Arhitecturi Avansate de Calculatoare

(Arhitecturi Paralele)

Arhitecturi Avansate de Calculatoare

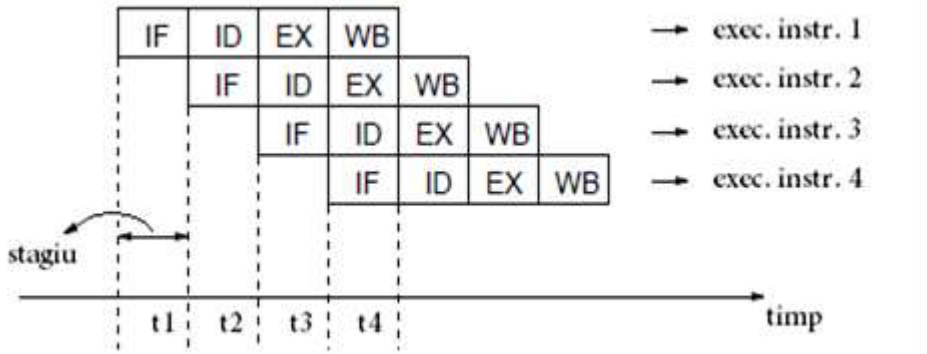
1. Notiuni introductive

- definitii; taxonomia arhitecturilor de calculatoare;
- familii de procesoare;
- metode de evaluare a performantelor; legea lui Amdahl;
- exemple de arhitecturi paralele;
- granularitatea sistemelor paralele

Arhitecturi Avansate de Calculatoare

2. Tehnologii utilizate in arhitecturile de procesoare

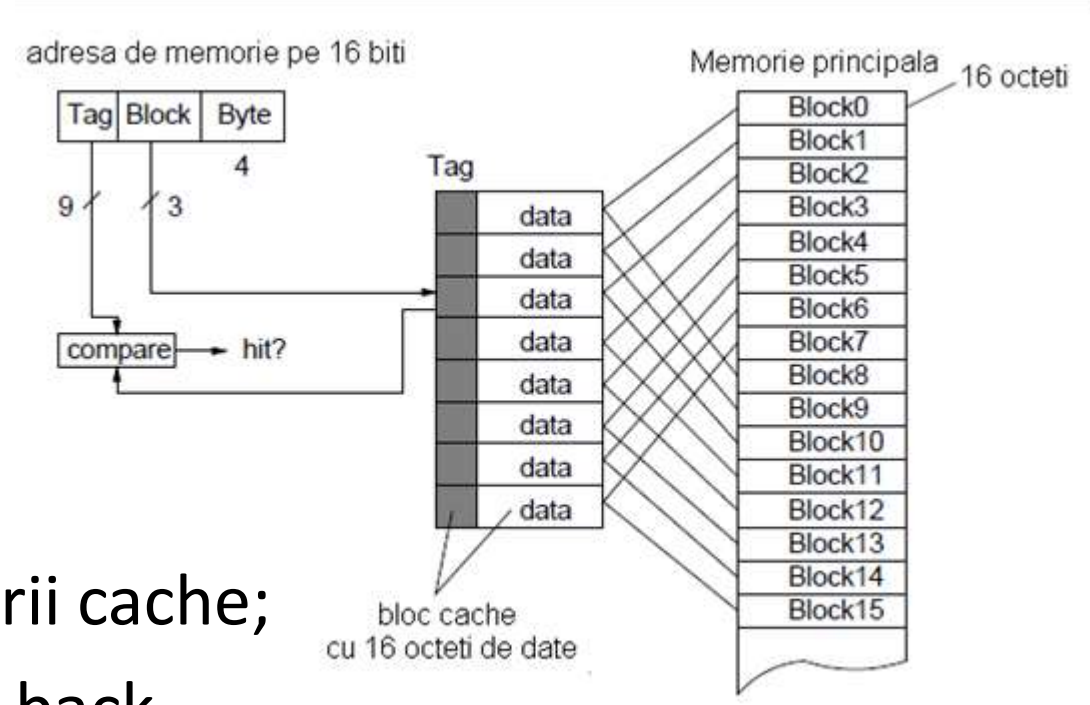
- Procesoare in sisteme paralele:
- transputerul;
- procesoare RISC;
- arhitectura pipeline; procesoare superscalare si superpipeline;
- procesoare VLIW; procesoare vectoriale



Arhitecturi Avansate de Calculatoare

3. Memorii cache

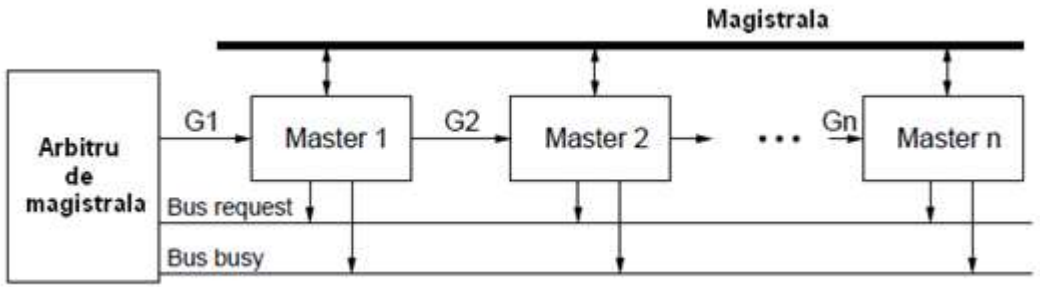
- memorii cache cu mapare directa;
- memorii cache complet asociative;
- memorii cache set-asociative;
- strategii de implementare pentru memorii cache;
- memorii write-through si memorii write-back



Arhitecturi Avansate de Calculatoare

4. Retele de comunicatie pentru sisteme paralele (I)

- comutarea si rutarea;
- clasificarea retelelor în functie de topologie
- retele directe; retele indirecte;
- retele bazate pe magistrala; magistrale sincrone si asincrone; magistrale cu tranzactii multiple



Arhitecturi Avansate de Calculatoare

5. Retele de comunicatie pentru sisteme paralele (II)

- retele multistagiu; retele blocante si neblocante;
- retelele Omega;
- retele crossbar;
- comparatie între retelele directe si cele indirecte;
- comutarea prin pachete: metodele *Store & Forward, Wormhole Routing* si *Virtual Cut-through*

Arhitecturi Avansate de Calculatoare

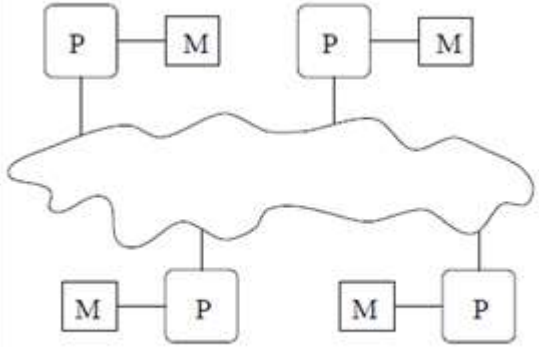
6. Tehnici de rutare

- rutare stabilita la nivelul nodului sursa;
- rutarea locala;
- rutarea adaptiva si rutarea determinista;
- eliminarea blocajelor de rutare; folosirea canalelor virtuale;
- metode pentru implementarea comunicatiei multicast

Arhitecturi Avansate de Calculatoare

7. Multicalculatoare

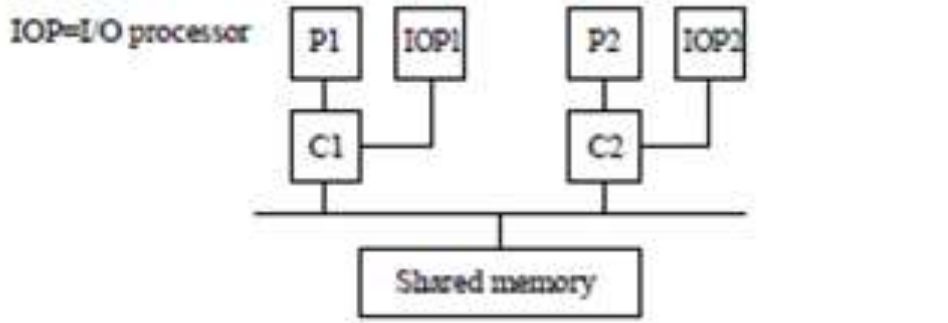
- implementarile VSM si SVM;
- abordari si metode folosite in proiectarea sistemelor multicalculator;
- metoda cu server central; metoda migratiei; metoda replicarii datelor la citire; metoda *Full Replication*;
- exemple de multicalculatoare: IBM SP2 si Parsytec CC



Arhitecturi Avansate de Calculatoare

8. Sisteme multiprocesor (I)

- coerenta memoriilor cache in sisteme cu memorie partajata;
- surse de inconsistenta a datelor;
- protocoalele write-invalidate si write-update;
- arhitecturile UMA, NUMA, CC-NUMA



Arhitecturi Avansate de Calculatoare

9. Sisteme multiprocesor (II)

- ascunderea latentei la accesul memoriei de la distanta;
- modele de consistenta a memoriei;
- arhitectura COMA; comparatie între arhitectura COMA si arhitectura CC-NUMA;
- arhitecturile S-COMA, R-NUMA

Arhitecturi Avansate de Calculatoare

10. Sisteme multiprocesor (III)

Coerenta memoriilor cache in sisteme multiprocesor:

- protocoalele *Snoopy-bus* si bazate pe director;
- metode de implementare pentru memoria director: implementarile *Full Map* si *Limited Map*; metoda cu director înlantuit;
- diagramele de stari si tranzitii pentru memoriile *write-through* si *write-back*

Arhitecturi Avansate de Calculatoare

11. Sisteme multiprocesor (IV)

Sisteme de operare pentru multiprocesoare:

- exploatarea concurentei, detectarea paralelismului in programe, mecanisme de sincronizare, exemple

12. Standarde si medii de programare pentru arhitecturi paralele:

- standardul MPI, mediul PVM, limbajul OCCAM

Referinte

- Fundamentals of Parallel Multicore Architecture, Yan Solihin, Chapman and Hall/CRC June 30, 2020, ISBN-13: 978-0367575281
- O. Buza, Arhitecturi Paralele de Calculatoare, Ed. Grinta, Cluj-Napoca, 2018, ISBN 978-606-037-012-3
- J. L. Hennessy, D. A. Patterson, Computer Architecture, 6th Edition (The Morgan Kaufmann Series in Computer Architecture and Design), Elsevier, 2017, ISBN 978-0128119051
- J. L. Hennessy, D. A. Patterson, Computer Organization and Design RISC-V Edition: The Hardware Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) 1st Edition, Elsevier, 2017, ISBN-13: 978-0128122754
- G. Lerman, L. Rudolph, Parallel Evolution of Parallel Processors (Evaluation in Education and Human Services), Springer, 2013, ISBN-13: 978-1461362371
- Michel Dubois, Parallel Computer Organization and Design, 1st Edition, Cambridge University Press August 1, 2012, ISBN-13 : 978-0521886758

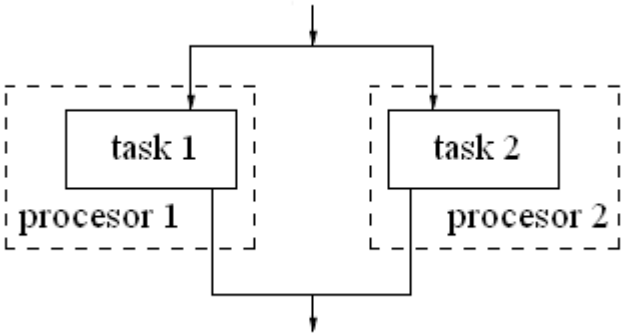
Online references

- Computer Architecture, 2020, https://www.sciencedirect.com/topics/computer-science/computer-architecture
- Computer Organization and Architecture, 2020, https://www.geeksforgeeks.org/computer-organization-and-architecture-tutorials

Curs 2 AAC

ARHITECTURI PARALELE DE CALCULATOARE
NOŢIUNI INTRODUCTIVE

Prelucrarea paralelă reprezintă utilizarea mai multor procesoare pentru execuţia simultană a mai multor părţi dintr-un program (task-uri).



Avantaje

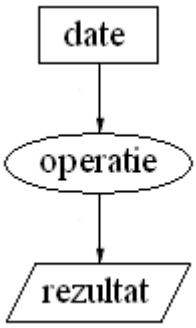
- Se elimină limitarea de viteză a procesoarelor secvenţiale (timp de execuţie mai scurt).
- Se elimină limita de miniaturizare a procesoarelor secvenţiale (nu putem construi un procesor complex la dimensiuni oricât de mici).
- Se elimină limitarea economică (imposibilitatea de a fabrica ieftin un procesor foarte rapid => e mai bine să se folosească două sau mai multe procesoare ieftine care să ruleze în paralel pe acelaşi calculator).

Evoluţia

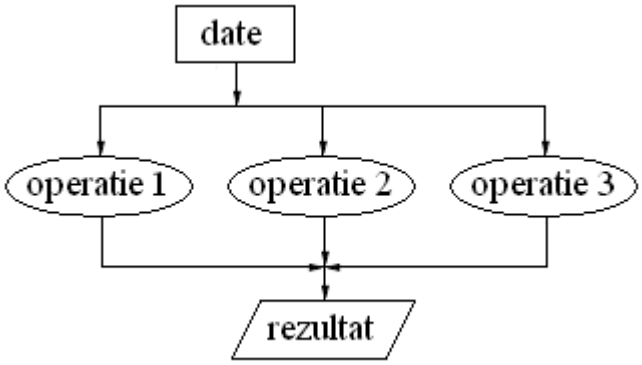
- Procesoare secvenţiale obişnuite, numite procesoare scalare: execută maxim o instrucţiune într-un ciclu procesor.
- Procesoare superscalare: pot executa mai multe instrucţiuni într-un ciclu procesor.
- Procesoare vectoriale: nu mai lucrează pe operanzi scalari ci pe structuri de date multiple: vectori uni / multi dimensionali.
- Arhitecturi multiprocesor: într-un singur cip există mai multe nuclee de procesare (dual core / quad core /octa core).

Clasificarea arhitecturilor de calculatoare

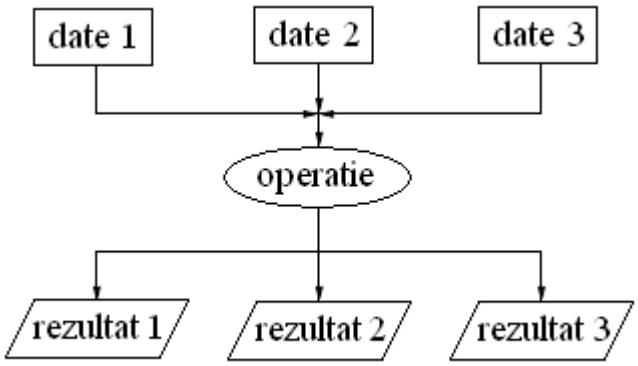
- **SISD** (*Single Instruction Single Data*): arhitectura secvenţială obişnuită.



- **MISD** (*Multiple Instruction Single Data*): un program e descompus în mai multe task-uri sau secvenţe de operaţii; operaţiile se execută în paralel asupra aceluiaşi set de date.

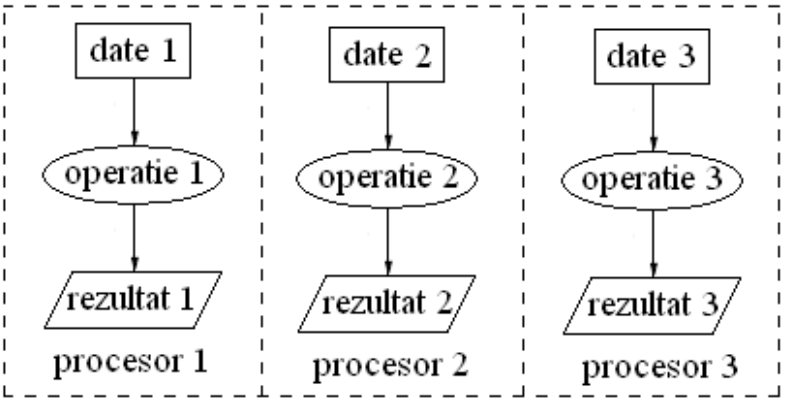


- **SIMD** (*Single Instruction Multiple Data*): un acelaşi program se execută pe mai multe seturi de date.



ex: prelucrari de masive (siruri, matrici); procesoare vectoriale

- **MIMD** (*Multiple Instruction Multiple Data*): reprezintă tipul general de arhitectură pentru calculatoarele paralele; se realizează operaţii diferite asupra unor seturi de date diferite.



Viteza de execuţie a programelor paralele

Acceleraţia: expresie a vitezei de execuţie: $Sp = \frac{Tes}{Tep}$ unde:

Tes – timpul de execuţie secvenţial;
Tep – timpul de execuţie paralel pe mai multe procesoare al aceluiaşi program.

Ideal: $Sp = \frac{Tes}{Tes/P} = P$ (numărul de procesoare)

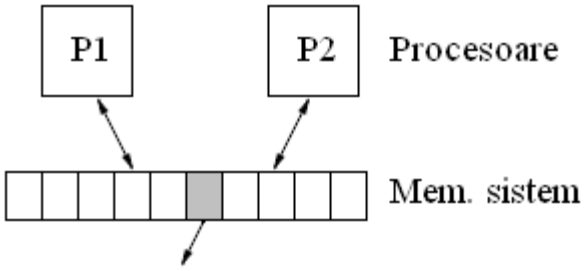
Real: $Tep \neq Tes/P$; $Tep = T\,\text{secv} + Tparalel$ (o porţiune din program se poate executa doar secvenţial în timpul Tsecv şi avem o porţiune care se poate executa în paralel)

$$\Rightarrow Sp = \frac{Tes}{T\,\text{secv} + Tparalel} \Rightarrow Sp < \frac{Tes}{T\,\text{secv}} = K$$ **- Legea lui Amdahl**

Oricât am creşte numărul de procesoare într-un sistem, acceleratia (viteza de execuţie în paralel) a unui program e întotdeauna limitată superior.

Accesul la memorie în procesarea paralelă

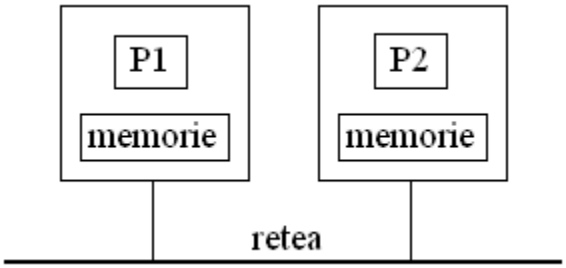
a) Acces partajat la memorie.
In acest caz, procesoarele împart acelaşi spaţiu de memorie.



La un anumit moment, o anumită locaţie din memoria partajată poate fi accesată doar de către un singur procesor.

Avantaj: realizarea unui schimb rapid de date între 2 task-uri care rulează pe 2 procesoare diferite.
Exemplu: arhitectura UMA – procesoarele accesează în mod identic orice locaţie din memoria partajată a sistemului; arhitectura Pentium multicore; arhitectura GPU.

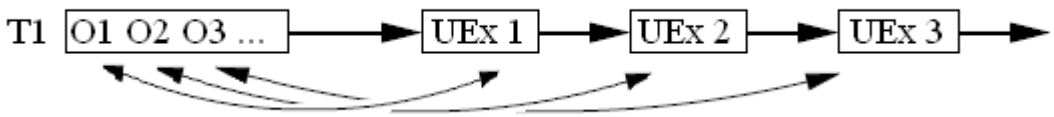
b) Acces distribuit la memorie.



Procesoarele pot avea acces atât la o memorie locală aflată pe calculatorul din care fac parte, cât şi la o memorie aflată la distanţă (alt calculator).
Exemplu: arhitectura NUMA – procesoarele au acces rapid la memoria locală şi acces lent la memoria aflată la distanţă; arhitectura COMA – în care memoriile locale ale procesoarelor sunt organizate ca nişte memorii cache.

Exemple de arhitecturi paralele

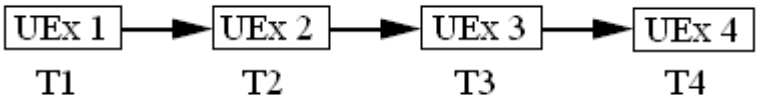
1) **Arhitectura Pipeline** – structură cu un singur procesor, dar care poate executa mai multe operaţii în paralel; astfel un anumit task se descompune într-o serie de operaţii care se execută independent de către unităţi de execuţie specializate.



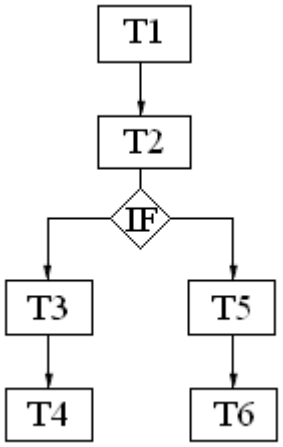
Există 2 tipuri de Pipeline: - de instrucţiuni; în care o instrucţiune se descompune într-o serie de microinstrucţiuni.
- aritmetic; în care o operaţie aritmetică se descompune într-o serie de microoperaţii.
În rezolvarea unui task pot exista anumite situaţii speciale, în care fluxul de operaţii din Pipeline se opreşte. Aceste situaţii speciale se numesc hazarduri (incidente) şi sunt de 3 tipuri:

a) Hazardul structural. El e generat de faptul că nu există destule unităţi de execuţie pentru operaţiile planificate.
b) Hazardul de date. El e generat din cauza dependenţelor de date. Un anumit task aşteaptă după datele unui alt task.
c) Hazardul de control. El e datorat ramificărilor care apar în program (instrucţiunile de salt).

De exemplu, dacă în structura Pipeline se încarcă la început patru operaţii:



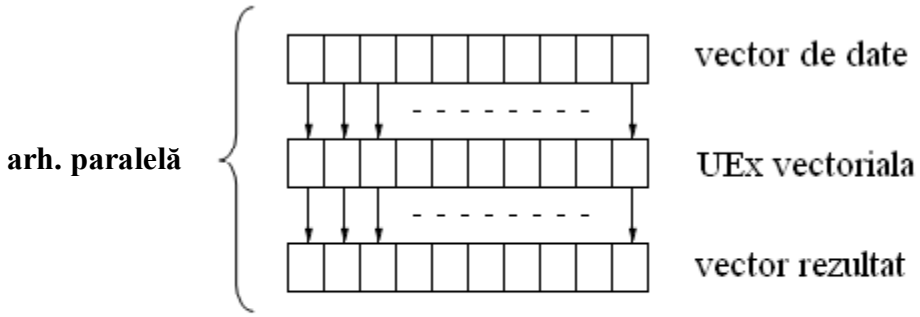
în cazul schimbării fluxului de instrucţiuni din program, instrucţiunile încărcate în avans în structura Pipeline (T3 şi T4) trebuie eliminate şi înlocuite cu instrucţiunile corecte (T5 şi T6).

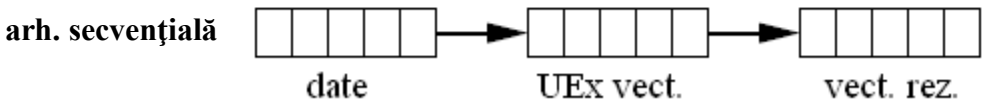


2) **Arhitecturi cu legături multiple** – acestea sunt structuri multiprocesor conectate în diverse configuraţii în care fiecare procesor realizează anumite operaţii simple într-un timp foarte scurt.

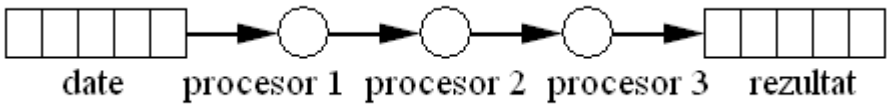
a) **Arhitecturi vectoriale**.
Exemple:

- **Procesoare vectoriale**



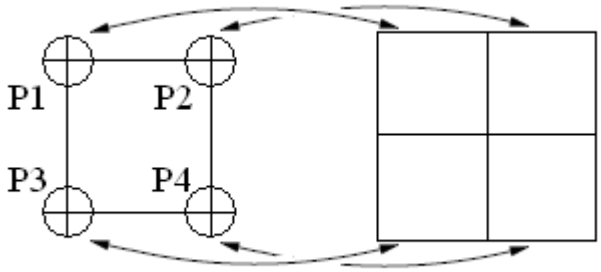


- **Procesoare sistolice**: la fiecare tact datele avansează o poziţie prin structura sistolică.



Există un şir de elemente de prelucrare simple, conectate în diferite topologii; elementele de prelucrare pot avea o memorie locală şi pot fi de tipul SIMD sau nu (fiecare element poate efectua aceeaşi sau o altă operaţie).

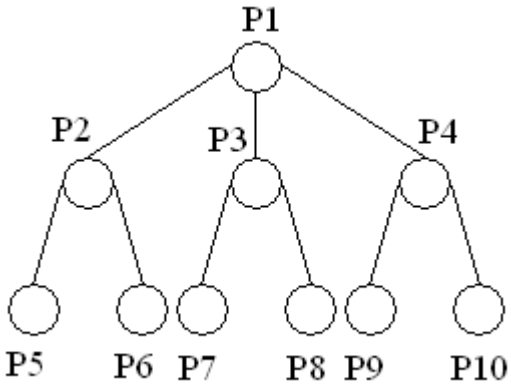
b) **Arhitecturi de tip reţea**.
Exemplu: - procesoare matriciale



Unităţile de prelucrare sunt autonome şi au o memorie locală ataşată.

De obicei există o unitate de control care distribuie sarcinile la structura matricială de procesare.

c) **Arhitectura de tip arbore**. Se foloseşte la calcule ştiinţifice / economice, programe de decizie.



d) **Arhitectura hipercub**. E o arhitectură mixtă între reţea şi arbore. Exemplu: fiecare nod al reţelei poate fi un arbore.

Granularitatea sistemelor multiprocesor

Granularitatea este rezoluţia (gradul de fineţe) cu care e rezolvată o anumită sarcină. În cazul de faţă, sarcina e execuţia paralelă a unui program $\Rightarrow$ granularitatea indică gradul de paralelizare pentru acel program.
Există 3 tipuri:

- Granularitate mică (grosieră) – realizarea paralelismului se face la nivel de proces.
- Granularitate medie – paralelismul se realizează la nivel de *thread* (fir de execuţie din interiorul unui proces alcătuit din mai multe instrucţiuni care utilizează aceleaşi resurse).
- Granularitate mare (fină) – paralelismul se realizează la nivel de instrucţiune.

PROCESOARE ÎN SISTEME PARALELE

În trecut, calculatoarele paralele conţineau procesoare proiectate special pentru a deservi sarcina paralelizării. Astăzi calculatoarele paralele utilizează procesoare obişnuite de uz general, soluţie mult mai convenabilă din punct de vedere economic dar şi tehnologic computaţional.

