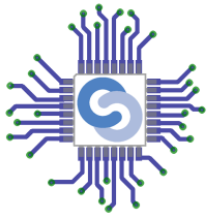


Calculatoare Numerice (2)

- Cursul 3 -

Memoria Cache (2)

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

Comic of the Day

http://dilbert.com/strips/comic/1995-11-17/

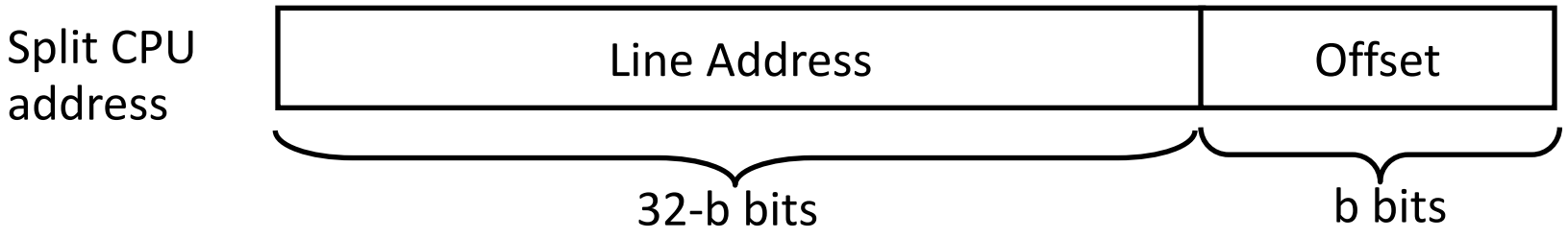
Din episodul anterior

- RAM Dinamic (DRAM) este principalul tip de memorie principală folosită în ziua de azi
  - Stochează valorile binare ca tensiuni pe condensatoare mici, care au nevoie de refresh (de aici partea de dinamic)
  - Acces lent, în mai mulți pași: precharge, read row, read column
- RAM Static (SRAM) este mai rapid dar mai scump
  - Folosit pentru a construi memoria cache
- Cache-ul conține un set de valori în memoria rapidă (SRAM) aproape de procesor
  - Trebuie să avem un algoritm de căutare a valorilor din cache și o politică de înlocuire pentru a face loc pentru locațiile nou accesate
- Memoriile cache exploatează două forme de predictibilitate în adresarea memoriei
  - Localitate temporală – aceeași locație accesată de mai multe ori
  - Localitate spațială – mai multe accese la locațiile învecinate

Dimensiunea liniilor și localitatea spațială

O linie este unitatea de transfer dintre cache și memoria principală

4 word line, b=2



2^b = line size *a.k.a* line size (in bytes)

