UNIUNEA EUROPEANĂ

GUVERNUL ROMÂNIEI

Instrumente Structurale
2007-2013


# Platformă de e-learning și curriculă e-content pentru învățământul superior tehnic

## Arhitectura Sistemelor de Calcul

## 36. Tehnici de optimizare utilizare in calculul de inalta performanta

# Imbunatatirea Performantelor Memoriei

- 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

# Optimizari de Compilator

- 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

# Exemplu: Initializarea unui vector 2-D

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

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

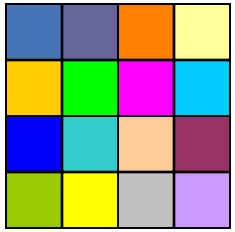
# Vectori 2-D in Memorie

- Un vector 2-D static se declara:

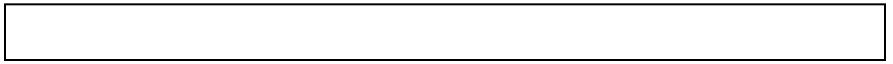
```
<type> <name>[<size>][<size>]
int array2D[10][10];
```

- 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

# Maparea de la 2-D la 1-D



Matrice nxn 2-D



Memoria sistemului 1-D

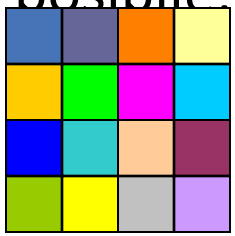
Maparea 2-D la 1-D

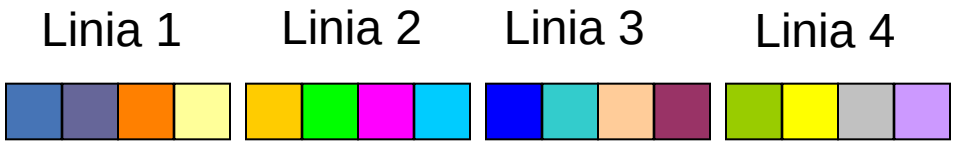
