Liefert das Protokoll Kohärenz?

- Benötigt: Konstruktion einer (totalen) Ordnung der Speicheroperationen unter Beachtung der Programmordnungen

- Voraussetzung: atomare Bus-Transaktionen und Speicheroperationen
  - alle Phasen einer Bus-Transaktion sind abgeschlossen bevor nächste Bus-Transaktion startet
  - Prozessor wartet auf Abschluss der Speicheroperation, bevor nächste Speicheroperation initiiert wird
  - die Invalidierung entfernter Caches wird zeitgleich zur Bus-Transaktion angenommen

Kohärenz beim Write-Through

Beweis: Zeige Erfüllung der notwendigen Eigenschaften

1) *Write*-Propagierung
2) *Write*-Serialisierung

zu 1)

alle *Writes* gehen sofort auf den Bus, wodurch

- *Writes* durch die Ordnung ihres Erscheinens auf dem Bus serialisiert sind (Bus-Ordnung)
- Invalidierungen in den Caches auch dieser Bus-Ordnung folgen

zu 2)

durch 1) und Atomarität der Bus-Operationen sehen alle Caches die *Write*-Operationen in gleicher, eindeutiger Reihenfolge

Ordnung von Reads

Aber wie erscheinen *Reads* in dieser Ordnung?

- wichtig, da Prozessoren *Writes* durch *Reads* sehen und damit bestimmt ist, ob die *Write*-Serialisierung erfüllt ist
- aber *Read-Hits* treten unabhängig von Bus-Transaktionen auf und sind deshalb nicht auf dem Bus sichtbar

- ***Read-Miss***: erscheint auf dem Bus und sieht letztes *Write* in Bus-Ordnung
- ***Read-Hit***: erscheint nicht auf dem Bus, aber:
  - gelesener Wert ist noch als aktuell im Cache, weil entweder
    - der Prozessor das letzte *Write* auf der Adresse selber durchführte oder
    - der Prozessor auf der Adresse zuletzt ein *Read-Miss* hatte.
  - beide Transaktionen erschienen zuvor auf dem Bus

  Daher sehen auch *Read-Hits* gelesene Werte in Bus-Ordnung

Probleme mit Write-Through

- ***Write-Through*** hat hohe Anforderungen an die Bus-Bandbandbreite
  - alle *Writes* aller Prozessors gehen über den Bus zum Speicher
  - betrachte 2.66 GHz, 0,5 CPI und 15% aller Instruktionen sind 8-Byte Speicheroperationen
    - ⇒ jeder Prozessorkern erzeugt 800 Mio. Schreiboperationen pro Sekunde bzw. 6.4 GiB Daten pro Sekunde
    - ⇒ 50 GiB/s Bus wird bereits mit ungefähr 8 Prozessorkernen gesättigt

  ⇒*Write-Through* wird selten in SMPs eingesetzt

- ***Write-Back*** absorbiert die meisten *Writes* durch *Cache-Hits*
  - *Write-Hits* gehen nicht auf den Bus
  - aber wie wird trotzdem *Write-Propagierung* und Serialisierung gewährleistet?
  - anspruchsvollere Protokolle erforderlich (großer Design-Space)

Zuerst betrachten wir weitere Aspekte des Orderings

Bestimmung einer Ordnung

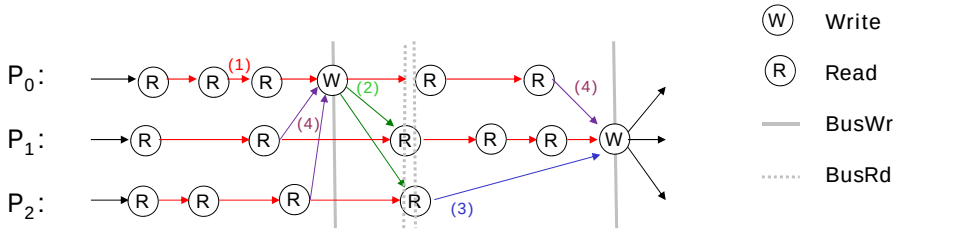
(1) Speicheroperation M_2 ist nachfolgend einer Speicheroperation M_1, falls
- beide Operationen durch den selben Prozessor angestoßen wurden und
- M_2 der Operation M_1 in Programmordnung folgt.

