

Kapitel 2

Architektur paralleler Plattformen

Wie in der Einleitung bereits angerissen wurde, hängen die Möglichkeiten einer parallelen Abarbeitung stark von den Gegebenheiten der benutzten Hardware ab. Wir wollen in diesem Kapitel daher den prinzipiellen Aufbau paralleler Plattformen vorstellen, auf die die auf Programmebene gegebene Parallelität abgebildet werden kann, um eine tatsächlich gleichzeitige Abarbeitung verschiedener Programmteile zu erreichen. In den Abschn. 2.1 und 2.2 beginnen wir mit einer kurzen Darstellung der innerhalb eines Prozessors oder Prozessorkerns zur Verfügung stehenden Möglichkeiten einer parallelen Verarbeitung. Hierbei wird deutlich, dass schon bei einzelnen Prozessorkernen eine Ausnutzung der verfügbaren Parallelität (auf Instruktionsebene) zu einer erheblichen Leistungssteigerung führen kann. Die weiteren Abschnitte des Kapitels sind Hardwarekomponenten von Parallelrechnern gewidmet. In den Abschn. 2.3 und 2.4 gehen wir auf die Kontroll- und Speicherorganisation paralleler Plattformen ein, indem wir zum einen die Flynnsche Klassifikation einführen und zum anderen Rechner mit verteiltem und Rechner mit gemeinsamem Speicher einander gegenüberstellen.

Eine weitere wichtige Komponente paralleler Hardware sind Verbindungsnetzwerke, die Prozessoren und Speicher bzw. verschiedene Prozessoren physikalisch miteinander verbinden. Verbindungsnetzwerke spielen auch bei Multi-core-Prozessoren eine große Rolle, und zwar zur Verbindung der Prozessorkerne untereinander sowie mit den Caches des Prozessorchips. Statische und dynamische Verbindungsnetzwerke und deren Bewertung anhand verschiedener Kriterien wie Durchmesser, Bisektionsbandbreite, Konnektivität und Einbettbarkeit anderer Netzwerke werden in Abschn. 2.5 eingeführt. Zum Verschicken von Daten zwischen zwei Prozessoren wird das Verbindungsnetzwerk genutzt, wozu meist mehrere Pfade im Verbindungsnetzwerk zur Verfügung stehen. In Abschn. 2.6 beschreiben wir Routingtechniken zur Auswahl eines solchen Pfades durch das Netzwerk sowie Switchingverfahren, die die Übertragung der Nachricht über einen vorgegebenen Pfad regeln. In Abschn. 2.7 werden Speicherhierarchien sequentieller und paralleler Plattformen betrachtet. Wir gehen insbesondere auf die bei parallelen Plattformen auftretenden Cachekohärenz- und Speicherkonsistenzprobleme ein. In

Abschn. 2.8 werden Prozesstechnologien wie simultanes Multithreading oder Multicore-Prozessoren zur Realisierung prozessorinterner Parallelverarbeitung auf Thread- oder Prozessebene vorgestellt. Abschließend enthält Abschn. 2.9 als Beispiel für die Architektur eines aktuellen Parallelrechners eine kurze Beschreibung der Architektur der IBM Blue Gene/Q Systeme.

2.1 Überblick über die Prozessorentwicklung

Bei der Prozessorentwicklung sind bestimmte Trends zu beobachten, die die Basis für Prognosen über die weitere voraussichtliche Entwicklung bilden. Ein wesentlicher Punkt ist die Performance-Entwicklung der Prozessoren, die von verschiedenen technologischen Faktoren beeinflusst wird. Ein wichtiger Faktor ist die Taktrate der Prozessoren, die die Zykluszeit des Prozessors und damit die Zeit für das Ausführen von Instruktionen bestimmt. Es ist zu beobachten, dass die Taktrate typischer Mikroprozessoren, wie sie z. B. in Desktop-Rechnern eingesetzt werden, zwischen 1987 und 2003 durchschnittlich um ca. 40 % pro Jahr gestiegen ist [76]. Seit 2003 ist die Taktrate dann ungefähr gleich geblieben und es sind in der nahen Zukunft auch keine signifikanten Steigerungen zu erwarten [73, 106]. Der Grund für diese Entwicklung liegt darin, dass mit einer Steigerung der Taktrate auch ein erhöhter Stromverbrauch einhergeht, der aufgrund von Leckströmen vor allem zu einer Erhöhung der Wärmeentwicklung führt, die wiederum einen erhöhten Aufwand für die Prozessorkühlung erforderlich macht. Mit der derzeitigen Luftkühlungstechnologie können jedoch ohne einen sehr großen Aufwand aktuell nur Prozessoren gekühlt werden, deren Taktrate ca. 3,3 GHz nicht wesentlich übersteigt.

Ein weiterer Einflussfaktor für die Performance-Entwicklung der Prozessoren ist die Anzahl der Transistoren eines Prozessorchips, die ein ungefähres Maß für die Komplexität des Schaltkreises ist und die pro Jahr um etwa 60 % bis 80 % wächst. Dadurch wird ständig mehr Platz für Register, Caches und Funktionseinheiten zur Verfügung gestellt. Diese von der Prozessorfertigungstechnik getragene, seit über 40 Jahren gültige empirische Beobachtung wird auch als Gesetz von Moore (engl. *Moore's law*) bezeichnet. Ein typischer Prozessor aus dem Jahr 2012 besteht aus ca. 1 bis 3 Milliarden Transistoren. Beispielsweise enthält ein Intel Core i7 Sandy Bridge Quadcore Prozessor ca. 995 Millionen Transistoren, ein Intel Core i7 Ivy Bridge-HE-4 Quadcore Prozessor ca. 1,4 Milliarden Transistoren und ein Intel Xeon Westmere-EX 10-Core Prozessor ca. 2,6 Milliarden Transistoren.

Zur Leistungsbewertung von Prozessoren können Benchmarks verwendet werden, die meist eine Sammlung von Programmen aus verschiedenen Anwendungsbereichen sind und deren Ausführung repräsentativ für die Nutzung eines Rechnersystems sein soll. Häufig verwendet werden die SPEC-Benchmarks (*System Performance and Evaluation Cooperative*), die zur Messung der Integer- bzw. Floating-Point-Performance eines Rechners dienen [83, 139, 170], vgl. auch

www.spec.org. Messungen mit diesen Benchmarks zeigen, dass zwischen 1986 und 2003 eine durchschnittliche Erhöhung der Performance von Prozessoren um ungefähr 50 % pro Jahr erreicht werden konnte [76]. Man beachte, dass dieser Zeitraum ungefähr mit dem oben genannten Zeitraum hoher jährlicher Steigerungen der Taktrate übereinstimmt. Seit 2003 kann nur noch eine jährliche Steigerung der Prozessor-Performance um ungefähr 22 % erreicht werden. Die Erhöhung der Leistung der Prozessoren über die Erhöhung der Taktrate hinaus lässt erkennen, dass die Erhöhung der Anzahl der Transistoren zu architektonischen Verbesserungen genutzt wurde, die die durchschnittliche Zeit für die Ausführung einer Instruktion reduzieren. Wir werden im Folgenden einen kurzen Überblick über diese Verbesserungen geben, wobei der Einsatz der Parallelverarbeitung im Vordergrund steht. Es sind vier Stufen der Prozessorentwicklung zu beobachten [32], deren zeitliche Entstehung sich z. T. überlappt:

1. **Parallelität auf Bitebene:** Bis etwa 1986 wurde die Wortbreite der Prozessoren, d. h. die Anzahl der Bits, die parallel zueinander verarbeitet werden können, sukzessive auf 32 Bits und bis Mitte der 90er Jahre allmählich auf 64 Bits erhöht. Diese Entwicklung wurde zum einen durch die Anforderungen an die Genauigkeit von Floating-Point-Zahlen getragen, zum anderen durch den Wunsch, einen genügend großen Adressraum ansprechen zu können. Die Entwicklung der Erhöhung der Wortbreite stoppte (vorläufig) bei einer Wortbreite von 64 Bits, da mit 64 Bits für die meisten Anwendungen eine ausreichende Genauigkeit für Floating-Point-Zahlen und die Adressierung eines ausreichend großen Adressraumes von 2^{64} Worten gegeben ist.
2. **Parallelität durch Pipelining:** Die Idee des Pipelinings auf Instruktionsebene besteht darin, die Verarbeitung einer Instruktion in Teilaufgaben zu zerlegen, die von zugeordneten Hardwareeinheiten (sogenannten Pipelinestufen) nacheinander ausgeführt werden. Eine typische Zerlegung besteht z. B. aus folgenden Stufen:
 - a) dem Laden der nächsten auszuführenden Instruktion (*fetch*),
 - b) dem Dekodieren dieser Instruktion (*decode*),
 - c) der Bestimmung der Adressen der Operanden und der Ausführung der Instruktion (*execute*) und
 - d) dem Zurückschreiben des Resultates (*write back*).

Der Vorteil der Pipelineverarbeitung besteht darin, dass die verschiedenen Pipelinestufen parallel zueinander arbeiten können (Fließbandprinzip), falls keine Kontroll- und Datenabhängigkeiten zwischen nacheinander auszuführenden Instruktionen vorhanden sind, vgl. Abb. 2.1. Zur Vermeidung von Wartezeiten sollte die Ausführung der verschiedenen Pipelinestufen etwa gleich lange dauern. Diese Zeit bestimmt dann den Maschinenzklus der Prozessoren. Im Idealfall wird bei einer Pipelineverarbeitung in jedem Maschinenzklus die Ausführung einer Instruktion beendet und die Ausführung der folgenden Instruktion begonnen. Damit bestimmt die Anzahl der Pipelinestufen den erreichbaren

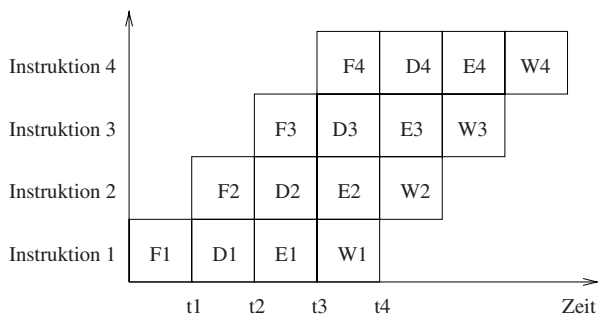


Abb. 2.1 Überlappende Ausführung voneinander unabhängiger Instruktionen nach dem Pipelining-Prinzip. Die Abarbeitung jeder Instruktion ist in vier Teilaufgaben zerlegt: *fetch* (F), *decode* (D), *execute* (E), *write back* (W)

Grad an Parallelität. Die Anzahl der Pipelinestufen hängt üblicherweise von der auszuführenden Instruktion ab und liegt meist zwischen 2 und 20 Stufen. Prozessoren, die zur Ausführung von Instruktionen Pipelineverarbeitung einsetzen, werden auch als (skalare) **ILP-Prozessoren** (*instruction level parallelism*) bezeichnet. Prozessoren mit relativ vielen Pipelinestufen heißen auch **superpipelined**. Obwohl der ausnutzbare Grad an Parallelität mit der Anzahl der Pipelinestufen steigt, kann die Zahl der verwendeten Pipelinestufen nicht beliebig erhöht werden, da zum einen die Instruktionen nicht beliebig in gleich große Teilaufgaben zerlegt werden können, und zum anderen eine vollständige Ausnutzung der Pipelinestufen oft durch Datenabhängigkeiten verhindert wird.

3. **Parallelität durch mehrere Funktionseinheiten: Superskalare** Prozessoren und **VLIW-Prozessoren** (*very long instruction word*) enthalten mehrere unabhängige Funktionseinheiten wie ALUs (*arithmetic logical unit*), FPUs (*floating point unit*), Speicherzugriffseinheiten (*load/store unit*) oder Sprungeinheiten (*branch unit*), die parallel zueinander verschiedene unabhängige Instruktionen ausführen und die zum Laden von Operanden auf Register zugreifen können. Damit ist eine weitere Steigerung der mittleren Verarbeitungsgeschwindigkeit von Instruktionen möglich. In Abschn. 2.2 geben wir einen kurzen Überblick über den Aufbau superskalarer Prozessoren.

Die Grenzen des Einsatzes parallel arbeitender Funktionseinheiten sind durch die Datenabhängigkeiten zwischen benachbarten Instruktionen vorgegeben, die für superskalare Prozessoren dynamisch zur Laufzeit des Programms ermittelt werden müssen. Dafür werden zunehmend komplexere Schedulingverfahren eingesetzt, die die auszuführenden Instruktionen den Funktionseinheiten zuordnen. Die Komplexität der Schaltkreise wird dadurch z. T. erheblich vergrößert, ohne dass dies mit einer entsprechenden Leistungssteigerung einhergeht. Simulationen haben außerdem gezeigt, dass die Möglichkeit des Absetzens von mehr

als vier Instruktionen pro Maschinenzyklus gegenüber einem Prozessor, der bis zu vier Instruktionen pro Maschinenzyklus absetzen kann, für viele Programme nur zu einer geringen Leistungssteigerung führen würde, da Datenabhängigkeiten und Sprünge oft eine parallele Ausführung von mehr als vier Instruktionen verhindern [32, 95].

4. **Parallelität auf Prozess- bzw. Threadebene:** Die bisher beschriebene Ausnutzung von Parallelität geht von *einem* sequentiellen Kontrollfluss aus, der vom Übersetzer zur Verfügung gestellt wird und der die gültigen Abarbeitungsreihenfolgen festlegt, d. h. bei Datenabhängigkeiten muss die vom Kontrollfluss vorgegebene Reihenfolge eingehalten werden. Dies hat den Vorteil, dass für die Programmierung eine sequentielle Programmiersprache verwendet werden kann und dass trotzdem eine parallele Abarbeitung von Instruktionen zumindest teilweise erreicht werden kann. Dem durch den Einsatz mehrerer Funktionseinheiten und Pipelining erreichbaren Potential an Parallelität sind jedoch Grenzen gesetzt, die – wie dargestellt – für aktuelle Prozessoren bereits erreicht sind. Nach dem Gesetz von Moore stehen aber ständig mehr Transistoren auf einer Chipfläche zur Verfügung. Diese können zwar z. T. für die Integration größerer Caches auf der Chipfläche genutzt werden, die Caches können aber auch nicht beliebig vergrößert werden, da größere Caches eine erhöhte Zugriffszeit erfordern, vgl. Abschn. 2.7.

Als eine zusätzliche Möglichkeit zur Nutzung der steigenden Anzahl von verfügbaren Transistoren werden seit 2005 sogenannte Multicore-Prozessoren gefertigt, die mehrere unabhängige Prozessorkerne auf der Chipfläche eines Prozessors integrieren. Jeder der Prozessorkerne hat alle Eigenschaften eines voll ausgebildeten Prozessors, hat also typischerweise mehrere Funktionseinheiten und setzt Pipelining zur Verarbeitung der Instruktionen ein. Im Unterschied zu bisherigen Einkern-Prozessoren muss jeder der Prozessorkerne eines Multicore-Prozessors mit einem separaten Kontrollfluss versorgt werden. Da die Prozessorkerne eines Multicore-Prozessors auf den Hauptspeicher und auf evtl. gemeinsame Caches gleichzeitig zugreifen können, ist ein koordiniertes Zusammenarbeiten dieser Kontrollflüsse erforderlich. Dazu können Techniken der parallelen Programmierung verwendet werden, wie sie in diesem Buch besprochen werden.

Wir werden im folgenden Abschnitt einen kurzen Überblick darüber geben, wie die Parallelität durch mehrere Funktionseinheiten innerhalb eines Prozessorkerns realisiert wird. Für eine detailliertere Darstellung verweisen wir auf [32, 75, 139, 171]. In Abschn. 2.8 gehen wir auf Techniken der Prozessororganisation wie simultanes Multithreading oder Multicore-Prozessoren ein, die eine explizite Spezifikation der Parallelität erfordern.

2.2 Parallelität innerhalb eines Prozessorkerns

Die meisten der heute verwendeten und entwickelten Prozessoren sind **superskalare** Prozessoren oder **VLIW-Prozessoren**, die mehrere Instruktionen *gleichzeitig* absetzen und unabhängig voneinander verarbeiten können. Dazu stehen mehrere Funktionseinheiten zur Verfügung, die unabhängige Instruktionen parallel zueinander bearbeiten können. Der Unterschied zwischen superskalaren Prozessoren und VLIW-Prozessoren liegt im Scheduling der Instruktionen: Ein Maschinenprogramm für superskalare Prozessoren besteht aus einer sequentiellen Folge von Instruktionen, die per Hardware auf die zur Verfügung stehenden Funktionseinheiten verteilt werden, wenn die Datenabhängigkeiten zwischen den Instruktionen dies erlauben. Dabei wird ein *dynamisches*, d.h. zur Laufzeit des Programmes arbeitendes Scheduling der Instruktionen verwendet, was eine zusätzliche Erhöhung der Komplexität der Hardware erfordert. Im Unterschied dazu wird für VLIW-Prozessoren ein *statisches* Scheduling verwendet, bei dem die Zuordnung von Instruktionen an Funktionseinheiten bereits vor dem Start des Programms festgelegt wird. Dazu erzeugt ein spezieller Übersetzer Maschinenprogramme mit Instruktionsworten, die für jede Funktionseinheit angeben, welche Instruktion zum entsprechenden Zeitpunkt ausgeführt wird. Ein Beispiel für ein solches statisches Schedulingverfahren ist Trace-Scheduling [44]. Die Instruktionsworte für VLIW-Prozessoren sind also in Abhängigkeit von der Anzahl der Funktionseinheiten recht lang, was den Prozessoren den Namen gegeben hat. Wichtigstes Beispiel für VLIW-Prozessoren ist die Intel IA64-Architektur, die für die Itanium-Serverprozessoren verwendet wird. Wir betrachten im Folgenden nur superskalare Prozessoren, da diese zzt. verbreiteter als VLIW-Prozessoren sind.

Abbildung 2.2a zeigt schematisch den typischen Aufbau eines superskalaren Prozessors. Zur Verarbeitung einer Instruktion wird diese von einer Zugriffseinheit (engl. *fetch unit*) über den Instruktionscache geladen und an eine Dekodiereinheit (engl. *decode unit*) weitergegeben, die die auszuführende Operation ermittelt. Damit mehrere Funktionseinheiten versorgt werden können, sind die Zugriffseinheit und die Dekodiereinheit in der Lage, in jedem Maschinenzyklus mehrere Instruktionen zu laden bzw. zu dekodieren. Nach der Dekodierung der Instruktionen werden diese, wenn keine Datenabhängigkeiten zwischen ihnen bestehen, an die zugehörigen Funktionseinheiten zur Ausführung weitergegeben. Die Ergebnisse der Berechnungen werden in die angegebenen Ergebnisregister zurückgeschrieben. Um bei superskalaren Prozessoren die Funktionseinheiten möglichst gut auszulasten, sucht der Prozessor in jedem Verarbeitungsschritt ausgehend von der aktuellen Instruktion nachfolgende Instruktionen, die wegen fehlender Datenabhängigkeiten direkt ausgeführt werden können (dynamisches Scheduling). Dabei spielen sowohl die Reihenfolge, in der die Instruktionen in die Funktionseinheiten geladen werden, als auch die Reihenfolge, in der Resultate der Instruktionen in die Register zurückgeschrieben werden, eine Rolle. Die größte

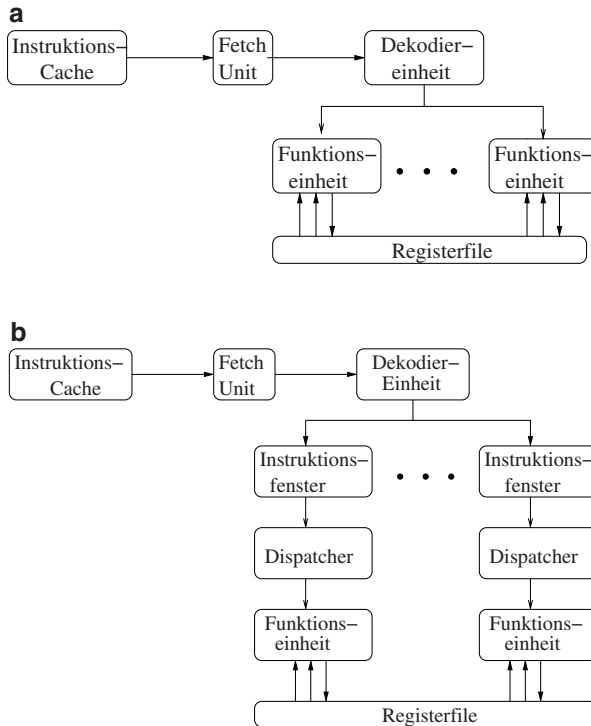


Abb. 2.2 Superskalare Prozessoren: **a** allgemeiner Aufbau, **b** Verwendung eines Instruktionsfensters

Flexibilität wird erreicht, wenn in beiden Fällen die Reihenfolge der Instruktionen im Maschinenprogramm nicht bindend ist (engl. *out-of-order issue*, *out-of-order completion*).

Um eine flexible Abarbeitung der Instruktionen zu realisieren, wird ein zusätzliches **Instruktionsfenster** (engl. *instruction window*, *reservation station*) verwendet, in dem die Dekodiereinheit bereits dekodierte Instruktionen ablegt, ohne zu überprüfen, ob diese aufgrund von Datenabhängigkeiten evtl. noch nicht ausgeführt werden können. Vor der Weitergabe einer Instruktion aus dem Instruktionsfenster an eine Funktionseinheit (*Dispatch*) wird ein Abhängigkeitstest durchgeführt, der sicherstellt, dass nur solche Instruktionen ausgeführt werden, deren Operanden verfügbar sind. Das Instruktionsfenster kann für jede Funktionseinheit getrennt oder für alle zentral realisiert werden. Abbildung 2.2b zeigt die Prozessororganisation für getrennte Instruktionsfenster. In der Praxis werden beide Möglichkeiten und Mischformen verwendet.

Im Instruktionsfenster abgelegte Instruktionen können nur dann ausgeführt werden, wenn ihre Operanden verfügbar sind. Werden die Operanden erst geladen,

wenn die Instruktion in die Funktionseinheit transportiert wird (*dispatch bound*), kann die Verfügbarkeit der Operanden mit Hilfe einer Anzeigetafel (engl. *score-board*) kontrolliert werden. Die Anzeigetafel stellt für jedes Register ein zusätzliches Bit zur Verfügung. Das Bit eines Registers wird auf 0 gesetzt, wenn eine Instruktion an das Instruktionsfenster weitergeleitet wird, die ihr Ergebnis in dieses Register schreibt. Das Bit wird auf 1 zurückgesetzt, wenn die Instruktion ausgeführt und das Ergebnis in das Register geschrieben wurde. Eine Instruktion kann nur dann an eine Funktionseinheit weitergegeben werden, wenn die Anzeigenbits ihrer Operanden auf 1 gesetzt sind. Wenn die Operandenwerte zusammen mit der Instruktion in das Instruktionsfenster eingetragen werden (engl. *issue bound*), wird für den Fall, dass die Operandenwerte noch nicht verfügbar sind, ein Platzhalter in das Instruktionsfenster eingetragen, der durch den richtigen Wert ersetzt wird, sobald die Operanden verfügbar sind. Die Verfügbarkeit wird mit einer Anzeigetafel überprüft. Um die Operanden im Instruktionsfenster auf dem aktuellen Stand zu halten, muss nach Ausführung jeder Instruktion ein evtl. errechnetes Resultat zum Auffüllen der Platzhalter im Instruktionsfenster verwendet werden. Dazu müssen alle Einträge des Instruktionsfensters überprüft werden. Instruktionen mit eingetragenen Operandenwerten sind ausführbar und können an eine Funktionseinheit weitergegeben werden.

In jedem Verarbeitungsschritt werden im Fall, dass ein Instruktionsfenster mehrere Funktionseinheiten versorgt, so viele Instruktionen wie möglich an diese weitergegeben. Wenn dabei die Anzahl der ausführbaren Instruktionen die der verfügbaren Funktionseinheiten übersteigt, werden diejenigen Instruktionen ausgewählt, die *am längsten* im Instruktionsfenster liegen. Wird diese Reihenfolge jedoch strikt beachtet (engl. *in-order dispatch*), so kann eine im Instruktionsfenster abgelegte Instruktion, deren Operanden nicht verfügbar sind, die Ausführung von später im Instruktionsfenster abgelegten, aber bereits ausführbaren Instruktionen verhindern. Um diesen Effekt zu vermeiden, wird meist auch eine andere Reihenfolge erlaubt (engl. *out-of-order dispatch*), wenn dies zu einer besseren Auslastung der Funktionseinheiten führt.

Die meisten aktuellen Prozessoren stellen sicher, dass die Instruktionen in der Reihenfolge beendet werden, in der sie im Programm stehen, so dass das Vorhandensein mehrerer Funktionseinheiten keinen Einfluss auf die Fertigstellungsreihenfolge der Instruktionen hat. Dies wird meist durch den Einsatz eines Umordnungspuffers (engl. *reorder buffer*) erreicht, in den die an die Instruktionsfenster abgegebenen Instruktionen in der vom Programm vorgegebenen Reihenfolge eingetragen werden, wobei für jede Instruktion vermerkt wird, ob sie sich noch im Instruktionsfenster befindet, gerade ausgeführt wird oder bereits beendet wurde. Im letzten Fall liegt das Ergebnis der Instruktion vor und kann in das Ergebnisregister geschrieben werden. Um die vom Programm vorgegebene Reihenfolge der Instruktionen einzuhalten, geschieht dies aber erst dann, wenn alle im Umordnungspuffer vorher stehenden Instruktionen ebenfalls beendet und ihre Ergebnisse in die zugehörigen Register geschrieben wurden. Nach der Aktualisierung des Er-

gebnisregisters werden die Instruktionen aus dem Umordnungspuffer entfernt. Für aktuelle Prozessoren können in einem Zyklus mehrere Ergebnisregister gleichzeitig beschrieben werden. Die zugehörigen Instruktionen werden aus dem Umordnungspuffer entfernt. Die Rate, mit der dies geschehen kann (engl. *retire rate*) stimmt bei den meisten Prozessoren mit der Rate überein, mit der Instruktionen an die Funktionseinheiten abgegeben werden können (engl. *issue rate*). In Abschn. 2.8.4 beschreiben wir als Beispiel eines superskalaren Prozessors die Architektur des Intel Core i7.

Diese kurze Darstellung der prinzipiellen Funktionsweise superskalarer Prozessoren zeigt, dass ein nicht unerheblicher Aufwand für die Ausnutzung von Parallelität auf Hardwareebene nötig ist und dass diese Form der Parallelität begrenzt ist.

2.3 Klassifizierung von Parallelrechnern

Bevor wir auf die Architektur von Multicore-Prozessoren näher eingehen, wollen wir eine grobe Klassifizierung von Parallelrechnern vorstellen, die für die nachfolgende Behandlung verschiedener Architekturen nützlich ist. Zuerst wollen wir uns jedoch der Frage zuwenden, was man überhaupt unter einem Parallelrechner versteht. Häufig verwendet wird folgende Definition [11]:

► **Parallelrechner** Ein Parallelrechner ist eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können.

Diese Definition ist bewusst vage gehalten, um die Vielzahl der entwickelten Parallelrechner zu erfassen und lässt daher auch viele z. T. wesentliche Details offen. Dazu gehören z. B. die Anzahl und Komplexität der Berechnungseinheiten, die Struktur der Verbindungen zwischen den Berechnungseinheiten, die Koordination der Arbeit der Berechnungseinheiten und die wesentlichen Eigenschaften der zu lösenden Probleme. Für eine genauere Untersuchung von Parallelrechnern ist eine Klassifizierung nach wichtigen Charakteristika nützlich. Wir beginnen mit der *Flynnschen Klassifizierung*, die häufig als erste grobe Unterscheidung von Parallelrechnern verwendet wird. Es handelt sich hierbei um eine eher theoretische Klassifizierung, die auch historisch am Anfang der Parallelrechnerentwicklung stand. Als erste Einführung in wesentliche Unterschiede möglichen parallelen Berechnungsverhaltens und als Abgrenzung gegenüber dem sequentiellen Rechnen ist diese Klassifizierung aber durchaus sinnvoll.

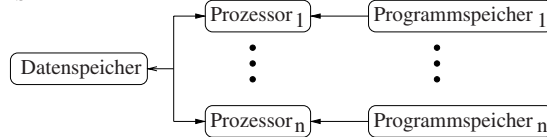
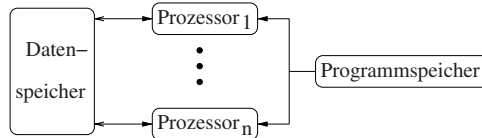
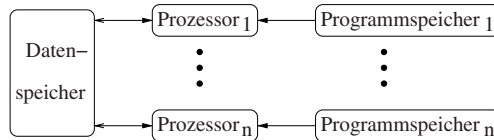
a SISD**b** MISD**c** SIMD**d** MIMD

Abb. 2.3 Darstellung der Modellrechner des Flynnschen Klassifikationsschemas: **a** SISD – Single Instruction, Single Data, **b** MISD – Multiple Instruction, Single Data, **c** SIMD – Single Instruction, Multiple Data und **d** MIMD – Multiple Instruction, Multiple Data

Flynnschen Klassifizierung

Die Flynnsche Klassifizierung [48] charakterisiert Parallelrechner nach der Organisation der globalen Kontrolle und den resultierenden Daten- und Kontrollflüssen. Es werden vier Klassen von Rechnern unterschieden:

- a) SISD – Single **I**nstruction, Single **D**ata,
- b) MISD – Multiple **I**nstruction, Single **D**ata,
- c) SIMD – Single **I**nstruction, Multiple **D**ata und
- d) MIMD – Multiple **I**nstruction, Multiple **D**ata.

Jeder dieser Klassen ist ein idealisierter Modellrechner zugeordnet, vgl. Abb. 2.3. Wir stellen im Folgenden die jeweiligen Modellrechner kurz vor.

Der **SISD**-Modellrechner hat *eine* Verarbeitungseinheit (Prozessor), die Zugriff auf *einen* Datenspeicher und *einen* Programmspeicher hat. In jedem Verarbeitungsschritt lädt der Prozessor eine Instruktion aus dem Programmspeicher, dekodiert

diese, lädt die angesprochenen Daten aus dem Datenspeicher in interne Register und wendet die von der Instruktion spezifizierte Operation auf die geladenen Daten an. Das Resultat der Operation wird in den Datenspeicher zurückgespeichert, wenn die Instruktion dies angibt. Damit entspricht der SISD-Modellrechner dem klassischen *von-Neumann-Rechnermodell*, das die Arbeitsweise aller sequentiellen Rechner beschreibt.

Der **MISD-Modellrechner** besteht aus *mehreren* Verarbeitungseinheiten, von denen jede Zugriff auf einen eigenen Programmspeicher hat. Es existiert jedoch nur ein gemeinsamer Zugriff auf den Datenspeicher. Ein Verarbeitungsschritt besteht darin, dass jeder Prozessor das *gleiche* Datum aus dem Datenspeicher erhält und eine Instruktion aus seinem Programmspeicher lädt. Diese evtl. unterschiedlichen Instruktionen werden dann von den verschiedenen Prozessoren parallel auf die erhaltene Kopie desselben Datums angewendet. Wenn ein Ergebnis berechnet wird und zurückgespeichert werden soll, muss jeder Prozessor den gleichen Wert zurückspeichern. Das zugrunde liegende Berechnungsmodell ist zu eingeschränkt, um eine praktische Relevanz zu besitzen. Es gibt daher auch keinen nach dem MISD-Prinzip arbeitenden Parallelrechner.

Der **SIMD-Modellrechner** besteht aus *mehreren* Verarbeitungseinheiten, von denen jede einen separaten Zugriff auf einen (gemeinsamen oder verteilten) Datenspeicher hat. Auf die Unterscheidung in gemeinsamen oder verteilten Datenspeicher werden wir in Abschn. 2.4 näher eingehen. Es existiert jedoch nur *ein* Programmspeicher, auf den eine für die Steuerung des Kontrollflusses zuständige Kontrolleinheit zugreift. Ein Verarbeitungsschritt besteht darin, dass jeder Prozessor von der Kontrolleinheit die *gleiche* Instruktion aus dem Programmspeicher erhält und ein separates Datum aus dem Datenspeicher lädt. Die Instruktion wird dann synchron von den verschiedenen Prozessoren parallel auf die jeweiligen Daten angewendet und eventuell errechnete Ergebnisse werden in den Datenspeicher zurückgeschrieben.

Der **MIMD-Modellrechner** besteht aus mehreren Verarbeitungseinheiten, von denen jede einen separaten Zugriff auf einen (gemeinsamen oder verteilten) Datenspeicher und auf einen lokalen Programmspeicher hat. Ein Verarbeitungsschritt besteht darin, dass jeder Prozessor eine separate Instruktion aus seinem lokalen Programmspeicher und ein separates Datum aus dem Datenspeicher lädt, die Instruktion auf das Datum anwendet und ein eventuell errechnetes Ergebnis in den Datenspeicher zurückschreibt. Dabei können die Prozessoren asynchron zueinander arbeiten.

Der Vorteil der SIMD-Rechner gegenüber MIMD-Rechnern liegt darin, dass SIMD-Rechner einfacher zu programmieren sind, da es wegen der streng synchronen Abarbeitung nur einen Kontrollfluss gibt, so dass keine Synchronisation auf Programmebene erforderlich ist. Ein Nachteil der SIMD-Rechner liegt in dem eingeschränkten Berechnungsmodell, das eine streng synchrone Arbeitsweise der Prozessoren erfordert. Daher muss eine bedingte Anweisung der Form

```
if (b==0) c=a; else c = a/b;
```

in zwei Schritten ausgeführt werden. Im ersten Schritt setzen alle Prozessoren, deren lokaler Wert von b Null ist, den Wert von c auf den Wert von a ; die anderen Prozessoren ignorieren diese Zuweisung. Im zweiten Schritt setzen alle Prozessoren, deren lokaler Wert von b nicht Null ist, den Wert von c auf $c = a/b$; die restlichen Prozessoren tun in diesem Schritt nichts. Das SIMD-Konzept ist in manchen Prozessoren als zusätzliche Möglichkeit für die prozessorinterne Datenverarbeitung integriert. Dazu stellen diese Prozessoren spezielle SIMD-Instruktionen für eine schnelle Verarbeitung großer, gleichförmiger Datenmengen zur Verfügung. Ein Beispiel dafür sind sogenannte *SIMD Multimedia Erweiterungen*, die von der Intel x86 Architektur ab dem Jahr 1999 in Form von SSE-Instruktionen (Streaming SIMD Extensions) bzw. ab 2010 in Form von AVX-Instruktionen (Advanced Vector Extensions) unterstützt werden. Auch die Verarbeitung von GPUs basiert auf dem SIMD-Prinzip, vgl. Abschn. 7.1. Das MIMD-Modell erfasst auch Multicore-Prozessoren sowie alle Parallelrechner, die aus Multicore-Prozessoren aufgebaut sind. Damit arbeiten fast alle der heute verwendeten Parallelrechner nach dem MIMD-Prinzip.

2.4 Speicherorganisation von Parallelrechnern

Fast alle der heute verwendeten Parallelrechner arbeiten nach dem MIMD-Prinzip, haben aber viele verschiedene Ausprägungen, so dass es sinnvoll ist, diese Klasse weiter zu unterteilen. Dabei ist eine Klassifizierung nach der Organisation des Speichers gebräuchlich, wobei zwischen der physikalischen Organisation des Speichers und der Sicht des Programmierers auf den Speicher unterschieden werden kann. Bei der physikalischen Organisation des Speichers unterscheidet man zwischen *Rechnern mit physikalisch gemeinsamem Speicher*, die auch *Multiprozessoren* genannt werden, und *Rechnern mit physikalisch verteiltem Speicher*, die auch *Multicomputer* genannt werden. Weiter sind *Rechner mit virtuell gemeinsamem Speicher* zu nennen, die als Hybridform angesehen werden können, vgl. auch Abb. 2.4.

Bzgl. der Sicht des Programmierers wird zwischen *Rechnern mit verteiltem Adressraum* und *Rechnern mit gemeinsamem Adressraum* unterschieden. Die Sicht des Programmierers muss dabei nicht unbedingt mit der physikalischen Organisation des Rechners übereinstimmen, d. h. ein Rechner mit physikalisch verteiltem Speicher kann dem Programmierer durch eine geeignete Programmierungsumgebung als Rechner mit gemeinsamem Adressraum erscheinen und umgekehrt. Wir betrachten in diesem Abschnitt die physikalische Speicherorganisation von Parallelrechnern.

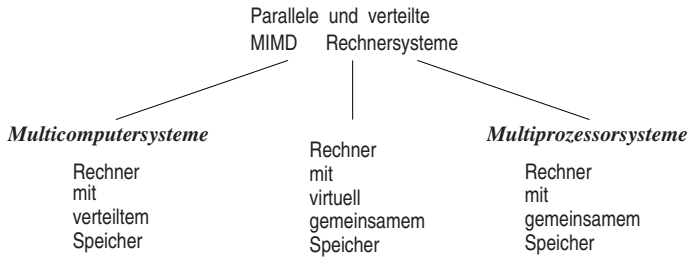


Abb. 2.4 Unterteilung der MIMD-Rechner bzgl. ihrer Speicherorganisation

2.4.1 Rechner mit physikalisch verteiltem Speicher

Rechner mit physikalisch verteiltem Speicher (auch als **DMM** für engl. *distributed memory machine* bezeichnet) bestehen aus mehreren Verarbeitungseinheiten (Knoten) und einem Verbindungsnetzwerk, das die Knoten durch physikalische Leitungen verbindet, über die Daten übertragen werden können. Ein Knoten ist eine selbständige Einheit aus einem oder mehreren Prozessoren, lokalem Speicher und evtl. I/O-Anschlüssen. Eine schematisierte Darstellung ist in Abb. 2.5a wiedergegeben.

Die Daten eines Programmes werden in einem oder mehreren der lokalen Speicher abgelegt. Alle lokalen Speicher sind *privat*, d. h. nur der zugehörige Prozessor kann direkt auf die dort abgelegten Daten zugreifen. Wenn ein Prozessor zur Verarbeitung seiner lokalen Daten auch Daten aus lokalen Speichern anderer Prozessoren benötigt, so müssen diese durch Nachrichtenaustausch über das Verbindungsnetzwerk bereitgestellt werden. Rechner mit verteiltem Speicher sind daher eng verbunden mit dem Programmiermodell der **Nachrichtenübertragung** (engl. *message-passing programming model*), das auf der Kommunikation zwischen kooperierenden sequentiellen Prozessen beruht und auf das wir in Kap. 3 näher eingehen werden. Zwei miteinander kommunizierende Prozesse P_A und P_B auf verschiedenen Knoten A und B des Rechners setzen dabei zueinander komplementäre Sende- und Empfangsbefehle ab. Sendet P_A eine Nachricht an P_B , so führt P_A einen Sendebehl aus, in dem die zu verschickende Nachricht und das Ziel P_B festgelegt wird. P_B führt einen Empfangsbefehl mit Angabe eines Empfangspuffers, in dem die Nachricht gespeichert werden soll, und des sendenden Prozesses P_A aus.

Die Architektur von Rechnern mit verteiltem Speicher hat im Laufe der Zeit eine Reihe von Entwicklungen erfahren, und zwar insbesondere im Hinblick auf das benutzte Verbindungsnetzwerk bzw. den Zusammenschluss von Netzwerk und Knoten. Frühe Multicomputer verwendeten als Verbindungsnetzwerk meist **Punkt-zu-Punkt-Verbindungen** zwischen Knoten. Ein Knoten ist dabei mit einer festen Menge von anderen Knoten durch physikalische Leitungen verbunden. Die Struktur des Verbindungsnetzwerkes kann als Graph dargestellt werden, dessen Kno-

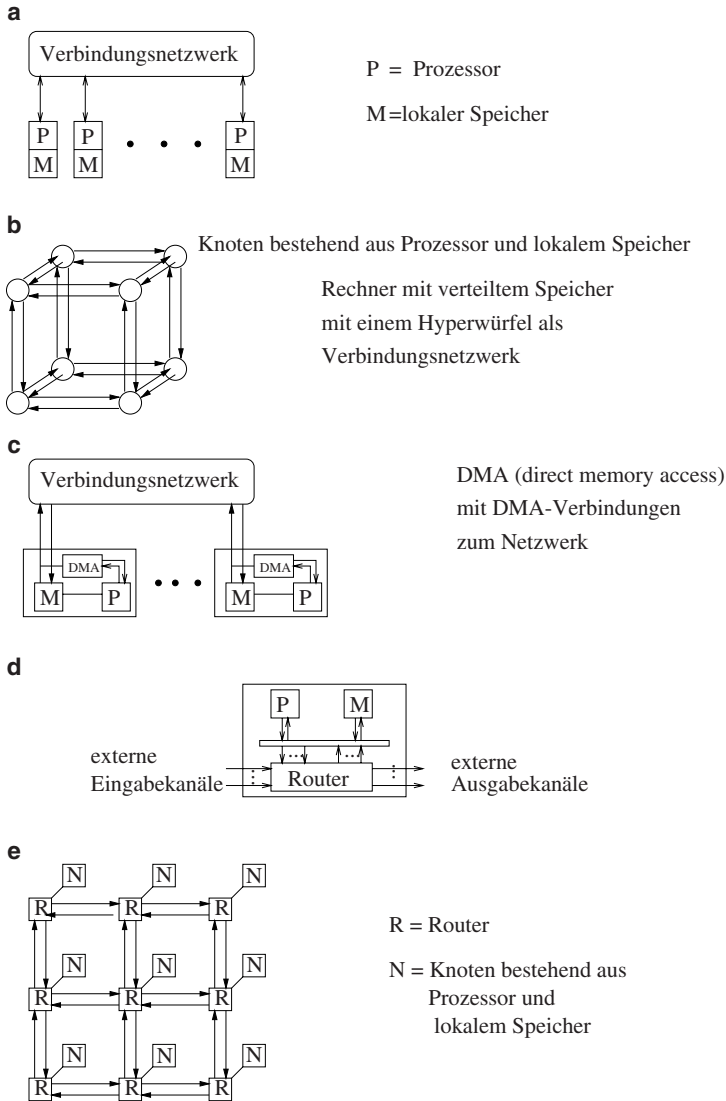


Abb. 2.5 Illustration zu Rechnern mit verteiltem Speicher, **a** Abstrakte Struktur, **b** Rechner mit verteiltem Speicher und Hyperwürfel als Verbindungsstruktur, **c** DMA (direct memory access), **d** Prozessor-Speicher-Knoten mit Router und **e** Verbindungsnetzwerk in Form eines Gitters zur Verbindung der Router der einzelnen Prozessor-Speicher-Knoten

ten die Prozessoren und dessen Kanten die physikalischen Verbindungsleitungen, auch Links genannt, darstellen. Die Struktur des Graphen ist meist regelmäßig. Häufig verwendete Netzwerke sind Gitter- und Torusnetzwerke sowie *Hyperwür-*

fel-Netzwerke, das auch in Abb. 2.5b zur Veranschaulichung verwendet wird. Bei solchen Verbindungsnetzwerken mit Punkt-zu-Punkt-Verbindungen ist die Kommunikation durch die Gestalt des Netzwerkes vorgegeben, da Knoten nur mit ihren direkten Nachbarn kommunizieren können. Nur direkte Nachbarn können in Send- und Empfangsoperationen als Absender bzw. Empfänger genannt werden. Kommunikation kann nur stattfinden, wenn benachbarte Knoten *gleichzeitig* auf den verbindenden Link schreiben bzw. von ihm lesen. Es werden zwar typischerweise Puffer bereitgestellt, in denen die Nachricht zwischengespeichert werden kann, diese sind aber relativ klein, so dass eine größere Nachricht nicht vollständig im Puffer abgelegt werden kann und so die Gleichzeitigkeit des Sendens und Empfangens notwendig wird. Dadurch ist die parallele Programmierung sehr stark an die verwendete Netzwerkstruktur gebunden und zum Erstellen von effizienten parallelen Programmen sollten parallele Algorithmen verwendet werden, die die vorhandenen Punkt-zu-Punkt-Verbindungen des vorliegenden Netzwerkes effizient ausnutzen [6, 111].