O alta mapare 2-D la 1-D

**Exista $n^2$! mapari posibile**

# Row-Major vs. Column-Major

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



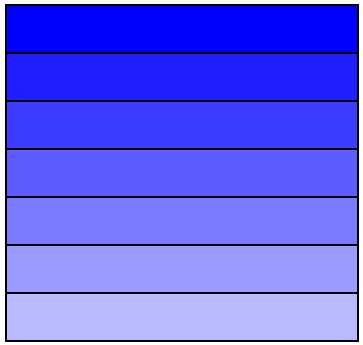


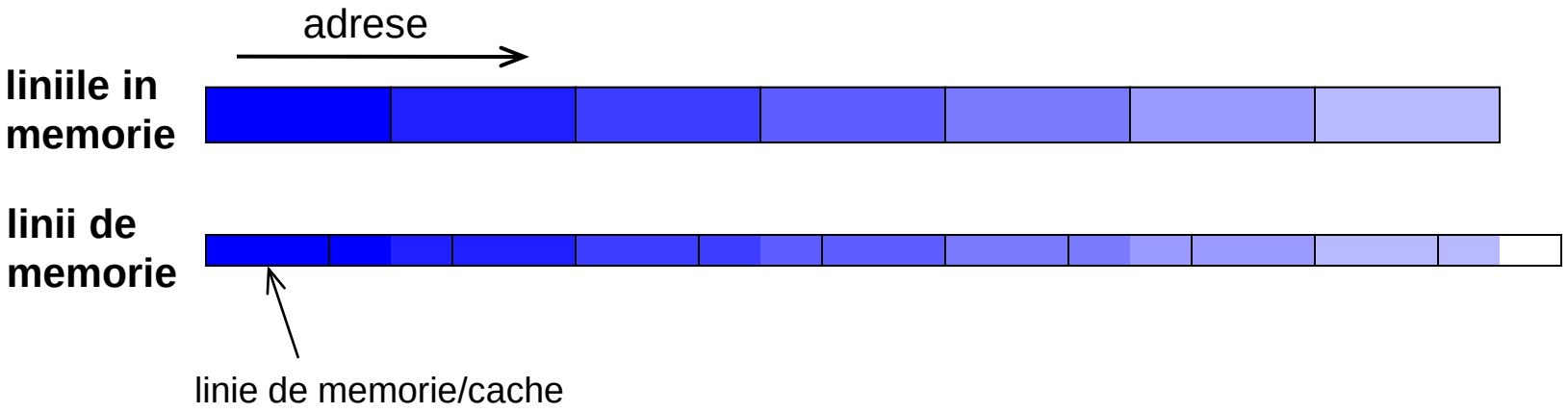

- Row-Major:
  - **Liniile sunt stocate continuu**


- Column-Major:
  - **Coloanele sunt stocate continuu**

# Row-Major

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



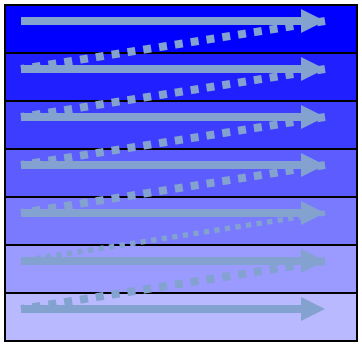



- Elementele matricii sunt stocate contiguu in memorie

# Row-Major

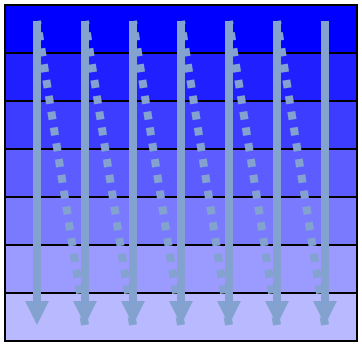
- C/C++ utilizeaza Row-Major
- Prima implementare (i, j)

```
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)

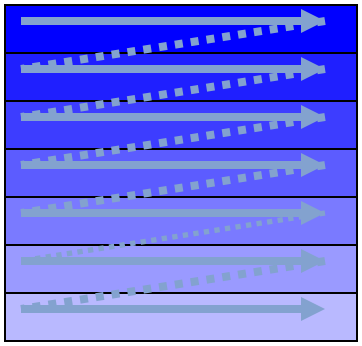
```
int a[100][100];
for (j=0;j<100;j++)
  for (i=0;i<100;i++)
    a[i][j] = 11;
```

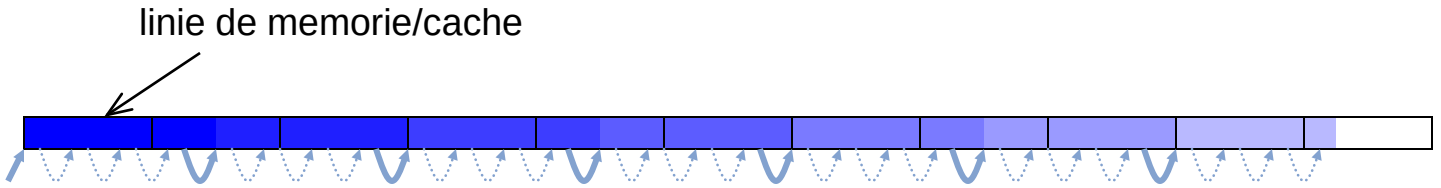


# Numarul de Miss-uri

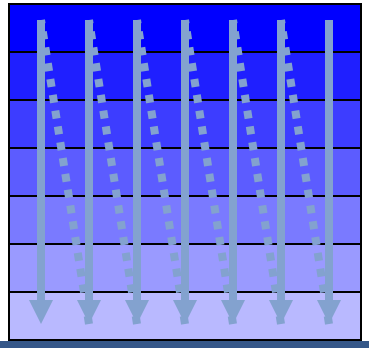
Definim:

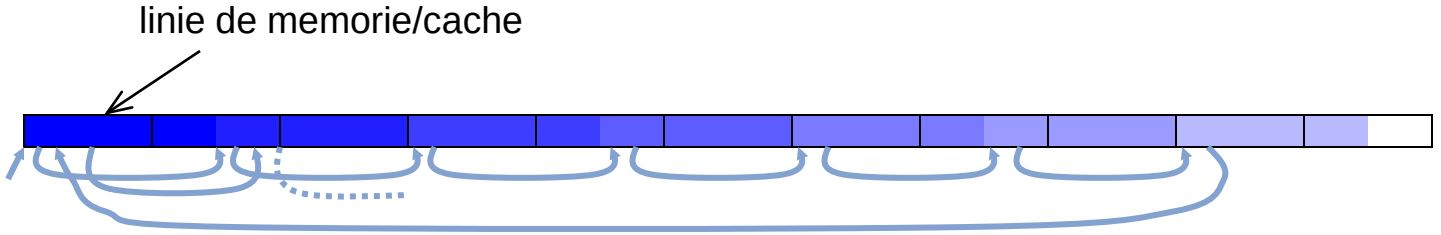
- Matricea nxn 2-D array,
- Fiecare element are e bytes,