(2) eine *Read* Op ***R*** ist nachfolgend einer *Write* Op ***W***, falls
- ***R*** eine Bus-Transaktion nachfolgend zu ***W*** erzeugt.

(3) eine *Write Op* ***W*** ist nachfolgend einem *Read* oder *Write* ***M***, falls
- ***M*** eine Bus-Transaktion erzeugt und
- diese vor der Bus-Transaktion vom *Write* ***W*** liegt.

(4) eine *Write Op* ***W*** ist nachfolgend einer *Read Op* ***R***, falls
- ***R*** keine Bus-Transaktion erzeugt (*Read-Hit*) und
- das ***R*** ist nicht durch eine andere Bus-Transaktion vom *Write* ***W*** separiert

Beispiel Ordering



Im Beispiel nur Operationen auf gleiche Speicherzelle

Allgemein
- *Writes* bilden eine partielle Ordnung
- Bus ordnet *Read-Misses*
- Bus erzwingt aber keine Ordnung der *Read-Hits*
  - jede Ordnung von *Reads* in *Writes* ist möglich, solange die Programmordnung gewahrt bleibt

Speicherkonsistenz

Bisher: Writes auf eine Adresse erscheinen für alle Prozessoren in der gleichen Ordnung.

- Wann genau wird nun ein *Write* sichtbar?
- Wie wird die Ordnung zwischen *Writes* und *Reads* auf verschiedenen Prozessoren gewährleistet (Synchronization)?

/* Initially: A and flag = 0 */

P_1

```
A = 1;
flag = 1;
```

P_2

```
while (flag == 0);   /*polling */
print A;
```

- Unsere Intuition ist nicht durch die Kohärenz abgedeckt.
- Speicherzugriffe die von einem Prozessor auf unterschiedliche Adresse wirken sollen aber auch einer Ordnung folgen.
- Kohärenz hilft hier nicht - da diese nur auf eine Adresse anwendbar ist.

Weiteres Beispiel zu „Ordnung“

/* Initially: A and B = 0 */

P_1

```
(1a) A = 1;
(1b) B = 2;
```

P_2

```
(2a) print B;
(2b) print A;
```

- Was sagt unsere Intuition?
- Was immer dabei heraus kommt, wir benötigen ein *Ordering*-Modell für eine eindeutige Semantik
  - auch über verschiedene Adressen hinweg
  - Programmierer können somit über mögliche Resultate nachdenken

⇒ Notwendigkeit eines Speicherkonsistenzmodells

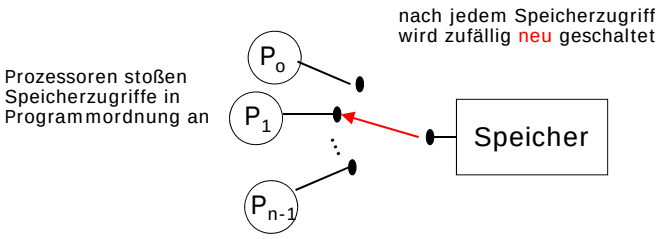
Speicherkonsistenzmodell

- spezifiziert die Anforderungen an eine Ordnung der Speicheroperationen wie diese von beliebigen Prozessoren gegenseitig erscheinen können
  - bestimmt welche Ordnung zulässig ist
  - Menge an möglichen Ergebnissen einer Programm-ausführung wird sinnvoll eingeschränkt
- ohne diese Anforderungen kann nur wenig über die Ausführung eines Programms gesagt werden
- Implikationen für Programmierer und Systemdesigner
  - Programmierer: Korrektheit und mögliche Ergebnisse
  - Systemdesigner: z.B. Möglichkeit zu einem Instruktions-Reorderings durch Compiler und Hardware