Das Hinzufügen eines speziellen **DMA-Controllers** (*DMA – direct memory access*) für den direkten Datentransfer zwischen lokalem Speicher und I/O-Anschluss ohne Einbeziehung des Prozessors entkoppelt die eigentliche Kommunikation vom Prozessor, so dass Send- und Empfangsoperationen nicht genau zeitgleich stattfinden müssen, siehe Abb. 2.5c. Der Sender kann nun eine Kommunikation initiieren und dann weitere Arbeit ausführen, während der Sendebefehl unabhängig beendet wird. Beim Empfänger wird die Nachricht vom DMA-Controller empfangen und in einem speziell dafür vorgesehenen Speicherbereich abgelegt. Wird beim Empfänger eine zugehörige Empfangsoperation ausgeführt, so wird die Nachricht aus dem Zwischenspeicher entnommen und in dem im Empfangsbefehl angegebenen Empfangspuffer gespeichert. Die ausführbaren Kommunikationen sind aber immer noch an die Nachbarschaftsstruktur im Netzwerk gebunden. Kommunikation zwischen Knoten, die keine physikalischen Nachbarn sind, wird durch Software gesteuert, die die Nachrichten entlang aufeinanderfolgender Punkt-zu-Punkt-Verbindungen verschickt. Dadurch sind die Laufzeiten für die Kommunikation mit weiter entfernt liegenden Knoten erheblich größer als die Laufzeiten für Kommunikation mit physikalischen Nachbarn und die Verwendung von speziell für das Verbindungsnetzwerk entworfenen Algorithmen ist aus Effizienzgründen weiterhin empfehlenswert.

Moderne Multicomputer besitzen zu jedem Knoten einen **HardwareRouter**, siehe Abb. 2.5d. Der Knoten selbst ist mit dem Router verbunden. Die Router allein bilden das eigentliche Netzwerk, das hardwaremäßig die Kommunikation mit allen auch weiter entfernten Knoten übernimmt, siehe Abb. 2.5e. Die abstrakte Darstellung des Rechners mit verteiltem Speicher in Abb. 2.5a wird also in dieser Variante mit Hardware-Routern am ehesten erreicht. Das hardwareunterstützte Routing verringert die Kommunikationszeit, da Nachrichten, die zu weiter entfernt liegenden Knoten geschickt werden, von Routern entlang eines ausgewählten Pfades weitergeleitet werden, so dass keine Mitarbeit der Prozessoren in den Knoten des Pfades erforderlich ist. Insbesondere unterscheiden sich die Zeiten für den Nachrichten-

austausch mit Nachbarknoten und mit entfernt gelegenen Knoten in dieser Variante nicht wesentlich. Da jeder physikalische I/O-Kanal des Hardware-Routers nur von einer Nachricht zu einem Zeitpunkt benutzt werden kann, werden Puffer am Ende von Eingabe- und Ausgabekanälen verwendet, um Nachrichten zwischenspeichern zu können. Zu den Aufgaben des Routers gehört die Ausführung von Pipelining bei der Nachrichtenübertragung und die Vermeidung von Deadlocks. Dies wird in Abschn. 2.6.1 näher erläutert.

Rechner mit physikalisch verteiltem Speicher sind technisch vergleichsweise einfach zu realisieren, da die einzelnen Knoten im Extremfall normale Desktop-Rechner sein können, die mit einem schnellen Netzwerk miteinander verbunden werden. Die Programmierung von Rechnern mit physikalisch verteiltem Speicher gilt als schwierig, da im natürlich zugehörigen Programmiermodell der Nachrichtenübertragung der Programmierer für die lokale Verfügbarkeit der Daten verantwortlich ist und alle Datentransfers zwischen den Knoten durch Sende- und Empfangsanweisungen explizit steuern muss. Üblicherweise dauert der Austausch von Daten zwischen Prozessoren durch Sende- und Empfangsoperationen wesentlich länger als ein Zugriff eines Prozessors auf seinen lokalen Speicher. Je nach verwendetem Verbindungsnetzwerk und verwendeter Kommunikationsbibliothek kann durchaus ein Faktor von 100 und mehr auftreten. Die Platzierung der Daten kann daher die Laufzeit eines Programmes entscheidend beeinflussen. Sie sollte so erfolgen, dass die Anzahl der Kommunikationsoperationen und die Größe der zwischen den Prozessoren verschickten Datenblöcke möglichst klein ist.

Ein **Cluster** ist eine parallele Plattform, die in ihrer Gesamtheit aus einer Menge von vollständigen, miteinander durch ein Kommunikationsnetzwerk verbundenen Rechnern besteht und das in seiner Gesamtheit als *ein einziger Rechner* angesprochen und benutzt wird. Von außen gesehen sind die einzelnen Komponenten eines Clusters anonym und gegenseitig austauschbar. Der Popularitätsgewinn des Clusters als parallele Plattform auch für die Anwendungsprogrammierung begründet sich in der Entwicklung von standardmäßiger Hochgeschwindigkeitskommunikation, wie z. B. FCS (*Fibre Channel Standard*), ATM (*Asynchronous Transfer Mode*), SCI (*Scalable Coherent Interconnect*), Gigabit Ethernet, 10 Gigabit Ethernet, Myrinet oder Infiniband, vgl. [142, 75, 139]. Ein natürliches Programmiermodell ist das Message-Passing-Modell, das durch Kommunikationsbibliotheken wie MPI und PVM unterstützt wird, siehe Kap. 5. Diese Bibliotheken basieren z. T. auf Standardprotokollen wie TCP/IP [108, 141].

Von den *verteilten Systemen* (die wir hier nicht näher behandeln werden) unterscheiden sich Clustersysteme dadurch, dass sie eine geringere Anzahl von Knoten (also einzelnen Rechnern) enthalten, Knoten *nicht* individuell angesprochen werden und oft das gleiche Betriebssystem auf allen Knoten benutzt wird. Clustersysteme können mit Hilfe spezieller Middleware-Software wie z. B. das Globus-Toolkit, vgl. www.globus.org [52], zu Gridsystemen zusammengeschlossen werden. Diese erlauben ein koordiniertes Zusammenarbeiten über Clustergrenzen hinweg. Die genaue Steuerung der Abarbeitung der Anwendungsprogramme wird von der Middleware-Software übernommen. Clustersysteme werden auch häufig zur Bereit-

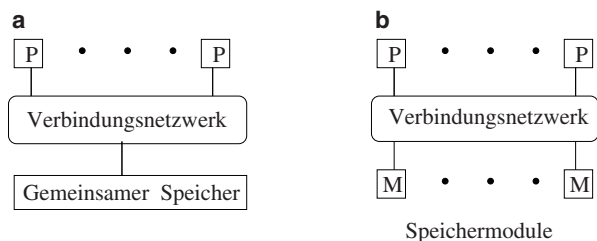


Abb. 2.6 Illustration eines Rechners mit gemeinsamem Speicher, **a** Abstrakte Sicht und **b** Realisierung des gemeinsamen Speichers mit Speichermodulen

stellung von Diensten im Bereich des Cloud-Computing verwendet, wobei jedem Nutzer durch die Cloud-Infrastruktur ein Teil des Clusters als virtuelle Ressource zur Verfügung gestellt wird. Der Nutzer kann die virtuelle Ressource entsprechend seiner Berechnungsanforderungen individuell dimensionieren. Dabei kann eine virtuelle Ressource einzelne Clusterknoten oder Teile davon, aber auch mehrere Knoten umfassen. Beispiele für Cloud-Computing-Infrastrukturen sind die Amazon Elastic Compute Cloud (EC2) oder die Windows Azure Plattform.

2.4.2 Rechner mit physikalisch gemeinsamem Speicher

Ein Rechner mit physikalisch gemeinsamem Speicher (auch als **SMM** für engl. *shared memory machine* bezeichnet) besteht aus mehreren Prozessoren oder Prozessorkernen, einem **gemeinsamen** oder **globalen** Speicher und einem Verbindungsnetzwerk, das Prozessoren und globalen Speicher durch physikalische Leitungen verbindet, über die Daten in den gemeinsamen Speicher geschrieben bzw. aus ihm gelesen werden können. Die Prozessorkerne eines Multicore-Prozessors greifen auf einen gemeinsamen Speicher zu und bilden daher eine Rechnerplattform mit physikalisch gemeinsamem Speicher, vgl. Abschn. 2.8. Der gemeinsame Speicher setzt sich meist aus einzelnen Speichermodulen zusammen, die gemeinsam einen einheitlichen Adressraum darstellen, auf den alle Prozessoren lesend (*load*) oder schreibend (*store*) zugreifen können. Eine abstrakte Darstellung zeigt Abb. 2.6.

Prinzipiell kann durch entsprechende Softwareunterstützung jedes parallele Programmiermodell auf Rechnern mit gemeinsamem Speicher unterstützt werden. Ein natürlicherweise geeignetes paralleles Programmiermodell ist die Verwendung **gemeinsamer Variablen** (engl. *shared variables*). Hierbei wird die Kommunikation und Kooperation der parallel arbeitenden Prozessoren über den gemeinsamen Speicher realisiert, indem Variablen von einem Prozessor beschrieben und von einem anderen Prozessor gelesen werden. *Gleichzeitiges* unkoordiniertes Schreiben verschiedener Prozessoren auf dieselbe Variable stellt in diesem Modell eine Operation dar, die zu nicht vorhersagbaren Ergebnissen führen kann. Für die Vermeidung

dieses sogenannten *Schreibkonfliktes* gibt es unterschiedliche Ansätze, die in den Kap. 3 und 6 besprochen werden.

Für den Programmierer bietet ein Rechnermodell mit gemeinsamem Speicher große Vorteile gegenüber einem Modell mit verteiltem Speicher, weil die Kommunikation über den gemeinsamen Speicher einfacher zu programmieren ist und weil der gemeinsame Speicher eine gute Speicherausnutzung ermöglicht, da ein Replizieren von Daten nicht notwendig ist. Für die Hardware-Hersteller stellt die Realisierung von Rechnern mit gemeinsamem Speicher aber eine größere Herausforderung dar, da ein Verbindungsnetzwerk mit einer hohen Übertragungskapazität eingesetzt werden muss, um jedem Prozessor schnellen Zugriff auf den globalen Speicher zu geben, wenn nicht das Verbindungsnetzwerk zum Engpass der effizienten Ausführung werden soll. Die Erweiterbarkeit auf eine große Anzahl von Prozessoren ist daher oft schwieriger zu realisieren als für Rechner mit physikalisch verteiltem Speicher. Rechner mit gemeinsamem Speicher arbeiten aus diesem Grund meist mit einer geringen Anzahl von Prozessoren.

Eine spezielle Variante von Rechnern mit gemeinsamem Speicher sind die **symmetrischen Multiprozessoren**, auch **SMP** (*symmetric multiprocessor*) genannt [32]. SMP-Maschinen bestehen üblicherweise aus einer kleinen Anzahl von Prozessoren, die über einen Crossbar-Switch oder über einen zentralen Bus miteinander verbunden sind. Jeder Prozessor hat dadurch Zugriff auf den gemeinsamen Speicher und die angeschlossenen I/O-Geräte. Es gibt keine zusätzlichen privaten Speicher für Prozessoren oder spezielle I/O-Prozessoren. Lokale Caches für Prozessoren sind aber üblich. Das Wort *symmetrisch* in der Bezeichnung SMP bezieht sich auf die Prozessoren und bedeutet, dass alle Prozessoren die gleiche Funktionalität und die gleiche Sicht auf das Gesamtsystem haben, d. h. insbesondere, dass die Dauer eines Zugriffs auf den gemeinsamen Speicher für jeden Prozessor unabhängig von der zugegriffenen Speicheradresse gleich lange dauert. In diesem Sinne ist jeder aus mehreren Prozessorkernen bestehende Multicore-Prozessor ein SMP-System. Wenn sich die zugegriffene Speicheradresse im lokalen Cache eines Prozessors befindet, findet der Zugriff entsprechend schneller statt. Die Zugriffszeit auf seinen lokalen Cache ist für jeden Prozessor gleich. SMP-Rechner werden üblicherweise mit einer kleinen Anzahl von Prozessoren betrieben, weil z. B. bei Verwendung eines zentralen Busses nur eine konstante Bandbreite zur Verfügung steht, aber die Speicherzugriffe aller Prozessoren nacheinander über den Bus laufen müssen. Wenn zu viele Prozessoren an den Bus angeschlossen sind, steigt die Gefahr von Kollisionen bei Speicherzugriffen und damit die Gefahr einer Verlangsamung der Verarbeitungsgeschwindigkeit der Prozessoren. Zum Teil kann dieser Effekt durch den Einsatz von Caches und geeigneten Cache-Kohärenzprotokollen abgemildert werden, vgl. Abschn. 2.7.2. Die maximale Anzahl von Prozessoren liegt für SMP-Rechner meist bei 32 oder 64 Prozessoren.

Das Vorhandensein mehrerer Prozessorkerne ist bei der Programmierung von SMP-Systemen prinzipiell sichtbar. Insbesondere das Betriebssystem muss die verschiedenen Prozessorkerne explizit ansprechen. Ein geeignetes paralleles Programmiermodell ist das *Thread-Programmiermodell*, wobei zwischen *Betriebssystem-*

Threads (engl. *kernel threads*), die vom Betriebssystem erzeugt und verwaltet werden, und Benutzer-Threads (engl. *user threads*), die vom Programm erzeugt und verwaltet werden, unterschieden werden kann, siehe Abschn. 6.1. Für Anwendungsprogramme kann das Vorhandensein mehrerer Prozessorkerne durch das Betriebssystem verborgen werden, d. h. Anwenderprogramme können normale sequentielle Programme sein, die vom Betriebssystem auf einzelne Prozessorkerne abgebildet werden. Die Auslastung aller Prozessorkerne wird dadurch erreicht, dass verschiedene Programme evtl. verschiedener Benutzer zur gleichen Zeit auf unterschiedlichen Prozessorkernen laufen.

SMP-Systeme können zu größeren Parallelrechnern zusammengesetzt werden, indem ein Verbindungsnetzwerk eingesetzt wird, das den Austausch von Nachrichten zwischen Prozessoren verschiedener SMP-Maschinen erlaubt. Alternativ können Rechner mit gemeinsamem Speicher hierarchisch zu größeren Clustern zusammengesetzt werden, was z. B. zu Hierarchien von Speichern führt, vgl. Abschn. 2.7. Durch Einsatz geeigneter Kohärenzprotokolle kann wieder ein logisch gemeinsamer Adressraum gebildet werden, d. h. jeder Prozessor kann jede Speicherzelle direkt adressieren, auch wenn sie im Speicher eines anderen SMP-Systems liegt. Da die Speichermodule physikalisch getrennt sind und der gemeinsame Adressraum durch den Einsatz eines Kohärenzprotokolls realisiert wird, spricht man auch von Rechnern mit **virtuell-gemeinsamem Speicher** (engl. *virtual shared memory*).

Dadurch, dass der Zugriff auf die gemeinsamen Daten tatsächlich physikalisch verteilt auf lokale, gruppenlokale oder globale Speicher erfolgt, unterscheiden sich die Speicherzugriffe zwar (aus der Sicht des Programmierers) nur in der angegebenen Speicheradresse, die Zugriffe können aber in Abhängigkeit von der Speicheradresse zu *unterschiedlichen Speicherzugriffszeiten* führen. Der Zugriff eines Prozessors auf gemeinsame Variablen, die in dem ihm physikalisch am nächsten liegenden Speichermodul abgespeichert sind, wird schneller ausgeführt als Zugriffe auf gemeinsame Variablen mit Adressen, die einem physikalisch weiter entfernt liegenden Speicher zugeordnet sind. Zur Unterscheidung dieses für die Ausführungszeit wichtigen Phänomens der Speicherzugriffszeit wurden die Begriffe **UMA-System** (Uniform Memory Access) und **NUMA-System** (Non-Uniform Memory Access) eingeführt. UMA-Systeme weisen für alle Speicherzugriffe eine einheitliche Zugriffszeit auf. Bei NUMA-Systemen hängt die Speicherzugriffszeit von der relativen Speicherstelle einer Variablen zum zugreifenden Prozessor ab, vgl. Abb. 2.7.

2.4.3 Reduktion der Speicherzugriffszeiten

Allgemein stellt die Speicherzugriffszeit eine kritische Größe beim Entwurf von Rechnern dar, insbesondere auch für Rechner mit gemeinsamem Speicher. Dies liegt daran, dass die technologische Entwicklung der letzten Jahre zu erheblicher Leistungssteigerung bei den Prozessoren führte, die Speicherkapazität in einer

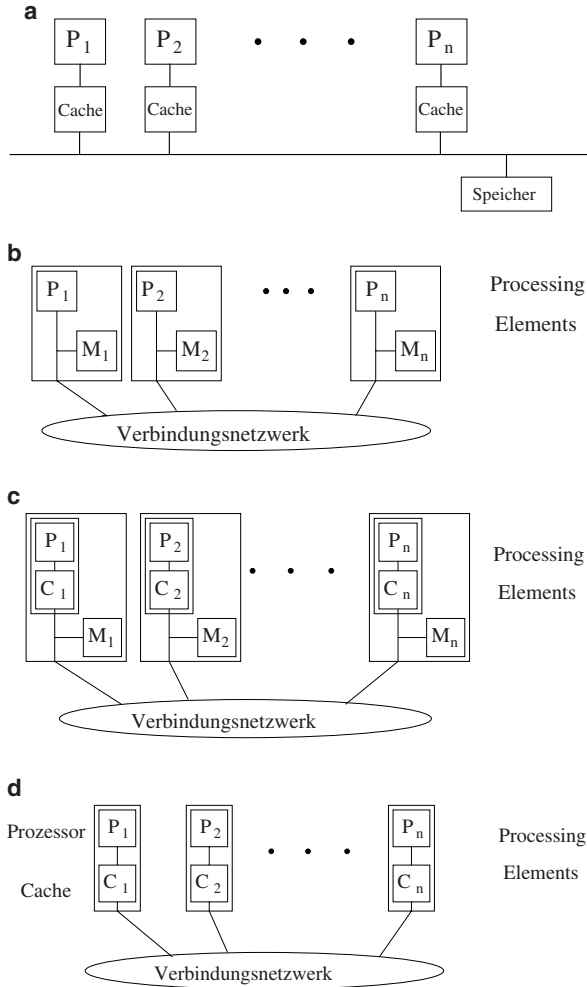


Abb. 2.7 Illustration der Architektur von Rechnern mit gemeinsamem Speicher: **a** SMP – symmetrische Multiprozessoren, **b** NUMA – non-uniform memory access, **c** CC-NUMA – cache coherent NUMA und **d** COMA – cache only memory access

ähnlichen Größenordnung anstieg, die Speicherzugriffszeiten sich jedoch nicht in dem gleichen Maße verringerten [32, 76]. Konkret stieg zwischen 1986 und 2003 die mit Hilfe der SPEC-Benchmarks gemessene Performance von Mikroprozessoren um ca. 50 % pro Jahr, seit 2003 um ca. 22 % pro Jahr, vgl. Abschn. 2.1. Für die Speicherkapazität von DRAM (dynamic random access memory) Chips, die zum Aufbau des Hauptspeichers von Rechnern verwendet werden, kann zwischen 1990 und 2003 eine jährliche Zunahme von ca. 60 % pro Jahr beobachtet

werden, seit 2003 liegt die jährliche Zunahme nur noch bei 25 %–40 % [76]. Die Zugriffs-Performance von DRAM-Chips zeigt jedoch wesentlich geringere jährliche Verbesserungsraten.

Zur Performance-Bewertung von DRAM-Chips können die Antwortzeit, auch Latenz genannt, sowie die Bandbreite, auch als Durchsatz bezeichnet, herangezogen werden. Dabei umfasst die **Latenz** die Gesamtzeit, die bis zum vollständigen Abschluss einer Speicherzugriffsoperation vergeht; die Latenz wird meist in Mikrosekunden oder Nanosekunden angegeben. Dagegen bezeichnet die **Bandbreite** die Anzahl der Datenelemente, die pro Zeiteinheit von einem DRAM-Chip ausgelesen werden können; eine typische Einheit ist Megabyte pro Sekunde (MB/s) oder Gigabyte pro Sekunde (GB/s). Für die Latenz von DRAM-Chips kann seit 1980 bis heute eine jährliche Verringerung von ca. 5 % beobachtet werden [76]. Im Jahr 2012 liegt die Latenz für die neueste verfügbare Technologie (DDR3, *Double Data Rate*) meist zwischen 24 und 30 ns. Bei der Bandbreite beträgt die jährliche Steigerung etwas über 10 %. Für die ab 2014 erwartete DDR4-Technologie wird eine Bandbreite zwischen 2100 MB/s und 3200 MB/s pro DRAM-Chip erwartet, wobei meist mehrere DRAM-Chips (üblicherweise zwischen 4 und 16) zu DIMMs (*dual inline memory module*) zusammengeschlossen werden.

Da die Entwicklung von Latenz und Bandbreite bei Speicherzugriffen nicht mit der Entwicklung der Rechenleistung Schritt halten konnte, stellen Speicherzugriffe für einige Anwendungen einen wesentlichen Engpass bei der Erzielung von hohen Rechenleistungen dar. Auch die Leistungsfähigkeit von Rechnern mit gemeinsamem Speicher hängt somit wesentlich davon ab, wie Speicherzugriffe gestaltet bzw. Speicherzugriffszeiten verringert werden können.

Zur Verhinderung großer Verzögerungszeiten beim Zugriff auf den lokalen Speicher werden im wesentlichen zwei Ansätze verfolgt [11]: der Einsatz von **lokalen Caches** zur Zwischenspeicherung von häufig benutzten Werten sowie die Simulation von **virtuellen Prozessoren** durch jeden physikalischen Prozessor (verschränktes Multithreading). Lokale Caches werden von allen gängigen Mikroprozessoren eingesetzt, verschränktes Multithreading wird z. B. für GPUs eingesetzt.

Einsatz von Caches

Caches oder *Cache-Speicher* sind kleine schnelle Speicher, die zwischen Prozessor und Hauptspeicher geschaltet werden und in denen, von der Hardware gesteuert, häufig zugriffene Daten gehalten werden. Die Caches sind meist auf dem Prozessorchip integriert, so dass ein Zugriff auf die Daten im Cache lokal auf dem Prozessorchip realisiert werden kann, ohne dass ein externer Speicherzugriff erforderlich ist. Das Ziel besteht also darin, durch den Einsatz der Caches die Menge der extern zugriffenen Daten zu reduzieren. Technisch wird dies so umgesetzt, dass jeder aus dem globalen Speicher geladene Wert automatisch im Cache des zugreifenden Prozessors zwischengespeichert wird. Dabei wird vor jedem Zugriff auf den globalen Speicher untersucht, ob die angeforderte Speicherzelle bereits im Cache

enthalten ist. Wenn dies der Fall ist, wird der Wert aus dem Cache geladen und der Zugriff auf den globalen Speicher entfällt. Dies führt dazu, dass Speicherzugriffe, die über den Cache erfolgen können, wesentlich schneller sind als Speicherzugriffe, deren Werte noch nicht im Cache liegen.

Cache-Speicher werden für fast alle Rechnertypen einschließlich Multicore-Systeme, SMPs und Parallelrechner mit verschiedenen Speicherorganisationen zur Verringerung der Speicherzugriffszeit eingesetzt. Dabei werden üblicherweise mehrstufige Caches genutzt, vgl. Abschn. 2.7. Bei Multiprozessorsystemen mit lokalen Caches, bei denen jeder Prozessor auf den gesamten globalen Speicher Zugriff hat, tritt das Problem der Aufrechterhaltung der *Cache-Kohärenz* (engl. *cache coherence*) auf, d.h es kann die Situation eintreten, dass verschiedene Kopien einer gemeinsamen Variablen in den lokalen Caches verschiedener Prozessoren geladen und möglicherweise mit unterschiedlichen Werten belegt sein können. Die Cache-Kohärenz würde verletzt, wenn ein Prozessor p den Wert einer Speicherzelle in seinem lokalen Cache ändern würde, ohne diesen Wert in den globalen Speicher zurückzuschreiben. Würde ein anderer Prozessor q danach diese Speicherzelle laden, so würde er fälschlicherweise den noch nicht aktualisierten Wert benutzen. Aber selbst ein Zurückschreiben des Wertes durch Prozessor p in den globalen Speicher ist nicht ausreichend, wenn Prozessor q die gleiche Speicherzelle in seinem lokalen Cache hätte. In diesem Fall muss der Wert der Speicherzelle auch im lokalen Cache von Prozessor q aktualisiert werden.

Zur korrekten Realisierung der Programmierung mit gemeinsamen Variablen muss sichergestellt sein, dass alle Prozessoren den *aktuellen* Wert einer Variable erhalten, wenn sie auf diese zugreifen. Zur Aufrechterhaltung der Cache-Kohärenz gibt es mehrere Ansätze, von denen wir in Abschn. 2.7.2 einige genauer vorstellen werden. Eine ausführliche Beschreibung ist auch in [11] und [70] zu finden. Da die Behandlung der Cache-Kohärenzfrage auf das verwendete Berechnungsmodell wesentlichen Einfluss hat, werden Multiprozessoren entsprechend weiter untergliedert.

CC-NUMA-Rechner (Cache Coherent NUMA) sind Rechner mit gemeinsamem Speicher, bei denen Cache-Kohärenz sichergestellt ist. Die Caches der Prozessoren können so nicht nur Daten des lokalen Speichers des Prozessors, sondern auch globale Daten des gemeinsamen Speichers aufnehmen. Die Verwendung des Begriffes CC-NUMA macht deutlich, dass die Bezeichnung NUMA einem Bedeutungswandel unterworfen ist und mittlerweile zur Klassifizierung von Hard- und Software in Systemen mit Caches benutzt wird. Multiprozessoren, die keine Cache-Kohärenz aufweisen (manchmal auch *NC-NUMA-Rechner* für engl. *Non-Coherent NUMA* genannt), können nur Daten der lokalen Speicher oder Variablen, die nur gelesen werden können, in den Cache laden. Eine Loslösung von der statischen Speicherallokation des gemeinsamen Speichers stellen die **COMA-Rechner** (für engl. *Cache Only Memory Access*) dar, vgl. Abb. 2.7, deren Speicher nur noch aus Cache-Speicher besteht, siehe [32] für eine ausführlichere Behandlung.

Multithreading

Die Idee des verschränkten Multithreading (engl. *interleaved multithreading*) besteht darin, die Latenz der Speicherzugriffe dadurch zu verbergen, dass jeder physikalische Prozessor eine feste Anzahl v von virtuellen Prozessoren simuliert. Für jeden zu simulierenden virtuellen Prozessor enthält ein physikalischer Prozessor einen eigenen Programmzähler und üblicherweise auch einen eigenen Registersatz. Nach jeder Ausführung eines Maschinenbefehls findet ein impliziter Kontextwechsel zum nächsten virtuellen Prozessor statt, d. h. die virtuellen Prozessoren eines physikalischen Prozessors werden von diesem pipelineartig reihum simuliert. Die Anzahl der von einem physikalischen Prozessor simulierten virtuellen Prozessoren wird so gewählt, dass die Zeit zwischen der Ausführung aufeinanderfolgender Maschinenbefehle eines virtuellen Prozessors ausreicht, evtl. benötigte Daten aus dem globalen Speicher zu laden, d. h. die Verzögerungszeit des Netzwerkes wird durch die Ausführung von Maschinenbefehlen anderer virtueller Prozessoren verdeckt. Der Pipelining-Ansatz reduziert also nicht wie der Einsatz von Caches die Menge der extern zum Prozessor zugegriffenen Daten, sondern bewirkt nur, dass ein virtueller Prozessor die von ihm aus dem Speicher angeforderten Daten erst dann zu benutzen versucht, wenn diese auch eingetroffen sind. Der Vorteil dieses Ansatzes liegt darin, dass aus der Sicht eines virtuellen Prozessors die Verzögerungszeit der Speicherzugriffe nicht sichtbar ist. Damit kann für die Programmierung ein PRAM-ähnliches Programmiermodell realisiert werden, das für den Programmierer sehr einfach zu verwenden ist, vgl. Abschn. 4.5.1. Der Nachteil liegt darin, dass für die Programmierung die vergleichsweise hohe Gesamtzahl der virtuellen Prozessoren zugrunde gelegt werden muss. Daher muss der zu implementierende Algorithmus ein ausreichend großes Potential an Parallelität besitzen, damit alle virtuellen Prozessoren sinnvoll beschäftigt werden können. Ein weiterer Nachteil besteht darin, dass die verwendeten physikalischen Prozessoren speziell für den Einsatz in den jeweiligen Parallelrechnern entworfen werden müssen, da übliche Mikroprozessoren die erforderlichen schnellen Kontextwechsel nicht zur Verfügung stellen. Beispiele für Forschungsrechner, die nach dem Pipelining-Ansatz arbeiteten, waren die Denelcor HEP (*Heterogeneous Element Processor*) mit 16 physikalischen Prozessoren [167], von denen jeder bis zu 128 Threads unterstützte, der NYU Ultracomputer [62], die SB-PRAM [1], und die Tera MTA [32, 87]. Verschränktes Multithreading wird auch bei GPUs zur Verdeckung der Latenzzeiten des Zugriffs auf den globalen Speicher genutzt; dies wird in Kap. 7 ausführlich erläutert.

Ein alternativer Ansatz zum verschränkten Multithreading ist das blockorientierte Multithreading [32]. Bei diesem Ansatz besteht das auszuführende Programm aus einer Menge von Threads, die auf den zur Verfügung stehenden Prozessoren ausgeführt werden. Der Unterschied zum verschränkten Multithreading liegt darin, dass nur dann ein Kontextwechsel zum nächsten Thread stattfindet, wenn der gerade aktive Thread einen Speicherzugriff vornimmt, der nicht über den lokalen Speicher des Prozessors befriedigt werden kann. Dieser Ansatz wurde z. B. von der MIT Alewife verwendet [2, 32].

2.5 Verbindungsnetzwerke

Eine physikalische Verbindung der einzelnen Komponenten eines parallelen Systems wird durch das **Verbindungsnetzwerk** (engl. *interconnection network*) hergestellt. Neben den Kontroll- und Datenflüssen und der Organisation des Speichers kann auch das eingesetzte Verbindungsnetzwerk zur Klassifikation paralleler Systeme verwendet werden. Intern besteht ein Verbindungsnetzwerk aus Leitungen und Schaltern, die meist in regelmäßiger Weise angeordnet sind. In Multicomputersystemen werden über das Verbindungsnetzwerk verschiedene Prozessoren bzw. Verarbeitungseinheiten miteinander verbunden. Interaktionen zwischen verschiedenen Prozessoren, die zur Koordination der gemeinsamen Bearbeitung von Aufgaben notwendig sind und die entweder dem Austausch von Teilergebnissen oder der Synchronisation von Bearbeitungsströmen dienen, werden durch das Verschieben von Nachrichten, der sogenannten **Kommunikation**, über die Leitungen des Verbindungsnetzwerkes realisiert. In Multiprozessorsystemen werden die Prozessoren durch das Verbindungsnetzwerk mit den Speichermodulen verbunden, die **Speicherzugriffe** der Prozessoren erfolgen also über das Verbindungsnetzwerk. In Zukunft ist zu erwarten, dass auch für die Zusammenschaltung der Prozessorkerne eines einzelnen Multicore-Prozessors ein prozessorinternes Verbindungsnetzwerk eingesetzt wird. Dies ist insbesondere für eine große Anzahl von Prozessorkernen zu erwarten, vgl. Abschn. 2.8 und wurde für Forschungsprozessoren wie den Intel Teraflops Research Chip [74] und den Intel SCC (Single-chip Cloud Computer) bereits erprobt.

Die Grundaufgabe eines Verbindungsnetzwerkes besteht in beiden Fällen darin, eine Nachricht, die Daten oder Speicheranforderungen enthält, von einem gegebenen Prozessor zu einem angegebenen Ziel zu transportieren. Dabei kann es sich um einen anderen Prozessor oder ein Speichermodul handeln. Die Anforderung an ein Verbindungsnetzwerk besteht darin, diese Kommunikationsaufgabe in möglichst geringer Zeit korrekt auszuführen, und zwar auch dann, wenn mehrere Nachrichten gleichzeitig übertragen werden sollen. Da die Nachrichtenübertragung bzw. der Speicherzugriff einen wesentlichen Teil der Bearbeitung einer Aufgabe auf einem parallelen System mit verteiltem oder gemeinsamem Speicher darstellt, ist das benutzte Verbindungsnetzwerk ein wesentlicher Bestandteil des Designs eines parallelen Systems und kann großen Einfluss auf dessen Leistung haben. Gestaltungskriterien eines Verbindungsnetzwerkes sind

- die **Topologie**, die die Form der Verschaltung der einzelnen Prozessoren bzw. Speichermodulen beschreibt, und
- die **Routingtechnik**, die die Nachrichtenübertragung zwischen einzelnen Prozessoren bzw. zwischen Prozessoren und Speichermodulen realisiert.

Topologie

Die **Topologie** eines Verbindungsnetzwerkes beschreibt die geometrische Struktur, mit der dessen Leitungen und Schalter angeordnet sind, um Prozessoren und Speichermodule miteinander zu verbinden. Diese Verbindungsstruktur wird oft als Graph beschrieben, in dem Schalter, Prozessoren oder Speichermodule die Knoten darstellen und die Verbindungsleitungen durch Kanten repräsentiert werden. Unterschieden wird zwischen *statischen* und *dynamischen* Verbindungsnetzwerken. **Statische Verbindungsnetzwerke** verbinden Prozessoren *direkt* durch eine zwischen den Prozessoren liegende physikalische Leitung miteinander und werden daher auch **direkte** Verbindungsnetzwerke oder **Punkt-zu-Punkt**-Verbindungsnetze genannt. Die Anzahl der Verbindungen für einen Knoten variiert zwischen einer minimalen Anzahl von einem Nachbarn in einem Stern-Netzwerk und einer maximalen Anzahl von Nachbarn in einem vollständig verbundenen Graphen, vgl. Abschn. 2.5.1 und 2.5.4. Statische Netzwerke werden im Wesentlichen für Systeme mit verteiltem Speicher eingesetzt, wobei ein Knoten jeweils aus einem Prozessor und einer zugehörigen Speichereinheit besteht. **Dynamische Verbindungsnetzwerke** verbinden Prozessoren und/oder Speichereinheiten *indirekt* über mehrere Leitungen und dazwischenliegende Schalter miteinander und werden daher auch als **indirekte** Verbindungsnetzwerke bezeichnet. Varianten sind *busbasierte* Netzwerke oder *schalterbasierte* Netzwerke (engl. *switching network*), bestehend aus Leitungen und dazwischenliegenden Schaltern (engl. *switches*). Eingesetzt werden dynamische Netzwerke sowohl für Systeme mit verteiltem Speicher als auch für Systeme mit gemeinsamem Speicher. Für letztere werden sie als Verbindung zwischen den Prozessoren und den Speichermodulen verwendet. Häufig werden auch hybride Netzwerktopologien benutzt.

Routingtechnik

Eine **Routingtechnik** beschreibt, *wie* und *entlang welchen Pfades* eine Nachricht über das Verbindungsnetzwerk von einem Sendeknoten zu einem festgelegten Zielknoten übertragen wird, wobei sich der Begriff des Pfades hier auf die Beschreibung des Verbindungsnetzwerkes als Graph bezieht. Die Routingtechnik setzt sich zusammen aus dem **Routing**, das mittels eines *Routingalgorithmus* einen Pfad vom sendenden Knoten zum empfangenden Knoten für die Nachrichtenübertragung auswählt und einer **Switching-Strategie**, die festlegt, wie eine Nachricht in Teilstücke unterteilt wird, wie einer Nachricht ein Routingpfad zugeordnet wird und wie eine Nachricht über die auf dem Routingpfad liegenden Schalter oder Prozessoren weitergeleitet wird.

Die Kombination aus Routing-Algorithmus, Switching-Strategie und Topologie bestimmt wesentlich die Geschwindigkeit der zu realisierenden Kommunikation. Die nächsten Abschn. 2.5.1 bis 2.5.4 beschreiben einige gebräuchliche direkte und indirekte Topologien für Verbindungsnetzwerke. Spezielle Routing-Algorithmen

und Varianten von Switching-Strategien stellen wir in den Abschn. 2.6.1 bzw. 2.6.2 vor. Effiziente Verfahren zur Realisierung von Kommunikationsoperationen für verschiedene Verbindungsnetzwerke enthält Kap. 4. Verbindungsnetzwerke und ihre Eigenschaften werden u. a. in [14, 32, 87, 64, 111, 164, 42] detailliert behandelt.

2.5.1 Bewertungskriterien für Netzwerke

In statischen Verbindungsnetzwerken sind die Verbindungen zwischen Schaltern oder Prozessoren fest angelegt. Ein solches Netzwerk kann durch einen Kommunikationsgraphen $G = (V, E)$ beschrieben werden, wobei V die Knotenmenge der zu verbindenden Prozessoren und E die Menge der direkten Verbindungen zwischen den Prozessoren bezeichnet, d. h. es ist $(u, v) \in E$, wenn es eine direkte Verbindung zwischen den beiden Prozessoren $u \in V$ und $v \in V$ gibt. Da für die meisten parallelen Systeme das Verbindungsnetzwerk bidirektional ausgelegt ist, also in beide Richtungen der Verbindungsleitung eine Nachricht geschickt werden kann, wird G meist als ungerichteter Graph definiert. Soll eine Nachricht von einem Knoten u zu einem anderen Knoten v gesendet werden, zu dem es *keine* direkte Verbindungsleitung gibt, so muss ein Pfad von u nach v gewählt werden, der aus mehreren Verbindungsleitungen besteht, über die die Nachricht dann geschickt wird. Eine Folge von Knoten (v_0, \dots, v_k) heißt *Pfad* der Länge k zwischen den Knoten v_0 und v_k , wenn $(v_i, v_{i+1}) \in E$ für $0 \leq i < k$. Als Verbindungsnetzwerke sind nur solche Netzwerke sinnvoll, für die es zwischen beliebigen Prozessoren $u, v \in V$ mindestens einen Pfad gibt.

Statische Verbindungsnetzwerke können anhand verschiedener Eigenschaften des zugrunde liegenden Graphen G bewertet werden. Neben der Anzahl der Knoten n werden folgende Eigenschaften betrachtet: Durchmesser, Grad, Bisektionsbandbreite, Knoten- und Kantenkonnektivität und Einbettung in andere Netzwerke. Wir gehen im Folgenden auf diese Eigenschaften näher ein.

Als **Durchmesser** (engl. *diameter*) $\delta(G)$ eines Netzwerkes G wird die maximale Distanz zwischen zwei beliebigen Prozessoren bezeichnet:

$$\delta(G) = \max_{u, v \in V} \min_{\substack{\varphi \text{ Pfad} \\ \text{von } u \text{ nach } v}} \{k \mid k \text{ ist Länge des Pfades } \varphi \text{ von } u \text{ nach } v\}.$$

Der Durchmesser ist ein Maß dafür, wie lange es dauern kann, bis eine von einem beliebigen Prozessor abgeschickte Nachricht bei einem beliebigen anderen Prozessor ankommt.

Der **Grad** (engl. *degree*) $g(G)$ eines Netzwerkes G ist der maximale Grad eines Knotens des Netzwerkes, wobei der Grad eines Knotens der Anzahl der adjazenten, d. h. ein- bzw. auslaufenden, Kanten des Knotens entspricht:

$$g(G) = \max\{g(v) \mid g(v) \text{ Grad von } v \in V\}.$$

Die **Bisektionsbreite** bzw. **Bisektionsbandbreite** (engl. *bisection bandwidth*) eines Netzwerkes G ist die minimale Anzahl von Kanten, die aus dem Netzwerk entfernt werden müssen, um das Netzwerk in zwei gleichgroße Teilnetzwerke zu zerlegen, d. h. in zwei Teilnetzwerke mit einer bis auf 1 gleichen Anzahl von Knoten. Die Bisektionsbandbreite $B(G)$ ist also definiert als

$$B(G) = \min_{\substack{U_1, U_2 \text{ Partition von } V \\ ||U_1| - |U_2|| \leq 1}} |\{(u, v) \in E \mid u \in U_1, v \in U_2\}|.$$

Bereits $B(G) + 1$ Nachrichten können das Netzwerk sättigen, falls diese zur gleichen Zeit über die entsprechenden Kanten übertragen werden sollen. Damit ist die Bisektionsbandbreite ein Maß für die Belastbarkeit des Netzwerkes bei der gleichzeitigen Übertragung von Nachrichten.

Die **Knotenkonnektivität** (engl. *node connectivity*) und **Kantenkonnektivität** (engl. *edge connectivity*) sind verschiedene Beschreibungen des Zusammenhangs der Knoten des Netzwerkes. Der Zusammenhang hat Auswirkungen auf die Ausfallsicherheit des Netzwerkes. Die Knotenkonnektivität eines Netzwerkes G ist die minimale Anzahl von Knoten, die gelöscht werden müssen, um das Netzwerk zu unterbrechen, d. h. in zwei unverbundene Netzwerke (nicht unbedingt gleicher Größe) zu zerlegen. Für eine genauere Definition bezeichnen wir mit $G_{V \setminus M}$ den Restgraphen, der durch Löschen der Knoten von $M \subset V$ und aller zugehörigen Kanten entsteht. Es ist also $G_{V \setminus M} = (V \setminus M, E \cap ((V \setminus M) \times (V \setminus M)))$. Die Knotenkonnektivität $nc(G)$ von G ist damit definiert als

$$nc(G) = \min_{M \subset V} \{|M| \mid \text{es existieren } u, v \in V \setminus M, \text{ so dass es in } G_{V \setminus M} \text{ keinen Pfad von } u \text{ nach } v \text{ gibt}\}.$$

Analog bezeichnet die Kantenkonnektivität eines Netzwerkes G die minimale Anzahl von Kanten, die man löschen muss, damit das Netzwerk unterbrochen wird. Für eine beliebige Teilmenge $F \subset E$ bezeichne $G_{E \setminus F}$ den Restgraphen, der durch Löschen der Kanten von F entsteht, d. h. $G_{E \setminus F} = (V, E \setminus F)$. Die Kantenkonnektivität $ec(G)$ von G ist definiert durch

$$ec(G) = \min_{F \subset E} \{|F| \mid \text{es existieren } u, v \in V, \text{ so dass es in } G_{E \setminus F} \text{ keinen Pfad von } u \text{ nach } v \text{ gibt}\}.$$

Die Knoten- oder Kantenkonnektivität eines Verbindungsnetzwerkes ist ein Maß für die Anzahl der unabhängigen Wege, die zwei beliebige Prozessoren u und v miteinander verbinden. Eine hohe Konnektivität sichert eine hohe Zuverlässigkeit bzw. Ausfallsicherheit des Netzwerkes, da viele Prozessoren bzw. Verbindungen ausfallen müssen, bevor das Netzwerk zerfällt. Eine Obergrenze für die Knoten- oder Kantenkonnektivität eines Netzwerkes bildet der kleinste Grad eines Knotens im

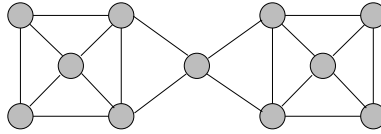


Abb. 2.8 Netzwerk mit Knotenkonnektivität 1, Kantenkonnektivität 2 und Grad 4. Der kleinste Grad eines Knotens ist 3

Netzwerk, da ein Knoten dadurch vollständig von seinen Nachbarn separiert werden kann, dass alle seine Nachbarn bzw. alle Kanten zu diesen Nachbarn gelöscht werden. Man beachte, dass die Knotenkonnektivität eines Netzwerkes kleiner als seine Kantenkonnektivität sein kann, vgl. Abb. 2.8.

