(Allgemeine) Gesetz von Amdahl

Amdahl´s Law:

> *„The performance improvements to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.“*

Damit ist der maximal erreichbare *Speedup* durch das Verhältnis Ausführungszeit des durch eine Verbesserung E beschleunigbaren Programmteils zu der Gesamtausführungszeit beschränkt.

$$Speedup(E) = \frac{ExTime_{w/o\_E}}{ExTime_{with\_E}} = \frac{Performance_{with\_E}}{Performance_{W/o\_E}}$$

Daraus folgt: (s. nächste Folie)

(Allgemeine) Gesetz von Amdahl

Oftmals lässt sich nur ein Teil des Programms beschleunigen.

Sei $Fraction_{enhanced}$ = Anteil der Ausführungszeit, der von der Beschleunigung E profitieren kann, zu der gesamten Ausführungszeit.

Dann gilt:

$$ExTime_{new} = ExTime_{old} \cdot \left[(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

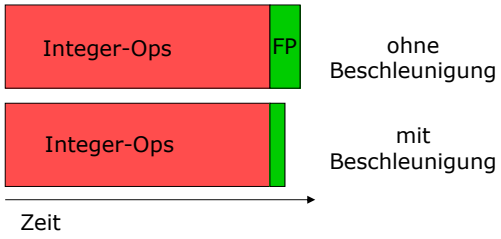
$$\Rightarrow Speedup_{overal} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Beispiel: Gesetz von Amdahl

Fließkomma-Operationen können um den Faktor zwei beschleunigt werden, aber nur 10% aller Instruktionen sind FP-Ops.

$$ExTime_{new} = ExTime_{old} \cdot (0.9 + \frac{0.1}{2}) = 0.95 \cdot Time_{old}$$

$$Speedup_{overall} = \frac{1}{0.95} = 1.053$$



Amdahlsches Gesetz für Parallelität

Sei: a = der Anteil der Ausführungszeit des nur sequentiell ausführbaren Programmteils

$$T(n) = T(1) \cdot a + T(1) \cdot \frac{1-a}{n}$$

$$\Rightarrow S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \cdot \frac{1-a}{n} + T(1) \cdot a}$$

$$= \frac{1}{\frac{1-a}{n} + a}$$

$$\Rightarrow S(n) \leq \frac{1}{a}$$

Grenzen der Skalierbarkeit

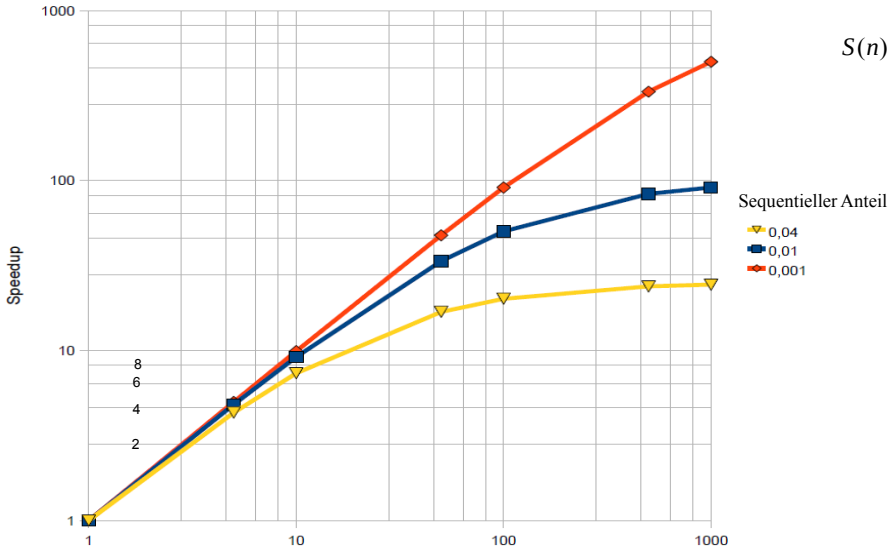
Beispiel: 100 Sekunden Ausführungszeit auf einem Prozessor

Anzahl Proz.	sequentieller Anteil							
	100%	50%	25%	12.5%	6.25%	3.125%	1.5625%	0.7812%
1	100	100	100	100	100	100	100	100
2	100	75	62,5	56,25	53,125	51,5625	50,78125	50,39062
4	100	62,5	43,75	34,375	29,6875	27,34375	26,17187	25,58593
8	100	56,25	34,375	23,4375	17,96875	15,23437	13,86718	13,18359
16	100	53,125	29,6875	17,96875	12,10937	9,179687	7,714843	6,982421
32	100	51,5625	27,34375	15,23437	9,179687	6,152343	4,638671	3,881835
64	100	50,78125	26,17187	13,86718	7,714843	4,638671	3,100585	2,331542
128	100	50,39062	25,58593	13,18359	6,982421	3,881835	2,331542	1,556396
256	100	50,19531	25,29296	12,84179	6,616210	3,503417	1,947021	1,168823
512	100	50,09765	25,14648	12,67089	6,433105	3,314208	1,754760	0,975036
1024	100	50,04882	25,07324	12,58549	6,341557	3,219604	1,658630	0,878143

Zeit in Sekunden.