⇒ *Art Vertrag zwischen Programmierer und System*

Konsistenz

Einfaches Modell eines Parallelrechners:



Annahme: kein Cache und nur ein zentraler Speicher verfügbar

- totale Ordnung durch Verschachtelung (*Interleaving*) der Zugriffe verschiedener Prozesse
- erhält Programmordnung und Speicheroperationen erscheinen von allen Prozessoren als atomar [issue, execute, complete]
- Intuition des Programmierers ist gewahrt

Definiert damit auch alle möglichen Resultate (das Ergebnis) einer Ausführung eines parallelen Programms.

Strikte Konsistenz

Anforderung:

- Jedes *Read* liefert den vom letzten *Write* geschriebenen Wert

Problem:

- Globale Taktung/Zeit zur Bestimmung der letzten Operation notwendig.

Verteilte Systeme kennen aber keine globale Zeit

=> ein schwächeres Modell für die Konsistenz wird benötigt

Definition: Sequentielle Konsistenz (SC)

Def.:

Ein Multiprozessorsystem ist **sequentiell konsistent,** falls das Ergebnis der Ausführung gleich einer Ausführung in beliebiger serialisierter Ordnung ist, bei der auf jedem Prozessor die Operationen in der vorgegebenen Programmordnung erscheinen.

Lamport, 1979

Frage: Was bedeutet dabei Programmordnung?

Was ist eine Programmordnung?

Intuition: Ordnung in der Operationen im Source-Code stehen
- direkte Übersetzung von Source-Code in Maschinensprache
- höchstens eine Speicheroperation pro Instruktion

- aber nicht unbedingt gleich der Ordnung, die durch den Compiler der Hardware vorgegeben wird
  - Z.B. *Reordering* durch Code-Optimierung
- und auch nicht gleich der Ordnung die dann die Hardware realisiert !!
  - Z.B. dynamisches *Reordering* im *Instruction Buffer*

Was ist nun die Programmordnung konkret?
- von Ebene des Betrachters abhängig

Im Folgenden wird die Sicht des Programmierers eingenommen

Beispiel: Sequentielle Konsistenz

Programmordnung ist die Ordnung in der eine Ausführung erscheint und nicht unbedingt in der konkret ausgeführt wird.

/* Initially: A and B = 0 */

P_1
```
(1a) A = 1;
(1b) B = 2;
```

P_2
```
(2a) print B;
(2b) print A;
```

- mögliche Ergebnisse von (A,B): (0,0), (1,0), (1,2);
  unmögliches Ergebnis von (A,B) unter SC: (0,2)
- durch Programmordnung wissen wir, dass 1a→1b und 2a → 2b
  - Ausgabe A = 0 impliziert 2b→1a, wodurch 2a→1b folgt
  - Ausgabe B = 2 impliziert 1b→2a, wodurch $\{P_1\} \rightarrow \{P_2\}$ folgt
- aber Ausführung **1b→1a→2b→2a ist SC, obwohl nicht in Programmordnung**, da das Ergebnis (1,2) aussieht wie 1a→1b→2a→2b
- Ausführung 1b→2a→2b→1a (0,2) ist hingegen nicht SC

