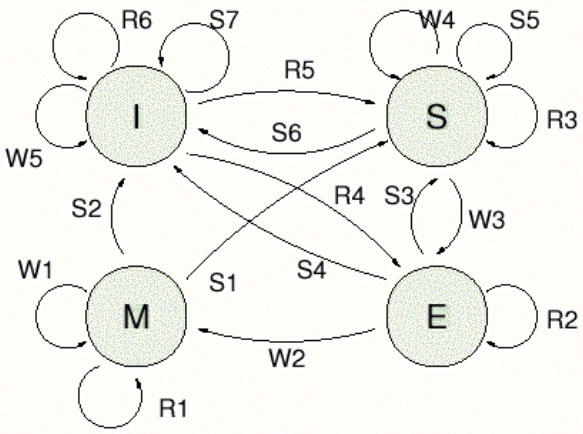
An Hand der Abbildung auf der nächsten Seite, werden verschiedene Zustände beschrieben:



a)
CPU 1 fordert eine Zeile A aus dem Speicher an, legt sie in ihrem Cache ab und markiert diese mit der Bitkombination für Exclusive.
b)
CPU 2 fordert ebenfalls aus dem Speicher die Zeile A an. Durch „schnüffeln“ auf dem Bus merkt sie, das sie nicht die alleinige Besitzerin dieser Cachezeile ist. Folglich markiert sie ihre Zeile mit dem Bits für shared und weißt CPU 1 an, dies auch zu tun.
c)
CPU 2 ändert den Wert von A im Cache und damit auch den Status („Modified“). Über den Bus wird für Zeile A ein „invalid“-Signal gesendet, so dass alle CPU's, die diese Zeile besitzen, aufgefordert werden ihren Status für diese Zeile in Invalid zu ändern. Der Wert wird jedoch noch nicht im Speicher aktualisiert.
d)
CPU 3 fordert jetzt die Zeile A an. CPU 2 bekommt dies mit und fordert CPU 3 auf zu warten, bis diese den aktuellen Wert aus dem Cache in den Speicher geschrieben hat. Danach kann CPU 3 problemlos den aktuellen Wert aus dem Speicher lesen und diesen mit dem Zustand shared im Cache ablegen. CPU 2 ändert ihren Status für Zeile A ebenfalls noch in shared um.
e)
CPU 2 ändert den Wert von Zeile A. Somit wird ihr Eintrag in modifiziert und der von CPU 3 in invalid gesetzt.
f)
CPU 1 möchte den Wert von Zeile A ändern, bekommt aber vorher von CPU 2 das Signal zu warten, bis CPU 2 den Wert von Zeile A in den Speicher geschrieben hat.

Alle MESI Zustände noch mal im Überblick

MESI	Cache Eintrag Zustand gültig ?	Wert im Speicher gültig?	Kopien in andere Caches?	Zugriff betrifft
M	ja	nein	nein	Cache
E	ja	ja	nein	Cache
S	ja	ja	möglich	Speicher
I	nein	unbekannt	möglich	Speicher



Snoop-Zyklen

M-S	S1	Hit, Speicher schreiben
M-I	S2	Hit, Speicher schreiben
E-S	S3	Hit, aber nicht modifiziert
usw.		

Lesezugriffe:

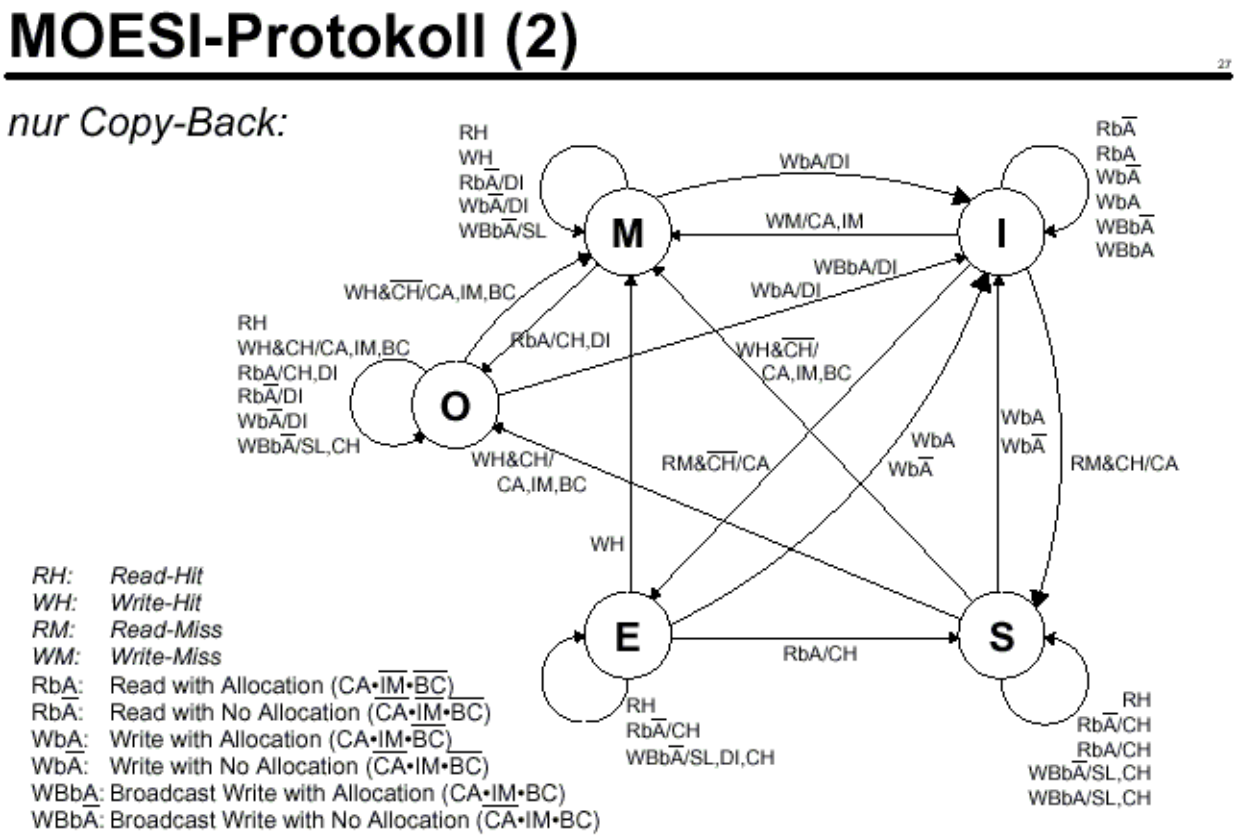
M-M	R1	Cache-Hit, CPU bekommt Daten
E-E	R2	Cache-Hit, CPU bekommt Daten
S-S	R3	Cache-Hit, CPU bekommt Daten
I-E	R4	Miss, Speicher liefert Daten
I-S	R5	Miss, externer Cache liefert Daten
I-I	R6	Miss, Adresse nicht cacheable