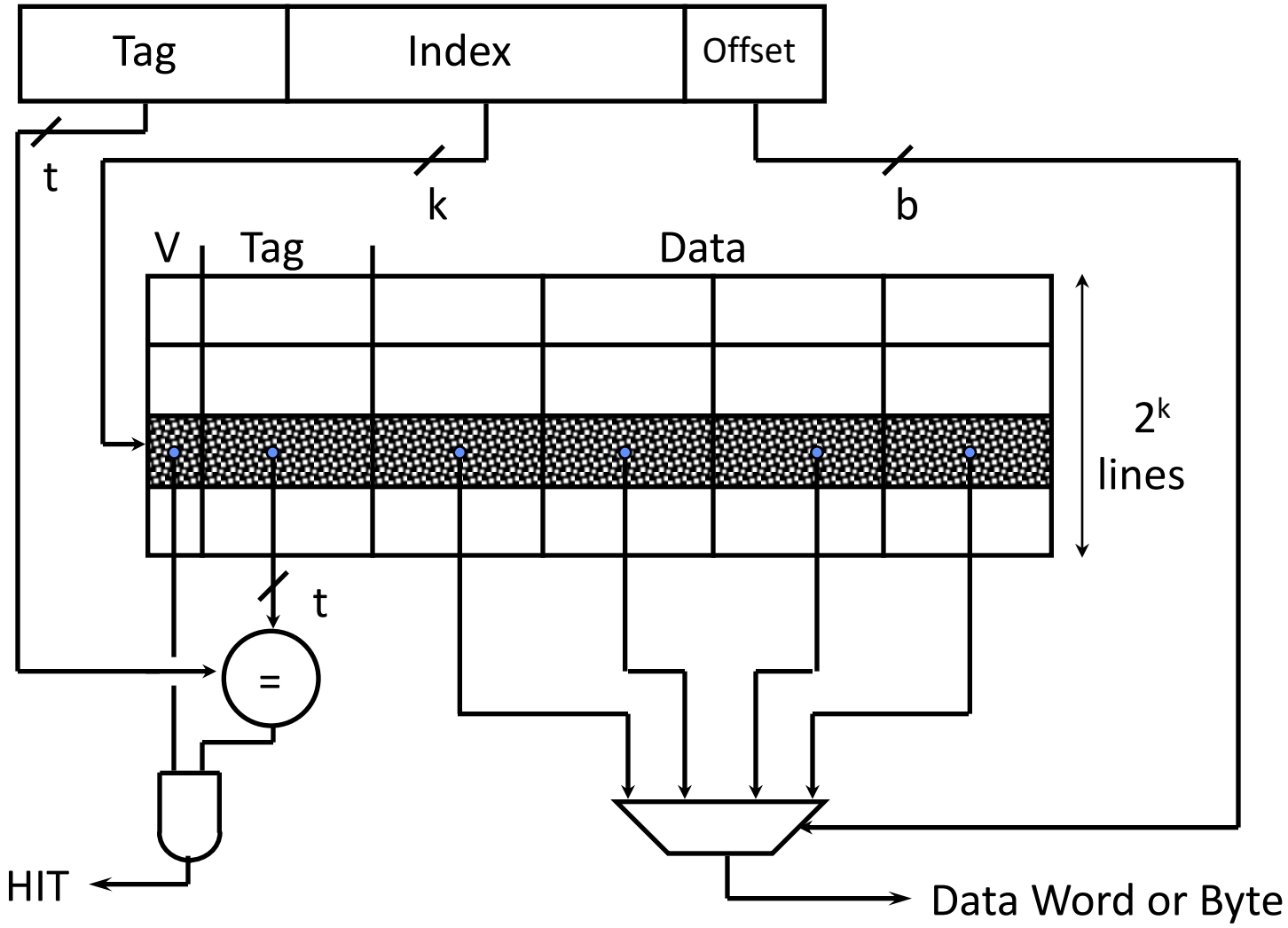
O linie de dimensiuni mari are avantaje din punct de vedere hardware

- overhead mai mic la tag-uri
- exploatează tranferurile în rafală din DRAM
- exploatează transferurile în rafală pe magistralele de date