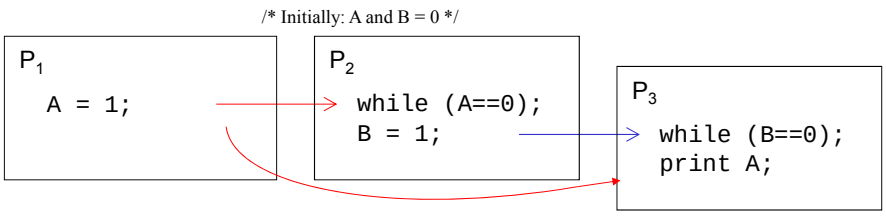
Implementierung von SC

- zwei Anforderungen
  - **Einhaltung der Programmordnung**
    - von einen Prozess angestoßene Speicheroperationen müssen in Programmordnung (für sich und andere) erscheinen
  - **Atomarität von Speicheroperationen**
    - in der totalen Ordnung sollte eine Speicheroperation allen Prozessen als *Complete* erscheinen, bevor die nächste Speicheroperation angestoßen wird
    - muss garantieren, dass die totale Ordnung konsistent über alle Prozesse ist

Trick: verwende atomare *Writes*

Write-Atomarität

- *Write*-Atomarität: Die Stelle, in der ein *Write* innerhalb der totalen Ordnung erscheint, muss für alle Prozesse gleich sein.
  - eine Operation, die von einem Prozess nach dem Schreiben von W ausgeführt wird, darf von anderen Prozessen keinesfalls vor dem W sichtbar sein
  - erweitert damit *Write*-Serialisierung um das Schreiben mehrerer Prozesse



- Problem, wenn P_2 Schleife verlässt und Wert B verändert hat, aber P_3 den neuen Wert von B und den alten Wert von A sieht (z.B. aus seinem Cache)
- SC + Transitivität führt zur Ausgabe des Werts A=1

Mehr Formal

- Die Programmordnung eines Prozesses liefert eine partielle Ordnung der Operationen
- eine beliebige Verschachtelung aller partiellen Ordnungen definiert eine totale Ordnung aller Operationen
- einige totale Ordnungen können SC sein (SC definiert aber keine bestimmte Verschachtelung)

SC-Execution:

- eine Ausführung eines Programms ist SC, falls das erzielte Ergebnis gleich dem einer möglichen totalen Ordnung (Verschachtelung) ist .

SC-System:

- ein System ist SC, falls jede mögliche Programmausführung auf dem System eine *SC-Execution* ist.

Hinreichende Bedingungen für SC

- jeder Prozess stößt seine Speicher-Ops in Programmordnung an
- eine Operation, die ein *Write* angestoßen hat, wartet auf dessen *Complete*, erst danach kann eine weitere Operation vom Prozess angestoßen werden
- eine Operation, die ein *Read* angestoßen hat, wartet auf das *Complete* des *Write*, dessen Wert durch das *Read* zurück gegeben werden soll. Erst danach kann eine weitere Operation angestoßen werden (setzt *Write*-Atomarität voraus)
  - hinreichende, aber nicht notwendige Bedingungen
  - Compiler sollten wegen SC kein *Reordering* machen
    - wird aber gemacht: Schleifentransformation, Registerzuweisung, usw.
  - auch wenn In-Order angestoßen wird, Hardware kann dies zerstören
    - *Write-Buffer*, *Out-Of-Order-Execution-Units, usw.*
  - Aus Gründen hoher Performance kümmern sich Einprozessorsysteme nur um Abhängigkeiten bzgl. einer Adresse
    - macht damit hinreichende Bedingungen sehr restriktiv

Behandlung des Ordering

angenommen, der Compiler macht kein *Reordering*

- benötigte Hardware-Mechanismen
  - erkennen von *Write-Completion* (*Read-Completion* ist einfach)
  - ermöglichen der *Write*-Atomarität
- für alle Protokolle und Implementierungen muss gezeigt werden
  - wie Kohärenz, insb. *Write*-Serialisierung, gewährleistet wird,
  - ob hinreichende Bedingungen für SC zutreffen (*Write-Completion* und *Write*-Atomarität),
  - wie SC sichergestellt wird, falls hinreichende Bedingungen nicht erfüllt sind.