Speedup = 114

Speedup Diagramm



Amdahl

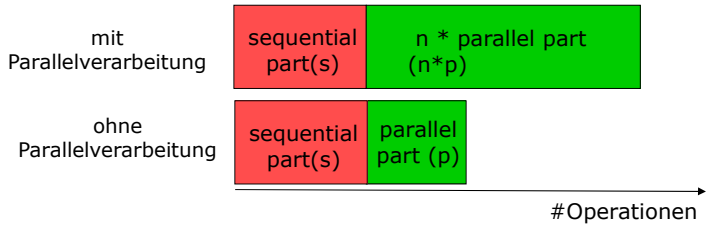
$$S(n) = \frac{1}{\frac{1-a}{n} + a}$$

Weitergehende Skalierung

- Gesetz von Amdahl berücksichtigt nicht, dass oftmals das zu lösende Problem mit Anzahl an Prozessoren wächst (weak scaling)
- Der sequentielle Anteil kann sich mit der Problemgröße bzw. Anzahl an Prozessoren relativ verkleinern (Ausführungszeit des seq. Anteils ist konstant)
- „Gesetz von Gustafson"

$$S(n) = a + n \cdot (1 - a)$$

mit a = Anteil der Laufzeit des sequentiellen Teilstückes des parallel ausgeführten Programms



Gesetz von Gustafson

Herleitung:

$T(1) = s + n \cdot p \qquad T(n) = s + p \qquad$, sei $a = \frac{s}{s+p} \implies 1 - a = \frac{p}{s+p}$

$$S(n) = T(1) / T(n) = \frac{s + n \cdot p}{s + p}$$

$$\Rightarrow S(n) = \frac{s}{s + p} + \frac{n \cdot p}{s + p}$$

$$\Rightarrow S(n) = a + n \cdot (1 - a)$$

Gesetz von Gustafson

Wie groß darf der seq. Anteil bei Ausführung mit einem Prozessor sein?

$$S(n) = a + n \cdot (1 - a) \geq {}^{n}/_{2} \quad \Longrightarrow \quad a = \frac{s}{s+p} \leq \frac{1}{2}$$

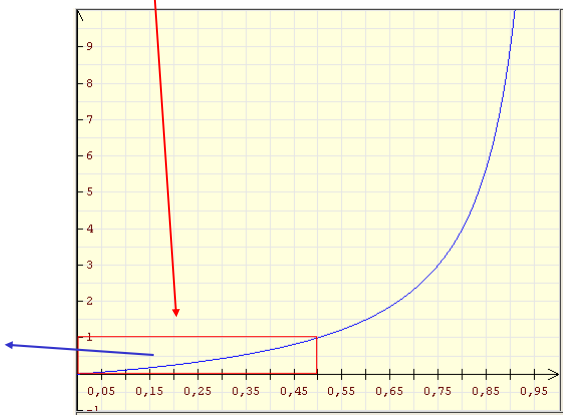
$$\frac{s}{T(1)} = \frac{s}{s + n \cdot p} \leq \frac{s}{n \cdot p}$$

$$= \frac{s}{n \cdot \left(\frac{s}{a} - s\right)} = \frac{1}{n} \cdot \frac{s}{\frac{s - s \cdot a}{a}}$$

$$= \frac{1}{n} \cdot \frac{s}{s \cdot \frac{1-a}{a}}$$

$$= \frac{1}{n} \cdot \frac{a}{1-a}$$

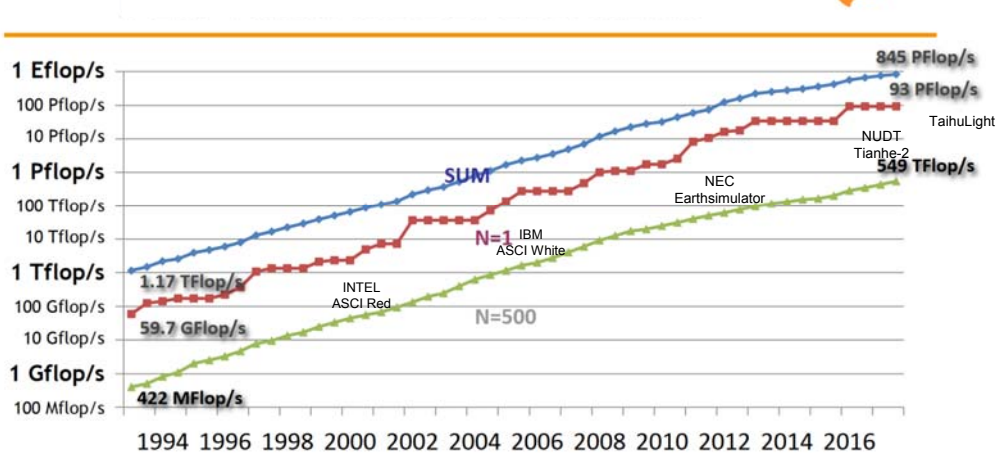
$$\leq \frac{1}{n} \qquad , falls \ a \leq \frac{1}{2}$$



f(a)=a/(1-a)

