

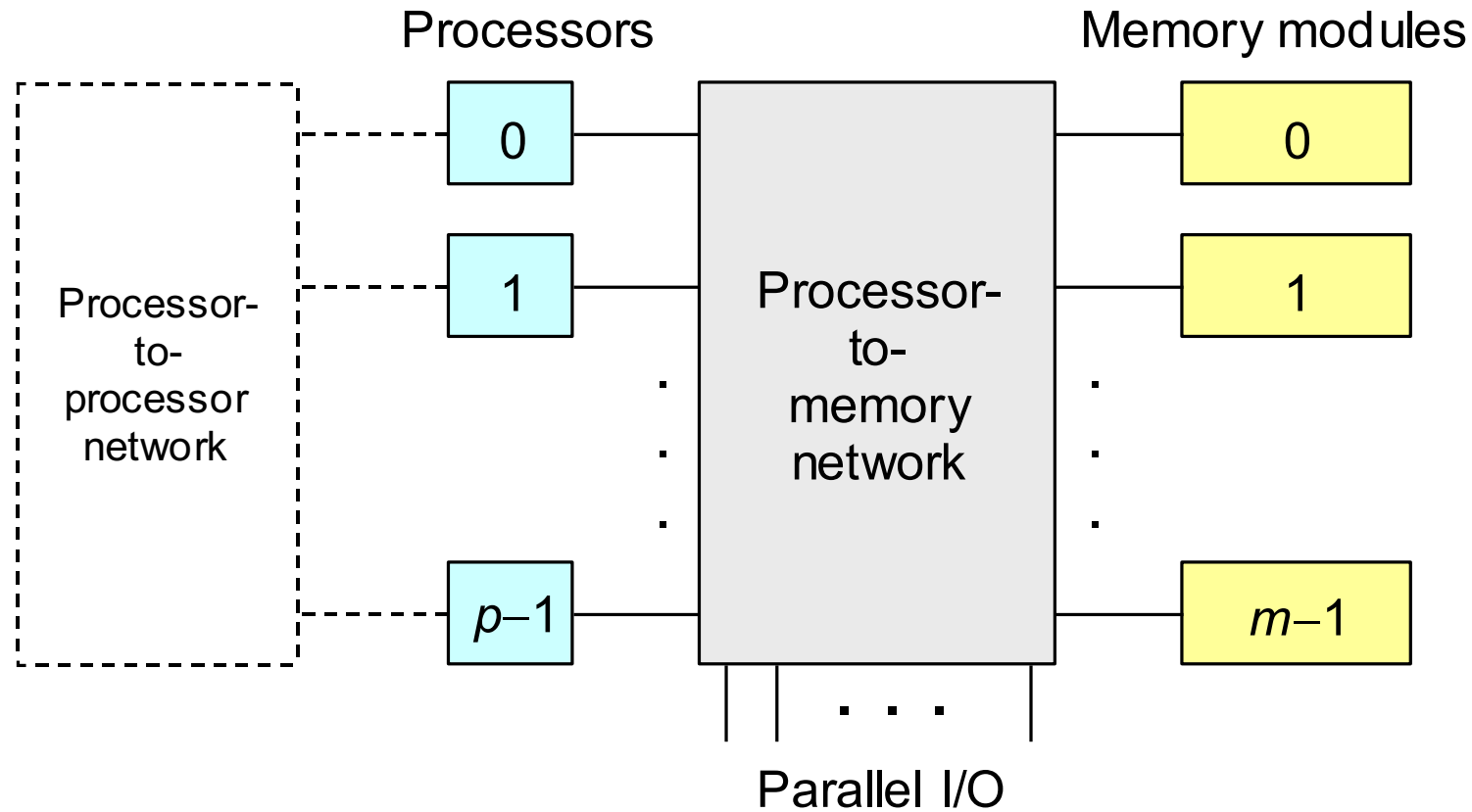
Calculatoare Numerice (2)

– Cursul 12 – Multiprocesoare 2

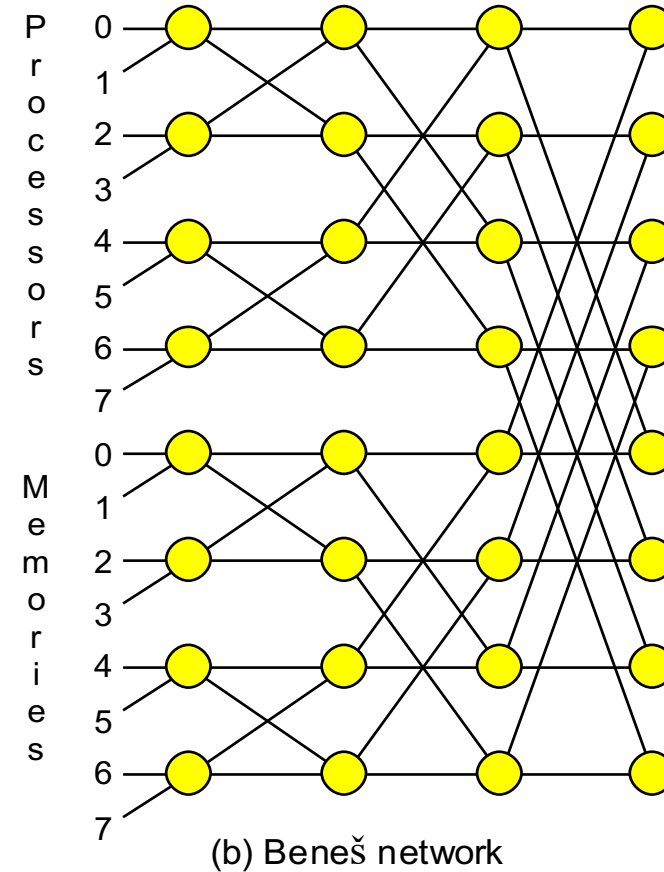
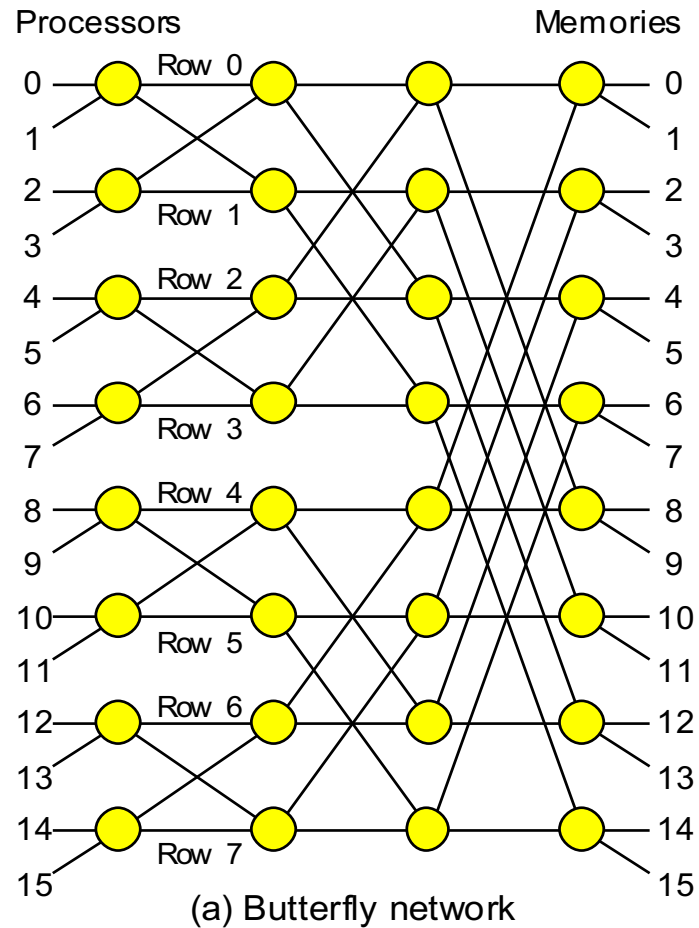
Facultatea de Automatică și Calculatoare
Universitatea Politehnica București



