Rechnerarchitekturen für Parallele und Verteilte Systeme

2

Zur Erhöhung der Rechenleistung durch parallele Auslegung und Vervielfachung der Prozessoren kristallisieren sich heute vier Möglichkeiten auf unterschiedlichen Rechnerarchitekturen heraus:

1. ***Eng gekoppelte Multiprozessoren*** und ***Multicore-Prozessoren***
   Eine Möglichkeit, die Verarbeitungsgeschwindigkeit von Prozessoren zu erhöhen, ist die Koppelung von mehreren Prozessoren. Auf diese Weise kann ein erhöhter Systemdurchsatz erreicht werden, wenn verschiedene Prozesse oder Threads echt parallel auf verschiedenen Prozessoren ausgeführt werden und nicht quasi parallel (durch Prozessumschaltung), wie bei Einprozessorsystemen.
   Ein erhöhter Systemdurchsatz ist vor allem bei parallelen Servern erwünscht, die für jede eingehende Anfrage (Request) einen Thread zur Bearbeitung der Anfrage starten. Dies bewirkt dann beim Server eine Erhöhung der Anzahl der zu verarbeitenden Anfragen pro Zeiteinheit. Beim ***eng gekoppelten Multiprozessor*** (***tightly coupled***), nutzen alle CPUs den Hauptspeicher gemeinsam. Die Synchronisation, Koordination und Kommunikation der parallelen Prozesse auf den verschiedenen CPUs geschieht über den gemeinsamen Speicher. Die einzelnen Prozessoren können ganz einfach in den gemeinsamen Speicher lesen und schreiben (siehe Abschn. 2.1).
2. ***General Purpose Computation on Graphic Processing Unit*** (***GPGPU***) und massive parallele Architekturen
   Eine ***Graphic Processing Unit*** (GPU) vor dem Jahr 2000 war eine Ansammlung von in Hardware gegossenen festen graphischen Funktionen und somit Prozessoren, wie z. B. vertex-spezifischen Funktionen, pixel-spezifische Berechnungen und Rasterberechnungen bis hin zu Shader-Funktionen. Durch den Weggang von in Hardware gegossenen festgelegten speziellen Funktionen und deren Ersatz durch einfache und beschränkt programmierbare Ausführungseinheiten, die auf Fließkommaoperationen spezialisiert sind, erlaubt die Berechnungen frei zu programmieren und führt somit zu einer flexibleren Ausnutzung der Hardware. Grafische Berechnungen und insbe-

© Springer Fachmedien Wiesbaden 2015

G. Bengel et al., *Masterkurs Parallele und Verteilte Systeme*,
DOI 10.1007/978-3-8348-2151-5_2

sondere 3D-Berechnungen lassen sich hervorragend parallelisieren. Zur Unterstützung der parallelen Verarbeitung bestehen GPUs aus bis zu Tausenden CPU-Kernen. Durch die hohe Anzahl von Kernen spricht man hier von massiv parallelen Architekturen [KH 12].
Nach der Flynn'schen Taxonomie handelt es sich dabei um Single Instruction und Multiple Data (SIMD)-Architekturen. Das heißt eine Vielzahl von CPUs führen den gleichen Code auf unterschiedlichen Daten aus. Die GPUs werden dadurch zu ***Streamprozessoren***, die Datenströme verarbeiten können (siehe Abschn. 2.2).

3. ***Many-Core-Prozessoren*** und ***Tile-Architekturen***
   Mehrere Prozessoren liegen dabei in einer ***Kachel*** (***Tile***). Jede Tile entspricht einem eng gekoppelten Multiprozessor oder Multicore-Prozessor. Jede Tile ist ein vollwertiges Rechensystem auf dem unabhängig von anderen Tiles ein Betriebssystem (Linux, Windows) läuft. Jede Tile ist zudem eine Frequenzinsel, die durch Software kontrollierbar ist. Mehrere Tiles zusammen bilden eine durch Software kontrollierbare ***Spannungsinsel***.
   Jede Tile besitzt einen Router. Der Router ermöglicht mehrere Tiles zu einem zweidimensionalen Gitter (2D Kommunikationsnetzwerk) zusammenzuschalten (siehe Abschn. 2.3). Anstatt eines Routers kommen auch Kreuzschienenschalter zum Einsatz für die vier Verbindungseinrichtungen Nord, Süd, Osten und Westen.
4. ***Lose gekoppelte Multiprozessoren*** und ***Cluster***
   Bei ***lose gekoppelten Multiprozessoren*** (loosly coupled) besitzt jeder Prozessor seinen eignen Speicher und es ist kein gemeinsamer Speicher vorhanden. Jeder Knoten ist eine selbstständige Recheneinheit mit einem eigenen Betriebssystem das über eine Verbindungseinrichtung oder lokalen Netz mit den anderen Recheneinheiten verbunden ist. Die Synchronisation, Koordination und Kommunikation der parallelen Prozesse, die auf den verschiedenen Prozessoren ablaufen, ist nur durch Nachrichtenaustausch zu bewerkstelligen, da kein gemeinsamer Speicher existiert (siehe Abschn. 2.4).
   Solche Rechnerverbünde dienen zur Erhöhung der Leistung (***High Performance Computing (HPC-)Cluster***) oder zur Erhöhung der Verfügbarkeit (***High Availibilty (HA-) Cluster***) oder einer Kombination von beidem.

2.1 Eng gekoppelte Multiprozessoren und Multicore-Prozessoren

Zur Erbringung der großen Rechenlast des Servers muss der Server parallel mit Prozessen, oder durch mehrere Threads, die unter dem Serverprozess laufen, ausgelegt werden. Zur Erreichung von einer Vielzahl von Threads können die Threads explizit vorgegeben sein oder man setzt parallelisierende Compiler ein, welche die Anwendung in parallele Threads aufteilt. Zum Ablauf des parallelen Servers steht die Möglichkeit

- der parallelen Abarbeitung der Threads auf Hardwareebene (***Simultaneous Multithreading***) oder

- der Einsatz eines ***eng gekoppelten Multiprozessorsystems*** oder eines ***Multicore-Prozessors*** mit gemeinsamem Speicher zur Verfügung.

2.1.1 Simultaneous Multithreading

Simultaneous Multithreading [EEL 97] bezeichnet die Fähigkeit eines Mikroprozessors, mittels getrennter Pipelines und zusätzlicher Registersätze mehrere Threads gleichzeitig auszuführen. Die wohl bekanntesten Prozessoren, die Simultaneous Multithreading realisieren, sind der Intel Pentium 4 und der Intel Xeon. Die von Intel vergebene Bezeichnung lautet ***Hyper-Threading Technology*** [M 02] mit der Abkürzung HT-Tech oder HTT.

Eine Leistungssteigerung eines einzelnen Prozessors lässt sich durch parallele Ausführung der Befehle erreichen. Parallelisieren kann nun auf folgenden Ebenen stattfinden:

1. Auf Instruktions- oder Befehlsebene (***Instruction Level Parallelism***) durch Abarbeitung mehrerer Instruktionen in einem Takt.
2. Auf Thread-Ebene (***Thread Level Parallelism***) durch paralleles Abarbeiten von Befehlen aus mehreren Threads.
3. Durch Zusammenfassung der Instruktions-Ebene und Thread-Ebene zum ***Simultaneous Multithreading***.

2.1.1.1 Instruction Level Parallelism

Befehle, die voneinander unabhängig sind und aus einem überschaubaren, logisch sequenziell angeordneten Programmausschnitt stammen, lassen sich parallel ausführen, sofern zusätzliche Funktionseinheiten und Datenregister zur Verfügung stehen. Diese Technik heißt ***Superskalarverarbeitung*** oder ***Superskalarität***.

Die ***In-Order-Execution*** [M 01] startet die Befehle aus mehreren nebenläufigen Pipelines in ihrer logischen Reihenfolge. Die Phasenpipeline beginnt häufig mit einer oder mehreren gemeinsamen Stufen (z. B.: Befehle holen, Befehle vordekodieren) und gabelt sich anschließend in mehrere nebenläufige Teilpipelines auf. Gleichzeitig oder nacheinander gestartete Befehle können sich dabei nicht überholen (***in-Order***). Datenabhängigkeiten zwischen den Befehlen, die sich gleichzeitig in den Pipelines befinden, werden dadurch aufgelöst, dass die Pipelines mit den logisch später folgenden Befehlen so lange angehalten werden, bis die logisch früher ankommenden Befehle ihre Ergebnisse zurück geschrieben haben. Ergebnisse werden also auch immer in der logischen korrekten Reihenfolge zurück geschrieben (***In-Order-Completion***).

