Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Basierend auf Folien von Julian Stecklina und Marcus Völp

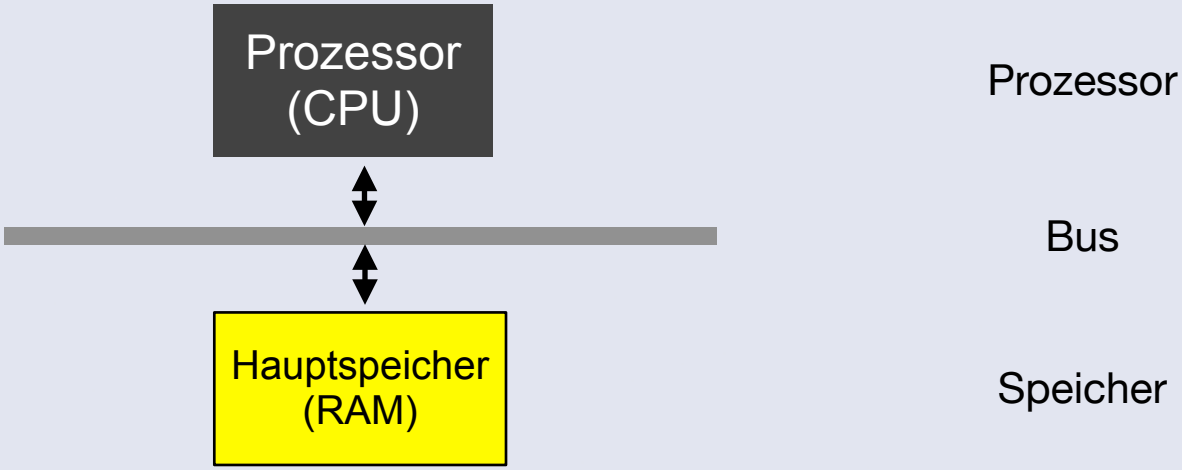
Vorlesung 16: Speicherkonsistenz

Cachekohärenz und Speicherkonsistenzmodelle

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

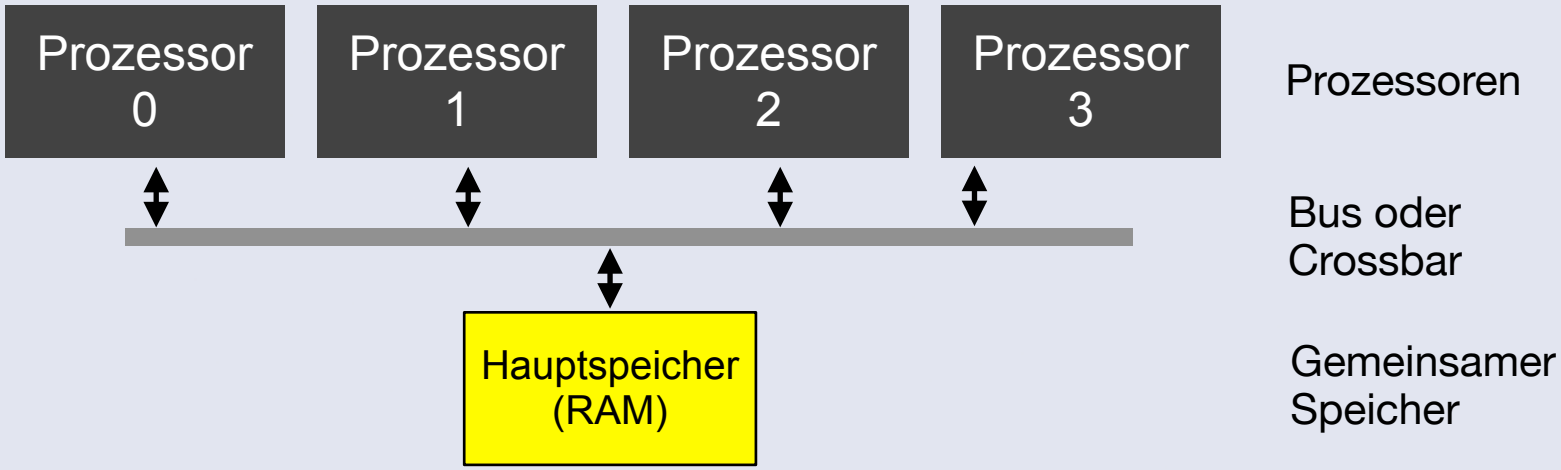
Einprozessorsysteme

- Bisher haben wir nur Systeme mit einem Prozessor betrachtet…



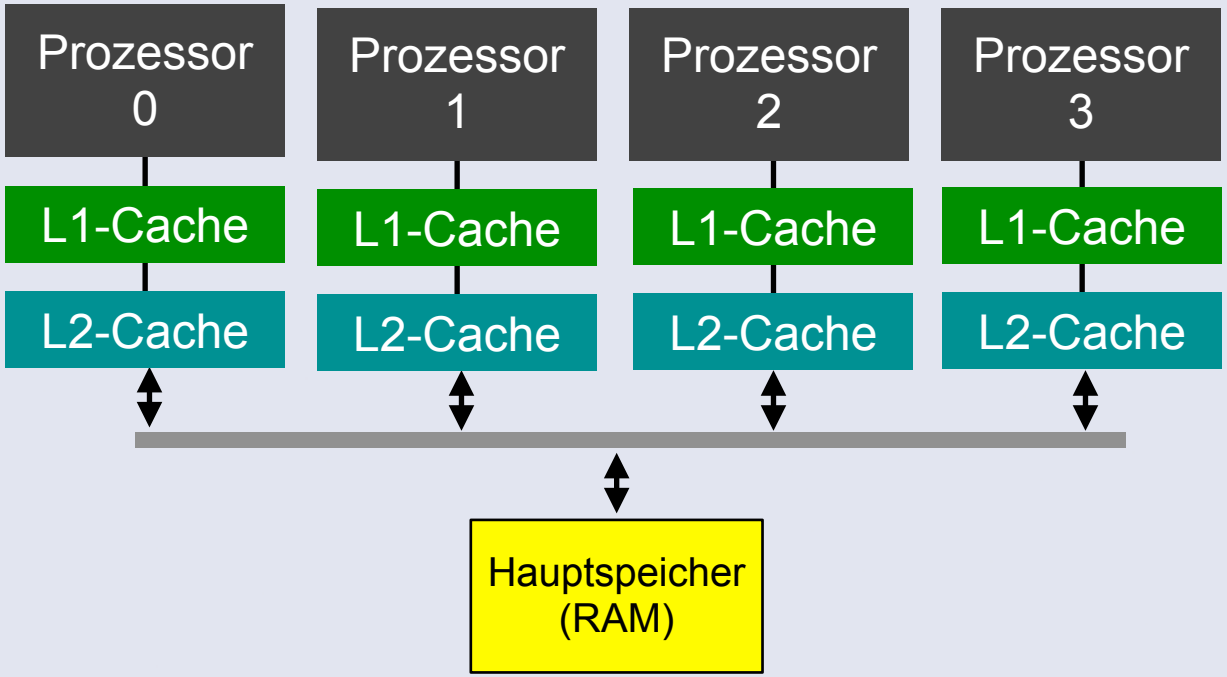
Mehrprozessorsysteme

- Moderne Computer nutzen **symmetrische Mehrprozessorsysteme** (Multiprozessor- oder Multicoresysteme)



Mehrprozessorsysteme

- **Caches** (vgl. Vorlesung 5) werden genutzt, um Performancenachteile durch langsamen Hauptspeicher auszugleichen