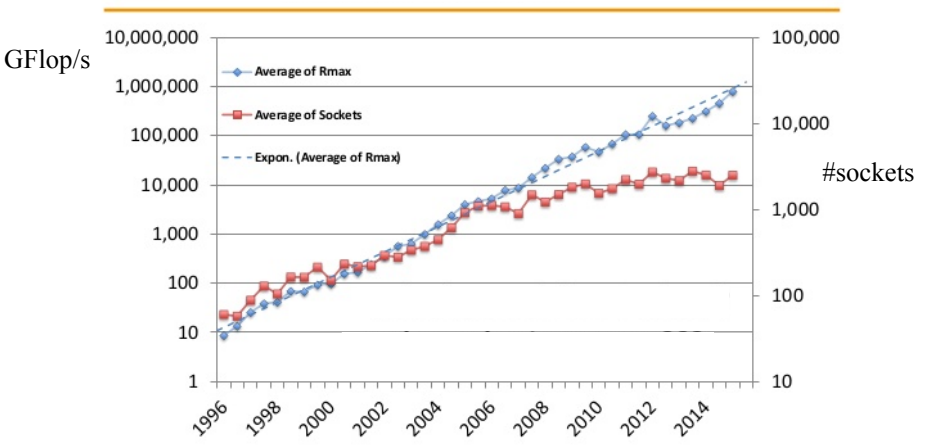
TOP500 Performance Development

TOP500 is using the weak scaling Linpack Benchmark. In this case Gustafson's law holds.



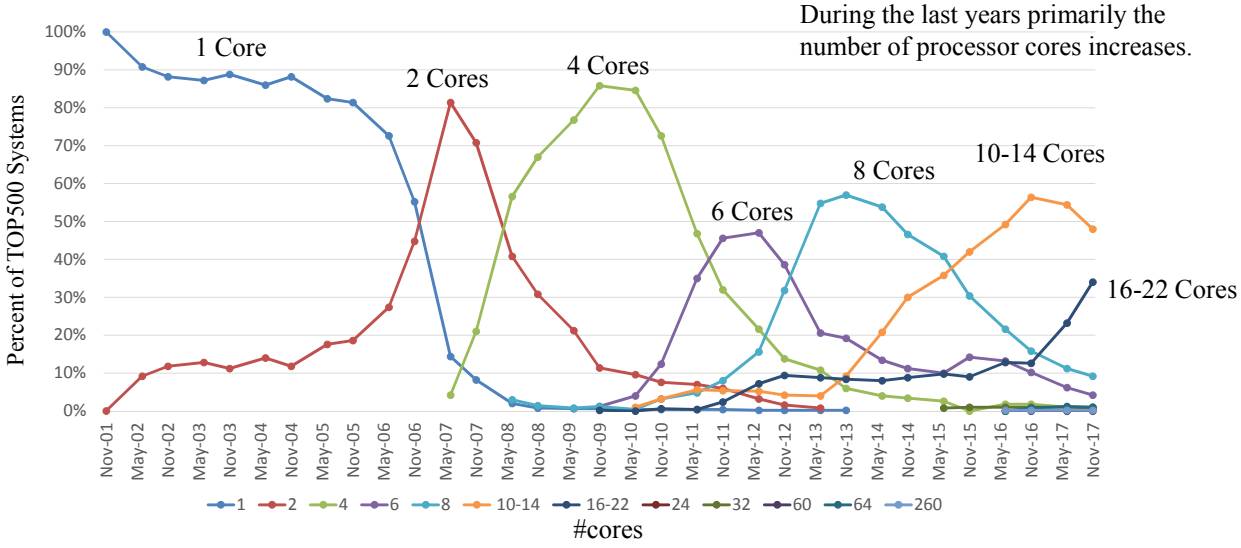
Trend: Performance vs. #Processors

Number of average processors (sockets) of TOP500 systems increases.