Die ***Out-of-Order-Befehlsverarbeitung*** [M 01] verändert die Ausführungsreihenfolge der Befehle dynamisch zur Laufzeit des Befehls (dynamisches Scheduling). Im Gegensatz zur In-order-Befehlsverarbeitung, bei der statisch durch einen Compiler die Reihenfolge der Abarbeitung der Befehle und Pipelineunterbrechungen festgelegt wird (***statisches Scheduling***), benutzt das Out-of-Order-Verfahren den Status der Befehlsverarbeitung zur Laufzeit des Befehls zur Festlegung der Befehlsabarbeitung (***dynamisches Scheduling***).

Mikroarchitekturen für Prozessoren, die ihre Programmbefehle out-of-order bearbeiten, erledigen die Teilaufgaben in folgenden Einheiten (siehe Abb. 2.1):

1. Die ***Fetch-Unit*** holt über den Bus eine feste oder variable Anzahl von Befehlen aus dem Code-Cache und lädt sie in einen Befehlspuffer.
2. Die ***Decode-Unit*** mit mehreren nebenläufigen Dekodierern, holt sich einen Teil dieser Befehle und versucht pro Taktzyklus diese zu dekodieren. Für Sprungbefehle wird mit einem Vorhersagealgorithmus eine Sprungvorhersage durchgeführt. Bei Sprüngen, die voraussichtlich ausgeführt werden, kann dadurch das Sprungziel in den Codecache geladen werden.
3. Die ***Dispatch-Unit*** holt die Befehle meistens in ihrer logischen Reihenfolge aus dem Befehlspuffer. Meistens deshalb, weil der Compiler die statischen Datenabhängigkeiten und Ressourcenkonflikte bereits erkannt und dementsprechend die Reihenfolge der Befehle umgestellt hat. Hat der Compiler nicht genügend unabhängige Befehle gefunden, so hat er NO-OP-Befehle (No Operation) eingefügt (Statisches Scheduling). Die Dispatch-Unit untersucht jeden Befehl auf Registerabhängigkeiten bei Operanden und Ergebnisregister und ändert dementsprechend die Befehlsausführungsreihenfolge (Dynamisches Scheduling). Die Veränderung der Ausführungsreihenfolge erfordert zusätzliche Schattenregister, um Ergebnisregisterkonflikte und daraus resultierende Pipelineunterbrechungen zu vermeiden. Die umgeordneten Befehle landen in einem Befehlspool, dem sogenannten Reorder Buffer, der die Out-of-Order-Befehlsverarbeitung widerspiegelt, und werden der Execution-Unit zugeführt.
4. Die ***Execution-Unit*** besteht aus mehreren nebenläufigen Funktionseinheiten. Eine Funktionseinheit ist typischerweise auf die Ausführung einer bestimmten Teilmenge der möglichen Befehlstypen beschränkt. Die Befehle werden schließlich durch die mehreren Funktionseinheiten parallel ausgeführt und die Ergebnisse in die Schattenregister geschrieben.
5. Die ***Completion-Unit*** schreibt die Ergebnisse, die in den Schattenregistern vorliegen, in die Register zurück und überprüft, ob ein Interrupt aufgetreten ist (Abb. 2.1).

Die Abb. 2.2 zeigt eine ***Superskalare Architektur*** bestehend aus vier nebenläufigen Funktionseinheiten. Wie bei dieser Architektur üblich, führen die Funktionseinheiten Instruktionen aus ***einem*** Programm oder ***einem*** Thread aus. Von diesem versucht sie so viel mehr Instruktionen zu finden, die in einem Zyklus nebenläufig ausführbar sind. Findet sie nicht genug, so bleibt die Funktionseinheit unbenutzt, was in Abb. 2.2 durch ein leeres Kästchen gekennzeichnet ist. Die benutzten Kästchen enthalten ein T1 (Instruktionen aus einem Thread oder einem Programm). Die leeren Kästchen resultieren meistens aus nicht genügend vorhandenem Instruktions-Level-Parallelismus. Eine horizontal komplette Reihe von leeren Kästchen kennzeichnet einen komplett ungenutzten Zyklus. Dies wird durch Instruktionen mit hoher Latenzzeit verursacht, die eine weitere Instruktionszufuhr verhindern. Die hohen Latenzzeiten rühren von Instruktionen mit Speicherreferenzen auf den L1- oder L2-Cache. Landet eine Speicherreferenz weder im L1- noch im L2-Cache,

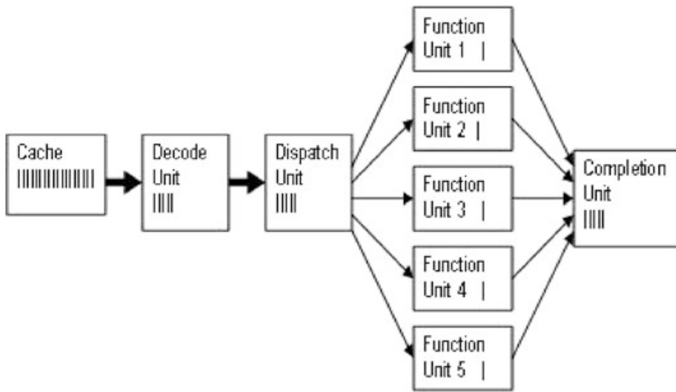


Abb. 2.1 Befehlspipeline bei Superskalarität

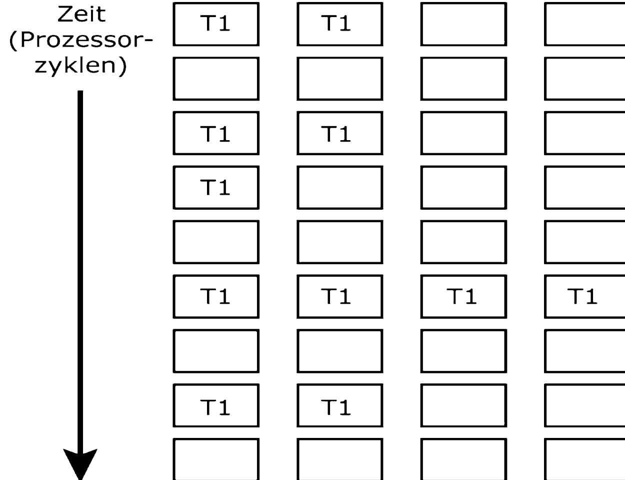


Abb. 2.2 Funktionseinheiten und ihre Ausnutzung bei Superskalaren Architekturen

so entsteht eine lange Wartezeit, bis das angeforderte Wort in den Cache geladen ist – die Pipeline muss angehalten werden.

2.1.1.2 Thread Level Parallelismus

Multithreaded Prozessoren besitzen Befehlszähler und Register für mehrere Threads. Pro Thread einen Befehlszähler und einen kompletten Registersatz. In jedem Zyklus führt der Prozessor Instruktionen von irgendeinem Thread aus. Im nächsten Zyklus schaltet er um auf den Hardwarekontext (Befehlszähler und Registersatz) eines anderen Threads und führt in superskalarer Arbeitsweise die Instruktionen dieses neuen Threads aus. Dadurch vermeidet man die komplett ungenutzten Zyklen, die durch die Latenzzeiten verursacht sind. Die Abb. 2.3 zeigt diesen Sachverhalt, wobei die Striche zwischen den einzelnen Prozessorzyklen das Umschalten auf einen anderen Thread andeuten.