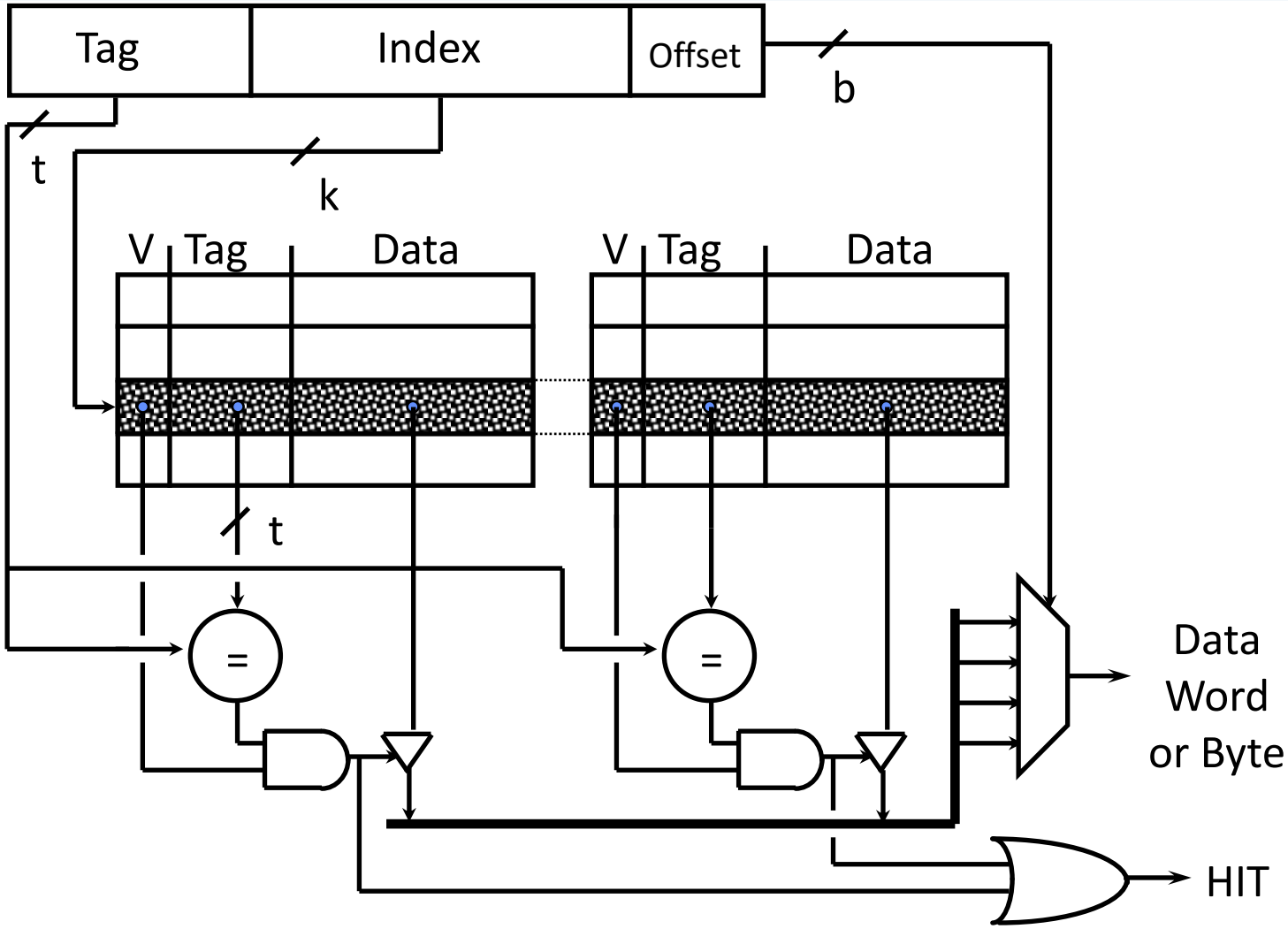
Care sunt dezavantajele unei dimensiuni mari pentru liniile de cache?

