



Arhitectura Sistemelor de Calcul – Curs 4



Computer Science
& Engineering
Department

Universitatea Politehnica Bucuresti
Facultatea de Automatica si Calculatoare

cs.pub.ro

curs.cs.pub.ro



Cuprins

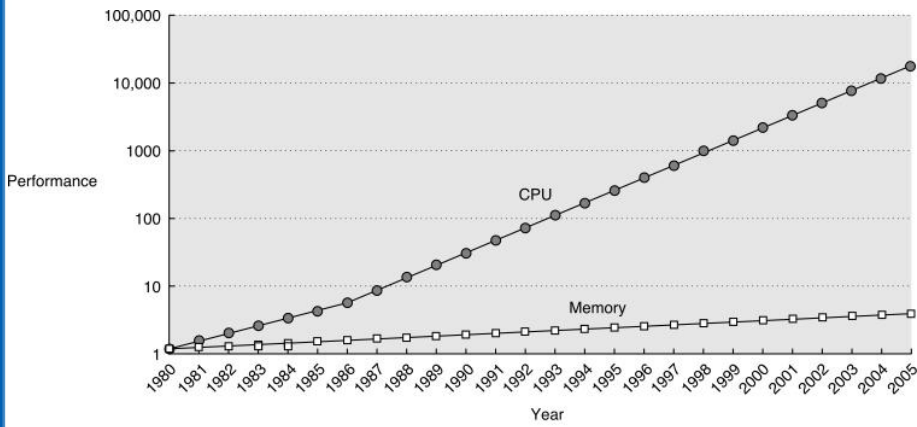
2

- Ierarhia de memorii – Bottleneck-ul SC
- Localitatea datelor
- Cache – design, implementari si exemple
- Imbunatatirea performantelor memoriei



Este Memoria o Problema?

3



© 2003 Elsevier Science (USA). All rights reserved.

©Henri Casanova, U of Hawaii



Ierarhia de Memorii

4

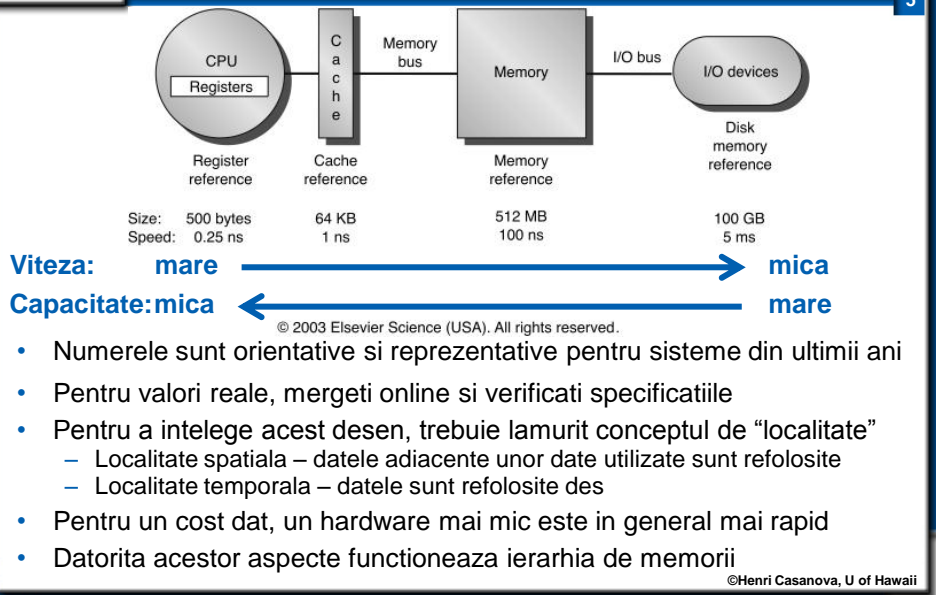
- Deoarece nu putem realiza memorii suficient de rapide(ieftin) – am dezvoltat **ierarhia de memorii**:
 - L1 Cache (pe chip)
 - L2 Cache
 - L3 Cache (uneori)
 - Memoria Principala
 - Discurile fixe

©Henri Casanova, U of Hawaii



Ierarhia de Memorii

5



Cuprins

6

- Ierarhia de memorii – Bottleneck-ul SC
- Localitatea datelor
- Cache – design, implementari si exemple
- Imbunatatirea performantelor memoriei



Ce este Localitatea?

7

- Programele accesează date din memorie apropiate între ele prin instrucțiuni apropiate între ele în **instruction stream**
 - **Instruction stream** = secvența de instrucțiuni de la începutul până la sfârșitul programului
- **Localitatea spațială**: când se accesează o locație anume de memorie, este foarte probabil ca următorul acces să fie în apropierea acestei locații în memorie
 - Ar fi astfel ideală aducerea “aproape” de procesor, a **blocurilor contigue** de date pentru ca instrucțiunile următoare să le găsească disponibile
- **Localitatea temporală**: când se accesează o locație anume de memorie, este foarte probabil ca această locație să mai fie accesată din nou
 - Ținerea **datelor accesate de curând**, “aproape” de procesor, pentru ca instrucțiunile următoare să le găsească disponibile

©Henri Casanova, U of Hawaii



Exemplu de Localitate

8

- Iată o secvență de adrese accesate de către procesor:
 - 1,2,1200,1,1200,3,4,5,6,1200,7,8,9,10
- Această secvență are o **localitate temporală bună**, deoarece locația @1200 este accesată de trei ori din 14 referințe
 - Poate că @1200 este un contor ce este actualizat
- Această secvență are și o **localitate spațială bună**, deoarece locațiile [1,2], [3,4,5,6] și [7,8,9,10] sunt accesate în ordine
 - Poate că în aceste locații se află elementele unui vector ce este accesat în ordine
- Secvența prezentată poate exploata ierarhia de memorii prezentată

©Henri Casanova, U of Hawaii



Exemplu de Localitate

9

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

- **Localitate de Date**

- Consideram secventa de locatii de memorie: `&sum, &i, &i, &n, &a[0], &sum, &sum, &i, &i, &i, &n, &a[1], &sum, &sum, &i, &i, &i, &n, &a[2], &sum, &sum, ..., &a[n-1], &sum, &sum, &sum, v`
- `sum, i, si n` pot fi localizati in registrii deoarece au o **localitate temporala foarte buna**
- Accesul la vectorul `a` are insa o **localitate spatiala foarte buna** → cache

- **Localitate de Instructiuni**

- Toate instructiunile din bucla sunt accesate in mod repetat si in secventa, si astfel au atat **localitate spatiala cat si temporala buna**

©Henri Casanova, U of Hawaii



Localitate si Ierarhia de Memorii

10

- Localitatea este exploatata de catre *キャッシング*
- Cand o locatie de memorie este accesata de catre procesor, se verifica urmatoarele:
 - Se afla in cache-ul L1?
 - Daca nu, se afla in cache-ul L2?
 - (Daca nu, se afla in cache-ul L3?)
 - Daca nu, se afla in memoria principala?
 - Daca nu, se afla pe memoria virtuala (discuri fizice)?
- Acest proces are loc deoarece nu putem construi memorii suficiente de mari si rapide la un pret rezonabil
- **Daca vom identifica multe din locatiile de memorie cautate in cache-ul L1, vom avea "iluzia" unei memorii mari si rapide, pentru o fractiune din cost**

©Henri Casanova, U of Hawaii



Cuprins

11

- Ierarhia de memorii – Bottleneck-ul SC
- Localitatea datelor
- Cache – design, implementari si exemple
- Imbunatatirea performantelor memoriei