- Dimensiunea liniei de cache este b bytes






- 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



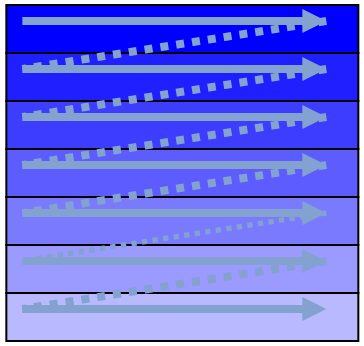



- Avem un miss la fiecare acces
- Exemplu: Miss rate = **100%**
  - In afara cazului in care vectorul este foarte mic

# Initializarea Matricelor in C

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

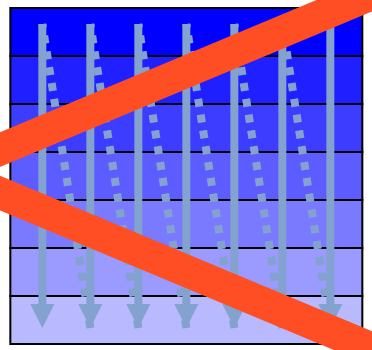
```
int a[100][100];
for (i=0;i<100;i++)
  for (j=0;j<100;j++)
    a[i][j] = 11;
```



Buna Localitate A Datelor

- A doua implementare (j, i)

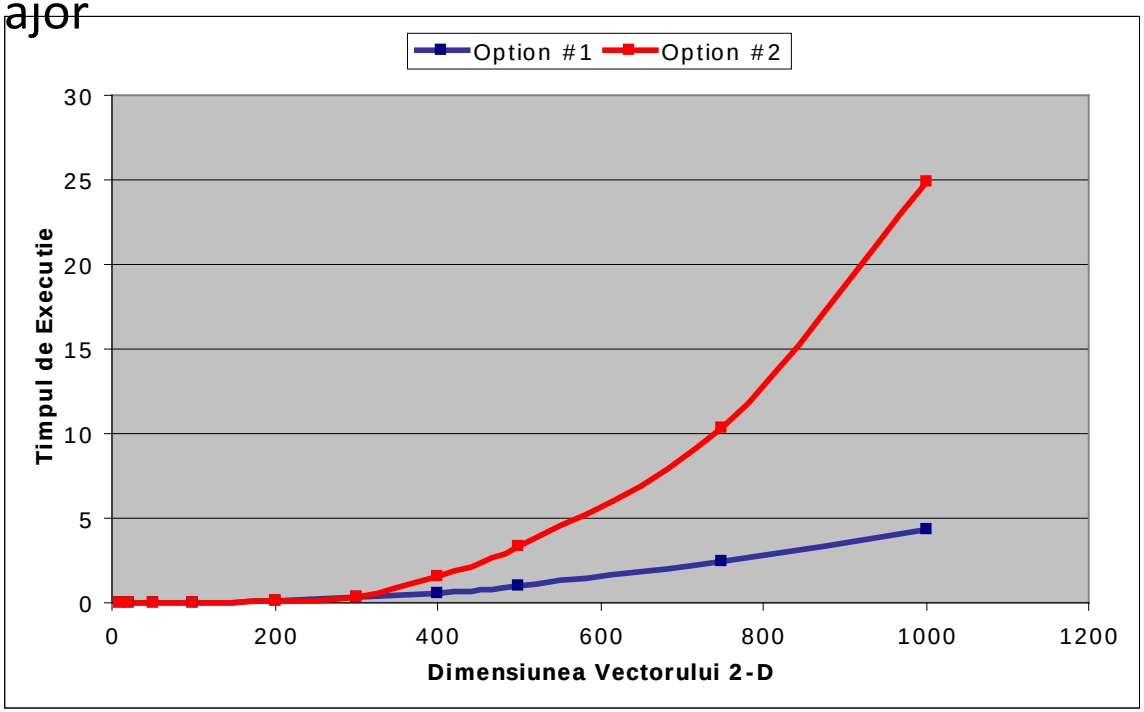
```
int a[100][100];
for (j=0;j<100;j++)
  for (i=0;i<100;i++)
    a[i][j] = 11;
```



# Masurarea Performantelor

- 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 limbare de programare utilizeaza column major
  - FORTRAN de exemplu

“

# What About Dynamic Arrays?

- In unele limbaje, putem declara vectori cu dimensiuni variabile
  - FORTRAN:

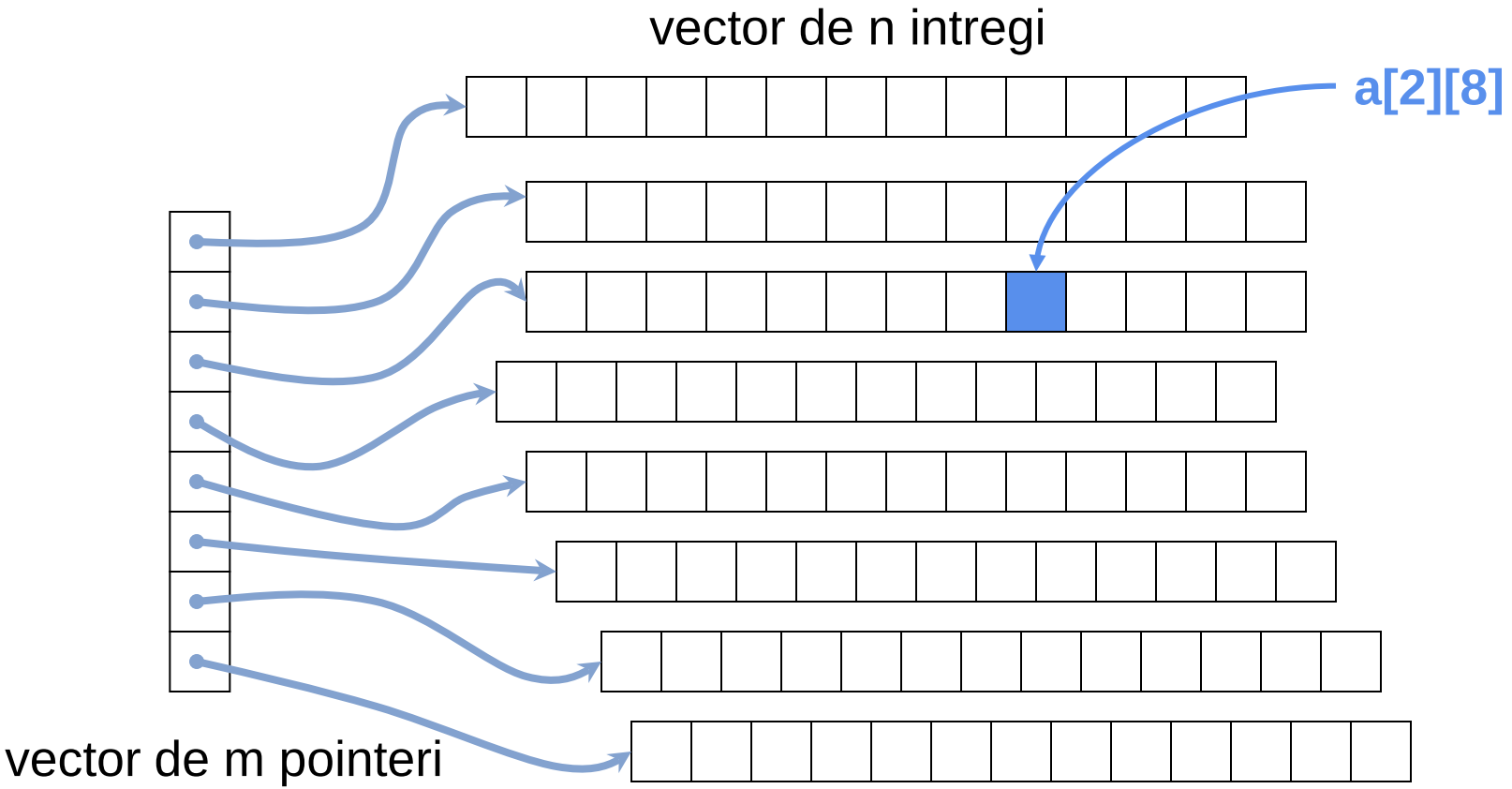
```
INTEGER A(M,N)
```

- 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));
```