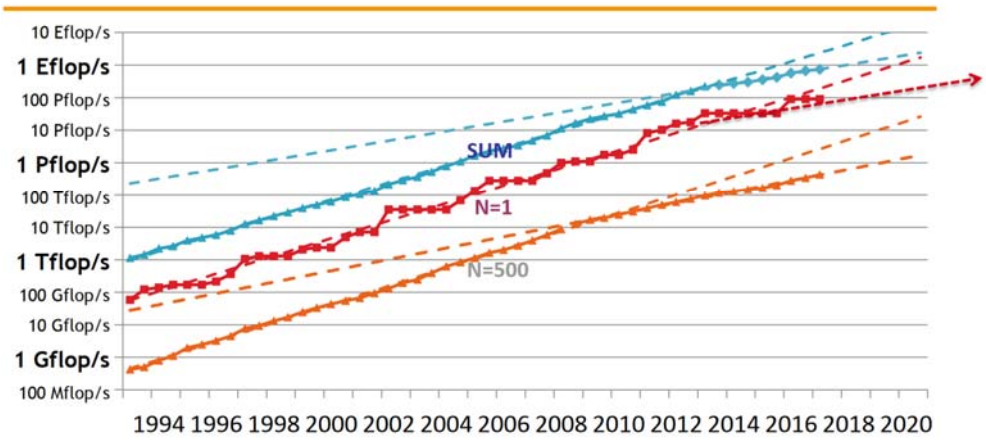
Trend: Processor Cores per Socket

During the last years primarily the number of processor cores increases.



TOP500 Performance Projection

Due to electrical power limitations, the performance improvement increases less.



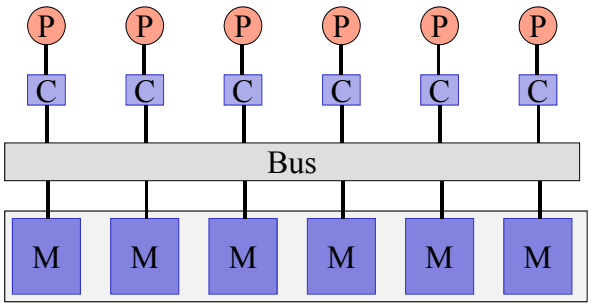
Skalierbare speichergekoppelte Systeme

Symmetrische Multiprozessorsysteme

Symmetrisches Multiprozessorsystem (SMP)

- globaler Speicher und globaler Adressraum
- SMPs gibt es seit Mitte der 60er Jahre
- SMPs haben typischerweise zwischen 2 und 64 Prozessoren
- BUS als einfache Verbindungsarchitektur
  - heute meistens ein skalierendes Netzwerk (NUMA)
- jeder Prozessor hat einen lokalen Cache
- Plattform für Vielzahl kommerzieller Anwendungen
- damit ein typischer Abteilungsserver