Prozessoren

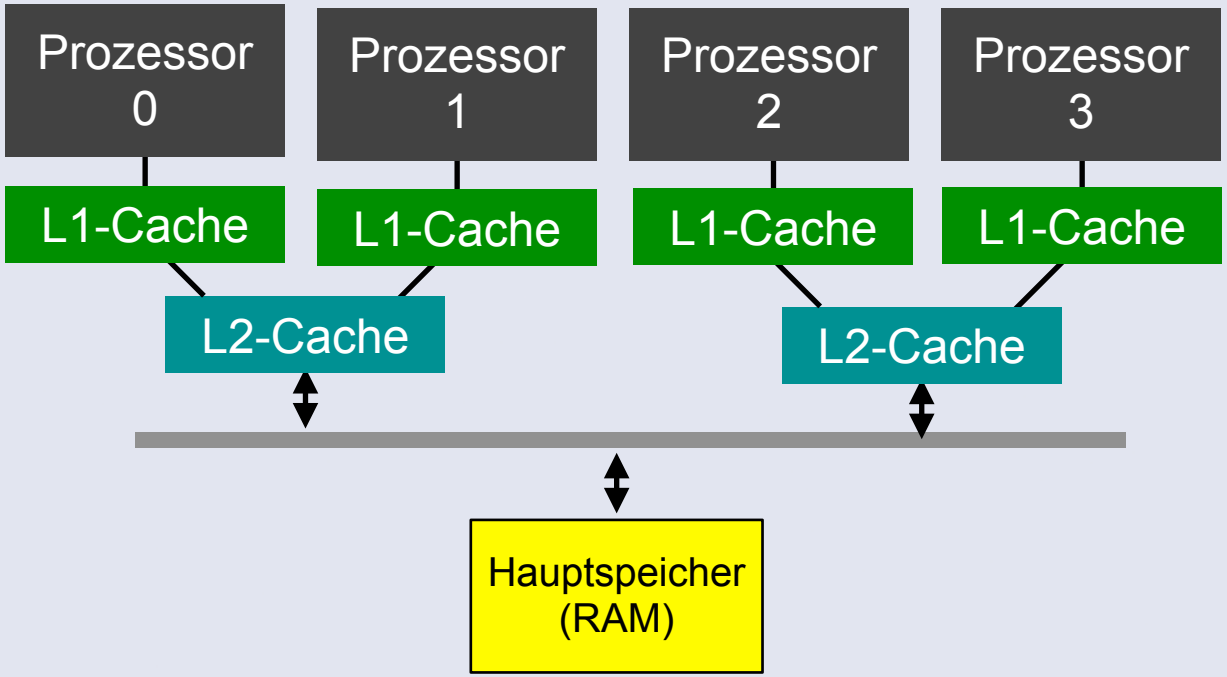
Prozessorlokale Caches

Bus oder Crossbar

Gemeinsamer Speicher

Multicore-Systeme

- Oft werden nachgelagerte Caches (L2 oder L3) von mehreren Cores gemeinsam genutzt



Prozessoren

Prozessorlokaler Cache

Gemeinsame L2-Caches

Bus oder Crossbar

Gemeinsamer Speicher

Cachekohärenz

- Bei Verwendung von Caches können mehrere Kopien des Inhalts einer einzelnen Hauptspeicherzelle im System vorliegen
- **Cache-Kohärenz** hält die Kopien “konsistent”
  - Finden aller Kopien
  - Invalidieren oder Aktualisieren des jeweiligen Cache-Inhalts
- **Reproduktion von Schreibzugriffen (*write propagation*)**
  - Schreibzugriffe auf den Speicher müssen für alle Prozessoren sichtbar werden
- **Serialisierung von Schreibzugriffen (*write serialization*)**
  - Jeder Prozessor sollte Schreibzugriffe auf die selbe Speicherzelle in der selben Reihenfolge beobachten

Alternative Definition: SWMR

Single-Writer, Multiple-Reader-Invariante
(ein Schreiber, mehrere Leser)

Für eine beliebige Speicherzelle A darf zu jedem beliebigen Zeitpunkt **entweder** nur ein einzelner Prozessor(-kern) den Inhalt von A schreiben **oder** eine beliebige Anzahl von Prozessoren dürfen den Inhalt von A lesen

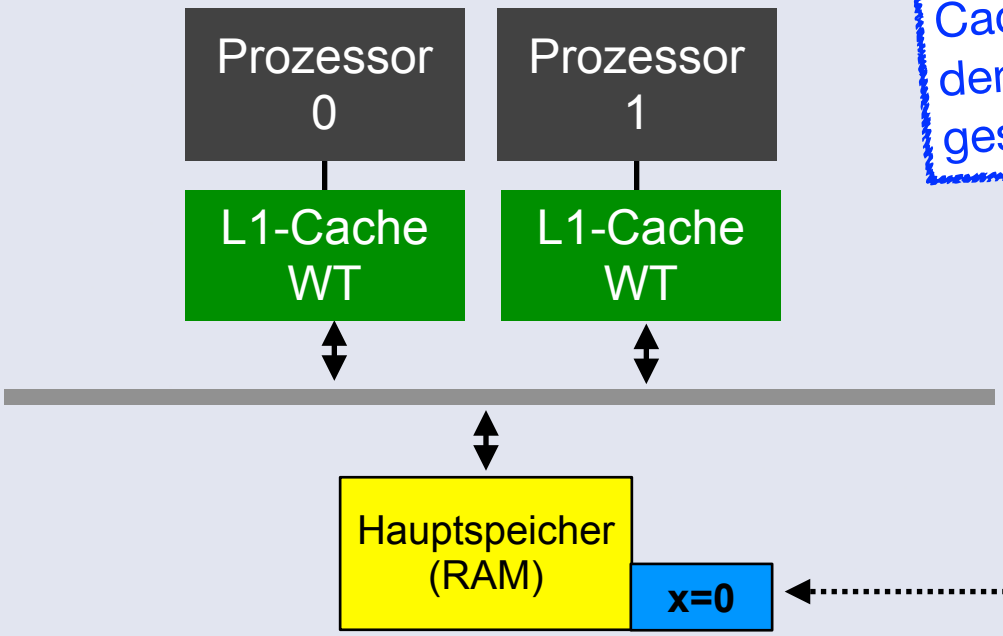
Data-Value-Invariante

Der Inhalt einer Speicherzelle zu Beginn einer Operation ist identisch zu deren Inhalt am Ende der ***vorherigen*** Schreiboperation

[nach Sorin et al., 2011]

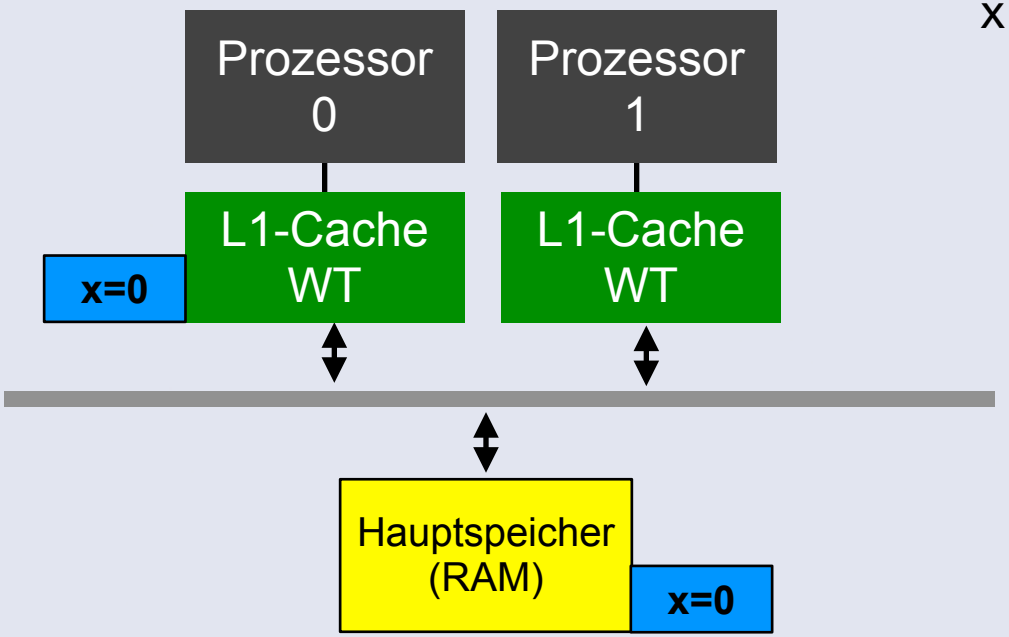
Ansatz 1:
"**write through**" (WT)-Caches

"**Write through**" bedeutet, dass Änderungen in einer Cache-Line direkt auch in den Hauptspeicher zurück-geschrieben werden