Memoria partajată centralizată

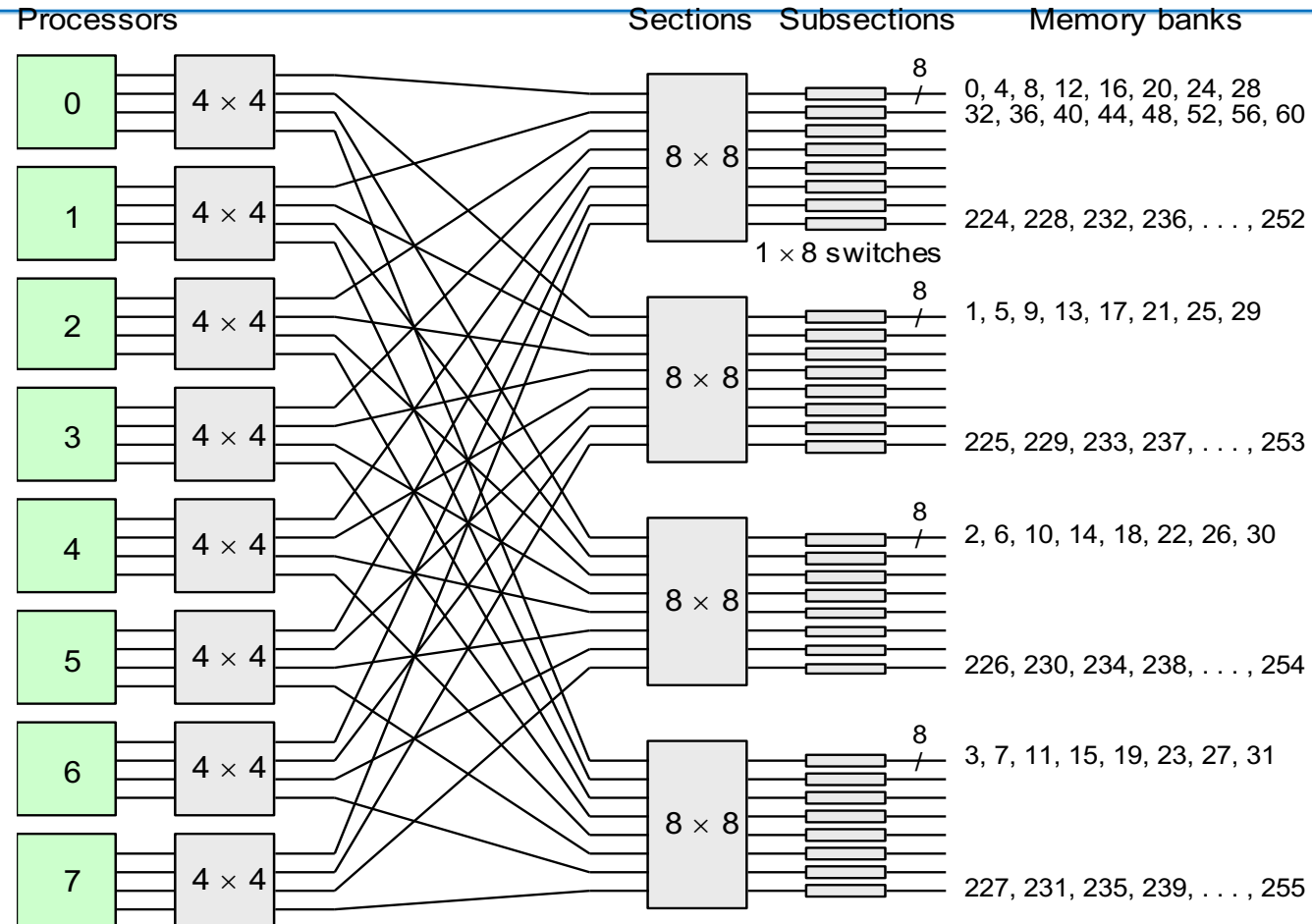


Structură multiprocesor cu memorie partajată



Rețelele fluture și Beneš: exemple de rețele de interconectare procesor-memorie

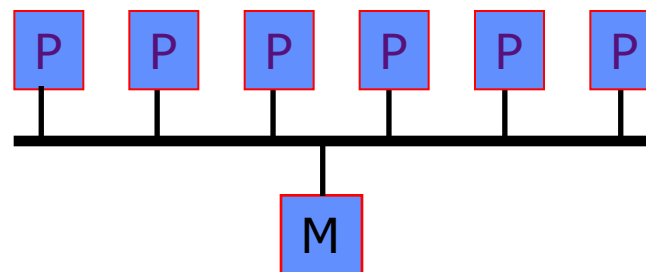
Rețele de interconectare Procesor-Memorie



Interconectarea a opt procesoare la 256 bancuri de memorie la Cray Y-MP (1988), un supercomputer cu procesoare vectoriale multiple

Consistența secvențială

Model de memorie



“ A system is *sequentially consistent* if 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 the order specified by the program”
Leslie Lamport

Sequential Consistency =
întreșesere arbitrară cu păstrarea ordinei
referințelor la memorie pentru programele secvențiale

Consistența secvențială

Task-uri secvențiale concurente: T1, T2
Variabile partajate: X, Y (inițial X = 0, Y = 10)

T1:
Store (X), 1 ($X = 1$)
Store (Y), 11 ($Y = 11$)

T2:
Load R₁, (Y)
Store (Y'), R₁ ($Y' = Y$)
Load R₂, (X)
Store (X'), R₂ ($X' = X$)

Care sunt răspunsurile corecte pentru X' și Y' ?

$(X', Y') \in \{(1, 11), (0, 10), (1, 10), (0, 11)\}$?

Dacă y este 11 atunci x nu poate fi 0

Consistența secvențială

Consistența secvențială impune mai multe contrângeri de ordonare de memorie ca și cele impuse de dependențele de memorie ale programelor uni-procesor (→)

Care sunt cele din exemplele noastre ?

T1:

Store (X), 1 ($X = 1$)
Store (Y), 11 ($Y = 11$)

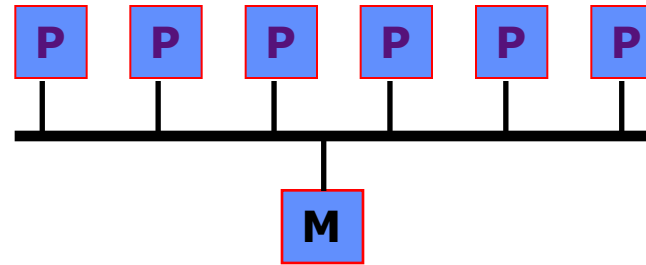
T2:

Load R₁, (Y)
Store (Y'), R₁ ($Y' = Y$)
Load R₂, (X)
Store (X'), R₂ ($X' = X$)

→ Cerințe adiționale SC

Poate un sistem cu cache și out-of-order execution să pună la dispoziție o imagine consistentă secvențial a memoriei?

Probleme în implementarea Consistenței Secvențiale



Implementarea CS este complicată de două probleme

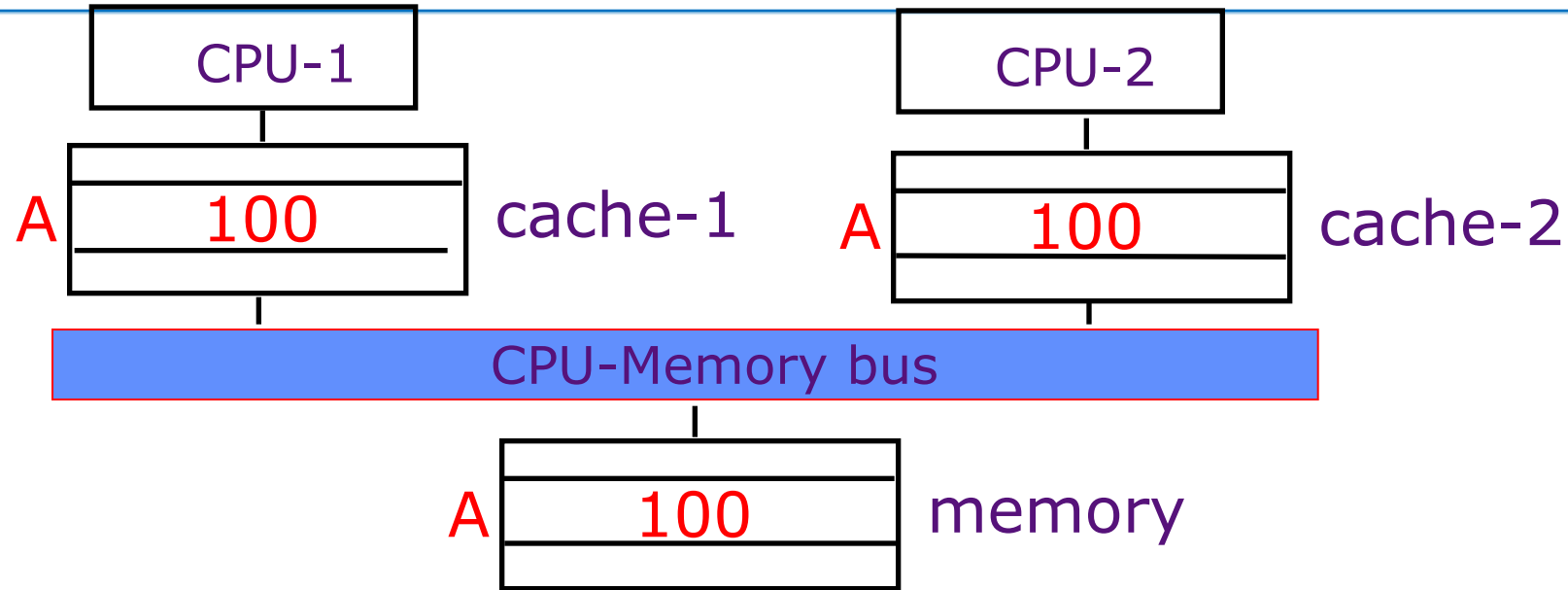
- Capabilități de execuție *Out-of-order*

Load(a); Load(b)	yes
Load(a); Store(b)	yes if $a \neq b$
Store(a); Load(b)	yes if $a \neq b$
Store(a); Store(b)	yes if $a \neq b$

- *Cache-uri*

Cache-urile pot preveni ca efectul unui store să fie văzut de alte procesoare

Consistența memoriei la SMP-uri



Presupunem că CPU-1 actualizează **A la 200**.

write-back: memoria și cache-2 au valori vechi

write-through: cache-2 are valoarea veche

Contează aceste valori neactualizate?

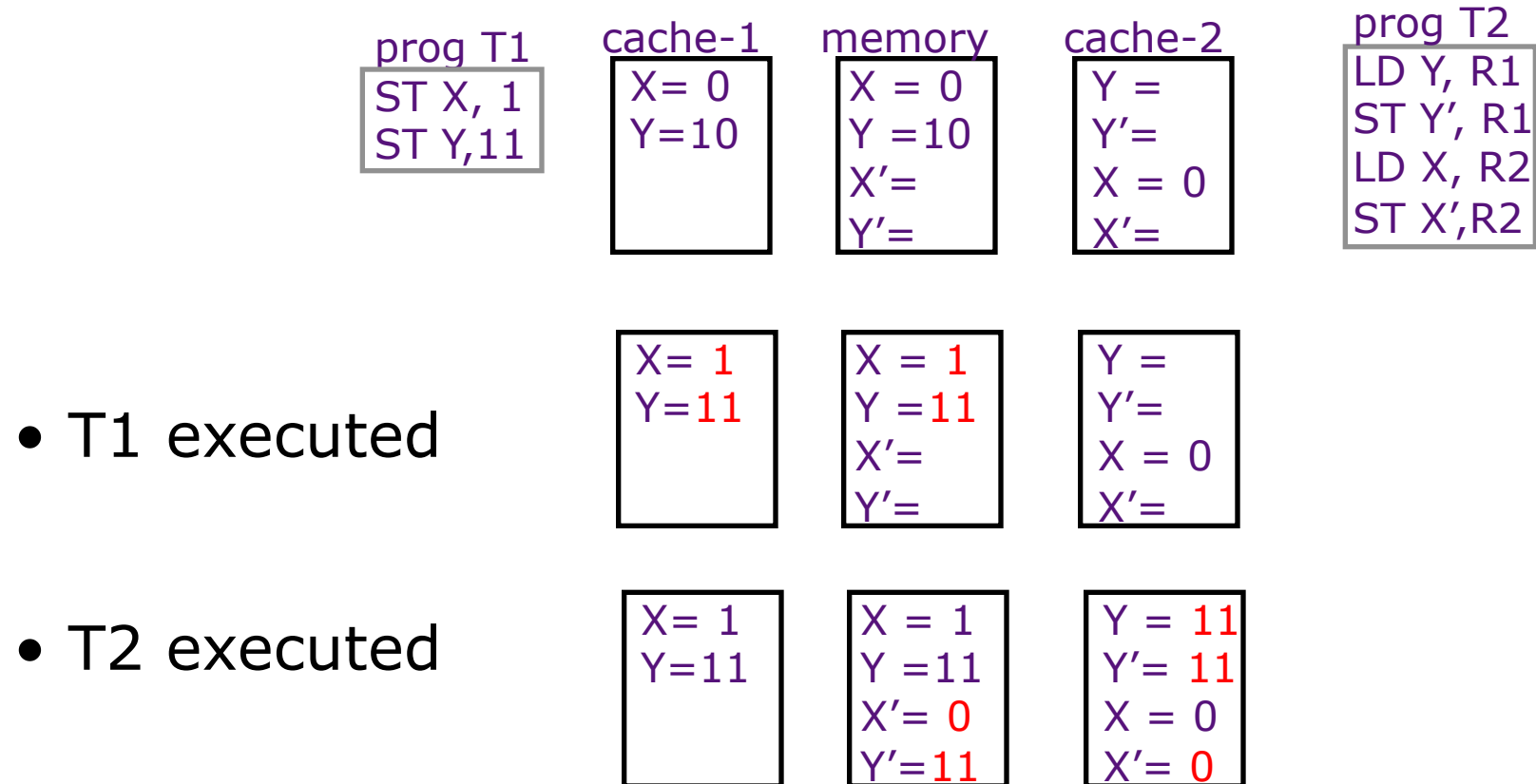
Cum este văzută memoria partajată de software?