Schreibzugriffe:

M-M	W1	Hit, CPU aktualisiert Cache
E-M	W2	Hit, CPU aktualisiert Cache
S-E	W3	Hit (write-back): Cache aktualisiert, Buszyklus markiert fremde Kopien als invalid
S-S	W4	Hit (write-through): Caches und Speicher aktualisiert
I-I	W5	Miss, Speicher schreiben, aber kein write-allocate

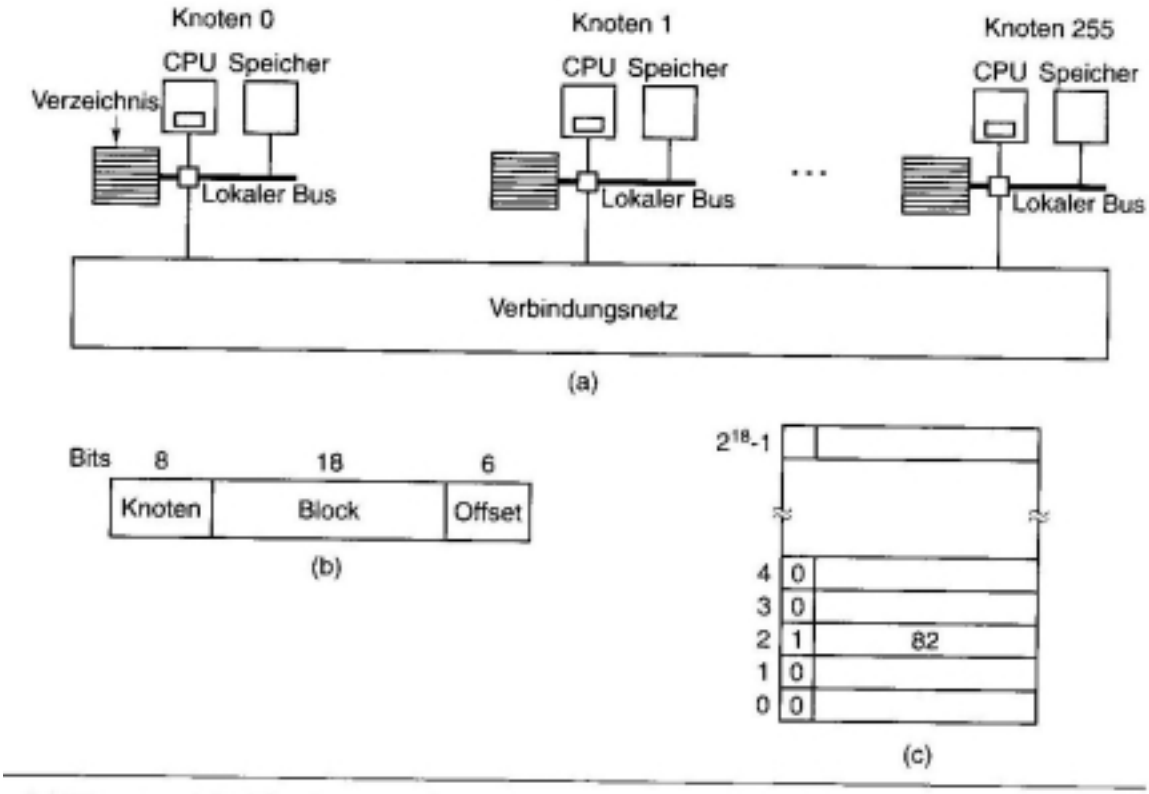
5. Cache Kohärenzprotokoll: MOESI

Das MOESI Protokoll ist eine erweiterte Variante des MESI Protokolls. Neben den vier Zuständen einer Cachezeile Modified, Exclusive, Shared und Invalid wurde für MOESI der Owner-Status hinzugefügt.
Die Funktion des Cache-Sharings basiert auf folgendem Prinzip:
CPU 0 ist Owner einer Cache Zeile. Erfolgt ein Lesezugriff von CPU 1 auf diesen Speicherbereich, leitet der Chipsatz die Anfrage direkt an den schnellen Cache von CPU 0 weiter. Ein Zugriff auf den langsamen Hauptspeicher bleibt aus. Damit sinkt nicht nur die Antwortzeit für den Lesezugriff, gleichzeitig ist der Speicherbus entlastet und für andere Zugriffe frei.



6. Directory Based

Die im folgenden beschriebene Technologie, um die Cache Kohärenz zu verwalten, wird heutzutage nur noch in den sogenannten CC-NUMA (Cache Coherent Non-Uniform Memory Access) Systemen eingesetzt. Sie setzt auf das „Central Directory“-Verfahren.



(a) Ein verzeichnisbasiertes Mehrprozessorsystem mit 256 Knoten; (b) Aufteilung einer 32-Bit-Speicheradresse in Felder; (c) das Verzeichnis von Knoten 36

Im Gegensatz zu den vorherigen Systemen ist der Speicher nun physikalisch verteilt, wird aber zu einem großen virtuellen Speicher zusammengefasst. Jeder Knoten verfügt über ein Verzeichnis, in das er seinen gesamten Speicher abgebildet hat. Kann nun eine Speicheranfrage von einem Knoten nicht selbst beantwortet werden, so wird diese Zeile vom Heimatknoten angefordert. Ist eine Zeile an einen anderen Knoten gesendet worden, dann wird diese Information im eigenen Verzeichnis mit einem Tag1 für invalid gekennzeichnet und die Angaben zum jetzigen Knoten, in diesem Fall 82, werden gespeichert. Wenn diese Zeile 2 jetzt von einem dritten Knoten angefordert wird, so würde folgendes umgesetzt:
Der erste Aufrufer würde aufgefordert die Zeile an den Dritten zu senden. Im eigenen Verzeichnis würde die Zeile 2 auf den aktuellen Knotenwert des Dritten geändert.
Wie man sieht, ist ein umfangreicher Nachrichten- und Datenaustausch notwenig, um auf Zeilen zugreifen zu können, die nicht im eigenen Knoten liegen.
Später wurde diese Technologie noch mit dem Snooping Verfahren erweitert, so dass hybride Systeme wie: DASH (Directory Architecture for SHared Memory) entwickelt wurden. Diese wurden dann zum Beispiel bei der SGI Orgin angewendet.

8 Zusammenfassung

In der Tabelle sind verschiedene aktuelle Prozessoren und Systeme mit den dazu verwendeten Cache Kohärenzprotokollen aufgelistet:

Prozessoren	***Protokollarten***
Solaris UltraSparc II	MOESI
Intel Prozessoren	MESI
Athlon MP	MOESI
Cray T3E; Prozessor Alpha 21264 (NUMA)	lokal
SGI Origin; Prozessor MIPS R10000 (ccNUMA)	Directory Based
IBM Power3 und Power4	MESI

Momentan sieht es so aus, als könnte sich das von Athlon und Solaris benutzte Protokoll MOESI auf dem Markt durchsetzen. Da aber die Verbindung zwischen den Speichern, Caches und Prozessoren immer noch eine der größeren Einschränkungen auf die gesamte Leistung eines Systems ist, ist abzuwarten ob das MOESI Protokoll der Stein der Weisen ist. Mit dem MOESI Protokoll nähern sich die Mehrprozessorsysteme immer mehr der Mehrrechnersystem-Architektur an.

9. Literatur:

Gregory F. Pfister: „In Search of Clusters“,Prentice Hall, 1998
Andrew s Tanebaum, James Goodman: „Computerarchitektur“, Pearson Studium,2001
pc-technologie :“smp“,http://tech-www.informatik.uni-hamburg.de/lehre/pc-technologie/05-smp.pdf, 2001
juergen Arndt:“Multiprozessorsysteme“,http://juergen.letzte-bankreihe.de/ps/#multiprozessorsysteme,2001
Dipl.-Inf. Titus Weber.Seminar: „Rechnersysteme“, http://www.isis.de/members/~tweber/rs_semi/4/rs_4.htm,1997
Bernd Schürmann: „Cache- Kohärenz in Multiprozessorsystemen“,
http://www.uni-paderborn.de/cs/heiss/lehre/ws96/projektgruppe/cache_coherence.ps.gz,1997

sonstige Quellen:
www.amd.com
www.intel.com
www.sun.com
www.ibm.com