Ein Maß für die Flexibilität eines Netzwerkes wird durch den Begriff der Einbettung (engl. *embedding*) bereitgestellt. Seien $G = (V, E)$ und $G' = (V', E')$ zwei Netzwerke. Eine **Einbettung** von G' in G ordnet jeden Knoten von G' einem Knoten von G so zu, dass unterschiedliche Knoten von G' auf unterschiedliche Knoten von G abgebildet werden und dass Kanten zwischen zwei Knoten in G' auch zwischen den zugeordneten Knoten in G existieren [14]. Eine Einbettung von G' in G wird beschrieben durch eine Funktion $\sigma : V' \rightarrow V$, für die gilt:

- Wenn $u \neq v$ für $u, v \in V'$ gilt, dann folgt $\sigma(u) \neq \sigma(v)$.
- Wenn $(u, v) \in E'$ gilt, dann folgt $(\sigma(u), \sigma(v)) \in E$.

Kann ein Netzwerk G' in ein Netzwerk G eingebettet werden, so besagt dies, dass G mindestens so flexibel ist wie Netzwerk G' , da ein Algorithmus, der die Nachbarschaftsbeziehungen in G' ausnutzt, durch eine Umnummerierung gemäß σ in einen Algorithmus auf G abgebildet werden kann, der auch in G Nachbarschaftsbeziehungen ausnutzt.

Anforderungen an ein Netzwerk

Das Netzwerk eines parallelen Systems sollte entsprechend den Anforderungen an dessen Architektur ausgewählt werden. Allgemeine Anforderungen an ein Netzwerk im Sinne der eingeführten Eigenschaften der Topologie sind:

- Ein kleiner Durchmesser für kleine Distanzen bei der Nachrichtenübertragung
- Ein kleiner Grad jedes Knotens zur Reduzierung des Hardwareaufwandes
- Eine hohe Bisektionsbandbreite zur Erreichung eines hohen Durchsatzes
- Eine hohe Konnektivität zur Erreichung hoher Zuverlässigkeit
- Die Möglichkeit der Einbettung von möglichst vielen anderen Netzwerken sowie
- Eine einfache Erweiterbarkeit auf eine größere Anzahl von Prozessoren (Skalierbarkeit).

Da sich diese Anforderungen z. T. widersprechen, gibt es kein Netzwerk, das alle Anforderungen im gleichen Umfang erfüllt. Im Folgenden werden wir einige häu-

fig verwendete direkte Verbindungsnetzwerke vorstellen. Die Topologien sind in Abb. 2.9 illustriert, die wichtigsten Eigenschaften sind in Tabelle 2.1 zusammengefasst.

2.5.2 Direkte Verbindungsnetzwerke

Die üblicherweise verwendeten direkten Verbindungsnetzwerke haben eine regelmäßige Struktur und sind daher durch einen regelmäßigen Aufbau des zugrunde liegenden Graphen $G = (V, E)$ gekennzeichnet. Bei der Beschreibung der Topologien wird die Anzahl der Knoten bzw. Prozessoren $n = |V|$ als Parameter benutzt, so dass die jeweilige Topologie kein einzelnes Netzwerk, sondern eine ganze Klasse von ähnlich aufgebauten Netzwerken beschreibt.

- (a) Ein **vollständiger Graph** ist ein Netzwerk G , in dem jeder Knoten direkt mit jedem anderen Knoten verbunden ist, vgl. Abb. 2.9a. Dies ergibt einen Durchmesser $\delta(G) = 1$. Entsprechend gilt für den Grad $g(G) = n - 1$ und für die Knoten- bzw. Kantenkonnektivität $nc(G) = ec(G) = n - 1$, da die Verbindung zu einem Knoten durch Entfernen der $n - 1$ adjazenten Kanten unterbrochen werden kann. Die Bisektionsbandbreite ist $n^2/4$ für gerades n : Teilt man einen vollständigen Graphen mit n Knoten in zwei gleichgroße Teilgraphen mit jeweils $n/2$ Knoten auf, so gibt es von jedem der $n/2$ Knoten des ersten Teilgraphen eine Kante zu jedem der $n/2$ Knoten des zweiten Teilgraphen. Also ergeben sich insgesamt $n/2 \cdot n/2$ viele Kanten. Eine Einbettung in einen vollständigen Graphen ist für alle anderen Netzwerke möglich, da ein vollständiger Graph für eine gegebene Anzahl von Knoten die maximal mögliche Anzahl von Kanten enthält. Die physikalische Realisierbarkeit ist wegen des hohen Knotengrades jedoch nur für eine kleine Anzahl n von Prozessoren gegeben.
- (b) In einem **linearen Feld** können die Knoten linear angeordnet werden, so dass zwischen benachbarten Prozessoren eine bidirektionale Verbindung besteht, vgl. Abb. 2.9b, d.h. es ist $V = \{v_1, \dots, v_n\}$ und $E = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. Bedingt durch den Abstand von Knoten v_1 zu Knoten v_n ergibt sich als Durchmesser $\delta(G) = n - 1$. Die Konnektivität ist $nc(G) = ec(G) = 1$, da bereits durch den Ausfall eines einzelnen Knotens oder einer einzelnen Kante das Netzwerk unterbrochen wird. Der Grad ist $g(G) = 2$ und die Bisektionsbandbreite ist $B(G) = 1$. Eine Einbettung ist in fast alle hier aufgeführten Netzwerke mit Ausnahme des Baumes (siehe (h) dieser Auflistung und Abb. 2.9h) möglich. Da es nur genau eine Verbindung zwischen zwei Knoten gibt, ist keine Fehlertoleranz bzgl. der Übermittlung von Nachrichten gegeben.
- (c) In einem **Ring-Netzwerk** können die Knoten in Form eines Ringes angeordnet werden, d.h. zusätzlich zu den Kanten des linearen Feldes existiert eine bidirektionale Kante vom ersten Knoten der linearen Anordnung zum letzten Knoten, vgl. Abb. 2.9c. Bei bidirektionalen Verbindungen ist der Durchmesser

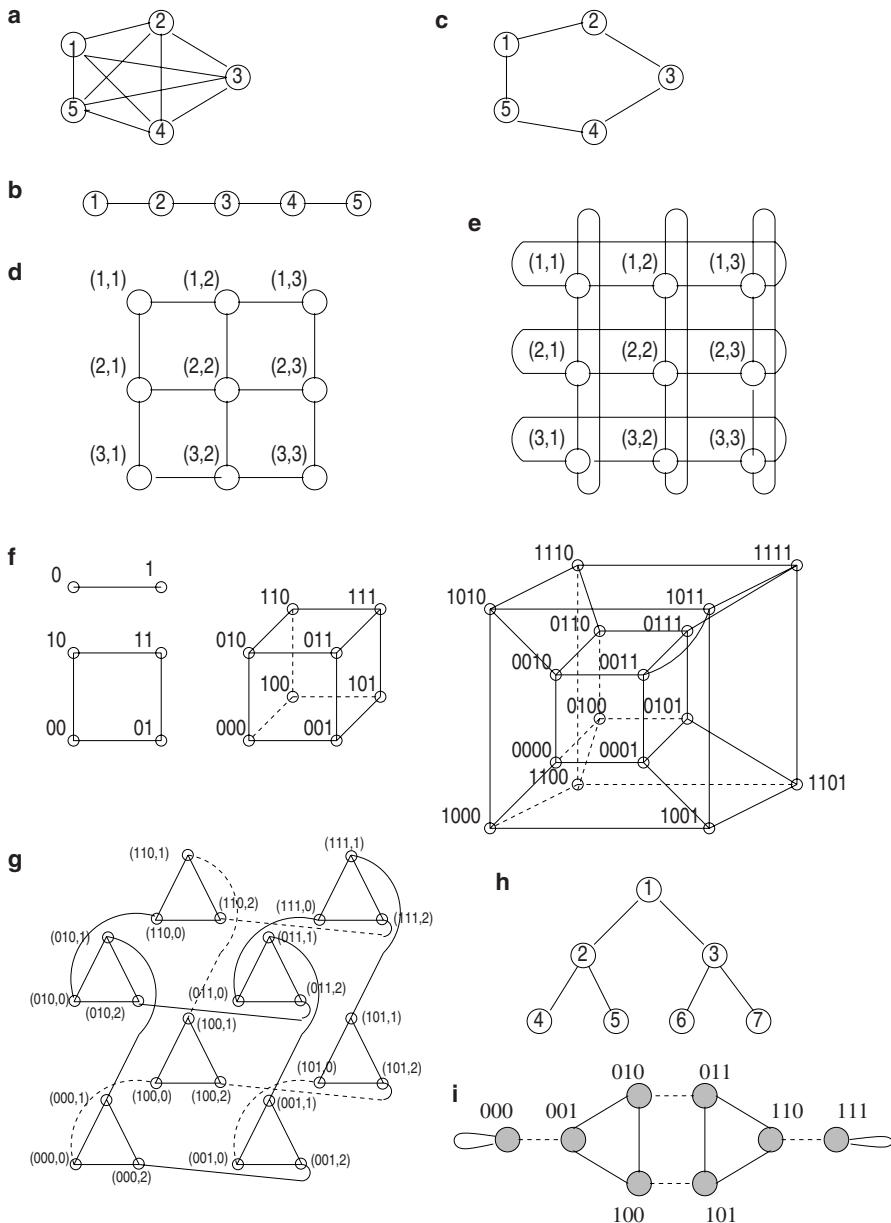


Abb. 2.9 Spezielle statische Verbindungsnetzwerke: **a** Vollständiger Graph, **b** Lineares Feld, **c** Ring, **d** 2-dimensionales Gitter, **e** 2-dimensionaler Torus, **f** k -dimensionaler Würfel für $k = 1, 2, 3, 4$, **g** Cube-connected-cycles Netzwerk für $k = 3$, **h** vollständiger binärer Baum, **i** Shuffle-Exchange-Netzwerk mit 8 Knoten, wobei die gestrichelten Kanten Austauschanten und die durchgezogenen Kanten Mischanten darstellen

Tab. 2.1 Zusammenfassung der Parameter statischer Verbindungsnetzwerke für ausgewählte Topologien

Netzwerk G mit n Knoten	Grad $g(G)$	Durchmesser $\delta(G)$	Kanten-konnektivität $ec(G)$	Bisektionsbandbreite $B(G)$
Vollständiger Graph	$n - 1$	1	$n - 1$	$(\frac{n}{2})^2$
Lineares Feld	2	$n - 1$	1	1
Ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d -dimensionales Gitter ($n = r^d$)	$2d$	$d(\sqrt[d]{n} - 1)$	d	$n^{\frac{d-1}{d}}$
d -dimensionaler Torus ($n = r^d$)	$2d$	$d \lfloor \frac{\sqrt[d]{n}}{2} \rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k -dimensionaler Hyperwürfel ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k -dimensionales CCC-Netzwerk ($n = k2^k$ für $k \geq 3$)	3	$2k - 1 + \lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
Vollständiger binärer Baum ($n = 2^k - 1$)	3	$2 \log \frac{n+1}{2}$	1	1
k -facher d -Würfel ($n = k^d$)	$2d$	$d \lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

$\delta(G) = \lfloor n/2 \rfloor$, die Konnektivität $nc(G) = ec(G) = 2$, der Grad $g(G) = 2$ und die Bisektionsbandbreite $B(G) = 2$. In der Praxis ist der Ring für kleine Prozessoranzahlen und als Bestandteil komplexerer Netzwerke einsetzbar.

- (d) Ein **d -dimensionales Gitter**, auch **d -dimensionales Feld** genannt, ($d \geq 1$) besteht aus $n = n_1 \cdot n_2 \cdot \dots \cdot n_d$ Knoten, die in Form eines d -dimensionalen Gitters angeordnet werden können, vgl. Abb. 2.9d. Dabei bezeichnet n_j für $j = 1, \dots, d$ die Ausdehnung des Gitters in Dimension j . Jeder Knoten in diesem d -dimensionalen Gitter kann durch seine Position (x_1, \dots, x_d) mit $1 \leq x_j \leq n_j$ für $j = 1, \dots, d$ beschrieben werden. Zwischen Knoten (x_1, \dots, x_d) und Knoten (x'_1, \dots, x'_d) gibt es genau dann eine Kante, wenn es ein $\mu \in \{1, \dots, d\}$ gibt mit

$$|x_\mu - x'_\mu| = 1 \text{ und } x_j = x'_j \text{ für alle } j \neq \mu.$$

Ist $n_j = r = \sqrt[d]{n}$ für alle $j = 1, \dots, d$ (also $n = r^d$), so ist der Durchmesser $\delta(G) = d \cdot (\sqrt[d]{n} - 1)$, da auf einem Pfad zwischen entgegengesetzten Eckpunkten in jeder Dimension $\sqrt[d]{n} - 1$ viele Kanten durchlaufen werden müssen. Die Knoten- und Kantenkonnektivität ist $nc(G) = ec(G) = d$, da z. B. die Eckknoten durch Löschen der d Nachbarknoten oder der d einlaufenden Kanten vom Rest des Netzwerkes abgetrennt werden können. Der Grad ist $g(G) = 2d$. Ein 2-dimensionales Gitter wurde z. B. für den Terascale-Chip von Intel vorgeschlagen, vgl. Abschn. 2.8.

- (e) Ein **d -dimensionaler Torus** ist eine Variante des d -dimensionalen Gitters, die zusätzlich zu den Knoten und Kanten des Gitters für jede Dimension $j = 1, \dots, d$ zusätzliche Kanten zwischen den Knoten $(x_1, \dots, x_{j-1}, 1, x_{j+1}, \dots, x_d)$ und $(x_1, \dots, x_{j-1}, n_j, x_{j+1}, \dots, x_d)$ enthält, vgl. Abb. 2.9e. Für den Spezialfall $n_j = \sqrt[d]{n}$ für alle $j = 1, \dots, d$, reduziert sich der Durchmesser gegenüber dem Gitter dadurch auf $\delta(G) = d \cdot \lfloor \sqrt[d]{n}/2 \rfloor$. Der Grad ist für alle Knoten $g(G) = 2d$ und die Konnektivität ist ebenfalls $nc(G) = ec(G) = 2d$. Ein 3-dimensionaler Torus wird als Topologie für die Cray XT3, XT4 und XT5 sowie für die Übertragung von Punkt-zu-Punkt-Nachrichten in den IBM BlueGene/L- und BlueGene/P-Systemen verwendet. Für die BlueGene/Q-Systeme wird ein 5-dimensionaler Torus verwendet.
- (f) Ein **k -dimensionaler Würfel** oder **Hyperwürfel** hat $n = 2^k$ Knoten, zwischen denen Kanten entsprechend eines rekursiven Aufbaus aus niedrigerdimensionalen Würfeln existieren, vgl. Abb. 2.9(f). Jedem Knoten wird ein binäres Wort der Länge k als Namen zugeordnet, wobei die Gesamtheit dieser k -Bitworte den Dezimalzahlen $0, \dots, 2^k - 1$ entsprechen. Ein 1-dimensionaler Würfel besteht aus zwei Knoten mit den 1-Bitnamen 0 bzw. 1 und einer Kante, die diese beiden Knoten verbindet. Ein k -dimensionaler Würfel wird aus zwei $(k - 1)$ -dimensionalen Würfeln (mit jeweiliger Knotennummerierung $0, \dots, 2^{k-1} - 1$) konstruiert. Dazu werden alle Knoten und Kanten der beiden $(k - 1)$ -dimensionalen Würfel übernommen und zusätzliche Kanten zwischen zwei Knoten gleicher Nummer gezogen. Die Knotennummerierung wird neu festgelegt, indem die Knoten des ersten $(k - 1)$ -dimensionalen Würfels eine zusätzliche 0 und die Knoten des zweiten $(k - 1)$ -dimensionalen Würfels eine zusätzliche 1 vor ihre Nummer erhalten. Werden die Knoten des k -dimensionalen Würfels mit ihrer Nummerierung identifiziert, also $V = \{0, 1\}^k$, so existiert entsprechend der Konstruktion eine Kante zwischen Knoten $\alpha_0 \dots \alpha_j \dots \alpha_{k-1}$ und Knoten $\alpha_0 \dots \bar{\alpha}_j \dots \alpha_{k-1}$ für $0 \leq j \leq k - 1$, wobei $\bar{\alpha}_j = 1$ für $\alpha_j = 0$ und $\bar{\alpha}_j = 0$ für $\alpha_j = 1$ gilt. Es gibt also Kanten zwischen solchen Knoten, die sich genau in einem Bit unterscheiden. Dieser Zusammenhang wird oft mit Hilfe der Hamming-Distanz beschrieben.

► **Hamming-Distanz** Die Hamming-Distanz zweier gleich langer binärer Worte ist als die Anzahl der Bits definiert, in denen sich die Worte unterscheiden.

Zwei Knoten des k -dimensionalen Würfels sind also direkt miteinander verbunden, falls ihre Hamming-Distanz 1 ist. Zwischen zwei Knoten $v, w \in V$ mit Hamming-Distanz d , $1 \leq d \leq k$, existiert ein Pfad der Länge d , der v und w verbindet. Dieser Pfad kann bestimmt werden, indem die Bitdarstellung von v von links nach rechts durchlaufen wird und nacheinander die Bits invertiert werden, in denen sich v und w unterscheiden. Jede Bitumkehrung entspricht dabei dem Übergang zu einem Nachbarknoten.

Der Durchmesser eines k -dimensionalen Würfels ist $\delta(G) = k$, da sich die Bitdarstellungen der Knoten in höchstens k Positionen unterscheiden können

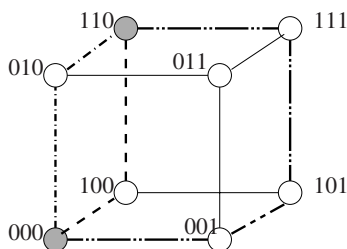


Abb. 2.10 In einem 3-dimensionalen Hyperwürfel gibt es drei unabhängige Pfade von Knoten 000 zu Knoten 110. Die Hamming-Distanz zwischen Knoten 000 und Knoten 110 ist $l = 2$. Es existieren zwei Pfade zwischen Knoten 000 und Knoten 110 der Länge $l = 2$, nämlich die Pfade $(000, 100, 110)$ und $(000, 010, 110)$, und $k - l = 1$ Pfade der Länge $l + 2 = 4$, nämlich $(000, 001, 101, 111, 110)$

und es daher zwischen beliebigen Knoten einen Pfad der Länge $\leq k$ gibt. Der Grad ist $g(G) = k$, da es in Bitworten der Länge k genau k mögliche einzelne Bitumkehrungen, also direkte Nachbarn, gibt. Für die Knoten- und Kantenkonnektivität gilt ebenfalls $nc(G) = ec(G) = k$, wie aus folgender Betrachtung ersichtlich ist.

Die Konnektivität ist höchstens k , d.h. $nc(G) \leq k$, da durch das Löschen der k Nachbarknoten bzw. -kanten ein Knoten vollständig vom Gesamtgraphen abgetrennt werden kann. Um zu zeigen, dass die Konnektivität nicht kleiner als k sein kann, wird nachgewiesen, dass es zwischen zwei beliebigen Knoten v und w genau k unabhängige Pfade gibt, d.h. Pfade, die keine gemeinsamen Kanten und nur gleiche Anfangs- und Endknoten haben. Seien nun A und B die Bitnummerierungen der Knoten v und w , die sich in l Bits, $1 \leq l \leq k$, unterscheiden und seien dies (nach evtl. Umnummerierung) die ersten l Bits. Man kann l Pfade der Länge l zwischen Knoten v und w durch Invertieren der ersten l Bits von v konstruieren. Für Pfad i , $i \in \{0, \dots, l-1\}$, werden nacheinander zunächst die Bits $i, \dots, l-1$ und anschließend die Bits $0, \dots, i-1$ invertiert. Weitere $k-l$ Pfade zwischen Knoten v und w , jeweils der Länge $l+2$, werden konstruiert, indem für $0 \leq i < k-l$ zunächst das $(l+i)$ -te Bit von v invertiert wird, dann nacheinander die Bits der Positionen $0, \dots, l-1$ invertiert werden und abschließend das $(l+i)$ -te Bit wieder zurückinvertiert wird. Abbildung 2.10 zeigt ein Beispiel. Alle k konstruierten Pfade sind unabhängig voneinander und es folgt, dass $nc(G) \geq k$ gilt. Wegen $nc(G) \leq k$ und $nc(G) \geq k$ ergibt sich $nc(G) = k$. Analog ergibt sich $ec(G) = k$.

In einen k -dimensionalen Würfel können sehr viele Netzwerke eingebettet werden, worauf wir später noch eingehen werden.

- (g) Ein CCC-Netzwerk (**Cube Connected Cycles**) entsteht aus einem k -dimensionalen Würfel, indem jeder Knoten durch einen Zyklus (Ring) aus k Knoten ersetzt wird. Jeder dieser Knoten im Zyklus übernimmt eine Verbindung zu einem Nachbarn des ehemaligen Knotens, vgl. Abb. 2.9g. Die Knotenmenge des

CCC-Netzwerkes wird gegenüber dem k -dimensionalen Würfel erweitert auf $V = \{0, 1\}^k \times \{0, \dots, k-1\}$, wobei $\{0, 1\}^k$ die Knotenbezeichnung des k -dimensionalen Würfels ist und $i \in \{0, \dots, k-1\}$ die Position im Zyklus angibt. Die Kantenmenge besteht aus einer Menge F von Zykluskanten und einer Menge E von Hyperwürfelkanten, d. h.

$$F = \{((\alpha, i), (\alpha, (i+1) \bmod k)) \mid \alpha \in \{0, 1\}^k, 0 \leq i < k\},$$

$$E = \{((\alpha, i), (\beta, i)) \mid \alpha_i \neq \beta_i \text{ und } \alpha_j = \beta_j \text{ für } j \neq i\}.$$

Jeder der insgesamt $k \cdot 2^k$ Knoten hat den Grad $g(G) = 3$, wodurch der Nachteil des evtl. großen Grades beim k -dimensionalen Würfel beseitigt wird. Die Konnektivität ist $nc(G) = ec(G) = 3$, denn ein Knoten kann durch Löschen von 3 Kanten bzw. 3 Knoten vom Restgraphen abgehängt werden. Eine obere Schranke für den Durchmesser ist $\delta(G) = 2k - 1 + \lfloor \frac{k}{2} \rfloor$.

Zur Konstruktion eines Pfades mit dieser Durchmesserlänge betrachten wir zwei Knoten in Zyklen mit maximalem Hyperwürfelabstand k , d. h. Knoten (α, i) und (β, j) , bei denen sich die k -Bitworte α und β in jedem Bit unterscheiden. Wir wählen einen Pfad von (α, i) nach (β, j) , indem wir nacheinander jeweils eine Hyperwürfelverbindung und eine Zyklusverbindung durchwandern. Der Pfad startet mit $(\alpha_0 \dots \alpha_i \dots \alpha_{k-1}, i)$ und erreicht den nächsten Knoten durch Invertierung von α_i zu $\bar{\alpha}_i = \beta_i$. Von $(\alpha_0 \dots \beta_i \dots \alpha_{k-1}, i)$ gelangen wir über eine Zykluskante zum nächsten Knoten; dieser ist $(\alpha_0 \dots \beta_i \dots \alpha_{k-1}, (i+1) \bmod k)$. In den nächsten Schritten werden ausgehend vom bereits erreichten Knoten nacheinander die Bits $\alpha_{i+1}, \dots, \alpha_{k-1}$, und dann $\alpha_0, \dots, \alpha_{i-1}$ invertiert. Dazwischen laufen wir jeweils im Zyklus um einen Position weiter. Dies ergibt $2k - 1$ Schritte. In maximal $\lfloor \frac{k}{2} \rfloor$ weiteren Schritten gelangt man von $(\beta, i+k-1 \bmod k)$ durch Verfolgen von Zykluskanten zum Ziel (β, j) .

- (h) Das Netzwerk eines **vollständigen, binären Baumes** mit $n = 2^k - 1$ Knoten ist ein binärer Baum, in dem alle Blattknoten die gleiche Tiefe haben und der Grad aller inneren Knoten $g(G) = 3$ ist. Der Durchmesser ist $\delta(G) = 2 \log \frac{n+1}{2}$ und wird durch den Pfad zwischen zwei Blättern in verschiedenen Unterbäumen des Wurzelknotens bestimmt, der sich aus dem Pfad des einen Blattes zur Wurzel des Baumes und dem Pfad von der Wurzel zum zweiten Blatt zusammensetzt. Die Konnektivität ist $nc(G) = ec(G) = 1$, da durch Wegnahme der Wurzel bzw. einer der von der Wurzel ausgehenden Kanten der Baum zerfällt.
- (i) Ein k -dimensionales **Shuffle-Exchange-Netzwerk** besitzt $n = 2^k$ Knoten und $3 \cdot 2^{k-1}$ Kanten [174]. Werden die Knoten mit den k -Bitworten für $0, \dots, n-1$ identifiziert, so ist ein Knoten α mit einem Knoten β genau dann verbunden, falls gilt:
 - a) α und β unterscheiden sich nur im letzten (rechtsten) Bit (Austauschkante, *exchange edge*) oder
 - b) α entsteht aus β durch einen zyklischen Linksshift oder einen zyklischen Rechtsshift von β (Mischkante, *shuffle edge*).

Damit ergeben sich 2^{k-1} Austauschanten, da es so viele k -Bitworte gibt, die sich nur in der letzten Bitposition unterscheiden. Die Anzahl der Mischanten ist 2^k : für jeden Knoten α der 2^k Knoten gibt es durch Ausführung eines zyklischen Linksshift einen Kantenpartner β ; der durch zyklische Rechtsshift aus β entstehende Kantenpartner ist dann wieder α . Damit ergeben sich insgesamt $2 \cdot 2^{k-1}$ unidirektionale Mischanten. Insgesamt resultieren $3 \cdot 2^{k-1}$ Austausch- oder Mischanten. Abbildung 2.9i zeigt ein Shuffle-Exchange-Netzwerk mit 8 Knoten. Die Permutation (α, β) , wobei β aus α durch zyklischen Linksshift entsteht, heißt auch **perfect shuffle**. Die Permutation (α, β) , wobei β aus α durch zyklischen Rechtsshift entsteht, heißt auch **inverse perfect shuffle**. Viele Eigenschaften von Shuffle-Exchange-Netzwerken sind in [111] beschrieben.

Ein **k -facher d -Würfel** (engl. *k-ary d-cube*) mit $k \geq 2$ ist eine Verallgemeinerung des d -dimensionalen Gitters mit $n = k^d$ Knoten, wobei jeweils k Knoten in einer Dimension i liegen, $i = 0, \dots, d-1$. Jeder Knoten im k -fachen d -Würfel erhält eine Bezeichnung aus n Ziffern (a_0, \dots, a_{d-1}) mit $0 \leq a_i \leq k-1$. Die i -te Ziffer a_i repräsentiert die Position des Knotens in Dimension i , $i = 0, \dots, d-1$. Zwei Knoten A und B mit Bezeichnung (a_0, \dots, a_{d-1}) bzw. (b_0, \dots, b_{d-1}) sind genau dann durch eine Kante verbunden, wenn für ein $j \in \{0, \dots, d-1\}$ gilt: $a_j = (b_j \pm 1) \bmod k$ und $a_i = b_i$ für alle $i = 0, \dots, d-1$, $i \neq j$. Bedingt durch einen bzw. zwei Nachbarn in jeder Dimension hat ein Knoten für $k = 2$ den Grad $g(G) = d$ und für $k > 2$ den Grad $g(G) = 2d$. Der k -fache d -Würfel umfasst einige der oben genannten speziellen Topologien als Spezialfälle. So entspricht ein k -facher 1-Würfel einem Ring mit k Knoten, ein k -facher 2-Würfel einem Torus mit k^2 Knoten, ein 3-facher 3-Würfel einem 3-dimensionalen Torus mit $3 \times 3 \times 3$ Knoten und ein 2-facher d -Würfel einem d -dimensionalen Hyperwürfel.

2.5.3 Einbettungen

Zur Illustration des Begriffes der Einbettung von Netzwerken betrachten wir im Folgenden die Einbettung eines Ringes und eines zweidimensionalen Gitters in einen k -dimensionalen Würfel.

Einbettung eines Rings in einen k -dimensionalen Würfel

Zur Konstruktion einer Einbettung eines Rings mit $n = 2^k$ Knoten in einen k -dimensionalen Würfel wird die Knotenmenge $V' = \{1, \dots, n\}$ des Rings durch eine bijektive Funktion so auf die Knotenmenge $V = \{0, 1\}^k$ abgebildet, dass die Kanten $(i, j) \in E'$ des Rings auf Kanten in E des Würfels abgebildet werden. Da die Knoten des Ringes mit $1, \dots, n$ durchnummeriert werden können, kann eine Einbettung dadurch konstruiert werden, dass eine entsprechende Aufzäh-

lung der Knoten im Würfel konstruiert wird, so dass zwischen aufeinanderfolgend aufgezählten Knoten eine Kante im Würfel existiert. Die Einbettungskonstruktion verwendet (gespiegelte) **Gray-Code-Folgen** (engl. *reflected Gray code* – RGC).

► **Gespiegelter Gray-Code – RGC** Ein k -Bit Gray-Code ist ein 2^k -Tupel aus k -Bitzahlen, wobei sich aufeinanderfolgende Zahlen im Tupel in genau einer Bitposition unterscheiden. Der gespiegelte k -Bit Gray-Code wird folgendermaßen rekursiv definiert:

- Der 1-Bit Gray-Code ist $RGC_1 = (0, 1)$.
- Der 2-Bit Gray-Code wird aus RGC_1 aufgebaut, indem einmal 0 und einmal 1 vor die Elemente von RGC_1 gesetzt wird und die beiden resultierenden Folgen $(00, 01)$ und $(10, 11)$ nach Umkehrung der zweiten Folge konkateniert werden. Damit ergibt sich $RGC_2 = (00, 01, 11, 10)$.
- Der k -Bit Gray-Code RGC_k für $k \geq 2$ wird aus dem $(k - 1)$ -Bit Gray-Code $RGC_{k-1} = (b_1, \dots, b_m)$ mit $m = 2^{k-1}$ konstruiert, dessen Einträge b_i für $1 \leq i \leq m$ binäre Worte der Länge $k - 1$ sind. Zur Konstruktion von RGC_k wird RGC_{k-1} dupliziert, vor jedes binäre Wort des Originals wird eine 0 und vor jedes binäre Wort des Duplikats wird eine 1 gesetzt. Die resultierenden Folgen sind $(0b_1, \dots, 0b_m)$ und $(1b_1, \dots, 1b_m)$. RGC_k resultiert durch Umkehrung der zweiten Folge und Konkatenation, also $RGC_k = (0b_1, \dots, 0b_m, 1b_m, \dots, 1b_1)$.

Die so konstruierten Gray-Codes RGC_k haben für beliebige k die Eigenschaft, dass sie alle Knotennummierungen eines k -dimensionalen Würfels enthalten, da die Konstruktion der oben beschriebenen Konstruktion eines k -dimensionalen Hyperwürfels aus zwei $(k - 1)$ -dimensionalen Hyperwürfeln entspricht. Weiter unterscheiden sich benachbarte Elemente von RGC_k in genau einem Bit. Dies lässt sich durch Induktion beweisen: Die Aussage gilt für Nachbarn aus den ersten bzw. letzten 2^{k-1} Elementen von RGC_k nach Induktionsannahme, da im Vergleich zu RGC_{k-1} nur eine 0 oder 1 vorangestellt wurde. Die Aussage gilt auch für die mittleren Nachbarelemente $0b_m$ und $1b_m$. Analog unterscheiden sich das erste und das letzte Element von RGC_k nach Konstruktion im vordersten Bit. Damit sind in RGC_k benachbarte Knoten durch eine Kante im Würfel miteinander verbunden. Die Einbettung eines Rings in einen k -dimensionalen Würfel wird also durch die Abbildung

$$\sigma : \{1, \dots, n\} \rightarrow \{0, 1\}^k \quad \text{mit } \sigma(i) := RGC_k(i)$$

definiert, wobei $RGC_k(i)$ das i -te Element der Folge RGC_k bezeichnet. Abbildung 2.11a zeigt ein Beispiel.

Einbettung eines 2-dimensionalen Gitters in einen k -dimensionalen Würfel

Die Einbettung eines zweidimensionalen Feldes mit $n = n_1 \cdot n_2$ Knoten in einen k -dimensionalen Würfel mit $n = 2^k$ Knoten stellt eine Verallgemeinerung der Ein-

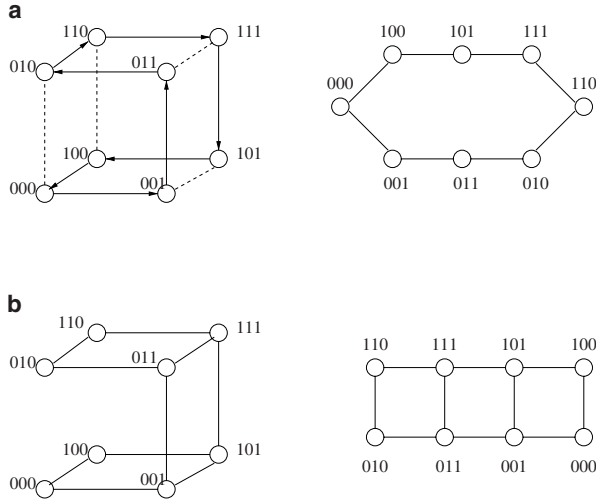


Abb. 2.11 Einbettungen in einen Hyperwürfel: **a** Einbettung eines Ringes mit 8 Knoten in einen 3-dimensionalen Hyperwürfel und **b** Einbettung eines 2-dimensionalen 2×4 Gitters in einen 3-dimensionalen Hyperwürfel

bettung des Rings dar. Für k_1 und k_2 mit $n_1 = 2^{k_1}$ und $n_2 = 2^{k_2}$, also $k_1 + k_2 = k$, werden Gray-Code $RG C_{k_1} = (a_1, \dots, a_{n_1})$ und Gray-Code $RG C_{k_2} = (b_1, \dots, b_{n_2})$ benutzt, um eine $n_1 \times n_2$ Matrix M mit Einträgen aus k -Bitworten zu konstruieren, und zwar $M(i, j) = \{a_i b_j\}_{i=1, \dots, n_1, j=1, \dots, n_2}$:

$$M = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_{n_2} \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_{n_2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n_1} b_1 & a_{n_1} b_2 & \dots & a_{n_1} b_{n_2} \end{bmatrix}.$$

Benachbarte Elemente der Matrix M unterscheiden sich in genau einer Bitposition. Dies gilt für in einer Zeile benachbarte Elemente, da identische Elemente von $RG C_{k_1}$ und benachbarte Elemente von $RG C_{k_2}$ verwendet werden. Analog gilt dies für in einer Spalte benachbarte Elemente, da identische Elemente von $RG C_{k_2}$ und benachbarte Elemente von $RG C_{k_1}$ verwendet werden. Alle Elemente von M sind unterschiedliche Bitworte der Länge k . Die Matrix M enthält also alle Namen von Knoten im k -dimensionalen Würfel genau einmal und Nachbarschaftsbeziehungen der Einträge der Matrix entsprechen Nachbarschaftsbeziehungen der Knoten im k -dimensionalen Würfel. Die Abbildung

$$\sigma : \{1, \dots, n_1\} \times \{1, \dots, n_2\} \rightarrow \{0, 1\}^k \quad \text{mit } \sigma((i, j)) = M(i, j)$$

ist also eine Einbettung in den k -dimensionalen Würfel. Abbildung 2.11b zeigt ein Beispiel.

Einbettung eines d -dimensionalen Gitters in einen k -dimensionalen Würfel

In einem d -dimensionalen Gitter mit $n_i = 2^{k_i}$ Gitterpunkten in der i -ten Dimension, $1 \leq i \leq d$, werden die insgesamt $n = n_1 \cdot \dots \cdot n_d$ Gitterpunkte jeweils als Tupel (x_1, \dots, x_d) dargestellt, $1 \leq x_i \leq n_i$. Die Abbildung

$$\sigma : \{(x_1, \dots, x_d) \mid 1 \leq x_i \leq n_i, 1 \leq i \leq d\} \longrightarrow \{0, 1\}^k$$

mit $\sigma((x_1, \dots, x_d)) = s_1 s_2 \dots s_d$ und $s_i = RGC_{k_i}(x_i)$

(d. h. s_i ist der x_i -te Bitstring im Gray-Code RGC_{k_i}) stellt eine Einbettung in den k -dimensionalen Würfel dar. Für zwei Gitterpunkte (x_1, \dots, x_d) und (y_1, \dots, y_d) , die durch eine Kante im Gitter verbunden sind, gilt nach der Definition des d -dimensionalen Gitters, dass es genau eine Komponente $i \in \{1, \dots, d\}$ mit $|x_i - y_i| = 1$ gibt und dass für alle anderen Komponenten $j \neq i$ $x_j = y_j$ gilt. Für die Bilder $\sigma((x_1, \dots, x_d)) = s_1 s_2 \dots s_d$ und $\sigma((y_1, \dots, y_d)) = t_1 t_2 \dots t_d$ sind also alle Komponenten $s_j = RGC_{k_j}(x_j) = RGC_{k_j}(y_j) = t_j$ für $j \neq i$ identisch und $RGC_{k_i}(x_i)$ unterscheidet sich von $RGC_{k_i}(y_i)$ in genau einer Bitposition. Die Knoten $s_1 s_2 \dots s_d$ und $t_1 t_2 \dots t_d$ sind also durch eine Kante im k -dimensionalen Würfel verbunden.

2.5.4 Dynamische Verbindungsnetzwerke

Dynamische Verbindungsnetzwerke stellen keine physikalischen Punkt-zu-Punkt-Verbindungen zwischen Prozessoren bereit, sondern bieten stattdessen die Möglichkeit der indirekten Verbindung zwischen Prozessoren (bei Systemen mit verteiltem Speicher) bzw. zwischen Prozessoren und Speichermodulen (bei Systemen mit gemeinsamem Speicher), worauf auch die Bezeichnung *indirektes* Verbindungsnetzwerk beruht. Aus der Sicht der Prozessoren stellt ein dynamisches Verbindungsnetzwerk eine Einheit dar, in die Nachrichten oder Speicheranforderungen eingegeben werden und aus der Nachrichten oder zurückgelieferte Daten empfangen werden. Intern ist ein dynamisches Verbindungsnetzwerk aus mehreren physikalischen Leitungen und dazwischenliegenden Schaltern aufgebaut, aus denen gemäß der Anforderungen einer Nachrichtenübertragung dynamisch eine Verbindung zwischen zwei Komponenten aufgebaut wird, was zur Bezeichnung *dynamisches Verbindungsnetzwerk* geführt hat.

Dynamische Verbindungsnetzwerke werden hauptsächlich für Systeme mit gemeinsamem Speicher genutzt, siehe Abb. 2.6. In diesem Fall kann ein Prozessor

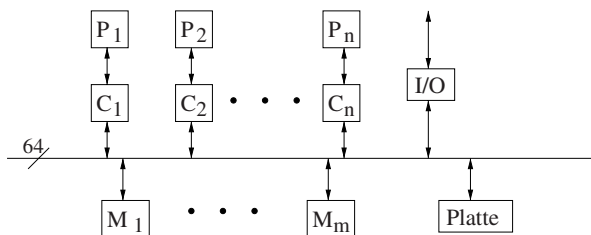


Abb. 2.12 Bus mit 64 Bit-Leitung zur Verbindung der Prozessoren P_1, \dots, P_n und ihrer Caches C_1, \dots, C_n mit den Speichermodulen M_1, \dots, M_m

nur *indirekt* über das Verbindungsnetzwerk auf den gemeinsamen Speicher zugreifen. Besteht der gemeinsame Speicher, wie dies meist der Fall ist, aus mehreren Speichermodulen, so leitet das Verbindungsnetzwerk die Datenzugriffe der Prozessoren anhand der spezifizierten Speicheradresse zum richtigen Speichermodul weiter.

Auch dynamische Verbindungsnetzwerke werden entsprechend ihrer topologischen Ausprägungen charakterisiert. Neben busbasierten Verbindungsnetzwerken werden mehrstufige Schaltnetzwerke, auch Switchingnetzwerke genannt, und Crossbars unterschieden.

Busnetzwerke

Ein *Bus* besteht im Wesentlichen aus einer Menge von Leitungen, über die Daten von einer Quelle zu einem Ziel transportiert werden können, vgl. Abb. 2.12. Um größere Datenmengen schnell zu transportieren, werden oft mehrere Hundert Leitungen verwendet. Zu einem bestimmten Zeitpunkt kann jeweils nur ein Datentransport über den Bus stattfinden (*time-sharing*). Falls zum gleichen Zeitpunkt mehrere Prozessoren gleichzeitig einen Datentransport über den Bus ausführen wollen, muss ein spezieller **Busarbitr** die Ausführung der Datentransporte koordinieren (*contention-Bus*). Busnetzwerke werden meist nur für eine kleine Anzahl von Prozessoren eingesetzt, also etwa für 32 bis 64 Prozessoren.

Crossbar-Netzwerke

Die höchste Verbindungskapazität zwischen Prozessoren bzw. zwischen Prozessoren und Speichermodulen stellt ein *Crossbar-Netzwerk* bereit. Ein $n \times m$ -Crossbar-Netzwerk hat n Eingänge, m Ausgänge und besteht aus $n \cdot m$ Schaltern wie in Abb. 2.13 skizziert ist. Für jede Datenübertragung bzw. Speicheranfrage von einem bestimmten Eingang zum gewünschten Ausgang wird ein Verbindungspfad im Netzwerk aufgebaut. Entsprechend der Anforderungen der Nachrichtenübertragung

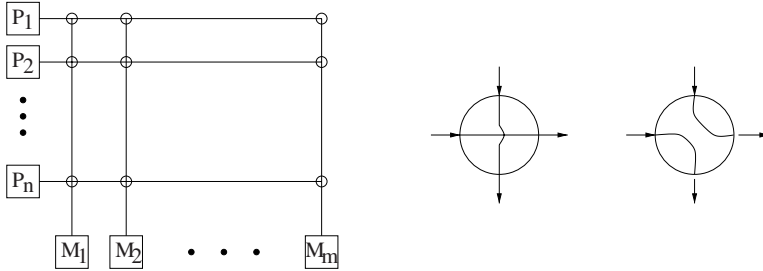


Abb. 2.13 Illustration eines $n \times m$ -Crossbar-Netzwerkes mit n Prozessoren und m Speichermodule (*links*) und der beiden möglichen Schalterstellungen der Schalter an Kreuzungspunkten des Crossbar-Netzwerkes (*rechts*)

können die Schalter an den Kreuzungspunkten des Pfades die Nachricht geradeaus oder mit einer Richtungsänderung um 90 Grad weiterleiten. Wenn wir davon ausgehen, dass jedes Speichermodule zu jedem Zeitpunkt nur eine Speicheranfrage befriedigen kann, darf in jeder Spalte von Schaltern nur *ein* Schalter auf Umlenken (also nach unten) gestellt sein. Ein Prozessor kann jedoch gleichzeitig mehrere Speicheranfragen an verschiedene Speichermodule stellen. Üblicherweise werden Crossbar-Netzwerke wegen des hohen Hardwareaufwandes nur für eine kleine Anzahl von Prozessoren realisiert.

Mehrstufige Schaltnetzwerke

Mehrstufige Schalt- oder Switchingnetzwerke (engl. *multistage switching network*) sind aus mehreren Schichten von Schaltern und dazwischenliegenden Leitungen aufgebaut. Ein Ziel besteht dabei darin, bei größerer Anzahl von zu verbindenden Prozessoren einen geringeren tatsächlichen Abstand zu erhalten als dies bei direkten Verbindungsnetzwerken der Fall wäre. Die interne Verbindungsstruktur dieser Netzwerke kann durch Graphen dargestellt werden, in denen die Schalter den Knoten und Leitungen zwischen Schaltern den Kanten entsprechen. In diese Graphdarstellung werden häufig auch die Verbindungen vom Netzwerk zu Prozessoren bzw. Speichermodule einbezogen: Prozessoren und Speichermodule sind ausgezeichnete Knoten, deren Verbindungen zu dem eigentlichen Schaltnetzwerk durch zusätzliche Kanten dargestellt werden. Charakterisierungskriterien für mehrstufige Schaltnetzwerke sind die *Konstruktionsvorschrift* des Aufbaus des entsprechenden Graphen und der *Grad* der den Schaltern entsprechenden Knoten im Graphen.