Write-back Caches & SC

	prog T1	cache-1	memory	cache-2	prog T2
• T1 is executed	ST X, 1 ST Y, 11	X= 1 Y=11	X = 0 Y = 10 X'= Y' =	Y = Y'= X = X' =	LD Y, R1 ST Y', R1 LD X, R2 ST X', R2
• cache-1 writes back Y		X= 1 Y=11	X = 0 Y = 11 X'= Y' =	Y = Y'= X = X' =	
• T2 executed		X= 1 Y=11	X = 0 Y = 11 X'= Y' =	Y = 11 Y' = 11 X = 0 X' = 0	
• cache-1 writes back X		X= 1 Y=11	X = 1 Y = 11 X'= Y' =	Y = 11 Y' = 11 X = 0 X' = 0	
• cache-2 writes back X' & Y'		X= 1 Y=11	X = 1 Y = 11 X' = 0 Y' = 11	Y = 11 Y' = 11 X = 0 X' = 0	

inconsistent

Write-through Caches & SC



Nici cache-urile write-through nu mențin consistența secvențială

Menținerea consistenței secvențiale (CS)

CS este suficientă pentru programe tip
producer-consumer și cu excluziune mutuală
(e.g., Dekker)

Copiile multiple ale unei locații în diferite
Cache-uri pot cauza degradarea CS.

Este nevoie de suport hardware pentru

- doar un singur procesor la un moment dat are permisiune de write la o locație de memorie
- nici un procesor nu poate să încarce o copie a vechii locații după o scriere

⇒ Protocoale de coerență a cache-ului

Protocoale de menținere a coerenței cache pentru CS

write request:

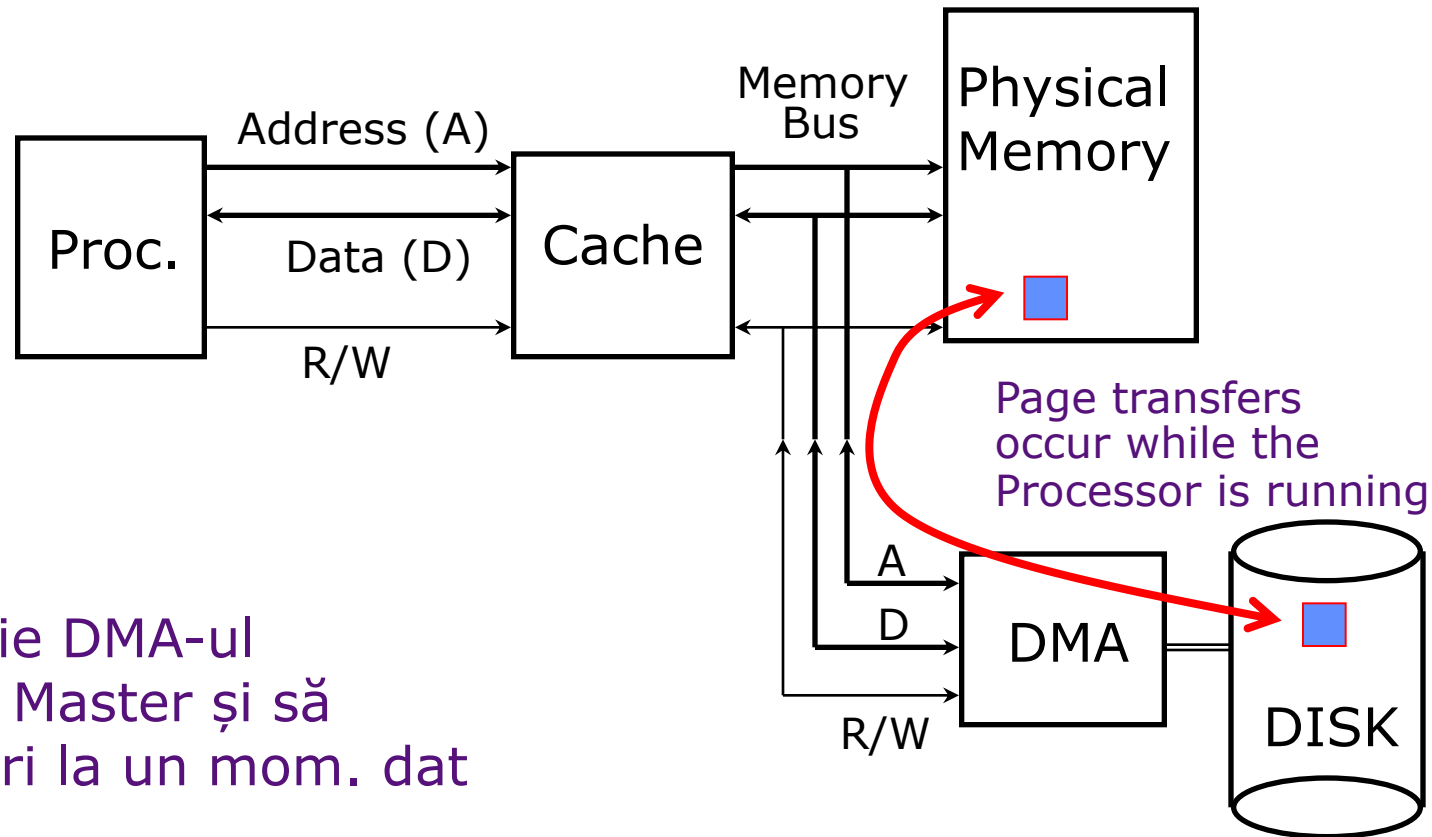
adresa este *invalidată (actualizată)* în toate cache-urile înainte (după) o operație de write

read request:

dacă o este găsită o copie "murdară" într-un cache, se efectuează un write-back înainte de citirea memoriei

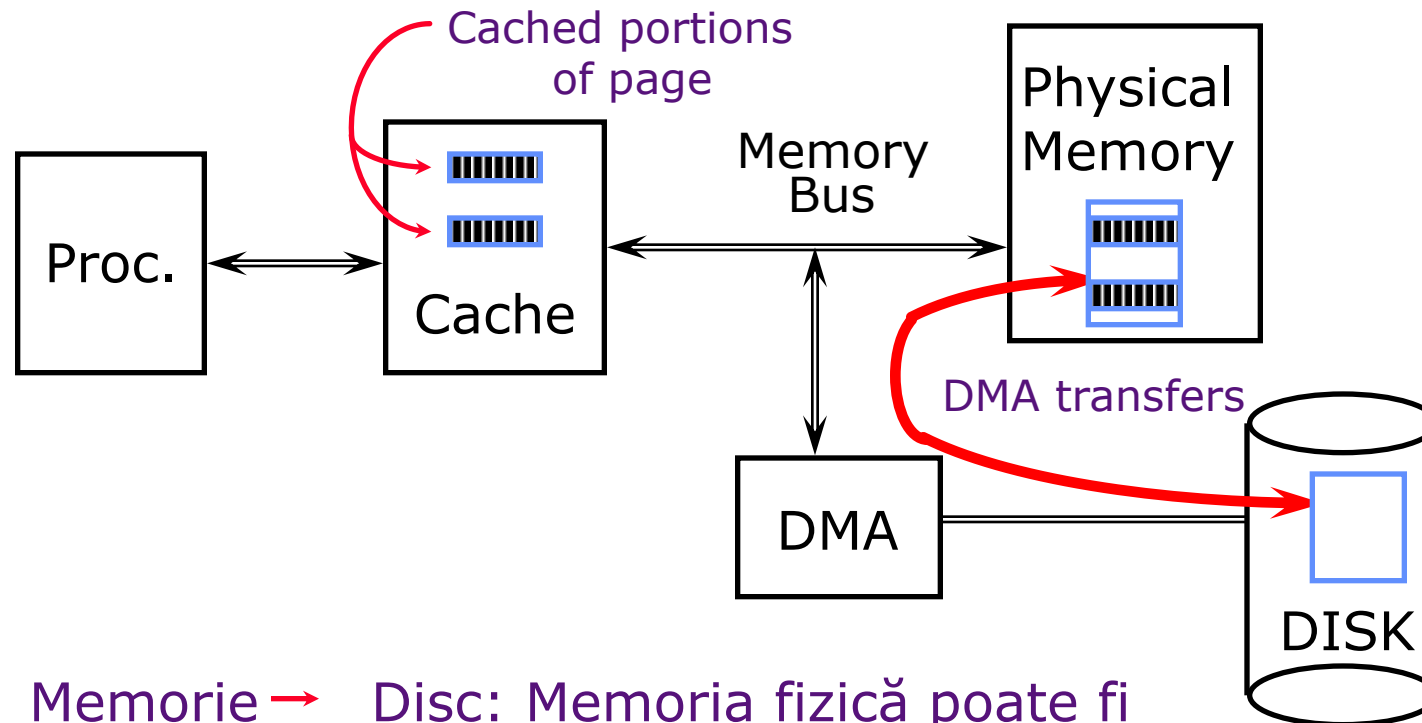
*Ne vom concentra pe protocoale de **Invalidare** și nu pe protocoale Update*