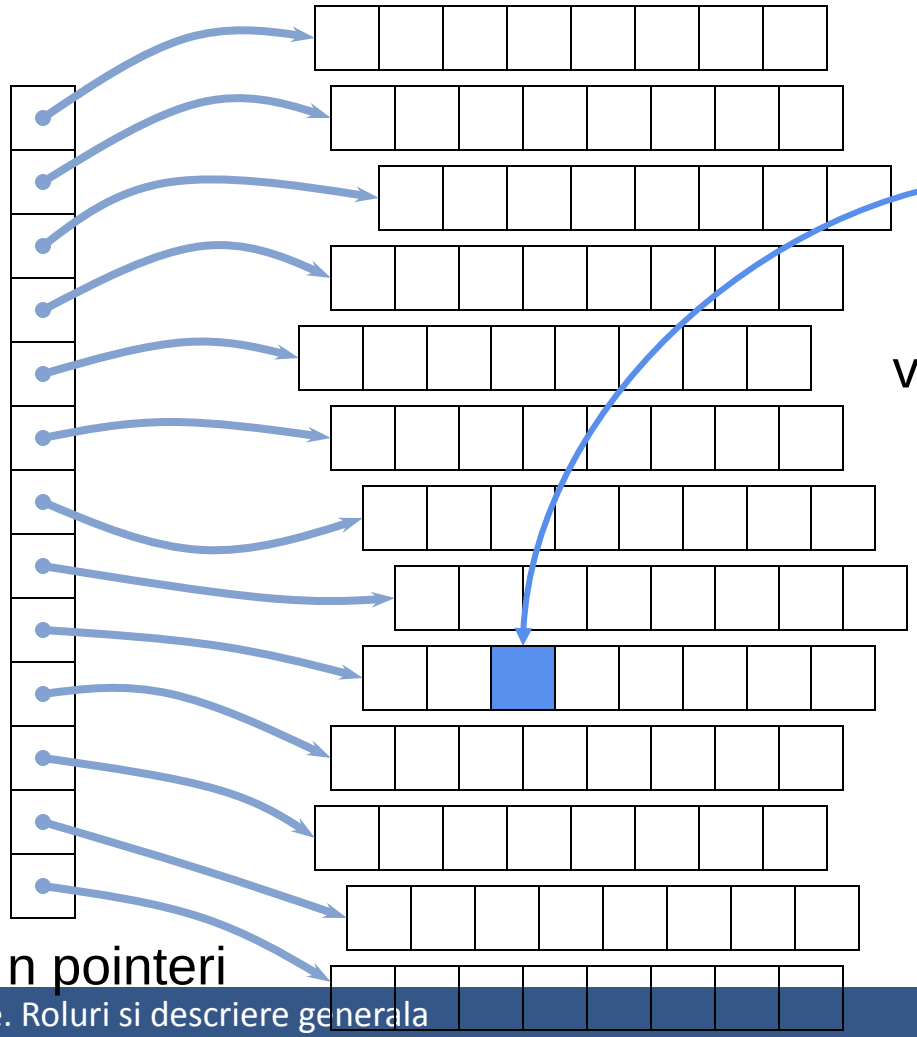
# Asezarea Memoriei

- O solutie “non contiguous”de tip row major



# Asezarea Memoriei

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

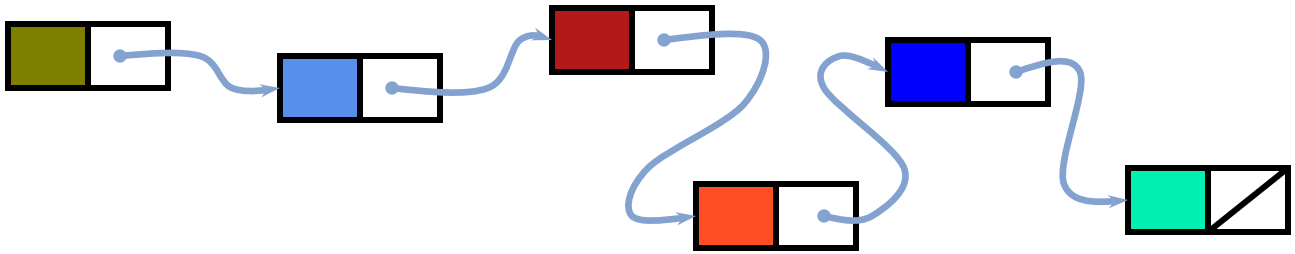


# Vectori Dinamici

- Programatorul trebuie sa aleaga
  - Asezarea row-major sau column-major
  - Ordinea in care face initializarea vectorilor
- In Java, toti vectorii sunt alocati dimanic
- 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