SMP-Architekturen: Bus

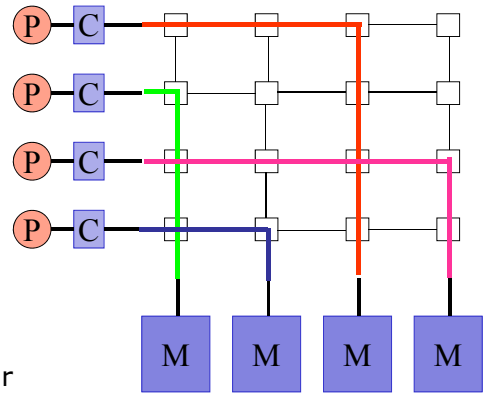


P: Prozessor
C: Cache
M: Speicher

Probleme

- Bus ist Engpass

SMP-Architekturen: Crossbar

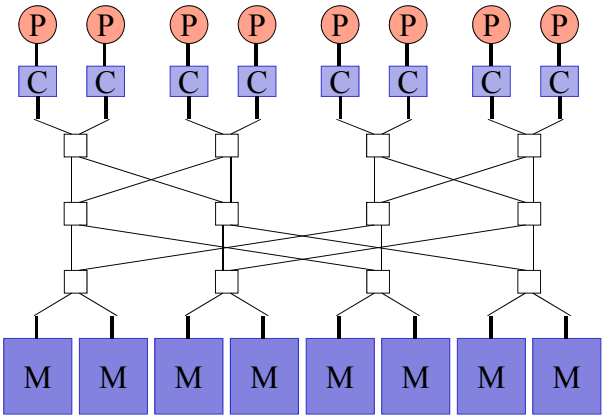


Probleme

- Aufwendiger Crossbar
- Kollisionsauflösung an den Speichermodulen

P: Prozessor
C: Cache
M: Speicher

SMP-Architekturen: Netzwerk

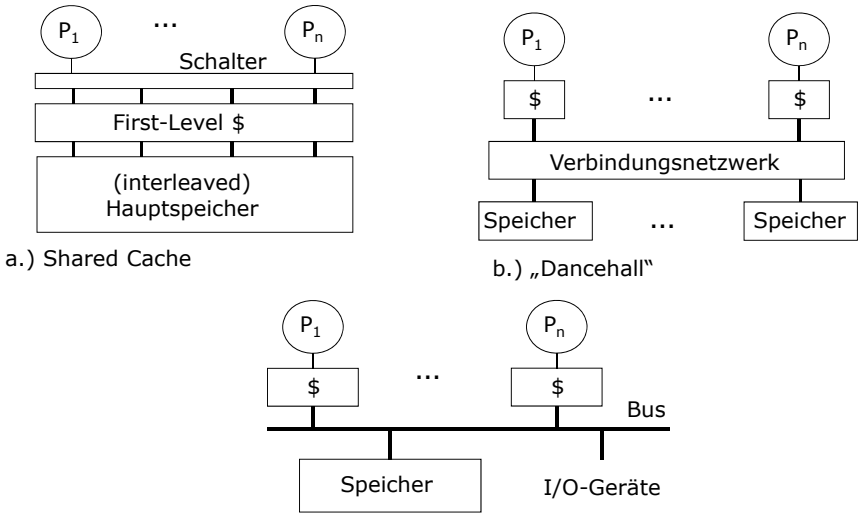


Probleme

- ggf. Verbindungsnetz als Engpass
- Kollisionsauflösung auch im Netzwerk

P: Prozessor
C: Cache
M: Speicher

Speichermodelle



Speichermodelle

a) Shared Cache

- ein Cache mit großer Kapazität
- geringe Latenz durch *Sharing* und *Prefetching* zwischen Prozessoren
  - ideal bei gemeinsamer Nutzung eines *Working-Sets*
- Hauptspeicher wird nicht durch verschiedene Caches gesehen

aber

- hohe Bandbreiten von Cache zu den Prozessoren notwendig
- Hit- und Miss-Latenz sind durch Schalter und den großen Cache bestimmt
- Mitte 80-iger Jahre eingesetzt für Platinen mit 2-8 Prozessoren
- Heute in kleine Multiprozessor-Module „On-a-Chip" genutzt

b) „Dancehall"

- gedacht um alle Speicherelemente **gleich weit** entfernt zu halten
- mittlerweile nur noch in NUMA-Systeme im Einsatz

Bus-basierter, zentraler Speicher

c) Bus-basierter, gemeinsamer Speicher
- private Caches
- oftmals mit zentralisiertem Hauptspeicher realisiert
- meist verwendete Architektur für kleine Server-Systeme oder Single-Chip Multiprozessoren

Charakteristika von Caches (1)

Die wichtigsten Eigenschaften von Caches
- Cache-Lines
  - Granularität der Zugriffe in den Hauptspeicher (Blöcke)
  - Blockgröße ≥32 Bytes
- Caches deutlich kleiner als Hauptspeicher
  => Mapping der Blöcke in Cache erforderlich
  - Direct mapped – jeder Block wird auf festen Bereich abgebildet
  - Fully associative – Block kann überall im Cache abgelegt werden
  - m-way set associative
    - Block beliebig in einer Menge von m Cache-Lines ablegbar
    - Replacement: zufällig oder LRU
- weitere Charakteristika
  - Kapazität des Caches
  - Latenzzeit (Hit/Miss)
  - Bandbreite

Charakteristika von Caches (2)

Cache-Write-Policies:

- Write-Through
  - schreibt in Cache und Hauptspeicher zur gleichen Zeit
  - zwischen Cache und Hauptspeicher befinden sich *Write-Buffer*
- Write-Back
  - Prozessor schreibt nur in den Cache
  - Block wird erst dann in den Hauptspeicher geschrieben, wenn dieser aus dem Cache verdrängt wird
  - „*dirty*" Status-Bit pro Cache-Line benötigt
  - reduziert benötigte Bandbreite zum Hauptspeicher
  - *Write-Buffer* eingesetzt
  - aufwendige Kontrolllogik notwendig
  - in den meisten Systemen eingesetzt

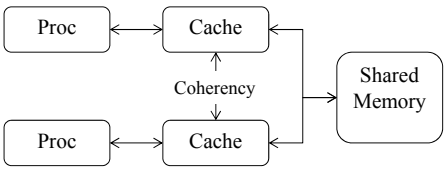
Caches und Speicherkohärenz

Caches sind wichtige Elemente

- reduzieren mittlere Datenzugriffszeit
- reduzieren Bandbreitenanforderung an gemeinsames Verbindungsnetzwerk