Die sogenannten **regelmäßigen mehrstufigen Verbindungsnetzwerke** zeichnen sich durch eine *regelmäßige* Konstruktionsvorschrift und einen gleichgroßen Grad von eingehenden bzw. ausgehenden Leitungen für alle Schalter aus. Die Schalter in mehrstufigen Verbindungsnetzwerken werden häufig als $a \times b$ -Crossbars

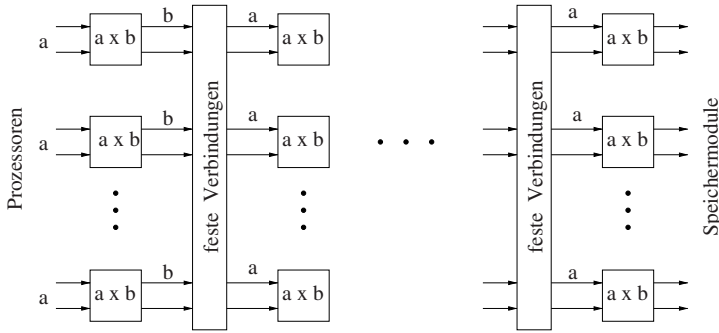


Abb. 2.14 Mehrstufige Schaltungsnetzwerke mit $a \times b$ -Crossbars als Schalter nach [87]

realisiert, wobei a den Eingangsgrad und b den Ausgangsgrad des entsprechenden Knotens bezeichnet. Die Schalter sind in einzelnen *Stufen* angeordnet, wobei benachbarte Stufen durch feste Verbindungsleitungen miteinander verbunden sind, siehe Abb. 2.14. Die Schalter der ersten Stufe haben Eingangskanten, die mit den Prozessoren verbunden sind. Die ausgehenden Kanten der letzten Schicht stellen die Verbindung zu Speichermodule (bzw. ebenfalls zu Prozessoren) dar. Speicherzugriffe von Prozessoren auf Speichermodule (bzw. Nachrichtenübertragungen zwischen Prozessoren) finden über das Netzwerk statt, indem von der Eingangskante des Prozessors bis zur Ausgangskante zum Speichermodule ein Pfad über die einzelnen Stufen gewählt wird und die auf diesem Pfad liegenden Schalter dynamisch so gesetzt werden, dass die gewünschte Verbindung entsteht.

Der Aufbau des Graphen für regelmäßige mehrstufige Verbindungsnetzwerke entsteht durch „Verkleben“ der einzelnen *Stufen* von Schaltern. Jede Stufe ist durch einen gerichteten azyklischen Graphen der Tiefe 1 mit w Knoten dargestellt. Der Grad jedes Knotens ist $g = n/w$, wobei n die Anzahl der zu verbindenden Prozessoren bzw. der nach außen sichtbaren Leitungen des Verbindungsnetzwerkes ist. Das Verkleben der einzelnen Stufen wird durch eine Permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ beschrieben, die die durchnummerierten ausgehenden Leitungen einer Stufe i so umsortiert, dass die entstehende Permutation $(\pi(1), \dots, \pi(n))$ der durchnummerierten Folge von eingehenden Leitungen in die Stufe $i + 1$ des mehrstufigen Netzwerkes entspricht. Die Partition der Permutation $(\pi(1), \dots, \pi(n))$ in w Teilstücke ergibt die geordneten Mengen der Empfangsleitungen der Knoten der nächsten Schicht. Bei regelmäßigen Verbindungsnetzwerken ist die Permutation für alle Schichten gleich, kann aber evtl. mit der Nummer i der Stufe parametrisiert sein. Bei gleichem Grad der Knoten ergibt die Partition von $(\pi(1), \dots, \pi(n))$ w gleichgroße Teilmengen.

Häufig benutzte regelmäßige mehrstufige Verbindungsnetzwerke sind das Omega-Netzwerk, das Baseline-Netzwerk und das Butterfly-Netzwerk (oder Banyan-Netzwerk), die jeweils Schalter mit Eingangs- und Ausgangsgrad 2 haben und

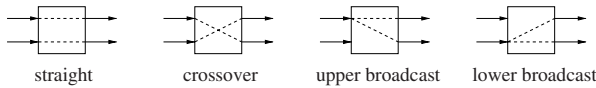


Abb. 2.15 Schalterstellungen, die ein Schalter in einem Omega-, Baseline- oder Butterfly-Netzwerk realisieren kann

aus $\log n$ Schichten bestehen. Die 2×2 Schalter können vier mögliche Schalterstellungen annehmen, die in Abb. 2.15 dargestellt sind.

Im Folgenden stellen wir reguläre Verbindungsnetzwerke wie das Omega-, das Baseline- und das Butterfly-Netzwerk sowie das Benes-Netzwerk und den Fat-Tree vor. Ausführliche Beschreibungen dieser Netzwerke sind z. B. in [111] zu finden.

Omega-Netzwerk

Ein $n \times n$ -Omega-Netzwerk besteht aus 2×2 -Crossbar-Schaltern, die in $\log n$ Stufen angeordnet sind, wobei jede Stufe $n/2$ Schalter enthält und jeder Schalter zwei Eingänge und zwei Ausgänge hat. Insgesamt gibt es also $(n/2) \log n$ Schalter. Dabei sei $\log n \equiv \log_2 n$. Jeder der Schalter kann vier Verbindungen realisieren, siehe Abb. 2.15. Die festen Verbindungen zwischen den Stufen, d. h. also die Permutationsfunktion zum Verkleben der Stufen, ist beim Omega-Netzwerk für alle Stufen gleich und hängt nicht von der Nummer der Stufe ab. Die Verklebungsfunktion des Omega-Netzwerkes ist über eine Nummerierung der Schalter definiert. Die Namen der Schalter sind Paare (α, i) bestehend aus einem $(\log n - 1)$ -Bitwort α , $\alpha \in \{0, 1\}^{\log n - 1}$, und einer Zahl $i \in \{0, \dots, \log n - 1\}$, die die Nummer der Stufe angibt. Es gibt jeweils eine Kante von Schalter (α, i) in Stufe i zu den beiden Schaltern $(\beta, i + 1)$ in Stufe $i + 1$, die dadurch definiert sind, dass

1. entweder β durch einen zyklischen Linksshift aus α hervorgeht oder
2. β dadurch entsteht, dass nach einem zyklischen Linksshift von α das letzte (rechteste) Bit invertiert wird.

Ein $n \times n$ -Omega-Netzwerk wird auch als $(\log n - 1)$ -dimensionales Omega-Netzwerk bezeichnet. Abbildung 2.16a zeigt ein 16×16 , also ein dreidimensionales, Omega-Netzwerk mit vier Stufen und acht Schaltern pro Stufe.

Butterfly-Netzwerk

Das k -dimensionale Butterfly-Netzwerk, das auch als **Banyan-Netzwerk** bezeichnet wird, verbindet ebenfalls $n = 2^{k+1}$ Eingänge mit $n = 2^{k+1}$ Ausgängen über ein Netzwerk, das aus $k + 1$ Stufen mit jeweils 2^k Knoten aus 2×2 -Crossbar-Schaltern aufgebaut ist. Die insgesamt $(k + 1)2^k$ Knoten des Butterfly-Netzwerkes können eindeutig durch Paare (α, i) bezeichnet werden, wobei i ($0 \leq i \leq k$) die

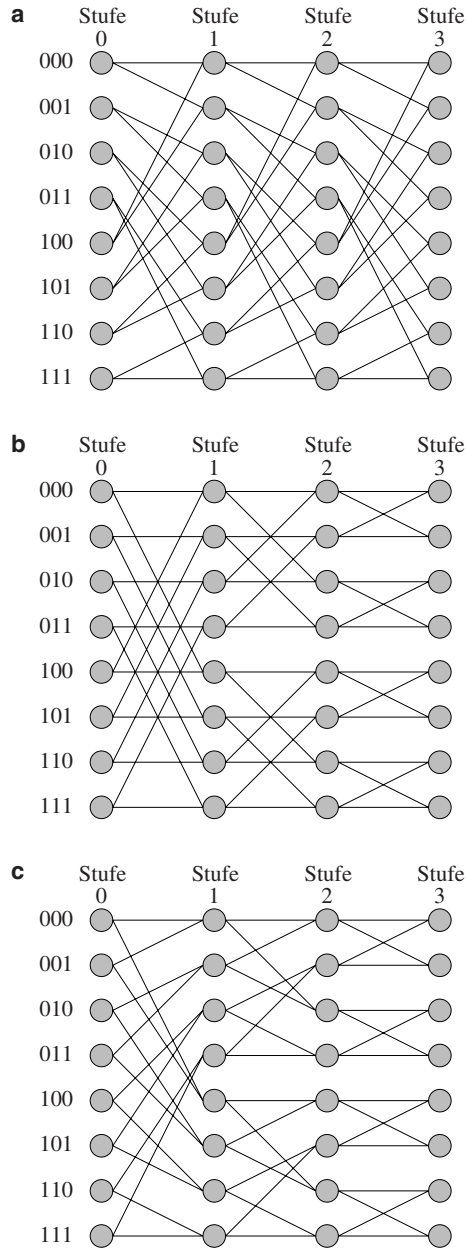


Abb. 2.16 Spezielle dynamische Verbindungsnetzwerke: **a** 16×16 Omega-Netzwerk, **b** 16×16 Butterfly-Netzwerk, **c** 16×16 Baseline-Netzwerk. Alle Netzwerke sind 3-dimensional

Stufe angibt und das k -Bit-Wort $\alpha \in \{0, 1\}^k$ die Position des Knotens in dieser Stufe. Die Verklebungsfunktion der Stufen i und $i + 1$ mit $0 \leq i < k$ des Butterfly-Netzwerkes ist folgendermaßen definiert. Zwei Knoten (α, i) und $(\alpha', i + 1)$ sind genau dann miteinander verbunden, wenn:

1. α und α' identisch sind (direkte Kante, engl. *straight edge*) oder
2. α und α' sich genau im $(i + 1)$ -ten Bit von links unterscheiden (Kreuzkante, engl. *cross edge*).

Abbildung 2.16b zeigt ein 16×16 -Butterfly-Netzwerk mit vier Stufen.

Baseline-Netzwerk

Das k -dimensionale Baseline-Netzwerk hat dieselbe Anzahl von Knoten, Kanten und Stufen wie das Butterfly-Netzwerk. Die Stufen werden durch folgende Verklebungsfunktion verbunden. Knoten (α, i) ist für $0 \leq i < k$ genau dann mit Knoten $(\alpha', i + 1)$ verbunden, wenn:

1. Das k -Bit-Wort α' aus α durch einen zyklischen Rechtsshift der letzten $k - i$ Bits von α entsteht oder
2. Das k -Bit-Wort α' aus α entsteht, indem zuerst das letzte (rechteste) Bit von α invertiert wird und dann ein zyklischer Rechtsshift auf die letzten $k - i$ Bits des entstehenden k -Bit-Wortes angewendet wird.

Abbildung 2.16c zeigt ein 16×16 -Baseline-Netzwerk mit vier Stufen.

Benes-Netzwerk

Das k -dimensionale Benes-Netzwerk setzt sich aus zwei k -dimensionalen Butterfly-Netzwerken zusammen und zwar so, dass die ersten $k + 1$ Stufen ein Butterfly-Netzwerk bilden und die letzten $k + 1$ Stufen ein bzgl. der Stufen umgekehrtes Butterfly-Netzwerk bilden, wobei die $(k + 1)$ -te Stufe des ersten Butterfly-Netzwerkes und die erste Stufe des umgekehrten Butterfly-Netzwerkes zusammenfallen. Insgesamt hat das k -dimensionale Benes-Netzwerk also $2k + 1$ Stufen mit je $N = 2^k$ Schalter pro Stufe und eine Verklebungsfunktion der Stufen, die (entsprechend modifiziert) vom Butterfly-Netzwerk übernommen wird. Ein Beispiel eines Benes-Netzwerkes für 16 Eingangskanten ist in Abb. 2.17a gegeben, vgl. [111].

Fat-Tree

Ein *dynamischer Baum* oder *Fat-Tree* hat als Grundstruktur einen vollständigen, binären Baum, der jedoch (im Gegensatz zum Baum-Netzwerk aus Abschn. 2.5.2) zur

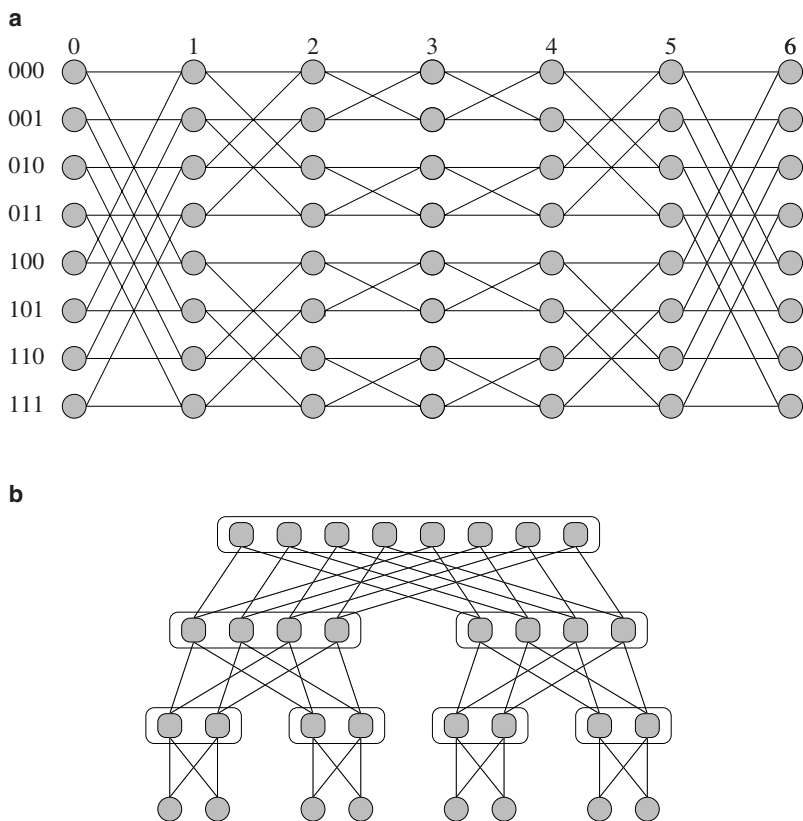


Abb. 2.17 Spezielle dynamische Verbindungsnetzwerke: **a** 3-dimensionales Benes-Netzwerk und **b** Fattree für 16 Prozessoren

Wurzel hin mehr Kanten aufweist und so den Flaschenhals des Baumes an der Wurzel überwindet. Innere Knoten des Fat-Tree bestehen aus Schaltern, deren Aussehen von der Ebene im Baum abhängen. Stellt ein Fat-Tree ein Netzwerk für n Prozessoren dar, die durch die Blätter des Baumes repräsentiert sind, so hat ein Knoten auf Ebene i für $i = 1, \dots, \log n$ genau 2^i Eingangskanten und 2^i Ausgangskanten. Dabei ist Ebene 0 die Blattebene. Realisiert wird dies z. B. dadurch, dass die Knoten auf Ebene i intern aus 2^{i-1} Schaltern mit je zwei Ein- und Ausgangskanten bestehen. Damit besteht jede Ebene i aus insgesamt $n/2$ Schaltern, die in $2^{\log n - i}$ Knoten gruppiert sind. Dies ist in Abb. 2.17b für einen Fat-Tree mit vier Ebenen skizziert, wobei nur die inneren Schalterknoten, nicht aber die die Prozessoren repräsentierenden Blattknoten dargestellt sind.

2.6 Routing- und Switching-Strategien

Direkte und indirekte Verbindungsnetzwerke bilden die physikalische Grundlage zum Verschicken von Nachrichten zwischen Prozessoren bei Systemen mit physikalisch verteiltem Speicher oder für den Speicherzugriff bei Systemen mit gemeinsamem Speicher. Besteht zwischen zwei Prozessoren keine direkte Punkt-zu-Punkt-Verbindung und soll eine Nachricht von einem Prozessor zum anderen Prozessor geschickt werden, muss ein Pfad im Netzwerk für die Nachrichtenübertragung gewählt werden. Dies ist sowohl bei direkten als auch bei indirekten Netzwerken der Fall.

2.6.1 Routingalgorithmen

Ein **Routingalgorithmus** bestimmt einen Pfad im Netzwerk, über den eine Nachricht von einem Sender *A* an einen Empfänger *B* geschickt werden soll. Üblicherweise ist eine topologiespezifische Vorschrift gegeben, die an jedem Zwischenknoten auf dem Pfad vom Sender zum Ziel angibt, zu welchem Folgeknoten die zu transportierende Nachricht weitergeschickt werden soll. Hierbei bezeichnen *A* und *B* zwei Knoten im Netzwerk (bzw. die zu den Knoten im Netzwerk gehörenden Verarbeitungseinheiten).

Üblicherweise befinden sich mehrere Nachrichtenübertragungen im Netz, so dass ein Routingalgorithmus eine gleichmäßige Auslastung der Leitungen im Netzwerk erreichen und Deadlockfreiheit garantieren sollte. Eine Menge von Nachrichten befindet sich in einer **Deadlocksituation**, wenn jede dieser Nachrichten jeweils über eine Verbindung weitergeschickt werden soll, die von einer anderen Nachricht derselben Menge gerade benutzt wird. Ein Routingalgorithmus wählt nach Möglichkeit von den Pfaden im Netzwerk, die Knoten *A* und *B* verbinden, denjenigen aus, der die geringsten Kosten verursacht. Die Kommunikationskosten, also die Zeit zwischen dem Absetzen einer Nachricht bei *A* und dem Ankommen bei *B*, hängen nicht nur von der Länge eines Pfades ab, sondern auch von der Belastung der Leitungen durch andere Nachrichten. Bei der Auswahl eines Routingpfades werden also die folgenden Punkte berücksichtigt:

- *Topologie*: Die Topologie des zugrunde liegenden Netzwerkes bestimmt die Pfade, die den Sender *A* mit Empfänger *B* verbinden und damit zum Versenden prinzipiell in Frage kommen.
- *Netzwerk-Contention bei hohem Nachrichtenaufkommen*: **Contention** liegt vor, wenn zwei oder mehrere Nachrichten zur gleichen Zeit über dieselbe Verbindung geschickt werden sollen und es durch die konkurrierenden Anforderungen zu Verzögerungen bei der Nachrichtenübertragung kommt.

- *Vermeidung von Staus bzw. Congestion.* **Congestion** entsteht, falls zu viele Nachrichten auf eine beschränkte Ressource (also Verbindungsleitung oder Puffer) treffen, so dass Puffer überfüllt werden und es dazu kommt, dass Nachrichten weggeworfen werden. Im Unterschied zu Contention treten also bei Congestion so viele Nachrichten auf, dass das Nachrichtenaufkommen nicht mehr bewältigt werden kann [141].

Routingalgorithmen werden in verschiedensten Ausprägungen vorgeschlagen. Eine Klassifizierung bzgl. der Pfadlänge unterscheidet **minimale** Routingalgorithmen und **nichtminimale** Routingalgorithmen. Minimale Routingalgorithmen wählen für eine Nachrichtenübertragung immer den kürzesten Pfad aus, so dass die Nachricht durch Verschieben über jede Einzelverbindung des Pfades näher zum Zielknoten gelangt, aber die Gefahr von Staus gegeben ist. Nichtminimale Routingalgorithmen verschicken Nachrichten über nichtminimale Pfade, wenn die Netzwerkauslastung dies erforderlich macht. Die Länge eines Pfades muss je nach Switching-Technik (vgl. Abschn. 2.6.2) zwar nicht direkt Auswirkungen auf die Kommunikationszeit haben, kann aber indirekt zu mehr Möglichkeiten für Contention oder Congestion führen. Ist das Netzwerk sehr belastet, muss aber der kürzeste Pfad zwischen Knoten *A* und *B* nicht der beste sein.

Eine weitere Klassifizierung ergibt sich durch Unterscheidung von **deterministischen** Routingalgorithmen und **adaptiven** Routingalgorithmen. Deterministisches Routing legt einen eindeutigen Pfad zur Nachrichtenübermittlung nur in Abhängigkeit von Sender und Empfänger fest. Die Auswahl des Pfades kann *quellenbasiert*, also nur durch den Sendeknoten, oder *verteilt* an den Zwischenknoten vorgenommen werden. Deterministisches Routing kann zu ungleichmäßiger Netzauslastung führen. Ein Beispiel für deterministisches Routing ist das **dimensionsgeordnete Routing** (engl. *dimension ordered routing*), das den Routing-Pfad entsprechend der Position von Quell- und Zielknoten und der Reihenfolge der Dimensionen der zugrunde liegenden Topologie auswählt. Adaptives Routing hingegen nutzt Auslastungsinformationen zur Wahl des Pfades aus, um Contention zu vermeiden. Bei adaptivem Routing werden mehrere mögliche Pfade zwischen zwei Knoten zum Nachrichtenaustausch bereitgestellt, wodurch nicht nur eine *größere Fehlertoleranz* für den möglichen Ausfall einzelner Verbindungen erreicht wird, sondern auch eine *gleichmäßigere* Auslastung des Netzwerkes. Auch bei adaptiven Routingalgorithmen wird zwischen minimalen und nichtminimalen Algorithmen unterschieden. Insbesondere für minimale adaptive Routingalgorithmen wird das Konzept von **virtuellen Kanälen** verwendet, auf die wir weiter unten eingehen. Routingalgorithmen werden etwa im Übersichtsartikel [124] vorgestellt, siehe auch [32, 87, 111]. Wir stellen im Folgenden eine Auswahl von Routingalgorithmen vor.

Dimensionsgeordnetes Routing

XY -Routing in einem 2-dimensionalen Gitter

XY -Routing ist ein dimensionsgeordneter Routingalgorithmus für zweidimensionale Gittertopologien. Die Positionen der Knoten in der Gittertopologie werden mit X - und Y -Koordinaten bezeichnet, wobei die X -Koordinate der horizontalen und die Y -Koordinate der vertikalen Ausrichtung entspricht. Zum Verschicken einer Nachricht von Quellknoten A mit Position (X_A, Y_A) zu Zielknoten B mit Position (X_B, Y_B) wird die Nachricht so lange in (positive oder negative) X -Richtung geschickt, bis die X -Koordinate X_B von Knoten B erreicht ist. Anschließend wird die Nachricht in Y -Richtung geschickt, bis die Y -Koordinate Y_B erreicht ist. Die Länge der Pfade ist $|X_A - X_B| + |Y_A - Y_B|$. Der Routingalgorithmus ist also deterministisch und minimal.

E -Cube-Routing für den k -dimensionalen Hyperwürfel

In einem k -dimensionalen Würfel ist jeder der $n = 2^k$ Knoten direkt mit k physikalischen Nachbarn verbunden. Wird jedem Knoten, wie in Abschn. 2.5.2 eingeführt, ein binäres Wort der Länge k als Namen zugeordnet, so ergeben sich die Namen der k physikalischen Nachbarn eines Knotens genau durch Invertierung eines der k Bits seines Namens. Dimensionsgerichtetes Routing für den k -dimensionalen Würfel [176] benutzt die k -Bitnamen von Sender und Empfänger und von dazwischenliegenden Knoten zur Bestimmung des Routing-Pfades. Soll eine Nachricht von Sender A mit Bitnamen $\alpha = \alpha_0 \dots \alpha_{k-1}$ an Empfänger B mit Bitnamen $\beta = \beta_0 \dots \beta_{k-1}$ geschickt werden, so wird beginnend bei A nacheinander ein Nachfolgerknoten entsprechend der Dimension gewählt, zu dem die Nachricht im nächsten Schritt geschickt werden soll. Ist A_i mit Bitdarstellung $\gamma = \gamma_0 \dots \gamma_{k-1}$ der Knoten auf dem Routing-Pfad $A = A_0, A_1, \dots, A_l = B$, von dem aus die Nachricht im nächsten Schritt weitergeleitet werden soll, so:

- Berechnet A_i das k -Bitwort $\gamma \oplus \beta$, wobei der Operator \oplus das bitweise ausschließende Oder (d.h. $0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0$) bezeichnet, und
- Schickt die Nachricht in Richtung der Dimension d , wobei d die am weitesten rechts liegende Position von $\gamma \oplus \beta$ ist, die den Wert 1 hat. Den zugehörigen Knoten A_{i+1} auf dem Routingpfad erhält man durch Invertierung des d -ten Bits in γ , d.h. der Knoten A_{i+1} hat den k -Bit-Namen $\delta = \delta_0 \dots \delta_{k-1}$ mit $\delta_j = \gamma_j$ für $j \neq d$ und $\delta_d = \bar{\gamma}_d$ (Bitumkehrung). Wenn $\gamma \oplus \beta = 0$ ist, ist der Zielknoten erreicht.

Beispiel

Um eine Nachricht von A mit Bitnamen $\alpha = 010$ nach B mit Bitnamen $\beta = 111$ zu schicken, wird diese also zunächst in Richtung Dimension $d = 2$ nach A_1 mit Bitnamen 011 geschickt (da $\alpha \oplus \beta = 101$ gilt) und dann in Richtung Dimension $d = 0$ zu β (da $011 \oplus 111 = 100$ gilt).

Deadlockgefahr bei Routingalgorithmen

Befinden sich mehrere Nachrichten im Netzwerk, was der Normalfall ist, so kann es zu Deadlocksituationen kommen, in denen der Weitertransport einer Teilmenge von Nachrichten für immer blockiert wird. Dies kann insbesondere dann auftreten, wenn Ressourcen im Netzwerk nur von jeweils einer Nachricht genutzt werden können. Werden z.B. die Verbindungskanäle zwischen zwei Knoten jeweils nur einer Nachricht zugeteilt und wird ein Verbindungskanal nur freigegeben, wenn der folgende Verbindungskanal für den Weitertransport zugeteilt werden kann, so kann es durch wechselseitiges Anfordern von Verbindungskanälen zu einem solchen Deadlock kommen. Genau dieses Zustandekommen von Deadlocksituationen kann durch geeignete Routingalgorithmen vermieden werden. Andere Deadlocksituationen, die durch beschränkte Ein- und Ausgabepuffer der Verbindungskanäle oder ungünstige Reihenfolgen von Send- und Empfangsbefehlen entstehen können, werden in den Abschnitten über Switching bzw. Message-Passing-Programmierung betrachtet, siehe Abschn. 2.6.2 und Kap. 5.

Zum Beweis der Deadlockfreiheit von Routingalgorithmen werden mögliche Abhängigkeiten zwischen Verbindungskanälen betrachtet, die durch beliebige Nachrichtenübertragungen entstehen können. Eine *Abhängigkeit zwischen den Verbindungskanälen* l_1 und l_2 besteht, falls es durch den Routingalgorithmus möglich ist, einen Pfad zu wählen, der eine Nachricht über Verbindung l_1 und direkt danach über Verbindung l_2 schickt. Diese Abhängigkeit zwischen Verbindungskanälen kann im **Kanalabhängigkeitsgraph** (engl. *channel dependency graph*) ausgedrückt werden, der die Verbindungskanäle als Knoten darstellt und für jede Abhängigkeit zwischen Kanälen eine Kante enthält. Enthält dieser Kanalabhängigkeitsgraph keine Zyklen, so ist der entsprechende Routingalgorithmus auf der gewählten Topologie deadlockfrei, da kein Kommunikationsmuster eines Deadlocks entstehen kann.

Für Topologien, die keine Zyklen enthalten, ist jeder Kanalabhängigkeitsgraph zyklensfrei, d.h. jeder Routingalgorithmus auf einer solchen Topologie ist deadlockfrei. Für Netzwerke mit Zyklen muss der Kanalabhängigkeitsgraph analysiert werden. Wir zeigen im Folgenden, dass das oben eingeführte XY -Routing für zweidimensionale Gitter mit bidirektionalen Verbindungen deadlockfrei ist.

Deadlockfreiheit für XY -Routing

Der für das XY -Routing resultierende Kanalabhängigkeitsgraph enthält für jede unidirektionale Verbindung des zweidimensionalen $n_x \times n_y$ -Gitters einen Knoten, also zwei Knoten für jede bidirektionale Kante des Gitters. Es gibt eine Abhängigkeit von Verbindung u zu Verbindung v , falls sich v in der gleichen horizontalen oder vertikalen Ausrichtung oder in einer 90-Grad-Drehung nach oben oder unten an u anschließt. Zum Beweis der Deadlockfreiheit werden alle unidirektionalen Verbindungen des Gitters auf folgende Weise nummeriert:

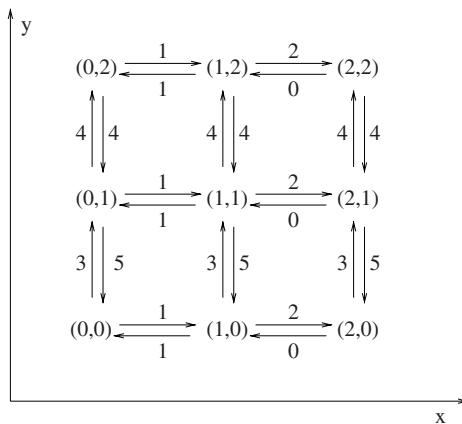
- Horizontale Kanten zwischen Knoten mit Position (i, y) und Knoten mit Position $(i + 1, y)$ erhalten die Nummer $i + 1$, $i = 0, \dots, n_x - 2$, und zwar für jede y -Position. Die entgegengesetzten Kanten von $(i + 1, y)$ nach (i, y) erhalten die Nummer $n_x - 1 - (i + 1) = n_x - i - 2$, $i = 0, \dots, n_x - 2$. Die Kanten in aufsteigender x -Richtung sind also aufsteigend mit $1, \dots, n_x - 1$, die Kanten in absteigender x -Richtung sind aufsteigend mit $0, \dots, n_x - 2$ nummeriert.
- Die vertikalen Kanten von (x, j) nach $(x, j + 1)$ erhalten die Nummern $j + n_x$, $j = 0, \dots, n_y - 2$ und die entgegengesetzten Kanten erhalten die Nummern $n_x + n_y - (j + 1)$.

Abbildung 2.18 zeigt ein 3×3 -Gitter und den zugehörigen Kanalabhängigkeitsgraphen bei Verwendung von XY -Routing, wobei die Knoten des Graphen mit den Nummern der zugehörigen Netzwerkkanten bezeichnet sind. Da alle Kanten im Kanalabhängigkeitsgraphen von einer Netzwerkkante mit einer niedrigeren Nummer zu einer Netzwerkkante mit einer höheren Nummer führen, kann eine Verzögerung einer Übertragung entlang eines Routingpfades nur dann auftreten, wenn die Nachricht nach Übertragung über eine Kante v mit Nummer i auf die Freigabe einer nachfolgenden Kante w mit Nummer $j > i$ wartet, da diese Kante gerade von einer anderen Nachricht verwendet wird (Verzögerungsbedingung). Zum Auftreten eines Deadlocks wäre es also erforderlich, dass es eine Menge von Nachrichten N_1, \dots, N_k und Netzwerkkanten n_1, \dots, n_k gibt, so dass jede Nachricht N_i für $1 \leq i < k$ gerade Kante n_i für die Übertragung verwendet und auf die Freigabe von Kante n_{i+1} wartet, die gerade von Nachricht N_{i+1} zur Übertragung verwendet wird. Außerdem überträgt N_k gerade über Kante n_k und wartet auf die Freigabe von n_1 durch N_1 . Wenn $n()$ die oben für die Netzwerkkanten eingeführte Nummerierung ist, gilt wegen der Verzögerungsbedingung

$$n(n_1) < n(n_2) < \dots < n(n_k) < n(n_1) .$$

Da dies ein Widerspruch ist, kann kein Deadlock auftreten. Jeder mögliche XY -Routing-Pfad besteht somit aus einer Folge von Kanten mit *aufsteigender* Kantennummerierung. Alle Kanten im Kanalabhängigkeitsgraphen führen zu einer höher nummerierten Verbindung. Es kann somit *keinen* Zyklus im Kantenabhängigkeitsgraphen geben.

Ein ähnliches Vorgehen kann verwendet werden, um die Deadlockfreiheit von E -Cube-Routing zu beweisen, vgl. [34].

2-dimensionalales Gitter mit 3×3 Knoten

Kanalabhängigkeitsgraph

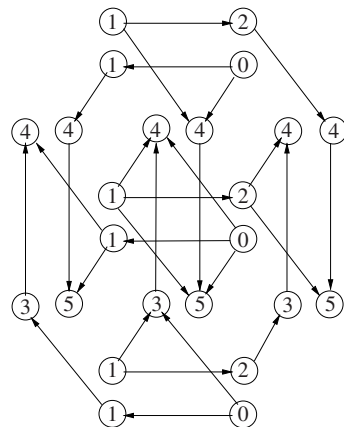


Abb. 2.18 3×3 -Gitter und zugehöriger Kanalabhängigkeitsgraph bei Verwendung von XY -Routing

Quellenbasiertes Routing

Ein weiterer deterministischer Routingalgorithmus ist das quellenbasierte Routing, bei dem der Sender den gesamten Pfad zur Nachrichtenübertragung auswählt. Für jeden Knoten auf dem Pfad wird der zu wählende Ausgabekanal festgestellt und die Folge der nacheinander zu wählenden Ausgabekanäle a_0, \dots, a_{n-1} wird als Header der eigentlichen Nachricht angefügt. Nachdem die Nachricht einen Knoten passiert hat, wird die Routinginformation im Header der den Knoten verlassenden Nachricht aktualisiert, indem der gerade passierte Ausgabekanal aus dem Pfad entfernt wird.

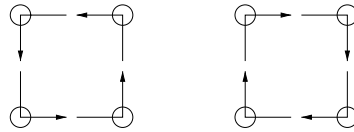
Tabellenorientiertes Routing

(engl. *table lookup routing*). Beim tabellenorientierten Routing enthält jeder Knoten des Netzwerkes eine Routingtabelle, die für jede Zieladresse den zu wählenden Ausgabekanal bzw. den nächsten Knoten enthält. Kommt eine Nachricht in einem Knoten an, so wird die Zielinformation betrachtet und in der Routingtabelle nachgesehen, wohin die Nachricht weiter zu verschicken ist.

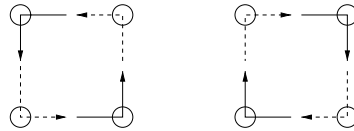
Turn-Modell

Das Turn-Modell (von [59] dargestellt in [124]) versucht Deadlocks durch geschickte Wahl erlaubter Richtungswechsel zu vermeiden. Die Ursache für das

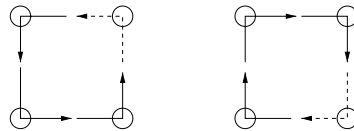
Mögliche Richtungswechsel im zwei-dimensionalen Gitter



Richtungswechsel bei XY-Routing



Richtungswechsel bei West-First-Routing



Erlaubte Richtungswechsel



Nicht erlaubte Richtungswechsel



Abb. 2.19 Illustration der Richtungswechsel beim Turn-Modell im zwei-dimensionalen Gitter mit Darstellung aller Richtungswechsel und der erlaubten Richtungswechsel bei XY-Routing bzw. West-First-Routing

Auftreten von Deadlocks besteht darin, dass Nachrichten ihre Übertragungsrichtung so ändern, dass bei ungünstigem Zusammentreffen ein zyklisches Warten entsteht. Deadlocks können vermieden werden, indem gewisse Richtungsänderungen untersagt werden. Ein Beispiel ist das *XY-Routing*, bei dem alle Richtungsänderungen von vertikaler Richtung in horizontale Richtung ausgeschlossen sind. Von den insgesamt acht möglichen Richtungsänderungen in einem zweidimensionalen Gitter, sind also nur vier Richtungsänderungen erlaubt, vgl. Abb. 2.19. Diese restlichen vier möglichen Richtungsänderungen erlauben *keinen* Zyklus, schließen Deadlocks also aus, machen allerdings auch adaptives Routing unmöglich. Im Turn-Modell für n -dimensionale Gitter und allgemeine k -fache n -Würfel wird eine minimale Anzahl von Richtungsänderungen ausgewählt, bei deren Ausschluss bei der Wahl eines Routingpfades die Bildung von Zyklen vermieden wird. Konkrete Beispiele sind das *West-First-Routing* bei zweidimensionalen Gittern oder das *P-cube-Routing* bei n -dimensionalen Hyperwürfeln.

Beim **West-First-Routing** für zweidimensionale Gitter werden nur zwei der insgesamt acht möglichen Richtungsänderungen ausgeschlossen, und zwar die Richtungsänderungen nach Westen, also nach links, so dass nur noch die Richtungsän-

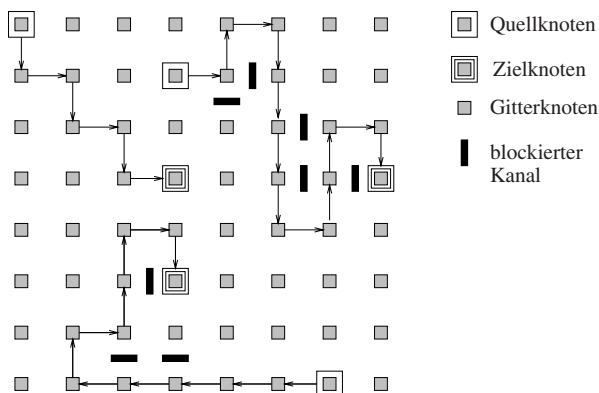


Abb. 2.20 Illustration der Pfadwahl beim West-First-Routing in einem 8×8 -Gitter. Die als blockiert gekennzeichneten Kanäle werden von anderen Nachrichten verwendet und stehen daher nicht für die Nachrichtenübertragung zur Verfügung. Einer der dargestellten Pfade ist minimal, die anderen beiden sind nicht-minimal, da bestimmte Kanäle blockiert sind

derungen, die in Abb. 2.19 angegeben sind, erlaubt sind. Routingpfade werden so gewählt, dass die Nachricht zunächst nach Westen (d. h. nach links) geschickt wird, bis mindestens die gewünschte x -Koordinate erreicht ist, und dann adaptiv nach Süden (d. h. unten), nach Osten (d. h. rechts) oder Norden (d. h. oben). Beispiele von Routingpfaden sind in Abb. 2.20 gegeben [124]. West-First-Routing ist deadlock-frei, da Zyklen vermieden werden. Bei der Auswahl von minimalen Routingpfaden ist der Algorithmus nur dann adaptiv, falls das Ziel im Osten (d. h. rechts) liegt. Bei Verwendung nichtminimaler Routingpfade ist der Algorithmus immer adaptiv.

Beim **P-cube-Routing** für den n -dimensionalen Hyperwürfel werden für einen Sender A mit dem n -Bitnamen $\alpha_0 \dots \alpha_{n-1}$ und einen Empfänger B mit dem n -Bitnamen $\beta_0 \dots \beta_{n-1}$ die unterschiedlichen Bits dieser beiden Namen betrachtet. Die Anzahl der unterschiedlichen Bits entspricht der Hammingdistanz von A und B und ist die Mindestlänge eines möglichen Routingpfades. Die Menge $E = \{i \mid \alpha_i \neq \beta_i, i = 0, \dots, n-1\}$ der Positionen der unterschiedlichen Bits wird in zwei Mengen zerlegt und zwar in $E_0 = \{i \in E \mid \alpha_i = 0 \text{ und } \beta_i = 1\}$ und $E_1 = \{i \in E \mid \alpha_i = 1 \text{ und } \beta_i = 0\}$. Das Verschicken einer Nachricht von A nach B wird entsprechend der Mengen in zwei Phasen unterteilt. Zuerst wird die Nachricht über die Dimensionsrichtungen in E_0 geschickt, danach erst über die Dimensionsrichtungen in E_1 .

Virtuelle Kanäle

Insbesondere für minimale adaptive Routingalgorithmen wird das Konzept von *virtuellen Kanälen* verwendet, da für manche Verbindungen mehrere Kanäle zwischen

2-dimensionales Gitter mit virtuellen Kanälen in y-Richtung

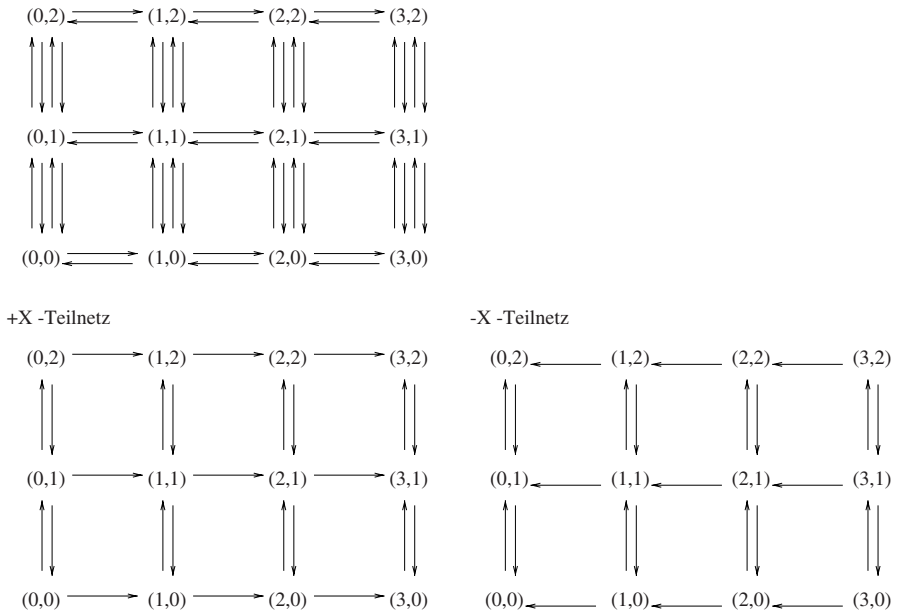


Abb. 2.21 Zerlegung eines zweidimensionalen Gitters mit virtuellen Kanälen in ein $+X$ -Teilnetz und ein $-X$ -Teilnetz für die Anwendung eines minimalen adaptiven Routingalgorithmus

benachbarten Knoten benötigt werden. Da die Realisierungen mehrerer physikalischer Verbindungen zu teuer ist, werden mehrere virtuelle Kanäle eingeführt, die sich eine physikalische Verbindung teilen. Für jeden virtuellen Kanal werden separate Puffer zur Verfügung gestellt. Die Zuteilung der physikalischen Verbindungen zu den virtuellen Verbindungen sollte *fair* erfolgen, d. h. jede virtuelle Verbindung sollte immer wieder genutzt werden können.

Der folgende *minimale adaptive Routingalgorithmus* benutzt virtuelle Kanäle und zerlegt das gegebene Netzwerk in logische Teilnetzwerke. Der Zielknoten einer Nachricht bestimmt, durch welches Teilnetz die Nachricht transportiert wird. Wir demonstrieren die Arbeitsweise für ein zweidimensionales Gitter. Ein zweidimensionales Gitter wird in zwei Teilnetze zerlegt, und zwar in ein $+X$ -Teilnetz und ein $-X$ -Teilnetz, siehe Abb. 2.21. Jedes Teilnetz enthält alle Knoten, aber nur einen Teil der virtuellen Kanäle. Das $+X$ -Teilnetz enthält in vertikaler Richtung Verbindungen zwischen allen benachbarten Knoten, in horizontaler Richtung aber nur Kanten in positiver Richtung. Das $-X$ -Teilnetz enthält ebenfalls Verbindungen zwischen allen vertikal benachbarten Knoten – was durch Verwendung von virtuellen Kanälen möglich ist – sowie alle horizontalen Kanten in negativer Richtung. Nachrichten von Knoten A mit x -Koordinate x_A nach Knoten B mit x -Koordinate x_B werden im $+X$ -Netz verschickt, wenn $x_A < x_B$ ist. Nachrichten von Kno-

ten A nach B mit $x_A > x_B$ werden im $-X$ -Netz verschickt. Für $x_A = x_B$ kann ein beliebiges Teilnetz verwendet werden. Die genaue Auswahl kann anhand der Auslastung des Netzwerkes getroffen werden. Dieser minimale adaptive Routingalgorithmus ist deadlockfrei [124]. Für andere Topologien wie den Hyperwürfel oder den Torus können mehr zusätzliche Leitungen nötig sein, um Deadlockfreiheit zu gewährleisten, vgl. [124].