Un exemplu de procesor special dezvoltat pentru calcul paralel este **transputerul**, care apărut in anii 1980. O unitate de calcul bazată pe transputer conţine (Fig. 1):

- un procesor capabil să execute operaţii aritmetice de bază precum şi operaţii de intrare/ieşire;
- o memorie locală de tip SRAM de dimensiuni mici dar foarte rapidă;
- 4 canale de comunicaţie cu exteriorul;
- o interfaţa cu memoria externă.

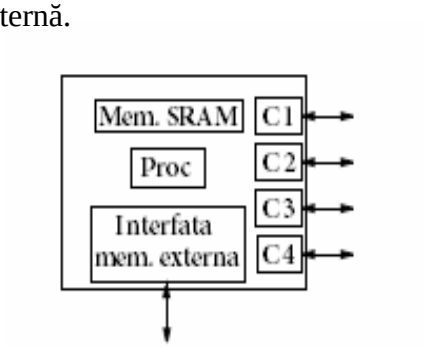


Figura 1. Structura de bază a unui transputer

Pe cele 4 canale de comunicaţie se pot lega alte transputere, formându-se o reţea multiprocesor cu topologie (configuraţie) diversă. Un exemplu de celulă cu 5 transputere este ilustrat în figura 2:

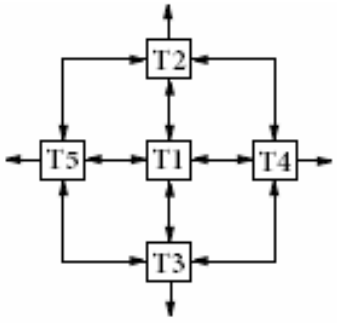


Figura 2. O celulă formată din 5 transputere

Dezavantajul folosirii transputerelor era dat de o putere computaţionala redusă, insuficientă pentru aplicaţiile tot mai complexe care au fost dezvoltate.

În anii 1990 au apărut procesoarele **RISC** (*Reduced Instruction Set Computer* – Calculatoare cu set redus de instrucţiuni), spre deosebire de procesoarele **CISC** (*Complex Instruction Set Computer*) care existau până atunci. Aceste procesoare RISC dădeau puterea computaţională de care era nevoie în aplicaţie şi ele au devenit din ce în ce mai ieftine, fiind produse pe scară largă.

Câteva exemple de procesoare RISC: procesorul MIPS de la SGI, procesorul POWER (IBM), SPARC (SUN), procesorul ALPHA (Cray).

Caracteristici ale procesoarelor RISC:

- au puţine moduri de adresare;
- au un format fix al instrucţiunilor (32b sau 64b);
- au un număr mare de registre care ajută la comutarea rapidă a contextului între programe;
- folosesc instrucţiuni de tip LOAD / STORE, instrucţiuni dedicate pentru încărcarea / salvarea datelor din memoria calculatorului în registrele procesorului (Fig.3):

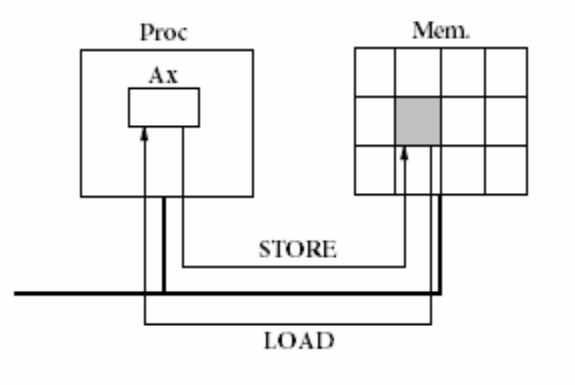


Figura 3. Instrucţiunile LOAD / STORE

- utilizează memoria cache pentru încărcarea datelor în procesor. Memoria cache aduce în avans instrucţiunile procesorului sau datele din memoria sistem DRAM, conferind avantajul unei viteze mărite de acces faţă de accesarea directă a datelor din memoria externă. (Fig.4):

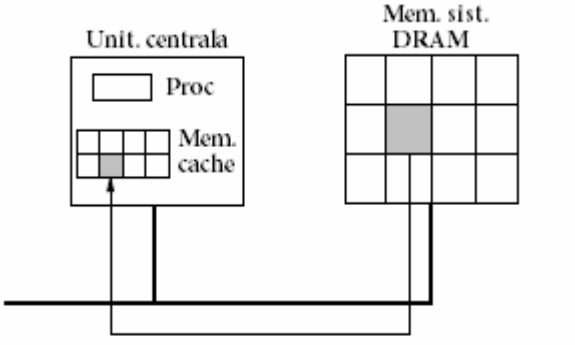


Figura 4. Memoria cache

Procesoarele RISC utilizează **arhitectura *pipeline*** (bandă de asamblare), în care o instrucţiune sau un task se descompune în mai multe operaţii care sunt executate independent de către unităţi de execuţie specializate (Fig. 5):

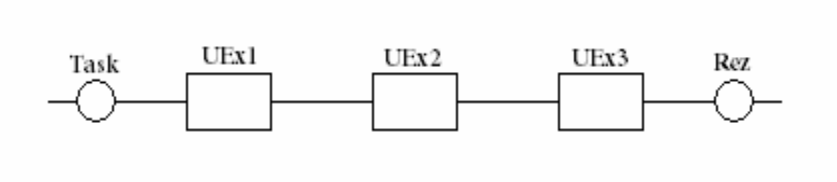


Figura 5. Arhitectura *pipeline*

În cazul procesoarelor RISC, arhitectura *pipeline* defineşte mai multe stagii (etape) de execuţie ale unei instrucţiuni. De exemplu, putem avea următoarele 4 stagii (Fig. 6):

Stagiul 1 – IF (*instruction fetch*): încărcarea instrucţiunii din memorie
Stagiul 2 – ID (*instruction decoding*): decodificarea instrucţiunii
Stagiul 3 – EX (*execute*): execuţia propriu-zisă a instrucţiunii
Stagiul 4 – WB (*write back*): scrierea rezultatului în registru sau memorie

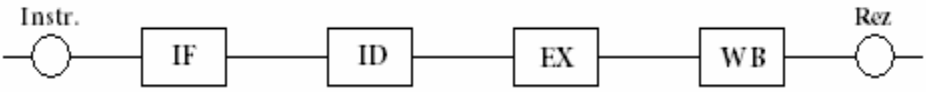


Figura 6. Stagiile de execuţie ale unei instrucţiuni

Diagrama de funcţionare în timp a acestei structuri este ilustrată în figura 7:

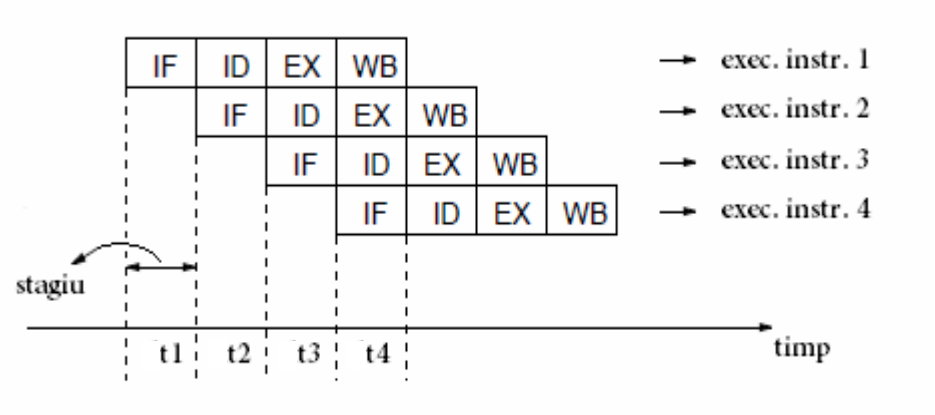


Figura 7. Diagrama execuţiei în timp pentru structura *pipeline*

Din această diagramă se poate observa că în orice moment de timp (de exemplu la momentul t4) se execută 4 stagii în paralel pentru 4 instrucţiuni diferite, rezultând astfel un paralelism temporal. La fiecare ciclu temporal, structura *pipeline* obţine un rezultat, astfel că putem executa câte o instrucţiune la fiecare ciclu procesor (şi nu în 4 cicli cum ar fi fost fără structura *pipeline*). Acest exemplu se aplică pentru procesoarele scalare. În acest caz există o singură linie *pipeline* şi se poate executa maxim o instrucţiune pe ciclu.

În prezent, majoritatea procesoarelor au o structură superscalară. La această structură există mai multe linii *pipeline* care operează în paralel. Ca rezultat, se pot executa mai multe instrucţiuni pe un ciclu procesor.

În figura 8 se prezintă o structură superscalară cu 3 linii *pipeline*:

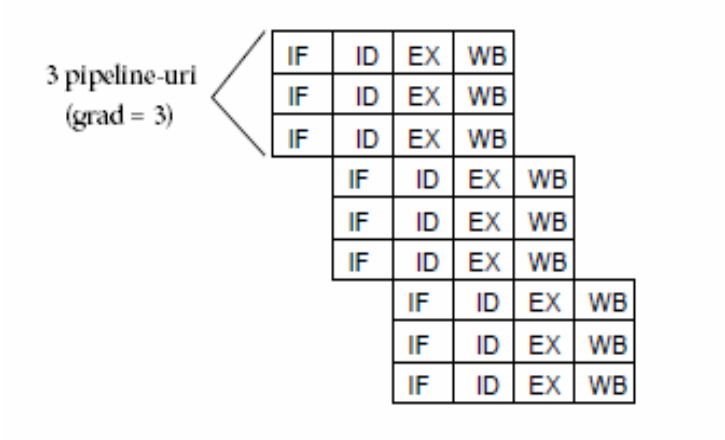


Figura 8. Diagrama de timp pentru structura superscalară

Câteva caracteristici ale arhitecturii superscalare:
- arhitectura permite executarea în paralel a mai multor instrucţiuni;
- se pot executa mai multe instrucţiuni pe un ciclu procesor;
- permite execuţia *out of order* (în afara ordinii) a instrucţiunilor: instrucţiuni independente pot fi executate în afara ordinii normale din program; condiţia pentru execuţia *out of order* este să nu existe dependenţe de date între instrucţiuni.

O altă metodă pentru creşterea eficienţei *pipeline* este folosirea arhitecturii *superpipeline*. În acest caz se reduce numărul de operaţii care se efectuează într-un stagiu, dar se măreşte numărul de stagii prin împarţirea fiecărui stagiu în mai multe substagii. Astfel putem creşte frecvenţa de tact a procesorului, pentru că se execută mai puţine operaţii într-un ciclu procesor decât la un stagiu obişnuit.

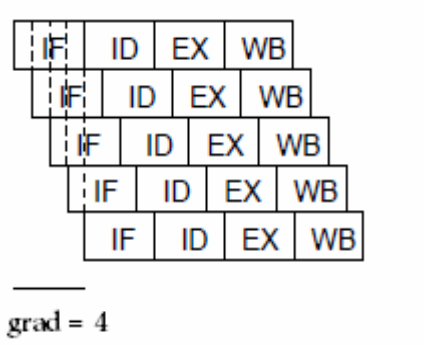


Figura 9. Diagrama de timp pentru structura *superpipeline*

În exemplul de mai sus, datorită faptului că un stagiu a fost împărţit în 4 substagii, vom putea creşte frecvenţa procesorului de 4 ori.

Avantajele arhitecturii *superpipeline:*

- creşte viteza de execuţie a instrucţiunilor datorită creşterii frecvenţei de ceas a procesorului;
- creşte numărul de operaţii procesate în paralel datorită creşterii numărului de stagii;
- creşte eficienţa operaţiilor executate într-un stagiu. De exemplu, dacă într-un *pipeline* obişnuit avem o operaţie care se execută într-un timp mai mic decât un ciclu procesor, rezultatul va fi disponibil tot la sfârşitul ciclului (Fig. 10 a). În schimb, la arhitectura *superpipeline*, rezultatul unei operaţii este disponibil cu frecvenţa tactului, care acum este mai rapid (Fig.10 b):

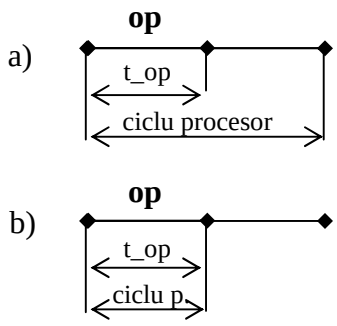


Figura 10. Execuţia unei operaţii corespunzătoare unui substagiu: a) într-o structură *pipeline*; b) într-o structură *superpipeline*

Dezavantaje ale arhitecturii *super pipeline*:

Principalul dezavantaj e reprezentat de faptul că dacă *pipeline*-ul trebuie golit de instrucţiunile încărcate în avans (de exemplu când apar instrucţiuni de salt în program), atunci există un număr mai mare de stagii care trebuie reiniţializate.

Aşa cum am văzut, procesoarele RISC sunt procesoare de uz general, dar există şi alte tipuri de procesoare care se aplică unui domeniu mai restrâns, obţinând performanţe mai bune pe acel domeniu specific (de exemplu procesoare DSP, procesoare VLIW si procesoare vectoriale). În continuare vor fi prezentate pe scurt procesoarele VLIW si procesoarele vectoriale.

Procesoarele VLIW (Very Large Instruction Word)

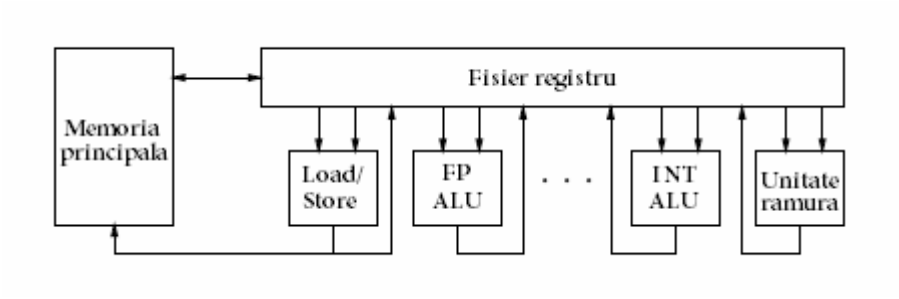


Figura 10. Structura internă a unui procesor VLIW

Procesorul VLIW e alcătuit din mai multe unităţi funcţionale echivalente unităţilor de execuţie din procesoarele superscalare, şi un fişier registru care conţine un număr mare de registre procesor (pot exista mai mult de 128 registre). O caracteristică a acestor procesoare este că ele dispun de un format mare al instrucţiunilor. Fiecare instrucţiune e compusă din mai multe sloturi de operaţii în care se plasează diferite operaţii de tip RISC, în aşa fel încât toate operaţiile dintr-o instrucţiune să poată fi executate in paralel.

Load/Store	FP add	FP mult	Ramura	. . .	INT ALU

Figura 11. Formatul instrucţiunii la procesorul VLIW

Programul compilator este cel care umple instrucţiunile cu operaţii care se pot executa în paralel. De aici rezultă caracteristica de *static scheduling* (planificare statică), adică determinarea paralelismului se face în timpul compilării şi nu al execuţiei. Rezultă astfel un avantaj al procesoarelor VLIW: ele nu necesită un hardware complex, cum era cazul la procesoarele superscalare (unde determinarea paralelismului se făcea prin hardware în timpul execuţiei).

Ca şi dezavantaj al procesoarelor VLIW: codul obiect rezultat în urma compilării este mai puţin compact decât la un procesor obişnuit, datorită faptului că nu toate sloturile dintr-o instrucţiune pot fi încărcate cu operaţii care se execută în paralel.

Diagrama execuţiei în timp a unei instrucţiuni este ilustrată în figura 12 (pentru un procesor având 3 sloturi pe instrucţiune).

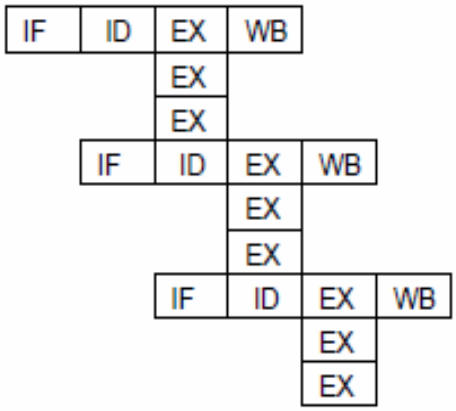


Figura 12. Diagrama execuţiei unei instrucţiuni la procesorul VLIW

Câteva exemple de procesoare VLIW:
- Philips Trimedia (utilizat pentru aplicaţii multimedia);
- arhitectura Intel pe 64 biţi (IA64) - o combinaţie între RISC si VLIW.

Procesoare vectoriale

În cele mai multe cazuri, procesoarele vectoriale nu funcţionează independent, ci sunt folosite pe post de coprocesoare alături de un procesor principal. Spre deosebire de procesoarele obişnuite, care operează cu scalari, procesoarele vectoriale pot opera pe vectori, adică pe şiruri de scalari. Ele pot fi de tipul registru-registru, în care vectorii se încarcă din registre vectoriale, iar rezultatele se salvează tot în registre vectoriale, sau pot să fie de tipul memorie-memorie, în care vectorii şi rezultatele se iau / se pun în memoria sistemului. Unităţile de execuţie vectoriale au o funcţionare de tip *pipeline* (fig.13).

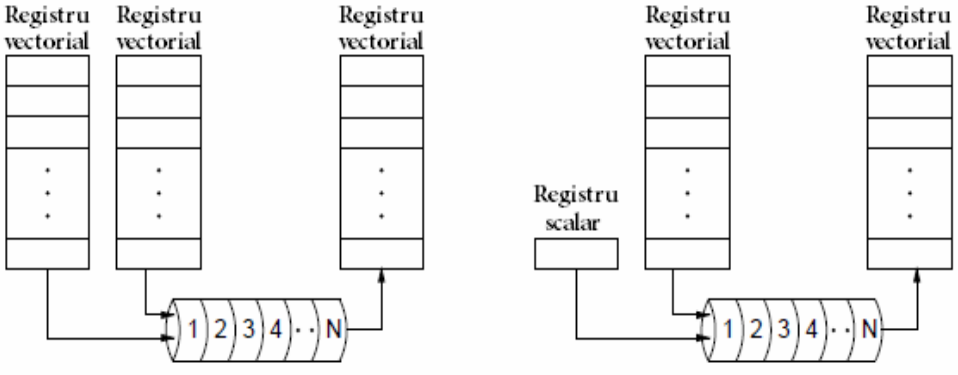


Figura 13. Unităţile de execuţie într-un procesor vectorial

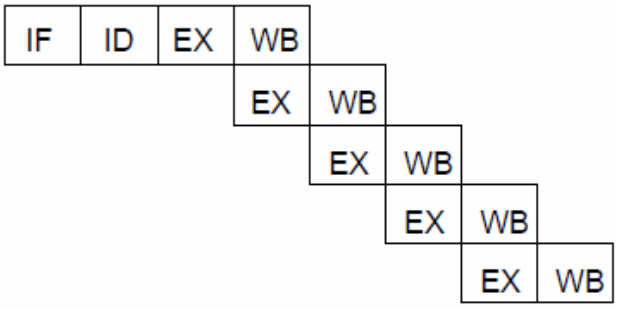


Figura 14. Diagrama de execuţie a unei instrucţiuni vectoriale

Domeniul de aplicabilitate al procesoarelor vectoriale este reprezentat de calculele ştiinţifice şi aplicaţiile multimedia, unde este necesară realizarea unui număr mare de operaţii pe structuri vectoriale.

MEMORIA CACHE

Memoria cache a apărut din necesitatea de a elimina diferenţa de viteză dintre procesorul rapid şi memoria sistem, mult mai lentă. Este o memorie statică (SRAM) de dimensiuni mici dar foarte rapidă, folosită pentru a aduce datele mai aproape de unităţile de execuţie din processor. Ca urmare a peformanţelor de funcţionare a memoriei cache, actualmente doar un procent de aproximativ 10% din accesele la memorie ale procesorului se fac din memoria sistem, iar restul de 90% se fac din memoria cache.

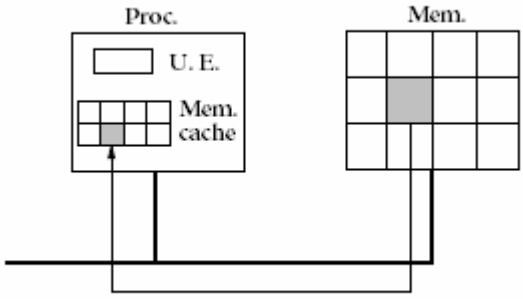


Figura 1. Memoria cache

Memoria cache se bazează pe principiul localităţii. Avem două tipuri de localitate:

a) localitate temporală – locaţiile de memorie accesate la un moment dat tind să fie accesate din nou în viitor;
b) localitate spaţială – datele aflate în vecinătatea unor date accesate tind sa fie şi ele accesate.

Exemple de memorii cache:

- memoria TLB (*Translation Look-aside Buffer*) este un cache special pentru translatarea între adresele fizice şi adresele virtuale din calculator. Datorită folosirii memoriei TLB, calculul adreselor pentru memoria virtuală nu se face la fiecare acces al memoriei, ci se ia din tabela TLB.

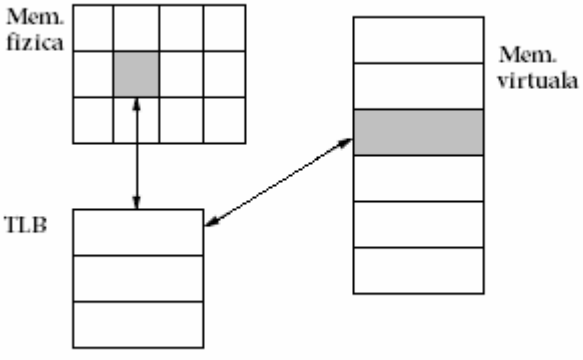


Figura 2. Memoria TLB

- memoria cache pentru instrucţiuni - este folosită pentru încărcarea în avans a instrucţiunilor în procesor ;
- memoria cache de date - folosită pentru încărcarea în avans în datelor necesare din memoria sistem.

Există 3 moduri de mapare a datelor din memoria fizică în memoria cache:

1) mapare directă
2) mapare cu asociere completă
3) mapare cu asociere pe blocuri (set asociativă)

1) Memoria cache cu mapare directă

În acest caz există o mapare directă, secvenţială între adresele din memoria principală şi locaţiile memoriei cache. Aici memoria fizică e împărţită în pagini, fiecare pagină fiind de dimensiunea memoriei cache. Atât paginile din memoria sistem cât şi memoria cache sunt împărţite în blocuri de aceeaşi dimensiune.

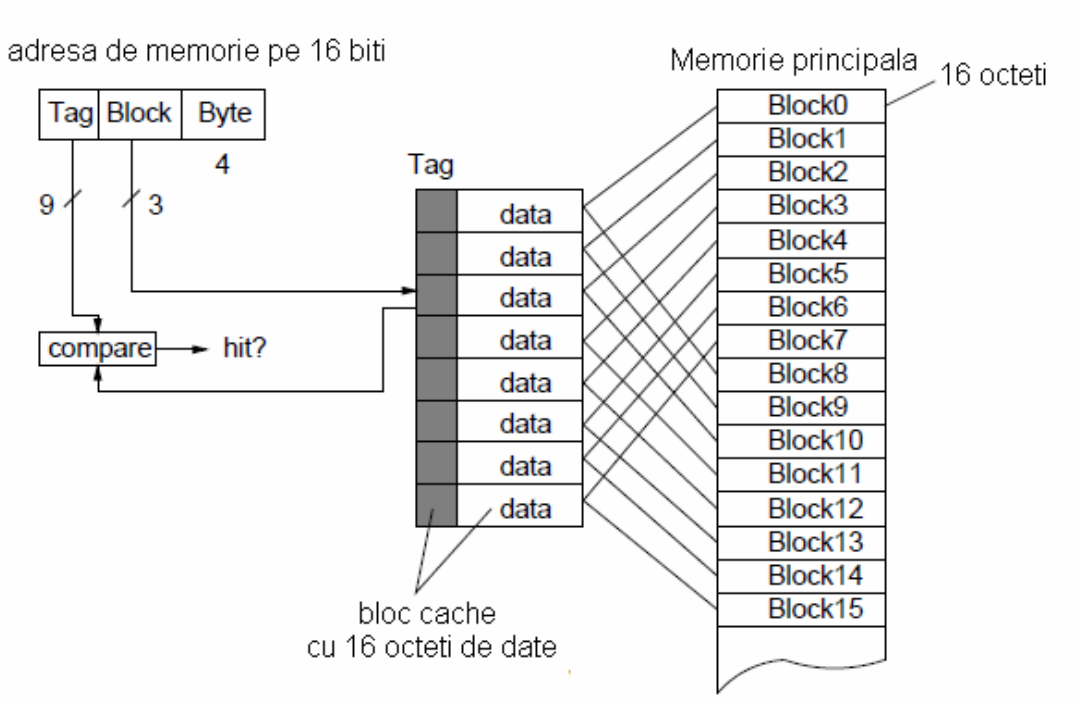


Figura 3. Memoria cache cu mapare directă

În exemplul de mai sus, o pagină conţine 8 blocuri; astfel, blocul 0 din memoria cache corespunde cu blocurile 0, 8, 16, etc. din memoria sistem.

Pentru a şti exact cu ce bloc din memorie corespunde un anumit bloc din memoria cache, se foloseşte un tag (indicator) la fiecare intrare din memoria cache. Tag-ul conţine cei mai semnificativi biţi ai adresei blocului din memoria principală, reprezentând de fapt adresa de pagină a blocului.

Dacă procesorul vrea să acceseze un bloc de date din memorie situat la o anumită adresă, mai întâi trebuie să verifice dacă blocul căutat se află sau nu în memoria cache. Pentru aceasta, se va proceda astfel:

a) se selectează o intrare din memoria cache pe baza informaţiei de bloc din adresa căutată ;

b) se compară câmpul tag din adresa cerută de procesor cu cel din intrarea cache;
c) dacă cele două sunt egale, avem asa-numitul cache-hit, adică blocul căutat din memoria principală se regăseşte în memoria cache;
d) dacă cele două tag-uri sunt diferite, atunci blocul căutat nu există în memoria cache şi el va fi încărcat din memoria principală.

Avantajul modului de mapare directă:
Memoriile cu mapare directă sunt memorii simple şi de mare viteză. Ele se pot utiliza ca memorii cache de nivel 2, putând avea dimensiuni mai mari.

2) **Memorii cache cu asociere completă**

Aici, un bloc din memoria sistem poate fi plasat oriunde în memoria cache.