Variable x mit Wert 0 liegt im Hauptspeicher

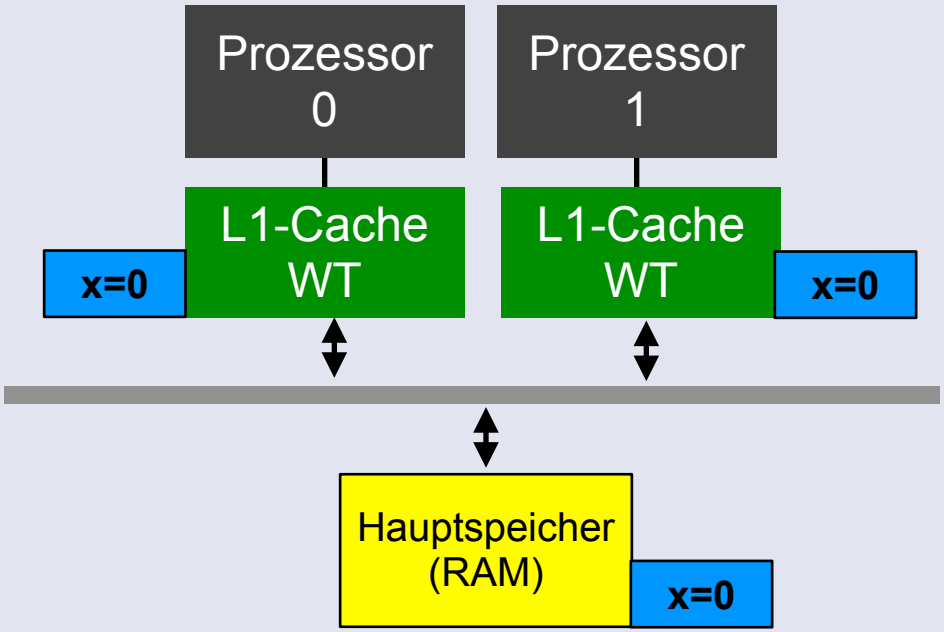
Ansatz 1:
"write through" (WT)-Caches



Prozessor 0 liest x

x = 0 im Cache gespeichert

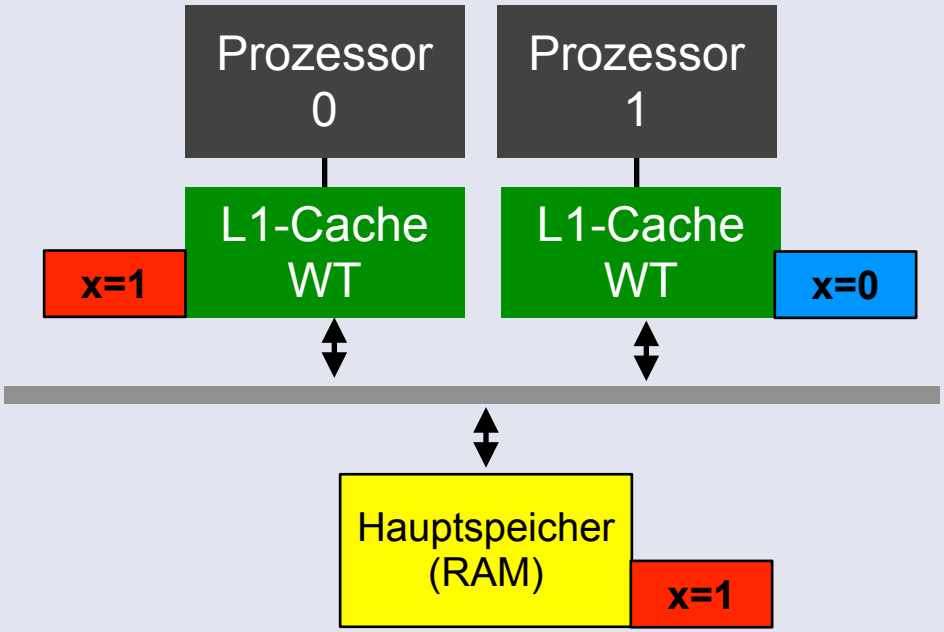
Ansatz 1:
"write through" (WT)-Caches



Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Ansatz 1:
"write through" (WT)-Caches

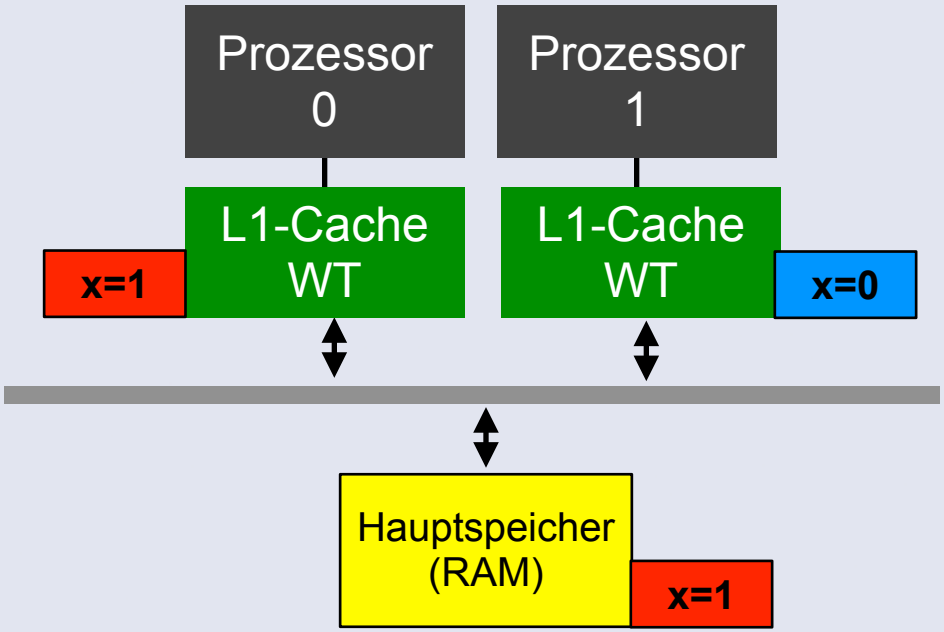


Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert
x = 1 im Hauptspeicher

Ansatz 1:
"write through" (WT)-Caches



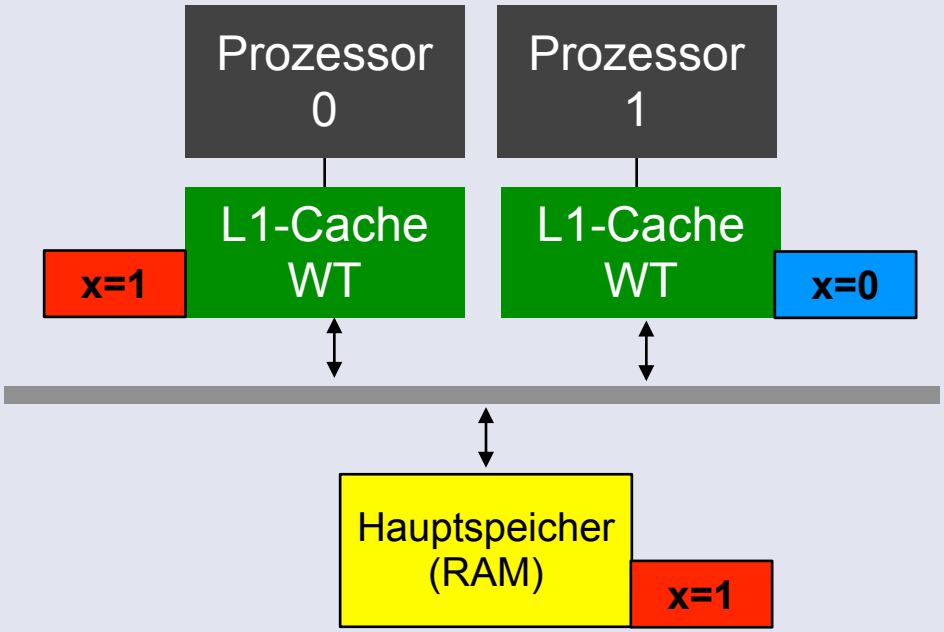
Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert
x = 1 im Hauptspeicher

Prozessor 1 liest x
x = 0 **aus Cache gelesen!**

Ansatz 1:
"write through" (WT)-Caches



Prozessor 0 liest x
x = 0 im Cache gespeichert

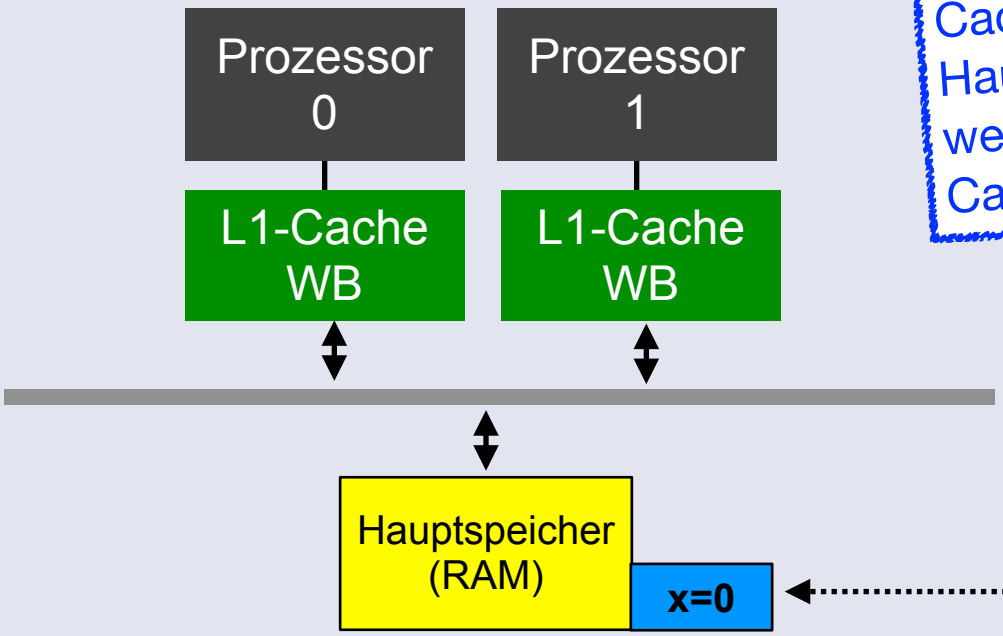
Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert
x = 1 im Hauptspeicher

Prozessor 1 liest x
x = 0 **aus Cache gelesen!**

Schreibzugriff auf x für Prozessor 1 nicht sichtbar!