Ein *nichtminimaler* adaptiver Routingalgorithmus kann Nachrichten auch über längere Pfade verschicken, falls kein minimaler Pfad zur Verfügung steht. Der **statische umgekehrt-dimensionsgeordnete Routingalgorithmus** (engl. *dimension reversal routing algorithm*) kann auf beliebige Gittertopologien und k -fache d -Würfel angewendet werden. Der Algorithmus benutzt r Paare von (virtuellen) Kanälen zwischen jedem durch einen physikalischen Kanal miteinander verbundenen Knotenpaar und zerlegt das Netzwerk in r Teilnetzwerke, wobei das i -te Teilnetzwerk für $i = 0, \dots, r-1$ alle Knoten und die i -ten Verbindungen zwischen den Knoten umfasst. Jeder Nachricht wird zusätzlich eine Klasse c zugeordnet, die zu Anfang auf $c = 0$ gesetzt wird und die im Laufe der Nachrichtenübertragung Klassen $c = 1, \dots, r-1$ annehmen kann. Eine Nachricht mit Klasse $c = i$ kann im i -ten Teilnetz in jede Richtung transportiert werden, wobei aber die verschiedenen Dimensionen in aufsteigender Reihenfolge durchlaufen werden müssen. Eine Nachricht kann aber auch entgegen der Dimensionsordnung, d. h. von einem höher-dimensionalen Kanal zu einem niedriger-dimensionalen Kanal transportiert werden. In diesem Fall wird die Klasse der Nachricht um 1 erhöht (umgekehrte Dimensionsordnung). Der Parameter r begrenzt die Möglichkeiten der Dimensionsumkehrung. Ist die maximale Klasse erreicht, so wird der Routingpfad entsprechend dem dimensionsgeordneten Routing beendet.

Routing im Omega-Netzwerk

Das in Abschn. 2.5.4 beschriebene Omega-Netzwerk ermöglicht ein Weiterleiten von Nachrichten mit Hilfe eines verteilten Kontrollschemas, in dem jeder Schalter die Nachricht *ohne* Koordination mit anderen Schaltern weiterleiten kann. Zur Beschreibung des Routingalgorithmus ist es günstig, die n Eingangs- und Ausgangskanäle mit Bitnamen der Länge $\log n$ zu benennen [111]. Zum Weiterleiten einer Nachricht vom Eingangskanal mit Bitnamen α zum Ausgangskanal mit Bitnamen β betrachtet der die Nachricht erhaltende Schalter auf Stufe k , $k = 0, \dots, \log n - 1$, das k -te Bit β_k (von links) des Zielnamens β und wählt den Ausgang für das Weitersenden anhand folgender Regel aus:

1. Ist das k -te Bit $\beta_k = 0$, so wird die Nachricht über den oberen Ausgang des Schalters weitergeleitet.
2. Ist das k -te Bit $\beta_k = 1$, so wird die Nachricht über den unteren Ausgang des Schalters weitergeleitet.

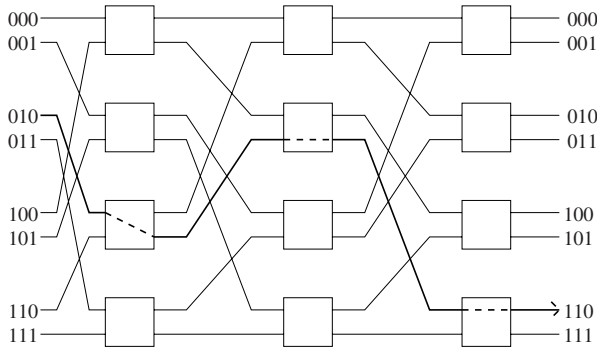


Abb. 2.22 8×8 Omega-Netzwerk mit Pfad von 010 nach 110 [11]

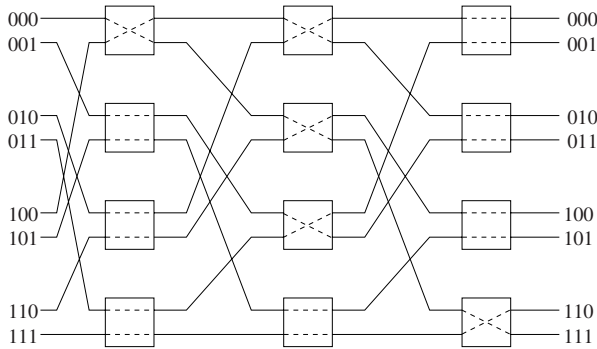


Abb. 2.23 8×8 Omega-Netzwerk mit Schalterstellungen zur Realisierung von π^8 aus dem Text

In Abb. 2.22 ist der Pfad der Nachrichtenübertragung vom Eingang $\alpha = 010$ zum Ausgang $\beta = 110$ angegeben. Maximal können bis zu n Nachrichten von verschiedenen Eingängen zu verschiedenen Ausgängen parallel zueinander durch das Omega-Netzwerk geschickt werden. Ein Beispiel für eine parallele Nachrichtenübertragung mit $n = 8$ im 8×8 -Omega-Netzwerk ist durch die Permutation

$$\pi^8 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 3 & 0 & 1 & 2 & 5 & 4 & 6 \end{pmatrix}$$

gegeben, die angibt, dass von Eingang i ($i = 0, \dots, 7$) zum Ausgang $\pi^8(i)$ jeweils eine Nachricht gesendet wird. Die entsprechende parallele Schaltung der 8 Pfade, jeweils von i nach $\pi^8(i)$, ist durch die Schaltereinstellung in Abb. 2.23 realisiert.

Viele solcher durch Permutation $\pi^8 : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ gegebener gewünschter Verbindungen sind jedoch nicht in einem Schritt, also parallel zueinander zu realisieren, da es zu **Konflikten** im Netzwerk kommt. So führen zum Beispiel die beiden Nachrichtenübersendungen von $\alpha_1 = 010$ zu $\beta_1 = 110$

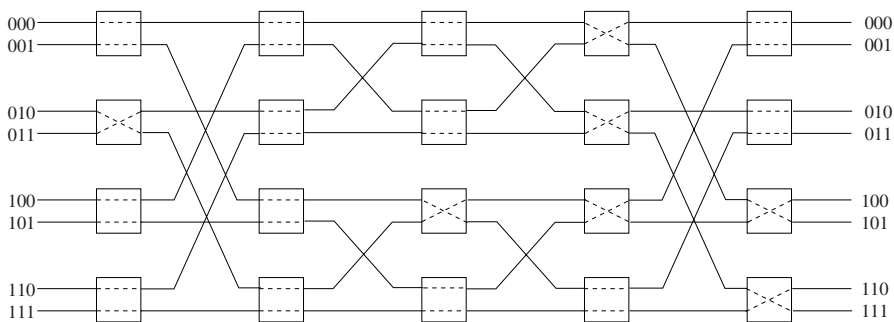


Abb. 2.24 8×8 Benes-Netzwerk mit Schalterstellungen zur Realisierung von π^8 aus dem Text

und von $\alpha_2 = 000$ zu $\beta_2 = 111$ in einem 8×8 Omega-Netzwerk zu einem Konflikt. Konflikte dieser Art können nicht aufgelöst werden, da es zu einem beliebigen Paar (α, β) von Eingabekante und Ausgabekante jeweils nur genau eine mögliche Verbindung gibt und somit kein Ausweichen möglich ist. Netzwerke mit dieser Eigenschaft heißen auch **blockierende Netzwerke**. Konflikte in blockierenden Netzwerken können jedoch durch mehrere Läufe durch das Netzwerk aufgelöst werden. Von den insgesamt $n!$ möglichen Permutationen (bzw. denen durch sie jeweils dargestellten gewünschten n Verbindungen von Eingangskanälen zu Ausgangskanälen) können nur $n^{n/2}$ in einem Durchlauf parallel zueinander, also ohne Konflikte realisiert werden. Denn da es pro Schalter 2 mögliche Schalterstellungen gibt, ergibt sich für die insgesamt $n/2 \cdot \log n$ Schalter des Omega-Netzwerkes eine Anzahl von $2^{n/2 \cdot \log n} = n^{n/2}$ mögliche Schaltungen des Gesamtnetzwerkes, die jeweils einer Realisierung von n parallelen Pfaden entsprechen.

Weitere blockierende Netzwerke sind das Butterfly- oder Banyan-Netzwerk, das Baseline-Netzwerk und das Delta-Netzwerk [111]. Im Unterschied dazu handelt es sich beim Benes-Netzwerk um ein *nicht-blockierendes* Netzwerk, das es ermöglicht, unterschiedliche Verbindungen zwischen einer Eingangskante und einer Ausgangskante herzustellen. Für jede Permutation $\pi : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ ist es möglich, eine Schaltung des Benes-Netzwerkes zu finden, die Verbindungen von Eingang i zu Ausgang $\pi(i)$, $i = 0, \dots, n-1$, gleichzeitig realisiert, so dass die n Kommunikationen parallel zueinander stattfinden können. Dies kann durch Induktion über die Dimension k des Netzwerkes bewiesen werden, vgl. dazu [111]. Ein Beispiel für die Realisierung der Permutation

$$\pi^8 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 5 & 3 & 4 & 7 & 0 & 1 & 2 & 6 \end{pmatrix}$$

ist in Abb. 2.24 gegeben, vgl. [111]. Weitere Details über Routingtechniken in indirekten Netzwerken sind vor allem in [111] zu finden.

2.6.2 Switching

Eine **Switching-Strategie** oder Switching-Technik legt fest, *wie* eine Nachricht den vom Routingalgorithmus ausgewählten Pfad von einem Sendeknoten zum Zielknoten durchläuft. Genauer gesagt, wird durch eine Switching-Strategie festgelegt

- *ob und wie* eine Nachricht in Stücke, z. B. in Pakete oder *flits* (für engl. *flow control units*), zerlegt wird,
- *wie* der Übertragungspfad vom Sendeknoten zum Zielknoten allokiert wird (vollständig oder teilweise) und
- *wie* Nachrichten (oder Teilstücke von Nachrichten) vom Eingabekanal eines Schalters oder Routers auf den Ausgabekanal gelegt werden. Der Routingalgorithmus legt dann fest, *welcher* Ausgabekanal zu wählen ist.

Die benutzte Switching-Strategie hat einen großen Einfluss auf die Zeit, die für eine Nachrichtenübertragung zwischen zwei Knoten benötigt wird. Bevor wir auf Switching-Strategien und den jeweils benötigten Zeitaufwand eingehen, betrachten wir zunächst den Zeitaufwand, der für eine Nachrichtenübertragung zwischen zwei benachbarten Netzwerknoten benötigt wird, wenn die Nachrichtenübertragung also über nur *eine* Verbindungsleitung erfolgt.

Nachrichtenübertragung benachbarter Prozessoren

Eine Nachrichtenübertragung zwischen zwei Prozessoren wird durch eine Folge von in Software realisierten Schritten (Protokoll genannt) realisiert. Sind die beiden Prozessoren durch eine bidirektionale Verbindungsleitung miteinander verbunden, so kann das im Folgenden skizzierte Beispielprotokoll verwendet werden. Zum Senden einer Nachricht werden vom sendenden Prozessor folgende Programmschritte ausgeführt:

1. Die Nachricht wird in einen Systempuffer kopiert.
2. Das Betriebssystem berechnet eine **Prüfsumme** (engl. *checksum*), fügt einen **Header** mit dieser Prüfsumme und Informationen zur Nachrichtenübertragung an die Nachricht an und startet einen Timer, der die Zeit misst, die die Nachricht bereits unterwegs ist.
3. Das Betriebssystem sendet die Nachricht zur Netzwerkschnittstelle und veranlasst die hardwaremäßige Übertragung.

Zum Empfangen einer Nachricht werden folgende Programmschritte ausgeführt:

1. Das Betriebssystem kopiert die Nachricht aus der Hardwareschnittstelle zum Netzwerk in einen Systempuffer.
2. Das Betriebssystem berechnet die Prüfsumme der erhaltenen Daten. Stimmt diese mit der beigefügten Prüfsumme überein, sendet der Empfänger eine Empfangsbestätigung (engl. *acknowledgement*) zum Sender. Stimmt die Prüfsumme

nicht mit der beigefügten Prüfsumme überein, so wird die Nachricht verworfen und es wird angenommen, dass der Sender nach Ablauf einer dem Timer vorgegebenen Zeit die Nachricht nochmals sendet.

3. War die Prüfsumme korrekt, so wird die Nachricht vom Systempuffer in den Adressbereich des Anwendungsprogramms kopiert und dem Anwendungsprogramm wird ein Signal zum Fortfahren gegeben.

Nach dem eigentlichen Senden der Nachricht werden vom sendenden Prozessor folgende weitere Schritte ausgeführt:

1. Bekommt der Sender die Empfangsbestätigung, so wird der Systempuffer mit der Kopie der Nachricht freigegeben.
2. Bekommt der Sender vom Timer die Information, dass die Schranke der Übertragungszeit überschritten wurde, so wird die Nachricht erneut gesendet.

In diesem Protokoll wurde angenommen, dass das Betriebssystem die Nachricht im Systempuffer hält, um sie gegebenenfalls neu zu senden. Wird keine Empfangsbestätigung benötigt, kann ein Sender jedoch erneut eine weitere Nachricht versenden, ohne auf die Ankunft der zuvor gesendeten Nachricht beim Empfänger zu warten. In Protokollen können außer der Zuverlässigkeit in Form der Empfangsbestätigung auch weitere Aspekte berücksichtigt werden, wie etwa die Umkehrung von Bytes beim Versenden zwischen verschiedenartigen Knoten, das Verhindern einer Duplizierung von Nachrichten oder das Füllen des Empfangspuffers für Nachrichten. Das Beispielprotokoll ist ähnlich zum weit verbreiteten UDP-Transportprotokoll [108, 141].

Die Zeit für eine Nachrichtenübertragung setzt sich aus der Zeit für die eigentliche Übertragung der Nachricht über die Verbindungsleitung, also die Zeit im Netzwerk, und der Zeit zusammen, die die Softwareschritte des jeweils verwendeten Protokolls benötigen. Zur Beschreibung dieser Zeit für eine Nachrichtenübertragung, die auch **Latenz** genannt wird, werden die folgenden Maße verwendet:

- Die **Bandbreite** (engl. *bandwidth*) ist die maximale Frequenz, mit der Daten über eine Verbindungsleitung geschickt werden können. Die Einheit ist Bytes/Sekunde.
- Die **Bytetransferzeit** ist die Zeit, die benötigt wird, um ein Byte über die Verbindungsleitung zu schicken. Es gilt:

$$\text{Bytetransferzeit} = \frac{1}{\text{Bandbreite}} .$$

- Die **Übertragungszeit** (engl. *transmission time*) ist die Zeit, die gebraucht wird, um eine Nachricht über eine Verbindungsleitung zu schicken. Es gilt:

$$\text{Übertragungszeit} = \frac{\text{Nachrichtengröße}}{\text{Bandbreite}} .$$

- Die **Signalverzögerungszeit** (engl. *time of flight* oder *channel propagation delay*) bezeichnet die Zeit, die das erste Bit einer Nachricht benötigt, um beim Empfänger anzukommen.
- Die **Transportlatenz** ist die Zeit, die eine Nachricht für die Übertragung im Netzwerk verbringt. Es gilt:

$$\text{Transportlatenz} = \text{Signalverzögerungszeit} + \text{Übertragungszeit} .$$

- Der **Senderoverhead** oder **Startupzeit** ist die Zeit, die der Sender benötigt, um eine Nachricht zum Senden vorzubereiten, umfasst also das Anfügen von Header und Prüfsumme und die Ausführung des Routingalgorithmus.
- Der **Empfängeroverhead** ist die Zeit, die der Empfänger benötigt, um die Softwareschritte für das Empfangen einer Nachricht auszuführen.
- Der **Durchsatz** (engl. *throughput*) wird zur Bezeichnung der Netzwerkbandbreite genutzt, die bei einer bestimmten Anwendung erzielt wird.

Unter Benutzung der obigen Maße setzt sich die gesamte Latenz der Übertragung einer Nachricht folgendermaßen zusammen:

$$\text{Latenz} = \text{Senderoverhead} + \text{Signalverzögerung} + \frac{\text{Nachrichtengröße}}{\text{Bandbreite}} + \text{Empfängeroverhead} . \quad (2.1)$$

In einer solchen Formel wird nicht berücksichtigt, dass eine Nachricht evtl. mehrmals verschickt wird bzw. ob Contention oder Congestion im Netzwerk vorliegt. Die Leistungsparameter für eine Nachrichtenübermittlung auf einer Verbindungsleitung sind aus der Sicht des Senders, des Empfängers und des Netzwerkes in Abb. 2.25 illustriert. Die Formel (2.1) kann vereinfacht werden, indem die konstanten Terme zusammengefasst werden. Es ergibt sich:

$$\text{Latenz} = \text{Overhead} + \frac{\text{Nachrichtengröße}}{\text{Bandbreite}}$$

mit einem konstanten Anteil Overhead und einem in der Nachrichtengröße linearen Anteil mit Faktor $\frac{1}{\text{Bandbreite}}$. Mit den Abkürzungen m für die Nachrichtengröße in Bytes, t_S für die den Overhead beschreibende Startupzeit und t_B für die Bytetransferzeit ergibt sich für die Latenz $T(m)$ in Abhängigkeit von der Nachrichtengröße m die Laufzeitformel

$$T(m) = t_S + t_B \cdot m . \quad (2.2)$$

Diese lineare Beschreibung des zeitlichen Aufwandes gilt für eine Nachrichtenübertragung zwischen zwei durch eine Punkt-zu-Punkt-Verbindung miteinander verbundene Knoten. Liegen zwei Knoten im Netzwerk *nicht* benachbart, so muss eine Nachricht zwischen den beiden Knoten über mehrere Verbindungsleitungen eines Pfades zwischen diesen beiden Knoten geschickt werden. Wie oben bereits erwähnt, kann dies durch verschiedene Switching-Strategien realisiert werden. Bei den Switching-Strategien werden u. a.:

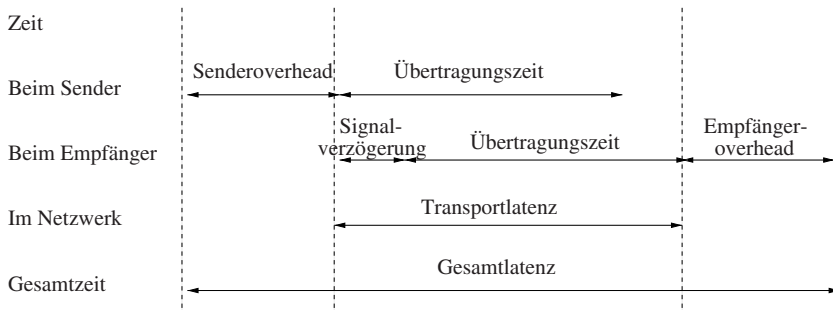


Abb. 2.25 Illustration zu Leistungsmaßen des Einzeltransfers zwischen benachbarten Knoten, siehe [75]

- Circuit-Switching
- Paket-Switching mit Store-und-Forward-Routing
- Virtuelles Cut-Through Routing und
- Wormhole Routing

unterschieden. Als Grundformen der Switching-Strategien kann man *Circuit-Switching* und *Paket-Switching* (engl. *packet switching*) ansehen [32, 124, 164].

Beim **Circuit-Switching** wird der gesamte Pfad vom Ausgangsknoten bis zum Zielknoten aufgebaut, d. h. die auf dem Pfad liegenden Switches, Prozessoren oder Router werden entsprechend geschaltet und exklusiv der zu übermittelnden Nachricht zur Verfügung gestellt, bis die Nachricht vollständig beim Zielknoten angekommen ist. Intern kann die Nachricht entsprechend der Übertragungsrate in Teilstücke unterteilt werden, und zwar in sogenannte **phits (physical units)**, die die Datenmenge bezeichnen, die pro Takt über eine Verbindung übertragen werden kann, bzw. die kleinste physikalische Einheit, die zusammen übertragen wird. Die Größe der *phits* wird im Wesentlichen durch die Anzahl der Bits bestimmt, die über einen physikalischen Kanal gleichzeitig übertragen werden können, und liegt typischerweise zwischen 1 und 64 Bits. Der Übertragungspfad wird durch das Versenden einer sehr kurzen Nachricht (*probe*) aufgebaut. Danach werden alle *phits* der Nachricht über diesen Pfad übertragen. Die Freigabe des Pfades geschieht entweder durch das Endstück der Nachricht oder durch eine zurückgesendete Empfangsbestätigung.

Die Kosten für das Versenden der Kontrollnachricht zum Aufbau des Pfades der Länge l vom Sender zum Empfänger benötigt die Zeit $t_c \cdot l$, wobei t_c die Kosten zum Versenden der Kontrollnachricht je Verbindung sind, d. h. $t_c = t_B \cdot m_c$ mit m_c = Größe des Kontrollpaketes. Nach der Reservierung des Pfades braucht die Versendung der eigentlichen Nachricht der Größe m die Zeit $m \cdot t_B$, so dass die Gesamtkosten des Einzeltransfers einer Nachricht auf einem Pfad der Länge l mit Circuit-Switching

$$T_{cs}(m, l) = t_S + t_c \cdot l + t_B \cdot m \quad (2.3)$$

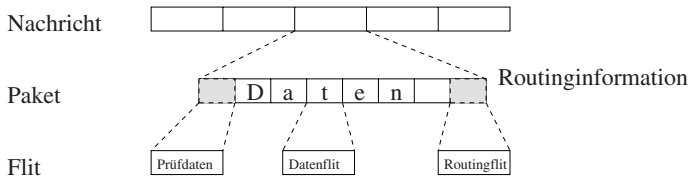


Abb. 2.26 Illustration zur Zerlegung einer Nachricht in Pakete und von Paketen in *flits* (flow control units)

sind. Ist m_c klein gegenüber m , so entspricht dies ungefähr $t_S + t_B \cdot m$, also einer Laufzeitformel, die linear in m und unabhängig von l ist. Die Kosten für einen Einzeltransfer mit Circuit-Switching sind in Abb. 2.27a illustriert.

Bei **Paket-Switching** wird eine Nachricht in eine Folge von Paketen unterteilt, die unabhängig voneinander über das Netzwerk vom Sender zum Empfänger transportiert werden. Bei Verwendung eines adaptiven Routing-Algorithmus können die Pakete einer Nachricht daher über unterschiedliche Pfade transportiert werden. Jedes **Paket** besteht aus drei Teilen, und zwar einem Header, der Routing- und Kontrollinformationen enthält, einem Datenteil, der einen Anteil der Gesamtnachricht enthält und einem Endstück (engl. *trailer*), das typischerweise den Fehlerkontrollcode enthält, siehe Abb. 2.26. Jedes Paket wird einzeln entsprechend der enthaltenen Routing- oder Zielinformation zum Ziel geschickt. Die Verbindungsleitungen oder Puffer werden jeweils nur von einem Paket belegt.

Die Paket-Switching-Strategie kann in verschiedenen Varianten realisiert werden. Paket-Switching mit **Store-and-Forward-Routing** versendet ein gesamtes Paket über je eine Verbindung auf dem für das Paket ausgewählten Pfad zum Empfänger. Jeder Zwischenempfänger, d. h. jeder Knoten auf dem Pfad speichert das gesamte Paket (*store*) bevor es weitergeschickt wird (*forward*). Die Verbindung zwischen zwei Knoten wird freigegeben, sobald das Paket beim Zwischenempfänger zwischengespeichert wurde. Diese Switching-Strategie wurde für frühe Parallelrechner verwendet und wird teilweise noch von Routern für IP-Pakete in WANs (*wide area networks*) benutzt. Ein Vorteil der Store-and-Forward-Strategie ist eine schnelle Freigabe von Verbindungen, was die Deadlockgefahr verringert und mehr Flexibilität bei hoher Netzbelastung erlaubt. Nachteile sind der hohe Speicherbedarf für die Zwischenpufferung von Paketen sowie eine Kommunikationszeit, die von der Länge der Pfade abhängt, was je nach Topologie und Kommunikationsanforderung zu hohen Kommunikationszeiten führen kann.

Die Kosten zum Versenden eines Paketes über eine Verbindung sind $t_h + t_B \cdot m$, wobei m die Größe des Paketes ist und t_h die konstante Zeit bezeichnet, die an einem Zwischenknoten auf dem Pfad zum Ziel benötigt wird, um z. B. das Paket im Eingangspuffer abzulegen und durch Untersuchung des Headers den nächsten Ausgabekanal auszuwählen. Die Gesamtkosten des Transfers eines Paketes bei Paket-Switching mit Store-and-Forward-Routing auf einem Pfad der Länge l betra-

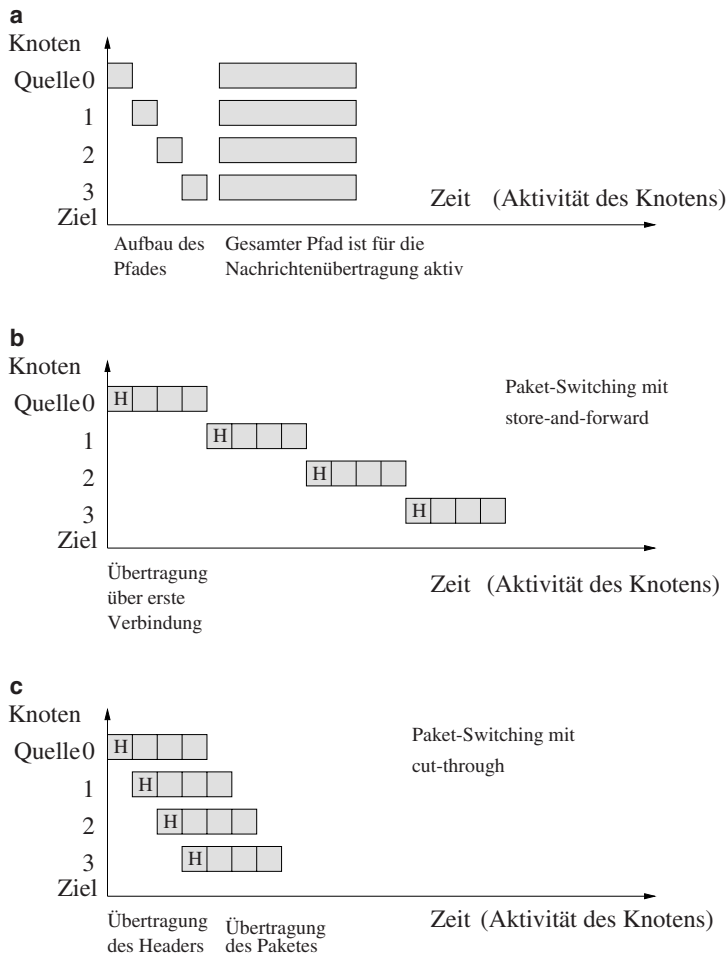


Abb. 2.27 Illustration zur Latenzzeit einer Einzeltransferoperation, über einen Pfad der Länge $l = 4$, **a** Circuit-Switching, **b** Paket-Switching mit store-and-forward und **c** Paket-Switching mit cut-through

gen damit

$$T_{sf}(m, l) = t_S + l(t_h + t_B \cdot m) . \quad (2.4)$$

Da t_h im Vergleich zu den anderen Größen üblicherweise recht klein ist, ist $T_{sf}(m, l) \approx t_S + l \cdot t_B \cdot m$. Die Kosten für die Zustellung eines Paketes hängen also vom Produkt der Nachrichtengröße m und der Pfadlänge l ab. Eine Illustration der Kosten für einen Einzeltransfer für Paket-Switching mit Store-and-Forward-Routing findet man in Abb. 2.27b. Die Kosten für den Transport einer aus mehreren Paketen bestehenden Nachricht vom Sendeknoten zum Empfängsknoten hängen

vom verwendeten Routingverfahren ab. Für ein deterministisches Routingverfahren ergibt sich die Transportzeit als die Summe der Transportkosten der einzelnen Pakete, wenn es nicht zu Verzögerungen im Netzwerk kommt. Für adaptive Routingverfahren können sich die Transportzeiten der einzelnen Pakete überlappen, so dass eine geringere Gesamtzeit resultieren kann.

Wenn alle Pakete einer Nachricht den gleichen Übertragungspfad verwenden können, kann die Einführung von **Pipelining** zur Verringerung der Kommunikationszeiten beitragen. Dazu werden die Pakete einer Nachricht so durch das Netzwerk geschickt, dass die Verbindungen auf dem Pfad von aufeinanderfolgenden Paketen überlappend genutzt werden. Ein derartiger Ansatz wird z. T. in softwarerealisierten Datenkommunikationen in Kommunikationsnetzwerken wie dem Internet benutzt. Bei Pipelining einer Nachricht der Größe m und Paketgröße m_p ergeben sich die Kosten

$$t_S + (m - m_p)t_B + l(t_h + t_B \cdot m_p) \approx t_S + m \cdot t_B + (l - 1)t_B \cdot m_p. \quad (2.5)$$

Dabei ist $l(t_h + t_B \cdot m_p)$ die Zeit, die bis zur Ankunft des ersten Paketes vergeht. Danach kommt in jedem Zeitschritt der Größe $m_p \cdot t_B$ ein weiteres Paket an.

Der Ansatz des gepipelineten Paket-Switching kann mit Hilfe von **Cut-Through-Routing** noch weitergetrieben werden. Die Nachricht wird entsprechend des Paket-Switching-Ansatzes in Pakete unterteilt und jedes einzelne Paket wird pipelineartig durch das Netzwerk geschickt. Die verschiedenen Pakete einer Nachricht können dabei prinzipiell verschiedene Übertragungspfade verwenden. Beim Cut-Through-Routing betrachtet ein auf dem Übertragungspfad liegender Schalter (bzw. Knoten oder Router) die ersten *phits* (*physical units*) des ankommenden Paketes, die den Header mit der Routinginformation enthalten, und trifft daraufhin die Entscheidung, zu welchem Knoten das Paket weitergeleitet wird. Der Verbindungspfad wird also vom Header eines Paketes aufgebaut. Ist die gewünschte Verbindung frei, so wird der Header weitergeschickt und der Rest des Paketes wird direkt hinterhergeleitet, so dass die *phits* des Paketes pipelineartig auf dem Übertragungspfad verteilt sind. Verbindungen, über die alle *phits* des Paketes einschließlich Endstück vollständig übertragen wurden, werden freigegeben. Je nach Situation im Netzwerk kann also der gesamte Pfad vom Ausgangsknoten bis zum Zielknoten der Übertragung eines Paketes zugeordnet sein.

Sind die Kosten zur Übertragung des Headers auf einer Verbindungsleitung durch t_H gegeben, d. h. $t_H = t_B \cdot m_H$, wobei m_H die Größe des Headers ist, so sind die Kosten zur Übertragung des Headers auf dem gesamtem Pfad der Länge l durch $t_H \cdot l$ gegeben. Die Zeit bis zur Ankunft des Paketes der Größe m am Zielknoten nach Ankunft des Headers beträgt $t_B \cdot (m - m_H)$. Die Kosten für den Transport eines Paketes betragen bei Verwendung von Paket-Switching mit Cut-Through-Routing auf einem Pfad der Länge l ohne Contention

$$T_{ct}(m, l) = t_S + l \cdot t_H + t_B \cdot (m - m_H). \quad (2.6)$$

Ist m_H im Vergleich zu der Gesamtgröße m der Nachricht klein, so entspricht dies ungefähr den Kosten $T_{ct}(m, l) \approx t_S + t_B \cdot m$. Verwenden alle Pakete einer Nachricht den gleichen Übertragungspfad und werden die Pakete ebenfalls nach dem Pipelining-Prinzip übertragen, gilt diese Formel auch für die Übertragung einer gesamten Nachricht der Größe m . Die Kosten für den Transport eines Paketes für Paket-Switching mit Cut-Through-Routing sind in Abb. 2.27c illustriert.

Sind außer dem zu übertragenden Paket noch andere Pakete im Netzwerk, so muss Contention, also die Anforderungen einer Verbindung durch mehrere Nachrichten berücksichtigt werden. Ist die nächste gewünschte Verbindung *nicht* frei, so werden bei **virtuellem Cut-Through-Routing** alle *phits* des Paketes im letzten erreichten Knoten aufgesammelt und dort zwischengepuffert. Geschieht dies an jedem Knoten, so kann Cut-Through-Routing zu Store-and-Forward-Routing degenerieren. Bei **partielltem Cut-Through-Routing** können Teile des Paketes weiter übertragen werden, falls die gewünschte Verbindung frei wird, bevor alle *phits* des Paketes in einem Knoten auf dem Pfad zwischengepuffert werden.

Viele derzeitige Parallelrechner benutzen eine Variante des Cut-Through-Routing, die **Wormhole-Routing** oder manchmal auch Hardware-Routing genannt wird, da es durch die Einführung von Routern eine hardwaremäßige Unterstützung gibt, vgl. Abschn. 2.4.1. Beim Wormhole-Routing werden die Pakete in kleine Einheiten zerlegt, die *flits* (für engl. *flow control units*) genannt werden und deren Größe typischerweise zwischen 1 und 8 Bytes liegt. Die Header-*flits* bestimmen den Weg durch das Netzwerk. Alle anderen *flits* des Paketes folgen pipelinemäßig auf demselben Pfad. Die Zwischenpuffer an den Knoten bzw. den Ein- und/oder Ausgabekanälen der Knoten sind nur für wenige *flits* ausgelegt. Ist eine gewünschte Verbindung nicht frei, so blockiert der Header bis die Verbindung frei wird. Alle nachfolgenden *flits* werden ebenfalls blockiert und verbleiben in ihrer Position. Im Gegensatz zur oben beschriebenen Form des Cut-Through-Routing werden *flits* also *nicht* bis zur den Header blockierenden Stelle nachgeholt, sondern blockieren einen gesamten Pfad. Dadurch gleicht dieser Ansatz eher dem Circuit-Switching auf Paketebene. Ein Vorteil von Wormhole-Routing ist der geringe Speicherplatzbedarf für die Zwischenspeicherung. Durch die Blockierung ganzer Pfade erhöht sich jedoch wieder die Deadlockgefahr durch zyklisches Warten, vgl. Abb. 2.28 [124].

Die Deadlockgefahr kann durch geeignete Routing-Algorithmen, z. B. dimensionsgeordnetes Routing, oder die Verwendung virtueller Kanäle vermieden werden.

2.6.3 Flusskontrollmechanismen

Flusskontrollmechanismen (engl. *flow control mechanism*) werden benötigt, wenn sich mehrere Nachrichten im Netzwerk befinden und geregelt werden muss, welchem Paket eine Verbindung oder ein Puffer zur Verfügung gestellt wird. Sind angeforderte Ressourcen bereits von anderen Nachrichten oder Nachrichtenteilen belegt, so entscheidet ein Flusskontrollmechanismus, ob die Nachricht blockiert

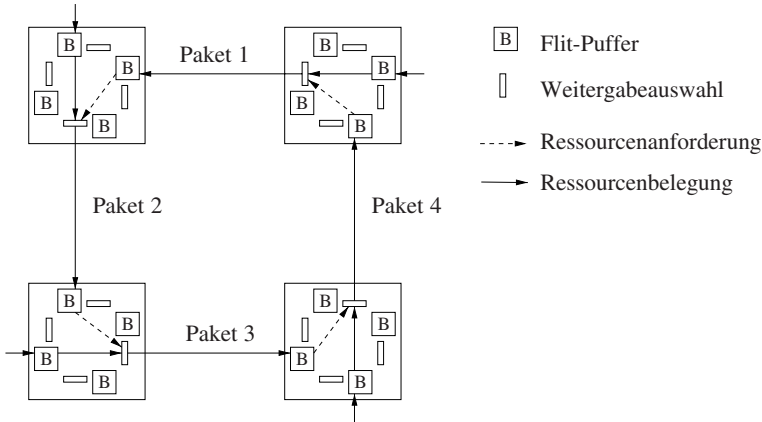


Abb. 2.28 Illustration zur Deadlock-Gefahr beim Wormhole-Routing von vier Paketen über vier Router. Jedes der vier Pakete belegt einen Flit-Puffer und fordert einen Flit-Puffer an, der von einem anderen Paket belegt ist. Es resultiert eine Deadlock-Situation, da keines der Pakete weitergeschickt werden kann

wird, wo sie sich befindet, in einem Puffer zwischengespeichert wird, auf einem alternativen Pfad weitergeschickt wird oder einfach weggeworfen wird. Die minimale Einheit, die über eine Verbindung geschickt und akzeptiert bzw. wegen beschränkter Kapazität zurückgewiesen werden kann, wird *flit* (*flow control unit*) genannt. Ein flit kann einem phit entsprechen, aber auch einem ganzen Paket oder einer ganzen Nachricht.

Flusskontrollmechanismen müssen in jeder Art von Netzwerk vorhanden sein und spielen bei Transportprotokollen wie TCP eine wichtige Rolle, vgl. z. B. [108, 141]. Die Netzwerke von Parallelrechnern stellen an die Flusskontrolle aber die besonderen Anforderungen, dass sich sehr viele Nachrichten auf engem Raum konzentrieren und die Nachrichtenübertragung für die korrekte Abarbeitung der Gesamtaufgabe sehr zuverlässig sein muss. Weiter sollten Staus (engl. *congestion*) in den Kanälen vermieden werden und eine schnelle Nachrichtenübertragung gewährleistet sein.

Flusskontrolle bzgl. der Zuordnung von Paketen zu Verbindungen (engl. *link-level flow-control*) regelt die Datenübertragungen über eine Verbindung, also vom Ausgangskanal eines Knotens über die Verbindungsleitung zum Eingangskanal des zweiten Knotens, an dem typischerweise ein kleiner Speicher oder Puffer ankommende Daten aufnehmen kann. Ist dieser Eingangspuffer beim Empfänger gefüllt, so kann die Nachricht nicht angenommen werden und muss beim Sender verbleiben, bis der Puffer wieder Platz bietet. Dieses Zusammenspiel wird durch einen Informationsaustausch mit *Anfrage* des Senders (*request*) und *Bestätigung* des Empfängers (*acknowledgement*) durchgeführt (*request-acknowledgement handshake*). Der Sender sendet ein Anfrage-Signal, wenn er eine Nachricht senden möchte. Der

Empfänger sendet eine Bestätigung, falls die Daten empfangen wurden. Erst danach kann eine weitere Nachricht vom Sender losgeschickt werden.

2.7 Caches und Speicherhierarchien

Ein wesentliches Merkmal der Hardware-Entwicklung der letzten Jahrzehnte ist, wie bereits oben geschildert, das Auseinanderdriften von Prozessorgeschwindigkeit und Speicherzugriffsgeschwindigkeit, was durch ein vergleichsweise geringes Anwachsen der Zugriffsgeschwindigkeit auf DRAM-Chips begründet ist, die für die physikalische Realisierung von Hauptspeichern verwendet werden. Um jedoch trotzdem die Prozessorgeschwindigkeit effektiv nutzen zu können, wurden Speicherhierarchien eingeführt, die aus Speichern verschiedener Größen und Zugriffsgeschwindigkeiten bestehen und deren Ziel die Verringerung der *mittleren* Speicherzugriffszeiten ist. Die einfachste Form einer Speicherhierarchie ist das Einfügen eines einzelnen Caches zwischen Prozessor und Speicher (einstufiger Cache). Ein Cache ist ein relativ kleiner, schneller Speicher mit einer im Vergleich zum Hauptspeicher geringen Speicherzugriffszeit, die meist durch Verwendung von schnellen SRAM-Chips erreicht wird. In den Cache werden entsprechend einer vorgegebenen Nachladestrategie Daten des Hauptspeichers geladen mit dem Ziel, dass sich die zur Abarbeitung eines Programms benötigten Daten zum Zeitpunkt des Zugriffs in den meisten Fällen im Cache befinden.

Für Multiprozessoren mit lokalen Caches der einzelnen Prozessoren stellt sich die zusätzliche Aufgabe der konsistenten Aufrechterhaltung des gemeinsamen Adressraumes. Typisch sind mittlerweile zwei- oder dreistufige Cache-Speicher für jeden Prozessor. Da viele neuere Multiprozessoren zur Klasse der Rechner mit *virtuell* gemeinsamem Speicher gehören, ist zusätzlich auch der gemeinsame Speicher als Stufe der Speicherhierarchie anzusehen. Dieser Trend der Hardware-Entwicklung in Richtung des virtuell gemeinsamen Speichers ist durch die geringeren Hardwarekosten begründet, die Rechner mit verteiltem Speicher gegenüber Rechnern mit physikalisch gemeinsamem Speicher verursachen. Der gemeinsame Speicher wird softwaremäßig realisiert. Da Caches die grundlegenden Bausteine von Speicherhierarchien darstellen, deren Arbeitsweise die weiterführenden Fragen der Konsistenz des Speichersystems wesentlich beeinflusst, beginnen wir mit einem kurzen Abriss über Caches. Für eine ausführlichere Behandlung verweisen wir auf [32, 70, 75, 139].

2.7.1 Charakteristika von Cache-Speichern

Ein **Cache** ist ein kleiner schneller Speicher, der zwischen Hauptspeicher und Prozessor eingefügt wird. Caches werden häufig durch SRAM-Chips (*Static Random*

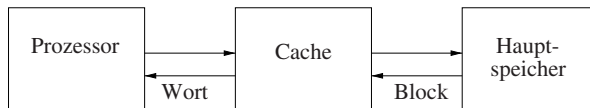


Abb. 2.29 Der Datentransport zwischen Cache und Hauptspeicher findet in Cacheblöcken statt, während der Prozessor auf einzelne Worte aus dem Cache zugreift

Access Memory) realisiert, deren Zugriffszeiten deutlich geringer sind als die von DRAM-Chips. Je nach verwendeter Speichertechnologie liegt die typische Zugriffszeit für DRAM-Chips, die für die Realisierung der Hauptspeicher verwendet werden, zwischen 20 und 70 ns. Damit braucht ein Speicherzugriff für einen Prozessor mit einer Taktrate von 3 GHz, was einer Zykluszeit von 0,33 ns entspricht, je nach verwendetem DRAM-Chip zwischen 60 und 200 Maschinenzyklen. Diese Anzahl kann durch den Einsatz von Caches zum Teil erheblich reduziert werden. Typische Cachezugriffszeiten liegen 2012 für einen L1-Cache im Bereich von 1 ns, für einen L2-Cache im Bereich von 3–10 ns und für einen L3-Cache im Bereich von 10–20 ns. Für einen Intel Core i7 Prozessor (Sandy Bridge) liegt z. B. die Zugriffszeit, ausgedrückt in Maschinenzyklen, für den L1-Cache bei 4–6 Maschinenzyklen, für den L2-Cache bei 12 Zyklen und für den L3-Cache bei 26–31 Zyklen, vgl. auch Abschn. 2.8.4. Demgegenüber steht bei Verwendung einer 3,3 GHz CPU und einem auf DDR1600 SDRAM basierenden Hauptspeicher unter Berücksichtigung der Cachefehlzugriffbehandlung eine Gesamtzugriffszeit von ca. 135 Maschinenzyklen [76]. Ein Zugriff auf die Festplatte braucht wegen der erforderlichen mechanischen Bewegung des Festplattenkopfes noch wesentlich länger: typischerweise liegt die Zugriffszeit auf eine Festplatte zwischen 10 und 20 ms, was für übliche CPU-Taktfrequenzen 30 Millionen bis 100 Millionen Maschinenzyklen entspricht. Dabei wird aber eine ganze Speicherseite geladen, deren Größe meist bei 4 KBytes oder 8 KBytes liegt.