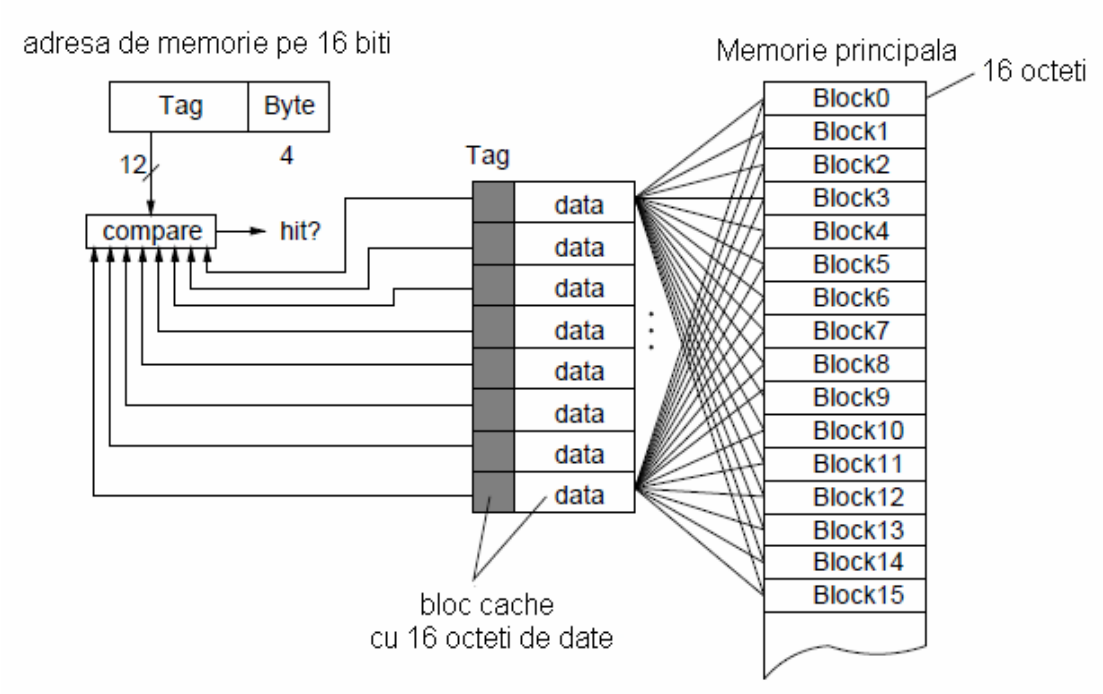


Figura 4. Memoria cache cu asociere directă

Pentru a detecta dacă un bloc din memoria sistem există în memoria cache, se face o comparare în paralel a tag-urilor din toate blocurile memoriei cache. Dacă se identifică un tag, înseamnă că s-a găsit blocul căutat.

Avantajul acestui tip de memorie: foloseşte un mod de mapare mai flexibil.

Dezavantaj: este mai costisitor de implementat datorită comparaţiilor care se fac în paralel.

3) **Memorii cache set asociative**

Acestea folosesc o combinaţie între maparea directă şi maparea cu asociere completă. Aici, memoria cache e împărţită în mai multe seturi de intrări, existând o mapare directă între blocurile din memoria principală şi seturile din memoria cache. În interiorul seturilor se face o mapare cu asociere completă.

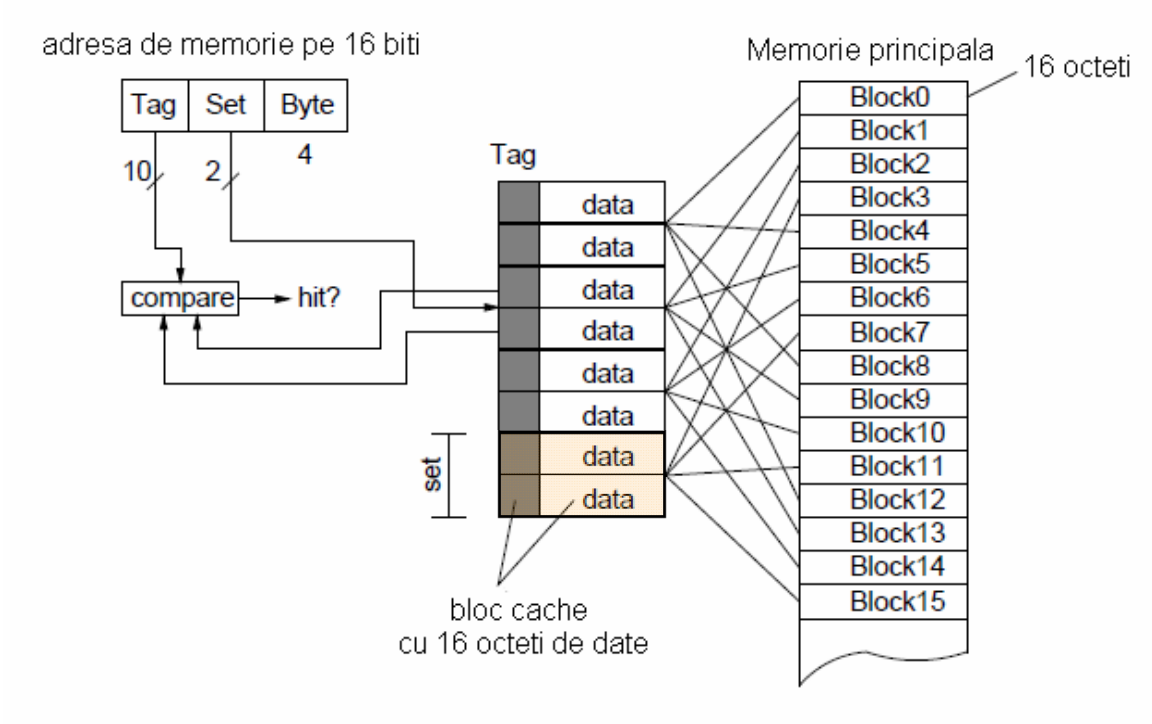


Figura 5. Memoria cache set-asociativă

Pentru a vedea dacă blocul dorit se găseste în memoria cache, se face o comparare în paralel a tag-urilor doar pentru intrările dintr-un singur set.

Avantajul memoriilor set-asociative: se reduce numărul de comparaţii efectuate în paralel; sunt mai puţin costisitoare decât memoriile complet asociative; se folosesc ca memorii cache de nivelul 1, având performanţe ridicate.

Observaţie: În sistemele actuale există o ierarhie de memorii cache organizată pe mai multe nivele; în funcţie de poziţia memoriei cache faţă de procesor, putem avea: memorie cache de nivel 1 (care se găseşte în interiorul capsulei procesorului), memorie cache de nivel 2, de nivel 3, etc.

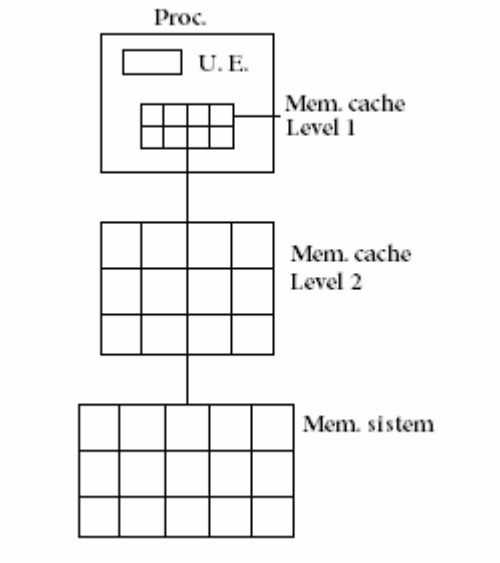


Figura 6. Ierarhia de memorii cache într-un sistem de calcul

Strategii utilizate pentru memoriile cache:

În legătură cu anumite evenimente ce pot sa apară la accesarea memoriei cache, se stabilesc următoarele strategii:

S1) Strategia *cache-miss*. Această strategie se referă la memorii cache complet asociative. Astfel dacă e nevoie de un bloc de memorie care nu se regăseşte în cache (*cache-miss*), trebuie să se stabilească care anume dintre blocurile existente în memoria cache va fi înlocuit de noul bloc care va fi adus din memoria sistem.
Strategia de înlocuire poate fi:

- aleatorie
- *first in, first out* - se înlocuieşte primul bloc introdus în cache
- se înlocuieşte cel mai puţin recent utilizat. Aici se înlocuieşte blocul care nu a fost accesat de cel mai mult timp (cel mai vechi).

S2) Strategii de scriere a memoriilor cache. Să presupunem că procesorul efectuează o operaţie STORE (scriere în memorie). Există 2 posibilităţi:

a) valoarea se scrie în memoria cache **şi** în memoria sistem. Aceste memorii se numesc *write-through*; valoarea respectivă se scrie prin cache în memoria sistemului.

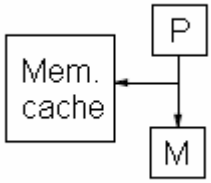


Figura 7. Memoria *write-through*

b) scrierea se face doar în memoria cache. Scrierea în memoria sistem se face ulterior doar când este necesar acest lucru. Aceste memorii se numesc *write-back*.

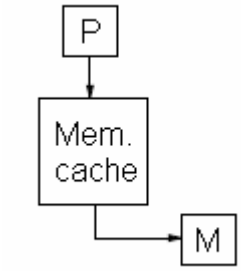


Figura 8. Memoria *write-back*

Avantajele memoriei *write-through*:

- memoria principală e întotdeauna consistentă cu memoria cache.

Dezavantajele memoriei *write-through*:

- această memorie este mai înceată pentru că se aşteaptă scrierea în memoria sistem. La o operaţie STORE se aşteaptă scrierea în memoria sistem, care este mai lentă.
- se măreşte traficul între magistrală şi memoria principală datorită faptului că fiecare acţiune STORE a procesorului se propagă în memoria principală.

Acest lucru nu se petrece la memoria *write-back*. În schimb, la acest tip de memorie este necesar un bit suplimentar pentru fiecare bloc din memoria cache, care să specifice dacă blocul este consistent cu memoria principală. Acest bit se numeşte bit *dirty*.

Dacă bitul *dirty* e 0, înseamnă că blocul din memoria cache e consistent (identic) cu blocul din memoria sistem. Dacă bitul *dirty* e 1, înseamnă că blocul nu e consistent (nu e actualizat în memoria sistem). Când un astfel de bloc *dirty* trebuie să fie eliminat din memoria cache, el va fi scris mai întâi în memoria principală.

Dezavantaj al memoriei *write-back*:

La schimbarea contextului program de către un sistem de operare multi-tasking se foloseste întotdeauna un context diferit de memorie, de aici rezultă că un mare număr de blocuri din memoria cache vor trebui să fie înlocuite şi astfel timpul de comutare a contextului program creşte foarte mult.

S3) Strategii *write-miss*. Evenimentul *write-miss* apare când există un *cache-miss* la o instrucţiune STORE, adică blocul dorit nu se află în memoria cache atunci când procesorul vrea să scrie în acesta.

Memoriile *write-back* folosesc 2 tipuri de strategii:

a) Strategia "alocă la scriere" (*allocate on write*). Aici memoria cache alocă un bloc cache, după care realizează acţiunea de a scrie în acel bloc.
b) Strategia "încarcă la scriere" (*fetch on write*). Aici memoria cache mai întâi citeşte blocul din memoria principală, apoi scrie data în blocul citit.

Memoriile *write-through* folosesc strategia "nu se alocă la scriere" (*no allocate on write*). Aici, dacă există un *cache-miss* la scriere, atunci nu se mai alocă un bloc în memoria cache, ci data trimisă de procesor va fi scrisă direct în memoria principală a sistemului.

REŢELE DE INTERCONECTARE

Scopul unei reţele de interconectare este de a permite schimbul de date între procesoarele unui sistem paralel. Există două noţiuni fundamentale legate de reţele: comutarea şi rutarea.

1. **Comutarea (*network switching*)**

Se referă la modul de transmitere a datelor între procesoarele din reţea. Există două tipuri principale:

a) Comutarea prin circuite
b) Comutarea prin pachete

Comutarea prin circuite se referă la transmiterea datelor prin circuite dedicate. Aici se stabileşte o legătură fizică între un procesor sursă şi un procesor destinaţie, legătură ce trebuie să rămână stabilă pe întreaga durată a transmisiei datelor.

Comutarea prin pachete - aici datele de transmis se împart în blocuri de dimensiuni mici numite pachete, iar apoi se alocă un canal de comunicaţie doar pentru transmiterea unui singur pachet.

2. **Rutarea (*network routing*)**

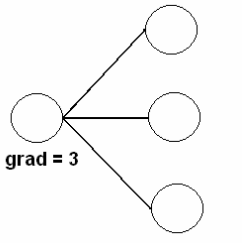
Se referă la modalitatea de a conduce datele prin reţea. Rutarea defineşte astfel o rută pe care datele o vor urma pentru a ajunge la destinaţie. Rutarea este în strânsă legătură cu topologia (structura) reţelei.

Clasificarea reţelelor în funcţie de topologie:

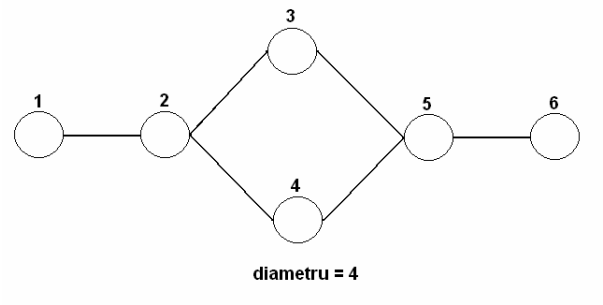
a) reţele directe; aici există o conectivitate punct la punct între procesoarele vecine
b) reţele indirecte; acestea utilizează canale de comunicaţie care sunt comune mai multor procesoare

Noţiuni fundamentale legate de reţele:

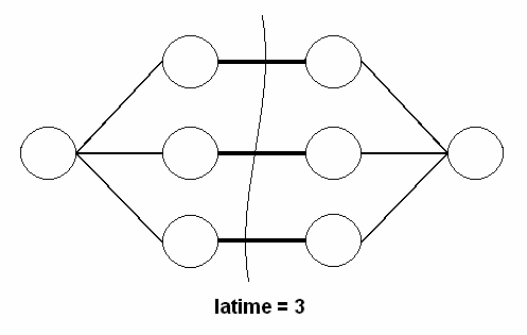
- gradul unui nod: reprezintă numărul de canale de comunicaţie care sunt legate de acel nod.



- diametrul reţelei: reprezintă distanţa maximă între două noduri ale reţelei.



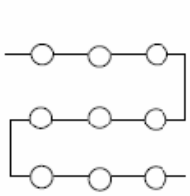
- lăţimea secţiunii: reprezintă numărul de canale de comunicaţie între două jumătăţi ale reţelei.



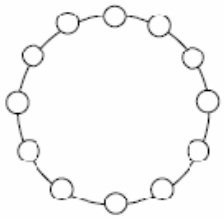
- toleranţa la erori: reprezintă numărul de rute alternative între două noduri.
- scalabilitatea reţelei: reprezintă posibilitatea de expandabilitate a reţelei.
- rata de transfer: reprezintă cantitatea de date transferate în unitatea de timp.
- latenţa: reprezintă durata maximă de transfer a unei date prin reţea.

A. Reţele directe

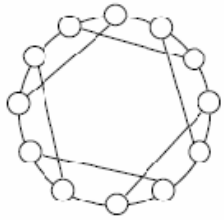
În acest caz există legături punct la punct între nodurile învecinate ale reţelei. Ele sunt numite reţele statice pentru că aceste legături punct la punct definite sunt fixe (nu se schimbă pe întreaga durată a transmisiei datelor).
Exemple:



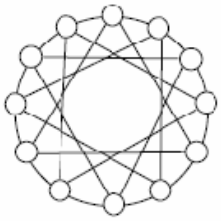
Linear array



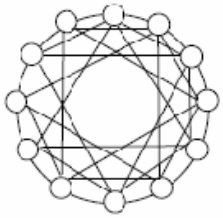
Ring



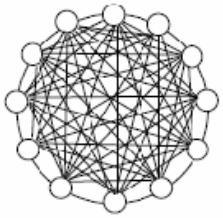
Chordal ring of degree 3



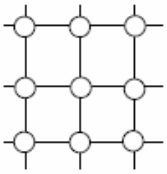
Chordal ring of degree 4
(same as Illiac mesh)



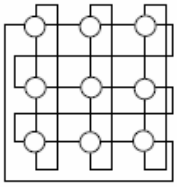
Barrel shifter



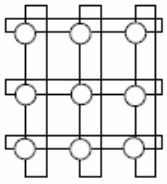
Completely connected



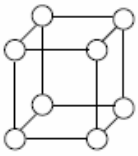
Mesh



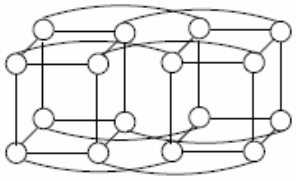
Illiac mesh



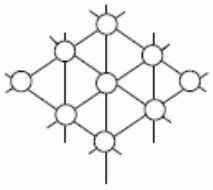
Torus



3-Hypercube



4-Hypercube



Systolic array

Tip de retea	Gradul	Diametrul	Latimea Sectiunii
Linear array	2	$N-1$	1
Ring	2	$\lfloor \frac{N}{2} \rfloor$	2
Completely conn.	$N-1$	1	$(\frac{N}{2})^2$
Binary tree	3	$2(log_2 N - 1)$	1
2D-mesh	4	$2(\sqrt{N}-1)$	$\sqrt{N}$
2D-torus	4	$2\lfloor \frac{\sqrt{N}}{2} \rfloor$	$2\sqrt{N}$
Hypercube	$log_2 N$	$log_2 N$	$\frac{N}{2}$

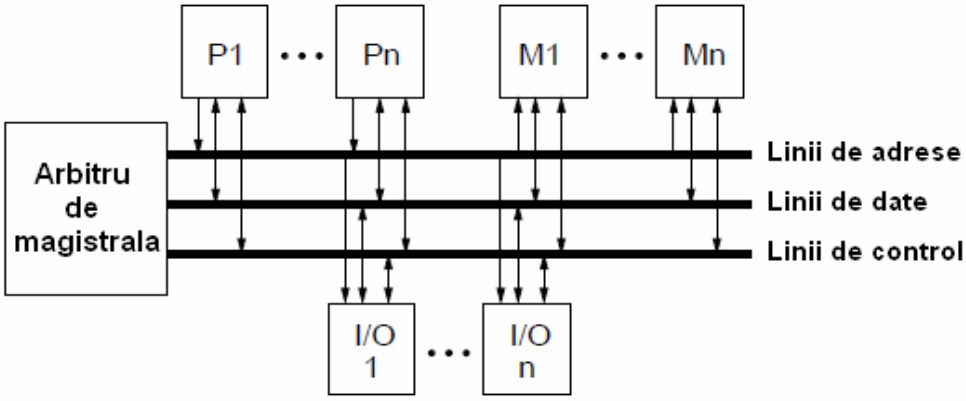
B. Reţele indirecte

Reţelele indirecte mai sunt numite şi reţele dinamice. Spre deosebire de reţelele statice, aici un nod nu are vecini fixaţi. Topologia reţelei poate fi schimbată dinamic în funcţie de cerinţele aplicaţiei. Există trei tipuri de reţele dinamice:

a) Reţele cu magistrală
b) Reţele multi-stagiu
c) Reţele *crossbar* (cu comutatoare)

a) Reţele bazate pe magistrală:

Ele se bazează pe o magistrală alcătuită din mai multe linii de bit, la care se conectează diferite resurse (procesoare, memorii, dispozitive de intrare/ieşire etc). Magistrala poate conţine linii de adresă, de date şi de control.



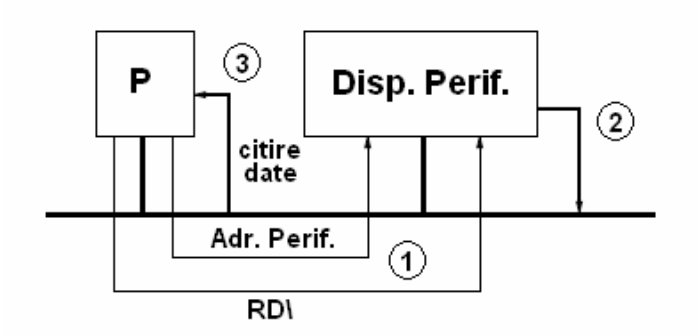
La o magistrală se pot conecta atât dispozitive de tip *master* (care iniţiază un transfer de date pe magistrală), cât şi dispozitive de tip *slave*. Dacă la magistrală sunt conectate mai multe dispozitive *master*, atunci va fi necesar un dispozitiv de arbitrare a magistralei.

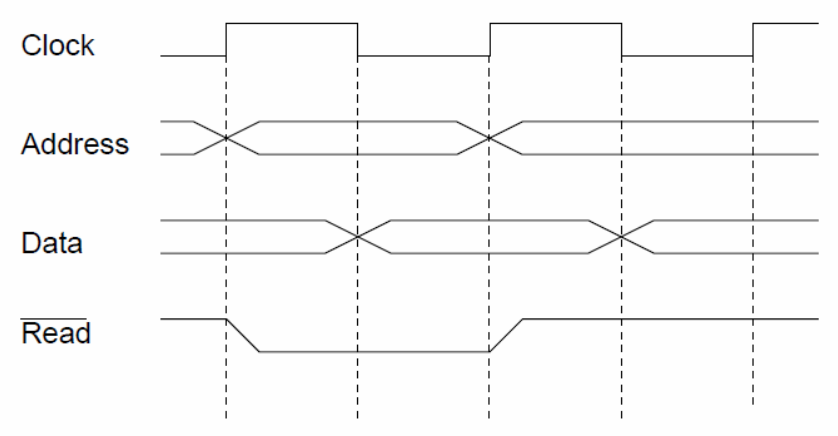
Există două tipuri de magistrale:

1) Magistrale sincrone
2) Magistrale asincrone

1) **Magistrale sincrone**

Aici, toate tranzacţiile pe magistrală sunt sincronizate folosind un semnal de ceas. Exemplu: citirea unei date de la un dispozitiv periferic.





2) Magistrale asincrone

Aici, acţiunile nu sunt la fel de predictibile ca la o magistrală sincronă. La magistrala asincronă se utilizează un protocol de tip *handshaking*, pentru a se indica apariţia unui eveniment pe magistrală, de exemplu validitatea datelor pe linia de date a magistralei poate fi indicată de un semnal de comandă numit *Strobe*.

În general, magistralele asincrone sunt mai complexe, mai scumpe şi mai puţin eficiente decât magistralele sincrone datorită protocolului care trebuie realizat cu semnale fizice. Totuşi, magistralele asincrone au avantajul că sunt mai flexibile şi prin intermediul lor se pot uşor conecta resurse cu viteze diferite.

Exemplu de magistrală sincronă: magistrala procesor-memorie.

Exemplu de magistrală asincronă: magistrala SCSI, care asigură comunicaţia cu dispozitivele periferice.

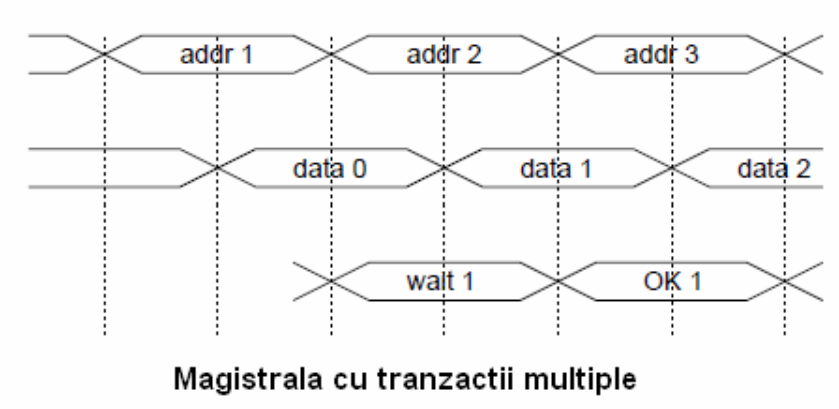
Magistrale *split transaction:*

În cazul unei magistrale obişnuite, magistrala rămâne ocupată pe întreaga durată a unei tranzacţii, chiar şi atunci când magistrala aşteaptă date de la un periferic, lucru care este ineficient. Pentru a mări eficienţa magistralei, s-au creat aşa numitele magistrale cu tranzacţii multiple (*split transaction bus*). În acest tip de magistrale, pot fi active la un moment dat mai multe tranzacţii diferite.

Exemplu:

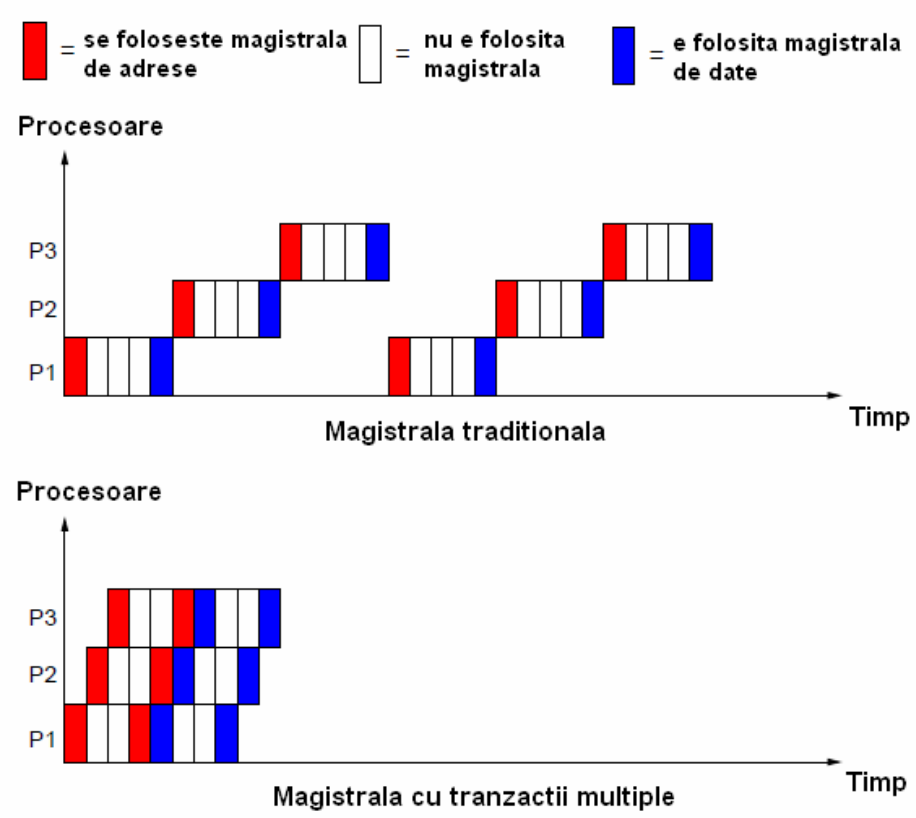
Considerăm o tranzacţie de citire de date de la un dispozitiv periferic pe o magistrală sincronă cu tranzacţii multiple. Această tranzacţie funcţionează astfel:

a) se pune adresa perifericului de la care se citesc datele pe magistrala de adrese, şi semnalul de citire READ pe magistrala de comenzi.
b) după ce adresa e citită de dispozitivul periferic, liniile de adresă sunt pregătite să primească o altă adresă de la un alt periferic.
c) când dispozitivul periferic e pregătit să furnizeze datele cerute, el va plasa aceste date pe magistrala de date şi va specifica destinatarul datelor prin intermediul unei etichete (*tag*) furnizate pe linii suplimentare de comandă.