# Alte Exemple: Liste Inlantuite

- Sa luam in considerare exemplul unei liste inlantuite



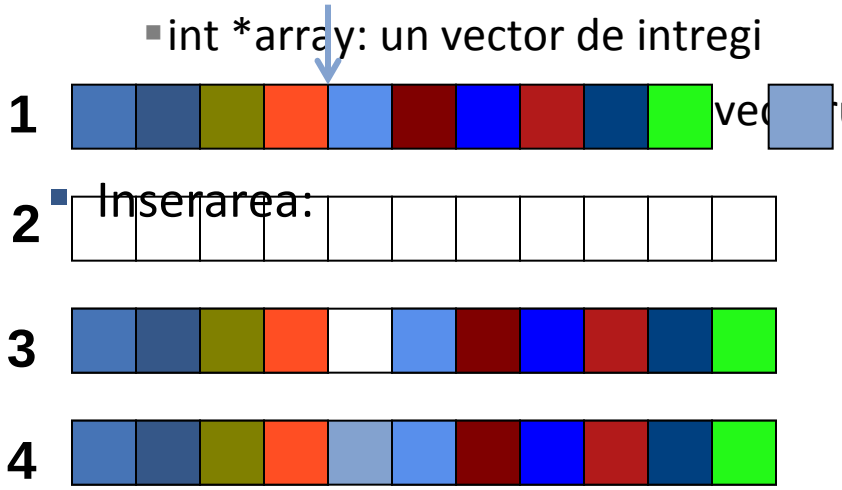
- Intr-o implementare tipica, fiecare element al listei va fi alocat dinamic la momentul inserarii
- Elementele listei nu vor fi contigue
- Parcurgerea listei in ordine va genera in mod normal cate un cache-miss pentru fiecare element!
- Acest lucru poate pune probleme majore de performanta daca lista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

# Liste Implementate ca Vectori

- 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)

# Liste – Inserarea

- Vom construi o lista de intregi:
- Tipul de date “lista” arata:
  - int *array: un vector de intregi



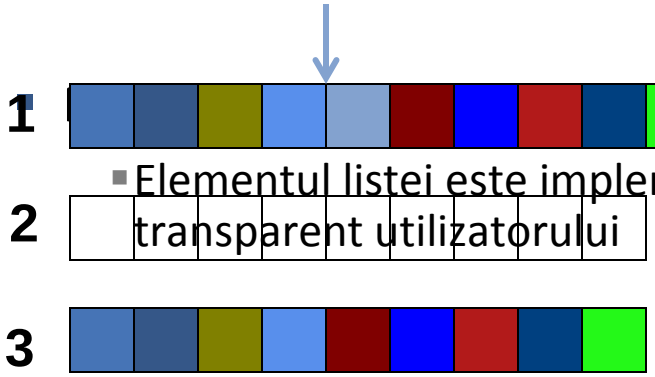

- Inserarea:

1. Avem 10 elemenete si o inserare
2. Alocam un nou vector de 11 elemente
3. Copiem elementele vechiului vector
4. Copiem noul element
5. Eliberam vechiul vector
6. Facem lista sa “pointeze” catre noul vector

- Stergerea:

# Liste – Stergerea & Next

- Elementul listei este implementat ca un întreg dar, de fapt, poate fi implementat oricum transparent utilizatorului




1. Avem o lista de 10 elemente si o stergere
2. Alocam un nou vector de 9 elemente
3. Copiem elementele din vechiul vector
4. Eliberam vechiul vector
5. Facem lista sa “pointeze” catre noul vector

# Liste – Concluzii

- 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”

# Ierarhia de Memorii vs. Programatori

- 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 buclelor
  - 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

# Masurarea Numarului de Cache-Miss-uri

- Sa presupunem urmatorul fragment de cod

```
char a[N];
 for (i=0;i<N;i++)
    a[i]++;
```

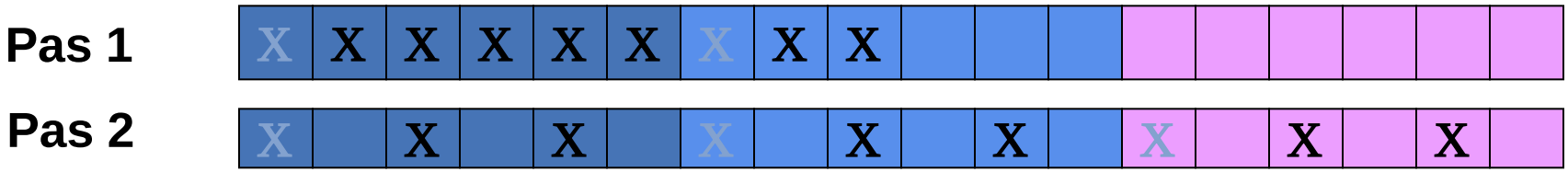
- Presupunem ca L este dimensiunea liniei de cache in bytes
- Numaram numarul de cache-miss-uri:
  - a[0]: miss (incarcam o noua linie de cache)
  - a[1]: hit (in linie de cache)
  - ...
  - a[L-1]: hit (in linie de cache)
  - a[L]: miss (incarcam o noua linie de cache)
  - ...
- Numarul de miss-uri este astfel: ~ N / L

# Masurarea L-ului?

- Codul anterior acceseaza elementele vectorului cu pasul 1
  - Pasul este diferenta in bytes intre doua adrese accesate in doua accesuri succesive la memorie
- Ce se intampla cand avem un pas 2?