zentraler Bus/Interconnect vereinfacht diese Betrachtung

Speicherkohärenz vs. Konsistenz

- Schwache Konsistenz
  - Konsistenz der Speicherzugriffe ist nicht zu allen Zeitpunkten gewährleistet
  - Synchronisationspunkte sind durch Prozessor zwingend einzuhalten
  - Realisierbar durch im Programm eingefügte Operationen
- Sequentielle Konsistenz
  - Schreiboperationen sind für alle Prozessoren in der selben Reihenfolge sichtbar
  - In Hardware realisierbar
- Kohärenz
  - sequenzielle Konsistenz bezogen auf jeweils eine Speicherzelle

SC mit Write-Through Caches

Write-Through Caches liefern SC und nicht nur Kohärenz

Beweisidee:

- erweitern der Argumente für Kohärenz
  - *Write*- und *Read-Misses* auf allen Adressen sind durch den Bus in Bus-Ordnung serialisiert
  - falls *Read* geschriebenen Wert *W* zurückbekommt, ist *W* als *complete* garantiert, da dafür zuvor eine Bus-Transaktion erzeugt wurde
  - wenn *Write W* bzgl. eines Prozessors ausgeführt wurde, sind auch alle vorherigen *Writes* in Bus-Ordnung als *complete* anzunehmen

Design-Space für Snooping-Protokolle

- Write-Through liefert schlechte Performance, deshalb andere Verfahren notwendig

- Dabei aber keine wesentlichen Änderung von Prozessor, Cache, Hauptspeicher, …
  - Cache-Controller erweitern und Serialisierung durch Bus nutzen
- Besondere Protokolle für Write-Back-Caches erforderlich
  - Weitere Zustände
    - Modified (*Dirty)*: aktueller Wert ausschließlich in einem Cache
    - *Exclusive*: nur ein Cache mit Kopie, Hauptspeicher ist aktuell
    - *Owner*: verantwortlich für Lieferung des Blocks bei Anfrage
    - …
- Design-Space
  - Invalidierungs- und *Update*-basierte Protokolle
    - Andere Zustandsmengen und Zustandsübergänge

Idee: Invalidierungs-basiertes Protokoll

- Read oder Write mit Cache-Miss
  - *Read*-Bus-Transaktion bzw. *Write*-Bus-Transaktion
- Neuer Zustand *Shared* kennzeichnet mögliche Teilhaber (Kopien)
  - ermöglicht nachfolgende *Read* ohne Bus-Transaktion
  - nachfolgendes *Write* erfordert aber Bus-Transaktion
    - erzeugt eine *Read-Exclusive*-Bus-Transaktion (RdX)
    - macht das *Write* über den Bus sichtbar
      - *Erzeugt ggf. Zustandsänderungen in anderen Caches* (Invalidierung)
    - nur ein RdX kann auf einem Block zu einer Zeit erfolgreich sein; Serialisierung über Bus
- Schreiben auf Block im Zustand Modified ohne Bus-Transaktion möglich
- *Write-Back*-Transaktionen ebenfalls notwendig
  - Vorherige Speicheroperation führte zur Verdrängung des Blocks
  - Laufen aber zeitlich verzögert zu der Speicheroperation ab

Idee: Update-basiertes Protokoll

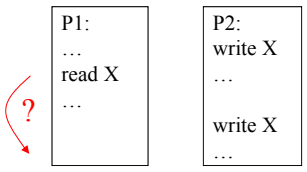
Write-Operation aktualisiert auch gleich den Wert in anderen Caches

⇒ Einführen einer neuen Bus-Transaktion „*Update*“

- Vorteil
  - andere Prozessoren haben keinen *Miss* beim nächsten Zugriff
    - beim Invalidierungs-Protokoll würde dieses zu einem *Miss* und einer weiteren Transaktionen führen
  - Nur eine Bus-Transaktion für gleichzeitiges *Update* mehrerer Caches
    - reduziert Bus-Nutzung, insbesondere, wenn nur der geschriebene Wert und nicht der ganze Block transferiert wird
- Nachteil
  - mehrere *Writes* vom gleichen Prozessor erzeugt ggf. mehrere *Update*-Transaktionen
    - beim Invalidierungs-Protokoll, erstes *Write* setzt *Exclusive*-Eigentumsrecht, weitere *Writes* erzeugen nur lokale Änderungen
- Detaillierte Untersuchung zu Trade-Offs wird sehr komplex

Invalidate versus Update

Wichtigste Frage bzgl. Programmverhalten:
Wurde ein Block zwischen zwei Schreiboperationen von einem anderen Prozessor gelesen?