Zur Vereinfachung gehen wir im Folgenden zuerst von einem einstufigen Cache aus. Der Cache enthält eine *Kopie* von Teilen der im Hauptspeicher abgelegten Daten. Diese Kopie wird in Form von **Cacheblöcken** (engl. *cache line*), die üblicherweise aus mehreren Worten bestehen, aus dem Speicher in den Cache geladen, vgl. Abb. 2.29. Die verwendete Blockgröße ist für einen Cache konstant und kann in der Regel während der Ausführung eines Programms nicht variiert werden.

Die Kontrolle des Caches ist vom Prozessor abgekoppelt und wird von einem eigenen Cache-Controller übernommen. Der Prozessor setzt entsprechend der Operanden der auszuführenden Maschinenbefehle Schreib- oder Leseoperationen an das Speichersystem ab und wartet gegebenenfalls, bis dieses die angeforderten Operanden zur Verfügung stellt. Die Architektur des Speichersystems hat in der Regel keinen Einfluss auf die vom Prozessor abgesetzten Zugriffsoperationen, d. h. der Prozessor braucht keine Kenntnis von der Architektur des Speichersystems zu haben. Nach Empfang einer Zugriffsoperation vom Prozessor überprüft der Cache-Controller eines einstufigen Caches, ob das zu lesende Wort im Cache gespeichert

ist (**Cachetreffer**, engl. *cache hit*). Wenn dies der Fall ist, wird es vom Cache-Controller aus dem Cache geladen und dem Prozessor zur Verfügung gestellt. Befindet sich das Wort *nicht* im Cache (**Cache-Fehlzugriff**, engl. *cache miss*), wird der Block, in dem sich das Wort befindet, vom Cache-Controller aus dem Hauptspeicher in den Cache geladen. Da der Hauptspeicher relativ hohe Zugriffszeiten hat, dauert das Laden des Blockes wesentlich länger als der Zugriff auf den Cache. Beim Auftreten eines Cache-Fehlzugriffs werden die vom Prozessor angeforderten Operanden also nur verzögert zur Verfügung gestellt. Bei der Abarbeitung eines Programms sollten daher möglichst wenige Cache-Fehlzugriffe auftreten.

Dem Prozessor bleibt die genaue Arbeitsweise des Cache-Controllers verborgen. Er beobachtet nur den Effekt, dass bestimmte Speicherzugriffe länger dauern als andere und er länger auf die angeforderten Operanden warten muss. Diese Entkopplung von Speicherzugriffen und der Ausführung von arithmetisch/logischen Operationen stellt sicher, dass der Prozessor während des Wartens auf die Operanden andere Berechnungen durchführen kann, die von den ausstehenden Operanden unabhängig sind. Dies wird durch die Verwendung mehrerer Funktionseinheiten und durch das Vorladen von Operanden (engl. *operand prefetch*) unterstützt, siehe Abschn. 2.2. Die beschriebene Entkopplung hat auch den Vorteil, dass Prozessor und Cache-Controller beliebig kombiniert werden können, d. h. ein Prozessor kann in unterschiedlichen Rechnern mit verschiedenen Speichersystemen kombiniert werden, ohne dass eine Adaption des Prozessors erforderlich wäre.

Wegen des beschriebenen Vorgehens beim Laden von Operanden hängt die Effizienz eines Programms wesentlich davon ab, ob viele oder wenige der vom Prozessor abgesetzten Speicherzugriffe vom Cache-Controller aus dem Cache bedient werden können. Wenn viele Speicherzugriffe zum Nachladen von Cacheblöcken führen, wird der Prozessor oft auf Operanden warten müssen und das Programm wird entsprechend langsam abgearbeitet. Da die Nachladestrategie des Caches von der Hardware vorgegeben ist, kann die Effektivität des Caches nur durch die Struktur des Programms beeinflusst werden. Dabei hat insbesondere das von einem gegebenen Programm verursachte Speicherzugriffsverhalten einen großen Einfluss auf seine Effizienz. Die für das Nachladen des Caches relevante Eigenschaft der Speicherzugriffe eines Programms versucht man mit dem Begriff der **Lokalität der Speicherzugriffe** zu fassen. Dabei unterscheidet man zwischen zeitlicher und räumlicher Lokalität:

- Die Speicherzugriffe eines Programms weisen eine hohe **räumliche Lokalität** auf, wenn zu aufeinanderfolgenden Zeitpunkten der Programmausführung auf im Hauptspeicher *räumlich benachbarte* Speicherzellen zugegriffen wird. Für ein Programm mit hoher räumlicher Lokalität tritt relativ oft der Effekt auf, dass nach dem Zugriff auf eine Speicherzelle unmittelbar nachfolgende Speicherzugriffe eine oder mehrere Speicherzellen desselben Cacheblockes adressieren. Nach dem Laden eines Cacheblockes in den Cache werden daher einige der folgenden Speicherzugriffe auf den gleichen Cacheblock zugreifen und es ist kein weiteres Nachladen erforderlich. Die Verwendung von Cacheblöcken, die meh-

rere Speicherzellen umfassen, basiert auf der Annahme, dass viele Programme eine hohe räumliche Lokalität aufweisen.

- Die Speicherzugriffe eines Programms weisen eine hohe **zeitliche Lokalität** auf, wenn auf *dieselbe* Speicherstelle zu *zeitlich* dicht aufeinanderfolgenden Zeitpunkten der Programmausführung zugegriffen wird. Für ein Programm mit hoher zeitlicher Lokalität tritt relativ oft der Effekt auf, dass nach dem Laden eines Cacheblockes in den Cache auf die einzelnen Speicherzellen dieses Cacheblockes mehrfach zugegriffen wird, bevor der Cacheblock wieder aus dem Cache entfernt wird.

Für ein Programm mit geringer räumlicher Lokalität besteht die Gefahr, dass nach dem Laden eines Cacheblockes nur auf eine seiner Speicherzellen zugegriffen wird, die anderen wurden also unnötigerweise geladen. Für ein Programm mit geringer zeitlicher Lokalität besteht die Gefahr, dass nach dem Laden eines Cacheblockes nur einmal auf eine Speicherzelle zugegriffen wird, bevor der Cacheblock wieder in den Hauptspeicher zurückgeschrieben wird. Verfahren zur Erhöhung der Lokalität der Speicherzugriffe eines Programms sind z. B. in [182] beschrieben.

Wir gehen im Folgenden auf wichtige Charakteristika von Caches näher ein. Wir untersuchen die Größe der Caches und der Cacheblöcke und deren Auswirkung auf das Nachladen von Cacheblöcken, die Abbildung von Speicherworten auf Positionen im Cache, Ersetzungsverfahren bei vollem Cache und Rückschreibestrategien bei Schreibzugriffen durch den Prozessor. Wir untersuchen auch den Einsatz von mehrstufigen Caches.

Cachegröße

Bei Verwendung der gleichen Technologie steigt die Zugriffszeit auf den Cache wegen der Steigerung der Komplexität der Adressierungsschaltung mit der Größe der Caches (leicht) an. Auf der anderen Seite erfordert ein großer Cache weniger Nachladeoperationen als ein kleiner Cache, weil mehr Speicherzellen im Cache abgelegt werden können. Die Größe eines Caches wird auch oft durch die zur Verfügung stehende Chipfläche begrenzt, insbesondere dann, wenn es sich um einen On-Chip-Cache handelt, d. h. wenn der Cache auf der Chipfläche des Prozessors untergebracht wird. Meistens liegt die Größe von Caches erster Stufe zwischen 8 K und 128 K Speicherworten, wobei ein Speicherwort je nach Rechner aus vier oder acht Bytes besteht.

Wie oben beschrieben, wird beim Auftreten eines Cache-Fehlzugriffs nicht nur das zugegriffene Speicherwort, sondern ein Block von Speicherworten in den Cache geladen. Für die Größe der Cacheblöcke müssen beim Design des Caches zwei Punkte beachtet werden: Zum einen verringert die Verwendung von größeren Blöcken die Anzahl der Blöcke, die in den Cache passen, die geladenen Blöcke werden also schneller ersetzt als bei der Verwendung von kleineren Blöcken. Zum anderen ist es sinnvoll, Blöcke mit mehr als einem Speicherwort zu verwenden, da der

Gesamttransfer eines Blockes mit x Worten zwischen Hauptspeicher und Cache schneller durchgeführt werden kann als x Einzeltransporte mit je einem Wort. In der Praxis wird die Größe der Cacheblöcke (engl. *cache line size*) für Caches erster Stufe meist auf vier oder acht Speicherworte festgelegt.

Abbildung von Speicherblöcken auf Cacheblöcke

Daten werden in Form von Blöcken einheitlicher Größe vom Hauptspeicher in sogenannte *Blockrahmen* gleicher Größe des Caches eingelagert. Da der Cache weniger Blöcke als der Hauptspeicher fasst, muss eine Abbildung zwischen Speicherblöcken und Cacheblöcken durchgeführt werden. Dazu können verschiedene Methoden verwendet werden, die die Organisation des Caches wesentlich festlegen und auch die Suche nach Cacheblöcken bestimmen. Dabei spielt der Begriff der **Cache-Assoziativität** eine große Rolle. Die Assoziativität eines Caches legt fest, in wie vielen Blockrahmen ein Speicherblock abgelegt werden kann. Es werden folgende Ansätze verwendet:

1. Bei direkt-abgebildeten Caches (engl. *direct-mapped cache*) kann jeder Speicherblock in genau einem Blockrahmen abgelegt werden.
2. Bei voll-assoziativen Caches (engl. *associative cache*) kann jeder Speicherblock in einem beliebigen Blockrahmen abgelegt werden.
3. Bei mengen-assoziativen Caches (engl. *set-associative cache*) kann jeder Speicherblock in einer festgelegten Anzahl von Blockrahmen abgelegt werden.

Alle drei Abbildungsmechanismen werden im Folgenden kurz vorgestellt. Dabei betrachten wir ein aus einem Hauptspeicher und einem Cache bestehendes Speichersystem. Wir nehmen an, dass der Hauptspeicher $n = 2^s$ Blöcke fasst, die wir mit B_j , $j = 0, \dots, n - 1$, bezeichnen. Die Anzahl der Blockrahmen \bar{B}_i im Cachespeicher sei $m = 2^r$. Jeder Speicherblock und Blockrahmen fasse $l = 2^w$ Speicherworte. Da jeder Blockrahmen des Caches zu einem bestimmten Zeitpunkt der Programmausführung verschiedene Speicherblöcke enthalten kann, muss zu jedem Blockrahmen eine Markierung (engl. *tag*) abgespeichert werden, die die Identifikation des abgelegten Speicherblockes erlaubt. Wie diese Markierung verwendet wird, hängt vom benutzten Abbildungsmechanismus ab und wird im Folgenden beschrieben. Als begleitendes Beispiel betrachten wir ein Speichersystem, dessen Cache 64 KBytes groß ist und der Cacheblöcke der Größe 4 Bytes verwendet. Der Cache fasst also $16\text{ K} = 2^{14}$ Blöcke mit je 4 Bytes, d. h. es ist $r = 14$ und $w = 2$. Der Hauptspeicher ist $16\text{ MBytes} = 2^{24}$ Bytes groß, d. h. es ist $s = 22$, wobei wir annehmen, dass die Speicherworte ein Byte umfassen.

1. **Direkt abgebildeter Cache:** Ein direkt-abgebildeter Cache stellt die einfachste Form der Cache-Organisation dar. Jeder Datenblock B_j des Hauptspeichers wird *genau einem* Blockrahmen \bar{B}_i des Caches zugeordnet, in den er bei Bedarf

eingelagert werden kann. Die Abbildungsvorschrift von Blöcken auf Blockrahmen ist z. B. gegeben durch:

$$B_j \text{ wird auf } \bar{B}_i \text{ abgebildet, falls } i = j \bmod m \text{ gilt.}$$

In jedem Blockrahmen können also $n/m = 2^{s-r}$ verschiedene Speicherblöcke abgelegt werden. Entsprechend der obigen Abbildung gilt folgende Zuordnung:

Blockrahmen	Speicherblock
0	$0, m, 2m, \dots, 2^s - m$
1	$1, m + 1, 2m + 1, \dots, 2^s - m + 1$
\vdots	\vdots
$m - 1$	$m - 1, 2m - 1, 3m - 1, \dots, 2^s - 1$

Der Zugriff des Prozessors auf ein Speicherwort erfolgt über dessen Speicheradresse, die sich aus einer Blockadresse und einer Wortadresse zusammensetzt. Die Blockadresse gibt die Adresse des Speicherblockes, der die angegebene Speicheradresse enthält, im Hauptspeicher an. Sie wird von den s signifikantesten, d. h. linken Bits der Speicheradresse gebildet. Die Wortadresse gibt die relative Adresse des angegebenen Speicherwortes bzgl. des Anfanges des zugehörigen Speicherblockes an. Sie wird von den w am wenigsten signifikanten, d. h. rechts liegenden Bits der Speicheradresse gebildet. Bei direkt-abgebildeten Caches identifizieren die r rechten Bits der Blockadresse denjenigen der $m = 2^r$ Blockrahmen, in den der entsprechende Speicherblock gemäß obiger Abbildung eingelagert werden kann. Die $s - r$ verbleibenden Bits können als Markierung (*tag*) interpretiert werden, die angibt, welcher Speicherblock aktuell in einem bestimmten Blockrahmen des Caches enthalten ist. In dem oben angegebenen Beispiel bestehen die Markierungen aus $s - r = 8$ Bits.

Der Speicherzugriff wird in Abb. 2.30a illustriert. Bei jedem Speicherzugriff wird zunächst der Blockrahmen, in dem der zugehörige Speicherblock abgelegt werden muss, durch die r rechten Bits der Blockadresse identifiziert. Anschließend wird die aktuelle Markierung dieses Blockrahmens, die zusammen mit der Position des Blockrahmens den aktuell abgelegten Speicherblock eindeutig identifiziert, mit den $s - r$ linken Bits der Blockadresse verglichen. Stimmen beide Markierungen überein, so handelt es sich um einen *Cachetreffer*, d. h. der zugehörige Speicherblock befindet sich im Cache und der Speicherzugriff kann aus dem Cache bedient werden. Wenn die Markierungen nicht übereinstimmen, muss der zugehörige Speicherblock in den Cache geladen werden, bevor der Speicherzugriff erfolgen kann.

Direkt abgebildete Caches sind zwar einfach zu realisieren, haben jedoch den Nachteil, dass jeder Speicherblock nur an einer Position im Cache abgelegt werden kann. Bei ungünstiger Speicheradressierung eines Programms besteht daher die Gefahr, dass zwei oft benutzte Speicherblöcke auf den gleichen Blockrahmen abgebildet sein können und so ständig zwischen Hauptspeicher und Cache

hin- und hergeschoben werden müssen. Dadurch kann die Laufzeit eines Programms erheblich erhöht werden.

2. **Voll-assoziativer Cache:** Beim voll-assoziativen Cache kann jeder Speicherblock in jedem beliebigen Blockrahmen des Caches abgelegt werden, wodurch der Nachteil des häufigen Ein- und Auslagerns von Blöcken behoben wird. Der Speicherzugriff auf ein Wort erfolgt wieder über die aus der Blockadresse (s linken Bits) und der Wortadresse (w rechten Bits) zusammengesetzten Speicheradresse. Als Markierung eines Blockrahmens im Cache muss nun jedoch die gesamte Blockadresse verwendet werden, da jeder Blockrahmen jeden Speicherblock enthalten kann. Bei einem Speicherzugriff müssen also die Markierungen *aller* Blockrahmen im Cache durchsucht werden, um festzustellen, ob sich der entsprechende Block im Cache befindet. Dies wird in Abb. 2.30b veranschaulicht. Der Vorteil von voll-assoziativen Caches liegt in der hohen Flexibilität beim Laden von Speicherblöcken. Der Nachteil liegt zum einen darin, dass die verwendeten Markierungen wesentlich mehr Bits beinhalten als bei direkt-abgebildeten Caches. Im oben eingeführten Beispiel bestehen die Markierungen aus 22 Bits, d. h. für jeden 32-Bit-Speicherblock muss eine 22-Bit-Markierung abgespeichert werden. Ein weiterer Nachteil liegt darin, dass bei jedem Speicherzugriff die Markierungen aller Blockrahmen untersucht werden müssen, was entweder eine sehr komplexe Schaltung erfordert oder zu Verzögerungen bei den Speicherzugriffen führt.
3. **Mengen-assoziativer Cache:** Der mengen-assoziative Cache stellt einen Kompromiss zwischen direkt-abgebildeten und voll assoziativen Caches dar. Der Cache wird in v Mengen S_0, \dots, S_{v-1} unterteilt, wobei jede Menge $k = m/v$ Blockrahmen des Caches enthält. Die Idee besteht darin, Speicherblöcke B_j für $j = 0, \dots, n-1$, nicht direkt auf Blockrahmen, sondern auf die eingeführten Mengen von Blockrahmen abzubilden. Innerhalb der zugeordneten Menge kann der Speicherblock beliebig positioniert werden, d. h. jeder Speicherblock kann in k verschiedenen Blockrahmen aufgehoben werden. Die Abbildungsvorschrift von Blöcken auf Mengen von Blockrahmen lautet:

B_j wird auf Menge S_i abgebildet, falls $i = j \bmod v$ gilt .

Der Speicherzugriff auf eine Speicheradresse (bestehend aus Blockadresse und Wortadresse) ist in Abb. 2.30c veranschaulicht. Die $d = \log v$ rechten Bits der Blockadresse geben die Menge S_i an, der der Speicherblock zugeordnet wird. Die linken $s - d$ Bits bilden die Markierung zur Identifikation der einzelnen Speicherblöcke in einer Menge. Beim Speicherzugriff wird zunächst die Menge im Cache identifiziert, der der zugehörige Speicherblock zugeordnet wird. Anschließend wird die Markierung des Speicherblockes mit den Markierungen der Blockrahmen *innerhalb* dieser Menge verglichen. Wenn die Markierung mit einer der Markierungen der Blockrahmen übereinstimmt, kann der Speicherzugriff über den Cache bedient werden, ansonsten muss der Speicherblock aus dem Hauptspeicher nachgeladen werden.

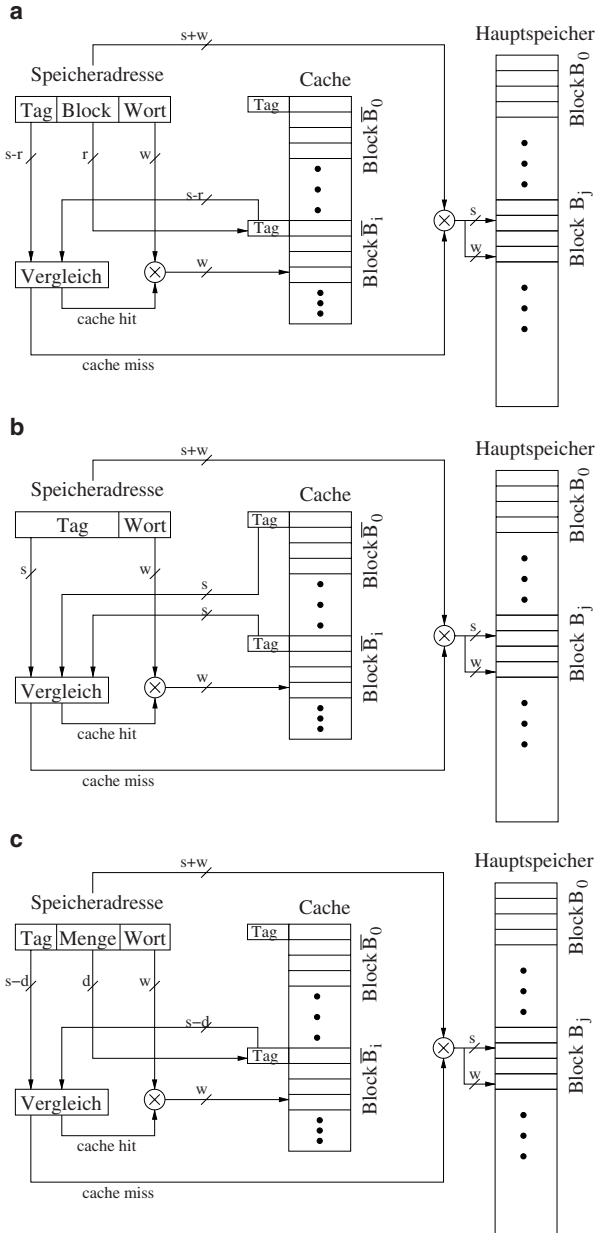


Abb. 2.30 Abbildungsmechanismen von Blöcken des Hauptspeichers auf Blockrahmen des Caches. **a** Direkt-abgebildeter Cache (oben), **b** voll-assoziativer Cache (Mitte), **c** mengen-assoziativer Cache (unten)

Für $v = m$ und $k = 1$ degeneriert der mengen-assoziative Cache zum direkt-abgebildeten Cache. Für $v = 1$ und $k = m$ ergibt sich der voll-assoziative Cache. Häufig verwendete Größen sind $v = m/4$ und $k = 4$ oder $v = m/8$ und $k = 8$. Im ersten Fall spricht man von einem *vier-Wege-assoziativen* Cache (engl. *4-way set-associative cache*), im zweiten Fall von einem *acht-Wege-assoziativen* Cache. Für $k = 4$ entstehen in unseren Beispiel 4K Mengen, für deren Identifikation $d = 12$ Bits verwendet werden. Für die Identifikation der in einer Menge abgelegten Speicherblöcke werden Markierungen mit 10 Bits verwendet.

Blockersetzungsmethoden

Soll ein neuer Speicherblock in den Cache geladen werden, muss evtl. ein anderer Speicherblock aus dem Cache entfernt werden. Für direkt-abgebildete Caches gibt es dabei wie oben beschrieben nur eine Möglichkeit. Bei voll-assoziativen und mengen-assoziativen Caches kann der zu ladende Speicherblock in mehreren Blockrahmen gespeichert werden, d. h. es gibt mehrere Blöcke, die ausgelagert werden könnten. Die Auswahl des auszulagernden Blockes wird gemäß einer Ersetzungsmethode vorgenommen. Die *LRU-Ersetzungsmethode* (*Least-recently-used*) entfernt den Block aus der entsprechenden Blockmenge, der am längsten unreferenziert ist. Zur Realisierung dieser Methode muss im allgemeinen Fall für jeden in einem Blockrahmen abgelegten Speicherblock der Zeitpunkt der letzten Benutzung abgespeichert und bei jedem Zugriff auf diesen Block aktualisiert werden. Dies erfordert zusätzlichen Speicherplatz zur Ablage der Benutzungszeitpunkte und zusätzliche Kontrolllogik zur Verwaltung und Verwendung dieser Benutzungszeitpunkte. Für zwei-Wege-assoziative Caches kann die LRU-Methode jedoch einfacher realisiert werden, indem jeder Blockrahmen jeder (in diesem Fall zweielementigen) Menge ein USE-Bit erhält, das wie folgt verwaltet wird: Wenn auf eine in dem Blockrahmen abgelegte Speicherzelle zugegriffen wird, wird das USE-Bit dieses Blockrahmens auf 1, das USE-Bit des anderen Blockrahmens der Menge auf 0 gesetzt. Dies geschieht bei jedem Speicherzugriff. Damit wurde auf den Blockrahmen, dessen USE-Bit auf 1 steht, zuletzt zugegriffen, d. h. wenn ein Blockrahmen entfernt werden soll, wird der Blockrahmen ausgewählt, dessen USE-Bit auf 0 steht.

Eine Alternative zur LRU-Ersetzungsmethode ist die *LFU-Ersetzungsmethode* (*Least-frequently-used*), die bei Bedarf den Block aus der Blockmenge entfernt, auf den am wenigsten oft zugegriffen wurde. Auch diese Ersetzungsmethode erfordert im allgemeinen Fall einen großen Mehraufwand, da zu jedem Block ein Zähler gehalten und bei jedem Speicherzugriff auf diesen Block aktualisiert werden muss. Eine weitere Alternative besteht darin, den zu ersetzenden Block zufällig auszuwählen. Diese Variante hat den Vorteil, dass kein zusätzlicher Verwaltungsaufwand notwendig ist.

Rückschreibestrategien

Bisher haben wir im Wesentlichen die Situation betrachtet, dass Daten aus dem Hauptspeicher gelesen werden und haben den Einsatz von Caches zur Verringerung der mittleren Zugriffszeit untersucht. Wir wenden uns jetzt der Frage zu, was passiert, wenn der Prozessor den Wert eines Speicherwortes, das im Cache aufgehoben wird, verändert, indem er eine entsprechende Schreiboperation an das Speichersystem weiterleitet. Das entsprechende Speicherwort wird auf jeden Fall im Cache aktualisiert, damit der Prozessor beim nächsten Lesezugriff auf dieses Speicherwort vom Speichersystem den aktuellen Wert erhält. Es stellt sich aber die Frage, wann die Kopie des Speicherwortes im Hauptspeicher aktualisiert wird. Diese Kopie kann frühestens nach der Aktualisierung im Cache und muss spätestens bei der Entfernung des entsprechenden Speicherblocks aus dem Cache aktualisiert werden. Der genaue Zeitpunkt und der Vorgang der Aktualisierung wird durch die *Rückschreibestrategie* festgelegt. Die beiden am häufigsten verwendeten Rückschreibestrategien sind die Write-through-Strategie und die Write-back-Strategie:

1. **Write-through-Rückschreibestrategie:** Wird ein im Cache befindlicher Speicherblock durch eine Schreiboperation modifiziert, so wird neben dem Eintrag im Cache auch der zugehörige Eintrag im Hauptspeicher aktualisiert, d. h. Schreiboperationen auf den Cache werden in den Hauptspeicher „durchgeschrieben“. Somit enthalten die Speicherblöcke im Cache und die zugehörigen Kopien im Hauptspeicher immer die gleichen Werte. Der Vorteil dieses Ansatzes liegt darin, dass I/O-Geräte, die direkt ohne Zutun des Prozessors auf den Hauptspeicher (DMA, *direct memory access*) zugreifen, stets die aktuellen Werte erhalten. Dieser Vorteil spielt auch bei Multiprozessoren eine große Rolle, da andere Prozessoren beim Zugriff auf den Hauptspeicher *immer* den aktuellen Wert erhalten. Der Nachteil des Ansatzes besteht darin, dass das Aktualisieren eines Wertes im Hauptspeicher im Vergleich zum Aktualisieren im Cache relativ lange braucht. Daher muss der Prozessor möglicherweise warten, bis der Wert zurückgeschrieben wurde (engl. *write stall*). Der Einsatz eines Schreibpuffers, in dem die in den Hauptspeicher zu transportierenden Daten zwischengespeichert werden, kann dieses Warten verhindern [75].
2. **Write-back-Rückschreibestrategie:** Eine Schreiboperation auf einen im Cache befindlichen Block wird zunächst *nur* im Cache durchgeführt, d. h. der zugehörige Eintrag im Hauptspeicher wird nicht sofort aktualisiert. Damit können Einträge im Cache aktuellere Werte haben als die zugehörigen Einträge im Hauptspeicher, d. h. die Werte im Hauptspeicher sind u. U. veraltet. Die Aktualisierung des Blocks im Hauptspeicher findet erst statt, wenn der Block im Cache durch einen anderen Block ersetzt wird. Um festzustellen, ob beim Ersetzen eines Cacheblockes ein Zurückschreiben notwendig ist, wird für jeden Cacheblock ein Bit (*dirty bit*) verwendet, das angibt, ob der Cacheblock seit dem Einlagern in den Cache modifiziert worden ist. Dieses Bit wird beim Laden eines Speicherblockes in den Cache mit 0 initialisiert. Bei der ersten Schreib-

operation auf eine Speicherzelle des Blockes wird das Bit auf 1 gesetzt. Bei dieser Strategie werden in der Regel weniger Schreiboperationen auf den Hauptspeicher durchgeführt, da Cacheeinträge mehrfach geschrieben werden können, bevor der zugehörige Speicherblock in den Hauptspeicher zurückgeschrieben wird. Der Hauptspeicher enthält aber evtl. ungültige Werte, so dass ein direkter Zugriff von I/O-Geräten nicht ohne weiteres möglich ist. Dieser Nachteil kann dadurch behoben werden, dass die von I/O-Geräten zugreifbaren Bereiche des Hauptspeichers mit einer besonderen Markierung versehen werden, die besagt, dass diese Teile nicht im Cache aufgehoben werden können. Eine andere Möglichkeit besteht darin, I/O-Operationen nur vom Betriebssystem ausführen zu lassen, so dass dieses bei Bedarf Daten im Hauptspeicher vor der I/O-Operation durch Zugriff auf den Cache aktualisieren kann.

Befindet sich bei einem Schreibzugriff die Zieladresse nicht im Cache (*write miss*), so wird bei den meisten Caches der Speicherblock, der die Zieladresse enthält, zuerst in den Cache geladen und die Modifizierung wird wie oben skizziert durchgeführt (*write-allocate*). Eine weniger oft verwendete Alternative besteht darin, den Speicherblock nur im Hauptspeicher zu modifizieren und nicht in den Cache zu laden (*write no allocate*).

Anzahl der Caches

In der bisherigen Beschreibung haben wir die Arbeitsweise eines einzelnen Caches beschrieben, der zwischen Prozessor und Hauptspeicher geschaltet ist und in dem Daten des auszuführenden Programms abgelegt werden. Ein in dieser Weise verwendeter Cache wird als **Datencache** erster Stufe bezeichnet. Neben den Programmdaten greift ein Prozessor auch auf die Instruktionen des auszuführenden Programms zu, um diese zu dekodieren und die angegebenen Operationen auszuführen. Dabei wird wegen Schleifen im Programm auf einzelne Instruktionen evtl. mehrfach zugegriffen und die Instruktionen müssen mehrfach geladen werden. Obwohl die Instruktionen im gleichen Cache wie die Daten aufgehoben und auf die gleiche Weise verwaltet werden können, verwendet man in der Praxis meistens einen separaten **Instruktionscache**, d. h. die Instruktionen und die Daten eines Programms werden in separaten Caches aufgehoben (*split cache*). Dies erlaubt eine größere Flexibilität beim Design der Caches, da getrennte Daten- und Instruktionscaches entsprechend der Prozessororganisation unterschiedliche Größe und Assoziativität haben und unabhängig voneinander arbeiten können.

In der Praxis werden häufig **mehrstufige Caches**, also mehrere hierarchisch angeordnete Caches, verwendet, siehe Abb. 2.31 zur Illustration einer zweistufigen Speicherhierarchie. Zur Zeit wird für die meisten Desktop-Prozessoren eine dreistufige Speicherhierarchie, bestehend aus L1-Cache, L2-Cache und L3-Cache verwendet, wobei alle Caches auf der Chipfläche des Prozessors integriert sind. Typische Cachegrößen sind 8 KBytes bis 64 KBytes für den L1-Cache, 128 KBytes bis

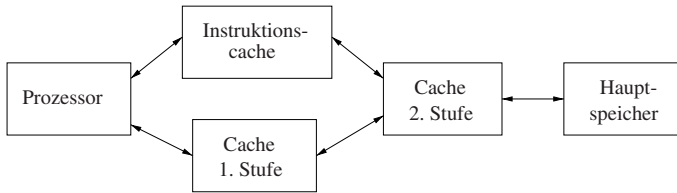


Abb. 2.31 Zweistufige Speicherhierarchie

512 KBytes für den L2-Cache und 2 MBytes bis 16 MBytes für den L3-Cache. Für einen typischen Desktoprechner liegt die Hauptspeichergroße zwischen 8 GBytes und 48 GBytes. Diese Angaben beziehen sich auf das Jahr 2012.

2.7.2 Cache-Kohärenz

Im letzten Abschnitt haben wir gesehen, dass die Einführung von schnellen Cache-Speichern zwar das Problem des zu langsamen Speicherzugriffs auf den Hauptspeicher löst, dafür aber die zusätzliche Aufgabe aufwirft, dafür zu sorgen, dass sich Veränderungen von Daten im Cache-Speicher auch auf den Hauptspeicher auswirken, und zwar spätestens dann, wenn andere Komponenten (also z. B. I/O-Systeme oder andere Prozessoren) auf den Hauptspeicher zugreifen. Diese anderen Komponenten sollen natürlich auf den *korrekten* Wert zugreifen, also auf den Wert, der zuletzt einer Variablen zugewiesen wurde. Wir werden dieses Problem in diesem Abschnitt näher untersuchen, wobei wir insbesondere Systeme mit mehreren unabhängig voneinander arbeitenden Prozessoren betrachten.

In einem Multiprozessor, in dem jeder Prozessor jeweils einen lokalen Cache besitzt, können Prozessoren gleichzeitig ein und denselben Speicherblock in ihrem lokalen Cache haben. Nach Modifikation derselben Variable in verschiedenen lokalen Caches können die lokalen Caches und der globale Speicher verschiedene, also *inkonsistente* Werte enthalten. Dies widerspricht dem Programmiermodell der gemeinsamen Variablen und kann zu falschen Ergebnissen führen. Diese bei Vorhandensein von lokalen Caches aufkommende Schwierigkeit bei Multiprozessoren wird als *Speicherkohärenz-Problem* oder häufiger als **Cache-Kohärenz-Problem** bezeichnet. Wir illustrieren das Problem an einem einfachen busbasierten System mit drei Prozessoren [32].

Beispiel

Ein busbasiertes System bestehe aus drei Prozessoren P_1, P_2, P_3 mit jeweils einem lokalen Cache C_1, C_2, C_3 . Die Prozessoren sind über einen zentralen Bus mit dem gemeinsamen Speicher M verbunden. Für die Caches nehmen wir eine Write-Through-Rückschreibestrategie an. Auf eine Variable u im Speicher M

mit Wert 5 werden zu aufeinanderfolgenden Zeitpunkten t_1, \dots, t_4 die folgenden Operationen angewendet:

Zeitpunkt	Operation
t_1 :	Prozessor P_1 liest Variable u . Der Block, der Variable u enthält, wird daraufhin in den Cache C_1 geladen.
t_2 :	Prozessor P_3 liest Variable u . Der Block, der Variable u enthält, wird daraufhin in den Cache C_3 geladen.
t_3 :	Prozessor P_3 schreibt den Wert 7 in u . Die Veränderung wird aufgrund der Write-Through-Rückschreibestrategie auch im Speicher M vorgenommen.
t_4 :	Prozessor P_1 liest u durch Zugriff auf seinen Cache C_1 .

Der Prozessor P_1 liest also zum Zeitpunkt t_4 den alten Wert 5 statt den neuen Wert 7, was für weitere Berechnungen zu Fehlern führen kann. Dabei wurde angenommen, dass eine *write-through*-Rückschreibestrategie verwendet wird und daher zum Zeitpunkt t_3 der neue Wert 7 direkt in den Speicher zurückgeschrieben wird. Bei Verwendung einer *write-back*-Rückschreibestrategie würde zum Zeitpunkt t_3 der Wert von u im Speicher nicht aktualisiert werden, sondern erst beim Ersetzen des Blockes, in dem sich u befindet. Zum Zeitpunkt t_4 des Beispiels würde P_1 ebenfalls den falschen Wert lesen.

Um Programme in einem Programmiermodell mit gemeinsamem Adressraum auf Multiprozessoren korrekt ausführen zu können, muss gewährleistet sein, dass bei jeder möglichen Anordnung von Lese- und Schreibzugriffen, die von den einzelnen Prozessoren auf gemeinsamen Variablen durchgeführt werden, jeweils der richtige Wert gelesen wird, egal ob sich der Wert bereits im Cache befindet oder erst geladen werden muss. Das Verhalten eines Speichersystems bei Lese- und Schreibzugriffen von eventuell *verschiedenen* Prozessoren auf die *gleiche* Speicherzelle wird durch den Begriff der **Kohärenz des Speichersystems** beschrieben. Ein Speichersystem ist kohärent, wenn für jede Speicherzelle gilt, dass jede Leseoperation den *letzten* geschriebenen Wert zurückliefert. Da mehrere Prozessoren gleichzeitig oder fast gleichzeitig auf die gleiche Speicherzelle schreibend zugreifen können, ist zunächst zu präzisieren, welches der *zuletzt geschriebene* Wert ist. Als Zeitmaß ist in einem parallelen Programm nicht der Zeitpunkt des physikalischen Lesens oder Beschreibens einer Variable maßgeblich, sondern die Reihenfolge im zugrunde liegenden Programm. Dies wird in nachfolgender Definition berücksichtigt [75].

Ein Speichersystem ist **kohärent**, wenn die folgenden Bedingungen erfüllt sind:

1. Wenn ein Prozessor P die Speicherzelle x zum Zeitpunkt t_1 beschreibt und zum Zeitpunkt $t_2 > t_1$ liest und wenn zwischen den Zeitpunkten t_1 und t_2 kein anderer Prozessor die Speicherzelle x beschreibt, erhält Prozessor P zum Zeitpunkt t_2 den von ihm geschriebenen Wert zurück. Dies bedeutet, dass für jeden Prozessor die für ihn geltende Programmreihenfolge der Speicherzugriffe trotz der parallelen Ausführung erhalten bleibt.

2. Wenn ein Prozessor P_1 zum Zeitpunkt t_1 eine Speicherzelle x beschreibt und ein Prozessor P_2 zum Zeitpunkt $t_2 > t_1$ die Speicherzelle x liest, erhält P_2 den von P_1 geschriebenen Wert zurück, wenn zwischen t_1 und t_2 kein anderer Prozessor x beschreibt und wenn $t_2 - t_1$ genügend groß ist. Der neue Wert muss also nach einer gewissen Zeit für andere Prozessoren sichtbar sein.
3. Wenn zwei beliebige Prozessoren die gleiche Speicherzelle x beschreiben, werden diese Schreibzugriffe so *sequentialisiert*, dass alle Prozessoren die Schreibzugriffe in der gleichen Reihenfolge sehen. Diese Bedingung wird *globale Schreibsequentialisierung* genannt.

Zur Sicherstellung der Cache-Kohärenz in parallelen Systemen mit Cache-Hierarchie werden Cache-Kohärenz-Protokolle eingesetzt. Diese Protokolle basieren darauf, für jeden in einem Cache abgelegten Speicherblock den aktuellen Modifikationszustand zu dokumentieren. Je nach Kopplung der Komponenten des parallelen Systems werden dabei **Snooping-Protokolle** oder **verzeichnisbasierte Protokolle** (engl. *directory-based protocol*) eingesetzt. Snooping-Protokolle beruhen dabei auf der Existenz eines gemeinsamen Mediums, über das die Speicherzugriffe der Prozessoren oder Prozessorkerne laufen. Dies kann ein gemeinsamer Bus oder ein gemeinsam genutzter Cache sein. Für verzeichnisbasierte Protokolle muss dagegen kein gemeinsames Zugriffsmedium existieren. Wir geben im Folgenden einen kurzen Überblick über die wichtigsten Aspekte dieser Protokolle und verweisen z. B. auf [32, 76, 117] für eine detaillierte Behandlung.

Snooping-Protokolle

Snooping-Protokolle können eingesetzt werden, wenn alle Speicherzugriffe über ein gemeinsames Medium laufen, das von allen Prozessoren beobachtet werden kann. Für frühere SMP-Systeme war dies häufig ein gemeinsamer Bus. Für aktuelle Multicore-Prozessoren wie z. B. dem Intel Core i7 kann dies die Verbindung zwischen den privaten L1- und L2-Caches und dem gemeinsam genutzten L3-Cache sein. Wie gehen im Folgenden von der Existenz eines solchen gemeinsamen Mediums aus und betrachten zuerst ein System mit write-through-Rückschreibstrategie der Caches. Als gemeinsames Medium nehmen wir einen gemeinsamen Bus an. Snooping-Protokolle beruhen darauf, dass alle relevanten Speicherzugriffe über das gemeinsame Medium laufen und von den *Cache-Controllern* aller anderen Prozessoren beobachtet werden können. Somit kann jeder Prozessor durch *Überwachung* der über den Bus ausgeführten Speicherzugriffe feststellen, ob durch den Speicherzugriff ein Wert in seinem lokalen Cache (der ja eine Kopie des Wertes im Hauptspeicher ist) aktualisiert werden sollte. Ist dies der Fall, so aktualisiert der beobachtende Prozessor den Wert in seinem lokalen Cache, indem er den an der Datenleitung anliegenden Wert kopiert. Die lokalen Caches enthalten so stets die aktuellen Werte. Wird obiges Beispiel unter Einbeziehung von Bus-Snooping betrachtet, so kann Prozessor P_1 den Schreib-

zugriff von P_3 beobachten und den Wert von Variable u im lokalen Cache C_1 aktualisieren.

Diese Snooping-Technik beruht auf der Verwendung von Caches mit write-through-Rückschreibestrategie. Deshalb tritt bei der Snooping-Technik das Problem auf, dass viel Verkehr auf dem zentralen Bus stattfinden kann, da *jede* Schreiboperation über den Bus ausgeführt wird. Dies kann einen erheblichen Nachteil darstellen und zu Engpässen führen, was an folgendem Beispiel deutlich wird [32]. Wir betrachten ein Bussystem mit 2 GHz-Prozessoren, die eine Instruktion pro Zyklus ausführen. Verursachen 15 % aller Instruktionen Schreibzugriffe mit 8 Bytes je Schreibzugriff, so erzeugt jeder Prozessor 300 Millionen Schreibzugriffe pro Sekunde. Jeder Prozessor würde also eine Busbandbreite von 2,4 GB/s benötigen. Ein Bus mit einer Bandbreite von 10 GB/s könnte dann also maximal vier Prozessoren ohne Auftreten von Staus (*congestion*) versorgen.

Für write-back-Caches kann ein ähnliches Schema verwendet werden: Jeder Prozessor beobachtet alle über das gemeinsame Medium laufenden Speicherzugriffe. Stellt ein Prozessor A fest, dass ein anderer Prozessor B einen Lesezugriff für einen Speicherblock abgesetzt hat, den A in seinem lokalen Cache modifiziert hat, stellt A diesen Speicherblock aus seinem lokalen Speicher B zur Verfügung und der eigentliche Speicherzugriff wird nicht mehr ausgeführt. Je nach Modifikationszustand eines Speicherblocks kann ein Speicherzugriff aus dem Speicher (oder dem gemeinsamen L3-Cache) oder aus dem privaten Cache eines anderen Prozessors befriedigt werden, was zu unterschiedlichen Ladezeiten führen kann. Wir beschreiben im Folgenden ein einfaches Protokoll für write-back-Caches wie sie für aktuelle Multicore-Prozessoren häufig eingesetzt werden. Für eine ausführlichere Behandlung verweisen wir auf [32].

Write-Back-Invalidierungs-Protokoll (MSI-Protokoll)

Das Write-Back-Invalidierungs-Protokoll benutzt drei Zustände, die ein im Cache befindlicher Speicherblock annehmen kann, wobei der gleiche Speicherblock in unterschiedlichen Caches unterschiedlich markiert sein kann:

- M** für *modified* (modifiziert) bedeutet, dass nur der betrachtete Cache die aktuelle Version des Speicherblocks enthält und die Kopien des Blockes im Hauptspeicher und allen anderen Caches *nicht* aktuell sind,
- S** für *shared* (gemeinsam) bedeutet, dass der Speicherblock im unmodifizierten Zustand in einem oder mehreren Caches gespeichert ist und alle Kopien den aktuellen Wert enthalten,
- I** für *invalid* (ungültig) bedeutet, dass der Speicherblock im betrachteten Cache ungültige Werte enthält.