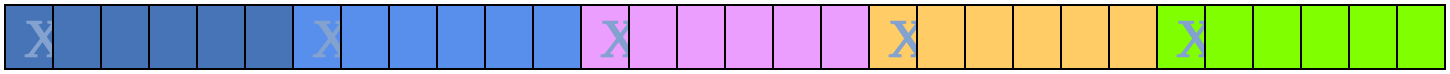
```
char a[N];
  for (i=0;i<N;i+=2)
          a[i]++;
```

- Avem practic un numar dublu de miss-uri fata de cazul anterior!
  - O modificare minora (aparent) are un impact major asupra performantelor

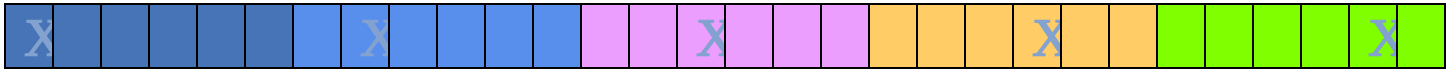


# Masurarea L-ului?

- Putem astfel creste pasul pana cand…
- Pasul ajunge sa fie egal cu L:
  - Fiecare acces are nevoie de o linie proprie de cache
  - N accesuri la memorie ➔ N cache-miss-uri

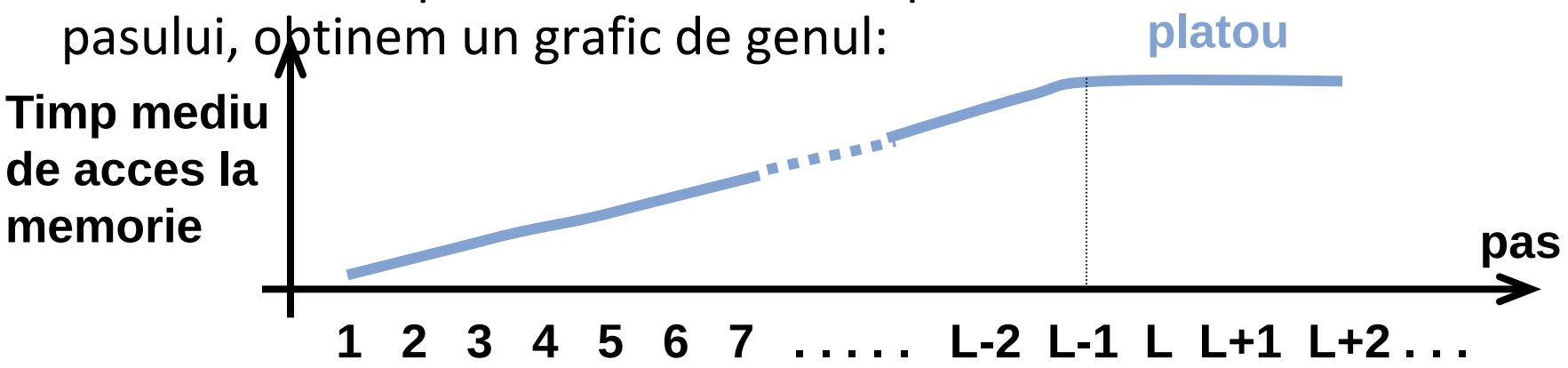


- Ce se intampla daca pasul este L+1?
  - Fiecare acces are nevoie de o linie proprie de cache
  - N accesuri la memorie ➔ N cache-miss-uri



# Masurarea L-ului?

- Cea mai buna performanta: pas=1
- Cea mai proasta performanta: pas ≥ 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?