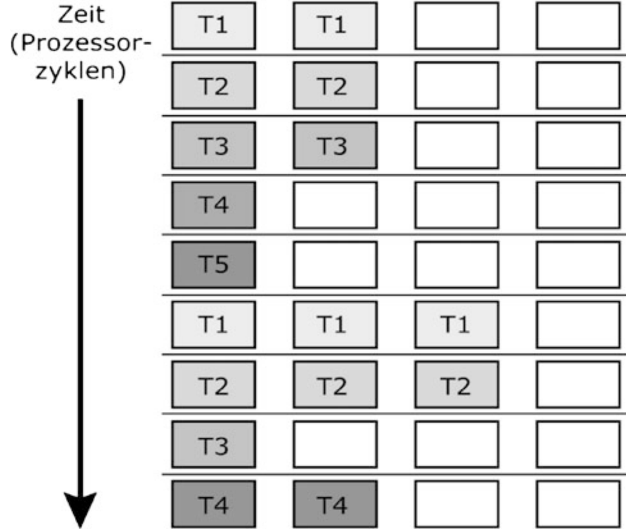


Abb. 2.3 Funktionseinheiten und ihre Ausnutzung bei Multithreaded-Architekturen

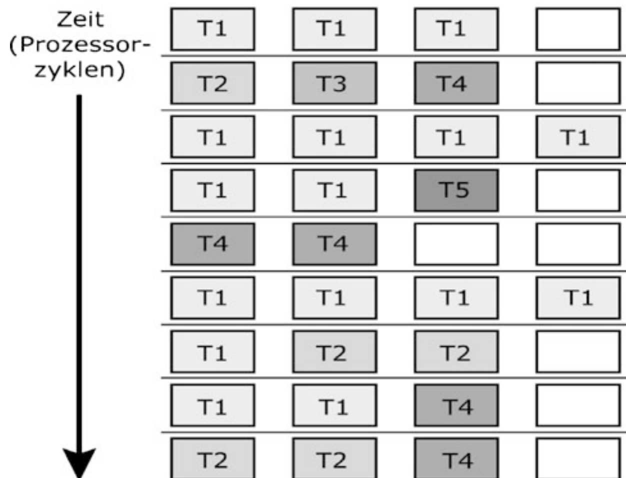


Abb. 2.4 Funktionseinheiten und ihre Ausnutzung beim Simultanous Multithreading

2.1.1.3 Arbeitsweise des Simultaneous Multithreading

Simultaneous Multithreading benutzt Instruction Level Parallelism und Thread Level Parallelism und vermeidet somit die leeren Kästchen in vertikaler und horizontaler Richtung. Simultaneous Multithreading nutzt den Instruction Level Parallelism von jedem Thread aus und startet dynamisch die parallel ausführbaren Instruktionen von einem Thread. Besitzt also ein Thread in einem Zyklus einen hohen Instruction Level Parallelism, so kann diese durch die Superskalar-Architektur befriedigt werden. Haben viele Threads einen niederen Instruction Level Parallelism, so können sie in einem Zyklus zusammen parallel abgearbeitet werden (siehe Abb. 2.4).

Simultaneous Multithreading ist deshalb attraktiv, weil dadurch mit nur geringfügigem zusätzlichem Steuerungsaufwand die Prozessorleistung gegenüber den Superskalar-Architekturen deutlich erhöht werden kann. So erzielten [TEE 96] mit einer Simulation für einen Simultaneous Multithreading-fähigen Prozessor Leistungsverbesserungen zwischen 1,8 und 2,5 gegenüber einem Superskalar-Rechner.

2.1.2 Architektur von eng gekoppelten Multiprozessoren

Alle Prozessoren bei einem eng gekoppelten Multiprozessor können den Adressraum des gemeinsamen Speichers gemeinsam benutzen. Jeder Prozessor kann ein Speicherwort lesen oder schreiben, indem er einfach einen LOAD- oder STORE-Befehl ausführt. Die üblich verwendete Speichertechnologie DRAM (Dynamic Random Access Memory) erlaubt Zugriffszeiten von ungefähr 10 ns [BM 06]. Dies entspricht einer Frequenz von 100 MHz, also nur einem Bruchteil der Taktfrequenz moderner Prozessoren. Der Verkehr zwischen Prozessor und dem Hauptspeicher bildet einen leistungsbegrenzenden Flaschenhals in einem Rechner, der ***von-Neumann-Flaschenhals*** [B 78] heißt. Eng gekoppelte Multiprozessoren verengen noch zusätzlich gegenüber Einprozessorsystemen den von-Neumann Flaschenhals und vergrößern den Prozessoren-Speicher-Verkehr: Jeder hinzukommende weitere Prozessor greift auf den gemeinsamen Speicher zu und belastet die gemeinsame Prozessor-Speicher-Verbindung. Aus diesem Grund sind bei eng gekoppelten Multiprozessoren die Anzahl der Prozessoren auf 8 bis höchstens 64 beschränkt.

Zur Reduktion dieses Flaschenhalses besitzt jeder Prozessor einen Cache (siehe Abb. 2.5), der Kopien von Teilen des Hauptspeichers enthält. Er besteht aus SRAM (Static Random Access Memory) und ermöglicht Zugriffsgeschwindigkeiten, die der Taktfrequenz der Prozessoren nahe kommen. Die Größe des Caches ist beschränkt und

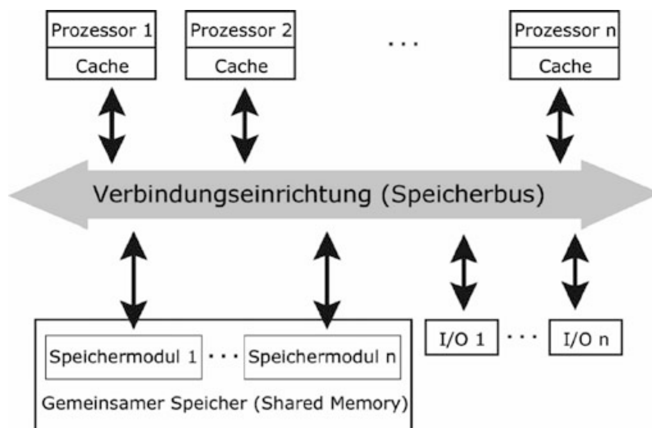


Abb. 2.5 Eng gekoppelter Multiprozessor

umfasst üblicherweise wenige MBytes. Zugriffe auf Befehle und Daten, die nicht im Cache liegen, und zu sogenannten Cache Misses führen, verursachen Leistungseinbußen des Rechners [BM 06].

Das Zusammenhängen der Prozessoren mit dem gemeinsamen Speicher bietet beliebige Leistungs- und Fehlertoleranzstufen. Durch den gemeinsamen Bus und somit durch den von Neumann-Flaschenhals sind die Leistungs- und Fehlertoleranzgrenzen nach oben beschränkt. Allerdings bietet der gemeinsame globale Speicher Vorteile bei der Prozesssynchronisation und -kommunikation, die hier wie bei Einprozessorsystemen über den gemeinsamen Speicher läuft. Somit sind bei eng gekoppelten Multiprozessoren die bekannten Synchronisationsverfahren wie Locks, Semaphoren und Monitoren einsetzbar. Die Kommunikation ist schneller, denn sie verläuft lokal und braucht nicht über eine Netzwerkverbindung zu laufen.

Zur Reduktion des Prozessor-Speicher-Verkehrs dient jedem Prozessor

- ein vorgelagerter Cache,
- und zur Erhöhung der Leistung des Prozessor-Speicher-Verkehrs dienen leistungsfähigere Verbindungseinrichtungen, wie
  - Kreuzschienenschalter oder
  - Mehrebenennetzwerke.

Über einen gemeinsamen Bus lassen sich nur durch Zwischenschalten von mehrfach gestuften Caches, wobei eine ausreichende Datenlokalität vorausgesetzt wird, mehrere Prozessoren koppeln. Eng gekoppelte Multiprozessoren mit großer Prozessoranzahl sind auf einer Bustechnologie nicht aufbaubar [R 97]. Der Busengpass lässt sich nur durch ein- und mehrstufige Verbindungsnetze wie Kreuzschienenschalter und Mehrebenennetzwerke eliminieren [D 90].

2.1.2.1 Cachekohärenzprotokolle

Die Daten im Cache und im Hauptspeicher werden zerlegt in gleich große Blöcke. Ein ***Block*** ist die Einheit, die zwischen dem Hauptspeicher und dem Cache transferiert wird. Als Transfer- und Speichereinheit für den Cache dient eine ***Cache-Zeile*** (Cache Line) mit normalerweise 32 oder 64 Byte [T 06]. Optimale Cache- und Blockgrößen sind in [P 90] diskutiert.

Die Einführung des Caches bringt jedoch Probleme, wenn gemeinsame Blöcke in mehreren verschiedenen Caches vorliegen. Nehmen wir dazu Folgendes an: Zwei Prozessoren lesen aus dem Hauptspeicher den gleichen Block in ihre Caches. Anschließend überschreibt einer dieser Prozessoren diesen Block. Liest nun der andere Prozessor diesen Block aus seinem Cache, so liest er den alten Wert und nicht den gerade überschriebene Wert. Die Daten in den Caches sind ***inkonsistent,*** und weiterhin sind die Daten in einem der Caches und dem Hauptspeicher ***inkonsistent***.

Datenkonsistenz bedeutet, dass im Hauptspeicher und in den Caches zu keinem Zeitpunkt verschiedene Kopien desselben Blocks existieren.

Zur Lösung des Konsistenzproblems bei Multiprozessoren genügt es, eine abgeschwächte Bedingung, nämlich die ***Datenkohärenz***, zu fordern. Datenkohärenz liegt vor, wenn beim Lesen des Blocks, welcher mehrfach überschrieben wurde, immer der zuletzt geschrieben Wert gelesen wird. Die Datenkonsistenz schließt die Kohärenz ein, aber nicht umgekehrt.

MESI Cachekohärenz-Protokoll

Datenkohärenz ist bei Multiprozessorsysteme gegeben, wenn

1. zwar jeder Prozessorcache über eine Kopie von Daten im Hauptspeicher verfügen darf,
2. aber nur ein Prozessorcache eine modifizierte Kopie der Daten besitzen darf, jedoch nur solange, wie kein anderer Prozessor dieselben Daten liest.