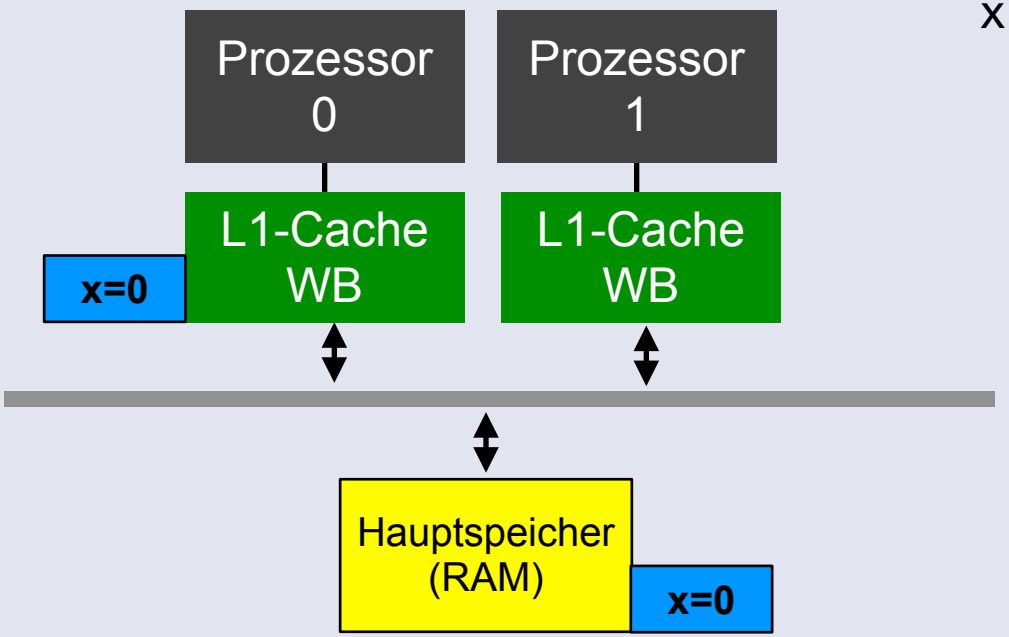
Ansatz 2:
"**write back**" (WB)-Caches



"**Write back**" bedeutet, dass Änderungen in einer Cache-Line erst in den Hauptspeicher geschrieben werden, wenn sie aus dem Cache verdrängt werden

Variable x mit Wert 0 liegt im Hauptspeicher

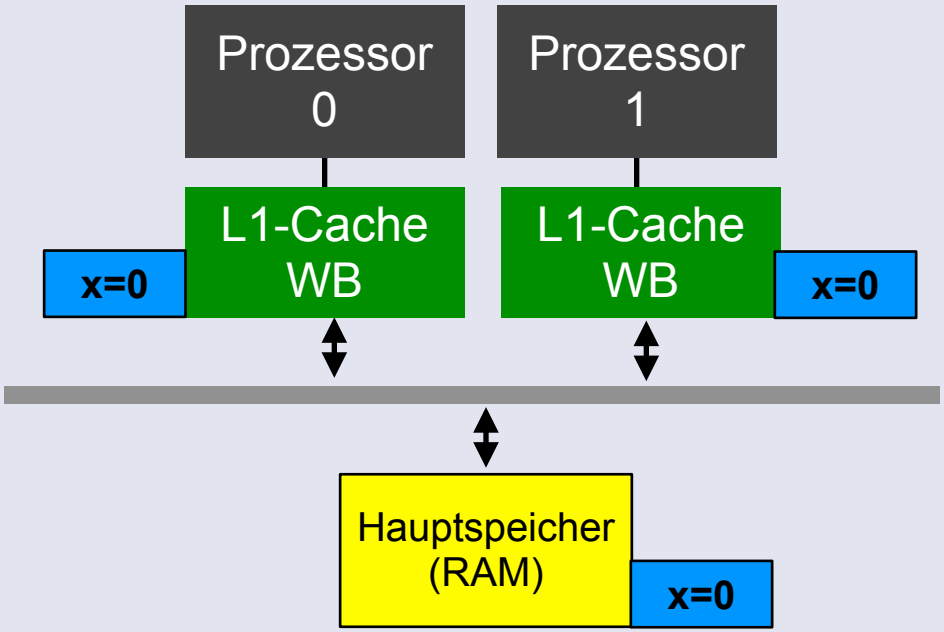
Ansatz 2:
"write back" (WB)-Caches



Prozessor 0 liest x

x = 0 im Cache gespeichert

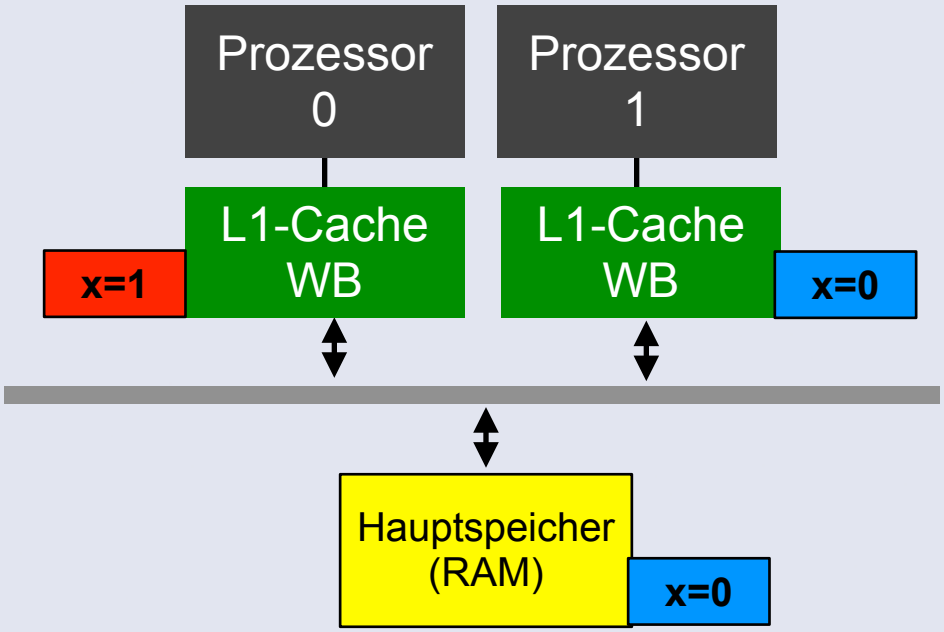
Ansatz 2:
"write back" (WB)-Caches



Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Ansatz 2:
"write back" (WB)-Caches

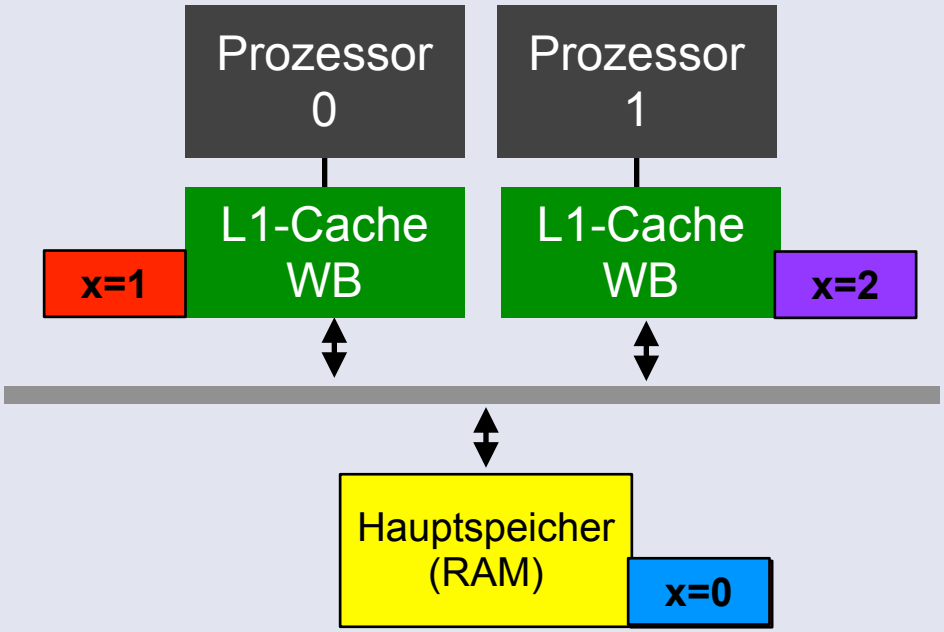


Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert

Ansatz 2:
"write back" (WB)-Caches



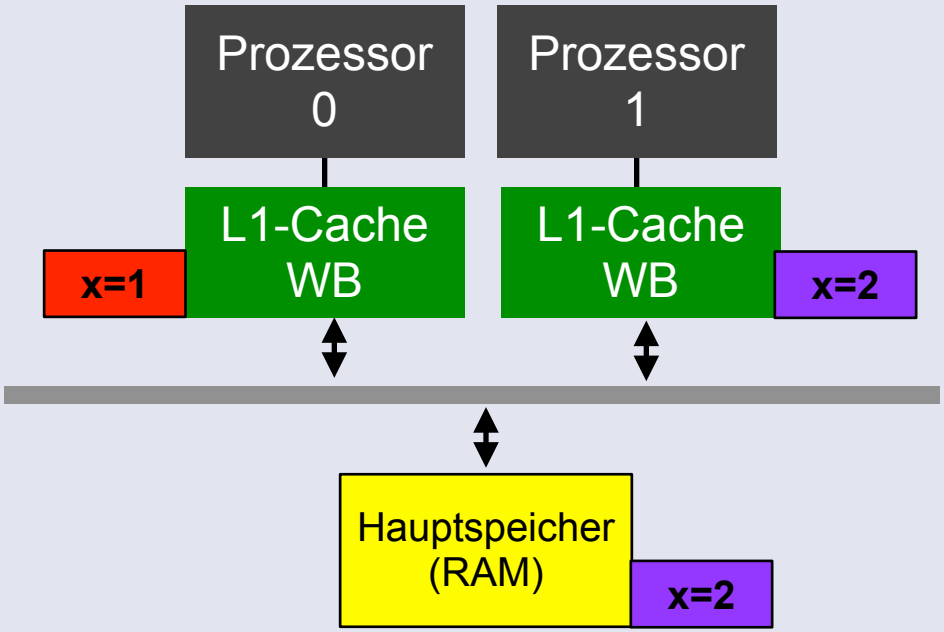
Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert

Prozessor 1 schreibt x = 2
x = 2 im Cache gespeichert

Ansatz 2:
"write back" (WB)-Caches



Prozessor 0 liest x
x = 0 im Cache gespeichert

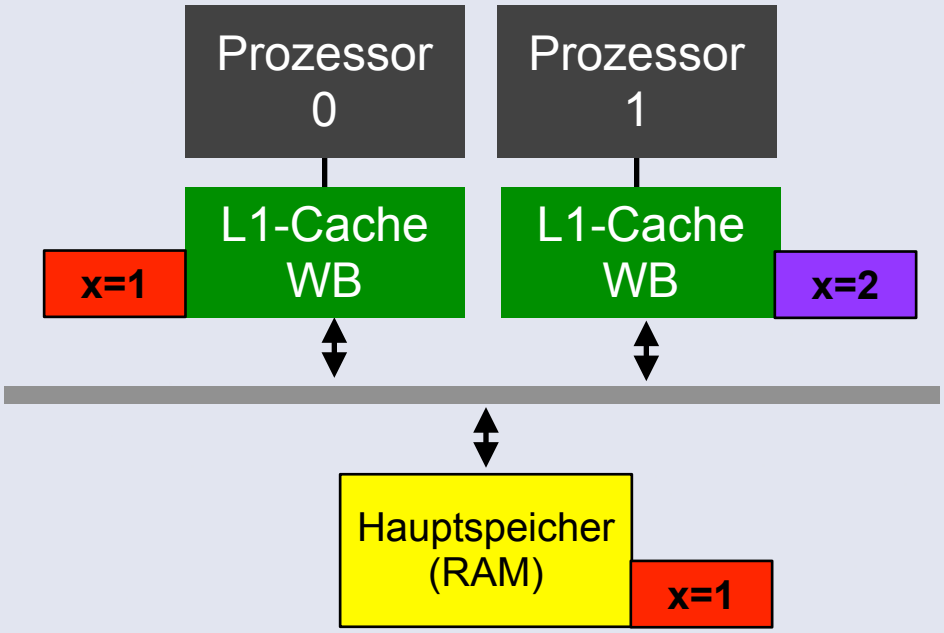
Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert

Prozessor 1 schreibt x = 2
x = 2 im Cache gespeichert

Writeback von Prozessor 1
x = 2 im Hauptspeicher

Ansatz 2:
"write back" (WB)-Caches



Prozessor 0 liest x
x = 0 im Cache gespeichert

Prozessor 1 liest x
x = 0 im Cache gespeichert

Prozessor 0 schreibt x = 1
x = 1 im Cache gespeichert

Prozessor 1 schreibt x = 2
x = 2 im Cache gespeichert

Writeback von Prozessor 1
x = 2 im Hauptspeicher

Writeback von Prozessor 0
x = 1 im Hauptspeicher

Der spätere Schreibzugriff x=2 von Prozessor 1 geht verloren!

Beide Beispiele verletzen die SWMR-Invariante!

Problem 1

CPU1 verwendet einen nicht mehr aktuellen ("*stale*") Wert, der bereits von CPU0 verändert wurde

– Lösung: Invalidiere Kopien in Caches vor der Ausführung der Schreiboperation!

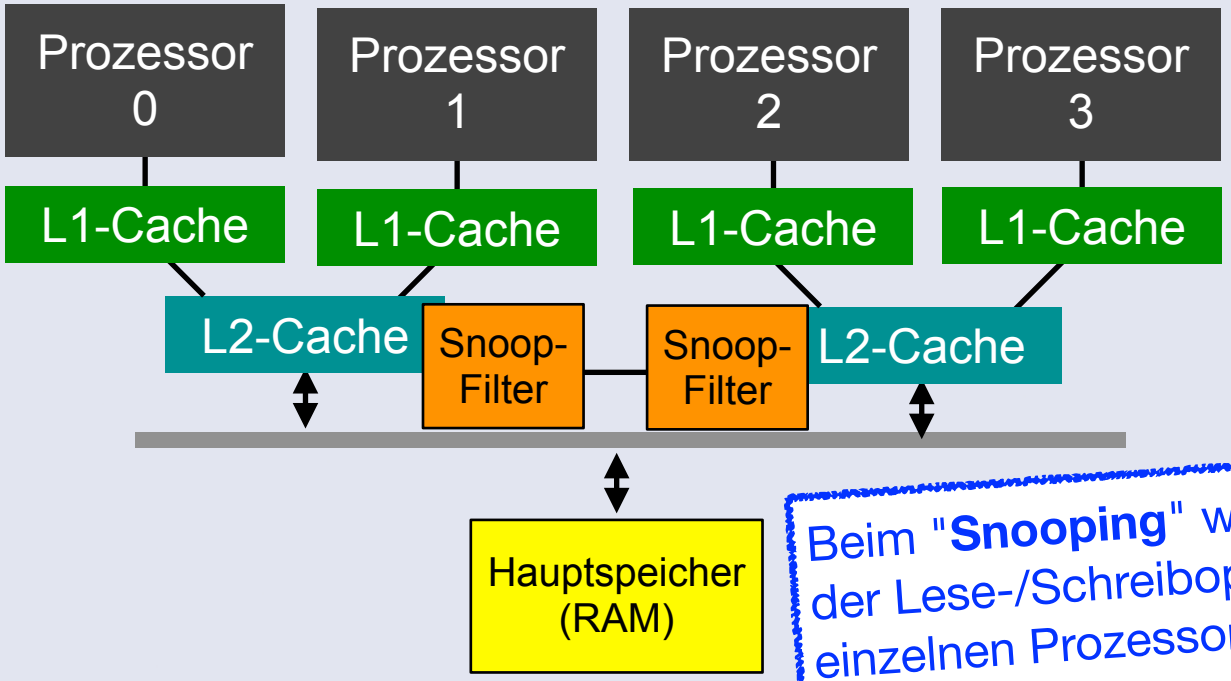
Problem 2

Inkorrekte Reihenfolge des Zurückschreibens modifizierter Cache-Zeilen

– Lösung: Verbieten der Existenz von mehr als einer modifizierten Kopie!