Mai puține linii => mai multe conflicte. Poate să irosească lățime de bandă

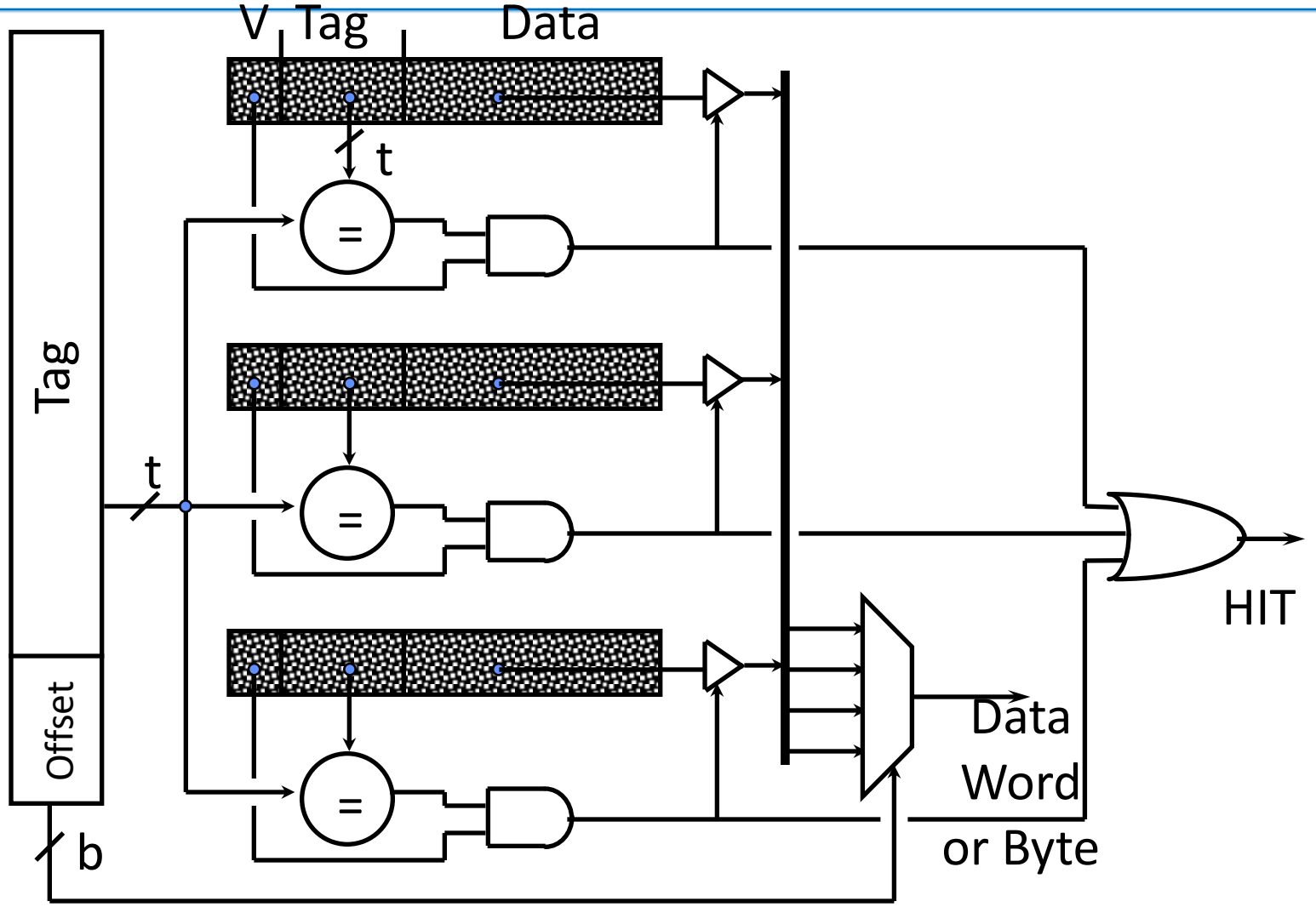
Cache mapat direct



2-Way Set-Associative Cache



Fully Associative Cache



Politica de înlocuire

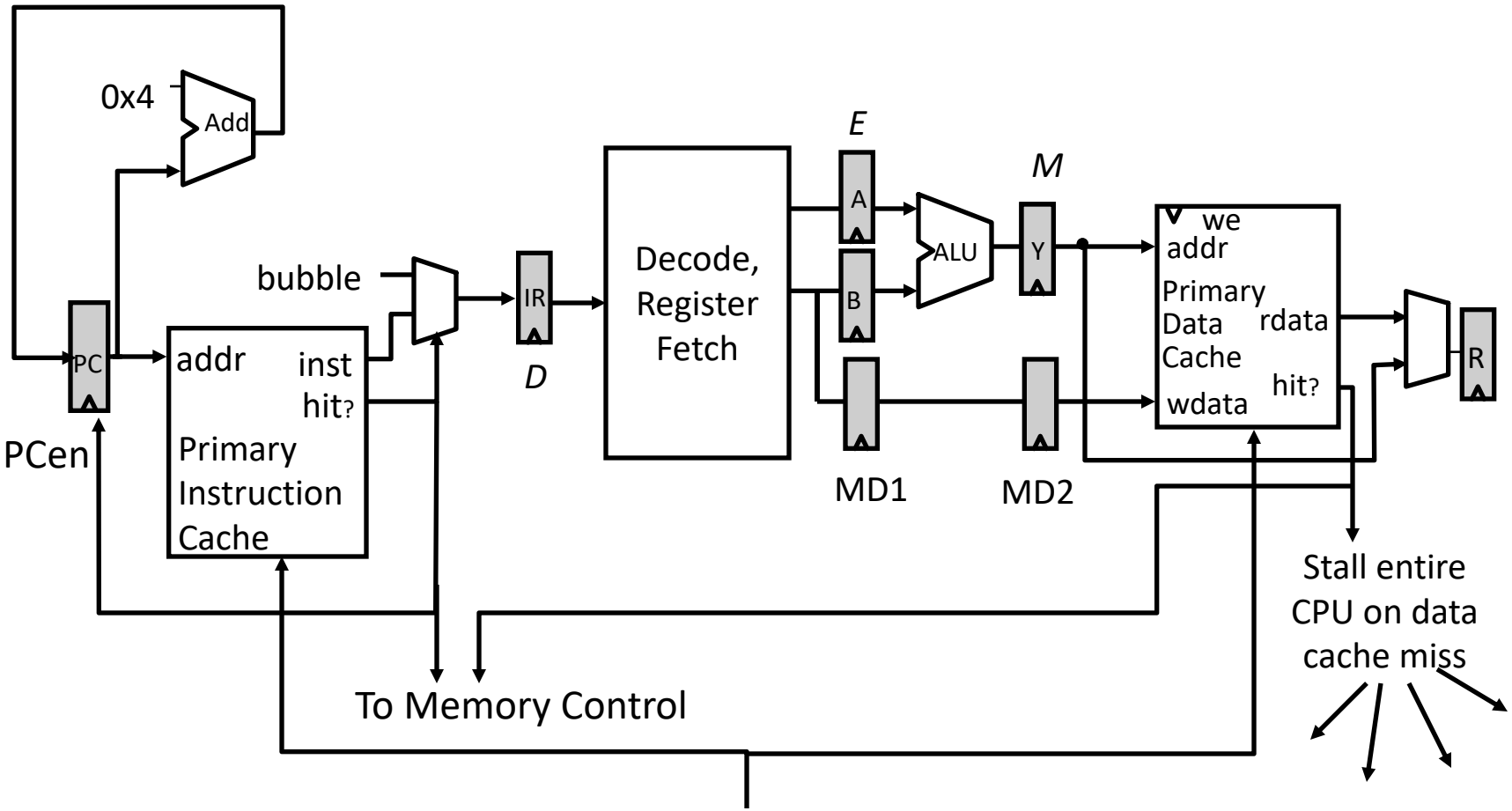
Într-un cache asociativ, care linie dintr-un set ar trebui invalidată atunci când setul se umple?

- Random
- Least-Recently Used (LRU)
  - Starea unui cache LRU trebuie actualizată la fiecare acces
  - Implementare fezabilă doar pentru seturi mici (2-way)
  - Arbore binar Pseudo-LRU folosit pentru 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - Folosit în cache-urile complet-asociative
- Not-Most-Recently Used (NMRU)
  - FIFO, cu excepția liniei utilizate cel mai recent

E un efect de ordin secundar. De ce?

O linie este înlocuită doar la un cache miss

Interacțiunea Cache-CPU
(5-stage pipeline)



Îmbunătățirea performanțelor memoriei cache

Average memory access time (AMAT) =

Hit time + Miss rate x Miss penalty

Pentru a îmbunătăți performanța:

- reducerea hit time
- reducerea miss rate
- reducerea miss penalty

Care este cel mai bun design cache pentru o b.a. Cu 5 etape?

Cel mai mare cache care nu mărește hit-time mai mult de un ciclu (approx 8-32KB în tehnologia modernă)

[probleme de design mai mari atunci când avem de-a face cu b.a. mai mari sau procesoare superscalare]

Cauzele pentru cache miss: The 3 C’s

Compulsory: prima referință la o linie (a.k.a. cold start misses)

- *Miss-uri care se petrec și pentru un cache de dimensiuni infinite*

Capacity: cache prea mic pentru a ține toate datele necesare unui program

- *Miss-uri care se petrec și dacă avem o politică perfectă de înlocuire a liniilor din cache*

Conflict: miss-uri care se petrec datorită coliziunilor datorate strategiei de amplasare a liniilor

- *Miss-uri care nu ar apare dacă am avea o asociativitate completă*

Efectele parametrilor cache asupra performanței

- Cache de dimensiuni mari
  + Reduce miss-urile de capacitate și conflict
  - Crește hit time

- Asociativitate mărită
  + Reduce conflict misses
  - Poate să marească hit time