Magistrala cu tranzactii multiple

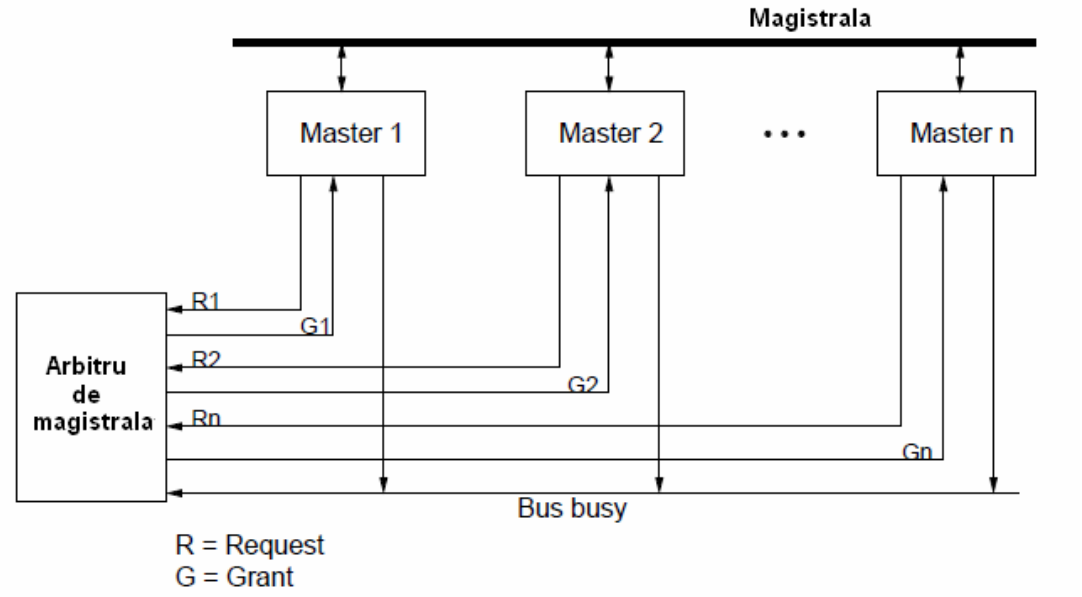
Avantaj: magistrala *split transaction* măreşte rata de transfer a datelor.
Dezavantaj: poate să mărească latenţa tranzacţiilor, pentru că acum magistrala multiplexează mai multe tranzacţii în acelaşi timp.



Arbitrarea magistralei:

Arbitrarea este necesară atunci când există mai multe dispozitive *master* conectate pe magistrală. Dacă două sau mai multe dispozitive *master* vor să acceseze magistrala în acelaşi timp, controlerul de magistrală va decide cărui dispozitiv i se va permite mai întâi accesul. Există două modalităţi de arbitrare:

a) Schema de arbitrare bazată pe un semnal de *Request* (cerere) şi *Grant* (acceptare). Când un *master* vrea să acceseze magistrala, trimite o cerere către arbitru (controler). În momentul în care arbitrul permite accesul, el va trimite dispozitivului *master* semnalul de grant. Dacă există mai multe dispozitive care vor să acceseze simultan magistrala, atunci va fi aplicată o politică de selectie bazată pe priorităţi.

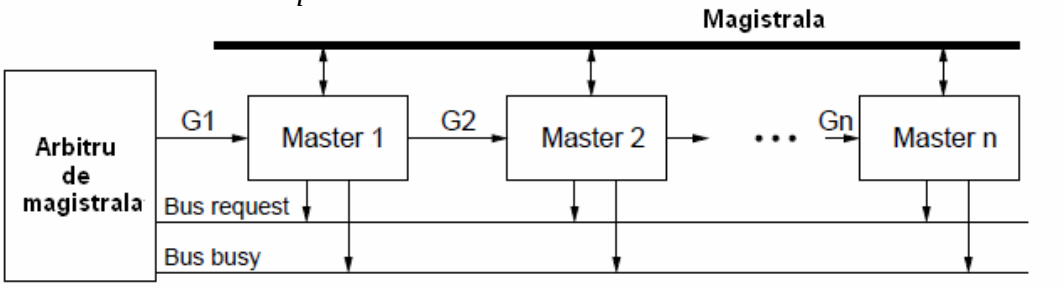


Când un *master* a câştigat accesul la magistrală, el va activa semnalul *Bus Busy*. Cât timp acest semnal este activ, arbitrul nu va mai acorda acces la magistrală unui alt dispozitiv *master*.

Avantaj: eficienţa în funcţionare.

Dezavantaj: reprezintă o implementare scumpă datorită unui număr mare de linii de control.

b) Schema de arbitrare *daisy-chaining* (arbitrare cu înlănţuire). Aici există un singur semnal de grant care se propagă de-a lungul unui lanţ de *mastere*. Când un *master* primeşte un semnal de grant, el poate face o cerere de acces la magistrală prin activarea semnalului *Request*.



Dacă un *master* nu vrea să acceseze magistrala, el dă mai departe semnalul de grant către următorul *master*.

Avantaj: este mai puţin costisitoare decât schema precedentă.

Dezavantaj: datorită faptului că semnalul de grant circulă mai lent pe lanţul de *mastere*, va dura mai mult până când o cerere de acces la magistrală va fi acceptată.

Reţele Multistagiu

O reţea multistagiu constă din mai multe nivele sau stagii de *switch*-uri (comutatoare) conectate între ele. Un *switch* asigură legături interne între mai multe intrări şi mai multe ieşiri.

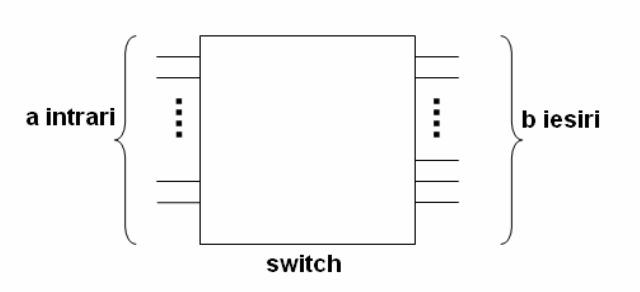


Figura 1. Un *switch* asigură conexiunea dintre *a* intrări şi *b* ieşiri.

Într-o reţea multistagiu, cu cât numărul de stagii este mai mare, cu atât întârzierea reţelei va fi mai mare. Alegând un anumit tipar de interconectare a *switch*-urilor, se pot realiza diferite topologii de reţele.

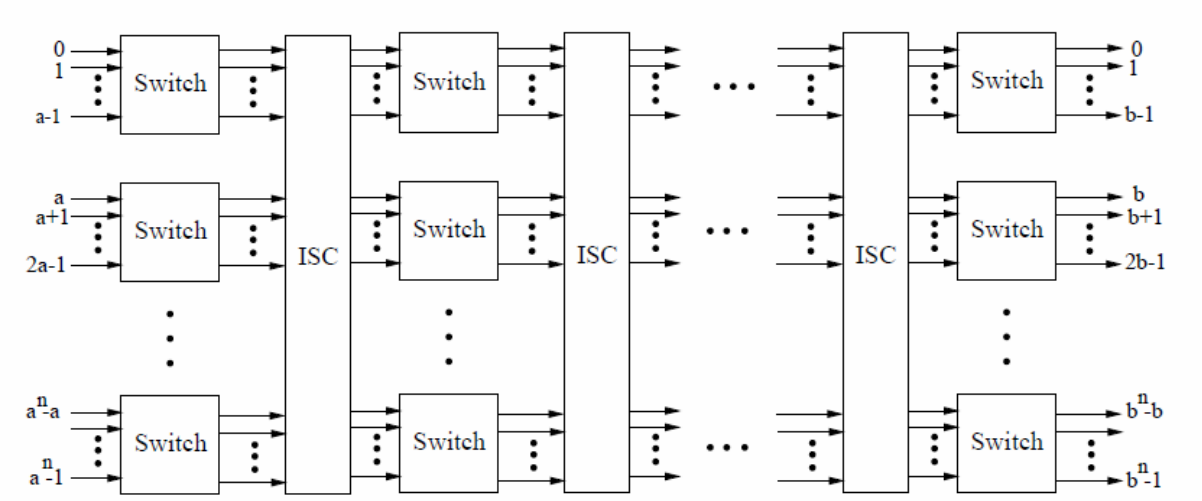


Figura 2. Generalizare a unei retele multistagiu interconectate (MIN), construita cu switch-uri a x b si un tipar pentru conexiunile interstagiu (ISC).

Un exemplu de reţele multustagiu sunt reţelele de Omega (fig. 3). Aici, conectarea *switch*-urilor realizează o distribuţie completă a semnalelor de intrare către ieşiri (de la orice intrare se poate ajunge la orice ieşire).

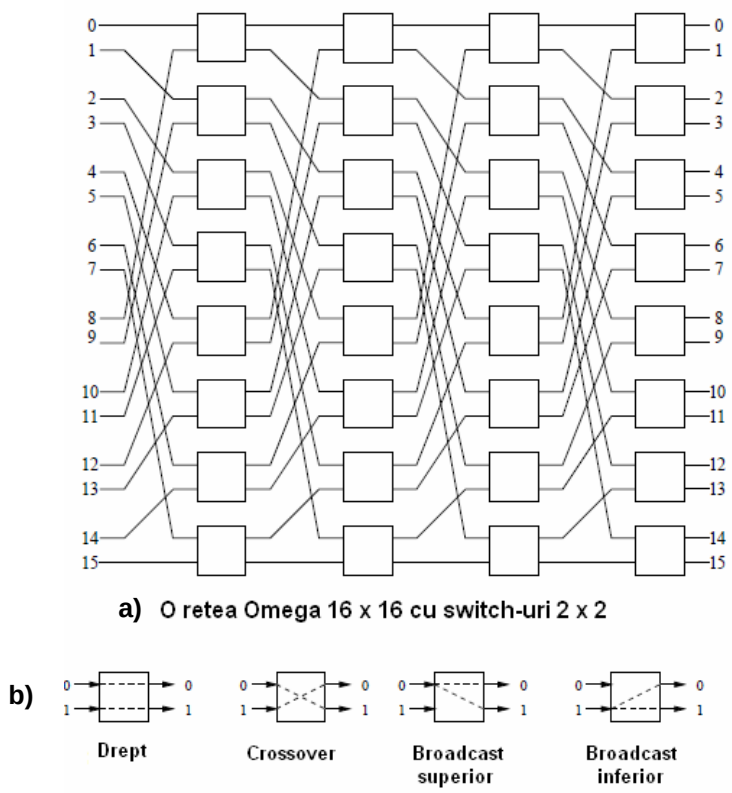


Figura 3. a) Exemplu de reţea Omega. b) Conexiuni aceptate ale *switch*-urilor

Reţelele multistagiu pot să fie **blocante** sau **neblocante**. Într-o reţea blocantă nu se pot realiza simultan toate legăturile de la intrare la ieşire din cauza unor conflicte de comutare internă a *switch*-urilor. La *switch*-urile unde apar astfel de conflicte de comutare, o parte din semnalele de la intrare sunt blocate până când canalele de ieşire necesare devin disponibile. Reţelele neblocante suportă orice conexiune fără blocare.
De exemplu, reţelele Omega sunt reţele blocante (fig. 4):

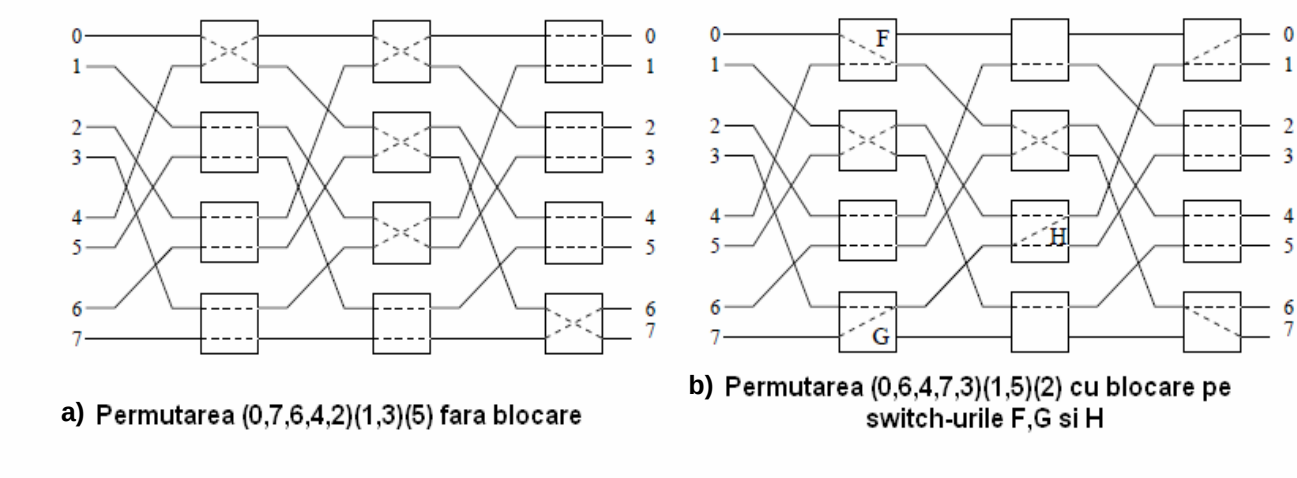


Figura 4. Exemplu de conexiuni într-o reţea Omega.
a) fără blocare; b) cu blocare

În figura 4 a), permutarea (0,7,6,4,2)(1,3)(5) semnifică următoarele conexiuni (neblocante):

0->7; 7->6; 6->4; 4->2; 2->0; 1->3; 3->1; 5->5.

În figura 4 b), conexiunile dorite sunt blocante:

0->6; 6->4; 4->7; 7->3; 3->0; 1->5; 5->1; 2->2.

Reţele *Crossbar*

O reţea de tip *crossbar* constă dintr-o matrice de comutatoare simple de tip *on/off* care pot realiza sau întrerupe o legătură de la o intrare către o ieşire:

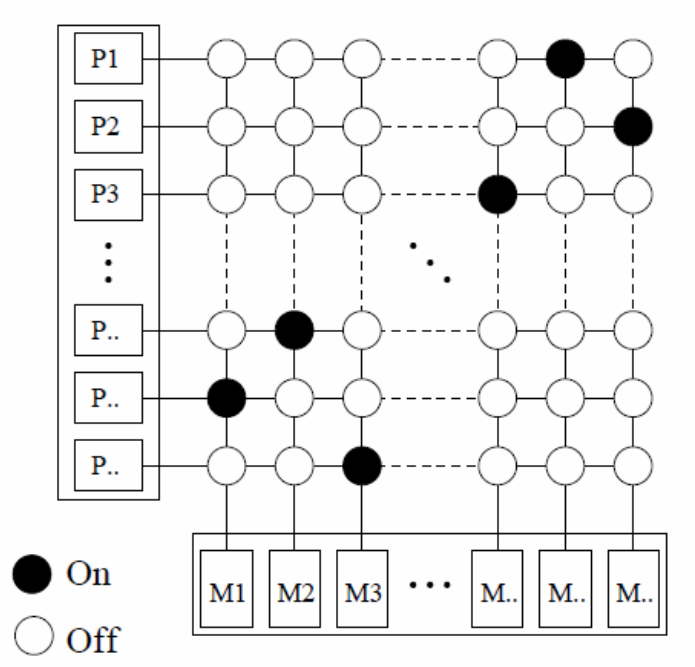


Figura 5. Reţea *Crossbar*

Într-o reţea de tip *crossbar*, pe o linie verticală nu putem avea mai multe *switch*-uri setate pe *on*, adică nu se poate ca mai multe procesoare să acceseze simultan un acelaşi bloc de memorie. Pe o linie orizontală putem avea mai multe *switch*-uri setate, rezultând în acest caz o emisie *multicast* de la un procesor la mai multe blocuri de memorie.

Avantaj: comunicarea prin *switch*-uri *crossbar* este neblocantă, adică toate combinaţiile de legături între procesoare şi memorii sunt posibile fără conflicte (respectând condiţiile anterioare).

Sunt posibile 3 implementări hardware pentru o reţea de tip *crossbar*(fig.6):

- folosind un lanţ de multiplexoare;
- folosind două reţele de semnale: una de intrări, alta de ieşiri, conectate prin circuite *three-state*;
- folosind un buffer de memorie în care intrările corespund cu procesoarele; fiecare element din buffer indică modulul de memorie RAM care poate fi accesat de procesorul respectiv.

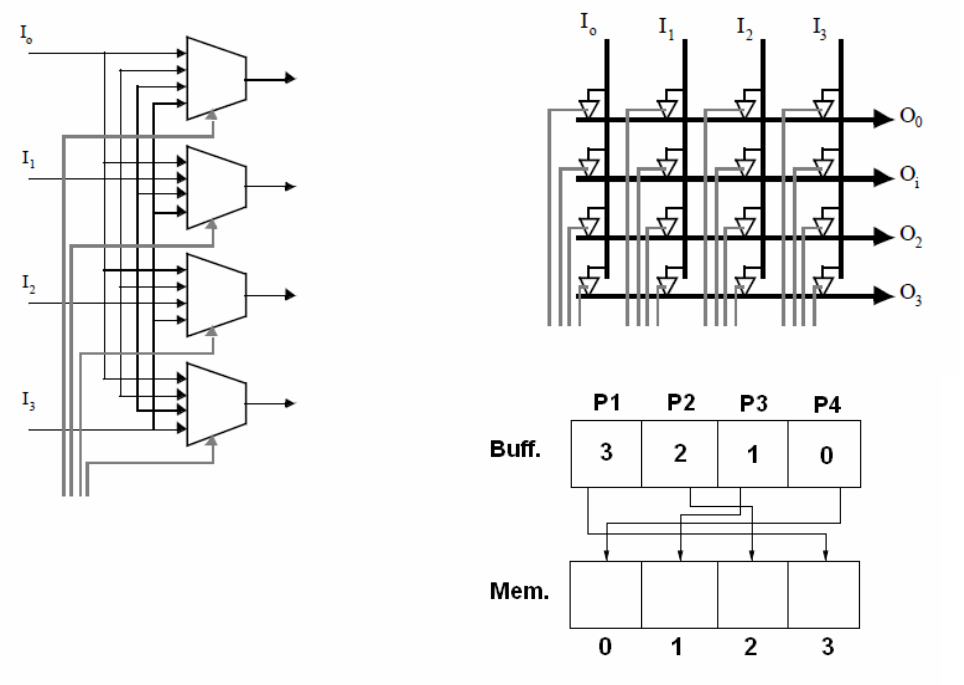


Figura 6. Implementări hardware pentru o reţea de tip *crossbar*

Comparaţie între reţelele directe şi cele indirecte:

a) La reţelele directe există canale de comunicaţie dedicate între procesoarele vecine. La reţelele indirecte nu există legături punct la punct între procesoare.

b) La reţelele directe este necesar un algoritm de rutare pentru ca informaţiile să ajungă de la sursă la destinaţie. La reţelele indirecte doar reţelele multistagiu necesită rutare. Aici *switch*-urile comutate într-un anumit mod vor asigura legătura între sursă şi destinaţie. Magistralele nu necesită rutare iar reţelele *crossbar* necesită comutarea unui singur *switch* pentru efectuarea conexiunii.

c) Reţelele directe pot fi uşor scalabile, adică se pot adăuga noduri în reţea cu modificări minime. Reţelele indirecte sunt mai greu scalabile, astfel magistralele pot ajunge la saturaţie când se depăşeşte un anumit număr de procesoare. La reţelele multistagiu şi *crossbar* creşterea numărului de procesoare devine scumpă datorită conexiunilor şi *switch*-urilor care trebuie adăugate în plus.

d) Reţelele directe folosesc de obicei comutarea prin pachete; reţelele *crossbar* şi magistralele folosesc comutarea prin circuite. La reţelele multistagiu, dacă

switch-urile sunt setate înaintea începerii comunicaţiei, atunci va fi realizată comutarea prin circuite. Dacă *switch*-urile pot comuta dinamic, în funcţie de ruta unui mesaj, atunci se poate implementa comutarea prin pachete.

Comutarea prin pachete (*packet switching*)

Reţelele care utilizează comutarea prin pachete folosesc *switch*-uri dedicate pentru transmiterea pachetelor. Aceste *switch*-uri transmit pachetele în reţea până când ele ajung la nodul destinaţie.

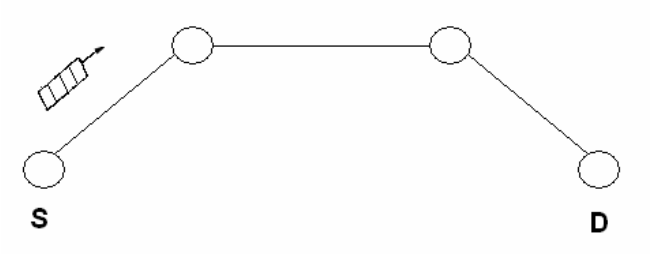


Figura 7. Transmiterea pachetelelor de la sursă la destinaţie

La reţelele directe, *switch*-urile fac parte din nodurile reţelei. Aici, *switch*-urile transmit pachetul mai departe dacă nodul curent este diferit de nodul destinaţie, sau copiază pachetul în memoria locală dacă nodul curent este acelaşi cu nodul destinaţie.

La reţelele indirecte, *switch*-urile nu fac parte din nodurile de procesare.

Există 3 modalităţi de comutare a pachetelor:

1) *Store & Forward*
2) *Wormhole routing*
3) *Virtual cut-through*

1) Store & Forward

Aici, pachetul este cea mai mică unitate de transfer a datelor. *Switch*-urile din reţea dispun de buffere pentru memorarea locală a pachetelor.

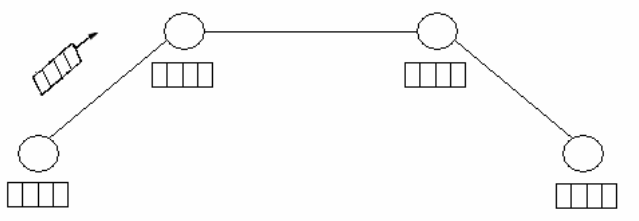


Figura 8. Transmisia ***Store & Forward***

În drumul de la sursă la destinaţie, un pachet ce tranzitează un nod va fi mai întâi memorat în întregime în nodul respectiv (etapa ***Store***), apoi va fi transmis mai departe către nodul destinaţie (etapa ***Forward***). Tehnica este simplu de implementat, dar toleranţa la erori nu e foarte bună.

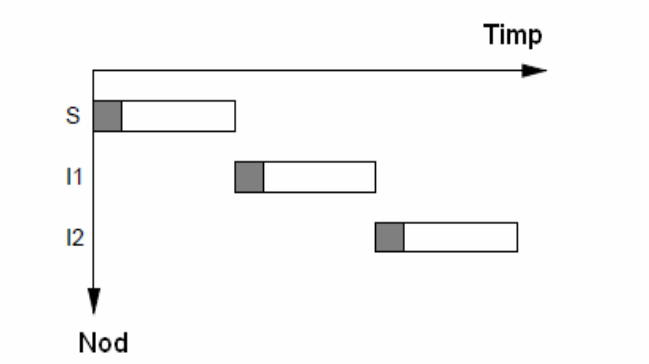


Figura 9. Diagrama de timp a transmisiei unui pachet - ***Store & Forward***

2) Wormhole routing

În cadrul acestei metode, pachetele sunt împărţite în blocuri de dimensiuni mici (de la 1 la câţiva octeţi). Blocul de date minimal care este transmis pe liniile de comunicaţie se numeşte **flit** (*Flow Control Digit*). Datele sunt conduse în reţea în manieră *pipeline* flit cu flit, toate flit-urile urmând acelaşi traseu. Când un nod recepţioneaza un flit, îl transmite imediat catre nodul următor şi apoi va recepţiona următorul flit.

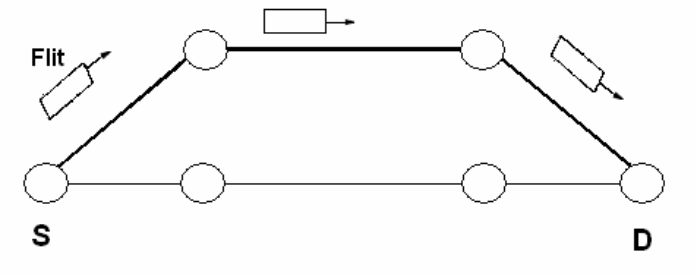


Figura 10. Transmisia ***Wormhole routing***

Informaţia de rutare e stocată în primele flit-uri din pachet. Pe toată durata transmiterii unui pachet, canalele de comunicaţie alocate la început rămân ocupate până la transmisia ultimului flit din pachet, care le eliberează.

Avantajul metodei: timpul de tranziţie al datelor în noduri scade, flit-urile având dimensiuni mai mici decât dimensiunea unui pachet.

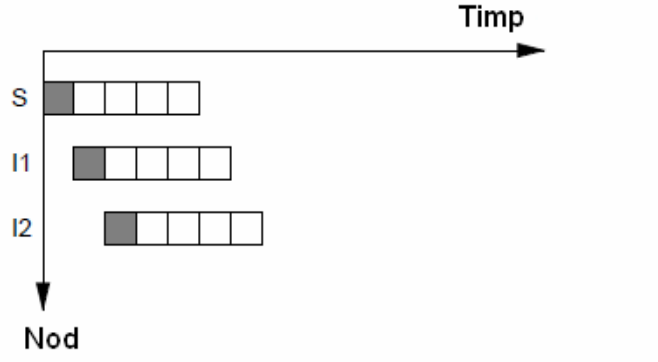


Figura 11. Diagrama de timp a transmisiei unui pachet - ***Wormhole routing***

Dezavantaj: metoda poate duce la saturarea reţelei. Astfel, atunci când *header*-ul unui pachet e blocat, rămân blocate toate canalele de comunicaţie alocate pentru transmisia pachetului, fiecare flit asteptând într-un nod intermediar. Aceasta situaţie poate duce de asemenea şi la blocarea altor pachete care circulă în reţea.