Das Kohärenzproblem tritt nicht nur bei Caches von Multiprozessoren auf, sondern auch bei einem Cache bei einem Einprozessorsystem, wenn der andere Prozessor DMA-Hardware (DMA – Direct Memory Access) bzw. ein Ein/Ausgabe-Prozessor ist, der zusätzlich zur CPU vorhanden ist.

Das obige unter 1. beschriebene Problem ist lösbar, wenn beim Schreiben neuer Werte in den Cache auch die entsprechenden Werte in den Hauptspeicher und in die anderen Caches (bei Multiprozessoren) überschrieben werden. Für dieses Vorgehen gibt es zwei Methoden:

1. Das ***Durchschreiben*** (***write through*** oder ***store through***), wo bei jedem Schreiben in den Cache gleichzeitig auch in den entsprechenden Hauptspeicher geschrieben wird.
2. Das ***verzögerte Rückschreiben*** (***deferred write***), bei dem die korrespondierende Kopie im Hauptspeicher nicht sofort ersetzt wird, sondern erst beim Verdrängen der Daten aus dem Cache.

Verfahren eins gewährleistet nicht nur die Kohärenz, sondern auch die Konsistenz, und besticht durch seine Einfachheit. Bei Caches, die gemäß dem Prinzip write through (Verfahren eins) arbeiten, ist jedoch nachteilig, dass jedes Schreiben einen Hauptspeicherzugriff bedingt, was bei einem Multiprozessorsystem wieder den Bus belastet. Deshalb wird dieses Verfahren nur bei Einprozessorsystemen zwischen dem ***Primary Cache*** (***Level 1 Cache***) und dem ***Secondary Cache*** (***Level 2 Cache***) angewandt. Die Daten oder Instruktionen werden dabei aus dem Primary Cache geholt, und die Ergebnisse werden beim Primary Cache in den Secondary Cache durchgeschrieben (write through). Bei Multiprozessoren ist zur Reduktion der Hauptspeicherzugriffe nur die verzögerte Rückschreibemethode angebracht (Verfahren zwei).

Zur Realisierung des verzögerten Rückschreibens gibt es auf der Hardwareebene zwei verschiedene Strategien beim Schreiben [S 90]:

1. ***Write invalidate*** und
2. ***Write update***.

Die Strategie Write invalidate arbeitet folgendermaßen:

Leseanfragen werden lokal befriedigt, falls eine Kopie des Blocks existiert. Überschreibt ein Prozessor einen Block, werden alle anderen Kopien (im Hauptspeicher und den anderen Caches) auf ungültig (***Invalidated***) gesetzt (siehe Abb. 2.6b). Ein weiteres Überschreiben des gleichen Prozessors kann nur auf seinem ihm gehörendem Cache durchgeführt werden, da keine weiteren Kopien mehr existieren. Will ein anderer Prozessor den überschriebenen Block lesen, so muss er warten, bis der Block wieder gültig ist.

Im Gegensatz zur Strategie Write invalidate- ändert (***updated***) die Strategie Write update beim Schreiben eines Prozessors alle Kopien in den anderen Caches (siehe Abb. 2.6c).

Die Strategien Write invalidate und Write update erfordern, dass die Konsistenzkommandos (Invalidation-Kommando und Update-Kommando) wenigstens diejenigen Caches erreichen, die Kopien des Blocks haben. Das bedeutet, dass jeder Cache die Konsistenzkommandos bearbeiten muss, um herauszufinden, ob es einen Block in seinem Cache betrifft. Deshalb heißen diese Protokolle ***Snoopy Cache Protocols***, da jeder Cache am Bus nach eingehenden Inkonsistenzkommandos „schnüffeln“ muss.

Ein Snoopy Cache Protocol, basierend auf der Strategie Write Invalidate, schreibt einen Block im Cache nur beim ersten Schreiben in den Hauptspeicher zurück (***Write-once Protocol*** oder ***Write-first Protocol***). Nachfolgendes Lesen und Schreiben geschieht dann auf dem lokalen Block im Cache und erfordert kein Rückschreiben in den Hauptspeicher. Dadurch bedingen nachfolgende Lese- und Schreiboperationen des Prozessors keinen Busverkehr mehr. Das Write-once Protocol entspricht beim ersten Schreiben dem write through. Dieser zusätzliche Hauptspeicherzugriff zum Durchschreiben wird hier in Kauf genommen, da der Prozessor warten muss, bis die restlichen Caches den Block invalidiert haben.

Das ***Write Invalidate Snoopy Cache Protocol*** assoziiert einen Zustand mit jeder Kopie eines Blocks im Cache. Die Zustände für eine Kopie sind:

- ***M***odified: Daten wurden einmal überschrieben, und die Kopie ist nicht konsistent mit der Kopie im Hauptspeicher (Write-once Protocol). Die Kopie im Hauptspeicher ist die veraltete Kopie.
- ***E***xclusive (unmodified): Die Daten wurden nicht modifiziert, und die Kopie ist die einzige Kopie im System, die im Cache liegt.
- ***S***hared (unmodified): Es existieren mehrere gültige Kopien, die konsistent mit der Kopie im Hauptspeicher sind.
- ***I***nvalid: Die Kopie ist ungültig.

Gemäß den Anfangsbuchstaben der Namen für die Zustände heißt das ***Snoopy Cache Invalidation Protocol*** auch ***MESI-Protokoll*** [H 93].

Zusätzlich zu den normalen Kommandos zum Lesen eines Blocks aus (`Read-Block`) und Schreiben eines Blocks (`Write-Block`) in den Speicher benötigt man noch die beiden Konsistenzkommandos

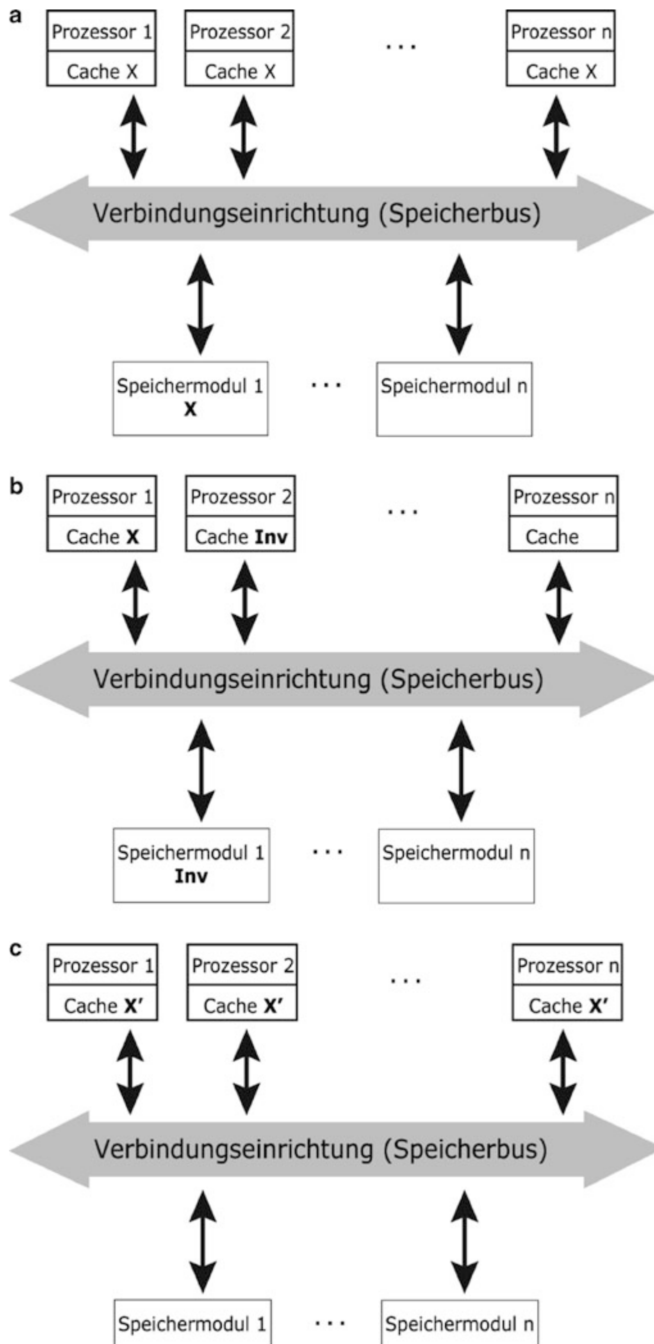


Abb. 2.6 **a** Ausgangszustand: Hauptspeicher und alle Caches haben konsistente Kopien des Blocks X, **b** Write invalidate: Alle Kopien mit Ausnahme von Cache in Prozessor 1 sind ungültig (Inv), wenn Prozessor 1 Block X überschreibt, angezeigt durch X′, **c** Write update: Alle Kopien (mit Ausnahme der Speicherkopie, was ignoriert wird) sind abgeändert

- `Write-Inv`: Setzt alle anderen Kopien eines Blocks auf ungültig (invalidate).
- `Shared-Signal`: Zum Anzeigen, dass eine weitere Kopie existiert.