Entwurfsraum von Kohärenzprotokollen

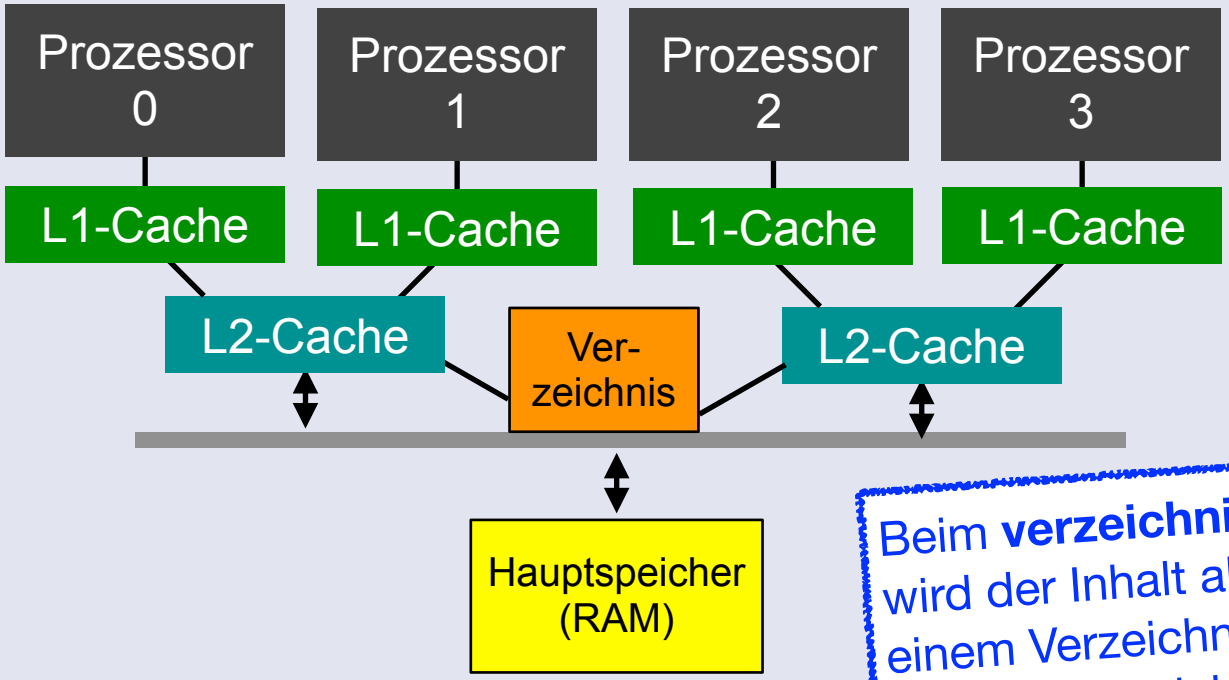
- **Snooping- vs. verzeichnisbasiert**



Beim "**Snooping**" werden die Adressen der Lese-/Schreiboperationen der einzelnen Prozessoren an die Caches der anderen Prozessoren weitergereicht, diese prüfen, ob sie eine Kopie der Daten besitzen und antworten entsprechend

Entwurfsraum von Kohärenzprotokollen

- **Snooping- vs. verzeichnisbasiert**



Beim **verzeichnisbasierten** Ansatz wird der Inhalt aller Caches zentral in einem Verzeichnis verwaltet und bei Bedarf Nachrichten zur Invalidierung an einzelne Caches gesendet

Entwurfsraum von Kohärenzprotokollen

- **Snooping-basiert**
  - Alle für die Cache-Kohärenz relevanten Informationen werden an alle Prozessoren gesendet (*broadcast*)
  - Jeder Prozessor liest diese Informationen mit und reagiert bei Bedarf:
    - Invalidiert eigener Cache-Zeile(n), deren Adresse(n) von anderen Prozessoren beschrieben wurden
  - **Geeignet für kleine Systeme (wenige Prozessoren)**
- **Directory-basiert**
  - Verwendet zentrales Verzeichnis, um Überblick über den aktuellen "Eigentümer" einer Cache-Zeile zu erhalten
  - Aktualisierung von Kopien dieser Cache-Zeile in Caches anderer CPUs
    - Caches aller CPUs können ***gleichzeitig*** aktualisiert werden (erfordert weniger Datentransfers für abwechselnde Lese-/ Schreiboperationen)
    - Mehrfaches Schreiben erfordert mehrfache Aktualisierungen (aufeinanderfolgende Schreibopratationen verursachen mehr Datentransfer)
  - **Geeignet auch für große Systeme mit vielen Prozessoren**

Invalidierung vs. Aktualisierung

- **Invalidierungs-basiert**
  - Nur Cache misses bei Schreiboperationen werden auf den Prozessorbus geleitet (geeignet für *write back*-Caches)
  - Folgende Schreiboperationen sind *write hits*
  - Gut geeignet für mehrfache Schreiboperationen auf die selbe Cache-Zeile vom selben Prozessor
- **Update-basiert**
  - Alle Kopien einer Cache-Zeile bleiben ein *cache hit*, nachdem ein Prozessor diese Zeile geschrieben hat
  - Die Aktualisierungen müssen zwischen den einzelnen Prozessoren weitergeleitet werden
- **Es existieren hier auch hybride Ansätze**

Grundlegendes Kohärenzprotokoll: MSI

Jede Cache-Line befindet sich in einem der folgenden Zustände:

- ***Modified* (M) – modifiziert**
  - Keine Kopien existieren in anderen Caches, die lokale Kopie wurde geändert
  - Der Hauptspeicherinhalt ist damit nicht mehr aktuell (*stale*)
- ***Shared* (S) – geteilt**
  - Nicht veränderte Kopien existieren in einem oder mehreren Caches
  - Der Hauptspeicherinhalt ist aktuell
- ***Invalid* (I) – ungültig**
  - Nicht im Cache vorhanden

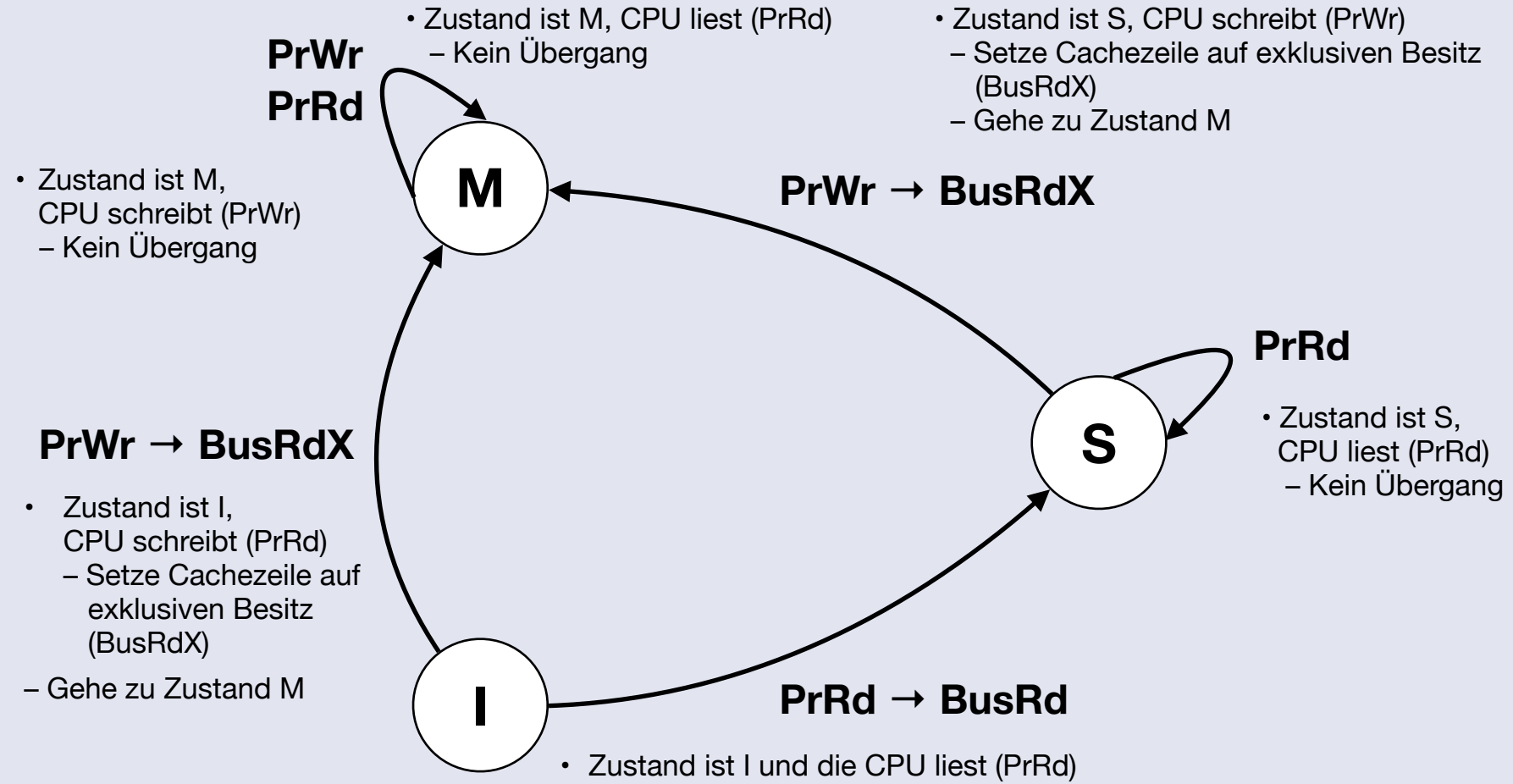
- Zustände werden aus Sicht des Cache-Controllers mitverfolgt
  - Dieser sieht Ereignisse von:
    - Lokalem Prozessor → Prozessor-Transaktion
    - Anderen Prozessoren → *snoop*-Transaktionen