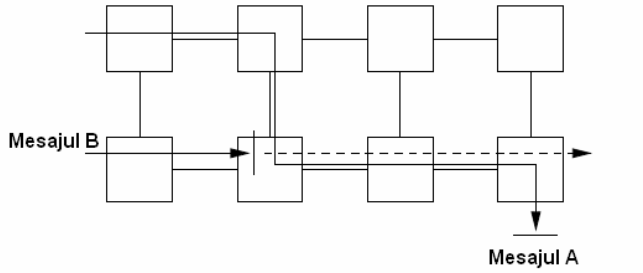


Figura 12. Exemplu de blocare a mesajelor la metoda ***Wormhole routing***

Timpii de transmisie a pachetelor pe liniile de comunicaţii se calculează astfel:

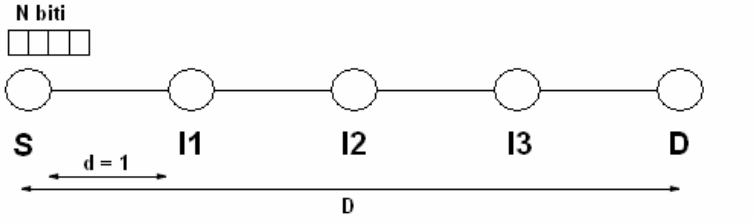


Figura 13. Transmiterea unui pachet – calculul timpilor de transmisie

- ***Store and forward***: - N = numărul de biţi dintr-un pachet
  - W = lăţimea de bandă a canalului

$$T1 = \frac{N}{W} \times D$$

- ***Wormhole routing***: - F = numărul de biţi dintr-un flit

$$T2 = \frac{N}{W} + \frac{F}{W} \times (D-1)$$

unde:

$\frac{N}{W}$ - timpul în care ultimul flit din pachet ajunge la nodul vecin I1

$\frac{F}{W} \times (D-1)$ - timpul după care ultimul flit (de dimensiune F) ajunge de la nodul I1 la destinaţie

3) Virtual cut-through

La această metodă, bufferele din fiecare nod au mărimea egală cu lungimea unui pachet. Transmisia pachetului se face *pipeline* flit cu flit ca şi la ***Wormhole Routing***. Dacă se blochează un *header* al unui pachet, transmisia celorlalte flit-uri continuă până când toate fliturile ajung în nodul care s-a blocat. Astfel canalele alocate se pot elibera, reducând gradul de saturare al reţelei.

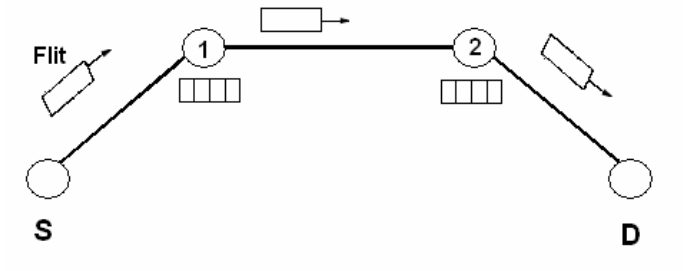


Figura 14. Metoda ***Virtual cut-through***

Pentru controlul fluxului de date (asigurarea faptului că datele nu se pierd in reţea), se poate folosi un protocol de tip *handshaking*, care este bazat pe o pereche de semnale ***Request/Acknoledge***:

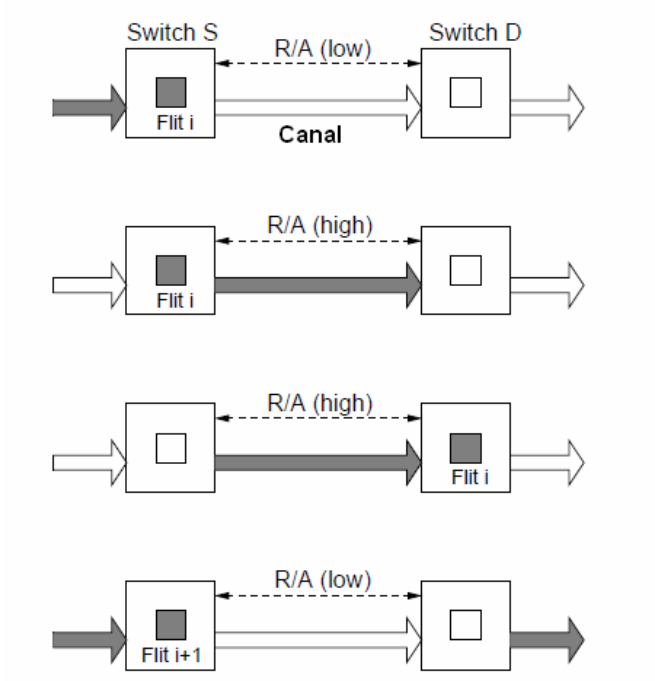


Figura 15. Protocol de transmisie bazat pe semnale ***Request/Acknoledge***

TEHNICI DE RUTARE

Rutarea determină calea sau ruta pe care datele o urmează de-a lungul reţelei pentru a ajunge la destinaţie. Există 2 metode de rutare:

a) Rutare stabilită de la nivelul nodului sursă (*Source-based Routing*);
b) Rutare locală (*Local Routing*), în care determinarea rutei se face în cadrul fiecărui *switch* din reţea.

a) Rutare stabilită la nivelul nodului sursă

În acest caz, nodul sursă stabileşte calea de urmat prin reţea. Calea se scrie în header-ul pachetului şi stabileşte direcţia pe care pachetul trebuie să o urmeze pentru fiecare nod.

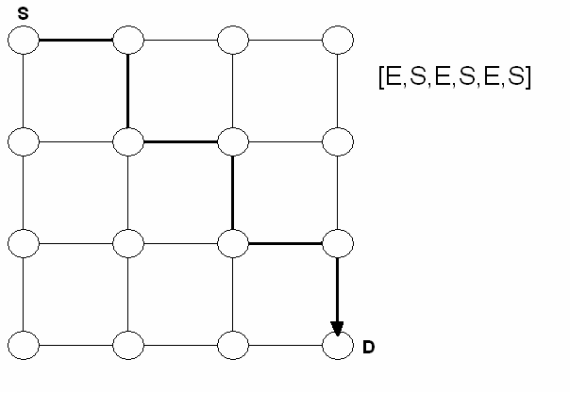


Figura 1. Rutare stabilită la nivelul nodului sursă

Switch-ul din fiecare nod al reţelei va trimite pachetul prin reţea în conformitate cu informaţia de direcţie curentă (care apoi e eliminată din header).

Dezavantaje:

- Deoarece pachetul include calea completă de parcurs prin reţea, aceasta va duce la creşterea dimensiunii fiecărui pachet.
- Pachetele nu pot anticipa defecţiunile *switch*-urilor din reţea şi nu pot alege o altă rută dacă un canal de comunicaţie e blocat.

b) **Rutarea locală**

Decizia rutei se face la nivelul fiecărui *switch* din reţea. Ca urmare, nodul sursă va specifica în headerul pachetului doar identificatorul nodului destinaţie, reducând astfel dimensiunea pachetului, dar pentru că fiecare *switch* trebuie sa decidă asupra căii de urmat, aceasta va duce la creşterea complexităţii constructive a fiecărui *switch*.

Rutarea poate duce la blocaje (*deadlocks*) datorită aşteptărilor pentru eliberarea canalelor de comunicaţie.

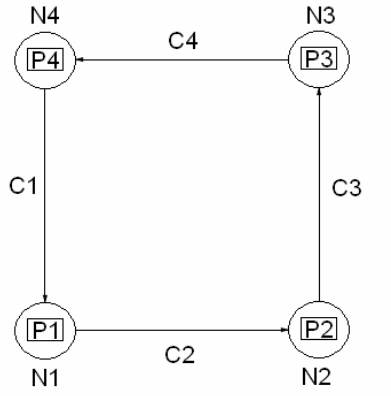


Figura 2. Blocaj datorat aşteptărilor
pentru eliberarea canalelor de comunicaţie

În exemplul din figura 2, pachetul P1 aşteaptă eliberarea canalului C2, dar în acelaşi timp ţine ocupat canalul C1 pentru a nu fi transmis alt pachet care să îl suprascrie în bufferul din nodul N1. La fel, P2 aşteaptă eliberarea lui C3 şi ţine ocupat canalul C2. La fel P3 asteaptă eliberarea lui C4 şi ţine ocupat C3. P4 aşteaptă eliberarea lui C1 şi ţine ocupat canalul C4. Astfel apare o dependenţă ciclică între canale (fig. 3):

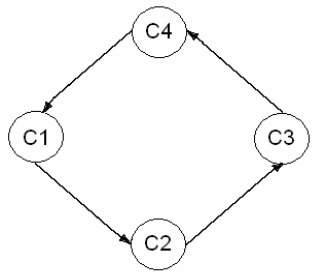


Figura 3. Graful dependenţei dintre canale

Rutarea locală poate fi minimală sau non-minimală. La rutarea minimală, algoritmul de rutare alege întotdeauna calea cea mai scurtă între sursă şi destinaţie. La rutarea non-minimală se poate alege orice cale.

Există două tehnici de rutare locală:

1) rutare deterministă;

2) rutare adaptivă.

1) **Rutarea deterministă**

În acest caz, pachetele trimise de la un nod sursă la un nod destinaţie urmează întotdeauna aceeaşi rută. Ruta este aleasă astfel încât să fie minimală şi să nu ducă la blocaje.

Două exemple pentru rutarea deterministă sunt: rutarea în coordonate XY şi etichetarea intervalelor:

R1) Rutarea în coordonate XY, care se foloseşte pentru reţele de tip *mesh* (grilă).

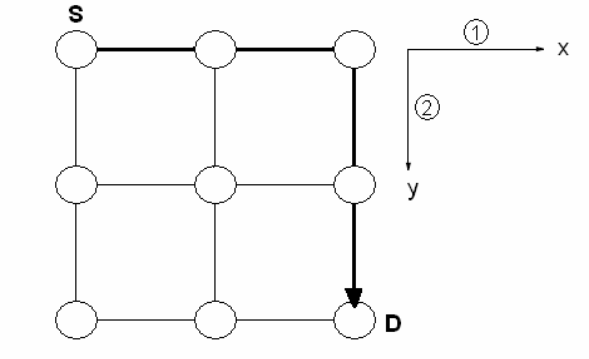


Figura 4. Rutarea în coordonate XY

Un pachet care merge de la sursă la destinaţie e condus mai întâi pe axa X pentru a ajunge la coordonata X a destinaţiei, apoi pe axa Y pentru a ajunge la coordonata Y a destinaţiei. Aici nu există blocaje datorate dependenţelor ciclice. De exemplu, un pachet care circulă pe coordonata Y nu va mai fi trimis pe coordonata X (fig.5):

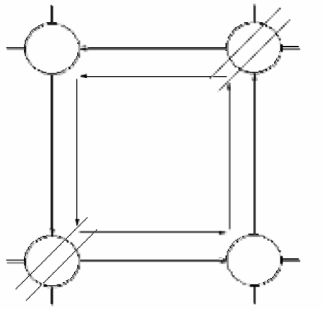


Figura 5. Eliminarea dependenţelor ciclice la rutarea în coordonate XY

R2) Etichetarea intervalelor. Aici, porturile de ieşire ale fiecărui *switch* au asociat un anumit interval care indică nodurile destinaţie posibile pentru acel port:

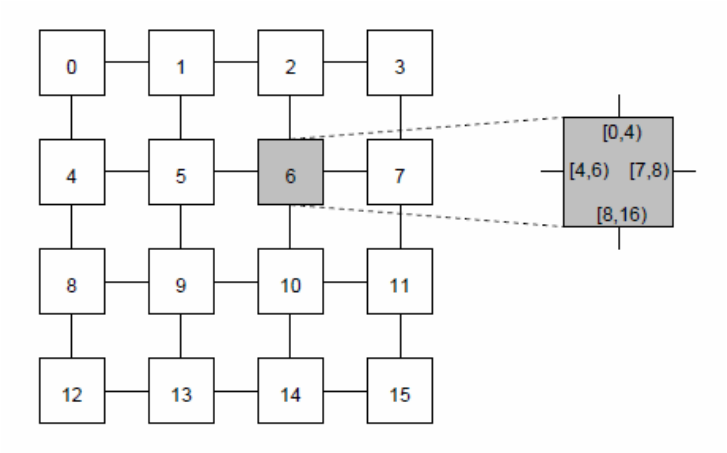


Figura 6. Etichetarea intervalelor

Un pachet care soseşte într-un anumit nod va fi trimis pe portul de ieşire care specifică intervalul ce conţine identificatorul nodului destinaţie. În acest caz se pot alege intervalele fiecărui *switch* astfel încât să nu apară blocaje în rutare.

2) Rutarea adaptivă

În acest caz, ruta se stabileşte în mod dinamic la fiecare *switch* în funcţie de integritatea reţelei şi de canalele alocate deja pentru alte transmisii.

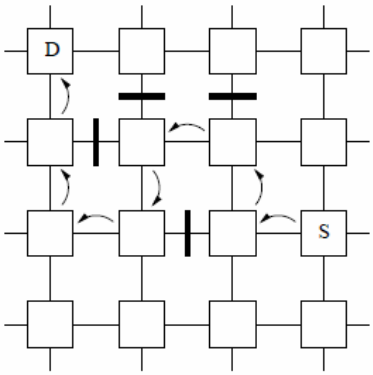


Figura 7. Rutarea adaptivă

Avantaje:
- rutarea adaptivă este mai tolerantă la erori;
- poate să evite blocajele din reţea;
- încărcarea reţelei este mai uniformă decât la rutarea deterministă:

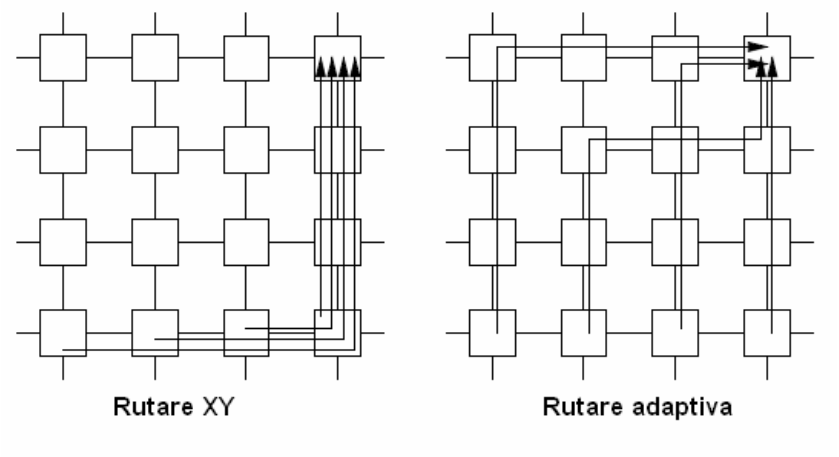


Figura 8. Uniformizarea încărcării reţelei
‘ la rutarea adaptivă faţă de rutarea XY

Dezavantaje:
- rutarea adaptivă necesită *switch*-uri mai inteligente, deci mai scumpe;
- algoritmul de rutare devine mai complicat, datorită necesităţii de a evita blocajele din reţea;

- pachetele care aparţin aceluiaşi mesaj pot să ajungă la destinaţie într-o ordine aleatoare, datorită faptului că ele pot fi rutate pe căi diferite. Astfel, reasamblarea pachetelor la destinaţie va mări durata transmisiei.

Rutarea adaptivă poate fi de 2 feluri:

1) Complet adaptivă, în care nu există constrângeri asupra direcţiilor pe care le poate urma un pachet.
2) Parţial adaptivă, în care pot exista astfel de constrângeri.

Eliminarea blocajelor de rutare:

Există mai multe metode pentru eliminarea blocajelor datorate dependenţelor ciclice. O primă metodă este folosirea rutării deterministe, de exemplu rutarea XY.

O altă metodă este folosirea unei metode parţial adaptivă care derivă din rutarea XY, numită ***West First Routing***, care se foloseşte pentru reţele de tip *mesh*. Aici se transmite prima dată pachetul către vest (dacă e nevoie) apoi se transmite adaptiv la nord, sud sau est.

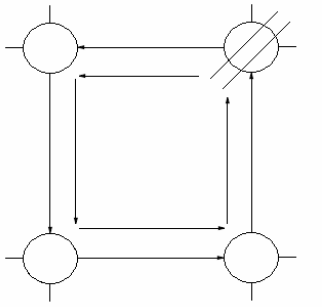


Figura 9. Eliminarea dependenţelor ciclice la rutarea ***West First Routing***

Rutarea ***West First Routing*** este non-minimală pentru a fi capabilă să elimine blocajele.

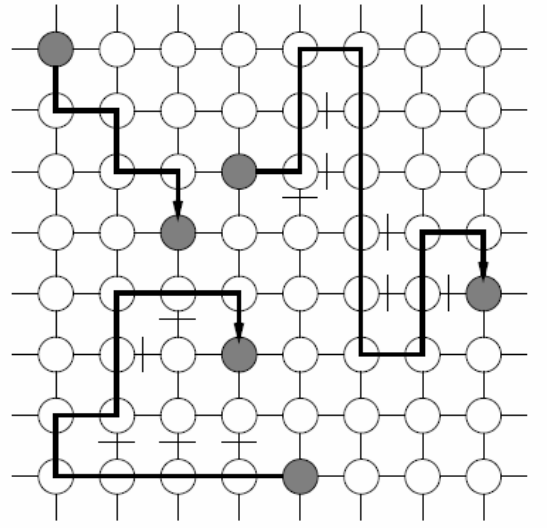


Figura 10. Exemple de rutare ***West First Routing***

O altă metodă utilizată pentru eliminarea dependenţelor ciclice este folosirea canalelor de comunicaţie virtuale. Un canal virtual este o legătură logică între 2 noduri, mai multe canale virtuale putând fi multiplexate pe un acelaşi canal de comunicaţii.

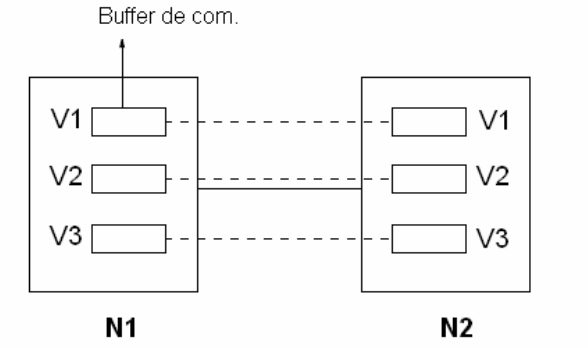


Figura 11. Folosirea canalelor de comunicaţie virtuale

Multiplexarea pe canalul de comunicaţie se face prin utilizarea unui mecanism rotativ prin care se selectează pe rând canalele virtuale pentru transmisie.

Introducerea de canale virtuale poate să evite dependenţele ciclice (fig. 12):

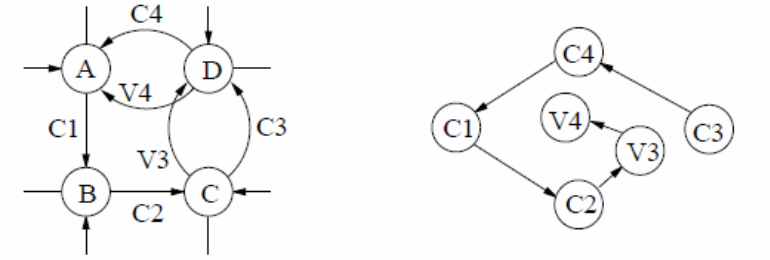


Figura 12. Eliminarea dependenţelor ciclice prin introducerea de canale virtuale

Un alt exemplu de folosire a canalelor virtuale se referă la o reţea de tip *mesh* cu 2 legături virtuale bidirecţionale pe axa Y şi o singură legătură bidirecţională pe axa X. Pe această reţea se construieşte subreţeaua "+X" pentru transmiterea unui pachet la est de sursă, şi subreţeaua "-X" pentru a transmite un pachet la vest de sursă.

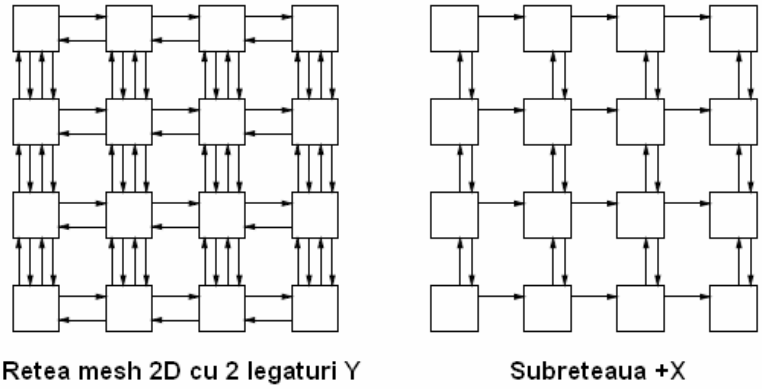


Figura 13. Reţea *mesh* cu 2 legături virtuale pe axa Y şi o legătură virtuală pe axa X

Avantajele folosirii canalelor virtuale:

- elimină dependenţele ciclice;
- se măreşte capacitatea de utilizare a reţelei; astfel, dacă un pachet se blochează pe un canal virtual aşteptând eliberarea unui alt canal de comunicaţie, alte pachete mai pot fi totuşi transmise de-a lungul legăturii fizice respective folosind un alt canal virtual de comunicaţii:

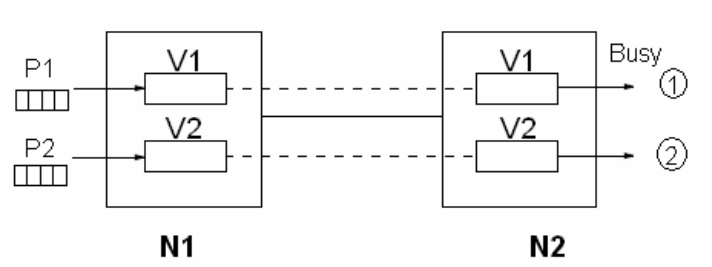


Figura 14. Alegerea unui canal virtual liber pe un canal fizic având două legături virtuale

- canalele virtuale facilitează maparea unei topologii logice peste topologia fizică a reţelei.

Dezavantaje:

- necesită resurse hardware mai complexe, de exemplu: buffere mai mari în noduri, implementarea fizică a unei politici de planificare a canalelor virtuale pe legătura fizică etc. Toate acestea vor duce la creşterea costurilor de implementare;
- canalele virtuale pot creşte latenţa transmisiei pentru că pachetele folosesc partajat lăţimea de bandă a canalelor de comunicaţii.

Metode pentru implementarea comunicaţiei multicast

Într-o reţea de comunicaţie există următoarele tipuri de transmisii:

- transmisie **unicast** - reprezintă transmisia punct la punct între un nod sursă şi un nod destinaţie;
- transmisie **multicast** - de la un nod sursă la mai multe noduri destinaţie;
- transmisie **broadcast** - de la un nod sursă la toate celelalte noduri.

Implementarea comunicaţiei multicast se poate face în 3 moduri:

1) Fără suport hardware special, caz în care se utilizează un software special pentru rutare. *Switch*-urile nu trebuie să fie inteligente, iar pentru transmisia către nodurile destinaţie se foloseşte comunicaţia unicast.

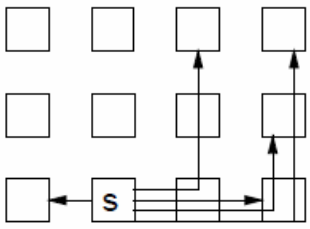


Figura 15. Implementare multicast folosind comunicaţia unicast

2) Se folosesc *switch*-uri care permit replicarea datelor. Astfel *switch*-urile copiază informaţia din pachete în bufferele proprii şi apoi o trimit mai departe la nodurile vecine în funcţie de destinaţie.

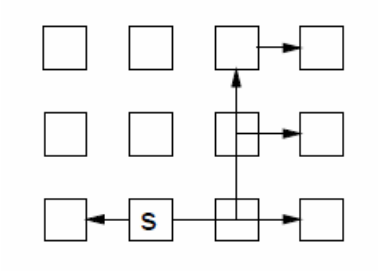


Figura 16. Implementare multicast prin replicarea datelor

3) Se pot folosi *switch*-uri inteligente care permit alegerea unei căi optime prin reţea, astfel încât pachetul să ajungă la toate nodurile destinaţie.

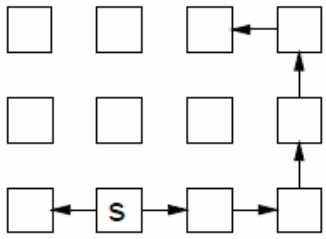


Figura 17. Implementare multicast prin folosirea *switch*-urilor inteligente

MULTICALCULATOARE

Multicalculatoarele sunt arhitecturi MIMD cu memorie distribuită. Ele folosesc metoda de comutare prin pachete pentru schimbul de date şi o reţea directă (punct la punct) pentru conectarea procesoarelor între ele.

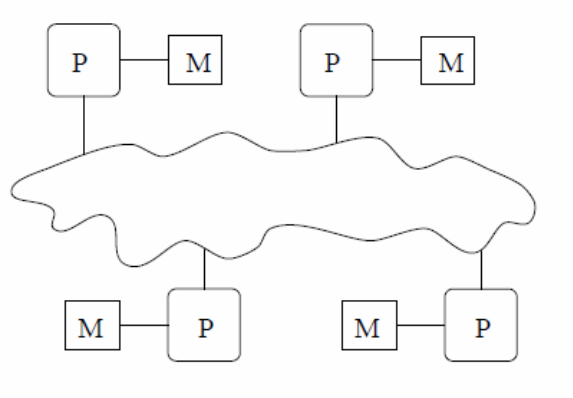


Figura 1. Structura unui multicalculator

Într-un multicalculator nu există un spaţiu global de adrese, un procesor putând accesa doar memoria privată (locală). Comunicarea şi sincronizarea între procesoare se realizează prin intermediul transmisiei de mesaje; utilizatorii sunt cei care trebuie să gestioneze comunicaţia prin codul aplicaţiei.

La modul global, multicalculatoarele sunt arhitecturi fără partajare în care toate resursele sistem (memorie, discuri, etc.) sunt distribuite şi accesibile doar de către procesorul local. În ceea ce priveşte rutarea, nodurile intermediare din reţea trebuie să găsească o rută pentru mesajele transmise la un nod aflat la distanţă. La început, această rută se stabilea prin software de către procesorul nodului respectiv, printr-un mecanism bazat pe întreruperi. În sistemele de astăzi, există un circuit integrat specializat în fiecare nod de comunicaţie, care se ocupă de rutarea, comutarea şi transferul pachetelor la nodul destinaţie.

Dezavantajul multicalculatoarelor derivă din nonexistenţa memoriei globale. Cu toate acestea, se poate simula funcţionarea unei memorii globale prin 2 metode:

1) VSM – *Virtual Shared Memory*
2) SVM – *Shared Virtual Memory*

În ambele abordări, programatorul vede o arhitectură cu o memorie globală partajată, accesibilă de către toate procesoarele. Referinţele la această memorie globală sunt rezolvate prin intermediul transmiterii unor mesaje care realizează schimbul de date între procesor şi memoria globală.

1) VSM – *Virtual Shared Memory*

VSM presupune o implementare hardware; memoria partajată global va face parte din memoria virtuală a sistemului, fiind dispusă în vârful acesteia. SO nu va face distincţie între un multicalculator de tip VSM şi o maşină cu memorie partajată.