Diese drei Zustände geben dem MSI-Protokoll seinen Namen. Bevor ein Prozessor einen in seinem lokalen Cache befindlichen Speicherblock beschreibt, ihn also modifiziert, werden alle Kopien dieses Blockes in anderen Caches als *ungültig* (I)

markiert. Dies geschieht durch eine Operation über den Bus. Der Cacheblock im eigenen Cache wird als modifiziert (M) markiert. Der zugehörige Prozessor kann nun mehrere Schreiboperationen durchführen, ohne dass eine weitere Busoperation nötig ist. Für die Verwaltung des Protokolls werden die drei folgenden Busoperationen bereitgestellt:

- a) **Bus Read** (BusRd): Die Operation wird durch eine Leseoperation eines Prozessors auf einen Wert ausgelöst, der sich nicht im lokalen Cache befindet. Der zugehörige Cache-Controller fordert durch Angabe einer Hauptspeicheradresse eine Kopie eines Cacheblockes an, die er nicht modifizieren will. Das Speichersystem stellt den betreffenden Block aus dem Hauptspeicher oder einem anderen Cache zur Verfügung.
- b) **Bus Read Exclusive** (BusRdEx): Die Operation wird durch eine Schreiboperation auf einen Speicherblock ausgelöst, der sich entweder nicht im lokalen Cache befindet oder nicht zum Modifizieren geladen wurde, d. h. nicht mit (M) markiert wurde. Der Cache-Controller fordert durch Angabe der Hauptspeicheradresse eine exklusive Kopie des Speicherblocks an, den er modifizieren will. Das Speichersystem stellt den Block aus dem Hauptspeicher oder einem anderen Cache zur Verfügung. Alle Kopien des Blockes in anderen Caches werden als ungültig (I) gekennzeichnet.
- c) **Write Back** (BusWr): Der Cache-Controller schreibt einen als modifiziert (M) gekennzeichneten Cacheblock in den Hauptspeicher zurück. Die Operation wird ausgelöst durch das Ersetzen des Cacheblockes.

Der Prozessor selbst führt nur übliche Lese- und Schreiboperationen (PrRd, PrWr) aus, vgl. Abb. 2.32 rechts. Der Cache-Controller stellt die vom Prozessor angefragten Speicherworte zur Verfügung, indem er sie entweder aus dem lokalen Cache lädt oder die zugehörigen Speicherblöcke mit Hilfe einer Busoperation besorgt. Die genauen Operationen und Zustandsübergänge sind in Abb. 2.32 links angegeben.

Das Lesen und Schreiben eines mit (M) markierten Cacheblockes kann ohne Busoperation im lokalen Cache vorgenommen werden. Dies gilt auch für das Lesen eines mit (S) markierten Cacheblockes. Zum Schreiben auf einen mit (S) markierten Cacheblock muss der Cache-Controller zuerst mit BusRdEx die alleinige Kopie des Cacheblockes erhalten. Die Cache-Controller anderer Prozessoren, die diesen Cacheblock ebenfalls mit (S) markiert in ihrem lokalen Cache haben, beobachten diese Operation auf dem Bus und markieren daraufhin ihre lokale Kopie als ungültig (I). Wenn ein Prozessor einen Speicherblock zu lesen versucht, der nicht in seinem lokalen Cache liegt oder der dort als ungültig (I) markiert ist, besorgt der zugehörige Cache-Controller durch Ausführen einer BusRd-Operation eine Kopie des Speicherblockes und markiert sie im lokalen Cache als shared (S). Wenn ein anderer Prozessor diesen Speicherblock in seinem lokalen Cache mit (M) markiert hat, d. h. wenn er die einzig gültige Kopie dieses Speicherblockes hat, stellt der zugehörige Cache-Controller den Speicherblock auf dem Bus zur Verfügung und markiert seine lokale Kopie als sha-

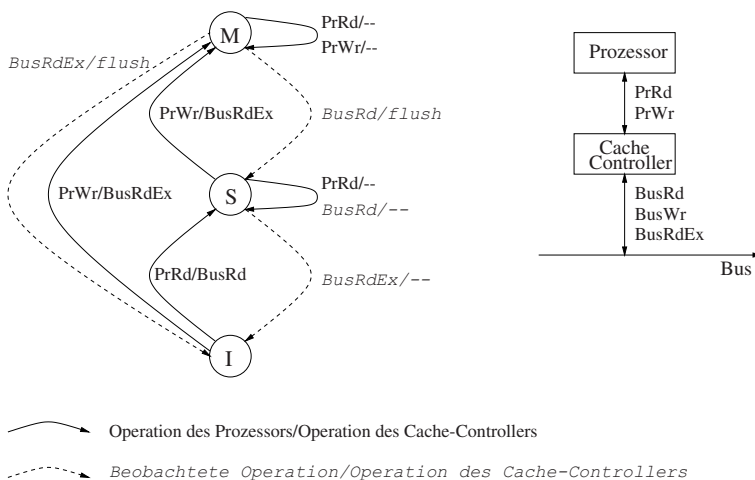


Abb. 2.32 Illustration des MSI-Protokolls: Die möglichen Zustände der Cacheblöcke eines Prozessors sind M (modified), S (shared) und I (invalid). Zustandsübergänge sind durch Pfeile angegeben, die durch Operationen markiert sind. Zustandsübergänge können ausgelöst werden durch: **a** (durchgezogene Pfeile) Operationen des eigenen Prozessors (PrRd und PrWr). Die entsprechende Busoperation des Cache-Controllers ist hinter dem Schrägstrich angegeben. Wenn keine Busoperation angegeben ist, muss nur ein Zugriff auf den lokalen Cache ausgeführt werden. **b** (gestrichelte Pfeile) Vom Cache-Controller auf dem Bus beobachtete Operationen, die durch andere Prozessoren ausgelöst sind. Die daraufhin erfolgende Operation des Cache-Controllers ist wieder hinter dem Schrägstrich angegeben. *flush* bedeutet hierbei, dass der Cache-Controller den gewünschten Wert auf den Bus legt. Wenn für einen Zustand für eine bestimmte Busoperation keine Kante angegeben ist, ist keine Aktion des Cache-Controllers erforderlich und es findet kein Zustandsübergang statt

red (S). Wenn ein Prozessor einen Speicherblock zu schreiben versucht, der nicht in seinem lokalen Cache liegt oder der dort als ungültig (I) markiert ist, besorgt der zugehörige Cache-Controller durch Ausführen einer BusRdEx-Operation die alleinige Kopie des Speicherblockes und markiert sie im lokalen Cache als modified (M). Wenn ein anderer Prozessor diesen Speicherblock in seinem lokalen Cache mit (M) markiert hat, stellt der zugehörige Cache-Controller den Speicherblock auf dem Bus zur Verfügung und markiert seine lokale Kopie als ungültig (I).

Der Nachteil des beschriebenen Protokolls besteht darin, dass ein Prozessor, der zunächst ein Datum liest und dann beschreibt, zwei Busoperationen BusRd und BusRdEx auslöst, und zwar auch dann, wenn kein anderer Prozessor beteiligt ist. Dies trifft auch dann zu, wenn ein einzelner Prozessor ein sequentielles Programm ausführt, was für kleinere SMPs häufig vorkommt. Dieser Nachteil des MSI-Protokolls wird durch die Einführung eines weiteren Zustandes (E) für *exclusive* im sogenannten MESI-Protokoll ausgeglichen. Wenn ein Speicherblock in einem Cache mit

E für *exclusive* (exklusiv) markiert ist, bedeutet dies, dass nur der betrachtete Cache eine Kopie des Blockes enthält und dass diese Kopie *nicht* modifiziert ist, so dass auch der Hauptspeicher die aktuellen Werte dieses Speicherblockes enthält.

Wenn ein Prozessor einen Speicherblock zum Lesen anfordert und kein anderer Prozessor eine Kopie dieses Speicherblocks in seinem lokalen Cache hat, markiert der lesende Prozessor diesen Speicherblock mit (E) statt bisher mit (S), nachdem er ihn aus dem Hauptspeicher über den Bus erhalten hat. Wenn dieser Prozessor den mit (E) markierten Speicherblock zu einem späteren Zeitpunkt beschreiben will, kann er dies tun, nachdem er die Markierung des Blockes lokal von (E) zu (M) geändert hat. In diesem Fall ist also keine Busoperation nötig. Wenn seit dem ersten Lesen durch den betrachteten Prozessor ein anderer Prozessor lesend auf den Speicherblock zugegriffen hätte, wäre der Zustand von (E) zu (S) geändert worden und die für das MSI-Protokoll beschriebenen Aktionen würden ausgeführt. Für eine genauere Beschreibung verweisen wir auf [32]. Varianten des MESI-Protokolls werden in vielen Prozessoren verwendet und spielen auch für Multicore-Prozessoren eine große Rolle.

Eine Alternative zu Invalidierungsprotokollen stellen **Write-Back-Update**-Protokolle dar. Bei diesen Protokollen werden nach Aktualisierung eines (mit (M) gekennzeichneten) Cacheblockes auch alle anderen Caches, die diesen Block ebenfalls enthalten, aktualisiert. Die lokalen Caches enthalten also immer die aktuellen Werte. In der Praxis werden diese Protokolle aber meist nicht benutzt, da sie zu erhöhtem Verkehr auf dem Bus führen.

Cache-Kohärenz in nicht-busbasierten Systemen

Bei nicht-busbasierten Systemen kann Cache-Kohärenz nicht so einfach wie bei busbasierten Systemen realisiert werden, da kein zentrales Medium existiert, über das alle Speicheranfragen laufen. Der einfachste Ansatz besteht darin, *keine* Hardware-Cache-Kohärenz zur Verfügung zu stellen. Um Probleme mit der fehlenden Cache-Kohärenz zu vermeiden, können die lokalen Caches nur Daten aus den jeweils lokalen Speichern aufnehmen. Daten aus den Speichern anderer Prozessoren können nicht per Hardware im lokalen Cache abgelegt werden. Bei häufigen Zugriffen kann ein Aufheben im Cache aber per Software dadurch erreicht werden, dass die Daten in den lokalen Speicher kopiert werden. Dem Vorteil, ohne zusätzliche Hardware auszukommen, steht gegenüber, dass Zugriffe auf den Speicher anderer Prozessoren teuer sind. Bei häufigen Zugriffen muss der Programmierer dafür sorgen, dass die benötigten Daten in den lokalen Speicher kopiert werden, um von dort über den lokalen Cache schneller zugreifbar zu sein.

Die Alternative besteht darin, Hardware-Cache-Kohärenz mit Hilfe eines alternativen Protokolls zur Verfügung zu stellen. Dazu kann ein **Directory-Protokoll** eingesetzt werden. Die Idee besteht darin, ein zentrales Verzeichnis (engl. *directory*) zu verwenden, das den Zustand jedes Speicherblockes enthält. Anstatt den zentralen

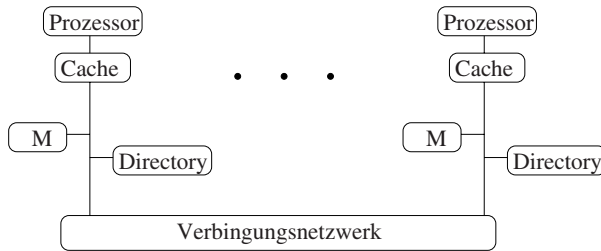


Abb. 2.33 Directory-basierte Cache-Kohärenz

Bus zu beobachten, kann ein Cache-Controller den Zustand eines Speicherblockes durch ein Nachschauen in dem Verzeichnis erfahren. Dabei kann das Verzeichnis auf die verschiedenen Prozessoren verteilt werden, um zu vermeiden, dass der Zugriff auf das Verzeichnis zu einem Flaschenhals wird. Um dem Leser eine Idee von der Arbeitsweise eines Directory-Protokolls zu geben, beschreiben wir im Folgenden ein einfaches Schema. Für eine ausführlichere Beschreibung verweisen wir auf [32, 75].

Wir betrachten einen Shared-Memory-Rechner mit physikalisch verteiltem Speicher und nehmen an, dass zu jedem lokalen Speicher eine Tabelle (*Directory* genannt) gehalten wird, die zu jedem Speicherblock des lokalen Speichers angibt, in welchem Cache anderer Prozessoren dieser Speicherblock zur Zeit enthalten ist. Für eine Maschine mit p Prozessoren kann ein solches Directory dadurch realisiert werden, dass ein Bitvektor pro Speicherblock gehalten wird, der p *presence*-Bits und eine Anzahl von Statusbits enthält. Jedes der *presence*-Bits gibt für einen bestimmten Prozessor an, ob der zugehörige Cache eine gültige Kopie des Speicherblock enthält (Wert 1) oder nicht (Wert 0). Wir nehmen im Folgenden an, dass nur ein Statusbit (*dirty*-Bit) verwendet wird, das angibt, ob der Hauptspeicher die aktuelle Version des Speicherblockes enthält (Wert 0) oder nicht (Wert 1). Jedes Directory wird von einem eigenen Directory-Controller verwaltet, der auf die über das Netzwerk eintreffenden Anfragen wie im Folgenden beschrieben reagiert. Abbildung 2.33 veranschaulicht die Organisation. In den lokalen Caches sind die Speicherblöcke wie oben beschrieben mit (M), (S) oder (I) markiert. Die Prozessoren greifen über ihre lokalen Cache-Controller auf die Speicheradressen zu, wobei wir einen globalen Adressraum annehmen.

Bei Auftreten eines Cache-Fehlzugriffs bei einem Prozessor i greift der Cache-Controller von i über das Netzwerk auf das Directory zu, das die Informationen über den Speicherblock enthält. Wenn es sich um einen lokalen Speicherblock handelt, reicht ein lokaler Zugriff aus, ansonsten muss über das Netzwerk auf das entsprechende Directory zugegriffen werden. Wir beschreiben im Folgenden den Fall eines nicht-lokalen Zugriffs. Wenn es sich um einen Lese-Fehlzugriff handelt (engl. *read miss*), reagiert der zugehörige Directory-Controller wie folgt:

- Wenn das *dirty*-Bit des Speicherblockes auf 0 gesetzt ist, liest der Directory-Controller den Speicherblock aus dem zugehörigen Hauptspeicher mit Hilfe eines lokalen Zugriffes und schickt dem anfragenden Cache-Controller dessen Inhalt über das Netzwerk zu. Das *presence*-Bit des zugehörigen Prozessors wird danach auf 1 gesetzt, um zu vermerken, dass dieser Prozessor eine gültige Kopie des Speicherblockes hat.
- Wenn das *dirty*-Bit des Speicherblockes auf 1 gesetzt ist, gibt es genau einen Prozessor, der die aktuelle Version des Speicherblockes enthält, d. h. das *presence*-Bit ist für genau einen Prozessor j gesetzt. Der Directory-Controller schickt über das Netzwerk eine Anfrage an diesen Prozessor. Dessen Cache-Controller setzt den Zustand des Speicherblockes von (M) auf (S) und schickt dessen Inhalt an den ursprünglich anfragenden Prozessor i und an den Directory-Controller des Speicherblockes. Letzterer schreibt den aktuellen Wert in den zugehörigen Hauptspeicher, setzt das *dirty*-Bit auf 0 und das *presence*-Bit von Prozessor i auf 1. Das *presence*-Bit von Prozessor j bleibt auf 1.

Wenn es sich um einen Schreib-Fehlzugriff handelt (engl. *write miss*), reagiert der Directory-Controller wie folgt:

- Wenn das *dirty*-Bit des Speicherblockes auf 0 gesetzt ist, enthält der Hauptspeicher den aktuellen Wert des Speicherblockes. Der Directory-Controller schickt an alle Prozessoren j , deren *presence*-Bit auf 1 gesetzt ist, über das Netzwerk eine Mitteilung, dass deren Kopien als ungültig zu markieren sind. Die *presence*-Bits dieser Prozessoren werden auf 0 gesetzt. Nachdem der Erhalt dieser Mitteilungen von allen zugehörigen Cache-Controller bestätigt wurde, wird der Speicherblock an Prozessor i geschickt, dessen *presence*-Bit und das *dirty*-Bit des Speicherblockes werden auf 1 gesetzt. Nach Erhalt des Speicherblockes setzt der Cache-Controller von i dessen Zustand auf (M).
- Wenn das *dirty*-Bit des Speicherblockes auf 1 gesetzt ist, wird der Speicherblock über das Netzwerk von dem Prozessor j geladen, dessen *presence*-Bit auf 1 gesetzt ist. Dann wird der Speicherblock an Prozessor i geschickt, das *presence*-Bit von j wird auf 0, das *presence*-Bit von i auf 1 gesetzt. Das *dirty*-Bit bleibt auf 1 gesetzt.

Wenn ein Speicherblock im Cache eines Prozessors i ersetzt werden soll, in dem er als einzige Kopie liegt, also mit (M) markiert ist, wird er vom Cache-Controller von i an den zugehörigen Directory-Controller geschickt. Dieser schreibt den Speicherblock mit einer lokalen Operation in den Hauptspeicher zurück, setzt das *dirty*-Bit und das *presence*-Bit von i auf 0. Ein mit (S) markierter Speicherblock kann dagegen ohne Mitteilung an den Directory-Controller ersetzt werden. Eine Mitteilung an den Directory-Controller vermeidet aber, dass bei einem Schreib-Fehlzugriff wie oben beschrieben eine dann unnötige Invalidierungsnachricht an den Prozessor geschickt wird.

2.7.3 *Speicherkonsistenz*

Speicher- bzw. Cache-Kohärenz liegt vor, wenn jeder Prozessor das *gleiche eindeutige* Bild des Speichers hat, d. h. wenn jeder Prozessor zu jedem Zeitpunkt für jede Variable den gleichen Wert erhält wie alle anderen Prozessoren des Systems (bzw. erhalten würde, falls das Programm einen entsprechenden Zugriff auf die Variable enthalten würde). Die Speicher- oder Cache-Kohärenz sagt allerdings nichts über die Reihenfolge aus, in der die Auswirkungen der Speicheroperationen *sichtbar* werden. **Speicherkonsistenzmodelle** beschäftigen sich mit der Fragestellung, in welcher Reihenfolge die Speicherzugriffsoperationen eines Prozessors von den anderen Prozessoren beobachtet werden. Die verschiedenen Speicherkonsistenzmodelle werden gemäß der folgenden Kriterien charakterisiert.

1. Werden die Speicherzugriffsoperationen der einzelnen Prozessoren in deren Programmreihenfolge ausgeführt?
2. Sehen alle Prozessoren die ausgeführten Speicherzugriffsoperationen in der gleichen Reihenfolge?

Das folgende Beispiel zeigt die Vielfalt der möglichen Ergebnisse eines Programmes für Multiprozessoren, wenn verschiedene Reihenfolgen der Anweisungen der Programme der einzelnen Prozessoren (also Sequentialisierungen des Multiprozessorprogramms) betrachtet werden, siehe auch [87].

Beispiel

Drei Prozessoren P_1 , P_2 , P_3 führen ein Mehrprozessorprogramm aus, das die gemeinsamen Variablen x_1 , x_2 , x_3 enthält. Die Variablen x_1 , x_2 und x_3 seien mit dem Wert 0 initialisiert. Die Programme der Prozessoren P_1 , P_2 , P_3 seien folgendermaßen gegeben:

Prozessor	P_1	P_2	P_3
Programm	(1) $x_1 = 1$; (2) print x_2, x_3 ;	(3) $x_2 = 1$; (4) print x_1, x_3 ;	(5) $x_3 = 1$; (6) print x_1, x_2 ;

Nachdem die Prozessoren P_i den Wert x_i mit 1 beschrieben haben, werden die Werte der Variablen x_j , $j = 1, 2, 3$, $j \neq i$ ausgedruckt, $i = 1, 2, 3$. Die Ausgabe des Multiprozessorprogramms enthält also 6 Ausgabewerte, die jeweils den Wert 0 oder 1 haben können. Insgesamt gibt es $2^6 = 64$ Ausgabekombinationen bestehend aus 0 und 1, wenn jeder Prozessor seine Anweisungen in einer beliebigen Reihenfolge ausführen kann und wenn die Anweisungen der verschiedenen Prozessoren beliebig gemischt werden können. Dabei können verschiedene globale Auswertungsreihenfolgen zur gleichen Ausgabe führen. Führt jeder Prozessor seine Anweisungen in der vorgegebenen Reihenfolge aus, also z. B. Prozessor P_1 erst (1) und dann (2), so ist die Ausgabe 000000 *nicht* möglich, da zunächst ein Beschreiben zumindest einer Variable mit 1 vor einer

Ausgabeoperation ausgeführt wird. Eine mögliche Sequentialisierung stellt die Reihenfolge (1), (2), (3), (4), (5), (6), dar. Die zugehörige Ausgabe ist 001011.

Sequentielles Konsistenzmodell – SC-Modell

Ein häufig verwendetes Speicherkonsistenzmodell ist das Modell der **sequentiellen Konsistenz** (engl. *sequential consistency*) [109], das von den verwendeten Konsistenzmodellen die stärksten Einschränkungen an die Reihenfolge der durchgeführten Speicherzugriffe stellt. Ein Multiprozessorsystem ist sequentiell konsistent, wenn die Speicheroperationen jedes Prozessors in der von seinem Programm vorgegebenen Reihenfolge ausgeführt werden und wenn der Gesamteffekt aller Speicheroperationen aller Prozessoren für alle Prozessoren in der *gleichen* sequentiellen Reihenfolge erscheint, die sich durch Mischung der Reihenfolgen der Speicheroperationen der einzelnen Prozessoren ergibt. Dabei werden die abgesetzten Speicheroperationen als *atomare* Operationen abgearbeitet. Eine Speicheroperation wird als atomar angesehen, wenn der Effekt der Operation für alle Prozessoren sichtbar wird, bevor die nächste Speicheroperation (irgendeines Prozessors des Systems) abgesetzt wird.

Der in der Definition der sequentiellen Konsistenz verwendete Begriff der *Programmreihenfolge* ist im Prinzip nicht exakt festgelegt. So kann u. a. die Reihenfolge der Anweisungen im *Quellprogramm* gemeint sein, oder aber auch die Reihenfolge von Speicheroperationen in einem von einem optimierenden Compiler erzeugten Maschinenprogramm, das eventuell Umordnungen von Anweisungen zur besseren Auslastung des Prozessors enthält. Wir gehen im Folgenden davon aus, dass das sequentielle Konsistenzmodell sich auf die Reihenfolge im Quellprogramm bezieht, da der Programmierer sich nur an dieser Reihenfolge orientieren kann.

Im sequentiellen Speicherkonsistenzmodell werden also alle Speicheroperationen als atomare Operationen in der Reihenfolge des Quellprogramms ausgeführt und zentral sequentialisiert. Dies ergibt eine *totale Ordnung* der Speicheroperationen eines parallelen Programmes, die für alle Prozessoren des Systems gilt. Im vorherigen Beispiel entspricht die Ausgabe 001011 dem sequentiellen Speicherkonsistenzmodell, aber auch 111111. Die Ausgabe 011001 ist bei sequentieller Konsistenz dagegen nicht möglich.

Die totale Ordnung der Speicheroperationen ist eine stärkere Forderung als bei der im letzten Abschnitt beschriebenen Speicherkohärenz. Die Kohärenz eines Speichersystems verlangte eine Sequentialisierung der Schreiboperationen, d. h. die Ausführung von Schreiboperationen auf die *gleiche* Speicherzelle erscheinen für alle Prozessoren in der gleichen Reihenfolge. Die sequentielle Speicherkonsistenz verlangt hingegen, dass *alle* Schreiboperationen (auf beliebige Speicherzellen) für alle Prozessoren in der gleichen Reihenfolge ausgeführt erscheinen. Das folgende Beispiel zeigt, dass die Atomarität der Schreiboperationen wichtig für die Definition der sequentiellen Konsistenz ist und dass die Sequentialisierung der Schreiboperationen alleine für eine eindeutige Definition nicht ausreicht.

Beispiel

Drei Prozessoren P_1 , P_2 , P_3 arbeiten folgende Programmstücke ab. Die Variablen x_1 und x_2 seien mit 0 vorbesetzt.

Prozessor	P_1	P_2	P_3
Programm	(1) $x_1 = 1$;	(2) while($x_1 == 0$); (3) $x_2 = 1$;	(4) while($x_2 == 0$); (5) print(x_1);

Prozessor P_2 wartet, bis x_1 den Wert 1 erhält, und setzt x_2 dann auf 1; Prozessor P_3 wartet, bis x_2 den Wert 1 annimmt, und gibt dann den Wert von x_1 aus.

Unter Einbehaltung der Atomarität von Schreiboperationen würde die Reihenfolge (1), (2), (3), (4), (5) gelten und Prozessor P_3 würde den Wert 1 für x_1 ausdrucken, da die Schreiboperation (1) von P_1 auch für P_3 sichtbar sein muss, bevor Prozessor P_2 die Operation (3) ausführt. Reine Sequentialisierung von Schreibbefehlen einer Variable *ohne* die in der sequentiellen Konsistenz geforderte Atomarität und globale Sequentialisierung würde die Ausführung von (3) vor Sichtbarwerden von (1) für P_3 erlauben und damit die Ausgabe des Wertes 0 für x_1 möglich machen. Um dies zu verdeutlichen, untersuchen wir einen mit einem Directory-Protokoll arbeitenden Parallelrechner, dessen Prozessoren über ein Netzwerk miteinander verbunden sind. Wir nehmen an, dass ein Invalidierungsprotokoll auf Directory-Basis verwendet wird, um die Caches der Prozessoren kohärent zu halten. Weiter nehmen wir an, dass zu Beginn der Abarbeitung des angegebenen Programmstücks die Variablen x_1 und x_2 mit 0 initialisiert seien und in den lokalen Caches der Prozessoren P_2 und P_3 aufgehoben werden. Die zugehörigen Speicherblöcke seien als shared (S) markiert.

Die Operationen jedes Prozessors werden in Programmreihenfolge ausgeführt und eine Speicheroperation wird erst nach Abschluss der vorangegangenen Operationen des gleichen Prozessors gestartet. Da über die Laufzeit der Nachrichten über das Netzwerk keine Angaben existieren, ist folgende Abarbeitungsreihenfolge möglich:

- 1) P_1 führt die Schreiboperation (1) auf x_1 aus. Da x_1 nicht im Cache von P_1 liegt, tritt ein Schreib-Fehlzugriff (*write miss*) auf, d. h. es erfolgt ein Zugriff auf den Directory-Eintrag zu x_1 und das Losschicken der Invalidierungsnachrichten an P_2 und P_3 .
- 2) P_2 führt die Leseoperation für (2) auf x_1 aus. Wir nehmen an, dass P_2 die Invalidierungsnachricht von P_1 bereits erhalten und den Speicherblock von x_1 bereits als ungültig (I) markiert hat. Daher tritt ein Lese-Fehlzugriff (*read miss*) auf, d. h. P_2 erhält den aktuellen Wert 1 von x_1 über das Netzwerk von P_1 und die Kopie im Hauptspeicher wird ebenfalls aktualisiert. Nachdem P_2 so den aktuellen Wert von x_1 erhalten und die while-Schleife verlassen hat, führt P_2 die Schreiboperation (3) auf x_2 aus. Dabei tritt wegen der Markierung mit (S) ein Schreib-Fehlzugriff (*write miss*) auf, was zum

Zugriff auf den Directory-Eintrag zu x_2 führt und das Losschicken von Invalidierungsnachrichten an P_1 und P_3 bewirkt.

- 3) P_3 führt die Leseoperation (4) auf x_2 aus und erhält den aktuellen Wert 1 über das Netzwerk, da die Invalidierungsnachricht von P_2 bereits bei P_3 angekommen ist.

Daraufhin führt P_3 die Leseoperation (5) auf x_1 aus und erhält den *alten* Wert 0 für x_1 aus dem lokalen Cache, da die Invalidierungsnachricht von P_1 noch *nicht* angekommen ist.

Das Verhalten bei der Ausführung von Anweisung (5) kann durch unterschiedliche Laufzeiten der Invalidisierungsnachrichten über das Netzwerk ausgelöst werden. Die sequentielle Konsistenz ist verletzt, da die Prozessoren unterschiedliche Schreibreihenfolgen sehen: Prozessor P_2 sieht die Reihenfolge $x_1 = 1$, $x_2 = 1$ und Prozessor P_3 sieht die Reihenfolge $x_2 = 1$, $x_1 = 1$ (da der *neue* Wert von x_2 , aber der alte Wert von x_1 gelesen wird).

Die sequentielle Konsistenz kann in einem parallelen System durch folgende *hinreichenden Bedingungen* sichergestellt werden [32, 43, 163]:

- 1) Jeder Prozessor setzt seine Speicheranfragen in seiner Programmreihenfolge ab (d. h. es sind *keine* sogenannten *out-of-order executions* erlaubt, vgl. Abschn. 2.2).
- 2) Nach dem Absetzen einer Schreiboperation wartet der ausführende Prozessor, bis die Operation abgeschlossen ist, bevor er die nächste Speicheranfrage absetzt. Insbesondere müssen bei Schreiboperationen mit Schreib-Fehlzugriffen alle Cacheblöcke, die den betreffenden Wert enthalten, als ungültig (I) markiert worden sein.
- 3) Nach dem Absetzen einer Leseoperation wartet der ausführende Prozessor, bis diese Leseoperation und die Schreiboperation, deren Wert diese Leseoperation zurückliefert, vollständig abgeschlossen sind und für *alle* anderen Prozessoren sichtbar sind.

Diese Bedingungen stellen keine Anforderungen an die spezielle Zusammenarbeit der Prozessoren, das Verbindungsnetzwerk oder die Speicherorganisation der Prozessoren. In dem obigen Beispiel bewirkt der Punkt 3) der hinreichenden Bedingungen, dass P_2 nach dem Lesen von x_1 wartet, bis die zugehörige Schreiboperation (1) *vollständig* abgeschlossen ist, bevor die nächste Speicherzugriffsoperation (3) abgesetzt wird. Damit liest Prozessor P_3 sowohl für x_1 als auch für x_2 bei beiden Zugriffen (4) und (5) entweder den alten oder den aktuellen Wert, d. h. die sequentielle Konsistenz ist gewährleistet.

Die sequentielle Konsistenz stellt ein für den Programmierer sehr einfaches Modell dar, birgt aber den Nachteil, dass alle Speicheranfragen atomar und nacheinander bearbeitet werden müssen und die Prozessoren dadurch evtl. recht lange auf den Abschluss der abgesetzten Speicheroperationen warten müssen. Zur Behebung der möglicherweise resultierenden Ineffizienzen wurden weniger strikte Konsistenzmodelle vorgeschlagen, die weiterhin ein intuitiv einfaches Modell der Zusammen-

arbeit der Prozessoren liefern, aber effizienter implementiert werden können. Wir geben im Folgenden einen kurzen Überblick und verweisen auf [32, 75] für eine ausführlichere Behandlung.

Abgeschwächte Konsistenzmodelle

Das Modell der sequentiellen Konsistenz verlangt, dass die Lese- und Schreiboperationen, die von einem Prozessor erzeugt werden, die folgende Reihenfolge einhalten:

1. $R \rightarrow R$: Die Lesezugriffe erfolgen in Programmreihenfolge.
2. $R \rightarrow W$: Eine Lese- und eine anschließende Schreiboperation erfolgen in Programmreihenfolge. Handelt es sich um die gleiche Speicheradresse, so ist dies eine *Anti-Abhängigkeit* (engl. *anti-dependence*), in der die Schreiboperation von der Leseoperation abhängt.
3. $W \rightarrow W$: Aufeinanderfolgende Schreibzugriffe erfolgen in Programmreihenfolge. Ist hier die gleiche Speicheradresse angesprochen, so handelt es sich um eine *Ausgabe-Abhängigkeit* (engl. *output dependence*).
4. $W \rightarrow R$: Eine Schreib- und eine anschließende Leseoperation erfolgen in Programmreihenfolge. Bezieht sich dieses auf die gleiche Speicheradresse, so handelt es sich um eine *Fluss-Abhängigkeit* (engl. *true dependence*).

Wenn eine Abhängigkeit zwischen den Lese- und Schreiboperationen besteht, ist die vorgegebene Ausführungsreihenfolge notwendig, um die Semantik des Programmes einzuhalten. Wenn eine solche Abhängigkeit nicht besteht, wird die Ausführungsreihenfolge von dem Modell der sequentiellen Konsistenz verlangt. Abgeschwächte Konsistenzmodelle (engl. *relaxed consistency*) verzichten nun auf einige der oben genannten Reihenfolgen, wenn die Datenabhängigkeiten dies erlauben.

Prozessor-Konsistenzmodelle (engl. *processor consistency*) verzichten auf die Ordnung 4., d.h. auf die Reihenfolge von atomaren Schreib- und Leseoperationen, um so die Latenz der Schreiboperation abzumildern: Obwohl ein Prozessor seine Schreiboperation noch nicht abgeschlossen hat, d.h. der Effekt für andere Prozessoren noch nicht sichtbar ist, kann er nachfolgende Leseoperationen ausführen, wenn es keine Datenabhängigkeiten gibt. Modelle dieser Klasse sind das **TSO-Modell** (*total store ordering*) und das **PC-Modell** (*processor consistency*). Im Unterschied zum TSO-Modell garantiert das PC-Modell *keine* Atomarität der Schreiboperationen. Der Unterschied zwischen sequentieller Konsistenz und dem TSO- oder dem PC-Modell wird im folgenden Beispiel verdeutlicht.

Beispiel

Zwei Prozessoren P_1 und P_2 führen folgende Programmstücke aus, wobei die Variablen x_1 und x_2 jeweils mit 0 initialisiert sind.

Prozessor	P_1	P_2
Programm	(1) $x_1 = 1$; (2) $\text{print}(x_2)$;	(3) $x_2 = 1$; (4) $\text{print}(x_1)$;

Im SC-Modell muss jede mögliche Reihenfolge Anweisung (1) vor Anweisung (2) und Anweisung (3) vor Anweisung (4) ausführen. Dadurch ist die Ausgabe 0 für x_1 und 0 für x_2 *nicht* möglich. Im TSO- und im PC-Modell ist jedoch die Ausgabe von 0 für x_1 und x_2 möglich, da z. B. Anweisung (3) nicht abgeschlossen sein muss, bevor P_1 die Variable x_2 für Anweisung (2) liest.

Partial-Store-Ordering (PSO)-Modelle verzichten auf die Bedingungen 4. und 3. obiger Liste der Reihenfolgebedingungen für das SC-Modell. In diesen Modellen können also auch Schreiboperationen in einer anderen Reihenfolge abgeschlossen werden als die Reihenfolge im Programm angibt, wenn keine Ausgabe-Abhängigkeit zwischen den Schreiboperationen besteht. Aufeinanderfolgende Schreiboperationen können also überlappt werden, was insbesondere beim Auftreten von Schreib-Fehlzugriffen zu einer schnelleren Abarbeitung führen kann. Wieder illustrieren wir den Unterschied zu den bisher vorgestellten Modellen an einem Beispiel.

Beispiel

Die Variablen x_1 und $flag$ seien mit 0 vorbesetzt. Die Prozessoren P_1 und P_2 führen folgende Programmstücke aus.

Prozessor	P_1	P_2
Programm	(1) $x_1 = 1$; (2) $flag = 1$;	(3) $\text{while}(flag == 0)$; (4) $\text{print}(x_1)$;

Im SC- und im PC- bzw. TSO-Modell ist die Ausgabe des Wertes 0 für x_1 *nicht* möglich. Im PSO-Modell kann die Schreiboperation (2) jedoch *vor* Schreiboperation (1) beendet sein und so die Ausgabe von 0 durch die Leseoperation in (4) ermöglichen. Diese Ausgabe stimmt nicht unbedingt mit dem intuitiven Verständnis der Arbeitsweise des Programmstückes überein.

Weak-Ordering-Modelle verzichten zusätzlich auf die Bedingungen (1) und (2), garantieren also keinerlei Fertigstellungsreihenfolge der Operationen. Es werden aber zusätzlich Synchronisationsoperationen bereitgestellt, die sicherstellen, dass

- a) alle Lese- und Schreiboperationen, die in der Programmreihenfolge *vor* der Synchronisationsoperation liegen, fertiggestellt werden, bevor die Synchronisationsoperation ausgeführt wird, und dass

- b) eine Synchronisationsoperation fertiggestellt wird, bevor Lese- und Schreiboperationen ausgeführt werden, die in der Programmreihenfolge *nach* der Synchronisationsoperation stehen.

Die zunehmende Verbreitung von Parallelrechnern hat dazu geführt, dass viele moderne Mikroprozessoren zur Vereinfachung der Integration in Parallelrechner Unterstützung für die Realisierung eines Speicherkonsistenzmodells bereitstellen. Unterschiedliche Hardwarehersteller unterstützen dabei unterschiedliche Speicherkonsistenzmodelle, d. h. es hat sich zzt. noch keine eindeutige Meinung durchgesetzt, welches der vorgestellten Konsistenzmodelle das beste ist. Sequentielle Konsistenz wird z. B. von SGI im MIPS R10000-Prozessor dadurch unterstützt, dass die Operationen eines Programmes in der Programmreihenfolge fertiggestellt werden, auch wenn in jedem Zyklus mehrere Maschinenbefehle an die Funktionseinheiten abgesetzt werden können. Die Intel Pentium Prozessoren unterstützen ein PC-Modell. Die SPARC-Prozessoren von Sun verwenden das TSO-Modell. Die Alpha-Prozessoren von DEC und die PowerPC-Prozessoren von IBM verwenden ein Weak-Ordering-Modell. Die von den Prozessoren unterstützten Speicherkonsistenzmodelle werden meistens auch von den Parallelrechnern verwendet, die diese Prozessoren als Knoten benutzen. Dies ist z. B. für die Sequent NUMA-Q der Fall, die Pentium-Pro-Prozessoren als Knoten verwenden.

2.8 Parallelität auf Threadebene

Parallelität auf Threadebene kann innerhalb eines Prozessorchips durch geeignete Architekturorganisation realisiert werden. Man spricht in diesem Fall von Threadparallelität auf Chipebene (engl. *Chip Multiprocessing*, CMP). Eine Möglichkeit CMP zu realisieren besteht darin, mehrere **Prozessorkerne** (engl. *execution cores*) mit allen Ausführungsressourcen dupliziert auf einen Prozessorchip zu integrieren. Die dadurch resultierenden Prozessoren werden auch als **Multicore-Prozessoren** bezeichnet, siehe Abschn. 2.1.

Ein alternativer Ansatz besteht darin, mehrere Threads dadurch gleichzeitig auf einem Prozessor zur Ausführung zu bringen, dass der Prozessor je nach Bedarf per Hardware zwischen den zur Verfügung stehenden ausführungsbereiten Threads umschaltet. Dies kann auf verschiedene Weise geschehen [116] und wird auch in GPU-Architekturen eingesetzt, vgl. Kap. 7. Der Prozessor kann nach fest vorgegebenen Zeitintervallen zwischen den Threads umschalten, d. h. nach Ablauf eines Zeitintervalls wird der nächste Thread zur Ausführung gebracht. Man spricht in diesem Fall von **Zeitscheiben-Multithreading** (engl. *timeslice multithreading*). Zeitscheiben-Multithreading kann dazu führen, dass Zeitscheiben nicht effektiv genutzt werden, wenn z. B. ein Thread auf das Eintreten eines Ereignisses warten muss, bevor seine Zeitscheibe abgelaufen ist, so dass der Prozessor für den Rest der Zeitscheibe keine Berechnungen durchführen kann. Solche unnötigen Wartezeiten können durch

den Einsatz von **ereignisbasiertem Multithreading** (engl. *switch-on-event multithreading*) vermieden werden. In diesem Fall kann der Prozessor beim Eintreten von Ereignissen mit langer Wartezeit, wie z. B. bei Cache-Fehlzugriffen, zu einem anderen ausführungsbereiten Thread umschalten. Ein weiterer Ansatz ist das **simultane Multithreading** (engl. *simultaneous multithreading*, SMT), bei dem mehrere Threads ohne explizites Umschalten ausgeführt werden. Wir gehen im folgenden Abschnitt auf diese Methode, die als **Hyperthreading-Technik** in Prozessoren zum Einsatz kommt, näher ein.

2.8.1 Hyperthreading-Technik

Die Hyperthreading-Technologie basiert auf dem Duplizieren des Prozessorbereiches zur Ablage eines Prozessorzustandes auf der Chipfläche des Prozessors. Dazu gehören die Benutzer- und Kontrollregister sowie der Interrupt-Controller mit den zugehörigen Registern. Damit erscheint der physikalische Prozessor aus der Sicht des Betriebssystems und des Benutzerprogramms als eine Ansammlung von **logischen Prozessoren**, denen Prozesse oder Threads zur Ausführung zugeordnet werden können. Diese können von einem oder mehreren Anwendungsprogrammen stammen.

Jeder logische Prozessor legt seinen Prozessorzustand in einem separaten Prozessorbereich ab, so dass beim Wechsel zu einem anderen Thread kein aufwendiges Zwischenspeichern des Prozessorzustandes im Speichersystem erforderlich ist. Die logischen Prozessoren teilen sich fast alle Ressourcen des physikalischen Prozessors, wie Caches, Funktions- und Kontrolleinheiten und Bussystem. Die Realisierung der Hyperthreading-Technologie erfordert daher nur eine geringfügige Vergrößerung der Chipfläche. Für zwei logische Prozessoren wächst z. B. für einen Intel Xeon Prozessor die erforderliche Chipfläche um weniger als 5 % [116, 185]. Die gemeinsamen Ressourcen des Prozessorchips werden den logischen Prozessoren reihum zugeteilt, so dass die logischen Prozessoren simultan zur Ausführung gelangen. Treten bei einem logischen Prozessor Wartezeiten auf, können die Ausführungs-Ressourcen den anderen logischen Prozessoren zugeordnet werden, so dass aus der Sicht des physikalischen Prozessors eine fortlaufende Nutzung der Ressourcen gewährleistet ist.

Gründe für Wartezeiten eines logischen Prozessors können z. B. Cache-Fehlzugriffe, falsche Sprungvorhersage, Abhängigkeiten zwischen Instruktionen oder Pipeline-Hazards sein. Da auch der Instruktionscache von den logischen Prozessoren geteilt wird, enthält dieser Instruktionen mehrerer logischer Prozessoren. Versuchen logische Prozessoren gleichzeitig eine Instruktion aus dem Instruktionscache in ihr lokales Instruktionsregister zur Weiterverarbeitung zu laden, erhält einer von ihnen per Hardware eine Zugriffserlaubnis. Sollte auch im nächsten Zyklus wieder eine konkurrierende Zugriffsanfrage erfolgen, erhält ein anderer

logischer Prozessor eine Zugriffserlaubnis, so dass alle logischen Prozessoren mit Instruktionen versorgt werden.

Untersuchungen zeigen, dass durch die fortlaufende Nutzung der Ressourcen durch zwei logische Prozessoren je nach Anwendungsprogramm Laufzeitverbesserungen zwischen 15 % und 30 % erreicht werden [116]. Da alle Berechnungsressourcen von den logischen Prozessoren geteilt werden, ist nicht zu erwarten, dass sich der Einsatz einer sehr großen Anzahl von logischen Prozessoren lohnt und eine entsprechende Laufzeitverbesserung erreicht werden kann. Die Anzahl der unterstützten logischen Prozessoren wird daher voraussichtlich auf einige wenige (zwischen zwei und acht) beschränkt bleiben. Zum Erreichen einer Laufzeitverbesserung durch den Einsatz der Hyperthreading-Technologie ist es erforderlich, dass das Betriebssystem in der Lage ist, die logischen Prozessoren anzusteuern. Aus Sicht eines Anwendungsprogramms ist es erforderlich, dass für jeden logischen Prozessor ein separater Thread zur Ausführung bereitsteht, d. h. für die Implementierung des Programms müssen Techniken der parallelen Programmierung eingesetzt werden.

2.8.2 Multicore-Prozessoren

Nach dem Gesetz von Moore verdoppelt sich die Anzahl der Transistoren pro Prozessorchip alle 18–24 Monate. Dieser enorme Zuwachs macht es seit vielen Jahren möglich, die Leistung der Prozessoren so stark zu erhöhen, dass ein Rechner spätestens nach 5 Jahren als veraltet gilt und die Kunden in relativ kurzen Abständen einen neuen Rechner kaufen. Die Hardwarehersteller sind daher daran interessiert, die Leistungssteigerung der Prozessoren mit der bisherigen Geschwindigkeit beizubehalten, um einen Einbruch der Verkaufszahlen zu vermeiden.