- Linii de dimensiuni mai mari
  + Reduce miss-urile obligatorii și de capacitate (la reload)
  - Crește miss-urile de conflict și penalizările pentru un miss

Exemplul 1 performanța cache

- Un program are 2000 de operații load și store
- 1250 de date sunt aduse de aceste operații în cache
- Restul sunt luate din alte locații de memorie din afara cache

- **Care este rata de hit și miss pentru cache?**

  Hit Rate = 1250/2000 **= 0.625**

  Miss Rate = 750/2000 = **0.375** = 1 – Hit Rate

Exemplul 2 performanța cache

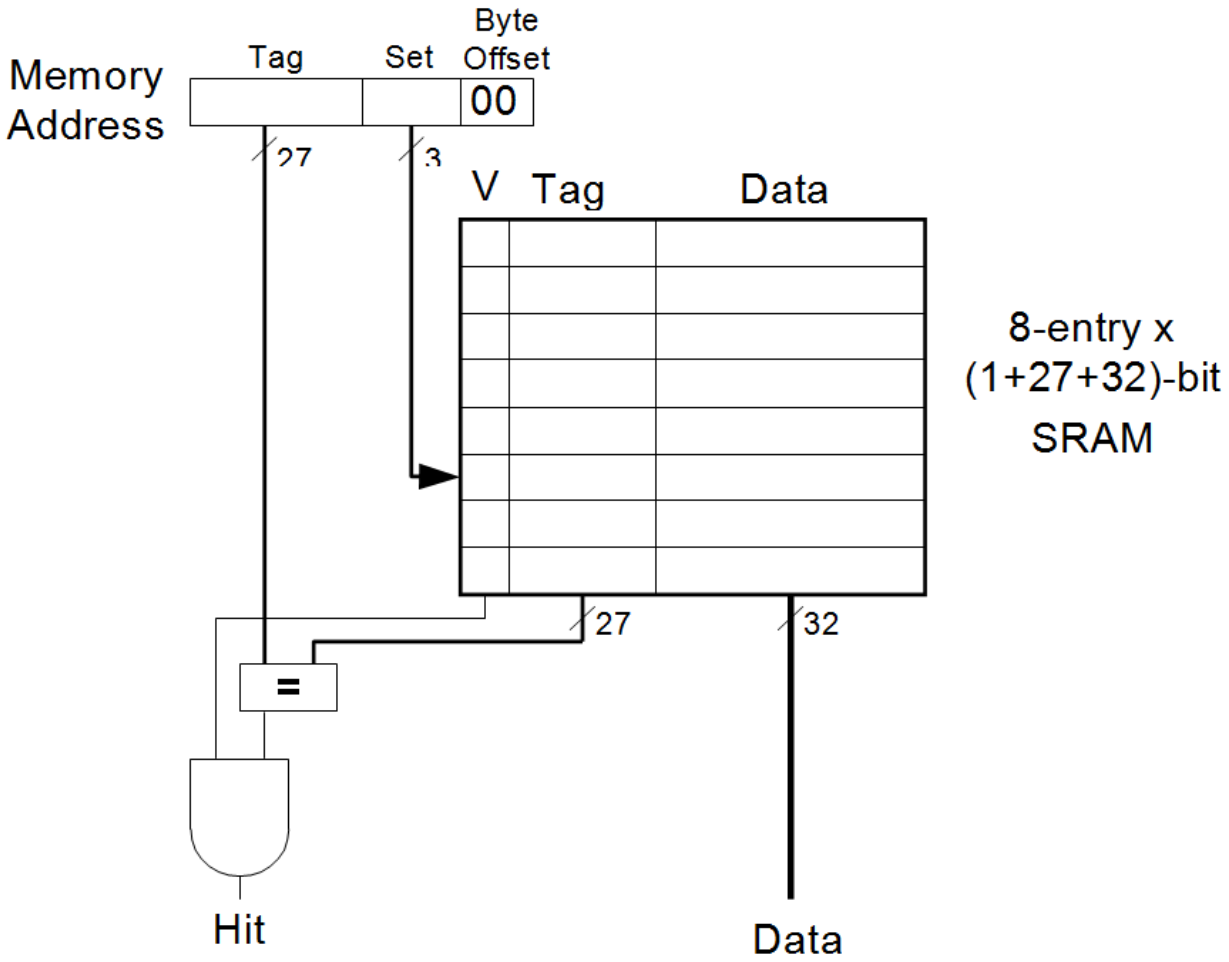
Presupunem că procesorul are două niveluri ierarhice de memorie: cache și mem. principală

- t_{cache} = 1 ciclu, t_{MM} = 100 cicli

Care este valoarea AMAT pentru exemplul 1?