Terminologie Cache

12

- Cache-ul reprezinta locatia in care procesorul **poate** gasi date cautate **mai aproape** de el decat in memoria principala a sistemului
- Un **cache hit** este momentul in care datele au fost gasite intr-un nivel de Cache
- Un **cache miss** este momentul in care datele nu au fost gasite
- Un exemplu de functionare:
 - L1 cache **miss**, L2 cache **miss**, L3 cache **hit**
- Cand se inregistreaza un **miss**, un **bloc** este adus in Cache
 - Un bloc este un set de date de dimensiune fixa
 - Blocul contine celulele cerute, si aditional alte celule ce speram ca vor fi folosite de alte instructiuni (localitate spatiala)
- Un cache miss determina un overhead semnificativ:
 - Definitiv
 - **lat** = latenta memoriei (in secunde)
 - **bw** = latimea de banda (in bytes/sec)
 - **S** = dimensiunea blocului in bytes
 - Un **cache miss** duce la un overhead de $lat + S/bw$ secunde ©Henri Casanova, U of Hawaii



Terminologie Cache

13

- Cache-ul este compus din seturi de blocuri (**block frames**)
 - Fiecare set poate contine unul sau mai multe blocuri (depinde de implementare)
- **Hit time** este timpul de accesare al Cache-ului
- **Miss penalty** este timpul necesar mutarii datelor pe diferite nivele de Cache catre procesor
- **Hit Ratio** este procentul din timp in care datele sunt gasite in Cache
- **Miss Ratio** este $1 - \text{Hit Ratio}$
- **Instruction Cache** este Cache-ul ce contine instructiuni (cod)
- **Data Cache** este Cache-ul ce contine date
- **Unified Cache** este Cache-ul ce contine atat date cat si instructiuni

©Henri Casanova, U of Hawaii



Memoria Virtuala

14

- Daca sistemul de calcul permite memoriei principale sa fie utilizata ca un cache pentru discurile fixe, atunci acesta suporta **memoria virtuala**
- In acest caz, blocurile memoriei sunt numite **pagini**
- In orice moment, o pagina este fie in memorie, fie pe discul fix
- Cand procesorul acceseaza o adresa dintr-o pagina ce nu se afla in memorie, se genereaza un **page fault** similar unui **cache miss**
- Un **page fault** este extrem de costisitor (latenta **lat** e mare si largimea de banda **bw** este mica) si procesorul executa in mod normal alte taskuri in acest timp
 - Aceste page fault-uri sunt realizate de catre sistemele de operare
 - Puteti da exemple de operatii similare realizate de sistemul de operare?

©Henri Casanova, U of Hawaii



Caracteristici Tipice ale Memoriilor

15

	Dimensiune	Latenta (ns)	Largime de banda (MB/sec)	Gestionat de
Registre	< 1KB	0.25-0.5	20,000-100,000	Compiler
Cache	<16MB	0.5 (on-chip) -25 (off-chip)	5000 - 10,000	Hardware
Memoria Principala	< 16GB	80-250	1000-5000	O/S
Discurile fixe	> 100GB	5,000,000	20-150	O/S

©Henri Casanova, U of Hawaii



Probleme in Designul Cache-ului

16

- Ce dimensiune trebuie sa aiba un bloc de Cache?
 - Mare: miss penalty ridicat
 - Mica: oportunitate scazuta de exploatare a localitatii spatiale
 - Trebuie determinat dupa un benchmarking riguros si numeroase simulari
 - Trebuie sa fie eficient pentru aplicatii “tipice”
- Patru intrebari necesare in designul unui cache (L1)
 - Unde trebuie sa fie scrisa un bloc in cache? – **block placement**
 - Cum se gaseste o linie in cache? – **block identification**
 - Ce linie trebuie sa fie inlocuita in cazul unui cache miss? – **block replacement**
 - Ce se intampla cand se scrie in cache? – **write strategy**
- Limitari – design-ul trebuie sa fie simplu, astfel incat cache-ul sa fie rapid!

©Henri Casanova, U of Hawaii



Q1 – Block Placement

17

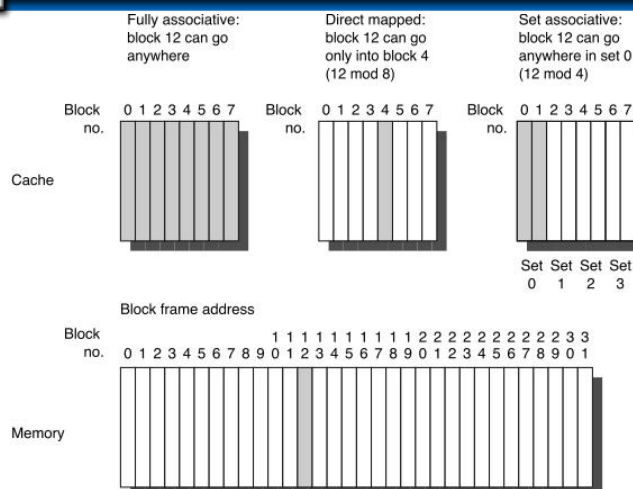
- Daca un bloc poate fi pozitionat doar intr-o singura pozitie in cache: **mapare directa**
- Daca un bloc poate fi asezat oriunde: **mapare total asociativa**
- Daca un bloc poate fi asezat intr-un subset de pozitii: **mapare set asociativa**
 - Varianta pentru n blocuri in fiecare subset: **mapare n set asociativa**
 - Astfel, maparea directa este in fapt o mapare 1 set asociativa

©Henri Casanova, U of Hawaii



Block Placement

18



© 2003 Elsevier Science (USA). All rights reserved.

©Henri Casanova, U of Hawaii



Trade-off – Compromisuri

19

- Maparea n-way set asociativa devine din ce in ce mai dificila si costisitoare de implementat pentru un n mare
 - Majoritatea cache-urilor sunt 1, 2 sau cel mult 4 set asociative
- Cu cat n-ul este mai mare, cu atat este mai mica probabilitatea de aparitie a fenomenului de **thrashing**
 - Momentul in care **doua** (sau mai multe) regiuni de memorie sunt accesate in **mod repetat** si pot incepea **impreuna** in **acelasi** bloc din cache

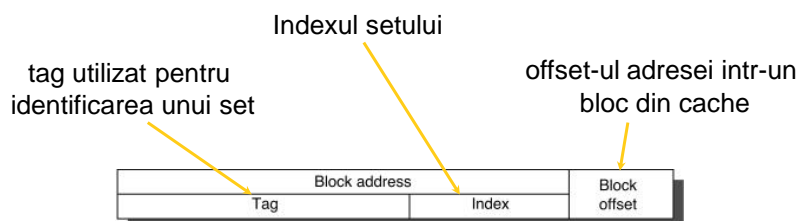
©Henri Casanova, U of Hawaii



Q2 – Block Identification

20

- Avand la dispozitie o adresa, cum putem identifica locatia acesteia in cache?
- Acest lucru se realizeaza impartind adresa in trei parti distincte



© 2003 Elsevier Science (USA). All rights reserved.

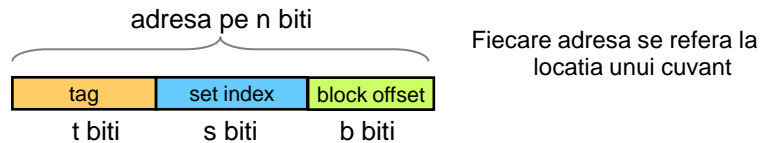
©Henri Casanova, U of Hawaii



Cache Mapat Direct

21

- Cea mai simpla solutie
 - O linie de memorie poate fi plasata intr-un singur loc in Cache



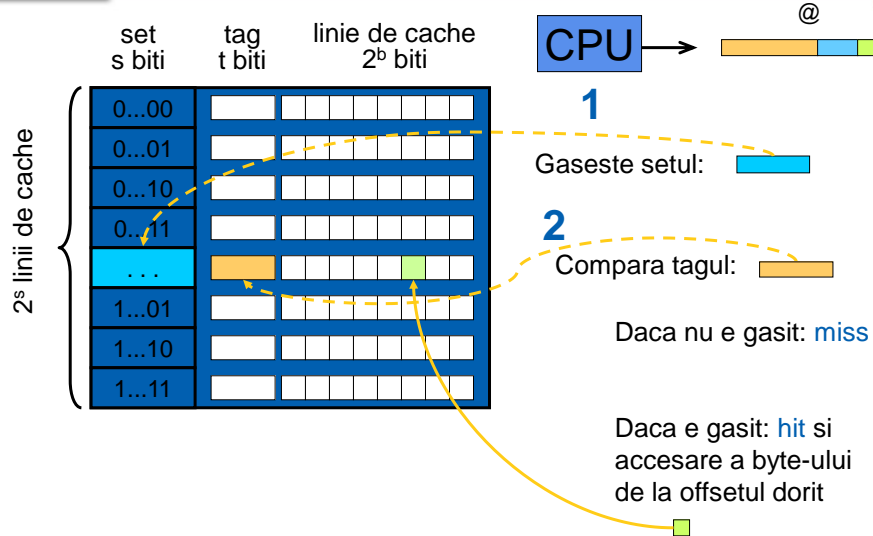
- Fiecare bloc de memorie/cache contine 2^b cuvinte
- Memoria Cache are contine 2^s linii
- Dimensiunea cache-ului este de 2^{s+b} cuvinte
- Fiecare linie din cache e identificata de tag

©Henri Casanova, U of Hawaii



Cum Functioneaza?