- Invalidierungs-Protokoll
  - Ja ⇒ Reader bekommt ein Miss
  - Nein ⇒ Writes ohne weiteren Verkehr möglich
    und „entsorgt“ dabei nicht mehr genutzte Kopien
- *Update-Protokoll*
  - Ja ⇒ Reader bekommt kein Miss, falls zuvor eine Kopie vorhanden
    nur eine Bus-Transaktion um allen Kopien das Update zu schicken
  - Nein ⇒ ggf. nutzlose Updates zu anderen nicht mehr verwendeten Kopien
- Vorteil ist abhängig vom Programmverhalten und Hardware-Komplexität
- Invalidierungs-Protokoll ist sehr weit verbreitet
  - Es gibt Systeme die beide Protokolle verwenden, auch als hybride Implementierung

Invalidierungs- und Update-basierte Cache-Kohärenz-Protokolle

SC mit Write-Back Caches

Beweisidee:

- Behandlung von *Reads* wie beim *Write-Through*
  - *Read-Miss* ist in Bus-Ordnung serialisiert
  - *Read-Hit* liest den zuletzt geschriebenen Wert

- *Write-Miss* eines Prozessors
  - Cache muss gesamten Block aus dem Speicher lesen
  - Bus-Transaktion erforderlich => Serialisierung in Bus-Ordnung

- Write-Hit eines Prozessors
  - Verändert Wert im Cache
  - Falls notwendig, führe Bus-Transaktion aus
    - Kritische Aktion
    - Unterschiedliche Ansätze , abhängig vom konkreten Protokoll

Einfaches MSI-Writeback-Inval-Protokoll

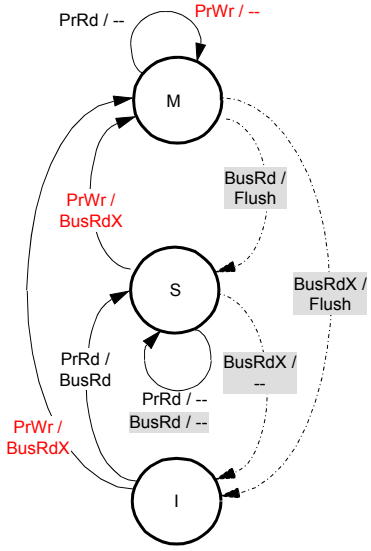
- Write-Back Cache
  - typischerweise *Write-allocate* bei einem Write-Miss
- Zustände
  - *Invalid* (I):
  - *Shared* (S): Kopien in einem oder mehreren Caches
  - *Dirty* or *Modified* (M): in genau einem Cache valider Wert
- Prozessor-Transaktionen:
  - PrRd (*Read*)
  - PrWr (*Write*)
- Bus-Transaktionen
  - BusRd: fragt nach Kopie ohne Intention zur Modifikation
  - **BusRdX**: fragt nach Kopie mit Intention zur Modifikation
  - BusWB: Write-Back (Block verdrängt), Update des Speichers
- Aktionen
  - Falls notwendig: Zustand ändern, Bus-Transaktion ausführen, Daten auf Bus legen