Zustandsänderungen werden entweder durch die Lese- und Schreibkommandos des Prozessors (`Proc-Read`, `Proc-Write`) oder über die Konsistenzkommandos, die über den Bus kommen (`Read-Block`, `Write-Block`, `Write-Inv`, `Shared-Signal`) bewirkt.

Der Ablauf des ***MESI-Protokolls*** lässt sich nun angeben durch die Aktionen, welche durchgeführt werden, wenn der Prozessor einen Block liest (`Proc-Read`) oder einen Block beschreibt (`Proc-Write`).

Bei einem `Proc-Read` und `Proc-Write` können die folgenden Fälle auftreten:

- Ein ***Read Hit*** tritt auf, wenn der Block im Cache vorhanden und gültig ist.
- Ein ***Read Miss*** tritt auf, wenn der Block im Cache nicht vorhanden oder ungültig ist (Invalid).
- Ein ***Write Hit*** tritt auf, wenn der Block im Cache vorhanden und gültig ist.
- Ein ***Write Miss*** tritt auf, wenn der Block im Cache nicht vorhanden oder ungültig ist (Invalid).

Die bei einem Hit oder Miss oder einer Ersetzung (Replacement) eines Blocks im Cache durchzuführenden Aktionen sind:

- ***Read Hit***: Benutze die lokale Kopie aus dem Cache.
- ***Read Miss***: Existiert keine oder keine Modified Kopie, dann hat der Hauptspeicher eine gültige Kopie. Kopiere Block vom Hauptspeicher in den Cache und setze den Block auf Exclusive. Existiert eine Modified-Kopie, dann ist es die einzige gültige Kopie im System und die Kopie im Hauptspeicher ist ungültig. Schreibe die Modified-Kopie in den Hauptspeicher, so dass er eine gültige Kopie enthält. Lade die gültige Kopie vom Hauptspeicher in den Cache. Setze mit dem Shared-Signal beide Kopien auf Shared.
- ***Write Hit***: Existiert keine Modified-Kopie, dann kann das Schreiben lokal ausgeführt werden. Ist die Kopie Modified, so muss sie vorher in den Speicher zurückgeschrieben werden. Der neue Zustand der Kopie ist Modified. Sende das Konsistenzkommando `Write-Inv` zu allen anderen Caches, so dass die Caches ihre Kopien auf `Invalid` setzen können.
- ***Write Miss***: Die Kopie besitzt den Zustand Modified, dann wird die Kopie in den Speicher zurückgeschrieben, andernfalls kann ein Zurückschreiben unterbleiben. Die Kopie wird vom Speicher geholt, und anschließend wird die Kopie überschrieben. Sende das Konsistenzkommando `Write-Inv` zu allen anderen Caches, so dass die Caches ihre Kopien auf Invalid setzen können. Der neue Zustand der Kopie ist Exclusive.
- ***Replacement***: Ist die Kopie Dirty (d. h. der Block ist im Cache geändert), so muss sie in den Hauptspeicher zurück geschrieben werden. Andernfalls ist keine Aktion notwendig.

Die Abb. 2.7a–e zeigen an einem Beispiel die Auswirkung des MESI-Protokolls und verdeutlicht, dass ein mehrmaliges Lesen und Schreiben eines Prozessors nur beim erstmaligen Schreiben Busverkehr erfordert und nachfolgendes Schreiben und Lesen nur lokal auf der Kopie im Cache ausgeführt wird. In Abb. 2.7 beziehen sich alle Speicherzugriffe auf dieselbe Adresse.

Bus-Snooping mit dem MESI-Protokoll, und somit mit der Write Invalidate-Strategie, verwendet der Intel Pentium 4 und viele andere CPUs [T 06]. Ein Beispiel für ein ***Write update Snoopy Cache Protocol*** ist das ***Firefly Protocol,*** das für eine Firefly Multiprocessor Workstation von Digital Equipment implementiert wurde, und das ***Dragon Protocol*** für eine Dragon Multiprocessor Workstation von Xerox PARC. Eine Übersicht und Beschreibung dieser Protokolle gibt Archibald und Baer [AB 86].

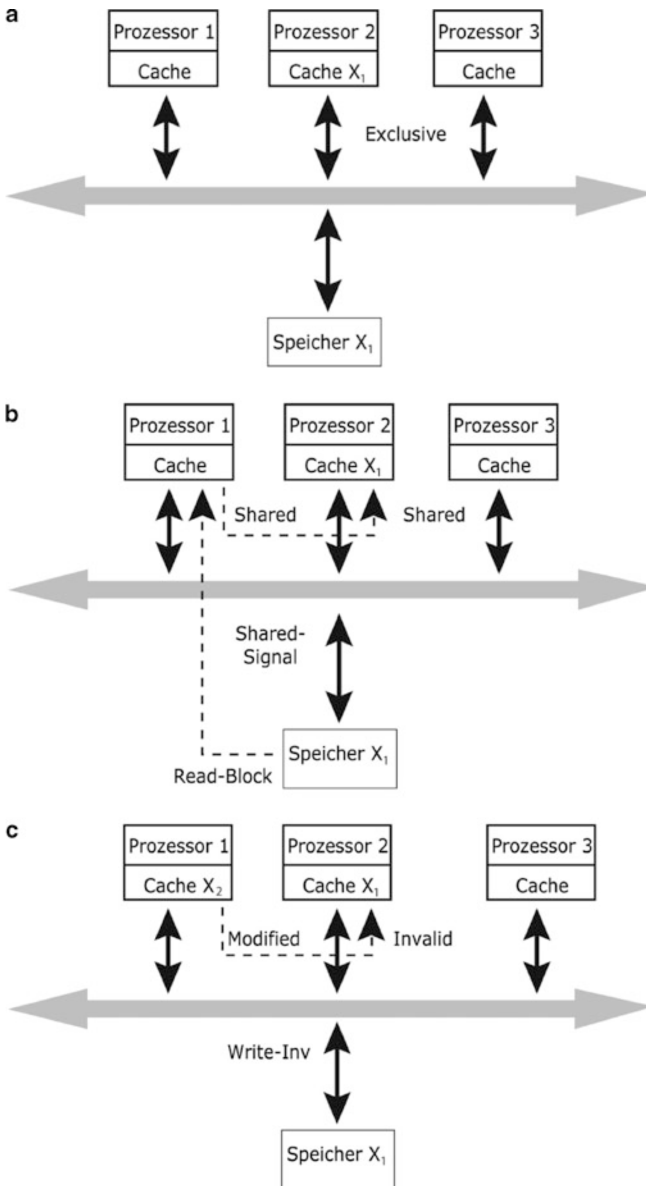
Die Techniken Write Update und Write Invalidate sind unter verschiedenen Belastungen unterschiedlich leistungsfähig. Update-Nachrichten befördern Nutzdaten und sind daher umfangreicher als Invalidierungs-Nachrichten. Die Update-Strategie vermeidet jedoch zukünftige Cache-Fehler [T 06]. Performance-Messungen, gewonnen durch Simulation des Write Invalidate- und Write Update-Protokoll, sind in [L 93] enthalten.

Die Hauptunterschiede zwischen einem Snoopy Cache und einem gewöhnlichen Cache für Einprozessorsysteme, oder den nachfolgend beschriebenen Verzeichnis-basierten Multiprozessor-Caches, liegen einmal im

- ***Cache Controller***, der Information in der Cache-Zeile abspeichert über jeden Zustand eines Blocks. Bei Snoopy Caches kann der Cache-Controller als Zustandsautomat ausgelegt werden, der das Cache Kohärenz-Protokoll gemäß den Zustandsübergängen implementiert. Wie bei den Snoopy-Protokollen benötigt ein Directory-basierter Cache zum Abspeichern des Zustandes mindestens zwei Bits. Zum anderen im
- ***Bus Controller***, der bei Snoopy Caches den Snooping-Mechanismus implementiert und alle Busoperationen überwacht und Entscheidungen fällt, ob Aktionen nötig sind oder nicht. Zum Weiteren im
- ***Bus*** selber, der zur effizienten Unterstützung der Write invalidate- oder Write update-Protokolle weitere Busleitungen besitzen muss. Zum Beispiel ist eine Shared Line bei der Write Update-Strategie angebracht.