În această metodă, unitatea folosită pentru partajarea memoriei este blocul cache. Astfel, la accesarea unui bloc de memorie, în cazul unui *cache miss*, controlerul de memorie cache determină dacă acel bloc se găseşte în memoria locală sau nu. Dacă nu, controlerul va trimite un mesaj la nodul *remote* (de la distanţă) care deţine blocul de memorie cerut, pentru a-l aduce prin reţea în nodul local.

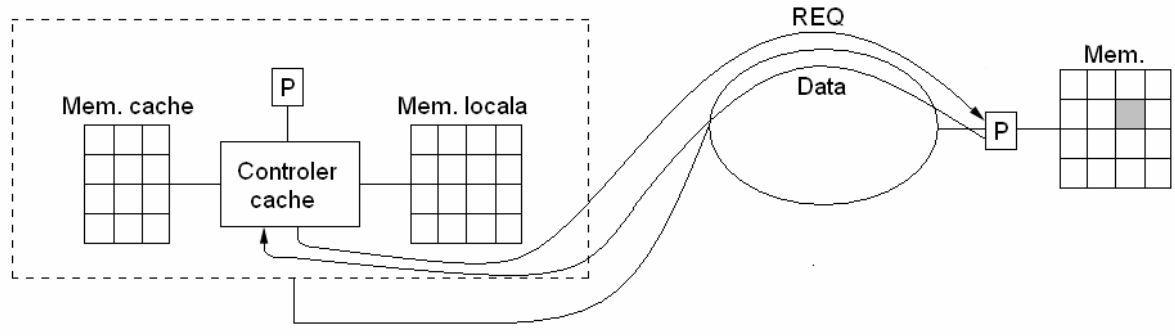


Figura 2. Implementarea VSM

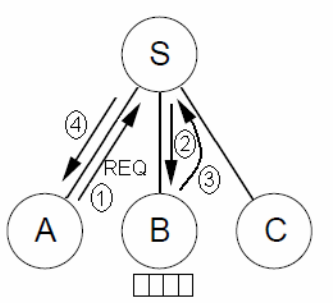
2) SVM - *Shared Virtual Memory*

SVM este o implementare software la nivelul SO, având ca suport hardware unitatea de management a memoriei (MMU) a procesorului. Aici, unitatea de partajare este pagina de memorie a SO. La adresarea unei locaţii de memorie, MMU determină dacă pagina ce conţine acea locaţie se afla în memoria locală sau nu. Într-un calculator obişnuit, dacă pagina de memorie nu se află în memoria locală, atunci va fi încărcată din memoria virtuală situată pe hard-disk. În cazul de faţă, SO va încărca acea pagină din nodul *remote* care o deţine.

Există 4 abordari pentru sistemele VSM, respectiv SVM:

a) **Metoda utilizând un server central**: există un singur nod care poate accesa un anumit bloc de memorie la un moment dat; toate cererile de acces la memorie sunt trimise către un server central. Serverul este cel care cunoaşte locaţia exactă în reţea a oricărui bloc de memorie şi răspunde la cererile venite din partea nodurilor din sistem.

În figura 3 este ilustrată o situaţie în care nodul A formulează o cerere pentru un bloc de memorie aflat pe nodul B:



1) A trimite o cerere către S
2) S trimite cererea la B
3) B răspunde si trimite blocul de memorie către S
4) S trimite răspuns la A

Figura 3. Metoda utilizând un server central

Avantaje: algoritmul este uşor de implementat; se menţine o puternică coerenţă si consistenţă a sistemului.
Dezavantaj: se pot produce gâtuiri în traficul de date al reţelei, datorită supraîncărcării cu cereri a serverului central. Pentru rezolvarea acestei probleme, datele partajate se pot distribui între mai multe servere.

b) **Metoda migraţiei (*Full Migration*)**: nu există un server central; fiecare nod ştie unde anume se află fiecare bloc de memorie, pe baza unei distribuţii statice a adreselor în reţea. La un anumit moment de timp, fiecare bloc de memorie se găseşte într-un anumit nod al reţelei sub controlul unui anumit procesor. Dacă se face o cerere prin reţea pentru o dată conţinută într-un bloc de memorie, procesorul care deţine acel bloc îl transmite mai întâi către nodul care a făcut cererea şi apoi îşi invalidează copia blocului din nodul local.

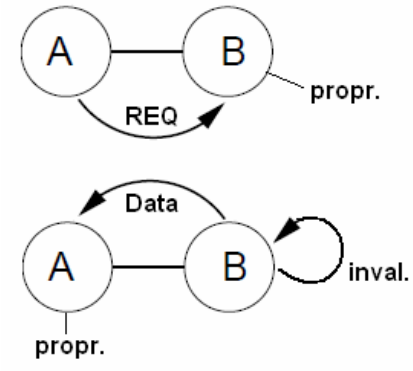


Figura 4. Metoda migraţiei

În întreaga reţea, în orice moment de timp, va exista o singură copie validă pentru fiecare bloc de memorie. Astfel, fiecare bloc de memorie va avea la un moment dat un singur proprietar, adică nodul care deţine copia validă a blocului.

Dacă nodul căruia îi este adresată o cerere pentru un bloc de memorie a trimis anterior acel bloc către alt procesor (nod), atunci el va trimite cererea mai departe către acel nod care a devenit noul proprietar (fig. 5):

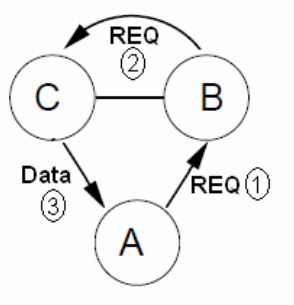
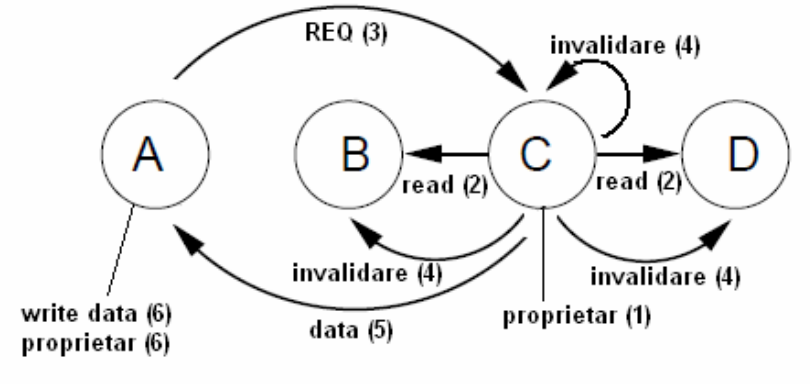


Figura 5. Migrarea cererilor în reţea

Ca urmare, apare dezavantajul că o cerere poate migra mai mult timp în reţea până să ajungă la destinatar.

c) **Metoda *Read Replication***: această metodă încearcă să reducă latenţa de comunicaţie permiţând mai multor noduri să deţină copii *read-only* ale blocurilor de memorie; şi aici fiecare nod al reţelei ştie pe baza unei distribuţii statice a adreselor unde se află un anumit bloc de memorie. Se face distincţie între citirea şi scrierea unei date:
   1) Când un procesor efectuează o citire non-locală, el preia prin copiere blocul de la nodul proprietar.
   2) Dacă se efectuează o scriere, atunci putem avea de exemplu următoarea situaţie:



1) C este proprietarul datelor
2) B,D citesc aceleaşi date
3) A face o cerere pentru scriere
4) C invalidează datele trimise anterior
5) C trimite date la A
6) A scrie datele şi devine noul proprietar

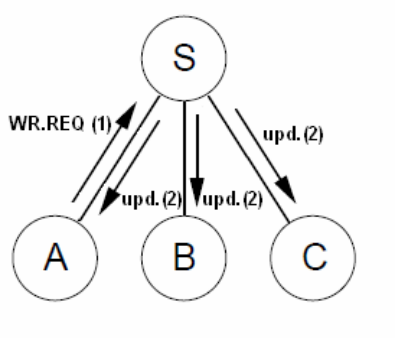
Figura 6. Metoda *Read Replication*

Diferenţa faţă de metoda anterioară (pct. b) este că citirea datelor nu schimbă proprietarul, deci nu măreşte latenţa comunicaţiei. Proprietarul se schimbă doar la scrierea datelor.

d) **Metoda *Full Replication***: se permite mai multor noduri să deţină copii ale blocurilor de memorie atât la citire cât şi la scriere. Toate nodurile cunosc

distribuţia blocurilor de memorie în reţea; citirea unui bloc se face de la orice nod care îl deţine; scrierea unui bloc se face astfel:

1) Se preia de la un proces secvenţiator un număr de secvenţă unic în reţea;
2) Valoarea care se doreşte a fi scrisă în blocul de date, împreună cu numărul de secvenţă, se trimit la toate nodurile care deţin o copie a blocului respectiv (fig. 7):



1) A trimite o cerere de scriere către S
2) S adaugă un număr de secvenţă la această cerere şi trimite un mesaj de actualizare către nodurile A,B şi C

Figura 7. Metoda *Full Replication*

Numărul de secvenţă este necesar pentru cazul în care în reţea circulă mai multe mesaje de actualizare pentru aceeaşi dată.

Câteva exemple de multicalculatoare: - IBM SP2
- Parsytec CC

IBM SP2

IBM SP2 foloseşte procesoare de tip Power 2, utilizând pentru conectarea procesoarelor plăci de conectare ce conţin fiecare o reţea multistagiu cu 2 nivele. Reţeaua este alcătuită din 8 *switch*-uri, fiecare *switch* având 8 legături bidirecţionale:

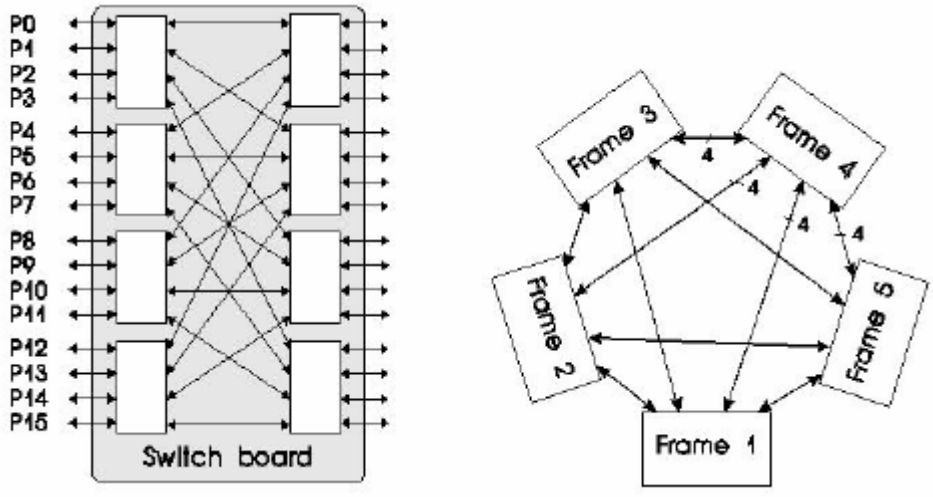


Figura 8. Structura de conectare la calculatorul IBM SP2

În figura următoare este ilustrat un exemplu de conectare cu 12 *switch*-uri:

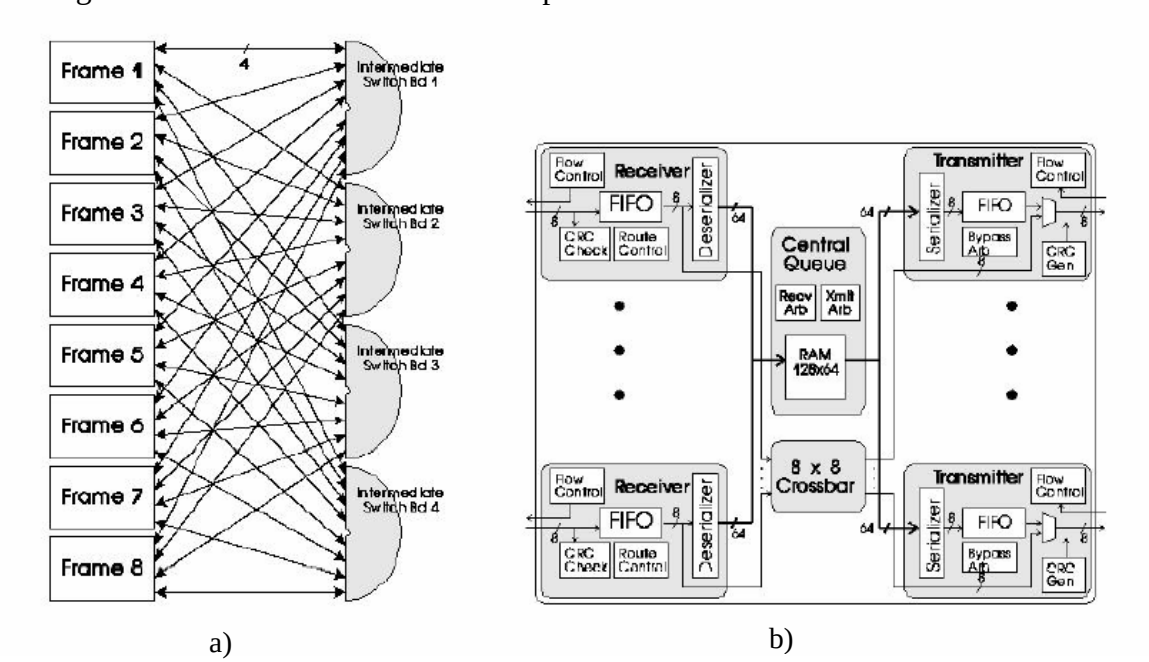


Figura 9. a) Exemplu de conectare cu 12 *switch*-uri; b) Arhitectura unui *switch*

Arhitectura unui *switch* cuprinde 8 receptoare (*receiver*-e) şi 8 transmiţătoare (*transmitter*-e), fiecare *receiver* este conectat la fiecare *transmitter* printr-o reţea *crossbar* (ce conectează orice intrare cu orice ieşire).

Se implementeaza rutarea de tip *wormhole routing*, bazată pe fliturі. Când un flit (o parte dintr-un pachet) ajunge la intrarea unui *receiver*, sunt posibile două situaţii:

a) Dacă *transmitter*-ul dorit este liber, flitul este transmis direct la transmiţător prin reţeaua *crossbar*;
b) Dacă *transmitter*-ul este ocupat, atunci *switch*-ul poate stoca până la 128x8 fliturі în bufferul central; apoi, dacă *transmitter*-ul rămâne ocupat, se blochează linia de intrare a *receiver*-ului.

Parsytec CC

Acest multicalculator realizează un echilibru între eficienţă şi costuri prin combinarea unei reţele multistagiu cu o reţea de tip *mesh* (grilă). Sistemul constă din conectarea unor procesoare de tip Power PC prin intermediul unei reţele numită ***Mesh of Clos*** bazată pe reţeaua multistagiu ***Clos***.

Reţeaua ***Clos*** se bazează pe *switch*-uri având un număr de 2k canale bidirecţionale.

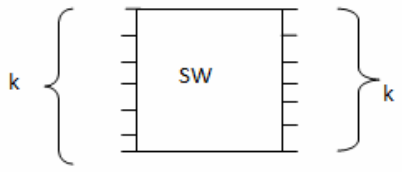


Figura 10. *Switch* având un număr de 2k canale bidirecţionale

O reţea ***Clos*** de dimensiune 1 este alcătuită dintr-un singur *switch* la care se conectează k procesoare:

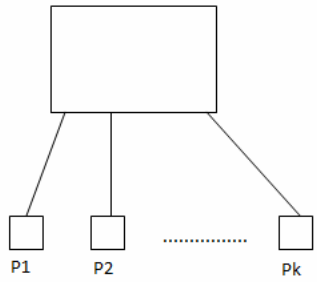


Figura 11. Reţeaua ***Clos*** de dimensiune 1

Reţeaua ***Clos*** de dimensiune h se realizează prin conectarea a k reţele de dimensiune h-1 printr-un număr de k^{h-1} *switch*-uri:

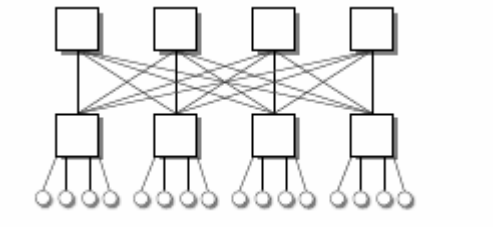


Figura 12. Reţeaua ***Clos*** de dimensiune h=2;
k=4, fiecare switch e conectat la 4 procesoare

Reţeaua ***Mesh of Clos*** se realizează prin înlocuirea unui număr de r stagii din reţeaua *Clos* printr-o reţea de tip *mesh*. Pentru r=1 se înlocuiesc *switch*-urile din reţea cu reţele *mesh*.

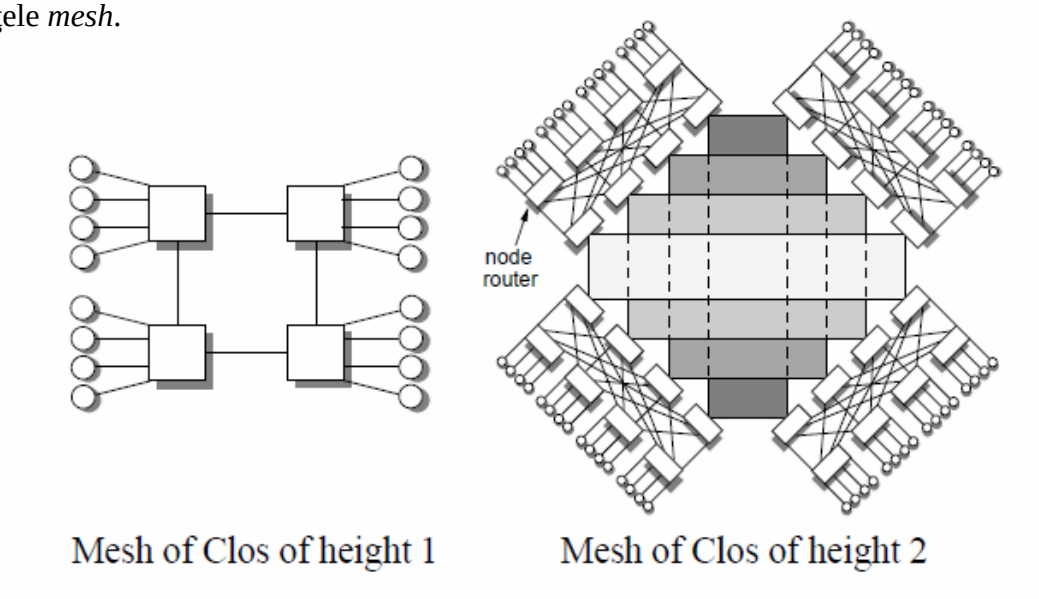


Figura 13. Reţele ***Mesh of Clos*** de dimensiuni h=1 şi h=2

S-a demonstrat că acest tip de conectare este mai bună decât o conectare simplă de tip *mesh* (***Mesh of Clos*** are un număr de *switch*-uri mai mic decât o reţea de tip mesh).

MULTIPROCESOARE

Multiprocesoarele sunt calculatoare paralele care dispun de o singură memorie globală partajată. Procesoarele din aceste sisteme sunt conectate printr-o reţea indirectă sau printr-o combinaţie de reţea indirectă și reţea punct la punct.

Comunicaţia și sincronizarea dintre procesoare este realizată prin intermediul unor locaţii partajate din memoria globală, dar pe aceste maşini se poate emula și interfaţa pe bază de transmisie de mesaje MPI (*Message Passing Interface*) caracteristică multicalculatoarelor.

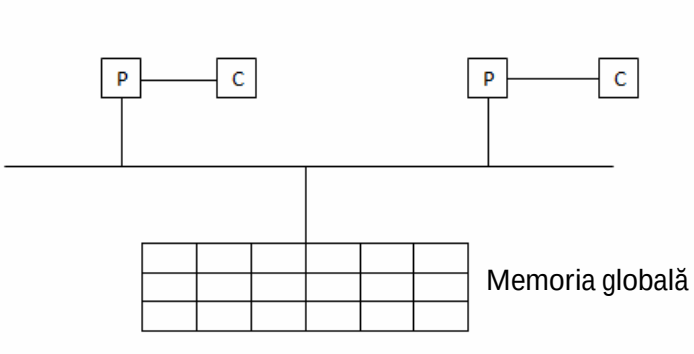


Figura 1. Arhitectura unui sistem multiprocesor

O problemă a acestor calculatoare este scalabilitatea redusă. De exemplu, calculatoarele multiprocesor bazate pe magistrală nu se pot scala la un număr mare de procesoare. Astfel, la un număr de 8-10 procesoare, magistrala intră în saturaţie. O altă problemă care trebuie rezolvată la calculatoarele multiprocesor este menţinerea coerenţei memoriilor cache.

Coerenţa memoriilor cache în calculatoarele multiprocesor cu memorie partajată

Cum fiecare procesor dispune de o memorie cache și fiecare procesor poate accesa o aceeași locaţie din memoria partajată, pot apărea inconsistenţe între diferitele memorii cache care au încărcat aceeași variabilă.

Sursele inconsistenţelor pot fi 3 la număr:

1) Scrierea datelor în zone partajate de memorie.

2) Migrarea proceselor între procesoare.

3) Activităţile de intrare-ieşire.

Exemple:

<u>Situatia 1:</u>

Scrierea datelor într-o zonă partajată:

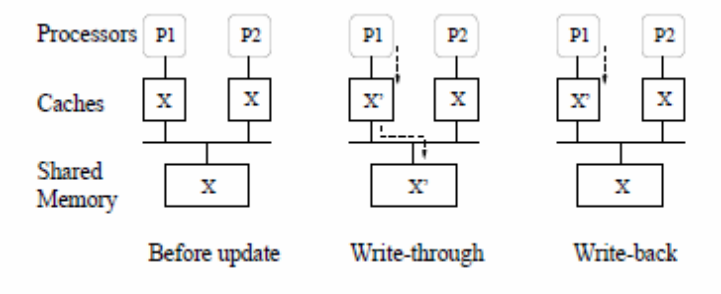


Figura 2. Inconsistenţă la scrierea datelor într-o zonă partajată

Situaţia iniţială: P1 și P2 citesc aceeași memorie X.

Write-through: Memoria cache permite scrierea, valoarea e scrisă imediat în memoria globală. P1 scrie valoarea X' în variabila X după care P2 vrea să citească această valoare, dar P2 va citi X, nu valoarea actualizata (X') => inconsistenţă.

Write-back: P1 scrie valoarea X' în X dar nu este trecută in memoria globală. P2 citește valoarea X => inconsistenţă.

Situaţia 2:

Migrarea proceselor.

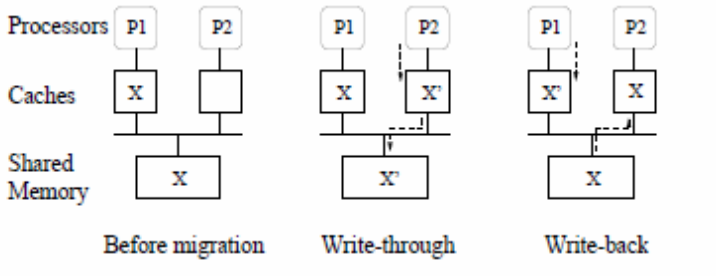


Figura 3. Inconsistenţă în cazul migrării proceselor

Situaţia iniţială: P1 citește valoarea X din memoria globală.

Write-through: Un proces din P2 scrie valoarea X' în variabila X după care procesul se mută în P1 și va citi din nou variabila X => inconsistenţă.

Write-back: Un proces din P1 scrie valoarea X' în variabila X apoi același proces migreaza în procesorul P2 care citește valoarea X din memoria globală => inconsistenţă.

Situatia 3:

Activităţile de intratre-ieșire.

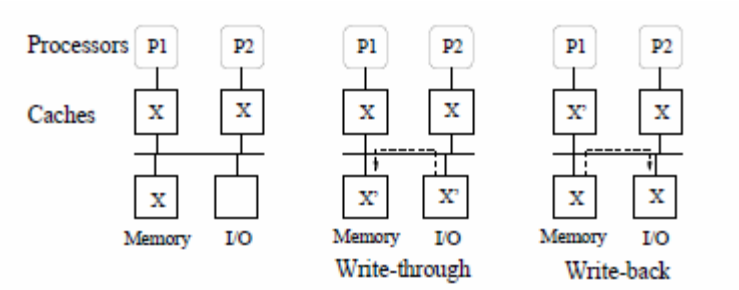


Figura 4. Inconsistenţa datelor în cazul activităţilor de intrare-ieşire

Situaţia iniţială: S-a adăugat un dispozitiv de intrare-iesire pe magistrala sistem, P1 și P2 citesc aceeași valoare X din memoria globală.

Write-through: Presupunem că perifericul vrea sa scrie valoarea X' în memoria principală în locul variabilei X. Scrierea se face fără intervenţia procesoarelor, de exemplu prin DMA. Apoi P1 sau P2 vor sa citeasca variabila X din memorie => inconsistenţă.

Write-back: P1 scrie X' în variabila X după care perifericul vrea să citească valoarea X din memorie => inconsistenţă.

Pentru rezolvarea ultimei probleme (la memoriile *write-back*) se poate muta interfaţa de intrare-ieşire lângă procesor astfel încât perifericul să poată accesa memoria globală doar prin memoria cache:

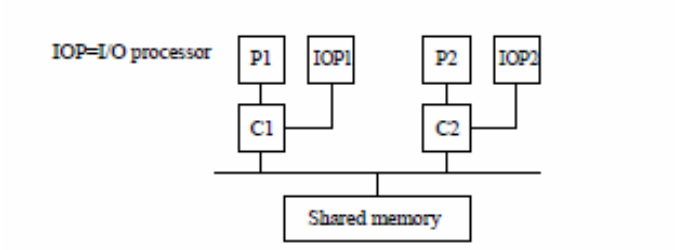


Figura 5. Rezolvarea inconsistenţei prin reamplasarea interfeţelor de intrare-ieşire

O altă soluţie tot pentru această problemă este ca procesorul să nu citească locaţiile de memorie din spaţiul de intrare-ieşire (alocat periferice lor) din memoria cache, ci direct din memoria globală.