22

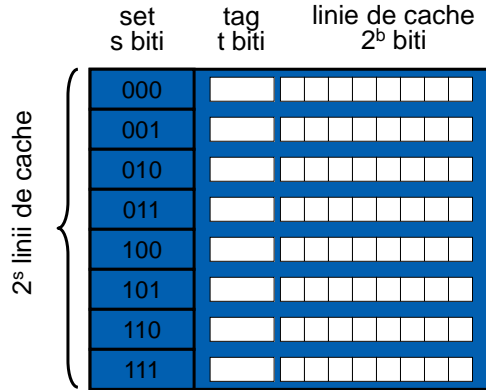


©Henri Casanova, U of Hawaii



Cache Mapat Direct

23



Liniile de cache contin date

Tag-urile contin o parte din adresa liniei de memorie

Seturile corespund unei parti a adresei liniei de memorie

Cache
s=3, t=4, b=3

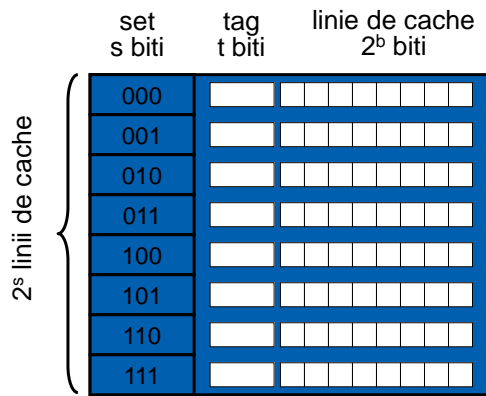
Locatiile de memorie intr-o linie de memorie sunt adrese prefixate cu **<tag><set>**

©Henri Casanova, U of Hawaii



Cache Mapat Direct

24

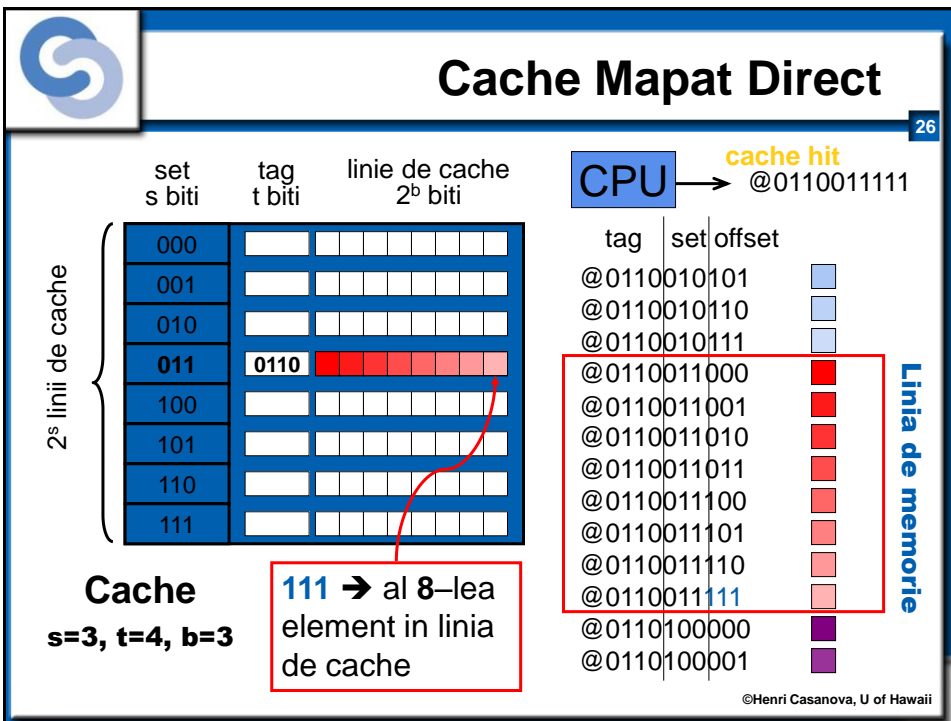
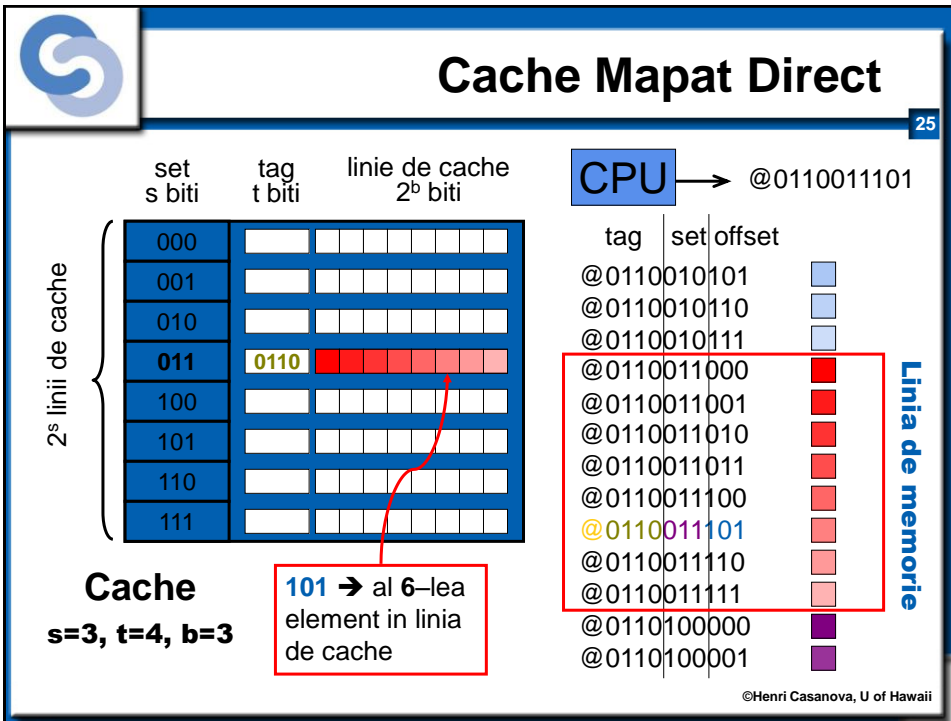


CPU → **cache miss** @0110011101

tag	set	offset
@0110010101		□
@0110010110		□
@0110010111		□
@0110011000		■
@0110011001		■
@0110011010		■
@0110011011		■
@0110011100		■
@0110011101		■
@0110011110		■
@0110011111		■
@0110100000		■
@0110100001		■

Cache
s=3, t=4, b=3

©Henri Casanova, U of Hawaii





Cache Mapat Direct – Concluzii

27

- Avantaje
 - Design simplu
 - Doar cateva comparatii intre parti din adrese de memorie
 - In plus, fiecare linie de cache are un bit de validare asociat
 - Astfel, Cache-ul mapat direct este:
 - Rapid
 - Necesita putin hardware
- Dezavantaj major
 - Este vulnerabil la **Thrashing** (murdarire)