Warmup: Remembering Parallel I/O



Fie Cache-ul fie DMA-ul pot să fie Bus Master și să facă transferuri la un mom. dat

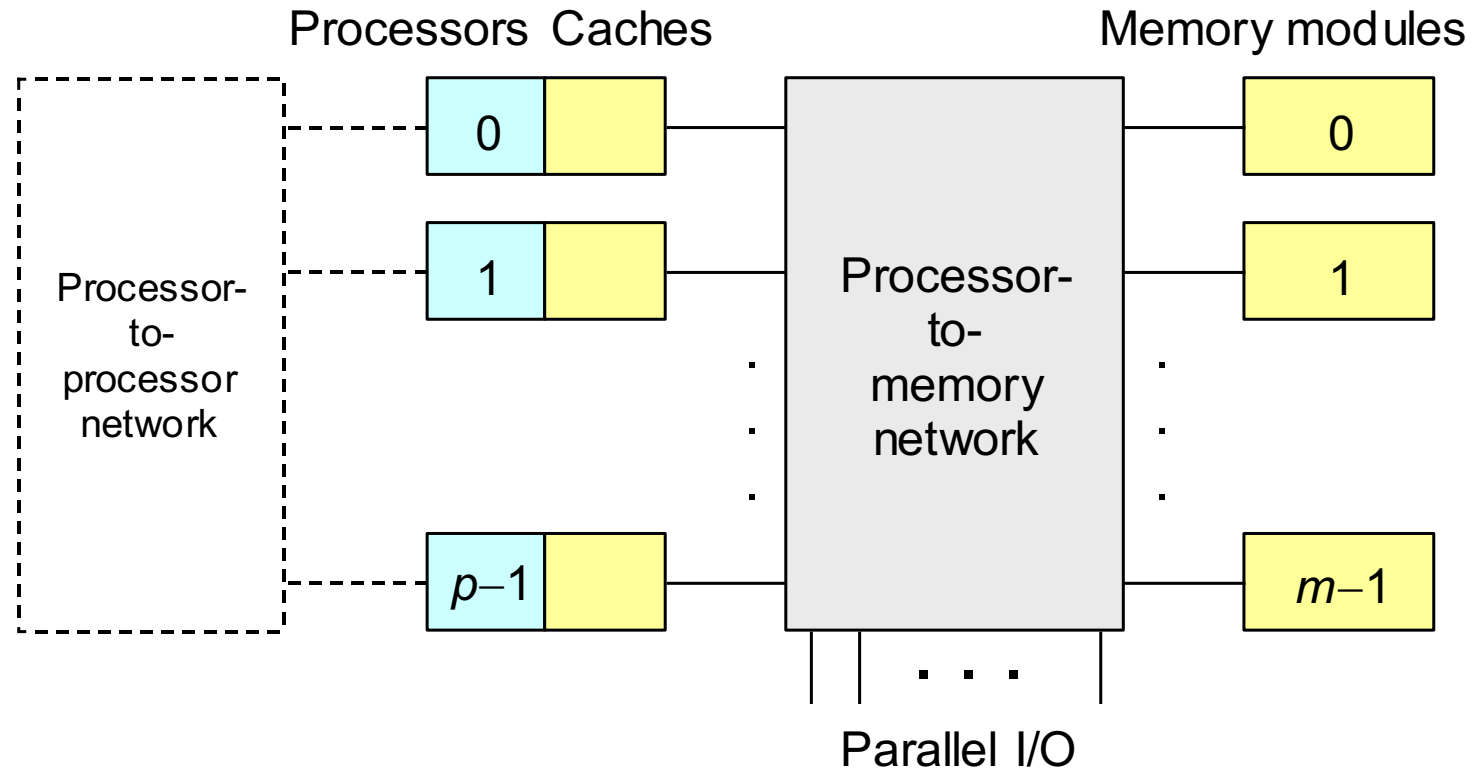
(DMA vine de la Direct Memory Access)



Memorie → Disc: Memoria fizică poate fi neactualizată dacă copia din Cache este "murdară"

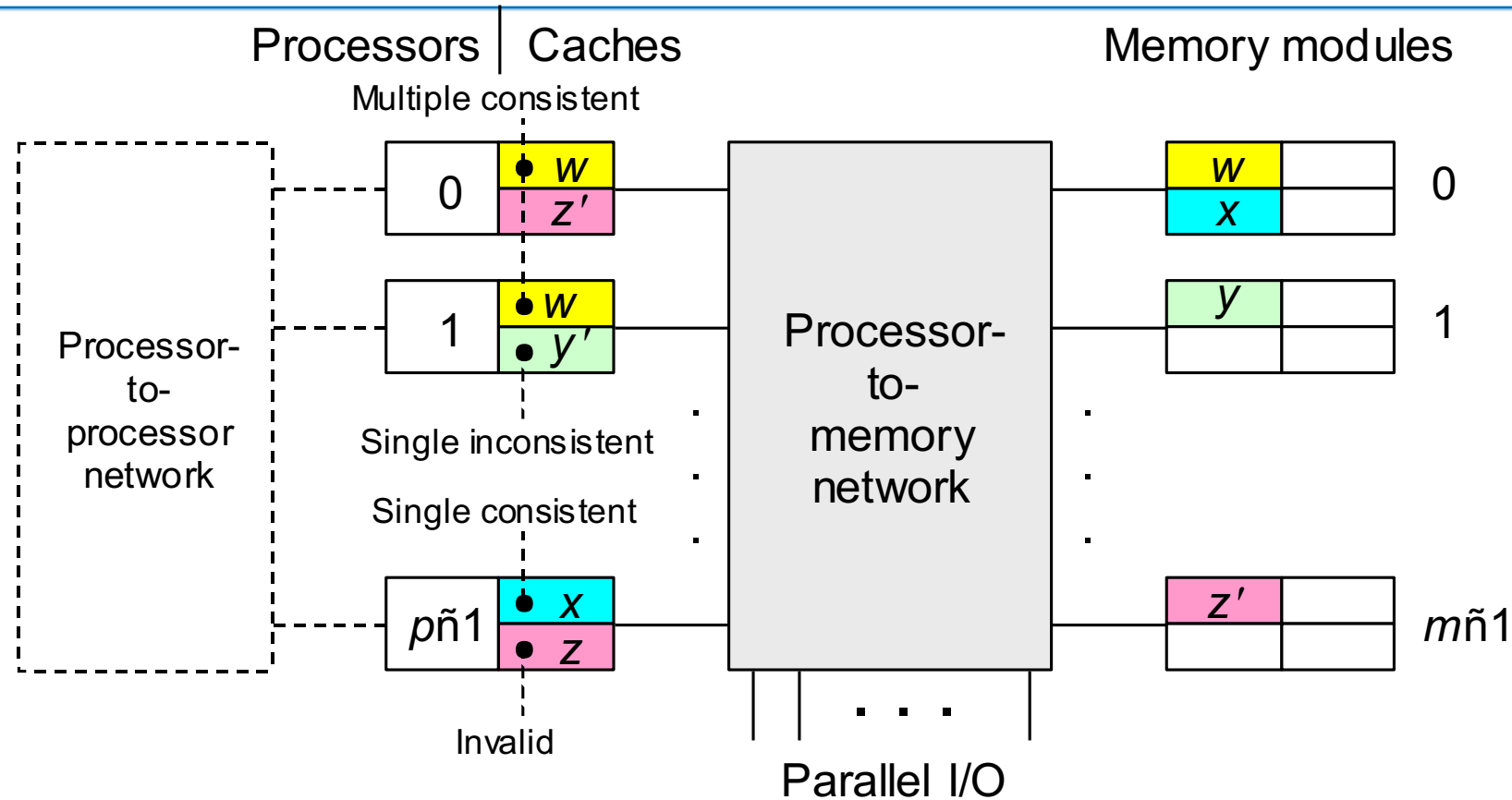
Disc → Memorie: Cache-ul poate să conțină date vechi și să nu vadă write-urile la memorie

Cache-uri multiple și coerența cache-urilor



Memoriile cache dedicate fiecărui procesor reduc traficul la memoria principală (prin rețeaua de interconectare) dar introduc o serie de probleme de consistență.

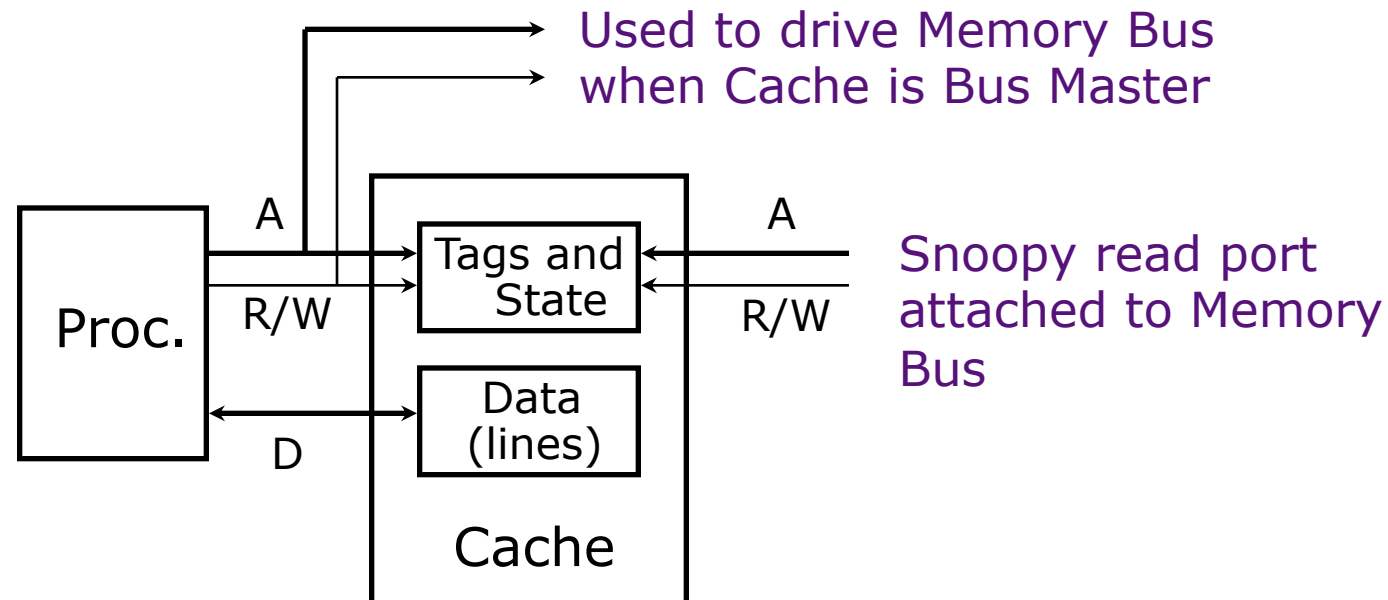
Starea copiilor de date



Diferite tipuri de blocuri de date din cache pentru un procesor paralel cu memorie principală centralizată și cache-uri locale pentru fiecare procesor