aber private Prozessor-Caches implizieren Probleme

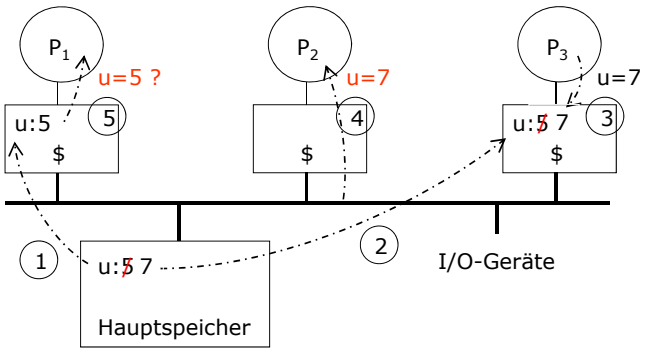
- Kopien einer Speicherzelle können in mehrere Caches sein
- Änderung einer Kopie muss nicht für andere sichtbar sein
  (Prozessoren greifen auf nicht gültige Werte ihres Caches zu)
- Problem der Cache-Kohärenz
- erfordert zusätzliche Aktionen zur Sicherstellung der Sichtbarkeit von Speicheroperationen



Intuition: Kohärentes Speichersystem

- Lesen einer Adresse liefert den zuletzt darauf geschriebenen Wert
  - einfach in Einprozessorsysteme
    - Ausnahme bei I/O: Kohärenz zwischen I/O-Geräten und Prozessor notwendig
    - I/O ist aber eher selten, deshalb sind Software-Lösungen ausreichend (*uncacheable* Speicherbereiche, *uncacheable* Operationen, *flush* von ganzen Speicherseiten, Durchreichen von I/O-Daten durch Caches)
  - sollte auch gelten, wenn ein Thread/Prozess nacheinander auf unterschiedliche Prozessoren ausgeführt wird (Prozessmigration)
    - auch bei verschachtelter Ausführung auf einem Prozessor
- In einem Multiprozessorsystem ist das Kohärenz-Problem deutlich komplexer
  - zwei Prozesse können den gemeinsamen Speicher durch zwei verschiedene Caches sehen
  - Performance kritisch, deshalb wichtiges Merkmal des Hardware-Designs

Beispiel: Kohärenzproblem (Write-Through)



(1) P_1 liest *u* vom Hauptspeicher
(2) P_3 liest *u* vom Hauptspeicher
(3) P_3 schreibt *u* mit 7 (Cache und Hauptspeicher)
(4) P_2 liest *u* aus Hauptspeicher
(5) P_1 liest *u* aus Cache

- nach (3) sehen Prozessoren unterschiedliche Werte von *u*
- mit *Write-back* Cache hängt der zurück geschriebene Wert vom zufälligen Ereignis ab, wann welcher Cache seinen Wert zurück schreibt
  - Prozesse können im Hauptspeicher veraltete Werte sehen
- für ein Programm inakzeptabel
- häufig anzutreffendes Szenario

Problem der intuitiven Definition

- Lesen von einer Adresse liefert den zuletzt darin geschriebenen Wert
  - Aber was ist mit „zuletzt" gemeint?
  - selbst im sequentiellen Fall wird „zuletzt" bzgl. der Programmordnung und nicht bzgl. Zeit definiert
    - Ordnung der Operationen in der ein Prozessor seine Maschineninstruktionen ausführt
    - „nachfolgende Operation" entsprechend
  - im parallelen Fall ist die Programmordnung nur innerhalb eines sequentiellen Programms definiert, muss aber auch über Prozessgrenzen hinweg einen Sinn ergeben

- Definition einer aussagekräftigen Semantik notwendig

Definitionen für Einprozessorsysteme (SP)

Speicheroperation
- ein einzelner Lese- (*load*), Schreib- (*store*) oder Lese-und-Schreib- (*read-modify-write*) Zugriff auf eine Speicheradresse

Zustand einer Speicheroperation
- *Issued*: eine Speicheroperation ist angestoßen, sobald diese den Prozessor verlassen und das Speichersystem erreicht (Cache, Buffer …)
- *Performed (SP)*: Operation ist ausgeführt, sobald andere Speicheroperationen des Prozessors dieses bestätigen
  - Schreiben ist für ein Prozessor ausgeführt, wenn ein von ihm nachfolgendes Lesen den Wert dieser Schreiboperation oder einer nachfolgenden Schreiboperation zurück gibt
  - Lesen ist für ein Prozessor ausgeführt, wenn ein von ihm nachfolgendes Schreiben den zu lesenden Wert nicht mehr ändern kann

Definitionen für Multiprozessorsysteme (MP)

Modifikation der Definition von *Performed*

–*Performed (MP)*: Operation ist ausgeführt, sobald andere Speicheroperationen eines Prozessors dieses bestätigen

- ein Schreiben ist für ein Prozessor ausgeführt, wenn ein von einem Prozessor nachfolgendes Lesen den Wert dieser Schreiboperation oder einer nachfolgenden Schreiboperation zurück gibt
- ein Lesen ist für ein Prozessor ausgeführt, wenn ein von einem Prozessor nachfolgendes Schreiben den zu lesenden Wert nicht ändern kann

Zusätzlich:

– *Completed*: Definition von *Performed* bezogen auf alle Prozessoren

– Notwendigkeit der Einbeziehung der Reihenfolge der Operationen verschiedener Prozesse

Kohärenz: Verfeinerung der Intuition (1)

Sei ein gemeinsamer Speicher ohne Prozessor-Caches gegeben

– jedes Lesen und Schreiben auf eine Adresse greift direkt auf die eindeutige Speicherzelle zu

– Ist eine Operation auf einem Prozessor ausgeführt, dann gilt diese Operation auch als *complete*

Speicher erzwingt eine *totale Ordnung* (serielle Ordnung) der Operationen auf eine Adresse

– Operationen eines bestimmten Prozessors auf eine Adresse sind in *Programmordnung*

– die Reihenfolge der Operationen von unterschiedlichen Prozessoren auf eine Adresse ist eine Verschachtelung der individuellen *Programmordnungen*

Der letzte Punkt meint eine hypothetische serielle Ordnung, die diese Eigenschaft beibehält

Kohärenz: Verfeinerung der Intuition (2)

- Konsistenz der seriellen Ordnung erfordert, dass **alle** Prozessoren jegliches Schreiben auf eine Adresse in der gleichen Reihenfolge sehen (falls Prozessoren die Adresse überhaupt lesen)
- Beachte, dass die totale Ordnung in realen Systemen niemals realisiert wurde
  - der Hauptspeicher, oder auch andere HW, soll, wenn möglich, nicht alle Operationen sehen
- Dennoch soll ein Programm so ablaufen, als ob eine *serielle Ordnung* beibehalten wurde

Definition Kohärenz

- Kohärenz beschreibt das Verhalten von mehreren Prozessoren bei Schreib- und Leseoperationen auf die gleiche Speicherzelle
- Ein Speichersystem ist kohärent, falls
  - die Programmordnung für Lese-/Schreiboperation gewahrt ist,
  - alle Schreiboperationen sichtbar werden,
  - alle Prozessoren die gleiche Reihenfolge an Schreiboperationen auf eine Variable sehen.

Kohärente Speichersystem

Speichersystem muss ermöglichen, dass

- von einem beliebigen Prozess angestoßene Speicheroperationen genau in der vom Prozess vorgegebenen Reihenfolge erscheinen.
- der Wert einer Lese-Operation der Wert des in serieller Ordnung letzten Schreibens ist.
- das Ergebnis einer Programmausführung aus einer hypothetischen seriellen Ordnung aller Speicheroperationen konstruierbar ist.

Die Lösung des Kohärenzproblems bestimmt im System

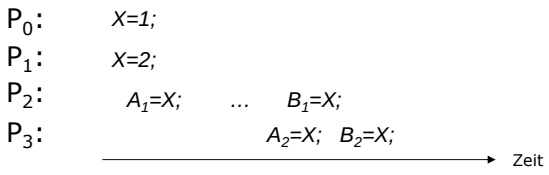
- sowohl die Korrektheit
- als auch dessen Leistungsfähigkeit.

Eigenschaften der Kohärenz

Zwei **notwendige Eigenschaften**:

1. *Write-Propagierung*: geschriebene Werte müssen für andere Prozessoren sichtbar sein
2. *Write*-Serialisierung: Schreiben auf eine Adresse wird durch alle Prozessoren in gleicher Reihenfolge gesehen
   - falls einer w2 nach w1 sieht, dann darf niemand w2 vor w1 sehen
   - kein Analogon fürs Lesen, da das Lesen für andere nicht sichtbar ist

Beispiel:



$A_1 = 1$ und $B_1 = 2$ $\Rightarrow$ nicht ($A_2 = 2$ und $B_2 = 1$)

Cache-Kohärenz auf dem Bus

Wie kann Kohärenz auf einem Bus realisiert werden?

- Multiprozessorsysteme nutzen Erweiterung der *Write-Through*- und *Write-Back*-Protokolle
- Verwendung der Grundlagen von Einprozessorsystemen
  - Bus-Transaktionen
    - drei Phasen: Arbitrierung, Kommando/Adresse, Datentransfer
    - alle Geräte lauschen auf Adressen, einer hat die Verantwortung
  - Zustände der Speicherblöcke im Cache
    - zu jedem Block im Cache gehört ein endlicher Zustandsautomat
    - *Write-Through*, write no-allocate hat zwei Zustände: *Valid* und *Invalid*
    - *Write-Back* Cache hat einen weiteren Zustand: *Modified (Dirty)*
  - Zustandsübergänge
    - getrieben durch Prozessor- und Bus-Transaktionen