©Henri Casanova, U of Hawaii



Mapare Set Asociativa

28

- Fiecare “set” din cache poate pastra mai multe linii din memorie (de la 2 pana la 8)
- O linie de memorie poate fi mapata pe oricare dintre aceste doua pozitii

	set s biti	tag t biti	linie de cache 2^b biti
2 ^s linii de cache	0...00		
	0...01		
	0...10		
	0...11		
	...		
	1...01		
	1...10		
	1...11		

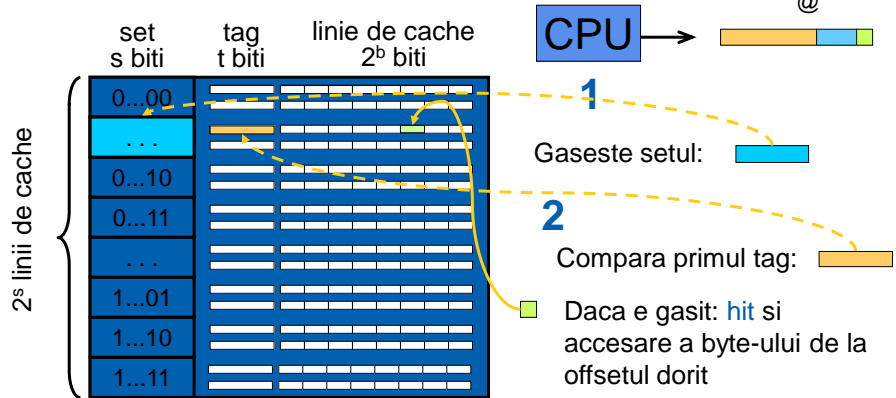
Toate adresele cu un set index similar “concoreaza” pentru **doua** frame bloc-uri posibile

©Henri Casanova, U of Hawaii



Cum Functioneaza?

29

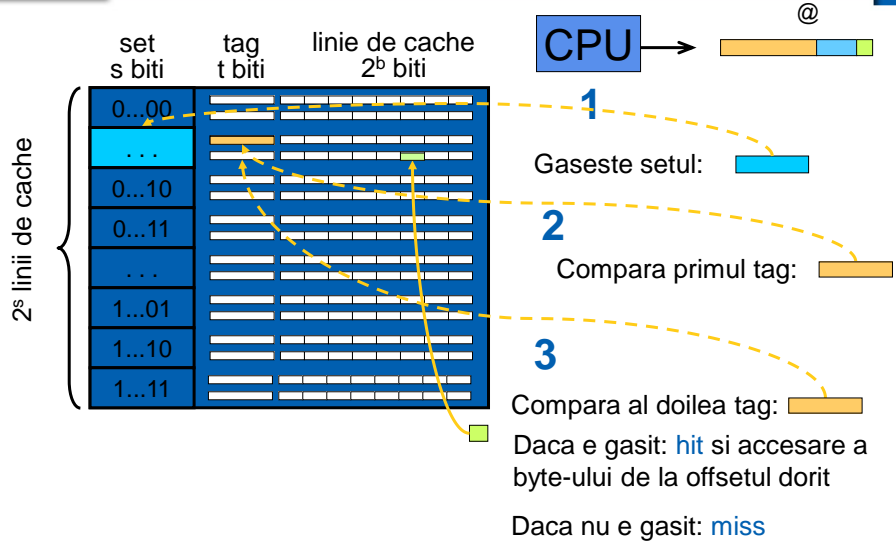


©Henri Casanova, U of Hawaii



Cum Functioneaza?

30



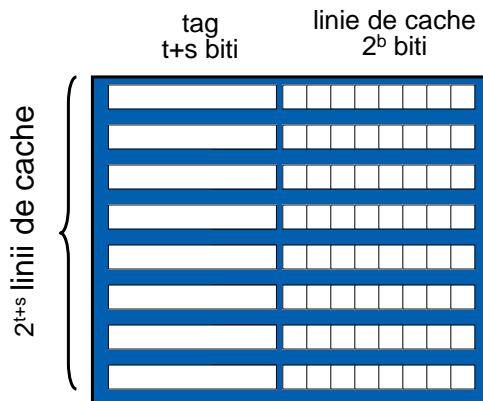
©Henri Casanova, U of Hawaii



Mapare Total Asociativa

31

- O linie de memorie poate fi plasata **oriunde** in Cache!
 - Se obtine considerabil mai putin thrashing
 - Complexitate crescuta a unitatii de comanda



Identificarea liniilor de Cache se va face cu bitii t+s din adresa liniei de memorie

©Henri Casanova, U of Hawaii



Q3 – Block Replacement

32

- Cand are loc un miss, controlerul de cache trebuie sa “faca ceva”
- Pe un cache mapat direct, este foarte simplu – se va suprascrie continutul unui block-frame cu date noi aduse din memorie
- Intr-o implementare n-set-asociativa insa, trebuie sa facem o alegere: care dintre cele n block-frame-uri trebuie suprascrise?
- In practica, exista trei strategii implementate pentru acest lucru
 - Random: foarte usor de implementat
 - First-In-First-Out (FIFO):
 - Un pic mai dificil de implementat
 - Aduce beneficii la cache-uri de dimensiuni mai mici
 - Least-Recently Used (LRU):
 - Si mai dificil de implementat, mai ales pentru cache-uri de dimensiuni mari
 - Aduce beneficii la cache-uri de dimensiuni mai mici
 - Atentie, trebuie sa avem mereu in vedere costul cache-ul vs latenta oferita de acesta: overhead-ul computational si hardware-ul additional necesar calcularii valorilor LRU

©Henri Casanova, U of Hawaii



Q4 – Write Strategy

33

- Sunt mult mai multe citiri decat scrieri in cache!
 - In general, toate instructiunile trebuiesc citite din memorie
 - Astfel, avem in medie:
 - 37% instructiuni load
 - 10% instructiuni store
 - Astfel, procentul de accese de tip scriere este: $.10 / (1.0 + .37 + .10) \sim 7\%$
 - Procentul de accese la date in memorie pentru scriere este astfel: $.10 / (.37 + .10) \sim 21\%$
- Amintiti-va principiul fundamental: **Make the common case fast!**
- Ca urmare, designul cache-urilor a fost optimizat pentru a face **citirile rapide**, si **NU** scrierile
 - Trebuie tinut cont si de faptul ca procesorul trebuie mereu sa astepte la citire, si aproape niciodata la scriere!
- Pe de alta parte, daca scrierile sunt extrem de incete, legea lui Amdahl ne spune ca performantele globale ale sistemului vor fi scazute
- Astfel, trebuie sa investim “ceva” efort si in imbunatatirea scrierilor

©Henri Casanova, U of Hawaii



Imbunatatirea Scrierilor in Cache

34

- A face citirile din cache rapid este simplu
 - Indiferent de locatia blocului din cache citit, se poate face cererea **imediat** ce adresa a fost generata de procesor
 - Necesita hardware pentru compararea **simultana** a tag-urilor si pentru citirea blocurilor
 - In cazul unui cache miss, acesta trebuie tratat ca atare
- Din pacate, a face scrierile rapid nu este la fel de simplu
 - Compararea tag-urilor **nu poate fi simultana** cu scrierea blocurilor in cache – trebuie sa ne asiguram ca nu se suprascrive un block frame ce nu este hit!
 - Scrierile sunt facute pentru o anumita dimensiune – pentru un subset al frame blocului
 - In citiri, bitii “**in plus**” pot fi cititi si apoi ignorati

©Henri Casanova, U of Hawaii



Politici de Scriere in Cache

35

- Dupa cum ne amintim de la CN...
- Write-through: datele sunt scrise in acelasi timp in blocul din cache si in blocul din memoria principala
- Write-back: datele sunt scrise in memoria principala doar cand un bloc frame este eliberat/inlocuit din cache
 - Se utilizeaza un bit “dirty” pentru a indica daca un bloc de inlocuit a fost modificat (s-a scris in el), pentru a salva scrieri inutile in memoria principala, cand un bloc este de fapt “curat”