Snoopy Cache *Goodman 1983*

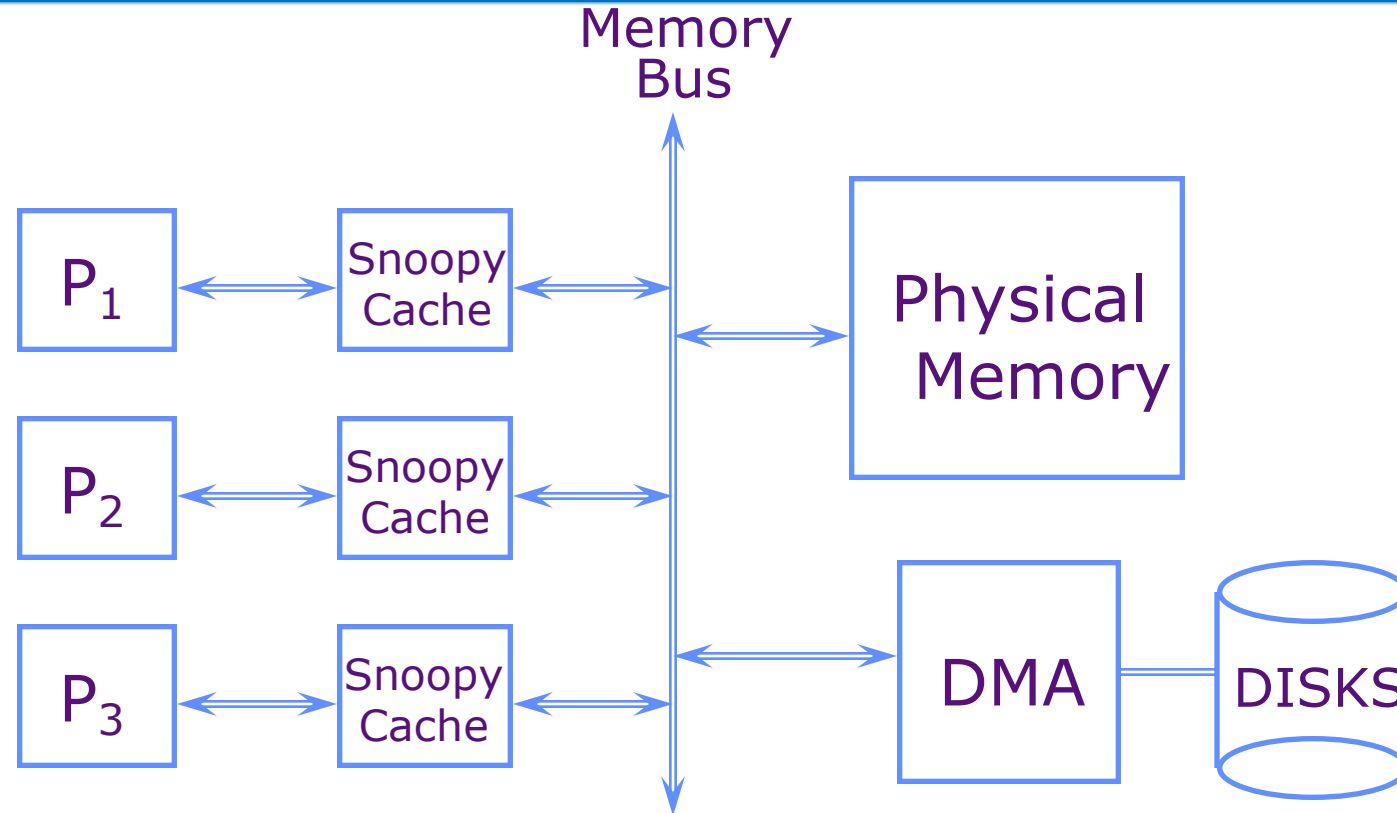
- Idee: Să avem un cache care spionează (snoop upon) transferurile DMA, și atunci “do the right thing”
- Etichetele snoopy cache sunt dual-port



Acțiunile Snoopy Cache pentru DMA

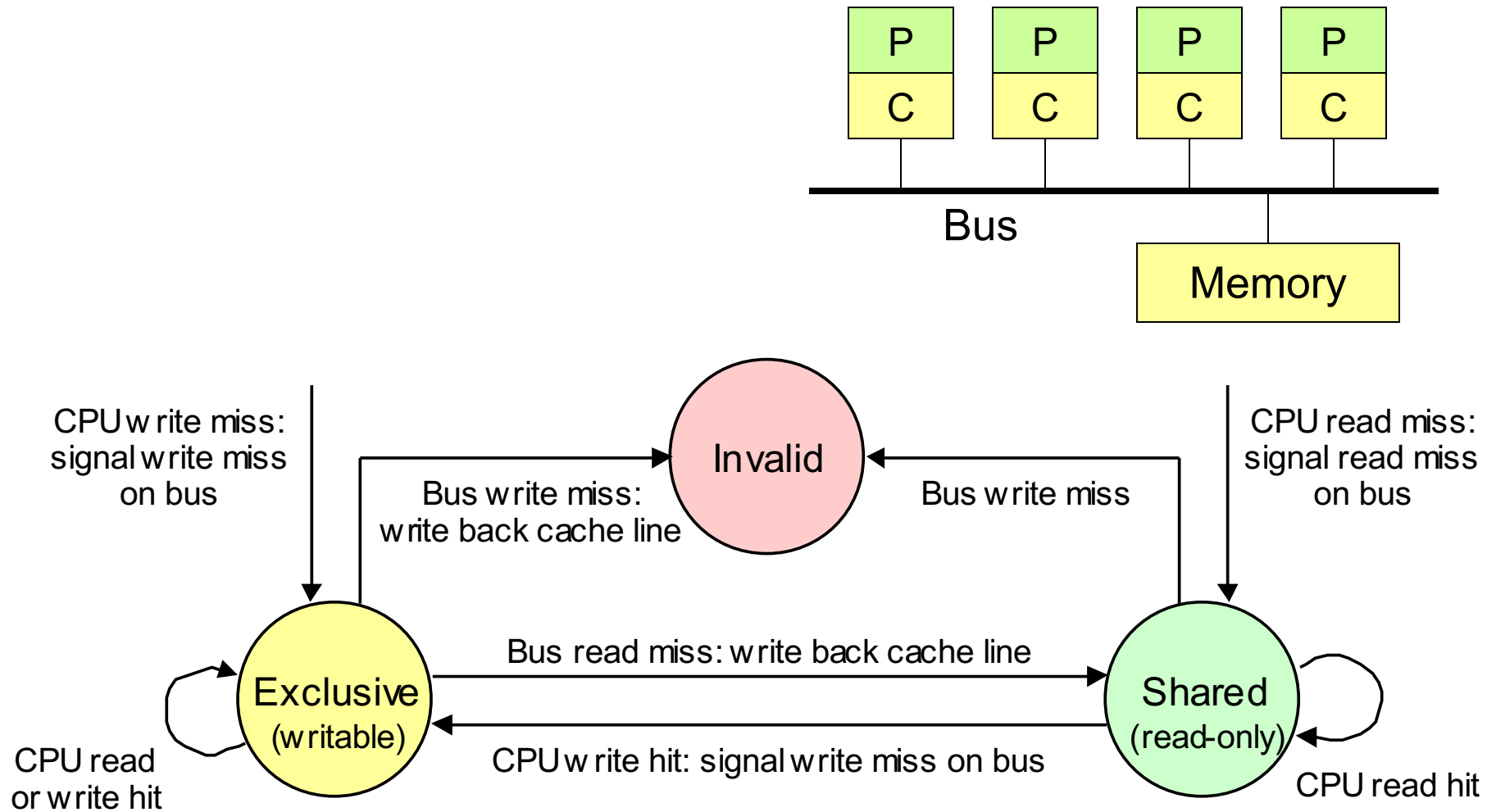
Observed Bus Cycle	Cache State	Cache Action
DMA Read Memory → Disk	Address not cached	No action
	Cached, unmodified	No action
	Cached, modified	Cache intervenes
DMA Write Disk → Memory	Address not cached	No action
	Cached, unmodified	Cache purges its copy
	Cached, modified	???