Snooping-basierte Kohärenz

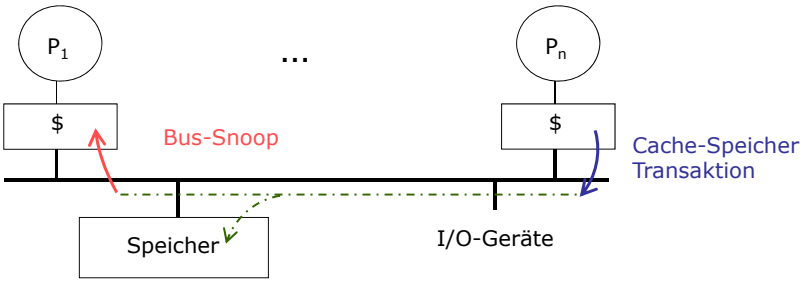
Die Idee:

- Transaktionen auf dem Bus sind von allen Prozessoren aus sichtbar
- Prozessoren, oder zugehörige Einheiten, können den Bus beobachten (*Snooping*) und bei relevanten Ereignissen aktiv werden (Zustandsänderung)

Implementierung eines Protokolls:

- Cache-Controller bekommt Eingaben von zwei Seiten:
  - Anfragen (R*equests*) vom Prozessor und Bus-*Requests/Responses* vom *Bus Snooper*
- in beiden Fällen werden keine oder mehrere Aktionen durchgeführt
  - Zustandsaktualisierung, Antworten mit Daten, Generierung einer neuen Bus-Transaktion
- Protokoll ist verteilter Algorithmus: kooperierende Zustandsautomaten, bestehend aus
  - Menge an Zuständen, Beschreibung der Zustandsübergänge und Aktionen
- Granularität der Kohärenz ist auf Cache-/Block-Ebene
  - Allokation im Cache und beim Transfer von/zum Cache

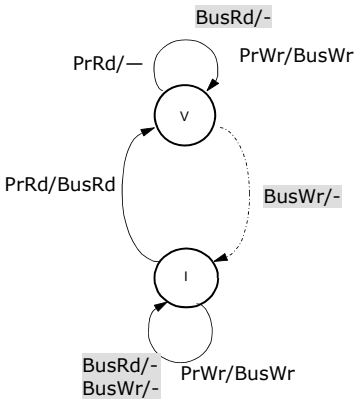
Kohärenz mit Write-Through-Cache



- Haupterweiterung gegenüber Einprozessorsystem: *Snooping*, *invalid/update* Caches
  - keine neuen Zustände oder Bus-Transaktionen benötigt
  - Invalidierungsprotokoll versus *Update*-basiertes Protokoll
- *Write-Propagierung*: auch bei *Invalid* sieht späteres Lesen neuen Wert
  - *Invalid* erzeugt ein *Miss* beim späteren Zugriff, wobei der Speicher bereits den aktuellen Wert durch das Write-Through hat

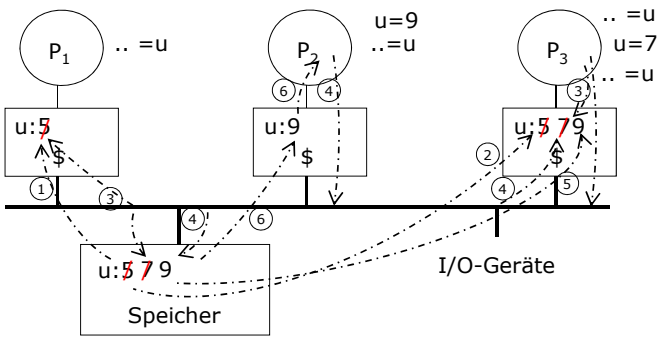
Write-Through: Zustandsübergangsdiagramm

- Write no-allocate Cache
- zwei Zustände pro Block in jedem Cache
  - Zustand eines Blocks kann als *p*-Vektor betrachtet werden (*p* = #Prozessoren)
- Zustands-Bits nur für im Cache enthaltene Blöcke
  - alle anderen Blöcke sind implizit *Invalid*
  - *Write* invalidiert ggf. Blöcke in anderen Caches (aber keine lokale Zustandsänderung)
  - gleichzeitig mehrere Sharer eines Blocks möglich, ein *Write* invalidiert dann alle Kopien



V: Valid
I: Invalid
Prozessor-Transak./Bus-Transak.
Bus-Transaktion/-

Beispiel: Write-Through



(1) P_1 liest *u*

(2) P_3 liest *u*

(3) P_3 beschreibt *u*

(4) P_2 beschreibt *u*

(5) P_3 liest *u*

(6) P_2 liest *u*

Aktion	*Pr-Aktion*	*Bus Aktion*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1	PrRd (1)	BusRd	Memory	V	-	-
2	PrRd (3)	BusRd	Memory	V	-	V
3	PrWr (3)	BusWr	Memory	I	-	V
4	PrWr (2)	BusWr	Memory	I	-	I
5	PrRd (3)	BusRd	Memory	I	-	V
6	PrRd (2)	BusRd	Memory	I	V	V