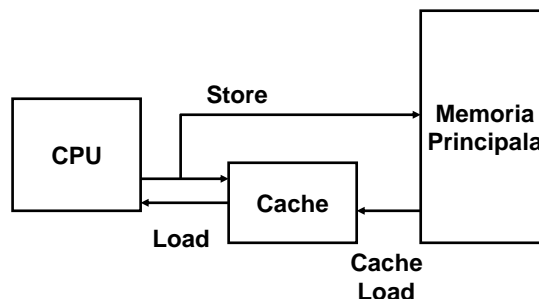
©Henri Casanova, U of Hawaii



Write-Through

36

- Ce se intampla cand procesorul modifica o locatie de memorie care se afla in Cache?
- Solutia 1: Write-Through
 - Se scrie in **acelasi** timp in cache si in memoria principala
 - Memoria si cache-ul sunt astfel mereu consistente



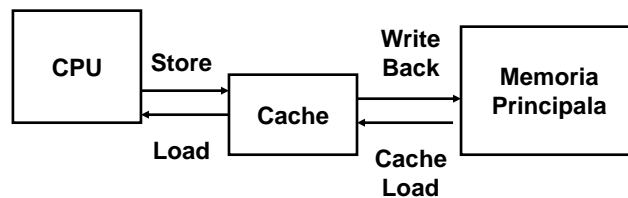
©Henri Casanova, U of Hawaii



Write-Back

37

- Solutia 2 – Write-Back
 - Se scrie doar in cache
 - Liniile de cache sunt scrise in memoria principala doar cand sunt evacuate/inlocuite
 - Necesita un bit “dirty” pentru a indica daca o linie de cache a fost modificata sau nu
 - Memoria si cache-ul **nu** mai sunt totdeauna consistente!



©Henri Casanova, U of Hawaii



Tradeoffs – Compromisuri

38

- Write-Back
 - Rapid – scrierile se petrec la viteza cache-ului si NU a memorie
 - Rapid – actualizari multiple ale aceluasi bloc de cache vor fi scrise inapoi in memoria principala in seturi de dimensiuni mai mari, utilizand mai putin din latimea de banda a memoriei
 - Solutie buna pentru servere cu multe procesoare/core-uri
- Write-Through
 - Mult mai usor de implementat ca Write-Back
 - Nivelul urmator de cache va avea intotdeauna o copie actualizata a datelor utilizate
 - Foarte important pentru sisteme multiprocesor – simplifica problemele de coerența a datelor

©Henri Casanova, U of Hawaii



Politica de Inlocuire a unei linii de Cache

40

- Cand doua linii de memorie sunt in cache si a treia linie vine, una din primele doua linii trebuie sa fie inlocuita: care din ele trebuie inlocuita?
- Deoarece nu cunoastem viitorul, utilizam euristici:
 - LRU: Least Recently Used
 - Este greu de implementat
 - FIFO
 - Este usor de implementat
 - Random
 - Si mai usor de implementat
- In general, cache-urile asociative:
 - Pot preveni fenomenul de thrashing
 - Necesita hardware mai complex decat cache-ul mapat direct
 - Sunt mai scumpe decat cache-ul mapat direct
 - Sunt mai incete decat cache-ul mapat direct

©Henri Casanova, U of Hawaii



Cuprins

41

- Ierarhia de memorii – Bottleneck-ul SC
- Localitatea datelor
- Cache – design, implementari si exemple
- **Imbunatatirea performantelor memoriei**



Imbunatatirea Performantelor Memoriei

42

- Exista mai multe moduri de a imbunatatii performanta cache-urilor si anume
 - Reducerea penalitatilor unui Cache Miss
 - Reducerea ratei de Cache Miss-uri
 - Reducerea timpului de rezolvare a unui Hit
 - Imbunatatirea memoriei
 - Performanta crescuta
 - Cost redus

©Henri Casanova, U of Hawaii



Optimizari de Compilator

43

- Sunt mai multe moduri in care un cod poate fi modificat pentru a genera mai putine miss-uri
 - Compilatorul
 - Utilizatorul
- Vom analiza un exemplu simplu – initializarea unui vector bidimensional (matrice)
- Vom presupune ca avem un compilator neperformant si vom optimiza codul in mod direct
 - Un compilator bun trebuie sa poata face acest lucru
 - Cateodata insa, compilatorul nu poate face tot ce este necesar

©Henri Casanova, U of Hawaii



Exemplu: Initializarea unui vector 2-D

44

```
int a[100][100];          int a[100][100];
for (i=0;i<100;i++) {    for (j=0;j<100;j++) {
  for (j=0;j<100;j++) {      for (i=0;i<100;i++) {
    a[i][j] = 11;          a[i][j] = 11;
  }                        }
}                          }
```

- Care varianta este mai buna?
 - i,j?
 - j,i?
- Pentru a raspunde la aceasta intrebare, trebuie sa intelegem modul de asezare in memorie al unui vector 2-D

©Henri Casanova, U of Hawaii



Vectori 2-D in Memorie

45

- Un vector 2-D static se declara:

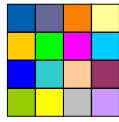
```
<type> <name>[<size>][<size>]
int array2D[10][10];
```
- Elementele unui vector 2-D sunt salvate in celule **contigue** de memorie
- Problema este ca:
 - Matricele sunt conceptual 2-D
 - Memoria unui sistem de calcul este 1-D
- Memoria 1-D a sistemului este descrisa de un singur numar: adresa de memorie
 - Similar cu numerele de pe axa reala
- Astfel, este necesara o mapare de la 2-D la 1-D
 - Cu alte cuvinte, de la abstractizarea 2-D utilizata in programare, la implementarea fizica in 1-D

©Henri Casanova, U of Hawaii

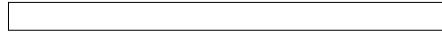


Maparea de la 2-D la 1-D

46



Matrice $n \times n$ 2-D



Memoria sistemului 1-D



Maparea 2-D la 1-D

Exista $n^2!$ mapari posibile



O alta mapare 2-D la 1-D

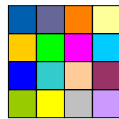
©Henri Casanova, U of Hawaii



Row-Major vs. Column-Major

47

- Din fericire, in orice limbaj sunt implementate maxim 2 din cele $n^2!$ mapari posibile, si anume:



- Row-Major:

– Liniile sunt stocate continuu



- Column-Major:

– Coloanele sunt stocate continuu



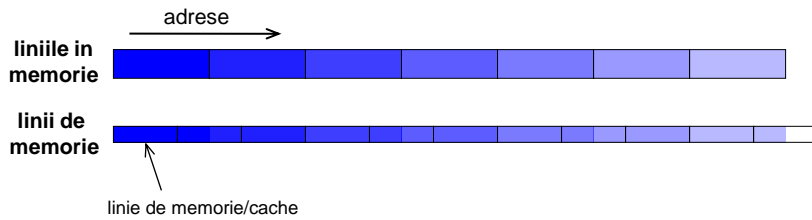
©Henri Casanova, U of Hawaii



Row-Major

48

- Row-Major este utilizat de C/C++



- Elementele matricii sunt stocate contiguu in memorie

©Henri Casanova, U of Hawaii

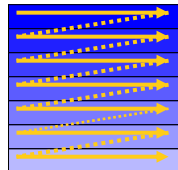


Row-Major

49

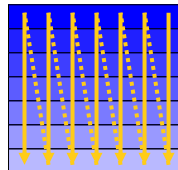
- C/C++ utilizeaza Row-Major
- Prima implementare (i, j)

```
int a[100][100];  
for (i=0; i<100; i++)  
  for (j=0; j<100; j++)  
    a[i][j] = 11;
```



- A doua implementare (j, i)

```
int a[100][100];  
for (j=0; j<100; j++)  
  for (i=0; i<100; i++)  
    a[i][j] = 11;
```