MSI Zustandsübergangsdiagramm

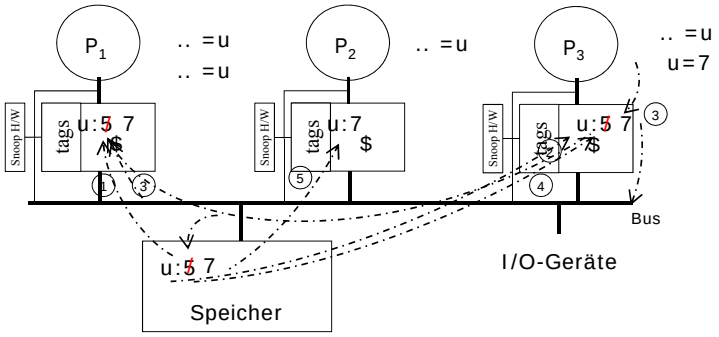
- Flush: Daten werden auf den Bus gelegt und daraufhin in den Hauptspeicher geschrieben
- BusRd, BusRdX: Speichersystem antwortet mit Daten falls kein Flush

- BusWB wird bei jeder Verdrängung einer Cache-Line angestoßen

"A / B"
Ereignis A / Transaktion oder Aktion B



Beispiel: MSI

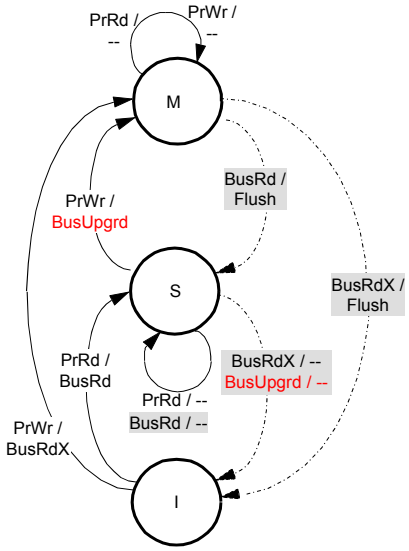


(1) P_1 liest *u* vom Hauptspeicher
(2) P_3 liest *u* vom Hauptspeicher
(3) P_3 beschreibt *u* mit 7 (Cache)
(4) P_1 liest *u* aus **P_3 Cache**
(5) P_2 liest *u* aus Hauptspeicher

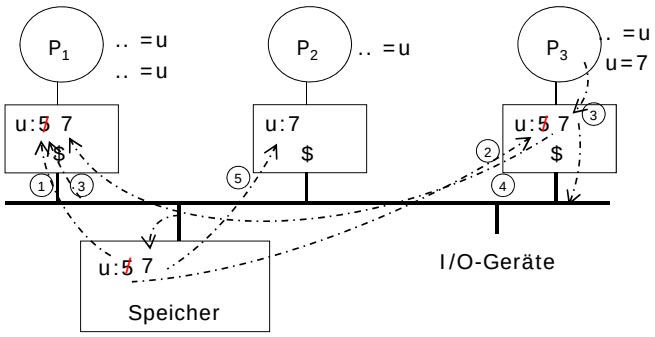
Aktion	*Pr-Aktion*	*Bus Aktion*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1	PrRd (1)	BusRd	Memory	S	-	-
2	PrRd (3)	BusRd	Memory	S	-	S
3	PrWr (3)	BusRdX	Memory	I	-	M
4a	PrRd (1)	BusRd		S	-	S
4b		Flush	Cache P_3			
5	PrRd (2)	BusRd	Memory	S	S	S

MSI Zustandsübergangsdiagramm (Upgrd)

BusUpgrd: Speichersystem antwortet nicht mit dem Block



Beispiel: MSI mit BusUpgrd



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

Aktion	*Pr-Aktion*	*Bus Aktion*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1	PrRd (1)	BusRd	Memory	S	-	-
2	PrRd (3)	BusRd	Memory	S	-	S
3	PrWr (3)	BusUpgrd	-	I	-	M
4a	PrRd (1)	BusRd		S	-	S
4b		Flush	Cache P_3			
5	PrRd (2)	BusRd	Memory	S	S	S