Wie in Abschn. 2.1 dargestellt, waren wesentliche Aspekte der Leistungssteigerung zum einen die Erhöhung der Taktrate des Prozessors und zum anderen der interne Einsatz paralleler Abarbeitung von Instruktionen, z. B. durch das Duplizieren von Funktionseinheiten. Dabei verursachte die Erhöhung der Taktrate das Problem, dass die Leistungsaufnahme zu stark stieg. Aber auch Effizienzprobleme beim Speicherzugriff limitieren eine weitere Erhöhung der Taktrate, da die Speicherzugriffsgeschwindigkeit nicht im gleichen Umfang wie die Prozessorgeschwindigkeit zugenommen hat, was zu einer Erhöhung der Zyklenanzahl pro Speicherzugriff führte. Die Speicherzugriffszeiten entwickelten sich daher zum limitierenden Faktor für eine weitere Leistungssteigerung. So brauchte z. B. um 1990 ein Intel i486 für einen Zugriff auf den Hauptspeicher zwischen 6 und 8 Maschinenzyklen, während 2012 ein Intel Core i7 Prozessor der Sandy-Bridge-Architektur ca. 180 Zyklen benötigt. Für die Zukunft ist zu erwarten, dass sich die Anzahl der für einen Speicherzugriff benötigten Zyklen nicht wesentlich ändern wird, da – wie unten beschrieben wird – die Taktfrequenz der Prozessoren nicht mehr wesentlich angehoben werden kann.

Die Grenzen beider in der Vergangenheit zur Verbesserung der Leistung der Prozessoren eingesetzten Techniken wurden jedoch ab ca. 2005 erreicht: Ein weiteres Duplizieren von Funktionseinheiten ist zwar möglich, bringt aber aufgrund vorhandener Abhängigkeiten zwischen Instruktionen kaum eine weitere Leistungssteigerung. Gegen eine weitere Erhöhung der Taktrate sprechen vor allem thermische Gründe [104]: Die oben angesprochene Erhöhung der Transistoranzahl wird auch durch eine Erhöhung der Packungsdichte erreicht, mit der aber auch eine Erhöhung der Wärmeentwicklung verbunden ist, die durch Leckströme verursacht wird. Diese fallen auch dann an, wenn der Prozessor keine Berechnungen durchführt; daher bezeichnet man den dabei verursachten Energieverbrauch auch als *statischen Energieverbrauch* im Gegensatz zu dem durch Berechnungen entstehenden *dynamischen Energieverbrauch*. Der Anteil des statischen Energieverbrauchs am Gesamtenergieverbrauch lag 2011 je nach Prozessor typischerweise zwischen 25 % und 50 % [76]. Die dabei entstehende Wärmeentwicklung wird zunehmend zum Problem, da die notwendige Kühlung entsprechend aufwendiger wird.

Zusätzlich werden auch durch eine Erhöhung der Taktfrequenz vermehrt Leckströme verursacht, die ebenfalls zu einem erhöhten Stromverbrauch führen und den Prozessorchip erwärmen. Modelle zur Erfassung dieses Phänomens gehen davon aus, dass der dynamische Stromverbrauch eines Prozessors proportional zu $V^2 \cdot f$ ist, wobei V die Versorgungsspannung und f die Taktfrequenz des Prozessors ist, vgl. z. B. [98]. Dabei ist V auch abhängig von f , so dass eine kubische Abhängigkeit des Stromverbrauchs von der Taktfrequenz resultiert. Damit steigt der Stromverbrauch mit steigender Taktfrequenz überproportional an. Dies ist auch an der Prozessorentwicklung zu beobachten: die ersten 32-Bit Mikroprozessoren hatten einen Stromverbrauch von etwa 2 Watt, ein 3,3 GHz Intel Core i7 (Nehalem) verbraucht hingegen etwa 130 Watt [76]. Eine weitere Steigerung der Taktfrequenz über 3,3 GHz hinaus würde einen noch wesentlich höheren Stromverbrauch verursachen. Dadurch ist auch zu erklären, dass sich die Taktraten von Desktop-Prozessoren seit 2003 nicht wesentlich erhöht haben.

Zur Einschränkung des Energieverbrauchs werden von modernen Mikroprozessoren mehrere Techniken eingesetzt. Dazu gehören neben dem Abschalten inaktiver Prozessorteile insbesondere auch Techniken zur dynamischen Anpassung der Spannung und der Frequenz; dies wird als DVFS (*Dynamic Voltage-Frequency Scaling*) bezeichnet. Die Idee dieser Techniken besteht darin, für Zeiten niedriger Aktivität die Taktfrequenz zu reduzieren, um Energie zu sparen, und die Frequenz für Zeiten hoher Aktivität wieder zu erhöhen. Mit der Erhöhung der Frequenz sinkt die Zykluszeit des Prozessors, d. h. in der gleichen Zeit können bei höherer Frequenz mehr Berechnungen durch Ausführung von Instruktionen durchgeführt werden als bei niedrigerer Frequenz. So kann z. B. für einen Intel Core i7 Prozessor (Sandy Bridge) die Taktfrequenz zwischen 2,7 GHz und 3,7 GHz in 100 MHz-Schritten variiert werden. Die Anpassung kann durch das Betriebssystem entsprechend der Prozessorbelastung vorgenommen werden. Werkzeuge wie `cpufreq_set` gestatten auch den Anwendungsprogrammierern eine Steuerung der Taktfrequenz. Für einige Prozessoren kann für Zeiten sehr hoher Aktivität sogar ein Turbomodus

eingeschaltet werden, in dem die Taktfrequenz kurzzeitig über die maximale Taktfrequenz hinaus erhöht wird (engl. *overclocking*), um die anfallenden Berechnungen besonders schnell abzuarbeiten, so dass danach wieder eine niedrigere Frequenz verwendet werden kann.

Aus den oben genannten Gründen ist eine weiterhin signifikante Leistungssteigerung mit den bisherigen Technologien nicht durchführbar. Stattdessen müssen neue Prozessorarchitekturen eingesetzt werden, wobei die Verwendung mehrerer Prozessorkerne auf einem Prozessorchip schon seit vielen Jahren als die vielversprechendste Technik angesehen wird. Man spricht in diesem Zusammenhang auch von Multicore-Prozessoren oder Mehrkern-Prozessoren. Die Idee besteht darin, anstatt eines Prozessorchips mit einer sehr komplexen internen Organisation mehrere Prozessorkerne mit einfacherer Organisation auf den Prozessorchip zu integrieren. Dies hat auch den Vorteil, dass der Stromverbrauch des Prozessorchips dadurch reduziert werden kann, dass vorübergehend ungenutzte Prozessorkerne abgeschaltet werden können [74].

Bei Multicore-Prozessoren werden also mehrere Prozessorkerne auf einem Prozessorchip integriert. Jeder Prozessorkern stellt für das Betriebssystem einen separaten logischen Prozessor mit separaten Ausführungsressourcen dar, die getrennt angesteuert werden müssen. Das Betriebssystem kann damit verschiedene Anwendungsprogramme parallel zueinander zur Ausführung bringen. So kann z. B. eine Anzahl von Hintergrundanwendungen, wie Viruserkennung, Verschlüsselung und Kompression, parallel zu Anwendungsprogrammen des Nutzers ausgeführt werden [146]. Es ist aber mit Techniken der parallelen Programmierung auch möglich, ein einzelnes rechenzeitintensives Anwendungsprogramm (z. B. aus den Bereichen Computerspiele, Bildverarbeitung oder naturwissenschaftliche Simulationsprogramme) auf mehreren Prozessorkernen parallel abzuarbeiten, so dass die Berechnungszeit im Vergleich zu einer Ausführung auf nur einem Prozessorkern reduziert werden kann. Man kann davon ausgehen, dass heutzutage die Nutzer von Standardprogrammen, z. B. aus dem Bereich der Computerspiele erwarten, dass diese die Berechnungsressourcen des Prozessorchips effizient ausnutzen, d. h. für die Implementierung der zugehörigen Programme müssen Techniken der parallelen Programmierung eingesetzt werden.

2.8.3 Designvarianten für Multicore-Prozessoren

Zur Realisierung von Multicore-Prozessoren gibt es verschiedene Designvarianten, die sich in der Anzahl der Prozessorkerne, der Größe und Anordnung der Caches, den Zugriffsmöglichkeiten der Prozessorkerne auf die Caches und dem Einsatz von heterogenen Komponenten unterscheiden. Für die interne Organisation der Prozessorchips können verschiedene Entwürfe unterschieden werden, die verschiedene Anordnungen der Prozessorkerne und der Cachespeicher verwenden [105]. Dabei können grob drei unterschiedliche Architekturen unterschieden werden, siehe

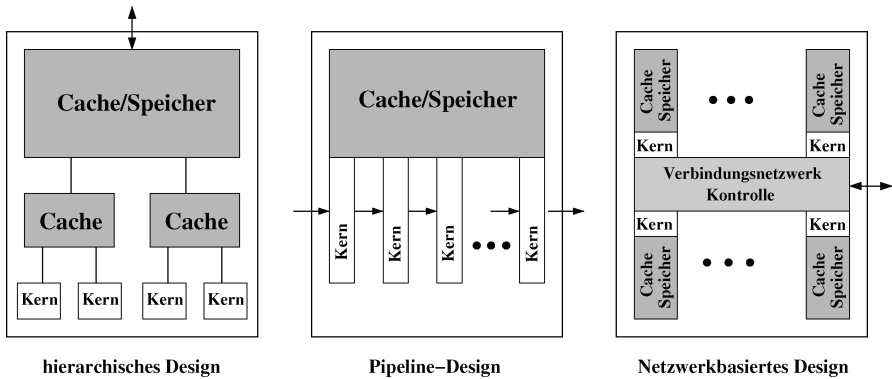


Abb. 2.34 Designmöglichkeiten für Multicore-Prozessoren nach [105]

Abb. 2.34, von denen auch Mischformen auftreten können. Wir werden diese Designvarianten im Folgenden kurz besprechen.

Hierarchisches Design

Bei einem *hierarchischen Design* teilen sich mehrere Prozessorkerne mehrere Caches, die in einer baumartigen Konfiguration angeordnet sind, wobei die Größe der Caches von den Blättern zur Wurzel steigt. Die Wurzel repräsentiert die Verbindung zum Hauptspeicher. So kann z. B. jeder Prozessorkern einen separaten L1-Cache haben, sich aber mit anderen Prozessorkernen einen L2-Cache teilen, und alle Prozessorkerne können auf den externen Hauptspeicher zugreifen. Dies ergibt dann eine dreistufige Hierarchie. Dieses Konzept kann auf mehrere Stufen erweitert werden und ist in Abb. 2.34 (links) für drei Stufen veranschaulicht. Zusätzliche Untersysteme können die Caches einer Stufe miteinander verbinden. Ein hierarchisches Design wird typischerweise für im Desktopbereich eingesetzte Multicore-Prozessoren mit einer kleinen Anzahl von Prozessorkernen verwendet. Ein Beispiel für ein hierarchisches Design ist der Intel Core i7 Quadcore-Prozessor, der vier unabhängige superskalare Prozessorkerne enthält, von denen jeder per Hyperthreading zwei logische Prozessorkerne simulieren kann. Jeder der physikalischen Prozessorkerne hat einen separaten L1-Cache (für Instruktionen und Daten getrennt), einen separaten L2-Cache und einen mit den anderen Prozessorkernen gemeinsam genutzten L3-Cache. Eine genaue Beschreibung der Architektur des Core i7 erfolgt in Abschn. 2.8.4. Andere Prozessoren mit hierarchischem Design sind die IBM Power7 Prozessoren mit maximal acht Kernen pro Chip, wobei jeder Kern per Hyperthreading vier Threads simulieren kann, sowie die AMD Opteron Prozessoren mit sechs Kernen pro Chip, aber ohne Hyperthreading.

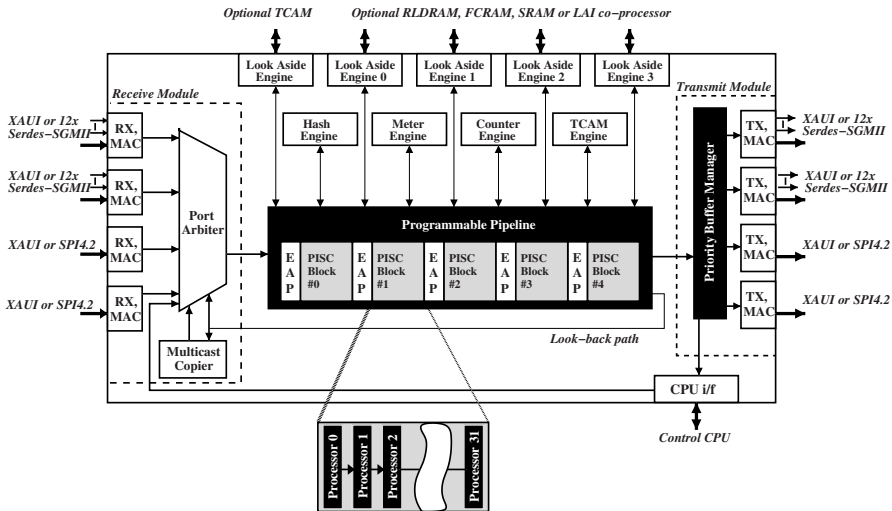


Abb. 2.35 Xelerator X11 Network Processor als Beispiel für ein Pipeline-Design [183]

Pipeline-Design

Bei einem *Pipeline-Design* werden die Daten schrittweise durch mehrere Prozessorkerne weiterverarbeitet, bis sie vom letzten Prozessorkern im Speichersystem abgelegt werden, vgl. Abb. 2.34 (Mitte). Router-Prozessoren und Grafikchips arbeiten oft nach diesem Prinzip. Ein Beispiel sind die X10 und X11 Prozessoren von Xelerator zur Verarbeitung von Netzwerkpaketen [183, 105]. Der Xelerator X10q enthält z. B. 200 separate VLIW-Prozessorkerne, die in einer logischen linearen Pipeline miteinander verbunden sind. Die Pakete werden dem Prozessor über mehrere Netzwerkschnittstellen zugeführt und dann durch die Prozessorkerne schrittweise verarbeitet, wobei jeder Prozessorkern einen Schritt ausführt. Die X11 Netzwerkprozessoren haben bis zu 800 Pipeline-Prozessorkerne, die in einer logischen linearen Pipeline angeordnet sind, vgl. Abb. 2.35.

Netzwerkbasiertes Design

Bei einem *netzwerkbasierten Design* sind die Prozessorkerne und ihre lokalen Caches oder Speicher über ein Verbindungsnetzwerk mit den anderen Prozessorkernen des Chips verbunden, so dass der gesamte Datentransfer zwischen den Prozessorkernen über das Verbindungsnetzwerk läuft, vgl. Abb. 2.34 (rechts). Eine Darstellung möglicher Verbindungsnetzwerk wird in Abschn. 2.5 gegeben. Ein Beispiel für ein netzwerkbasiertes Design ist der Sun Ultra SPARC T4 Prozessor mit acht physikalischen Kernen, von denen jeder per Hyperthreading acht Threads parallel

zueinander abarbeiten kann. Die Kerne sind über eine Crossbar-Verbindung zusammengeschaltet, die jeden Kern direkt mit jedem anderen Kern verbindet. Netzwerkorientierte Entwürfe wurden auch für Forschungschips verwendet. Beispiele sind der Intel Teraflops Forschungschip sowie der Intel SCC (Single-chip Cloud Computer) Prozessor, der von Intel als experimenteller Chip zur Untersuchung der Skalierbarkeit von Multicore-Prozessoren entworfen und in kleiner Serie gefertigt wurde.

Weitere Entwicklung

Das Potential der Multicore-Prozessoren wurde von Hardwareherstellern wie Intel und AMD erkannt und seit 2005 bieten viele Hardwarehersteller Prozessoren mit zwei oder mehr Kernen an. Seit Ende 2006 wurden von Intel Quadcore-Prozessoren ausgeliefert, ab 2010 waren die ersten Octocore-Prozessoren verfügbar. Der seit 2012 ausgelieferte SPARC T4 Prozessor von Sun/Oracle hat acht Prozessorkerne, von denen jeder durch Einsatz von simultanem Multithreading acht Threads simultan verarbeiten kann. Damit kann ein T4-Prozessor bis zu 64 Threads simultan ausführen. Der für die BG/Q-Systeme verwendete Prozessor hat 16 Prozessorkerne, vgl. Abschn. 2.9.

Intel untersucht im Rahmen des *Tera-scale Computing Program* die Herausforderungen bei der Herstellung und Programmierung von Prozessoren mit Dutzenden von Prozessorkernen [73]. Diese Initiative beinhaltet auch die Entwicklung eines Teraflops-Prozessors, der 80 Prozessorkerne enthält, die als 8×10 -Gitter angeordnet sind. Jeder Prozessorkern kann Fließkomma-Operationen verarbeiten und enthält neben einem lokalen Cachespeicher auch einen Router zur Realisierung des Datentransfers zwischen den Prozessorkernen und dem Hauptspeicher. Zusätzlich kann ein solcher Prozessor spezialisierte Prozessorkerne für die Verarbeitung von Videodaten, graphischen Berechnungen und zur Verschlüsselung von Daten enthalten. Je nach Einsatzgebiet kann die Anzahl der spezialisierten Prozessorkerne variiert werden.

Ein wesentlicher Bestandteil eines Prozessors mit einer Vielzahl von Prozessorkernen ist ein effizientes Verbindungsnetzwerk auf dem Prozessorchip, das an eine variable Anzahl von Prozessorkernen angepasst werden kann, den Ausfall einzelner Prozessorkerne toleriert und bei Bedarf das Abschalten einzelner Prozessorkerne erlaubt, falls diese für die aktuelle Anwendung nicht benötigt werden. Ein solches Abschalten ist insbesondere zur Reduktion des Stromverbrauchs sinnvoll.

Für eine effiziente Nutzung der Prozessorkerne ist entscheidend, dass die zu verarbeitenden Daten schnell genug zu den Prozessorkernen transportiert werden können, so dass diese nicht auf die Daten warten müssen. Dazu sind ein leistungsfähiges Speichersystem und I/O-System erforderlich. Das Speichersystem nutzt private L1-Caches, auf die nur von jeweils einem Prozessorkern zugegriffen wird, sowie gemeinsame, evtl. aus mehreren Stufen bestehende L2-Caches, die Daten verschiedener Prozessorkerne enthalten. Für einen Prozessorchip mit Dutzenden

von Prozessorkernen muss voraussichtlich eine weitere Stufe im Speichersystem eingesetzt werden [73]. Das I/O-System muss in der Lage sein, Hunderte von Gigabytes pro Sekunde auf den Prozessorchip zu bringen. Hier arbeitet z. B. Intel an der Entwicklung geeigneter Systeme.

2.8.4 Beispiel: Architektur des Intel Core i7

Als Beispiel für einen Multicore-Prozessor für Desktoprechner betrachten wir im Folgenden den Intel Core i7 Prozessor. Die Core 13, i5 und i7 Prozessoren wurden im Jahr 2008 als Nachfolger der Core-2-Familie eingeführt. Wir verweisen auf die ausführlichere Darstellung in [76, 97]. Der Core i7 unterstützt die Intel x86-64 Architektur, eine 64-Bit-Erweiterung der für die Pentium-Prozessoren verwendeten x86 Architektur. Seit dem Jahr 2011 basieren die Core i7 Prozessoren auf der Sandy-Bridge-Mikroarchitektur, die neben den Prozessorkernen und der Cache-hierarchie auch eine graphische Berechnungseinheit, einen Speicher-Controller und einen PCI-Expressbus-Controller auf den Prozessorchip integriert.

Ein Core i7 Prozessor besteht aus zwei oder vier Prozessorkernen, wobei jeder Prozessorkern per Hyperthreading zwei simultane Threads unterstützt. Eine Illustration des internen Aufbaus eines einzelnen Prozessorkerns ist in Abb. 2.36 wiedergegeben. Jeder Prozessorkern kann pro Maschinenzyklus bis zu vier x86-Instruktionen ausführen. Für die Bereitstellung ausführungsbereiter Instruktionen ist eine Instruktionseinheit (*instruction fetch unit*) verantwortlich. Da der Instruktionsfluss häufig von Sprüngen unterbrochen wird, versucht die Instruktionseinheit mit Hilfe einer Sprungvorhersage (*branch prediction*) das voraussichtliche Sprungziel zu ermitteln. Die Sprungvorhersage verwendet einen Sprungzielpuffer (*branch target buffer*), in dem vorausgegangene Sprungziele abgelegt werden. Auf der Basis der vorhergesagten Adresse lädt die Instruktionseinheit 16 Bytes in einen Vor-Dekodierpuffer (*predecode instruction buffer*). Das Laden geschieht aus dem 32 KB großen L1-Instruktioncache (Blockgröße 64 Bytes, 8-Wege-assoziativ), es wird aber auch der unten besprochene μ op-Cache berücksichtigt.

Ein Vor-Dekodierer zerlegt die 16 Bytes im Vor-Dekodierpuffer in x86-Instruktionen, die dann in einer Instruktionsschlange (*instruction queue*) mit 18 Einträgen abgelegt werden. Vor ihrer Ausführung müssen die x86-Instruktionen (variabler Länge und stark unterschiedlicher Komplexität) in Mikrobefehle konstanter Länge (*micro-ops*) umgewandelt werden. Dazu stehen vier Dekodiereinheiten zur Verfügung: drei dieser Dekodiereinheiten behandeln x86-Instruktionen, die direkt in Mikrobefehle umgewandelt werden können, eine Dekodiereinheit ist für die Umwandlung komplexerer x86-Instruktionen verantwortlich, für die eine Folge von Mikrobefehlen erzeugt werden muss. Die Dekodierer können bis zu vier Mikrobefehle (sogenannte μ ops) pro Zyklus absetzen. Die dekodierten Mikrobefehle werden in einem μ op-Cache abgelegt, der im Prinzip einen Teil des Instruktionscaches in dekodierter Form enthält.

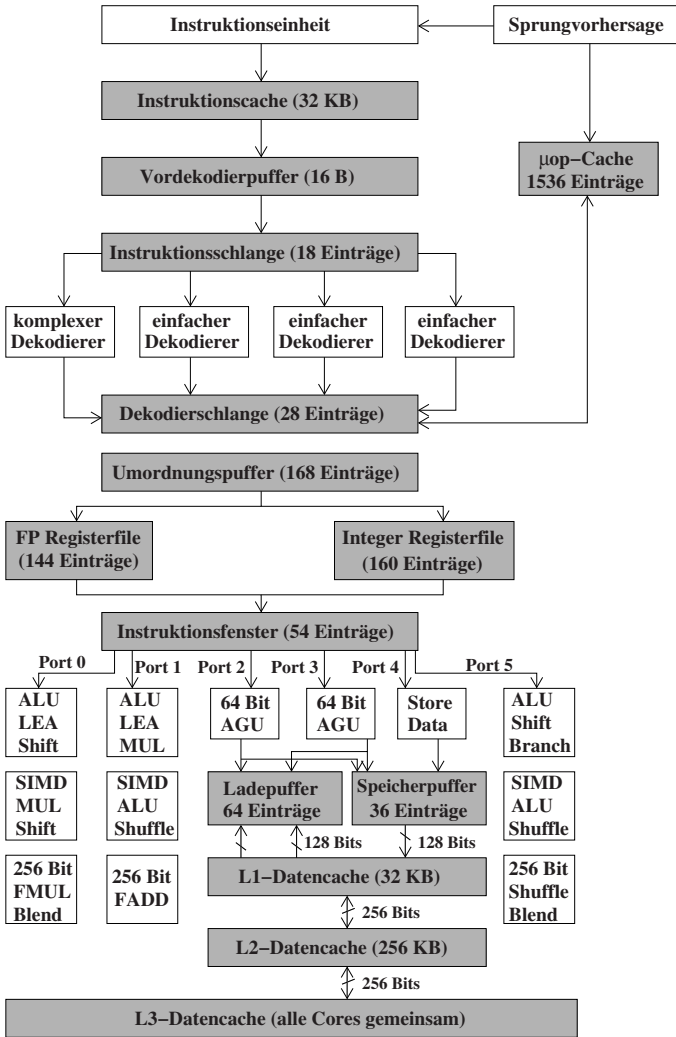


Abb. 2.36 Blockdiagramm zur Veranschaulichung der Architektur eines Prozessorkerns des Intel Core i7 Prozessors (Sandy Bridge)

Der μop -Cache fasst 1536 μops und ist 8-Wege-assoziativ, vgl. Abschn. 2.7. Jeder Cacheblock des μop -Caches umfasst sechs Mikrobefehle. Wird die vorhergesagte Adresse der nächsten auszuführenden Instruktion im μop -Cache gefunden (μop -Cache hit), werden maximal drei Cacheblöcke mit benachbarten Instruktionen aus dem μop -Cache in einen speziellen Puffer geladen. In diesem Fall wird die übliche Fetch-und-Dekodier-Hardware mit den oben erwähnten vier Dekodierern umgangen, so dass diese weitere dekodierte Instruktionen für den μop -Cache be-

reitstellen kann. Die von den Dekodierern oder aus dem μ op-Cache bereitgestellten Mikrobefehle werden in einer 28 Einträge fassenden Dekodierschlange abgegeben, die z. B. alle Instruktionen kleiner Schleifen abspeichern kann. Passen die Instruktionen einer kleinen Schleife vollständig in die Dekodierschlange, brauchen die Dekodierer während der Abarbeitung der Schleife keine Instruktionen zu laden und zu dekodieren, was den Energieverbrauch reduzieren kann. Die verwendete Registerumbenennungstechnik basiert auf einem physikalischen Registerfile, in dem 160 64-Bit-Integerwerte abgelegt werden können. Für Floating-Point-Werte stehen zusätzlich 144 Einträge mit je 256 Bits zur Verfügung, so dass letztere auch die Verarbeitung von SIMD AVX-Instruktionen unterstützt, vgl. Abschn. 3.6. Zur Zuordnung der Instruktionen an Funktionseinheiten wird ein zentrales Instruktionsfenster mit 54 Einträgen verwendet. Wenn eine Instruktion ausführungsbereit ist, wird sie einer passenden Funktionseinheit zugeordnet. Dabei können pro Zyklus bis zu sechs Instruktionen an Funktionseinheiten abgesetzt werden, und bis zu vier Instruktionen können pro Zyklus fertiggestellt werden.

Die eigentliche Ausführung der Instruktionen erfolgt durch die Funktionseinheiten, die über drei Ausführungsports (Port 0, Port 1, Port 5) angesteuert werden. Jeder der Ausführungsports unterstützt Funktionseinheiten für drei verschiedene Typen von Instruktionen: Integer-Instruktionen, SIMD Integer-Instruktionen und Floating-Point-Instruktionen (entweder skalar oder SIMD). Nach der Ausführung einer Instruktion durch die zugehörige Funktionseinheit wird bei Berechnung eines Wertes das angegebene Zielregister im Registerfile aktualisiert. Neben den drei Ausführungsports gibt es drei Ports für die Speicheranbindung (Port 2, Port 3, Port 4). Dabei stehen an Port 2 und 3 jeweils eine allgemeine Adress-Erzeugungseinheit (*Address Generation Unit*, AGU) zur Verfügung, die für das Laden und Speichern verwendet werden können. Zusätzlich steht an Port 4 eine Einheit zum Speichern von Daten zur Verfügung. Zur Ablage von geladenen oder zu speichernden Werten wird ein Ladepuffer (*load buffer*) mit 64 Einträgen bzw. ein Speicherpuffer (*store buffer*) mit 36 Einträgen verwendet.

Das Speichersystem verwendet neben dem Hauptspeicher eine dreistufige Cachehierarchie, bestehend aus L1-Cache, L2-Cache und L3-Cache. Dabei hat jeder physikalische Prozessorkern einen eigenen L1- und L2-Cache, der L3-Cache wird von allen Kernen eines Prozessors gemeinsam genutzt. Der L1-Datencache ist 32 KB groß und 8-Wege-assoziativ. Die Blockgröße umfasst 64 Bytes und es wird eine write-back-Rückschreibstrategie verwendet. Die Latenzzeit beträgt vier Zyklen für Integerwerte und 5-6 Zyklen für Floating-Point- oder SIMD-Werte. In jedem Zyklus können maximal 128 Bit aus dem L1-Cache geladen und maximal 128 Bit in den L1-Cache gespeichert werden.

Der L2-Cache ist 256 KB groß, 8-Wege-assoziativ und verwendet eine write-back-Rückschreibstrategie. Die Latenzzeit beträgt 12 Zyklen. Der L3-Cache wird von allen Kernen eines Prozessors sowie dessen Graphikeinheit gemeinsam genutzt. Diese Einheiten greifen auf den L3-Cache über ein spezielles Ringnetzwerk zu, das aus vier Unterringen für Anfrage (*request*), Bestätigung (*acknowledge*), Cachekohärenz (*snoop*) und Daten (32 Bytes) besteht. Die Größe des L3-Caches

Tab. 2.2 Zusammenfassung der Eigenschaften der Cache-Hierarchie des Intel Core i7 Prozessors (Sandy Bridge)

Eigenschaft	L1	L2	L3
Größe	32 KB Instruktionen 32 KB Daten	256 KB	2 MB pro Kern
Assoziativität	4-Wege Instruktionen 8-Wege Daten	8-Wege	16-Wege
Zugriffslatenz	4–6 Zyklen	12 Zyklen	26–31 Zyklen
Rückschreibestrategie	write-back	write-back	write-back

ist skalierbar und hängt von der Prozessorversion ab. Eine typische Größe liegt 2012 bei 2 MB pro Prozessorkern. Der L3-Cache ist 16-Wege-assoziativ und die Latenzzeit liegt zwischen 26 und 31 Zyklen. Tabelle 2.2 fasst die wichtigsten Eigenschaften des Speichersystems noch einmal zusammen. Eine detaillierte Analyse der Leistung der Core-i7-Architektur und deren Speichersystem wird in [76] vorgestellt.

2.9 Beispiel: IBM Blue Gene Supercomputer

Derzeit aktuelle Beispiele großer paralleler Systeme sind die IBM Blue Gene Supercomputer, deren Aufbau wir im Folgenden kurz beschreiben. Das Entwurfsziel der IBM Blue Gene Supercomputer besteht darin, ein System mit einem möglichst geringen Stromverbrauch und einer hohen Performance pro Watt, gemessen z. B. in Anzahl der Floating-Point-Operationen pro Sekunde (FLOPS), die pro Watt berechnet werden können, zur Verfügung zu stellen. Die Blue Gene (BG) Supercomputer wurden 2004 als Blue Gene/L (BG/L) Systeme eingeführt und waren, wie der Name andeutet, ursprünglich für wissenschaftliche Berechnungen aus dem Life-Science-Bereich, wie z. B. Proteinfaltung oder Molekulardynamik-Simulationen, vorgesehen, werden aber mittlerweile auch für eine Vielzahl anderer Berechnungen aus dem wissenschaftlich-technischen Bereich eingesetzt. Im Jahr 2007 folgten als Nachfolger der BG/L-Systeme die BG/P-Systeme, die in [88] detailliert beschrieben werden. Im Jahr 2012 wurden die BG/Q-Systeme eingeführt.

Der angestrebte geringe Stromverbrauch wird in den BG-Systemen insbesondere durch eine niedrige Taktrate erreicht, die zu geringeren Leckströmen als hohe Taktfrequenzen und damit auch zu einer niedrigeren Wärmeentwicklung führt, vgl. auch die kurze Diskussion des Zusammenhangs in Abschn. 2.8.2. Die BG-Systeme basieren jeweils auf speziell für diese Systeme entwickelten BG-Prozessoren. Die BG/L-Prozessoren waren mit 700 MHz, die BG/P-Prozessoren mit 850 MHz getaktet; die Taktrate der BG/Q-Prozessoren beträgt 1,6 GHz. An den BG-Prozessoren kann die Entwicklung hin zu Multicore-Prozessoren beobachtet werden: Die BG/L-

Tab. 2.3 Zusammenstellung der wichtigsten Charakteristika der BG/L-, BG/P- und BG/Q-Systeme bzgl. Prozessor, Netzwerk und Stromverbrauch, vgl. [89]. Der L2-Cache wird von allen Kernen gemeinsam genutzt. Die Hauptspeicherangabe bezieht sich auf den Hauptspeicher pro Knoten. Die Angabe der Latenz zwischen Nachbarknoten im Netzwerk bezieht sich dabei auf Pakete mit einer Größe von 32 Bytes

Charakteristik	BG/L	BG/P	BG/Q
Prozessor	32-Bit PPC 440	32-Bit PPC 450	64-Bit PPC A2
Taktfrequenz	0,7 GHz	0,85 GHz	1,6 GHz
Fertigungstechnik	130 nm	90 nm	45 nm
Prozessorkerne	2	4	16
Performance Knoten	5,6 GFLOPS	13,6 GFLOPS	204,8 GFLOPS
L1-Cache pro Kern	32 KB	32 KB	16/16 KB
L2-Cache	4 MB	8 MB	32 MB
Hauptspeicher	0,5/1 GB	2/4 GB	16 GB
Topologie Netzwerk	3D-Torus	3D-Torus	5D-Torus
Bandbreite Netzwerk	2,1 GB/s	5,1 GB/s	32 GB/s
Latenz Nachbar	200 ns	100 ns	80 ns
maximale Latenz	6,4 μ s	5,5 μ s	3 μ s
Performance Rack	5,7 TF	13,3 TF	209 TF
Stromverbrauch Rack	20 KW	32 KW	100 KW
Stromeffizienz	0,23 GF/W	0,37 GF/W	2,1 GF/W

bzw. BG/P-Prozessoren waren Dual- bzw. Quadcore-Prozessoren, die BG/Q-Prozessoren sind bereits 16-Core-Prozessoren. Wir werden uns im Folgenden auf die Beschreibung des BG/Q-Systems konzentrieren und verweisen für weiterführende Informationen auf den genauen Überblick in [71, 24]. Ein Vergleich wichtiger Charakteristika der BG/L-, BG/P- und BG/Q-Systeme ist in Tab. 2.3 zusammengestellt, siehe auch [88, 89].

Ähnlich wie die BG/L- und BG/P-Prozessoren wurde der BG/Q-Prozessor als System-on-a-Chip-Design entworfen. Der Prozessorchip enthält neben den Prozessorkernen auch Hardwareeinheiten für die Speicherzugriffs- und Netzwerkkontrolle. Abbildung 2.37 zeigt ein Blockdiagramm des Prozessors, der aus ca. 1,47 Milliarden Transistoren aufgebaut ist. Der Prozessorchip enthält 16 Berechnungskerne, von denen jeder einen PowerPC A2-Prozessorkern (PPC) auf der Basis des IBM PowerEN-Chips [94] und eine speziell für den BG/Q-Rechner entwickelte SIMD-basierte Quad-FPU (Floating-Point Unit) enthält. Jeder A2-Prozessorkern implementiert die 64-Bit Power Instruktionen (Power ISA (*Instruction Set Architecture*)) und kann per Hyperthreading vier Threads parallel zueinander abarbeiten. Die Quad-FPU erweitert die skalaren Floating-Point-Operationen der Power ISA um spezielle SIMD-Floating-Point-Instruktionen (*Quad Processing eXtensions* genannt, QPX), die als SIMD-Instruktionen auf 32 speziellen 256-Bit-Registern arbeiten und pro Zyklus vier Floating-Point-Operationen auf 64-Bit-Werten ausführen

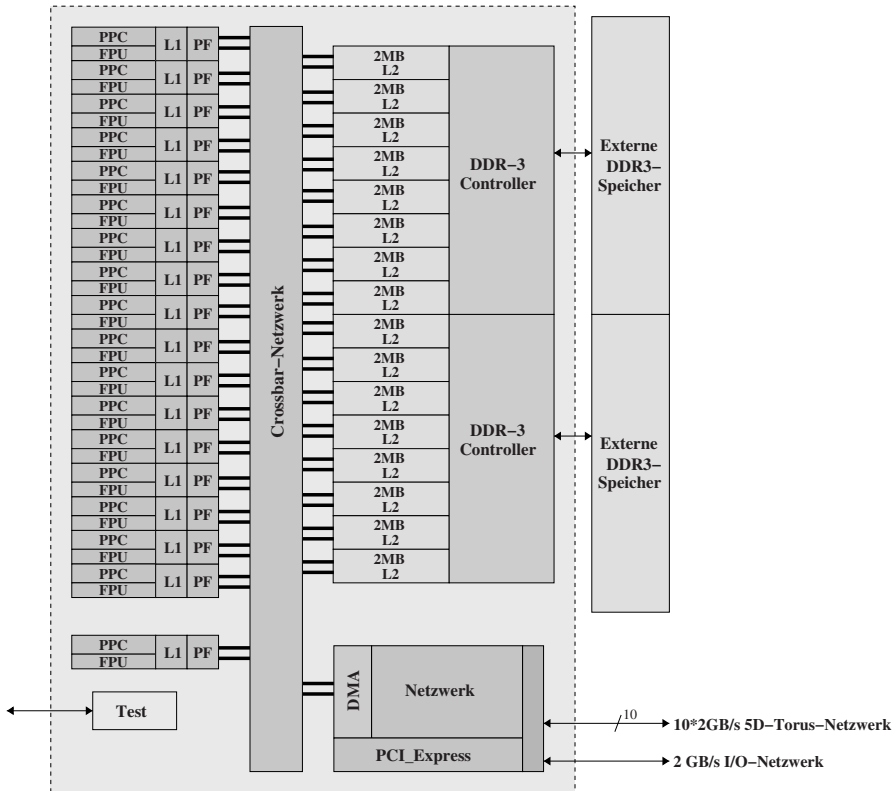


Abb. 2.37 Blockdiagramm des Blue Gene/Q Prozessorchips

können. SIMD-Instruktionen werden ausführlicher in Abschn. 3.6 behandelt. Dabei wird auch eine FMA-Operation (*Fused Multiply-Add*) zur Verfügung gestellt, so dass pro Zyklus bis zu acht Floating-Point-Operationen ausgeführt werden können. Dies ergibt bei einer Taktrate von 1,6 GHz eine maximale Performance von 12,8 GFLOPS pro Berechnungskern. Für den gesamten Prozessorchip mit 16 Berechnungskernen resultiert damit eine maximale Performance von 204,8 GFLOPS.

Zusätzlich zu den 16 Berechnungskernen wird ein spezialisierter Service-Berechnungskern für spezielle Betriebssystemaufgaben wie z. B. Interrupt- oder I/O-Behandlung bereitgestellt. Darüber hinaus gibt es auf dem Prozessorchip einen 18ten Berechnungskern, der aktiviert werden kann, falls beim Fertigungstest einer der Kerne nicht richtig funktionieren sollte und deaktiviert werden muss.

Auf den BG/Q-Prozessoren werden verschiedene Cachespeicher zur Verfügung gestellt. Jeder Berechnungskern hat einen privaten 16 KBytes großen L1-Instruktionscache (4-Wege-assoziativ) und einen ebenso großen L1-Datencache (8-Wege-assoziativ mit Cacheblöcken der Größe 64 Bytes), die von einem L1-Prefetcher (PF

in Abb. 2.37) zum vorgezogenen Laden von Daten unterstützt werden, um so die Latenz von Zugriffen auf den L2-Cache und den Hauptspeicher zu verdecken, vgl. die detaillierte Beschreibung in [71]. Zusätzlich steht ein von allen Prozessorkernen gemeinsam genutzter L2-Cache zur Verfügung, der zur Erhöhung der Bandbreite in 16 Bereiche mit jeweils 2 MB Größe unterteilt ist. Jeder Bereich ist 16-Wege-assoziativ und verwendet eine write-back Rückschreibestrategie. Auf der Ebene des L2-Caches unterstützt der BG/Q-Prozessor auch eine spekulative Ausführung von Instruktionen z. B. zur Unterstützung eines auf Transaktionsspeichern (engl. *transactional memory*) basierenden Ansatzes sowie atomare Operationen z. B. zur schnellen Implementierung von Sperroperationen. Auf den Prozessorchip integriert ist ebenfalls die Kontrolllogik für das 5D-Torus-Netzwerk des BG/Q-Systems.

Das zentrale Verbindungselement des Prozessorchips ist ein Crossbar-Netzwerk, das die Berechnungskerne, den L2-Cache und den Netzwerk-Controller miteinander verbindet. Das Crossbar-Netzwerk und der L2-Cache sind mit der halben Taktrate des Prozessors (800 MHz) getaktet. Die maximale on-Chip Bisektionsbandbreite des Crossbar-Netzwerkes beträgt 563 GB/s. Fehlzugriffe in den L2-Cache werden von zwei ebenfalls auf der Chipfläche integrierten Speicher-Controllern (DDR-3 Controller) verarbeitet, wobei jeder der beiden Controller für jeweils acht L2-Cachebereiche zuständig ist.

BG/Q-Systeme enthalten Berechnungsknoten und I/O-Knoten, wobei letztere die Verbindung zum Dateisystem herstellen. Die eigentlichen Berechnungen werden auf den Berechnungsknoten ausgeführt, die durch ein 5D-Torus-Netzwerk miteinander verbunden sind. Dazu besitzt jeder Knoten 10 bidirektionale Verbindungskanäle, von denen jeder simultan eine Datenmenge von 2 GB/s senden und empfangen kann. Die Kontrolllogik für das Netzwerk (*message unit (MU)* genannt) ist auf dem Prozessorchip integriert; sie stellt für jeden der 10 Torus-Kanäle eine Sende- und eine Empfangseinheit mit FIFO Sende- und Empfangspuffern zur Verfügung. Die MU dient als Schnittstelle zwischen dem Netzwerk und dem BG/Q-Speichersystem und ist an das prozessorinterne Crossbar-Netzwerk angeschlossen. Zusätzlich ist für einen Teil der Berechnungsknoten ein I/O-Kanal aktiviert, der die Verbindung zu den I/O-Knoten herstellt und der ebenfalls eine bidirektionale Bandbreite von 2 GB/s zur Verfügung stellt. Eine detaillierte Darstellung des Netzwerks der BG/Q-Systeme ist in [24] zu finden.

Im Hinblick auf das verwendete Verbindungsnetzwerk ist ebenfalls eine Entwicklung der BG-Systeme zu beobachten. Für die BG/L- und BG/P-Systeme war noch ein 3D-Torus-Netzwerk eingesetzt worden, siehe Tabelle 2.3. Der Übergang zu einem 5D-Torus ermöglicht eine wesentlich höhere Bisektionsbandbreite sowie eine Reduktion der Anzahl der zwischen nicht benachbarten Knoten zu durchlaufenden Zwischenstationen (*hops* genannt) und damit eine Reduktion der auftretenden Latenz. Zusätzlich zum Torus-Netzwerk hatten die BG/L- und BG/P-Systeme ein Barrier- und ein Broadcast-Netzwerk zur Verfügung gestellt. Diese Funktionalität ist bei den BG/Q-Systemen jetzt in das Torus-Netzwerk integriert.

Ein BG/Q-System ist aus mehreren Stufen aufgebaut, siehe [90]. Ein einzelner BG/Q-Prozessor wird mit 72 SDRAM DDR3 Speicherchips auf eine sogenannte

Compute-Card integriert; damit stehen pro Compute-Card 16 GB Hauptspeicher zur Verfügung. 32 dieser Compute-Cards werden in der logischen Form eines $2 \times 2 \times 2 \times 2 \times 2$ Torus auf einem Knoten-Board zusammengefasst. Wiederum 16 dieser Knoten-Boards werden in eine Midplane integriert, sodass insgesamt ein $4 \times 4 \times 4 \times 4 \times 2$ Torus entsteht. Ein BG/Q-Rack besteht aus ein oder zwei solchen Midplanes, wobei für zwei Midplanes ein $4 \times 4 \times 4 \times 8 \times 2$ Torus-Netzwerk resultiert. Für ein aus 128 Racks bestehendes BG/Q-System wird z. B. ein $16 \times 16 \times 16 \times 16 \times 2$ Torus-Netzwerk verwendet. Die maximal unterstützte Systemgröße ist 512 Racks, was 524 288 BG/Q-Prozessoren mit jeweils 16 Prozessorkernen entspricht. Dies entspricht einer maximalen Performance von 100 PetaFLOPS.



<http://www.springer.com/978-3-642-13603-0>

Parallele Programmierung

Rauber, Th.; Rünger, G.

2012, X, 522 S. 154 Abb., Softcover

ISBN: 978-3-642-13603-0