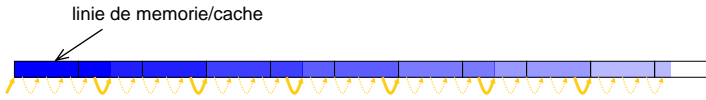
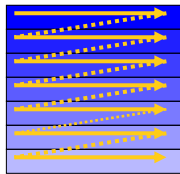
©Henri Casanova, U of Hawaii



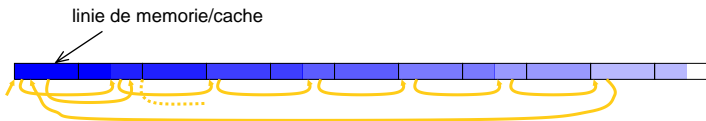
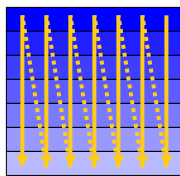
Numarul de Miss-uri

50

- Definim:
 - Matricea $n \times n$ 2-D array,
 - Fiecare element are e bytes,
 - Dimensiunea liniei de cache este b bytes



- Se obtine un miss la fiecare linie de cache: $n^2 \times e / b$
- Daca avem es: n^2 accese la memorie (toata matricea)
- Rata de miss-uri este: e/b
- Exemplu: Miss rate = 4 bytes / 64 bytes = **6.25%**
 - In afara cazului in care vectorul este foarte mic



- Avem un miss la fiecare acces
- Exemplu: Miss rate = **100%**
 - In afara cazului in care vectorul este foarte mic

©Henri Casanova, U of Hawaii

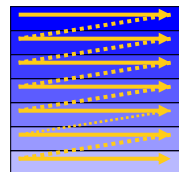


Initializarea Matricelor in C

51

- C/C++ utilizeaza Row-Major
- Prima implementare (i, j)

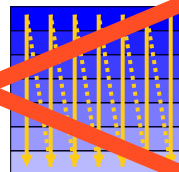
```
int a[100][100];
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    a[i][j] = 11;
```



Buna Localitate A Datelor

- A doua implementare (j, i)

```
int a[100][100];
for (j=0; j<100; j++)
  for (i=0; i<100; i++)
    a[i][j] = 11;
```



©Henri Casanova, U of Hawaii



Masurarea Performantelor

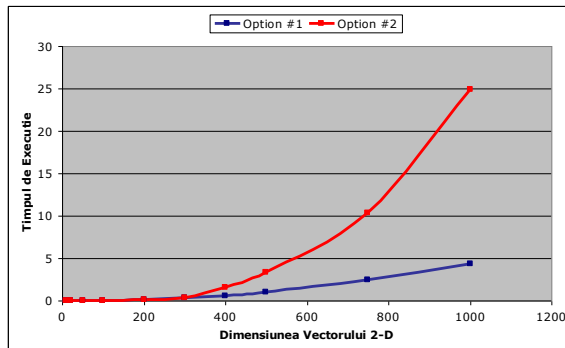
52

- C/C++ utilizeaza Row-Major
- Prima implementare

```
int a[100][100];
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
    a[i][j] = 11;
```

- A doua implementare

```
int a[100][100];
for (j=0;j<100;j++)
  for (i=0;i<100;i++)
    a[i][j] = 11;
```



Experimente pe un PC normal

- Alte limbaje de programare utilizeaza column major
 - FORTRAN de exemplu

©Henri Casanova, U of Hawaii



What About Dynamic Arrays?

53

- In unele limbaje, putem declara vectori cu dimensiuni variabile
 - FORTRAN:
`INTEGER A(M,N)`
- C-ul de exemplu nu permite acest lucru
- In C, trebuie sa alocam explicit memoria ca un **vector de vectori**:

```
int **a;
a = (int **)malloc(m*sizeof(int));
for (i=0;i<m;i++)
  a[i] = (int *)malloc(n*sizeof(int));
```

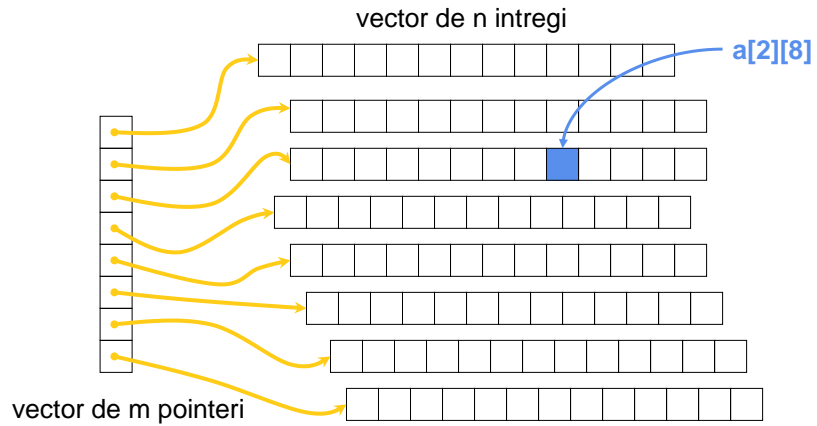
©Henri Casanova, U of Hawaii



Asezarea Memoriei

54

- O solutie “non contiguous” de tip row major



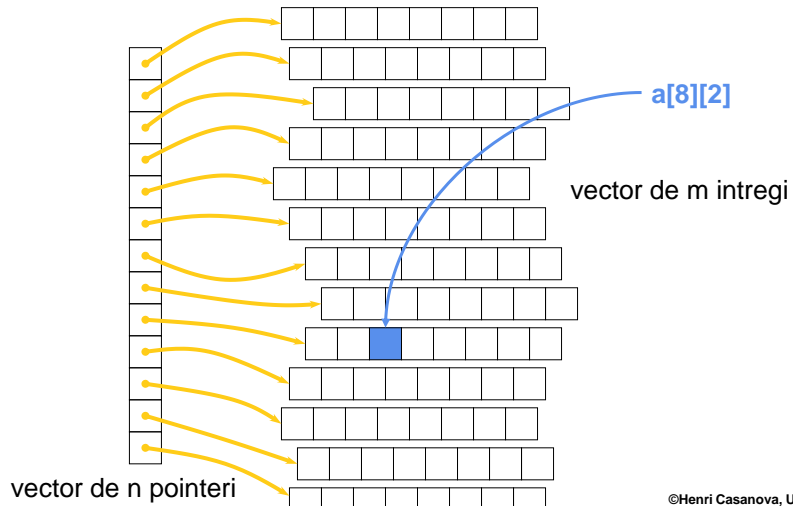
©Henri Casanova, U of Hawaii



Asezarea Memoriei

55

- Se poate inasa face si intr-o implementare column-major



©Henri Casanova, U of Hawaii



Vectori Dinamici

56

- Programatorul trebuie sa aleaga
 - Asezarea row-major sau column-major
 - Ordinea in care face initializarea vectorilor
- In Java, toti vectorii sunt alocati dinamic
- Vectorii de dimensiuni mari
 - **Dinamici** – de dimensiuni N-D sunt vectori (N-1)-D de vectori 1-D
 - **Statici** – o generalizare a solutiei row/column-major

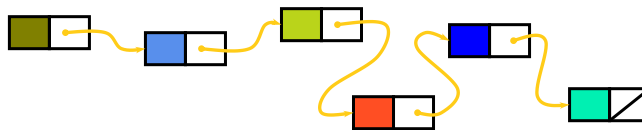
©Henri Casanova, U of Hawaii



Alte Exemple: Liste Inlantuite

57

- Sa luam in considerare exemplul unei liste inlantuite



- Intr-o implementare tipica, fiecare element al listei va fi alocat dinamic la momentul inserarii
- Elementele listei **nu** vor fi contigue
- Parcurgerea listei in ordine va genera in mod normal cate un cache-miss pentru **fiecare element!**
- Acest lucru poate pune probleme majore de performanta daca lista este suficient de lunga si ea este parcursa des...