Shared Memory Multiprocessor



Folosește mecanismul snoopy pentru a păstra consistența memoriei pentru toate procesoarele

Protocol snoopy de coerență cache

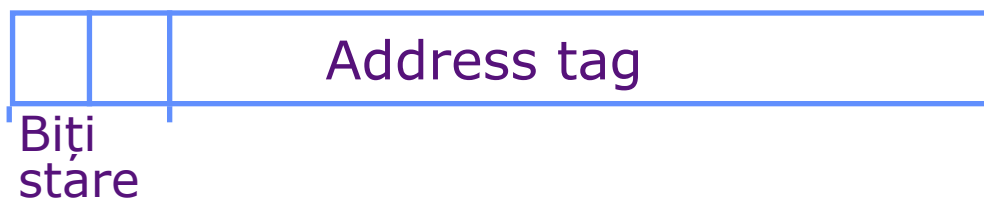


Automat FSM pentru un protocol de coerență cache ce folosește cache write-back

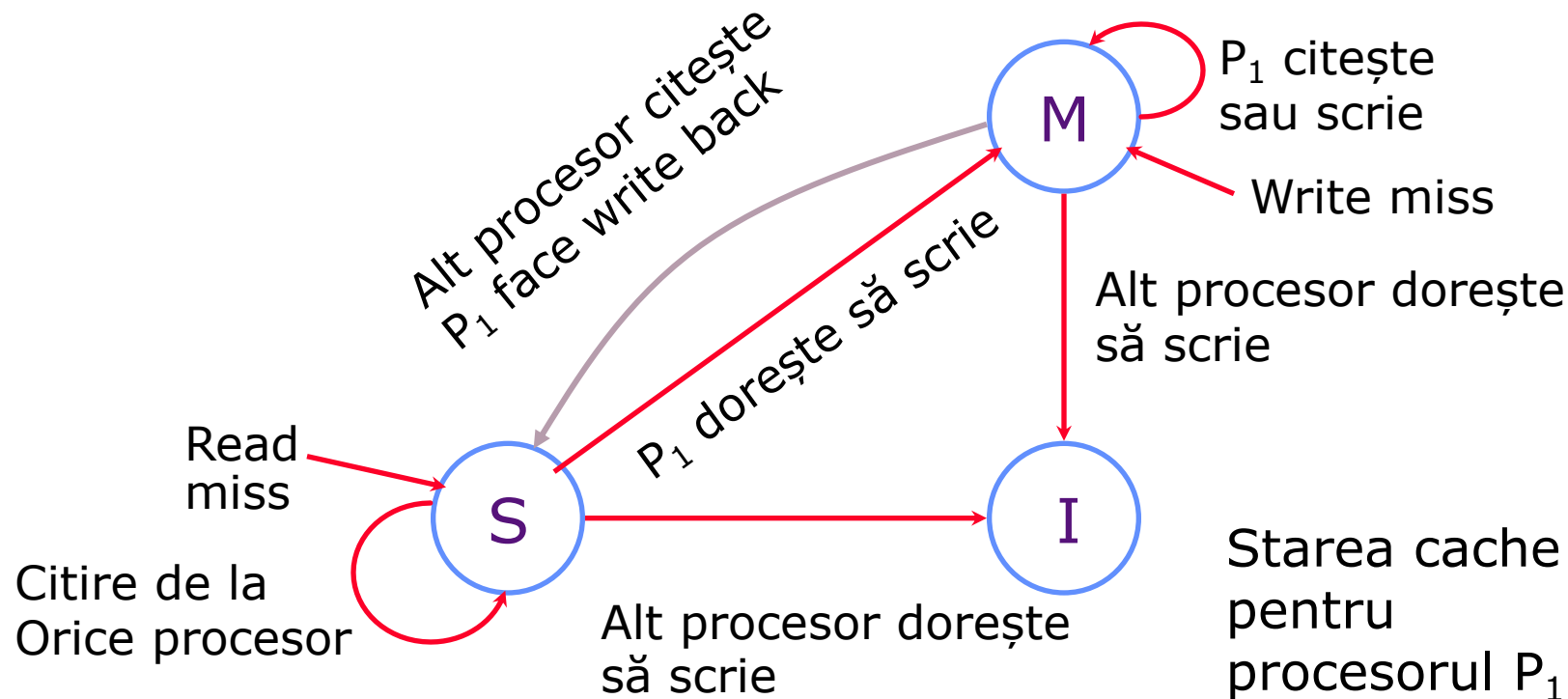
Diagrama de tranziții pentru Cache

Protocolul MSI

Fiecare linie din cache are tag



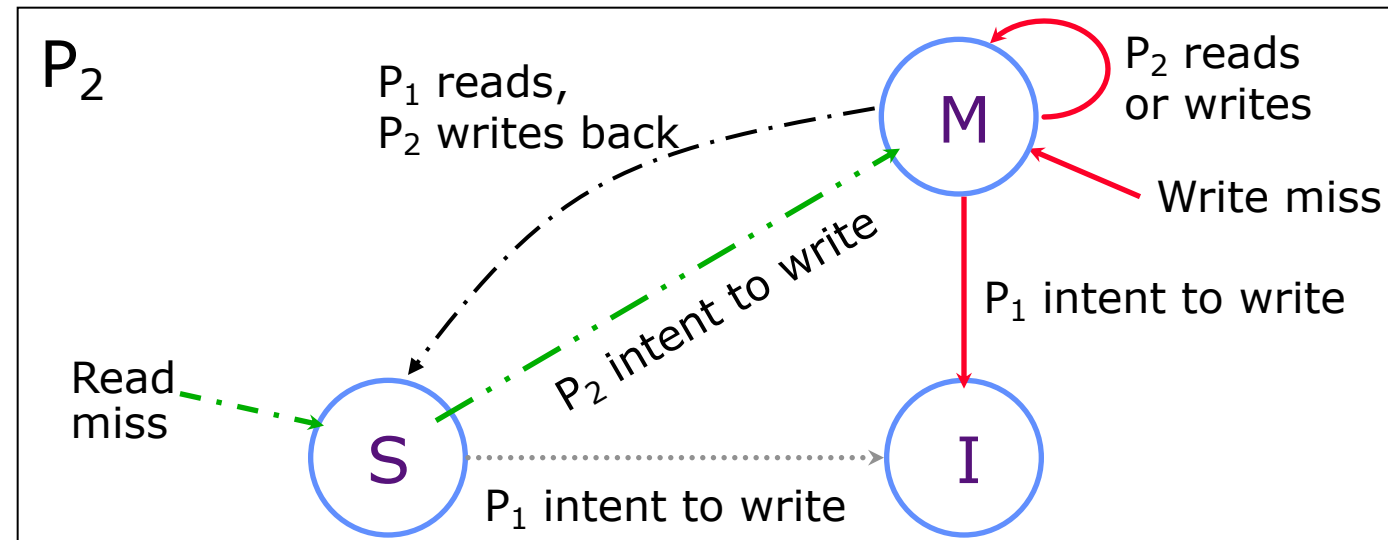
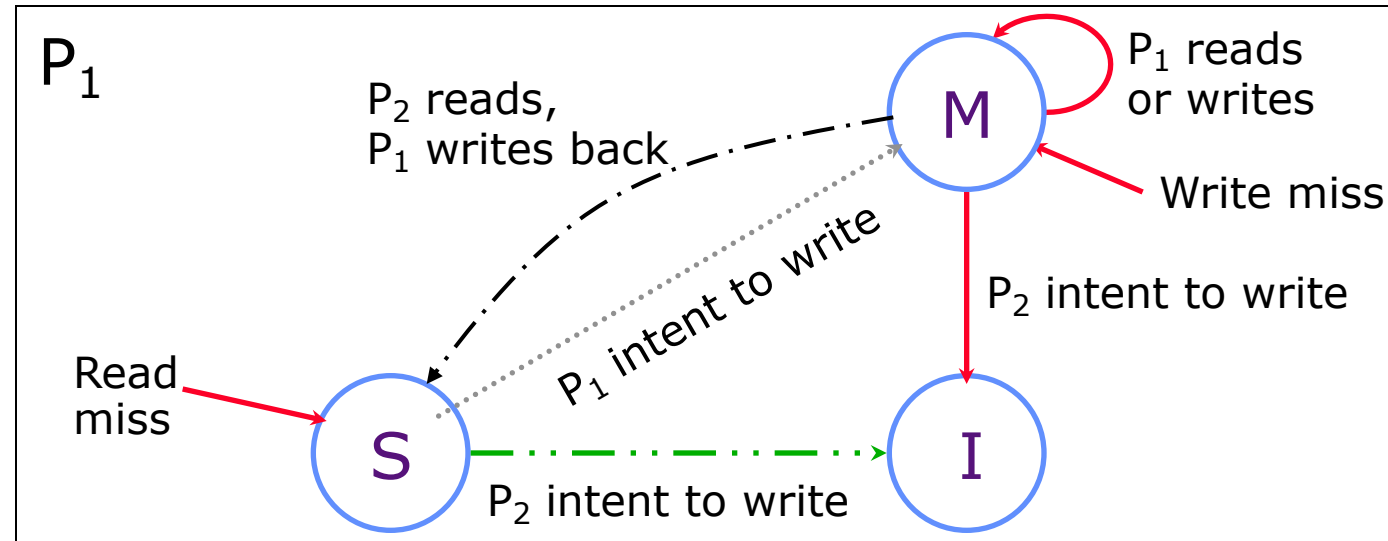
M: Modified
S: Shared
I: Invalid

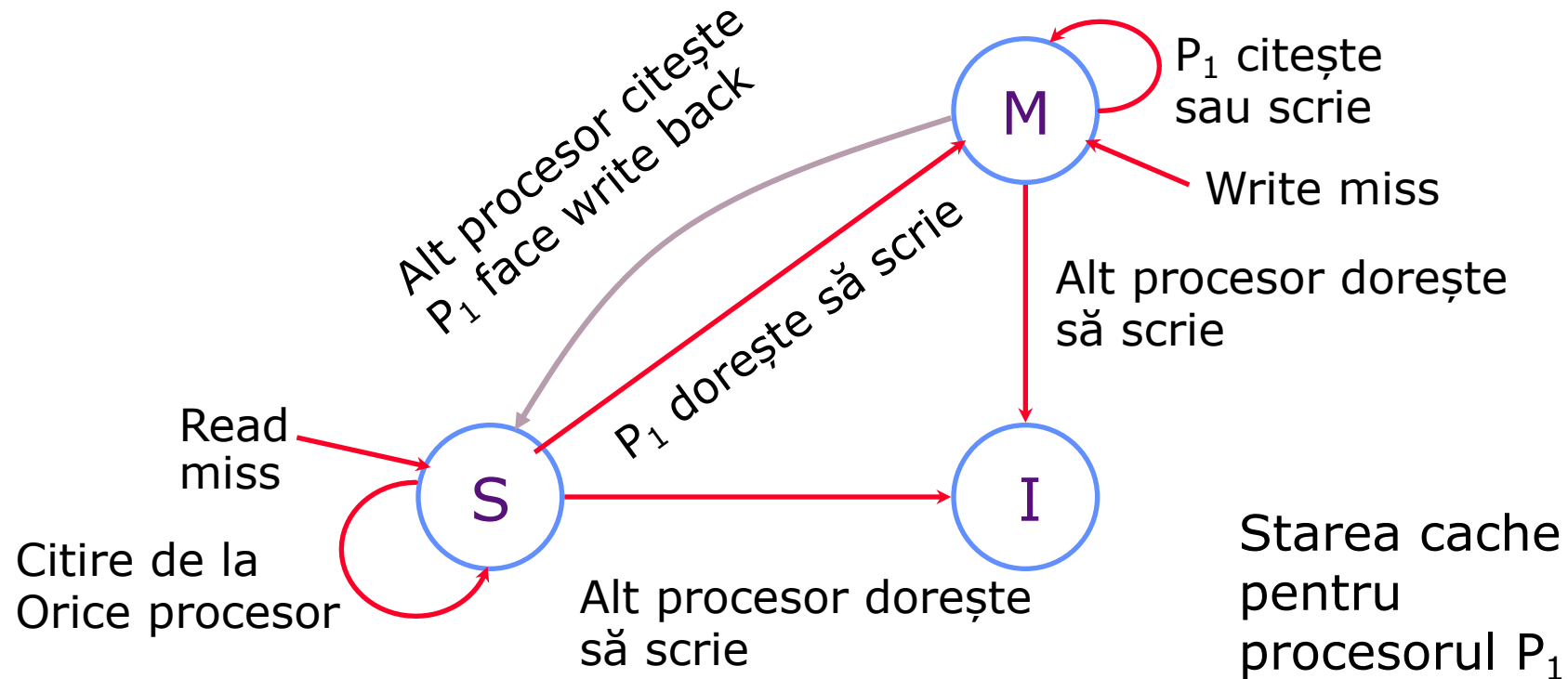


Exemplu cu două procesoare

(Citesc și scriu aceeași linie din cache)