Pentru menţinerea coerenţei memoriilor cache se folosesc așa-numitele protocoale de coerenţă. Există **două tipuri de protocoale pentru menţinerea coerenţei**:

1) *Write invalidate* (scriere cu invalidare)

2) *Write update* (actualizare la scriere)

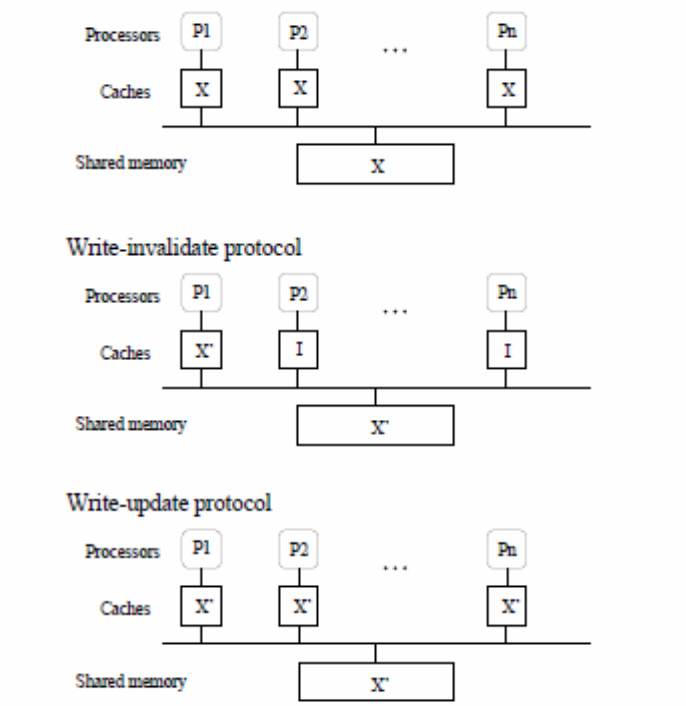


Figura 6. Protocoale pentru menţinerea coerenţei datelor

Protocolul *Write-Invalidate*: dacă P1 scrie valoarea X' în variabila X din memorie atunci se va trimite un mesaj de invalidare tuturor celorlalte memorii cache pentru blocul care conţine variabila X.

Protocolul *Write-Update*: dacă P1 scrie valoarea X' în variabila X atunci se vor trimite mesaje de actualizare cu valoarea X' la toate celelalte memorii cache. Protocolul *Write-Update* necesită mai multă laţime de bandă din partea reţelei, pentru că această metodă este echivalentă cu o operaţie de broadcast pentru actualizarea datelor.

Metoda *Write-Invalidate* suferă în schimb de un fenomen numit *false-sharing*; în acest caz se pot face anumite invalidări ale datelor care nu sunt necesare în program:

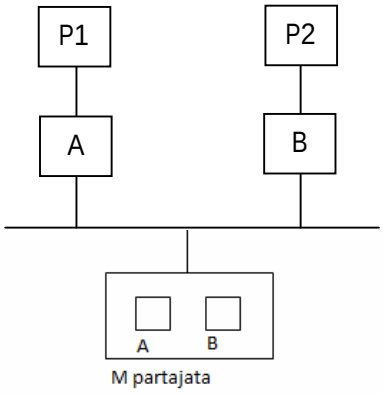


Figura 7. Situaţie în care poate apărea fenomenul de *false-sharing*

În exemplul de mai sus, A şi B sunt două variabile ce există în același bloc de memorie. Dacă P1 modifică variabila A, atunci se invalidează și variabila B (invalidarea realizându-se la nivel de bloc), deși această din urmă invalidare nu este necesară.

1. Arhitectura UMA (*Uniform Memory Access*)

Această arhitectură se utilizează în sisteme multiprocesor tradiţionale care folosesc o reţea de comunicaţie indirectă, de obicei o magistrală. În sistemele UMA, modalitatea de conectare și timpul de acces la memoria globală partajată sunt aceleași pentru toate procesoarele.

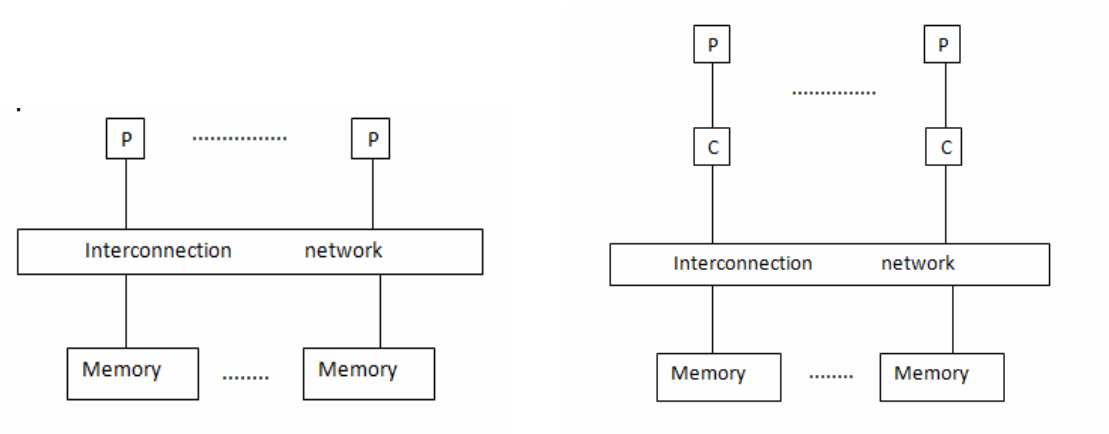


Figura 8. Arhitectura UMA

O clasă specială de calculatoare care folosește arhitectura UMA o reprezintă multiprocesoarele simetrice SMP (*Simetric Multiprocessor*).

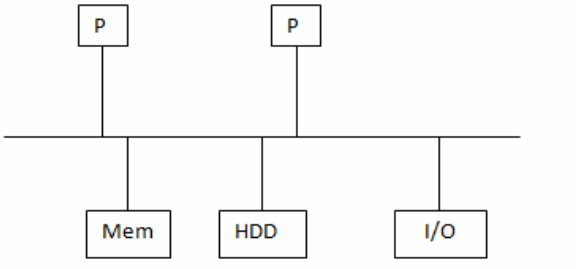


Figura 9. Arhitectura unui sistem SMP

În cazul unui sistem SMP, toate resursele sistem (memorii, hard disk-uri, dispozitive de I/O) sunt accesibile tuturor procesoarelor într-o manieră uniformă.

Câteva probleme ale calculatoarelor cu arhitectura UMA:

1) Aceste arhitecturi sunt foarte greu scalabile.

Arhitectura bazată pe magistrală suferă fenomenul de saturare dacă se conectează prea multe procesoare. Folosirea unei reţele *cross-bar* limitează scalabilitatea datorită costurilor. Folosirea reţelelor multistagiu este scumpă, apărând o latenţă mare a acesului la memoriile partajate atunci când se măreşte numărul de stagii.

Soluţii:

a) Reducerea traficului pe magistrală prin folosirea memoriilor cache, dar și aici crescând numărul de procesoare crește numărul de evenimente *cache-miss* (crește numărul de blocuri cache invalide), evenimente a căror rezolvare va conduce din nou la saturarea magistralei.

b) Gruparea unui număr mic de procesoare împreună cu o singură memorie partajată într-un cluster și conectarea mai multor clustere printr-o reţea directă (arhitectura NUMA).

2) Poate apărea blocarea memoriei atunci când mai multe procesoare vor să acceseze memoria simultan.

Soluţie:

Pentru rezolvarea problemei putem împărţi memoria în mai multe blocuri (sau bancuri de memorie) astfel încât fiecare banc să se acceseze într-un anumit interval de adresă. De exemplu putem avea o memorie cu două bancuri, primul banc fiind accesat la adrese pare iar al doilea la adrese impare.

Dar şi în cazul impărţirii memoriei în bancuri apar probleme dacă mai multe procesoare vor să acceseze simultan același banc de memorie.

Exemplu: Presupunem că avem mai multe procesoare care sunt conectate printr-o reţea multistagiu la mai multe bancuri de memorie (fig.10). În momentul în care mai multe procesoare accesează simultan un banc de memorie (B1) apare congestia sau saturarea reţelei:

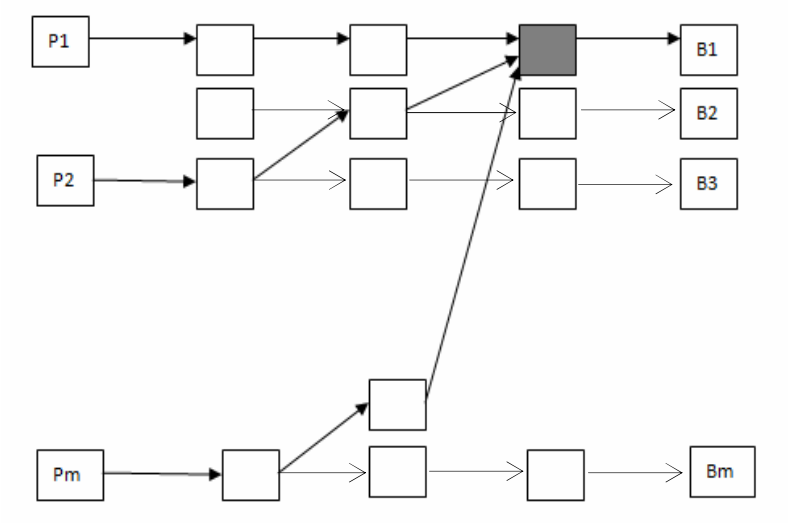


Figura 10. Congestia unei reţele multistagiu

În exemplul de mai sus, *switch*-ul conectat la bancul B1 reprezintă un punct de congestie.

Ca o soluţie pentru acestă problemă putem folosi combinarea mesajelor:

De exemplu se foloseşte o operaţie atomică numită *Fetch and Add* (o operaţie atomică nu poate fi divizată sau întreruptă de nici un alt proces). Operaţia *Fetch and Add* (x,e) este echivalentă cu înlocuirea variabilei x din memorie prin valoarea x+e.

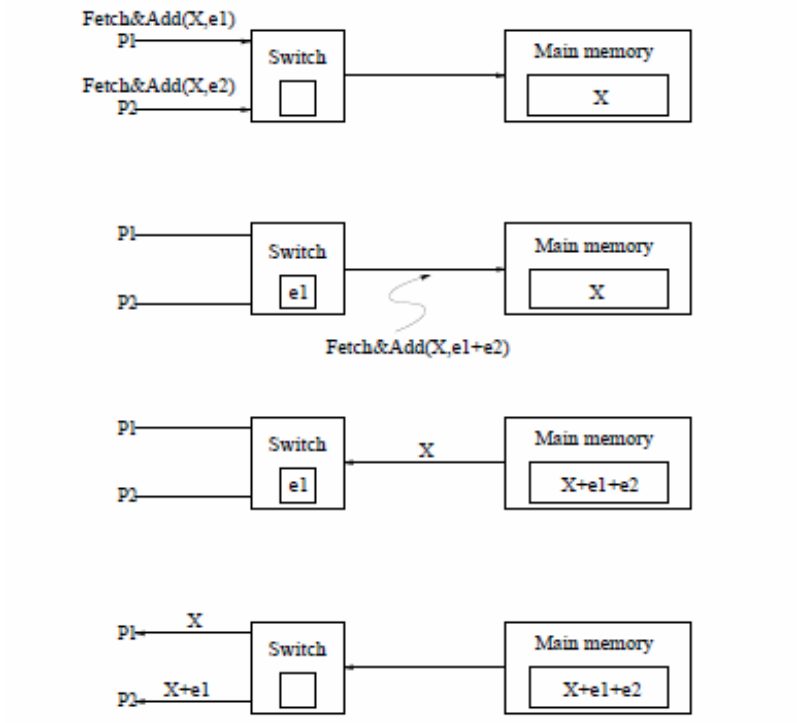


Figura 11. Accesul simultan a două procesoare la o locaţie de memorie prin operaţii *Fetch and Add*

Pasul 1: P1 și P2 vor să acceseze simultan variabila X prin operaţii atomice Fetch and Add.

Pasul 2: *Switch*-ul memorează valoarea de adunare e1 a primei operaţii, apoi combină cele două operaţii *Fetch and Add* într-una singură având ca valoare de adunare e1+e2, şi trimite această operaţie ca un singur mesaj memoriei principale.

Pasul 3: Valoarea din memorie se actualizează şi se trimite înapoi către *switch* un mesaj cu valoarea iniţială a lui X.

Pasul 4: *Switch*-ul trimite valoarea iniţială către cele două procesoare; astfel P1 va primi valoarea X în timp ce P2 va primi valoarea X+e1 (presupunem ca P1 și P2 nu au memorii cache).

Acest exemplu ilustrează avantajul metodei de combinare a mesajelor: memoria este citită sau scrisă o singură dată la accesul simultan a mai multor procesoare, astfel încât nu va mai apărea fenomenul de saturare a reţelei.

2. Arhitectura NUMA (Non-Uniform Memory Access)

Un dezavantaj al arhitecturii UMA este acela că ea duce la realizarea de sisteme greu scalabile; pentru a înlătura acest dezavantaj s-au dezvoltat sistemele NUMA. În cadrul acestei arhitecturi, mai multe procesoare cu acces uniform la o memorie partajată sunt conectate într-un cluster, mai multe clustere fiind conectate printr-o reţea punct la punct scalabilă, ce foloseşte pentru comunicare transmiterea de mesaje.

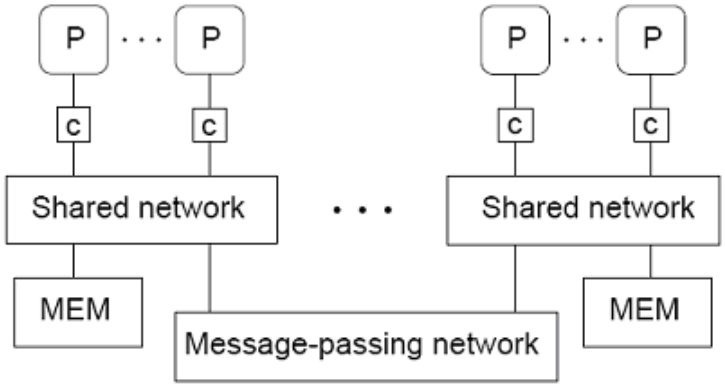


Figura 12. Arhitectura NUMA

Din punct de vedere al programării există un singur spaţiu de memorie globală, astfel că NUMA devine o arhitectură partajată din punct de vedere logic, deși din punct de vedere fizic ea este o arhitectură distribuită.

Când un procesor accesează memoria sistemului, controlerul de memorie cache (sau în alte implementări unitatea de management a memoriei MMU) verifică dacă locaţia adresată se găsește în memoria locală sau într-o memorie aflată la distanţă. În cazul în care se accesează un bloc de memorie aflat la distanţă (*remote*), se trimite un mesaj în reţeaua de clustere pentru accesarea blocului *remote* prin reţea. Acest mod de funcţionare are ca și consecinţă faptul că accesul la memoria locală este rapid, pe când accesul la memoria *remote* este lent (de aici vine și numele arhitecturii NUMA).

Pentru a reduce numărul de accesări ale memoriilor aflate la distanţă, procesoarele aduc în memoria cache proprie blocurile *remote* dorite:

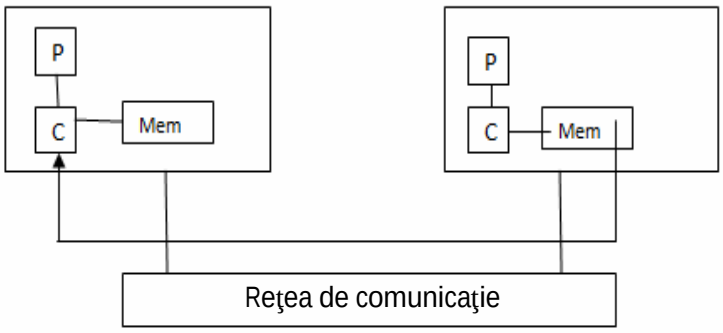


Figura 13. Aducerea în memoria locală a blocurilor *remote*

Dar în această situaţie va trebui rezolvată problema coerenţei memoriilor cache care au citit aceeași variabilă. În acest scop, au fost create sistemele numite CC-NUMA (*Cache Coerent* NUMA), sisteme care folosesc un protocol de tip *Write-Invalidate* pentru menţinerea coerenţei memoriilor cache.

ARHITECTURA CC-NUMA (continuare)

Ascunderea latenţei la accesul memoriei de la distanţă în sistemele CC-NUMA

Arhitectura CC-NUMA bazată pe memorii cache reduce latenţa accesării memoriilor de la distanţă, dar este important să existe metode pentru acoperirea timpilor de aşteptare datoraţi latenţei acestor accese *remote*.

Există 3 metode pentru acoperirea latenţei:

a) Preîncărcarea datelor – prin care se încearcă predicţia datelor necesare şi încărcarea lor în memoria cache înainte ca ele să fie propriu-zis accesate;
b) Metoda *multithreading* – în care se foloseşte un *thread* separat pentru accesul la un bloc de memorie *remote*. Astfel, în timpul aşteptării accesului la memorie se poate comuta pe un alt *thread* care poate fi rulat:

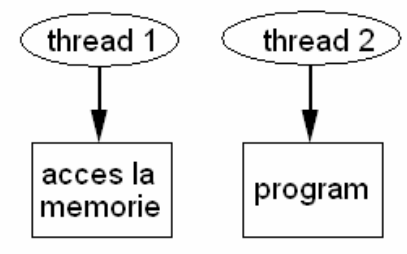


Figura 1. Metoda *multithreading*

c) Folosirea unor modele relaxate de consistenţă a memoriei.

Modele de consistenţă a memoriei:

M1. Modelul nerelaxat de consistenţă secvenţială (SC)

În acest model, toate accesele la memorie sunt atomice (nu pot fi întrerupte) şi au o ordine precisă:

a) un procesor scrie într-un bloc partajat de memorie;
b) toate celelalte procesoare primesc mesaje de invalidare pentru blocul respectiv;
c) un alt procesor scrie, ş.a.m.d.

Conceptul de consistenţă secvenţială se poate implementa printr-o memorie cu un singur port de intrare-ieşire. Procesoarele sunt conectate la memorie prin intermediul unui *switch* (fig.2). Acest *switch* alege la un moment dat un singur procesor care să aibă acces la memorie. Un alt procesor care vrea să acceseze memoria va trebui să aştepte până când procesorul anterior a terminat accesul.

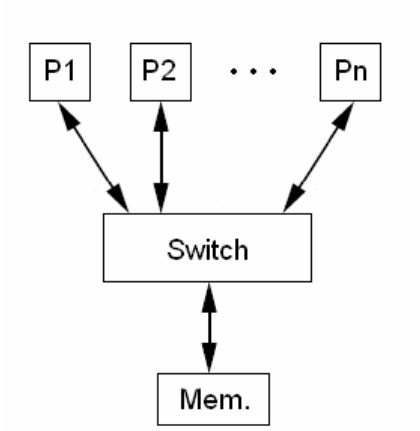


Figura 2. Implementarea modelului de consistenţă secvenţială

Modelul de consistenţă secvenţială nu este optimal datorită stricteţii sale. Procesoarele superscalare nu folosesc modelul de consistenţă secvenţial. Pentru ascunderea latenţei accesului la memorie ele folosesc execuţia *out of order* a instrucţiunilor şi de asemenea scrierea datelor de ieşire în *buffere* de scriere, şi nu scrierea directă în memorie.

M2. Modele de consistenţă relaxată a memoriei (RC)

Există mai multe modele RC:

a) Modelul de consistenţă a procesoarelor – în care operaţii de tip LOAD pot devansa operaţii WRITE în cadrul unui program.
b) Modelul cu ordine de memorare parţială – în care operaţiile LOAD urmate de WRITE pot devansa operaţii WRITE precedente dintr-un program.
c) Modelul de consistenţă slabă – accesele la memorie se pot face fără a respecta ordinea instrucţiunilor din program (pentru aceasta nu trebuie să existe dependenţe de date între instrucţiuni) .

În toate modelele de consistenţă relaxată s-au introdus aşa numitele bariere de sincronizare. O barieră reprezintă o operaţie sincronizată care determină ca toate operaţiile de acces la memorie care au fost lansate anterior să se finalizeze înainte de execuţia operaţiei sincronizate. Plasând barierele în locuri bine determinate din program se obţin avantajele specifice modelului de consistenţă secvenţială, dar cu o performanţă mai bună, pentru că accesele la memorie nu sunt restricţionate între bariere.

Dezavantajele arhitecturii CC-NUMA:

Principalul dezavantaj derivă din faptul că datele *remote* sunt aduse în memoriile cache ale procesoarelor, memorii care au dimensiuni reduse. Astfel, dacă este necesar accesul la blocuri de date *remote* de dimensiuni mari, aceste blocuri nu vor putea fi memorate în întregime în memoriile cache, şi ca urmare ele vor trebui încărcate secvenţial de la distanţă, timpul de acces crescând foarte mult.

Ca soluţii posibile la această situaţie putem avea:

1) Creşterea dimensiunii memoriilor cache. În acest caz va creşte însă şi latenţa acceselor la memoriile locale (pentru că memoriile cache devin mai lente), şi cresc de-asemenea costurile de implementare.
2) Implementarea la nivelul sistemului de operare a unui mecanism de migrare a paginilor de memorie. Astfel, atunci când o dată *remote* e accesată frecvent, sistemul de operare poate decide replicarea la citire sau migrarea la scriere a paginii în care se găseşte data cerută. Astfel pagina va fi adusă în memoria locală a procesorului care a cerut date. Această soluţie are ca dezavantaj creşterea complexităţii metodei şi funcţionarea la o granularitate mare (la nivel de pagină).

ARHITECTURA COMA (CACHE ONLY MEMORY ARHITECTURE)

Arhitectura COMA seamănă cu arhitectura NUMA, diferenţa fiind că memoriile locale DRAM funcţionează precum memoriile cache (realizându-se o mapare de tip direct sau asociativ).

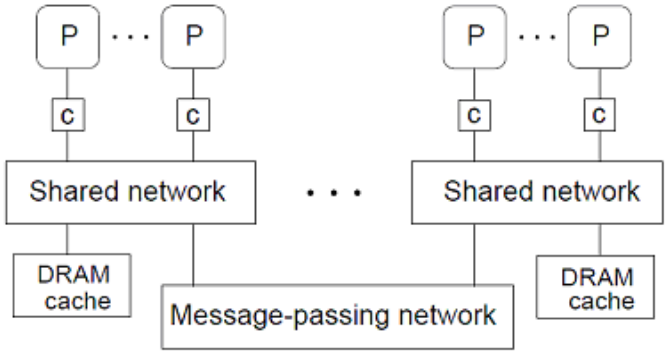


Figura 3. Arhitectura COMA

Datele încărcate de la distanţă (*remote*) sunt aduse în memoria locală şi sunt păstrate în corespondenţă cu adresele lor din sistem. Blocurile de date nu au o locaţie fixă într-o anumită memorie locală, ci ele pot să migreze liber în sistem.

Arhitectura COMA utilizează o reţea cu transmitere de mesaje având o structură arborescentă. Fiecare nod sau *switch* din această reţea conţine un director care indică datele conţinute în arborele subiacent (al cărui părinte este nodul respectiv) (figura 4). Cum datele nu au o locaţie fixă în sistem, ele trebuie căutate în memoriile locale distribuite folosind structura de directoare a reţelei.

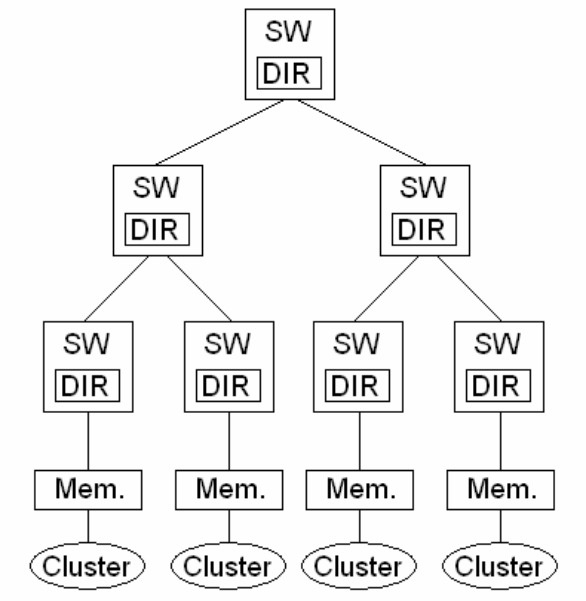


Figura 4. Structura arborescentă a reţelei de clustere (COMA)

Există şi posibilitatea combinării mesajelor: dacă un *switch* primeşte două cereri pentru o aceeaşi dată de la un subarbore atunci cele două cereri se vor combina într-una singură, care va fi înaintată *switch*-ului părinte. Când datele cerute sunt disponibile, ele vor fi trimise înapoi către cele două *switch*-uri care au trimis cererile iniţiale.

Pentru menţinerea coerenţei memoriilor cache, arhitectura COMA foloseşte protocolul *write-invalidate*, în care coerenţa se realizează la granularitatea fină a blocurilor din memoria locală. Acest protocol de coerenţă a arhitecturii COMA este însă mai greu de implementat decât la CC-NUMA.

De exemplu, o situaţie de care trebuie să se ţină seamă este următoarea: dacă într-o memorie locală există o ultimă copie a unui bloc de date din sistem şi dacă acel bloc urmează să fie eliberat din memorie ca urmare a algoritmului de înlocuire de blocuri din memoria cache, atunci acel bloc va trebui mutat în altă memorie locală a sistemului.

Avantajul arhitecturii COMA constă în memoriile locale cache de dimensiuni mari capabile să memoreze cantităţi mari de date de la distanţă. Ca şi cerinţe constructive, la memoriile locale cache este necesar să se adauge memorii suplimentare atât pentru *tag*-urile cache cât şi pentru informaţiile de stare ale blocurilor cache.

Comparaţie între arhitectura COMA şi arhitectura CC-NUMA:

Avantajul arhitecturii COMA faţă de CC-NUMA: arhitectura este mai flexibilă şi suportă migrarea blocurilor de date. Aceasta se face la nivel hardware la o granularitate fină, reducându-se fenomenul de *false sharing*.

Dezavantajele arhitecturii COMA sunt următoarele:

a) Arhitectura este mai costisitoare şi mai greu de implementat;
b) Protocolul de coerenţă este şi el dificil de implementat (trebuie să se ţină seama ca ultimele copii a blocurilor de date să nu fie şterse din sistem);
c) Accesele la distanţă sunt mai lente decât la CC-NUMA din cauza structurii arborescente a reţelei de comunicaţii.

Performanţele celor două arhitecturi depind şi de aplicaţii: dacă e necesar accesul la blocuri mari de date de la distanţă, atunci arhitectura COMA va avea performanţe mai bune, în schimb dacă există o rată mare a invalidărilor blocurilor de date cache (datorită scrierilor repetate) atunci arhitectura COMA va avea o latenţă mai mare a acceselor *remote*, deci performanţe mai slabe, arhitectura CC-NUMA fiind mai potrivită.

ARHITECTURA S-COMA (SIMPLE COMA)

Pentru a reduce complexitatea şi costurile arhitecturii COMA, a fost propusă o variantă a acestei arhitecturi numită SIMPLE COMA. În această arhitectură, alocarea blocurilor în memoria locală de tip cache se face la granularitatea sistemului de operare, adică la nivelul paginilor.

Asemănător cu arhitectura SVM (*Shared Virtual Memory*), şi aici sistemul de operare, prin unitatea de management a memoriei (MMU), determină dacă pagina în care se află data cerută se găseşte sau nu în memoria locală. Dacă da, avem un eveniment *cache hit*, dacă nu, avem un eveniment *cache miss*. Cum MMU este cea care determină prezenţa unei pagini în memoria cache, nu mai e nevoie de memorii suplimentare pentru tagurile asociate blocurilor de memorie.