©Henri Casanova, U of Hawaii



Liste Implementate ca Vectori

58

- Cum putem avea o localitate mai buna a datelor in liste?
- Le implementam ca vectori
- Tipul de date lista, poate fi implementat sub forma unui vector uni-dimensional
- Sunt trei operatii fundamentale cu liste:
 - Insert (list, current, next)
 - Remove (list, current)
 - Next (list, current)

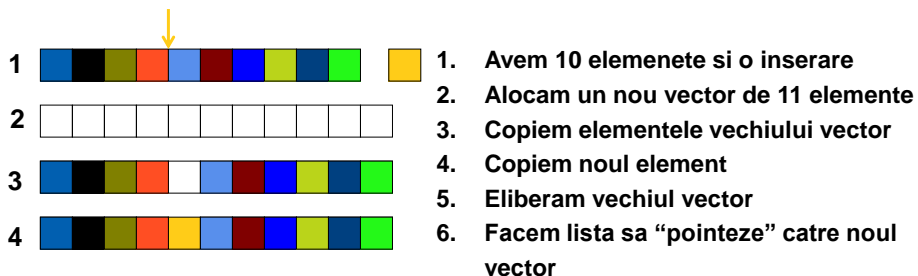
©Henri Casanova, U of Hawaii



Liste – Inserarea

59

- Vom construi o lista de intregi:
- Tipul de date “lista” arata:
 - int *array: un vector de intregi
 - int array_size: dimensiunea vectorului/listei
- Inserarea:



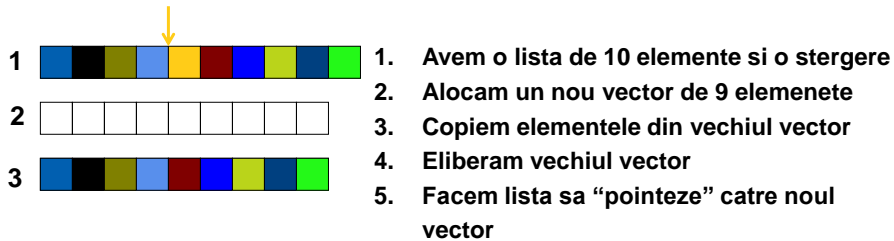
©Henri Casanova, U of Hawaii



Liste – Stergerea & Next

60

- Stergerea:



- Next: Trebuie doar sa returnam un index
 - Elementul listei este implementat ca intreg aici, dar poate fi implementat oricum transparent utilizatorului

©Henri Casanova, U of Hawaii



Liste – Concluzii

61

- O optimizare simpla:
 - In C, puteti utiliza “realloc” pentru a minimiza copierea elementelor
- Trade-off
 - Inserarea poate dura considerabil mai mult decat in implementarea traditionala
 - Au loc mai multe copieri de date
 - Lucrurile stau la fel si la stergere
 - Operatia Next este insa aproape instantanee
- Utilizatorul trebuie astfel sa identifice “the common case” si sa selecteze implementarea corespunzatoare
- Alte optimizari posibile
 - La inserare: dublati dimensiunea vectorului daca acesta e “prea mic”
 - La stergere: permiteti utilizarea unui vector “mai mare”

©Henri Casanova, U of Hawaii



Ierarhia de Memorii vs. Programatori

62

- Toate lucrurile prezentate aici pot fi foarte frumoase (poate)...
- Dar de ce ar fi importante pentru programatori?
- Am vazut ca putem determina ordinea de executie a buclilor
 - Poate ca acest lucru il poate face compilatorul pentru noi
- Totusi, un programator ce doreste performante de la codul sau, **trebuie** sa stie cum arata ierarhia de memorii pe care programeaza!
 - Daca stim dimensiunea cache-ului L2 (256KB), poate putem descompune problema in subprobleme de aceasta dimensiune pentru a exploata localitatea datelor
 - Daca stim ca o linie de cache are 32 bytes, putem calcula precis numarul de cache-miss-uri cu o formula si astfel seta un parametru optim pentru programul nostru
- Pentru a avea o experienta activa cu ierarhia cache-urilor, e util sa vedem cum putem scrie programe care sa masoare in mod automat aceste caracteristici

©Henri Casanova, U of Hawaii



Masurarea Numarului de Cache-Miss-uri

63

- Sa presupunem urmatorul fragment de cod

```
char a[N];
for (i=0; i<N; i++)
    a[i]++;
```
- Presupunem ca L este dimensiunea liniei de cache in bytes
- Numaram numarul de cache-miss-uri:
 - a[0]: miss (incarcam o noua linie de cache)
 - a[1]: hit (in linie de cache)
 - ...
 - a[L-1]: hit (in linie de cache)
 - a[L]: miss (incarcam o noua linie de cache)
 - ...
- Numarul de miss-uri este astfel: $\sim N / L$

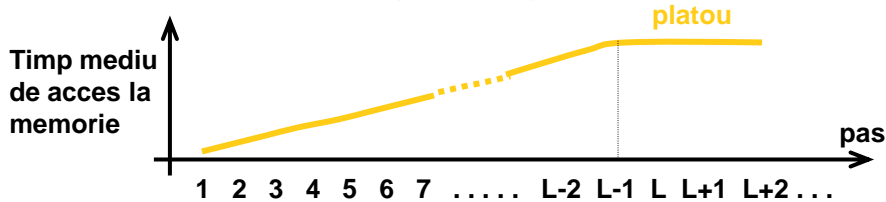
©Henri Casanova, U of Hawaii



Masurarea L-ului?

66

- Cea mai buna performanta: pas=1
- Cea mai proasta performanta: pas $\geq L$
- Daca masuram performanta codului pentru diverse valori ale pasului, obtinem un grafic de genul:



- Gaseste inceputul platoului (pasul in bytes) pentru care performanta codului nu se mai inrautateste odata cu cresterea pasului
- Aces pas (in bytes) este dimensiunea liniei de cache!



Masurarea L-ului?

67

- Cum scriem un program pentru masurarea performantei pentru mai multe valori pentru pas?
- Performanta – timpul mediu pentru accesul la memorie
- Alocati vectori **mari** de caractere
- Creati bucle cu valori pentru pas intre 1 si 256
- Pentru fiecare pas ales, parcurgeti in mod repetat vectorul
 - Faceti operatii cu fiecare element al vectorului (etc)
 - Masurati timpul cat dureaza aceste operatii
 - Masurati cate operatii ati operat in total
 - Impartiti numarul de operatii la timpul masurat



Masurarea Dimensiunii Cache-ului

68

- Daca L este dimensiunea liniei de cache
- Sa consideram urmatorul cod

```
char x[1024];
for (step=0;step<1000;step++)
    for (i=0;i<1024;i+=L)
        x[i]++;
```
- Daca cache-ul este **mai mare de 1024 de bytes**, dupa prima iteratie a buclei “step”, tot vectorul x e in cache si **nu vom mai avea** miss-uri deloc
- Daca cache-ul este **mai mic decat 1024 de bytes**, vom avea mereu un numar de miss-uri

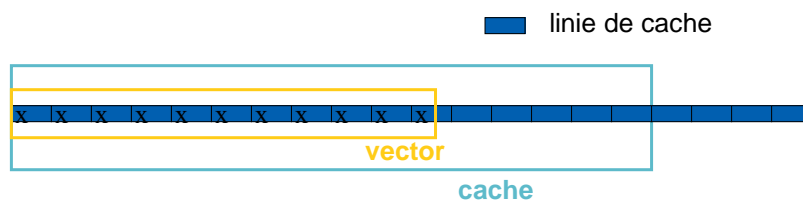
©Henri Casanova, U of Hawaii



Masurarea Dimensiunii Cache-ului

69

- Exemplificare:



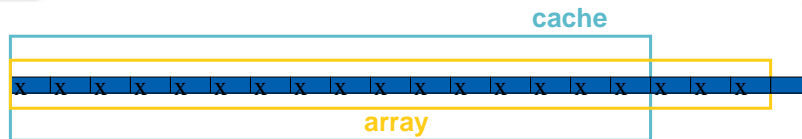
- Cache-ul este suficient de mare pentru a contine 16 linii
- Vectorul intra in 11 linii de cache
- Astfel, vor fi 11 miss-uri si apoi vor exista doar hit-uri
- Pentru un numar mare de iteratii, hit-rate-ul va fi aproape de 100%