P₁ reads
P₁ writes
P₂ reads
P₂ writes
P₁ reads
P₁ writes
P₂ writes
P₁ writes





- Dacă o linie este în starea **M** atunci nici un alt cache nu poate să aibă o copie a linei!
 - Memoria rămâne coerentă, nu pot exista copii diferite

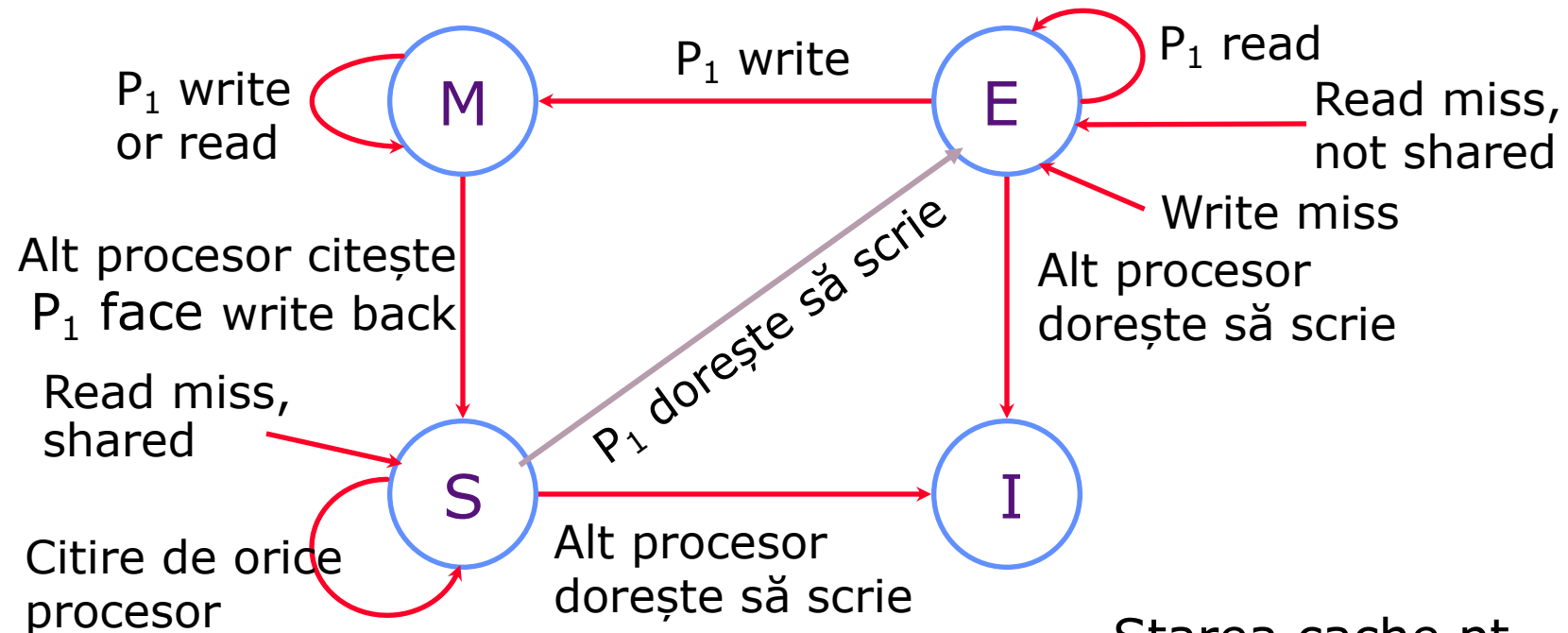
MESI: Protocol MSI îmbunătățit

performanțe mărite pentru date locale

Fiecare linie are un tag

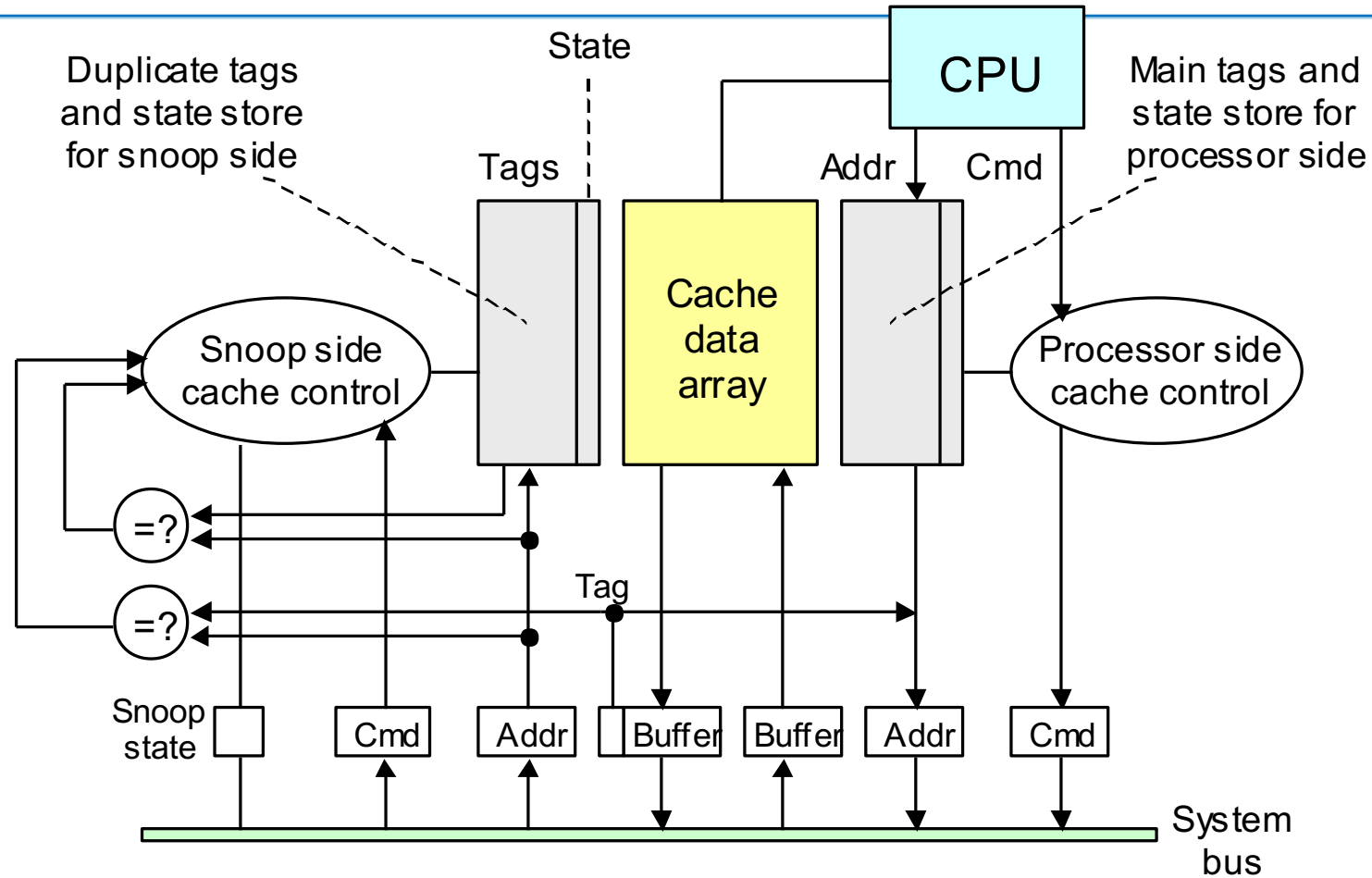


M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid



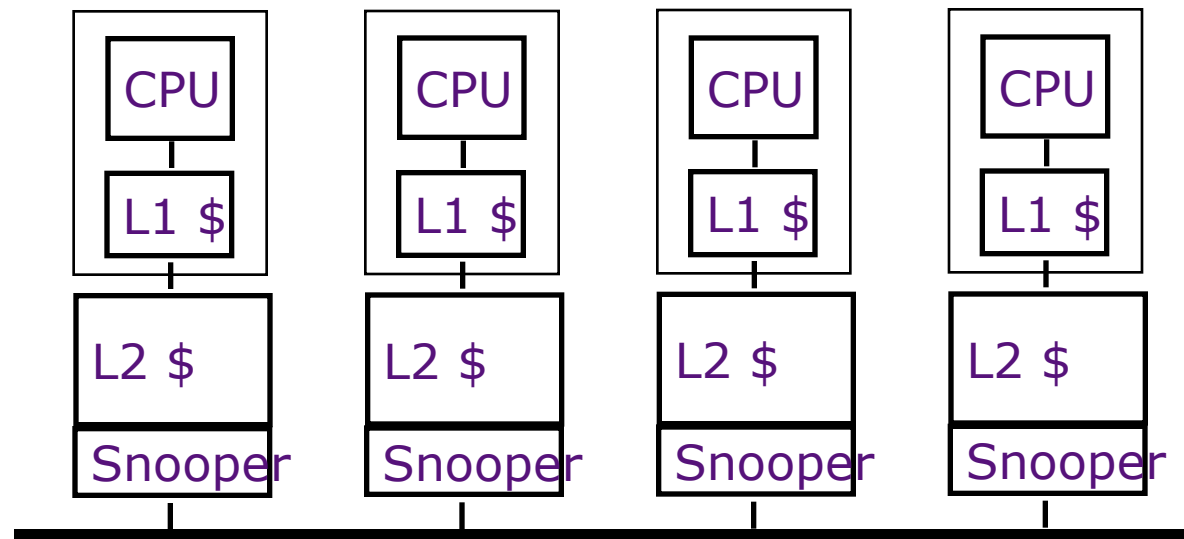
Starea cache pt.
procesorul P1

Implementarea Algoritmului Snoopy Cache

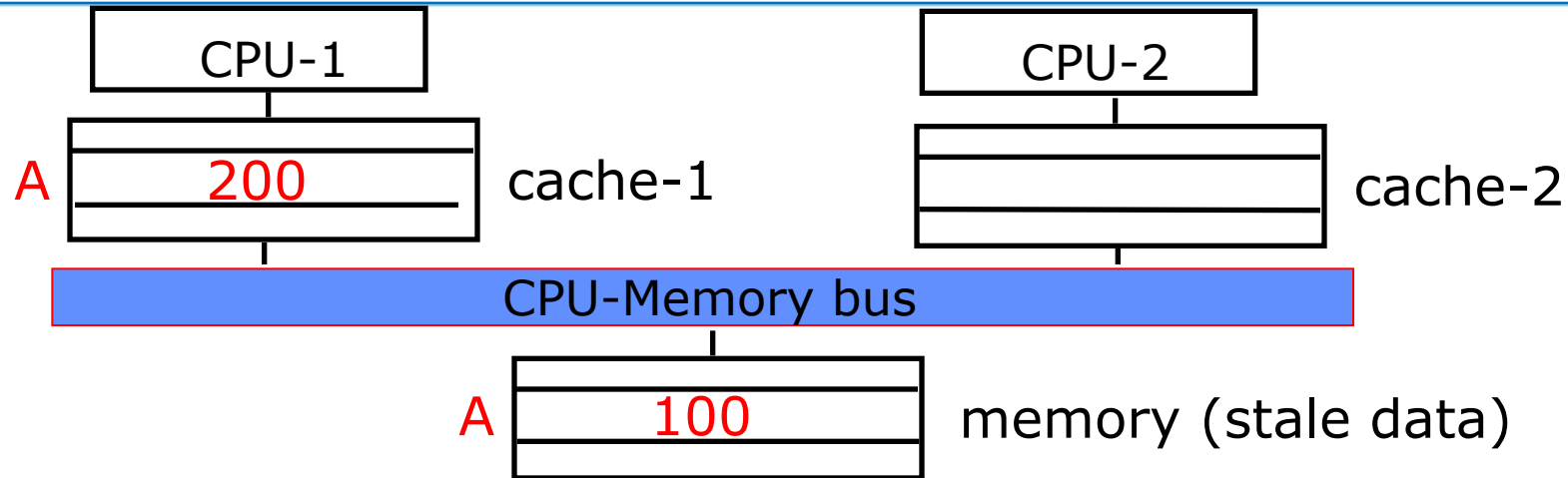


Structura principală a unui algoritm snoopy de coerență cache

Snoop optimizat cu cache-uri Level-2



- Procesoarele au de obicei cache pe două niveluri
 - mic L1, mare L2 (de obicei ambele on chip acum)
- *Proprietatea de incluziune*: intrările din L1 trebuie să fie în L2
invalidare în L2 \Rightarrow invalidare în L1
- Snooping în L2 nu afectează lățimea de bandă CPU-L1
Ce probleme pot să apară?



Când avem un read-miss pentru **A** în cache-2,
se emite pe bus un read request pentru **A**

- Cache-1 trebuie să facă vizibilă și să își schimbe starea în "shared"
- Memoria poate să răspundă și ea la cerere!

Știe memoria că are date vechi?

Cache-1 trebuie să intervină prin controllerul de memorie pentru a da datele corecte pentru cache-2

False Sharing



Un bloc cache conține mai mult de un cuvânt de date

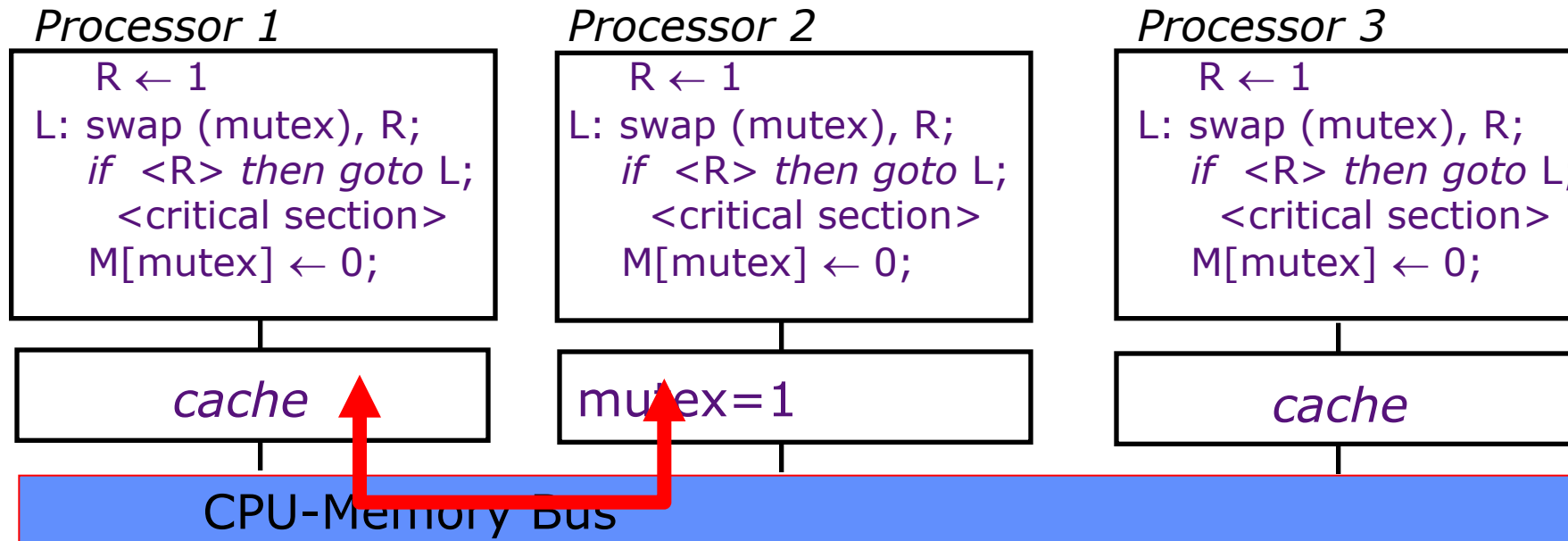
Coerența cache-ului este făcută la nivel e bloc și nu la nivel de cuvânt

Presupunem că P_1 scrie $word_i$ și P_2 scrie $word_k$ și ambele cuvinte au aceeași adresă de bloc.

Ce se poate întâmpla?

Sincronizarea și cache-urile:

Probleme de performanță



Protocoloale de coerență a cache-ului vor face **mutex-ul** să facă *ping-pong* între cache-urile lui P1 și P2.

Acest fenomen poate fi redus prin citirea locației inițiale a **mutex-ului** (non-atomic) și execuția unui schimb doar dacă acesta are valoarea zero.

În general, o instrucțiune *read-modify-write* necesită două operații pe magistrală, dacă nu avem alte operații la memorie de către alte procesoare

Într-un scenariu multiprocesor, accesul tuturor celorlalte procesoare la magistrală trebuie să fie blocat pe durata execuției operației atomice de read-modify-write