Beweis: Kohärenz + Konsistenz

Für das MSI-Protokoll werden betrachtet:

- Notwendige Bedingungen der Speicherkohärenz
  - Write-Propagierung
  - Write-Serialisierung
- Hinreichende Bedingungen der sequenziellen Konsistenz
  - Programmordnung
  - Write-Atomarität
  - Write-Completion

Erfüllung der Kohärenz (1)

- *Write*-Propagierung
  - *Write* mit BusRdX oder Bus-Upgrd ist für die anderen Caches sichtbar (Invalidierung, danach Zugriff nur mit einem BusRd möglich)
  - Write ohne Bus-Transaktion musste zuvor schon ein *Write* mit Bus-Transaktion durchgeführt haben, damit wird auch das letzte *Write* implizit erkannt
- *Write*-Serialisierung?
  - alle *Writes* (BusRdX / Bus-Upgrd) die auf dem Bus erscheinen sind in Bus-Ordnung
    - Schreiben in den Cache geschieht vor anderen Transaktionen des Caches, deshalb ebenso geordnet
    - Durch Snooping führen alle betroffenen Caches die gleichen Aktionen aus

Aber was ist mit Writes die nicht auf dem Bus sichtbar sind?

Erfüllung der Kohärenz (2)

Annahme: Write nicht sichtbar auf dem Bus

- zuvor durchgeführte *Write* muss vom gleichen Prozessor (P) stammen
  - vorheriges Write vom anderen Prozessor, dann wäre eine Bus-Transaktion notwendig geworden
- die nach der relevanten letzten Bus-Transaktionen liegende Sequenz an *Writes* muss vom Proz. P stammen
  - ergibt sich aus der vorherigen Argumentation
- die letzte relevante Bus-Transaktion wurde durch ein *Write* von Proz. P verursacht
  - diese Bus-Transaktion erschien in Bus-Ordnung
- jedes *Read* von P sieht die *Writes* in der richtigen Ordnung
- *Reads* anderer Prozessoren müssen entweder vor oder erst nach der Sequenz liegen
  - ansonsten wäre eine zusätzliche Bus-Transaktion notwendig geworden
  - zukünftige *Reads* erfordern eine Transaktion in Bus-Ordnung

Erfüllung der sequentiellen Konsistenz

Definition fordert konsistente globale Verschachtelung:

- Bus legt totale Ordnung der Bus-Transaktionen fest
  - Durch Snooping auch in allen Caches realisiert
- Zwischen zwei aufeinander folgenden Bus-Transaktionen
  - führen Prozessoren lokale *Reads*/*Writes* in Programmordnung aus
  - werden von P gesehene *Writes* folgendermaßen serialisiert:
    - durch eigene Programmordnung bei einem *Write* von P oder
    - durch eine diesbezüglich in Bus-Ordnung vorhergegangen Speicheroperation (mit für alle sichtbaren Bus-Transaktion)

Damit führt jedes verschachteln von Operationen der Prozessoren zu einer konsistenten totalen Ordnung innerhalb und zwischen diesen Abschnitten

MSI: Sequentielle Konsistenz

Beweis durch Erfüllung der hinreichenden Bedingungen

- *Write-Completion*
  - *a) Write* wird mit Bus-Transaktion abgeschlossen
    (Prozessoren/Cache-Controller warten auf Ende der Bus-Trans.)
  - *b) Write* wirkt nur auf eigenen Cache (Proz. wartet auf Cache)

- *Write-Atomarität*
  - Write erzeugt eine Bus-Transaktion die bis zum Abschluss der Transaktion auf dem Bus sichtbar ist,
    erst danach ist eine weitere Bus-Transaktion möglich
  - Ein Write ohne Bus-Transaktion ist nur für den ausführenden Prozessor relevant
  - sobald ein *Read* einen Wert eines *Writes* zurück gibt, wurde das *Write* bereits für alle anderen Prozessoren sichtbar