©Henri Casanova, U of Hawaii



Masurarea Dimensiunii Cache-ului

70



- Cache-ul poate contine 16 linii
- Vectorul intra insa doar in 19 linii de cache
- Initial vor fi 19 miss-uri
- Apoi va trebui sa citim aceleasi 19 linii de cache
- Doar 16 intra insa in cache
- Vom avea astfel 13 hit-uri si 3 miss-uri
- Acest comportament va fi repetat pentru fiecare iteratie
- Pentru un numar mare de iteratii, se ajunge la un hit-rate de aproximativ 81%

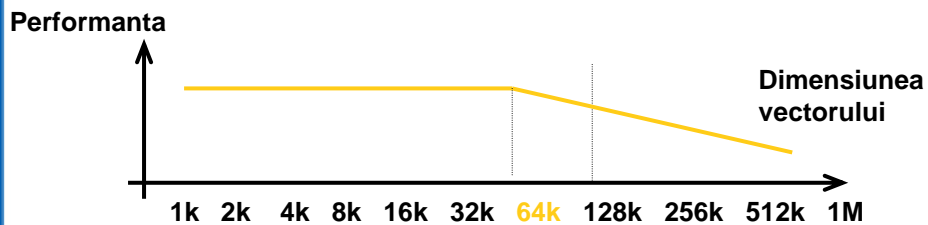
©Henri Casanova, U of Hawaii



Masurarea Dimensiunii Cache-ului

71

- Astfel:
 - Daca vectorii sunt mici si intra in cache hit-rate = 100%
 - Daca vectorii sunt mai mari decat cache-ul hit rate < 100%



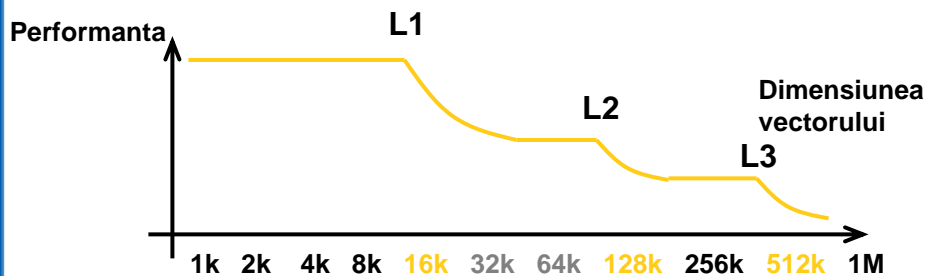
©Henri Casanova, U of Hawaii



Mai multe nivele de Cache?

72

- In general inasa, exista mai multe nivele de cache: L1, L2, L3
- Avand in vedere acest fapt, graficul anterior devine

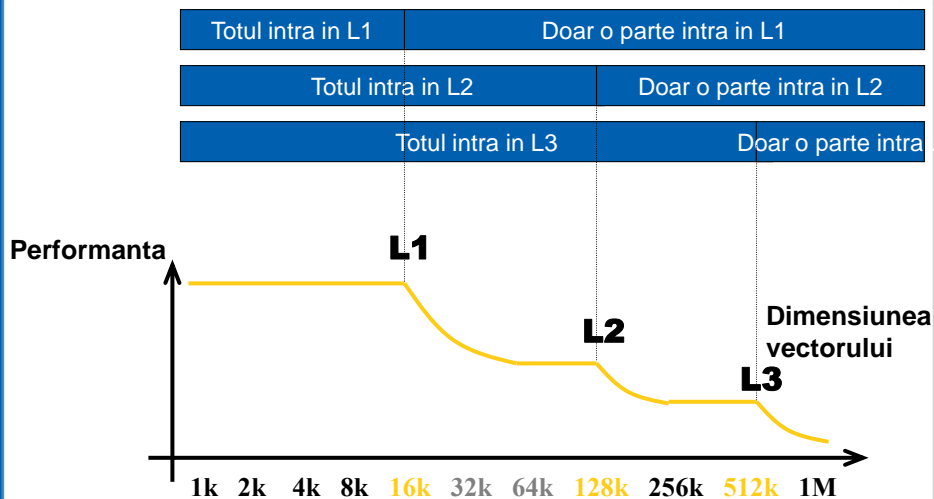


©Henri Casanova, U of Hawaii



Mai multe nivele de Cache?

73



©Henri Casanova, U of Hawaii



Cache-uri in Procesoare Reale

74

- Exista 2 sau 3 nivele de cache
- Cache-urile aproape de procesor sunt in general mapate direct, si cele mai departate sunt asociative
- Cache-uri diferite de date/instructiuni aproape de procesor, si unificate in rest
- Write-through si write-back sunt la fel de des intalnite, dar nu va exista nicidodata o implementare write-through pana la memoria principala
- Liniile de cache aveau in mod normal 32-byte, dar acum exista foarte des linii de 64- si 128-bytes
- Cache-uri neblockante
 - La un miss, nu bloca procesorul ci executa instructiuni de dupa load/store-ul curent
 - Blocheaza procesorul doar daca aceste operatii utilizeaza date din load
 - Cache-urile trebuie sa poata sa serveasca multiple accese “invechite”

©Henri Casanova, U of Hawaii



Performanta Cache-urilor

75

- Performanta cache-ului este data de accesul mediu la memorie
 - Programatorii sunt interesati in general doar de timpul de executie – insa timpul de acces la memorie este o componenta extrem de importanta
- Formula e simpla:
 - $\text{timpul de acces la memorie} = \text{hit time} + \text{miss rate} * \text{miss penalty}$
 - Miss-rate este procentul de miss-uri pe acces la date si **NU** la instructiuni!
- La fel ca in cazul timpului de executie procesor, termenii ecuatiei **NU** sunt independenti
 - Nu putem spune: reduc numarul de instructiuni, fara a creste numarul de instructiuni pe ciclu sau viteza ceasului sistemului
 - Similar, nu putem spune: reduc miss-rate-ul fara a creste hit-time

©Henri Casanova, U of Hawaii



Impactul Miss-urilor

76

- Miss penalty depinde de tehnologia de implementare a memoriei
- Procesorul masoara numarul de cicluri pierdute
- Un procesor mai rapid e "lovit" mai tare de **memorii lente** si **miss-rate-uri crescute**
- Astfel, cand incercati sa estimati performantele unui sistem de calcul, comportamentul cache-urilor **trebuie** luat in considerare (Amdahl – CPU vs. memorie)
- **De ce ne intereseaza?**
- Pentru ca putem rescrie/rearanja codul pentru a utiliza mai bine localitatea datelor

©Henri Casanova, U of Hawaii



De ce ne Intereseaza?

77

- Putem citi dimensiunile cache-urilor din specificatiile masini pe care rulam
 - Dar... ceva poate lipsi
 - Se poate sa nu avem acces la specificatii
- E util sa avem o optimizare automatizata
 - Ce trebuie sa fac pentru a scrie un program ce sa ia in considerare Cache-ul?
 - Utilizeaza constanta CACHE_SIZE pentru a seta dimensiunea cache-ului sistemului
 - Utilizeaza aceasta constanta cand aloca memoria

```
char x[CACHE_SIZE]
for (i=0; i < CACHE_SIZE; i++)
```
 - Inainte de a compila un program, ruleaza un alt utilitar pentru a descoperi dimensiunea cache-ului
 - Seteaza apoi constanta CACHE_SIZE la valoarea astfel determinata

```
#define CACHE_SIZE 1024*32
```
- **Un programator (bun) nu poate ignora cache-ul**
- Desi unele structuri de date par naturale, ele pot fi extrem de ineficiente in cache (liste, pointeri, etc) si de aceea ele ar trebui evitate cand se doreste o performanta crescuta a codului

©Henri Casanova, U of Hawaii