Zustandsübergänge bei MSI

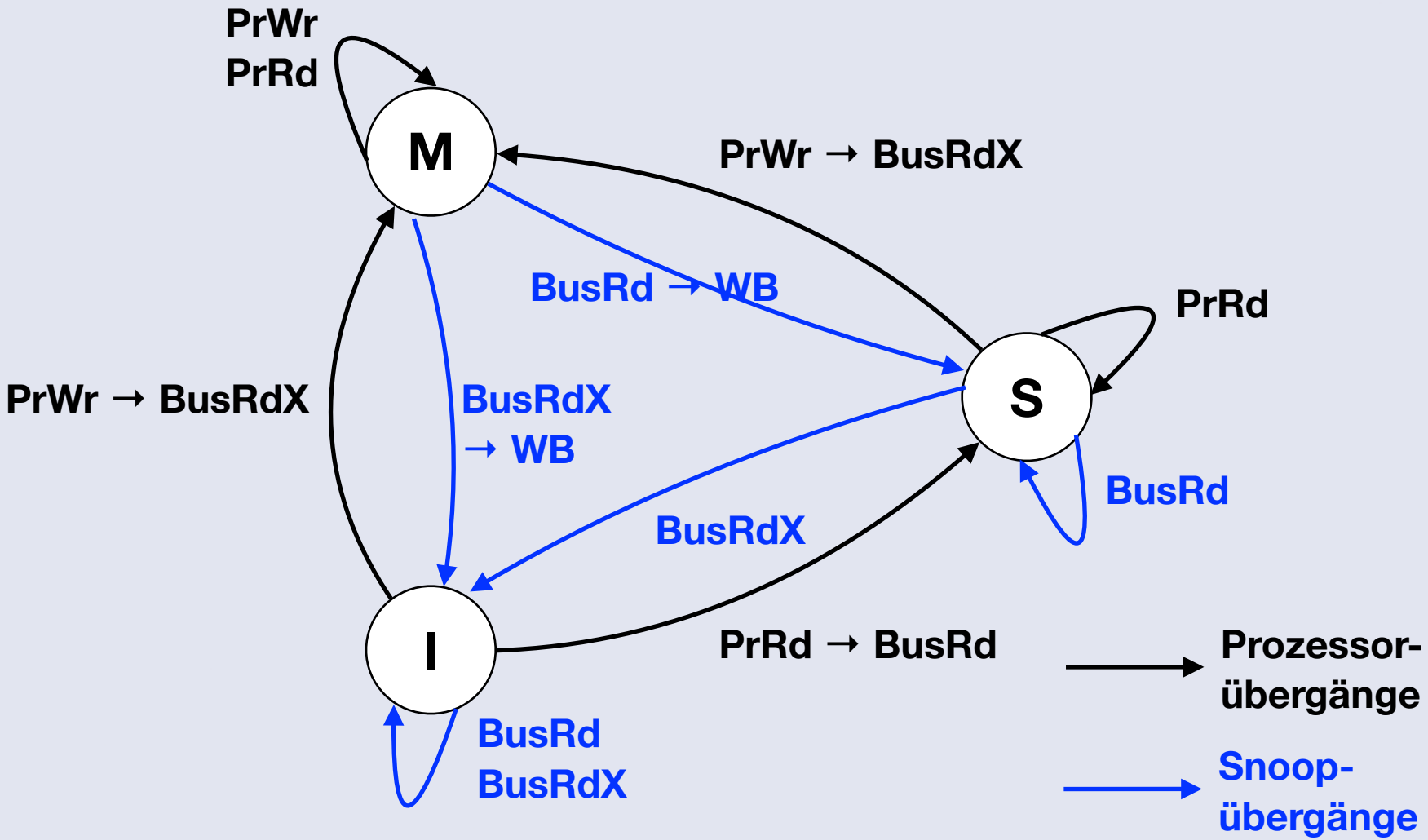
Jede Cachezeile ist in einem der drei Zustände:

- Modified (M) – modifiziert
- Shared (S) – geteilt
- Invalid (I) – ungültig

Prozessorzustandsübergänge bei MSI



Zustandsübergänge bei MSI



MSI: Zustandsübergänge bei *snoop*

Empfangen eines Lese-*snoops* (BusRd) für eine Cache-Zeile

- Wenn in M, schreibe Cache-Zeile zurück in Hauptspeicher (WB) und nimm Übergang zu S
- Wenn in S, dann kein Übergang (bleibt in S)

Empfangen eines Exklusive-Besitz-*snoops* (BusRdX)

- Wenn in M, schreibe Cache-Zeile zurück in Hauptspeicher (WB) und nimm Übergang zu I
- Wenn in S, dann invalidiere die Cache-Zeile und nimm Übergang zu I

MESI: Probleme von MSI lösen

Ein übliches Speicherzugriffsmuster ist:

- Lies Variable A: S
- Modifziere A: BusRdX wird gesendet, Übergang S → M

Invalidierungs-Nachricht überflüssig, wenn kein anderer Cache eine Kopie von A beinhaltet

Lösung: Hinzufügen eines Exclusive (E)-Zustands → MESI-Protokoll

- Es existieren keine Kopien in anderen Caches
- Der Hauptspeicherinhalt ist aktuell

Varianten von MESI werden von den meisten aktuellen Prozessoren verwendet

- Es gibt weitere Verbesserungen, z.B. MOESI

Speicherkonsistenz: Nebenläufige Programme

Globale Variable:

```
int i = 0;
int k = 0;
```

Programm 1:

```
i = 1;
if (i>1) k = 3;
```

||

Programm 2:

```
i = i + 1;
if (k==0) k = 4;
```

Speicherkonsistenz: Nebenläufige Programme

Globale Variable:

```
int i = 0;
int k = 0;
```

Programm 1:

```
i = 1;
if (i>1) k = 3;
```

||

Programm 2:

```
i = i + 1;
if (k==0) k = 4;
```

Assemblerversionen (x86):

```
 mov $1, [%i]
 cmp [%i], $1
 jgt end
 mov $3, [%k]
end:
```

||

```
 inc [%i]
 cmp [%k], $0
 jne end
 mov $4, [%k]
end:
```

Speicherkonsistenzmodelle

Ein **Speicherkonsistenzmodell…**

definiert *korrektes Verhalten* bei Lade- und Speicheroperationen auf geteiltem Speicher in Hinblick darauf, wie Operationen auf unterschiedliche Speicherzellen in Bezug auf andere Operationen sichtbar werden

Verschiedene Speicherkonsistenzmodelle existieren:

- Komplexe Modelle können performanter sein
- Einige Hardwareplattformen unterstützen verschiedene Modelle (SPARC)

Wichtige Begriffe:

- **Programmreihenfolge** (der Operationen eines Prozessors):
  Die Reihenfolge von Speicherzugriffen auf jedem Prozessor wird durch das ausgeführte Programm (also durch Software) bestimmt

- **Sichtbarkeitsreihenfolge** (aller Operationen im System):
  Reihenfolge von Speicherzugriffen, die von einem oder mehreren Prozessoren beobachtet werden

Sequentielle Konsistenz (SC)

“The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
A multiprocessor satisfying this condition will be called **sequentially consistent.”**

[Lamport 1979]

Anforderungen für sequentielle Konsistenz:

- **Einhaltung der Programmreihenfolge**
  – Jede CPU führt Speicheroperationen in Programmreihenfolge durch

- **Einhaltung der Atomizität** (Unteilbarkeit):
  – Der Hauptspeicher bedient Operationen *nacheinander* (nicht gleichzeitig)
  – Speicheroperationen scheinen *atomar* im Hinblick auf andere Speicheroperationen zu sein (eine Speicheroperation wird nicht durch eine andere unterbrochen)

Sequentielle Konsistenz wurde in MIPS R10000 RISC-Prozessoren implementiert

Sequentielle Konsistenz: Beispiel 1

CPU 0	CPU 1	
`[A] = 1;` (i)	`u = [B];` (ii)	[A],[B]: Speicher
`[B] = 1;` (iii)	`v = [A];` (iv)	u,v: Register

Sequentielle Konsistenz: Beispiel 1

CPU 0	CPU 1	
`[A] = 1; (i)`	`u = [B]; (ii)`	[A],[B]: Speicher
`[B] = 1; (iii)`	`v = [A]; (iv)`	u,v: Register

Resultat: **(u,v) = (1,1)**

Sequentiell konsistent – Reihenfolge: `i, iii, ii, iv`

Sequentielle Konsistenz: Beispiel 1

CPU 0	CPU 1	
`[A] = 1;` (i)	`u = [B];` (ii)	[A],[B]: Speicher
`[B] = 1;` (iii)	`v = [A];` (iv)	u,v: Register

Resultat: (u,v) = (1,1)
Sequentiell konsistent – Reihenfolge: `i, iii, ii, iv`

Resultat: **(u,v) = (1,0)**
Nicht sequentiell konsistent – Reihenfolge: `iii, ii, iv, i`
Mißachtung der Programmreihenfolge bei CPU0 (oder 1)

Sequentielle Konsistenz: Beispiel 2

CPU 0	CPU 1	
`[A] = 1; (i)`	`[B] = 1; (ii)`	[A],[B]: Speicher
`u = [B]; (iii)`	`v = [A]; (iv)`	u,v: Register

Sequentielle Konsistenz: Beispiel 2

CPU 0	CPU 1	
`[A] = 1; (i)`	`[B] = 1; (ii)`	[A],[B]: Speicher
`u = [B]; (iii)`	`v = [A]; (iv)`	u,v: Register

Resultat: **(u,v) = (1,1)**

Sequentiell konsistent – Reihenfolge: `i, ii, iii, iv`

Sequentielle Konsistenz: Beispiel 2

CPU 0	CPU 1	
`[A] = 1;` (i)	`[B] = 1;` (ii)	[A],[B]: Speicher
`u = [B];` (iii)	`v = [A];` (iv)	u,v: Register