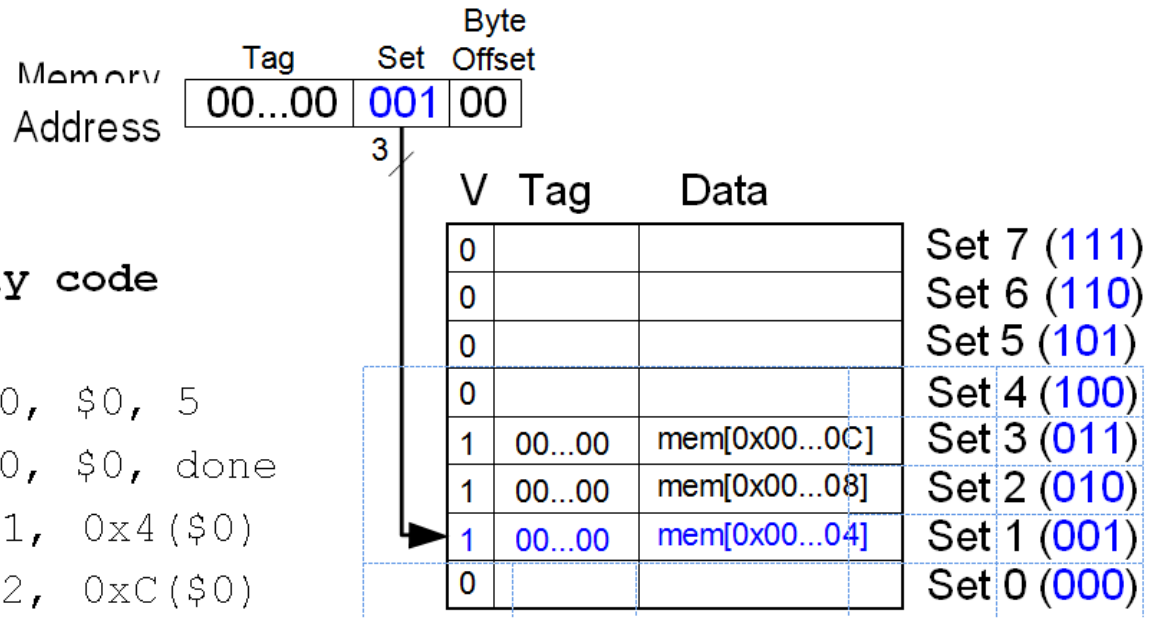
AMAT = *t*cache + *MR*cache(*tMM*)
= [1 + 0.375(100)] cicli
= **38.5 cicli**