Spre deosebire de SVM, arhitectura S-COMA gestionează coerenţa blocurilor de date prin hardware la o granularitate fină la dimensiunea blocurilor cache. Astfel este redus şi aici fenomenul de *false sharing*. O altă consecinţă este că în procesul de migrare a datelor se transferă blocuri cache de dimensiuni mici, şi nu pagini (fig. 5).

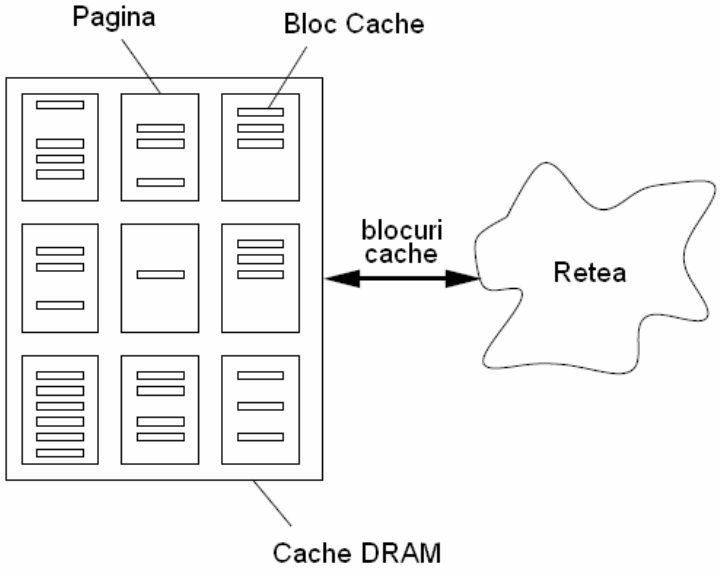


Figura 5. Gestionarea blocurilor de memorie la S-COMA

Dacă la un moment dat există un eveniment *cache miss* (pagina corespunzătoare nu se găseşte în memorie) atunci:

1) se alocă o nouă pagină în memoria locală;
2) blocul de date cerut e încărcat dintr-o altă memorie (*remote*).

O consecinţă a acestui mod de funcţionare este că o pagină din memorie poate fi la un moment dat umplută doar parţial cu blocuri de date. Astfel, la un eveniment *cache hit* (pagina căutată se află în memorie), trebuie să se verifice şi dacă blocul de date cerut există şi este valid în acea pagină.

Arhitectura S-COMA poate implementa memorii cache locale complet asociative în care sistemul de operare poate încărca orice pagină la orice adresă din memorie.

Dezavantaje ale arhitecturii S-COMA:

1) Evenimentele *cache miss* sunt tratate mai lent decât la arhitectura COMA din cauza tratării la nivelul sistemului de operare şi nu prin hardware.
2) Fenomenul de *false replacement*: mecanismul de înlocuire a paginilor în memorie poate duce la înlocuirea unei pagini complet umplute cu blocuri de memorie printr-o pagină ce conţine un singur bloc ce este necesar la un moment dat, reducându-se astfel eficienţa lucrului cu date *remote*.

ARHITECTURA R-NUMA (REACTIVE NUMA)

Această arhitectură poate comuta dinamic între arhitectura CC-NUMA (când blocurile *remote* sunt aduse în memoriile cache ale procesoarelor) şi arhitectura S-COMA (când blocurile de date *remote* sunt aduse în memoriile locale de dimensiuni mari).

COERENŢA MEMORIILOR CACHE ÎN SISTEME MULTIPROCESOR

Toate sistemele cu arhitectură multiprocesor folosesc protocoale pentru menţinerea coerenţei memoriilor cache. Protocoalele de coerenţă se bazează pe existenţa unor stări asociate blocurilor de memorie cache şi a unor tranziţii între stări. Aceste tranziţii între stările unui bloc de memorie depind atât de acţiunile procesorului local, cât şi de acţiunile altor procesoare ce deţin copii ale acelui bloc de memorie.

De exemplu, un bloc de memorie (**b**) situat în memoria cache a unui procesor (P1) va trece în starea invalid dacă un alt procesor (P2) scrie în copia aceluiaşi bloc de memorie pe care a făcut-o în propria memorie cache (fig. 1):

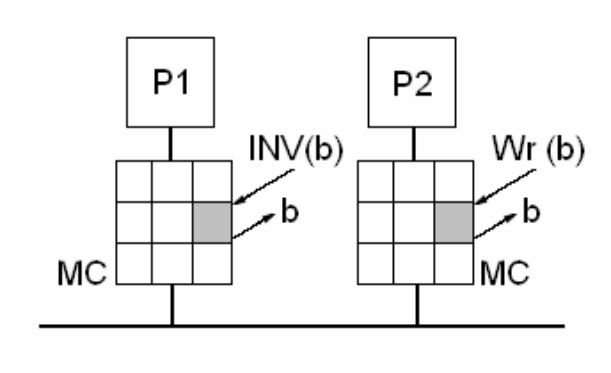


Figura 1. Invalidarea unui bloc de memorie a cărui copie a fost modificată

Din cele de mai sus rezultă necesitatea de a ţine evidenţa tuturor cópiilor care există pentru fiecare bloc de memorie cache din sistem.

Protocoalele de coerenţă în sistemele multiprocesor se împart în două clase:

1) Protocol cu urmărirea tranzacţiilor pe magistrală (*Snoopy Bus*)
2) Protocol bazat pe director

1) **Protocolul *Snoopy Bus***

Acesta se aplică mai ales în cazul sistemelor UMA bazate pe magistrală. În cadrul acestui protocol, controlerul memoriei cache urmăreşte tranzacţiile pe magistrală şi verifică dacă un alt procesor accesează un bloc de memorie care este prezent şi în memoria sa locală. Dacă există un astfel de procesor şi dacă accesul este unul de scriere în memorie, atunci controlerul cache va invalida blocul din memoria locală. Protocolul *Snoopy Bus* poate să fie atât de tip *write-invalidate* cât şi de tip *write-update*.

2) **Protocolul bazat pe director**

Aici există o memorie director care păstrează localizarea tuturor cópiilor din sistem pentru toate blocurile din memoriile cache. Acest protocol se foloseşte pentru menţinerea coerenţei memoriilor cache în sistemele CC-NUMA şi respectiv COMA. Aceste arhitecturi se bazează pe reţele cu transmitere de mesaje în mod *non-broadcast*. În aceste arhitecturi, protocolul *Snoopy Bus* (care necesită urmărirea întregului trafic din reţea) nu ar fi potrivit. De obicei protocoalele bazate pe director sunt de tipul *write-invalidate*.

Există mai multe metode de implementare pentru memoria director:

a) ***Implementarea Full-Map***. Aici pentru fiecare bloc dintr-o memorie cache se păstrează localizarea cópiilor blocului respectiv din toate memoriile cache ale sistemului. Astfel, pentru fiecare bloc **X** din memoria cache se păstrează un vector de N biţi, unde N e egal cu numărul de procesoare (şi de memorii cache) din sistem; în acest vector, bitul $\mathbf{b_i}$ indică dacă o copie a blocului respectiv se găseşte în memoria cache de indice **i** (fig. 2).

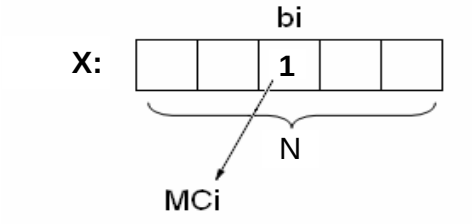


Figura 2. Vectorul din memoria director corespunzător blocului **X**

În acest caz, dacă un procesor scrie în blocul X, atunci se vor invalida toate cópiile blocului X din sistem (fig. 3a-3c):

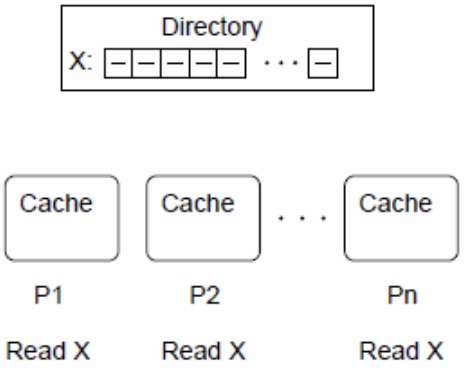


Figura 3a. Procesoarele P1, P2, Pn citesc blocul X în memoria cache proprie

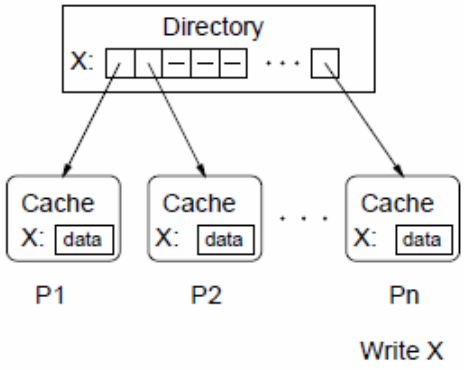


Figura 3b. Se setează biţii corespunzători din vectorul blocului X din memoria director; apoi procesorul Pn scrie în blocul X

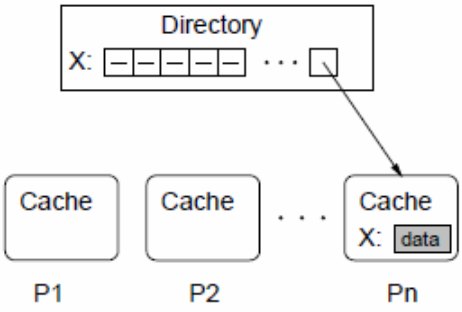


Figura 3c. Cópiile blocului X din P1 şi P2 se invalidează prin resetarea biţilor din memoria director

Un dezavantaj al acestei metode este dat de mărimea memoriei director auxiliare, dimensiune care are ordinul O(N^2).

De exemplu, dacă avem N memorii cache în sistem, şi presupunem că fiecare memorie cache are un număr de P blocuri, rezultă un număr total de N x P blocuri de memorie cache; acestea corespund cu N x P vectori în memoria director, rezultând dimensiunea totală a memoriei director: (N x P) x N = (N^2) x P.

b) O implementare mai restrictivă a memoriei director este cea numită ***Limited Map*** (director cu dimensiune limitată). În acest caz poate exista doar un număr limitat de cópii ale unui bloc de memorie în întregul sistem:

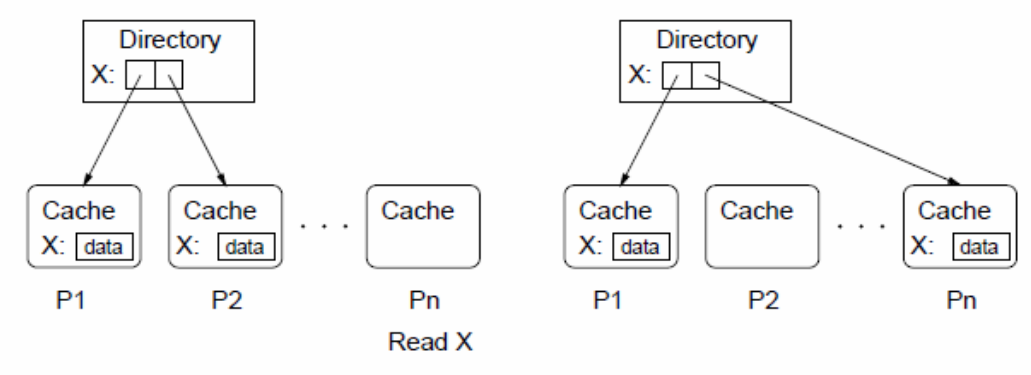


Figura 4. Implementarea ***Limited Map*** a memoriei director

În figura 4 se arată un exemplu în care se permite existenţa a doar două cópii ale fiecărui bloc de memorie în sistem. Avantajul acestei metode este că scade dimensiunea memoriei director; dezavantajul este că se limitează paralelismul.

c) ***Metoda cu director înlănţuit (Chained Directory)***

În cadrul acestei metode, o intrare în memoria director corespunzătoare unui bloc cache conţine un pointer la o listă de cópii ale blocului respectiv. Când un procesor recepţionează un bloc de memorie pentru a-l citi, se pune o referinţă la acel procesor în capul listei memoriei director, iar vechea referinţă din director e stocată în memoria cache a procesorului care a citit blocul. Astfel se creează o listă înlănţuită a tuturor cópiilor unui bloc din sistem (fig. 5).

În acest caz, acţiunea de a invalida cópiile în bloc de memorie se reduce la o operaţie de parcurgere a listei înlănţuite. Dezavantajul metodei este că la înlocuirea unui bloc dintr-o memorie cache, referinţa la acest bloc trebuie îndepărtată din lista în care este inclusă, crescând astfel complexitatea operaţiei.

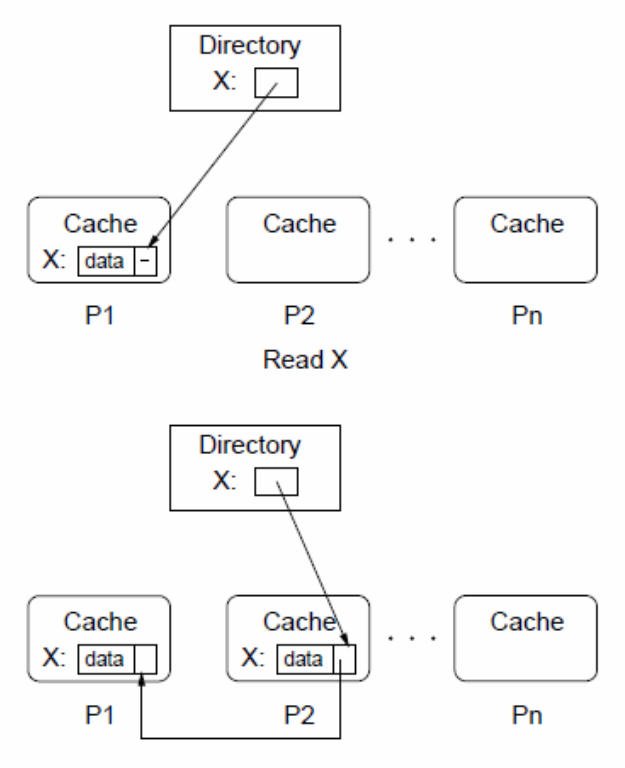
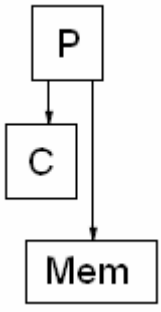


Figura 5. Metoda cu director înlănţuit

STĂRI ŞI TRANZIŢII ÎN PROTOCOALELE DE COERENŢĂ

Protocoalele de coerenţă pentru memoriile cache se bazează pe stări şi tranziţii între stări. De exemplu, procesul de invalidare a unui bloc de memorie cache presupune tranziţia între două stări: valid şi invalid. Dacă un bloc se află în starea invalid, atunci procesorul nu va mai putea utiliza datele din acel bloc.

În continuare se prezintă două exemple de protocoale de coerenţă: pentru memoriile ***Write Through***, respectiv ***Write Back***, împreună cu diagramele de stări.



1) Protocol de coerenţă pentru memoriile de tip **Write Through Cache**.

Fiecare bloc din memoria cache se poate afla în doar două stări: valid sau invalid. Un bloc se consideră invalid şi în momentul în care el este înlocuit din memoria cache. Se definesc trei acţiuni pentru blocul respectiv: citire bloc (**R** -*Read*), scriere bloc (**W** -*Write*) şi înlocuire (**Z** -*Replace*). Aceste trei acţiuni se pot efectua de către memoria cache curentă, de indice i, fiind notate: R(i), W(i), Z(i), sau aceste acţiuni se pot efectua în memoria cache de indice j, notate prin R(j), W(j), Z(j).

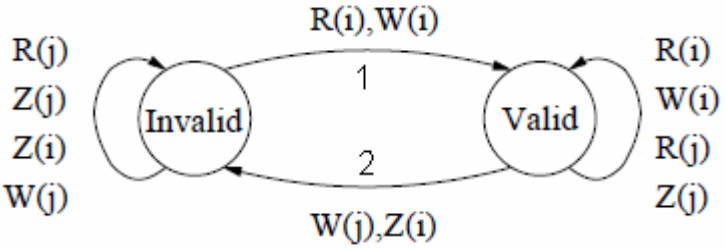
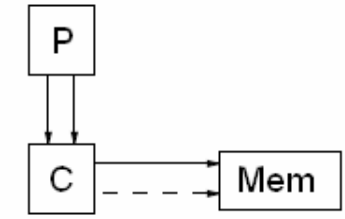


Figura 6. Diagrama de stări şi tranziţii pentru memoria ***Write Through***

În diagrama de mai sus, tranziţiile între stările valid-invalid se realizează în următoarele situaţii:

1) Dacă blocul este invalid, blocul devine valid prin citire R(i) - deoarece el este adus în memoria locală din locaţia sursă, sau prin scriere W(i) - când el este actualizat în propria memorie cache.

2) Dacă blocul este valid, la scrierea blocului de către alt procesor în altă memorie cache W(j), blocul curent se invalidează. De asemenea, blocul se invalidează şi la înlocuirea sa din memoria cache (Z(i)).



2) Protocol pentru memoria de tip **Write Back** cache.

Aici există trei stări în care se poate afla un bloc dintr-o memorie cache: invalid (**INV)**, partajat (**S** -*Shared*) sau modificat (**M** -*Modified*).

Diagrama de stări şi tranziţii pentru memoria ***Write Back*** este ilustrată în figura 7:

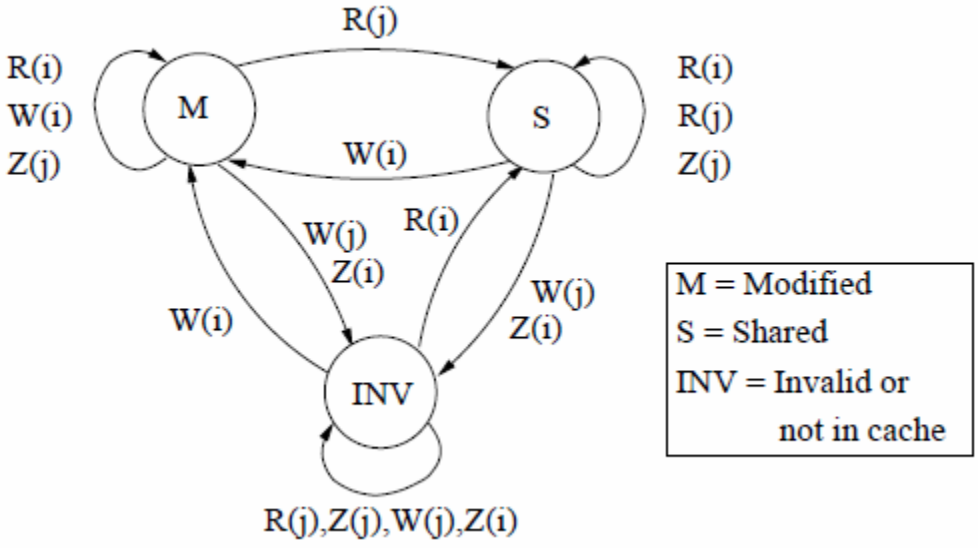


Figura 7. Diagrama de stări şi tranziţii pentru memoria ***Write Back***

În figura 8 sunt ilustrate câteva situaţii posibile în care se poate găsi un bloc de memorie cache (de tipul ***Write Back***).

În figura 8 (a) este prezentată situaţia iniţială, în care avem două module de memorie cache Ci şi Cj (în stânga, respectiv dreapta), şi memoria principală (în centru). Să considerăm un bloc X din memoria centrală. La început, el nu este încărcat în memoriile cache; el apare ca *Invalid* în ambele memorii Ci şi Cj.

În figura 8 (b) memoriile cache citesc blocul X din memoria principală. În ambele memorii cache, blocul X devine *Shared*, adică partajat; copiile multiple ale aceluiaşi bloc de memorie vor fi întotdeauna în starea *Shared*.

În situaţia din figura 8 (c), procesorul asociat cu memoria cache Ci scrie în blocul X din Ci; ca urmare, blocul devine *Modified* (modificat) în Ci, iar în toate celelalte memorii Cj care deţineau copia partajată a blocului X, starea acestui bloc devine *Invalid*. Astfel, la orice moment de timp, doar o singură copie a unui bloc de memorie din sistem se poate găsi în starea *Modified*. Acesta va fi scris în memoria principală în momentul în care va fi necesară eliminarea lui din memoria cache.

Dacă apoi o altă memorie Cj citeşte blocul X (situaţie ilustrată în figura 8 (d) şi detaliată în paragraful următor), blocul din memoria Ci aflat în starea *Modified* devine *Shared*, fiind totodată scris şi în memoria principală.

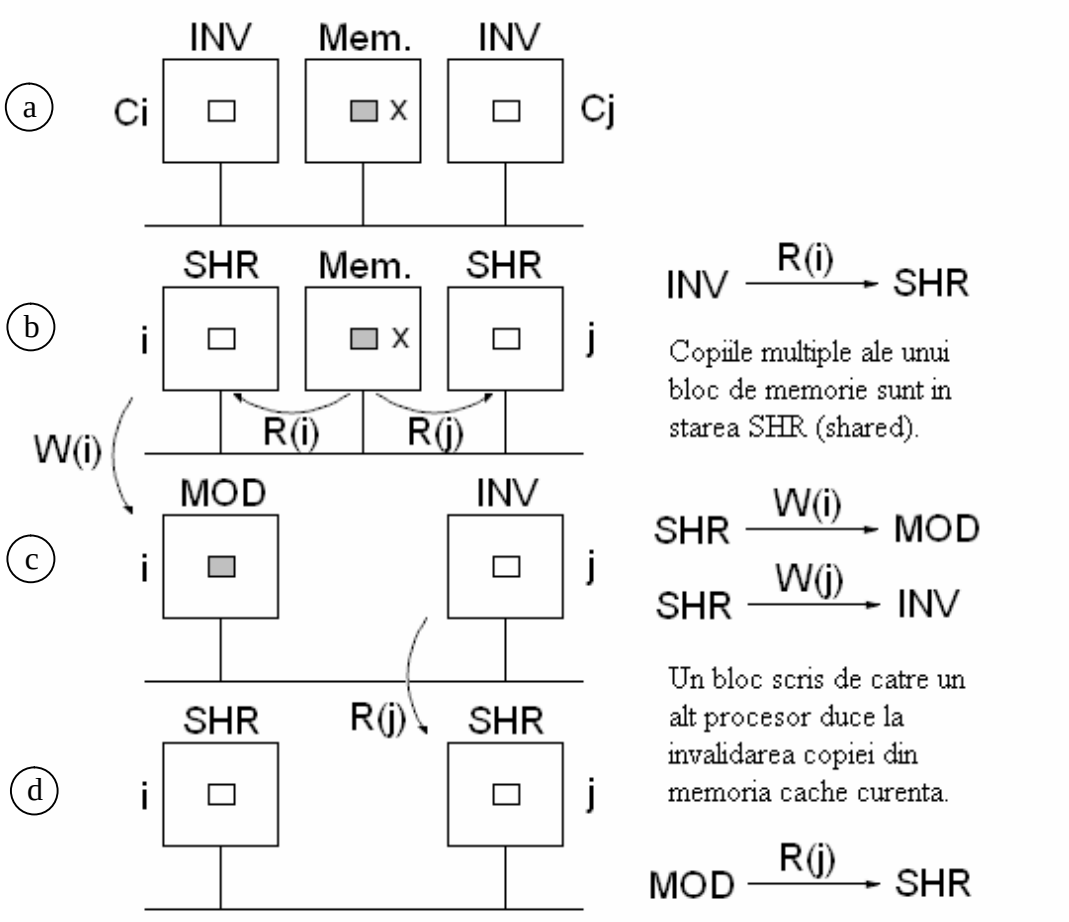


Figura 8. Stări şi tranziţii pentru un bloc de memorie din categoria ***Write Back***

Comparaţie între memoriile *Write Back* şi *Write Through*

Configuraţia ***Write Through*** cache are avantajul că memoria principală a sistemului este întotdeauna consistentă. Astfel, dacă apare un eveniment *cache miss* (blocul căutat nu se regăseşte în memoria cache), memoria cache poate să încarce imediat blocul dorit din memoria principală.

În cazul memoriei ***Write Back***, memoria cache nu poate încărca blocul direct din memoria sistem pentru că el poate exista în starea *Modified* într-o altă memorie cache. Această situaţie se poate rezolva în 2 moduri:

Modul 1. a) Tranzacţia în curs de derulare cu memoria principală se opreşte;
b) Memoria cache care deţine copia modificată scrie blocul actualizat în memoria principală şi modifică starea blocului în *Shared*;
c) Tranzacţia cu memoria principală se reia, obţinându-se copia actualizată a blocului cerut.

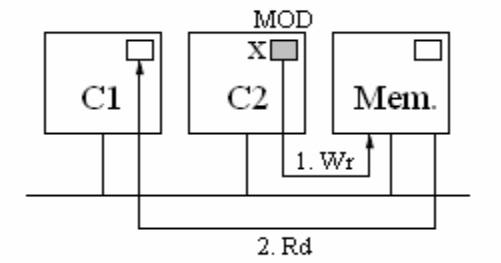


Figura 9. Prima modalitate de citire a unui bloc aflat în starea *Modified*. Memoria cache C2 scrie mai întâi blocul în memoria principală

Modul 2. a) Tranzacţia cu memoria principală se redirecţionează către memoria cache care deţine copia modificată a blocului;
b) Memoria cache de la care s-a citit scrie blocul în memoria principală şi îi modifică starea în *Shared*.

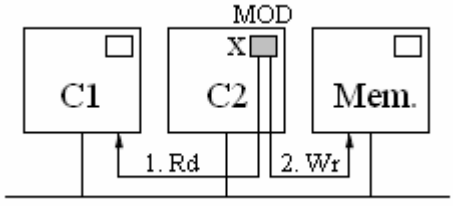


Figura 10. A doua modalitate de citire a unui bloc aflat în starea *Modified*. Memoria cache C1 citeşte direct din memoria în care se găseşte blocul modificat

Prima metodă creşte latenţa evenimentelor *cache miss*. A doua metodă are o complexitate mai mare la implementare.

Dezavantajul memoriilor ***Write Through*** îl constituie faptul că ele cauzează un trafic de reţea suplimentar datorită scrierilor frecvente din memoria cache în memoria principală. În plus, un protocol de coerenţă pentru memoriile ***Write Through*** ce utilizează două stări: valid / invalid, mai are dezavantajul că pentru fiecare scriere a unui bloc de către o memorie cache trebuie invalidate toate copiile acelui bloc din sistem; în schimb, la memoriile ***Write Back,*** dacă blocul se găseşte în starea *Modified*, atunci va exista o singură copie actualizată în sistem care trebuie invalidată.