Resultat: (u,v) = (1,1)
Sequentiell konsistent – Reihenfolge: `i, ii, iii, iv`

Resultat: **(u,v) = (0,0)**
Nicht sequentiell konsistent – Reihenfolge: `iii, iv, i, ii`
Mißachtung der Programmreihenfolge bei CPU0 (oder 1)

Schwächere Konsistenzmodelle

Für sequentielle Konsistenz müssen folgende Operationen in Programmreihenfolge erfolgen:

– Lesen → Lesen
– Lesen → Schreiben
– Schreiben → Lesen
– Schreiben → Schreiben

Diese Abhängigkeiten in der Programmreihenfolge werden auch entsprechend als Speicheroperationen sichtbar

Schwächere Modelle weichen die Reihenfolge-Anforderungen an einige oder alle von diesen auf

Aufweichen von Schreiben→Lesen oder Schreiben→Schreiben

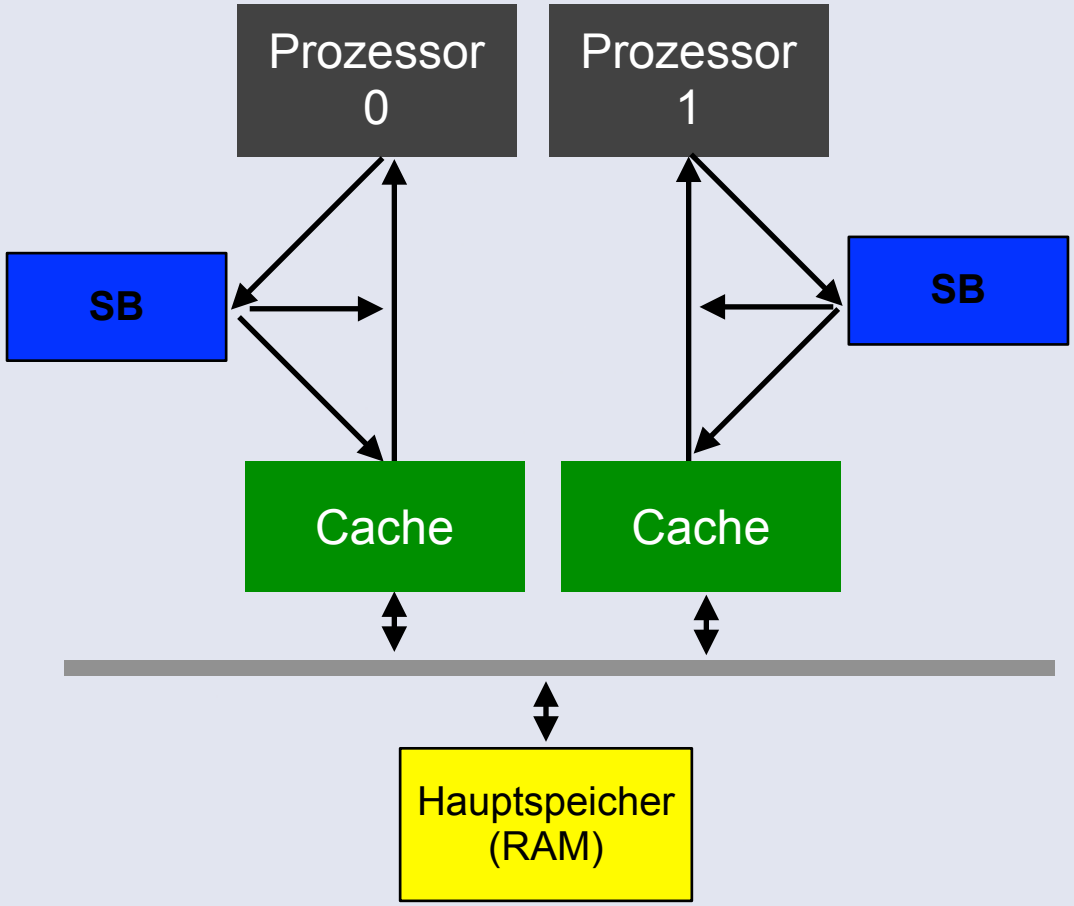
Aufweichen von Schreiben → Lesen:
(spätere Leseoperationen können frühere Schreiboperationen überholen)

- Schreiboperation gefolgt von Leseoperation kann in unterschiedlicher Reihenfolge stattfinden (*out-of-order*)
- Typische Hardware, die dies realisiert: Speicherpuffer (*store buffer*)
  - Schreiboperationen müssen warten, bis der zugehörige Prozessor die Cache-Zeile besitzt
  - Leseoperationen können im Schreibpuffer wartende Schreiboperationen "überholen"
  - Nutzen: "Verstecken" der *Schreiblatenz*

Aufweichen von Schreiben → Schreiben:
(spätere Schreiboperationen können frühere Schreiboperationen überholen)

- Aufeinanderfolgende Schreiboperationen können in unterschiedlicher Reihenfolge stattfinden
- Typische Hardware, die dies realisiert: Speicherpuffer, die Schreibzugriffe zusammenfassen (*coalescing store buffer*)

Schreibpuffer – Store Buffer (SB)



Schreibpuffer optimieren Schreiboperationen in den Hauptspeicher und/oder in Caches, um Zugriffe auf die Verbindungen zwischen den Prozessoren (z.B. Busse) zu optimieren

Prozessoren können mit der Programmausführung fortfahren, bevor eine Schreiboperation abgeschlossen wurde

Schreibweiterleitung (*Store forwarding*) ermöglicht Leseoperationen des lokalen Prozessors, die ausstehenden Schreiboperationen im SB zu sehen

Schreibpuffer ist **unsichtbar für andere Prozessoren**

FIFO vs. keine FIFO: Der Schreibpuffer kann Schreiboperationen kombinieren oder deren Reihenfolge permutieren

Aufweichen aller Programmreihenfolgen

Zusätzlich zu bisher betrachteten Aufweichungen:

- Aufweichen von **Lesen→Lesen**
  - spätere Leseoperationen können frühere Leseoperationen überholen
- Aufweichen von **Lesen→Schreiben**
  - spätere Schreiboperationen können frühere Leseoperationen überholen
- Beispiele für diese Speicherkonsistenzmodelle:
  - Weak Ordering (WO)
  - Release Consistency (RC)
  - DEC Alpha
  - SPARC V9 Relaxed Memory Model (RMO)
  - PowerPC
  - Itanium (IA-64)

Total Store Order (TSO)

Diese Speicheroperationen finden in **Programmreihenfolge** statt:

- Lesen → Lesen
- Lesen → Schreiben
- Schreiben → Schreiben

Speicheroperationen, die **geänderte Reihenfolge** haben dürfen:

- Schreiben → Lesen
  (spätere Leseoperationen können frühere Schreiboper. überholen)
  - Weiterleitung ausstehender Schreiboperationen an den Schreibpuffer an nachfolgende Leseoperationen für die selbe Speicherzelle

- Schreibpuffer ist eine FIFO

Vergleich TSO und Sequentielle Konsistenz

CPU 0	CPU 1
`[A] = 1;` (i)	`[B] = 1;` (ii)
`u = [A];` (iii)	`v = [B];` (iv)
`w = [B];` (v)	`x = [A];` (vi)

Resultat: **(u,v,w,x) = (1,1,0,0)**

- Nicht möglich bei SC!
- Möglich bei TSO:
  Reihenfolge: iii, iv, v, vi, i, ii
  b1 liest [A] aus dem eigenen Schreibpuffer

Programmierung…

Viele Hochsprachen stellen dem Programmierer ein standardisiertes Speicherkonsistenzmodell zur Verfügung:

- C/C++ 2011 und später
- Java

Grundlegendes Modell:

- **Garantie der sequentiellen Konsistenz** für Programme, die keine *data race*-Bedingungen aufweisen (SC-DRF):
  - Ein Programm, das keine *data* races enthält, wird sequentiell konsistent ausgeführt
  - Der Compiler fügt bei Bedarf Synchronisationsoperationen ein, falls die grundlegende Architektur keine sequentielle Konsistenz garantiert

Definition: **Data Race** (informell):

- Mehrere Threads eines Programms greifen auf eine Speicherzelle ohne Synchronisation zu, wobei eine Thread eine Schreiboperation ausführt

Referenzen

[1] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood:
A Primer on Memory Consistency and Cache Coherence, second ed.
Synthesis Lectures on Computer Architecture #49,
Morgan & Claypool Publishers, 2020

[2] Sarita Adve, Kourosh Gharachorloo:
Shared memory consistency models: a tutorial,
IEEE Computer, vol. 29, no. 12, pp. 66-76, Dec. 1996

[3] Paul E. McKenney:
Memory Ordering in Modern Microprocessors
Linux Journal, Issue 136, 2005

[4] Leslie Lamport:
How to Make a Multiprocessor Computer That Correctly Executes
Multiprocess Programs,
IEEE Transactions on Computers, vol. C-28, no. 9, pp. 690-691, Sept. 1979