Verzeichnis-basierte Cachekohärenz-Protokolle

Snoopy Cache-Protokolle passen sehr gut mit dem Bus zusammen und sind nicht geeignet für allgemeine Verbindungsnetzwerke, wie die nachfolgend beschriebenen Kreuzschienenschalter und Mehrebenennetzwerke. Der Grund liegt darin, dass sie einen Broadcast erfordern, der mit einem Bus oder sogar mit einer dafür vorgesehenen Busleitung einfacher zu bewerkstelligen ist als mit einem Verbindungsnetzwerk. Anstatt eines Broadcast sollten die Konsistenzkommandos nur die Caches erreichen, die eine Kopie des Blocks haben. Dies bedingt, es muss Information vorhanden sein, welche Caches Kopien besitzen von allen in den Caches vorhandenen Blöcken. Cachekohärenz-Protokolle, die irgendwie


a
Prozessor 1
Cache
Prozessor 2
Cache X1
Prozessor 3
Cache
Exclusive
Speicher X1
b
Prozessor 1
Cache
Prozessor 2
Cache X1
Prozessor 3
Cache
Shared
Shared
Shared-
Signal
Speicher X1
Read-Block
c
Prozessor 1
Cache X2
Prozessor 2
Cache X1
Prozessor 3
Cache
Modified
Invalid
Write-Inv
Speicher X1

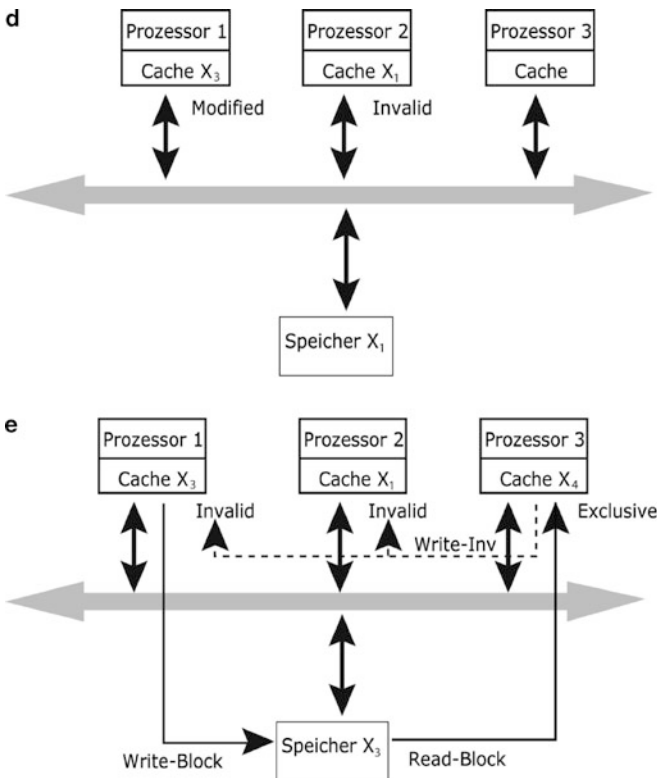


Abb. 2.7 **a** Anfangszustand: X_1 ist im Hauptspeicher und im Cache von Prozessor 2, **b** Prozessor 1 liest X_1: Read Miss: Prozessor 2 reagiert auf das Lesen von Prozessor 1 mit neuem Zustand Shared. Der Block kommt vom Hauptspeicher. Beide Zustände sind Shared, **c** Prozessor 1 überschreibt X_1: Write Hit: Erstes Schreiben von Prozessor 1 macht die Kopie im Cache von Prozessor 2 ungültig, **d** Prozessor 1 überschreibt X_2: Write Hit: Nachfolgendes Schreiben und Lesen geschieht lokal im Cache von Prozessor 1. Kein Busverkehr!, **e** Prozessor 3 überschreibt X_2: Write Miss: Prozessor 1 liefert den Block. Alle anderen Caches werden auf ungültig gesetzt

Information abspeichern, welche Caches eine Kopie des Blocks besitzen, arbeiten mit einem ***Verzeichnis Schema*** (***Directory Scheme***) [S 90].

Ein ***Verzeichnis*** (***Directory***) oder eine Datenbank enthält für jeden Speicherblock (memory-line) einen Eintrag, welcher den Zustand des Blocks und einen Bitvektor mit den Prozessoren, welche Kopien besitzen, speichert. Durch Auswertung dieser Einträge kann jederzeit bestimmt werden, welcher Cache, wo aktualisiert werden muss.

Eine Konkretisierung des Konzeptes der Verzeichnisse an einem System mit 256 Knoten, wobei jeder Knoten aus einer CPU und 16 MB RAM besteht, erläutert Tanenbaum [T 06]. Stenström [S 90] untersucht die Anzahl der Bits zur Speicherung der Information für jeden Cache-Block und betrachtet dann den Netzwerkverkehr für das Write Invalidate-Cache-Protokoll. Für verschiedene Directory-Schemas führt Stenström dann eine Leis-

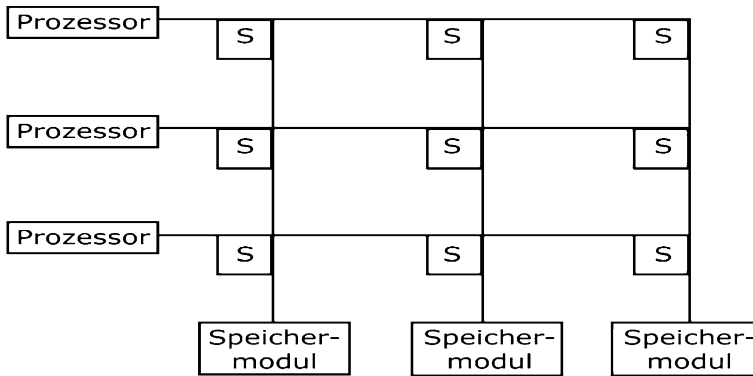


Abb. 2.8 Verbindung von Prozessoren und Speichermodule mit Kreuzschienenschalter

tungsoptimierung durch. Hennessy und Patterson [HP 06] geben ein Directory-based Cache Coherence Protocol mit den Zuständen Invalid, Modified und Shared an. Im Anhang C erläutert [HP 06] die Cache Performance und die daraus resultierende Cache-Optimierung.

2.1.2.2 Kreuzschienenschalter-basierte Multiprozessoren

Bei eng gekoppelten Multiprozessoren wächst der Busverkehr linear mit der Anzahl der Prozessoren. Der einzige Weg, den Prozessor-Speicher-Engpass zu beseitigen, besteht darin, den Hauptspeicher in mehrere Module aufzuteilen und mehrere Pfade zwischen die CPUs und die Speichermodule zu legen. Dies erhöht nicht nur die Bandbreite der Zugriffe, sondern erlaubt auch die parallele Abarbeitung der Speicherzugriffe von verschiedenen CPUs zu verschiedenen Speichermodulen. Die einfachste Schaltung, um n CPUs mit k Speichermodule zu verbinden, ist ein ***Kreuzschienenschalter*** (Crossbar Switch) mit n*k Schalter, wie Abb. 2.8 zeigt.

Jeder Kreuzungspunkt enthält einen Schalter, um entweder die horizontale oder vertikale Linie zu verbinden. Beim Zugriff eines Prozessors auf ein bestimmtes Speichermodul wird der entsprechende Pfad durchgeschaltet.

Der Vorteil eines Kreuzschienenschalters liegt darin, dass sich zwischen den CPUs und den Speichermodulen ein nicht blockierendes Netzwerk befindet. Das bedeutet, dass keine CPU verzögert werden muss, weil sie einen bestimmten Kreuzungspunkt benötigt oder weil eine Leitung besetzt ist. Natürlich ist hierbei vorausgesetzt, dass das Speichermodul verfügbar ist und somit kein anderer Prozessor zur gleichen Zeit auf das gleiche Speichermodul zugreift.

Ein Nachteil von kreuzschienenschalter-basierten Multiprozessoren ist, dass die Anzahl der Kreuzungspunkte quadratisch mit der Anzahl der CPUs und Speichermodulen wächst. Mit 100 CPUs und 100 Speichermodulen erhält man 10.000 Kreuzungspunkte und somit auch Schalter.

Kreuzschienenschalter setzt man schon seit Jahrzehnten in Telefonnetzen ein, um eine Gruppe ankommender Leitungen in beliebiger Weise auf eine Gruppe abgehender Leitungen durchzuschalten [T 06].

Der Parallelrechner ***IBM RS/6000 SP*** [IRS 06] ist ein skalierbarer Parallelrechner (SP: scalable POWERparallel) dessen Grundeinheit so genannte ***Frames*** sind. Ein Frame kann bis zu 16 Knoten besitzen. Je nach Prozessortyp besteht ein Knoten aus 1 bis 4 Prozessoren. Jeder Frame enthält einen so genannten ***High Performance Switch*** (***HPS***), der einem Kreuzschienenschalter entspricht. Der Rechner Sun Fire E25K [T 06] benutzt ebenfalls einen Kreuzschienenverteiler.