⇒ costisitor pentru magistrale simple

ISA-urile moderne folosesc

load-reserve

store-conditional

Registre speciale pentru stocarea flag-urilor de reserve, adresa și rezultatul store-conditional

```
Load-reserve R, (a):  
  <flag, adr> ← <1, a>;  
  R ← M[a];
```

```
Store-conditional (a), R:  
  if <flag, adr> == <1, a>  
  then cancel other procs'  
    reservation on a;  
    M[a] ← <R>;  
    status ← succeed;  
  else status ← fail;
```

Dacă un alt procesor vede o tranzacție de store la adresa din registrul de rezervare, bitul de rezervare este setat la 0

- Mai multe procesoare pot rezerva 'a' simultan
- Aceste instrucțiuni sunt similare cu load și store obișnuite, din pct de vedere al traficului pe bus

Numărul total de tranzacții pe bus nu este neapărat redus, dar spargerea unei instrucțiuni atomice în load-reserve & store-conditional:

- crește utilizarea magistralei, mai ales pentru magistralele care efectuează tranzacții în mai multe etape
- *reduce efectul ping-pong pentru cache* deoarece procesoarele care încearcă să obțină un semafor nu trebuie să facă un store de fiecare dată

Avem un sistem multiprocesor cu memorie partajată construit în jurul unui singur bus cu lățimea de bandă de x GB/s. Cuvintele de instrucțiuni și date au fiecare 4B lățime, fiecare instrucțiune necesită accesul în medie la 1.4 cuvinte din memorie (inclusiv la instrucțiunea însăși). Rata combinată de hit a cache-urilor este de 98%. Calculați limita superioară a performanței sistemului multiprocesor în GIPS. Liniile de adresă sunt separate și nu afectează lățimea de bandă de date.

Soluție

Execuția unei instrucțiuni implică un transfer de $1.4 \times 0.02 \times 4 = 0.112$ B. Astfel, limita absolută a performanței este de $x/0.112 = 8.93x$ GIPS. Dacă presupunem o lățime a busului de 32 de biți, că nici un ciclu pe bus nu se irosește și o frecvență de ceas pe bus de y GHz, limita superioară a performanței devine $286y$ GIPS. Magistralele operează la frecvențe de 0.1 - 1 GHz. Prin urmare, o performanță apropiată de 1 TIPS (chiar și $\frac{1}{4}$ TIPS) este peste capacitățile acestui tip de arhitectură.

- These slides contain material developed and copyright by:
 - Arvind (MIT)
 - Krste Asanovic (MIT/UCB)
 - Joel Emer (Intel/MIT)
 - James Hoe (CMU)
 - John Kubiatowicz (UCB)
 - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252