- 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

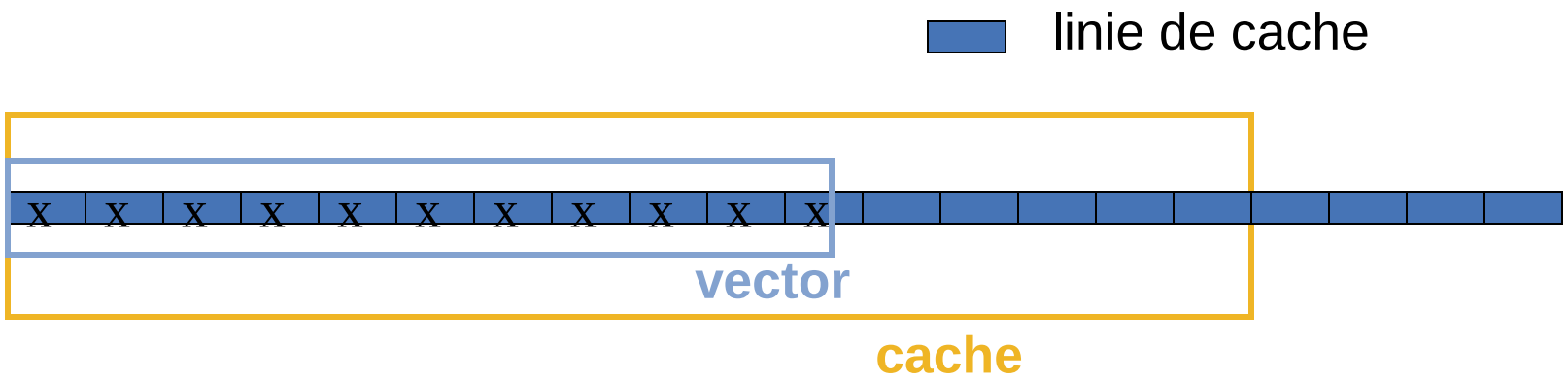
- Daca L este dimensiunea liniei de cache
- Sa consideram urmatorul cod

```
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

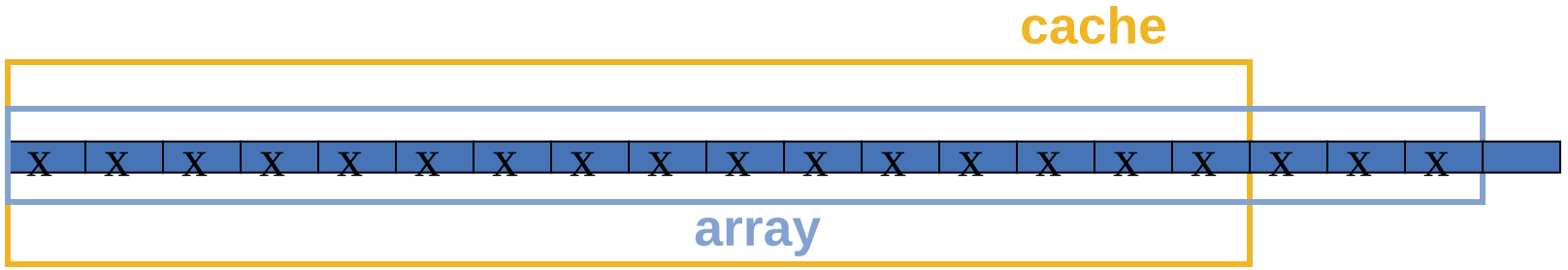
# Masurarea Dimensiunii Cache-ului

- 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%

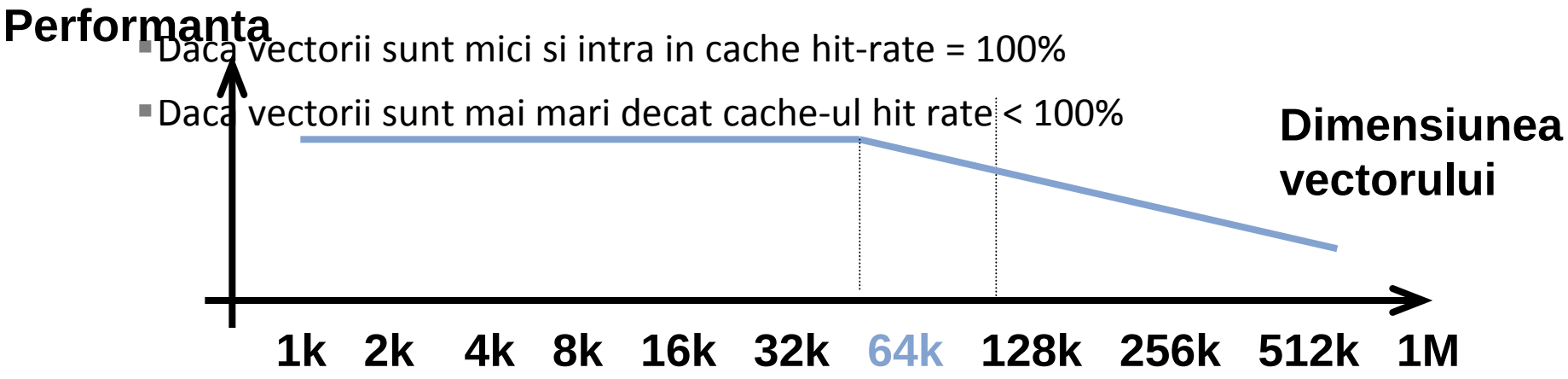
# Masurarea Dimensiunii Cache-ului




- 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%

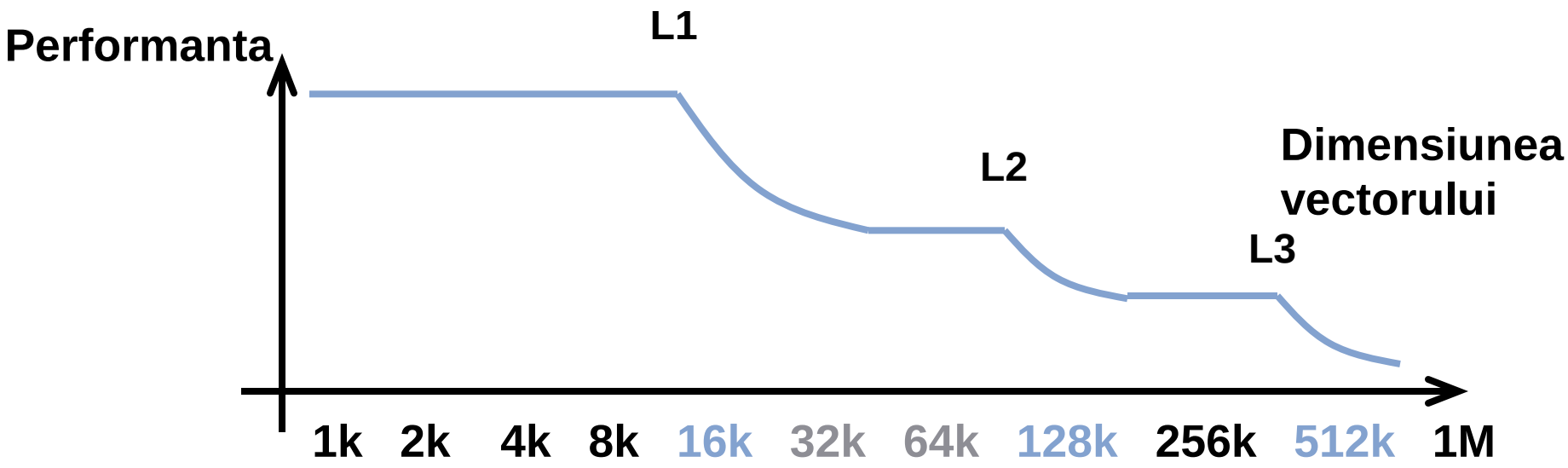
# Masurarea Dimensiunii Cache-ului

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

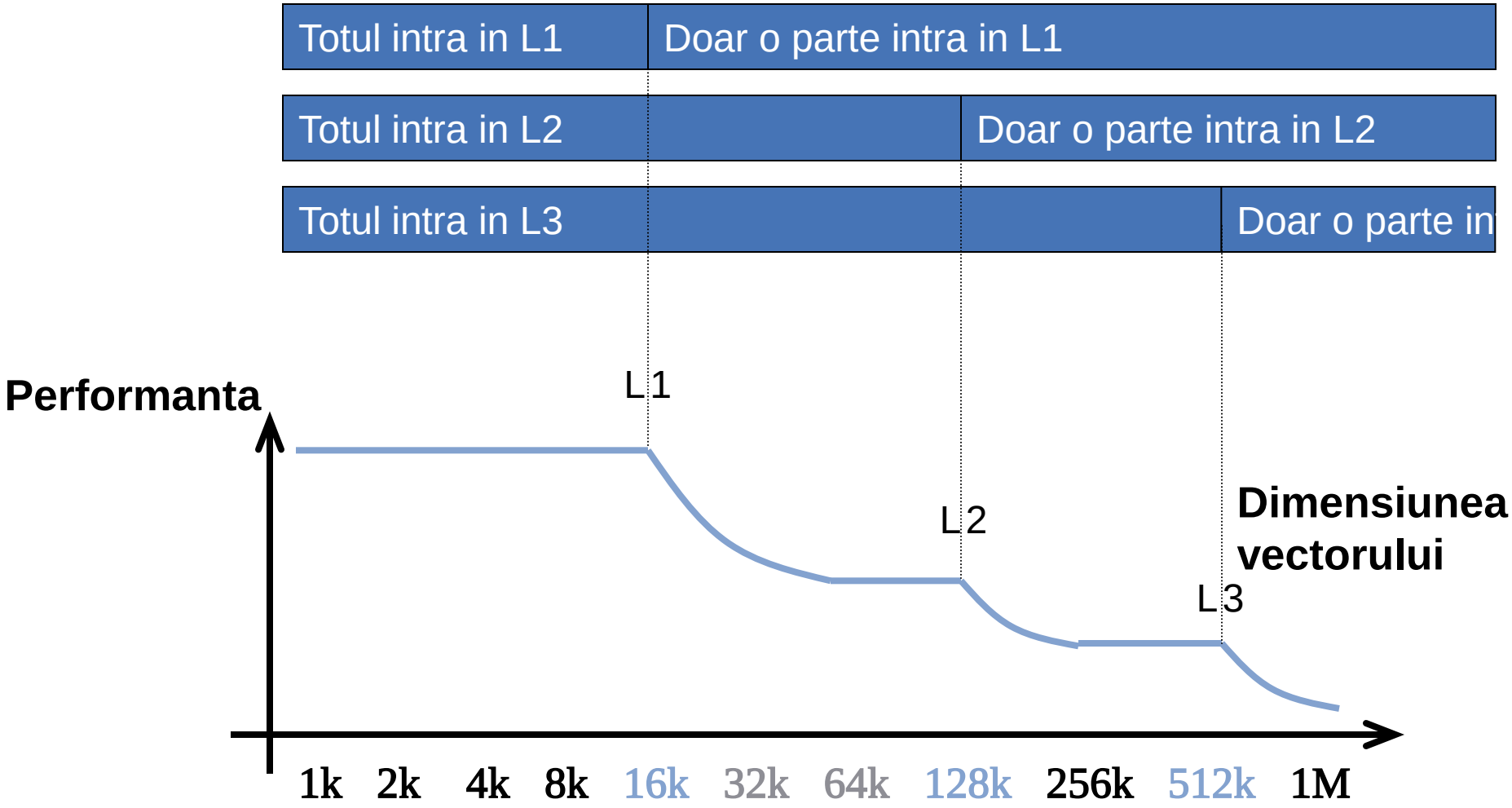


# Mai multe nivele de Cache?

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



# Mai multe nivele de Cache?



# Cache-uri in Procesoare Reale

- 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 neblocante
  - 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”

# Performanta Cache-urilor

- Performanta cache-ului este data de accesul mediu la memorie
  - Programatorii sunt interesati in general doar de timpul de excutie – insa timpul de acces la memorie este o componenta extrem de importanta
- Formula e simpla:
  - timpul de acces la memorie = hit time + miss rate * 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

# Impactul Miss-urilor

- 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-urile 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

# De ce ne Intereseaza?

- 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 aloci 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