Beim Athlon 64 X2 und dem Opteron [A 06] hängen beide Rechnerkerne an einem Crossbar Switch. Über diesen greifen sie auf den Speicher und die Peripherie zu. Bei der Cache-Verwaltung hat AMD die Cacheverwaltung des Alpha-Prozessors von DEC übernommen. Diese sieht fünf Bits zur Markierung von Cache-Zellen vor: Modify, Owner, Exclusive, Shared und Invalid (***MOESI-Protokoll***) [H 93]. Über einen eigenen Kanal, den „Snoop Channel", kann ein Kern den Status einer Cache-Zelle des anderen abfragen, ohne den restlichen Datentransfer zu bremsen.

2.1.2.3 Mehrebenennetzwerke-basierte Multiprozessoren

Ein $n*n$ ***Mehrebenennetzwerk***, oder auch ***Omega-Netzwerk*** [R 97] genannt, verbindet n Prozessoren mit n Speichermoduln. Dabei liegen mehrere Ebenen oder Bänke von Schaltern auf dem Weg vom Prozessor zum Speicher. Ist n eine Potenz von 2, so benötigt man $\log n$ Ebenen und $n/2$ Schalter pro Ebene. Die Schalter haben zwei Eingänge und zwei Ausgänge. Ein Prozessor, der zum Speicher zugreifen möchte, gibt den Zugriffswert auf das Speichermodul als Bitwert an. Diese Bitkette enthält für jede Ebene ein Kontrollbit. Der Schalter auf der Ebene i entscheidet dann, ob der Eingabekanal auf den oberen oder unteren Ausgabekanal gelegt wird:

- Ist das Kontrollbit für den Schalter eine ***Null***, so wird der Eingang mit dem ***oberen Ausgang*** verbunden.
- Ist das Kontrollbit für den Schalter eine ***Eins***, so wird der Eingang mit dem ***unteren Ausgang*** verbunden.

Die Abb. 2.9 zeigt ein Netzwerk, das acht Prozessoren mit acht Speichermodulen verbindet. Weiterhin zeigt die Abbildung, wie Prozessor 3 eine Speicheranfrage an den Speichermodul 3 stellt. Das Speichermodul 3 hat den Bitwert 011, und diese Bitkette enthält die Kontrollbits für die drei Schalter.

Nehmen Sie nun an, dass parallel zum Zugriff von Prozessor 3 auf Speichermodul 3, Prozessor 7 auf Speichermodul 1 zugreifen möchte. Dabei kommt es zu einem Konflikt der beiden Speicheranfragen bei einem Schalter auf Ebene 1 und einer Verbindungsleitung zwischen einem Schalter der Ebene 1 und Ebene 2. Um die parallele Abfrage der beiden Prozessoren abzuarbeiten, muss eine Anfrage blockiert werden. Damit ist ein

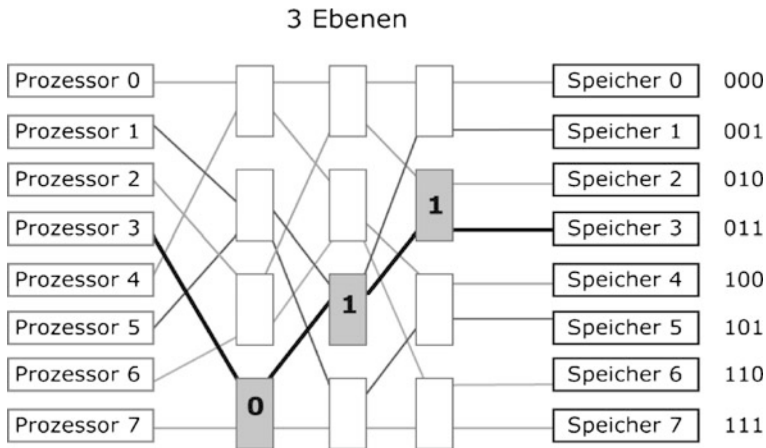


Abb. 2.9 8*8 Mehrebenen-Netzwerk, wobei Prozessor 3 eine Anfrage an den Speichermodul 3 stellt

Mehrebenen-Netzwerk im Vergleich zu einem Kreuzschienenschalter ein ***blockierendes Netzwerk***.

Eng gekoppelte Multiprozessoren, welche als Prozessor-Speicherverbindung ein Mehrebenen-Netzwerk einsetzen, waren in den 1980er Jahren die von BBN Technologies (Bolt, Bernak, und Newman) gebauten ***Butterfly-Maschinen*** [BBN 89, RT 86]. Butterfly deshalb, weil die eingesetzten Verschaltung einem Schmetterling mit vier Flügeln entsprechen. Eine Butterfly-Maschine konnte bis zu 512 CPUs mit lokalem Speicher haben, und durch das Mehrebenen-Netzwerk konnte jede CPU auf den Speicher der anderen CPUs zugreifen. Die eingesetzten CPUs waren gewöhnliche CPUs (Motorola 68020 und später Motorola 88100). Weitere experimentelle Multiprozessoren mit Mehrebenen-Netzwerken waren der ***RP3*** von IBM [PBG 85] mit bis zu 512 Prozessoren und der ***NYU-Ultracomputer*** [GGK 83] von der New York University mit bis zu 4096 Prozessoren. Die Butterfly-Maschine, der RP3 und der NYU-Ultracomputer gehören heute zur Geschichte des Supercomputing.

2.1.2.4 Multicore-Prozessoren

Durch die Fortschritte in der VLSI-Technologie ist es heute möglich, zwei oder mehr leistungsfähige CPU-Kerne auf einem einzigen Chip zu vereinen. Diese CPUs haben alle einen eigenen Cache und nutzen den Hauptspeicher gemeinsam. Somit handelt es sich um eng gekoppelte Multiprozessoren auf einem Chip. Demgemäß heißen sie auch ***Multiprocessor Systems-on-Chip*** (MPSoc) [JW 05]. Bei zwei Kernen heißen sie ***Doppelkern-Prozessor*** (Dual-Core), bei vier Kernen spricht man von einem ***Quad-Core-Prozessor*** und mit mehreren Kernen heißen sie ***Mehrkern-Prozessor*** (Multicore-Prozessor).

In den 1990er Jahren stand die Takt-Frequenz von wenigen MHz bis zu heute vier GHz im Vordergrund zur Leistungssteigerung von CPUs. Dieser Trend wurde beschränkt, da

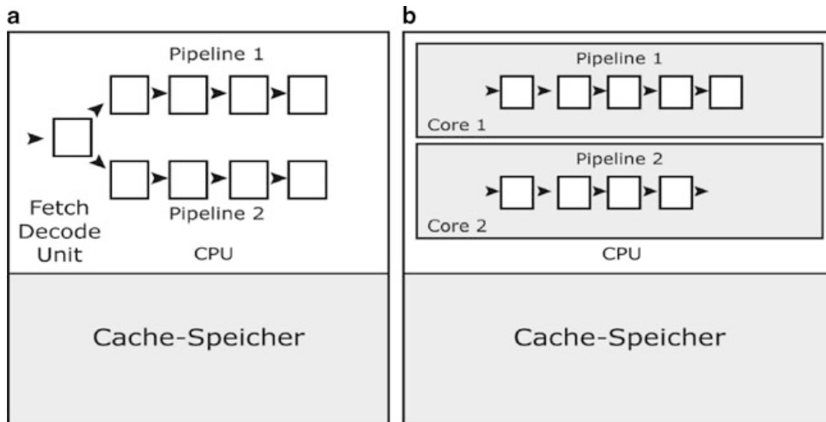


Abb. 2.10 Chip-Multicoreprozessor-Architekturen. **a** Duplikation der Befehlspipeline, **b** ein Chip mit zwei Kernen

durch die Erhöhung der Takt-Frequenz erhöhter Stromverbrauch und erhöhte Wärmeabgabe mit einhergehen [Bo 06, G 01]. Aus diesem Grund wird heute versucht, durch neue Prozessorarchitekturen und der Integration von mehreren CPU-Kernen auf einem Chip Leistungssteigerungen zu erhalten. Von den bisher vorgestellten Hardwarearchitekturen Superskalar, Simultaneous Multithreading und Multicore-Prozessoren bringen Letztere die beste Leistungssteigerung [HNO 97] und bestätigen diesen Trend.

Bei Chip-Multiprozessoren mit wenigen Kernen herrschen die beiden Richtungen vor [T 06], siehe Abb. 2.10:

1. Durch ***Duplikation der Befehlspipeline*** entstehen mehrere Befehlsausführungseinheiten. Eine Fetch/Decode-Instruktionseinheit führt den Ausführungseinheiten die Arbeit zu. Dieses Vorgehen entspricht dem Funktionsprinzip der Vektorrechner [HB 84] (Abb. 2.10a).
2. Durch den ***Einsatz von mehreren CPUs***, jede mit ihrer eigenen Fetch/Decode-Einheit, die wie ein eng gekoppelter Multiprozessor arbeitet (Abb. 2.10b).

Von den beiden in Abb. 2.10 dargestellten Architekturen setzen sich in letzter Zeit hauptsächlich die Richtung mehrere Kerne auf einem Chip (Multicore) unterzubringen, durch. Ein Multicore-Prozessor heißt

- ***symmetrisch*** oder ***homogen***, wenn alle Kerne gleich sind. Ein für diesen Prozessor übersetztes Programm kann auf jedem beliebigen seiner Kerne laufen. Das darauf ablaufende Betriebssystem unterstützt meistens ***Symmetric Multiprocessing*** (***SMP***).
- ***asymmetrisch*** oder ***heterogen***, wenn die Kerne unterschiedlich sind und spezielle Aufgaben haben. Ein Programm kann nur auf einem seiner Übersetzung entsprechenden Kern ausgeführt werden. Einige Kerne arbeiten wie klassische Prozessoren, andere

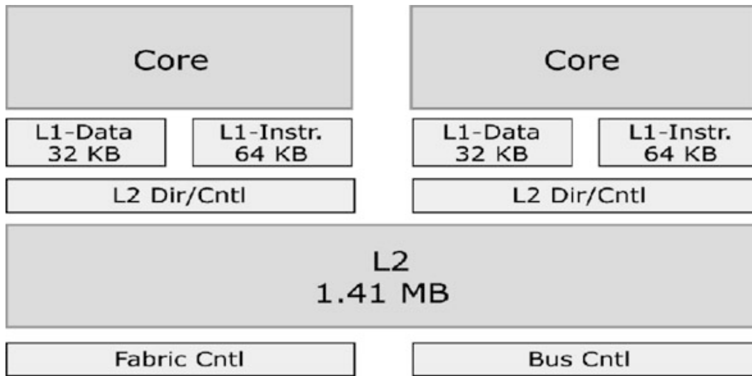


Abb. 2.11 Power4 Chip

wie asynchrone Coprozessoren. Auf jedem Prozessorkern läuft ein separates Betriebssystem oder eine separate Installation desselben Betriebssystems [S 06]. Dieser Betrieb heißt ***Asymmetric Multiprocessing*** (AMP). Einsatzgebiete solcher Multicore-Prozessoren sind eingebettete Systeme, insbesondere audiovisuelle Unterhaltungselektronik, wie zum Beispiel Fernsehgeräte, DVD-Player, Camcorder, Spielkonsolen, Mobiltelefone usw. [T 06]. Eine typische Architektur für asymmetrisches Multicore besitzt der nachfolgend vorgestellte Cell-Prozessor mit der Cell Broadband Engine Architecture.

Schon im Jahr 2001 hat IBM den ***Power4*** Prozessor [G 01, BTR 02] auf den Markt gebracht (siehe Abb. 2.11). Dem folgte 2005 AMD mit dem ***Dual Core Opteron*** [A 06, KGA 03] und 2005/06 Intel mit dem ***Pentium D*** und dem ***Xeon DP*** [I 06].

Sun entwickelt den ***Niagara Chip*** auf Basis des ULTRASparc mit vier, sechs oder acht SPARC-Kernen [KAO 05, S 07]. Im Gegensatz zum Power4 Prozessor, der einen Shared L2 Cache besitzt, (siehe Abb. 2.11) besitzt beim Niagara Chip jeder Kern seinen eigenen L1-Cache und L2-Cache.

Toshiba, Sony und IBM entwickeln seit dem Jahr 2000 gemeinsam einen Prozessor namens ***Cell***, der in der Playstation 3 läuft und in HDTV-Geräten und Servern eingesetzt werden soll [C 07, KDH 05]. Die ***Cell Broadband Engine Architecture*** (CBEA) besitzt einen 64 Bit PowerPC-Kern (Power Processor Element – PPE) und acht speziell ausgelegte „Synergistic" Kerne (Synergistic Processor Elements – SPE) auf einem Chip, die mit einem Hochgeschwindigkeits-Bus (Element Interconnect Bus – EIB) verbunden sind. Zusätzlich auf dem Chip integriert ist ein Hochgeschwindigkeits-Speicher- und ein -I/O-Interface.

Das Power Processor Element (PPE) besitzt einen 32 Kbyte großen Instruktions- und Datencache und einen 512 Kbyte großen einheitlichen Cache auf der zweiten Ebene. Zur Reduktion der Hauptspeicherzugriffe der acht ***Synergistic Processor Elements*** (SPE) besitzen sie keinen Cache, sondern einen 256 Kbyte großen lokalen Speicher. Zum Daten-

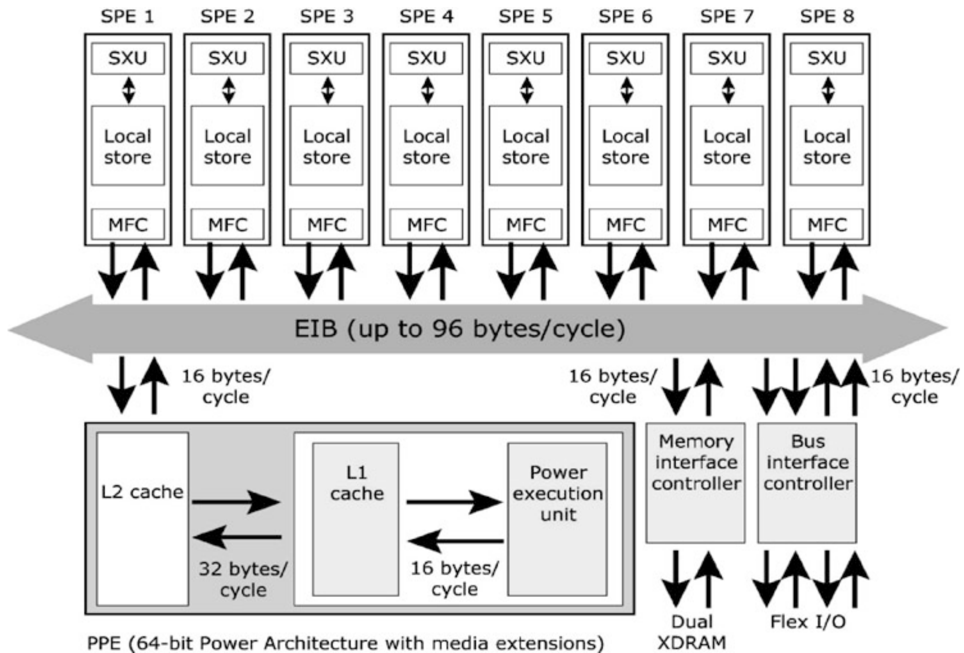


Abb. 2.12 Cell Broadband Engine

transfer zwischen dem lokalen Speicher und dem gemeinsamen Hauptspeicher über den Element Interconnect Bus (EIB) [KPP 06] besitzt jede SPE einen Memory Flow Controller (MFC) (siehe Abb. 2.12).

Synergistische Architekturen [GHF 06] sind Datenparallelität ausnutzende Architekturen und unterstützen einen hohen Thread Level-Parallelismus durch eine Vielzahl von Prozessoren auf einem Chip.

Das Betriebssystem für die Cell Broadband Engine Architecture (Abb. 2.12) ist ***Cell Linux***. Cell Linux ist, wie die restliche Cell-Software, freie Software (Open Source). Der gegenwärtige Stand von Cell Linux für die heterogenen Prozessoren, des Laders, der Compiler und Compilerprototypen sowie der Werkzeuge und des Debuggers ist in [GEM 07] beschrieben.

Auf der Integrated Solid State Circuits Conference (ISSCC) zeigte Intel den Prototypen eines Terascale-Prozessors mit 80 Kernen. Die Multi-Core-CPU soll eine Rechenleistung von einem Teraflop bieten und dabei nur 62 Watt benötigen. Intel präsentierte den ***80 Core-Prototypen*** auf dem Intel Developer Forum im September 2006 in San Francisco. Der Computerkonzern ist mit dem Prototyp eines Computerchips mit 80 Rechenkernen eigenen Angaben zufolge in eine neue Dimension vorgestoßen. Der Prozessor sei „kaum größer als ein Fingernagel“ und verbrauche mit 62 Watt weniger als viele heutige herkömmliche Chips. Die Rechenleistung liege im Teraflop-Bereich. Die Marktreife sei ca. 2010 erreicht.

http://www.springer.com/978-3-8348-1671-9

Masterkurs Parallele und Verteilte Systeme
Grundlagen und Programmierung von
Multicore-Prozessoren, Multiprozessoren, Cluster, Grid und
Cloud

Bengel, G.; Baun, C.; Kunze, M.; Stucky, K.-U.
2015, XXI, 495 S. 119 Abb., Softcover
ISBN: 978-3-8348-1671-9