Cache mapat direct



Cache mapat direct – performanță

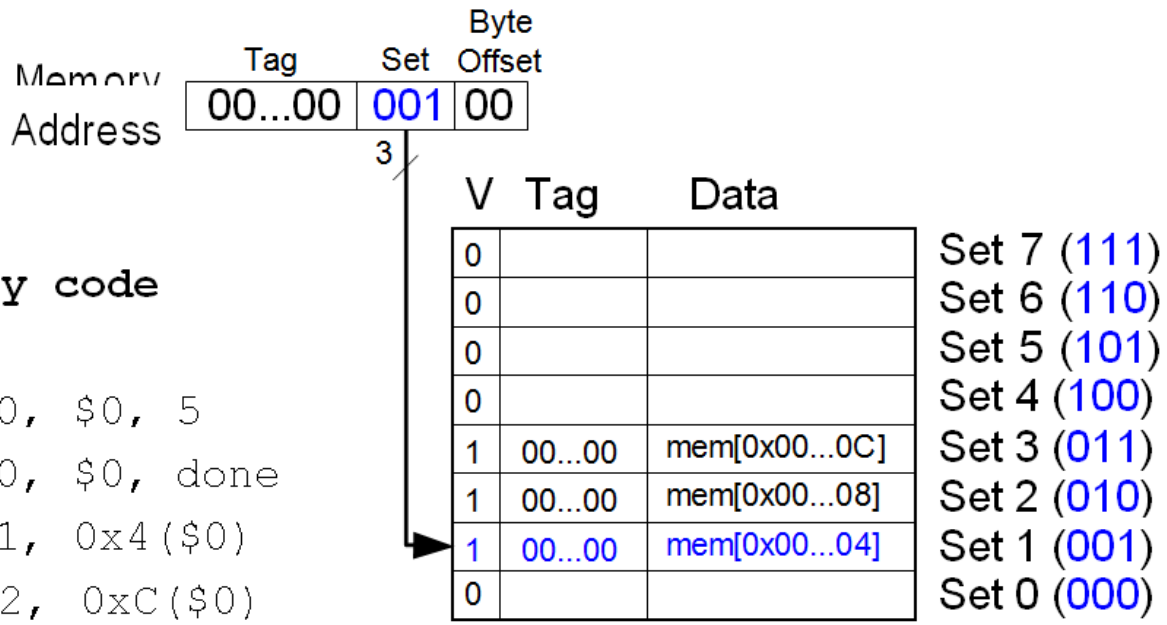
```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```



Miss Rate = ?

Cache mapat direct – performanță

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```



Miss Rate = 3/15

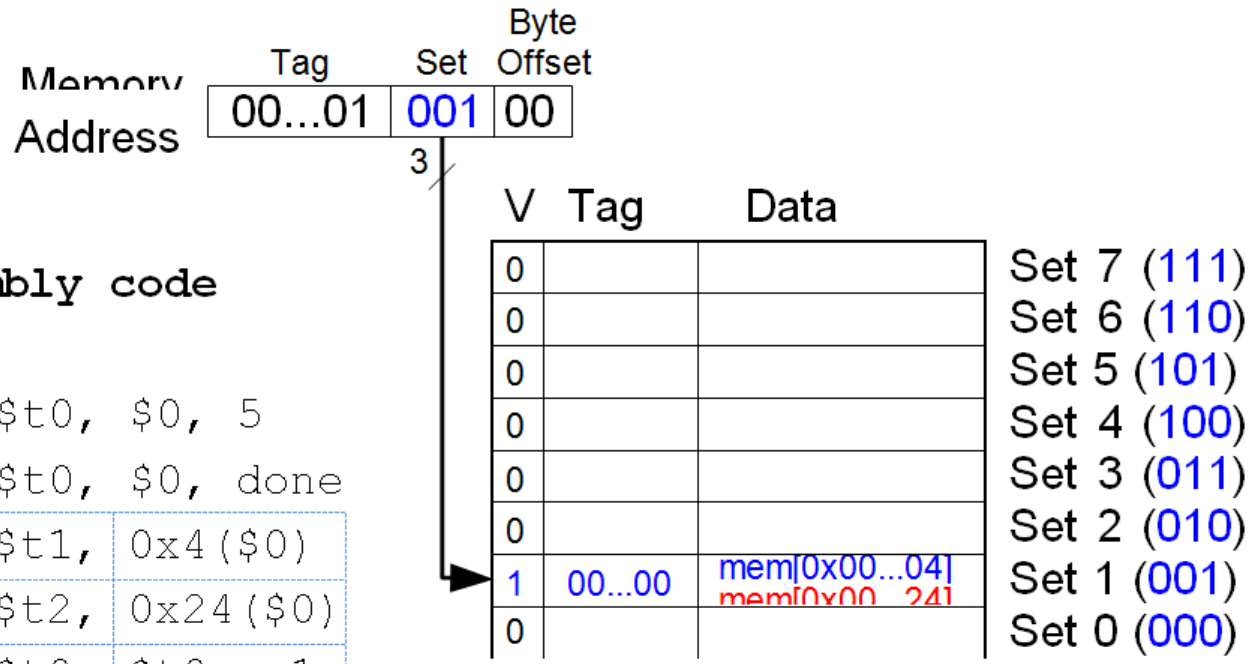
= 20%

Temporal Locality

Compulsory Misses

Cache mapat direct – conflict miss

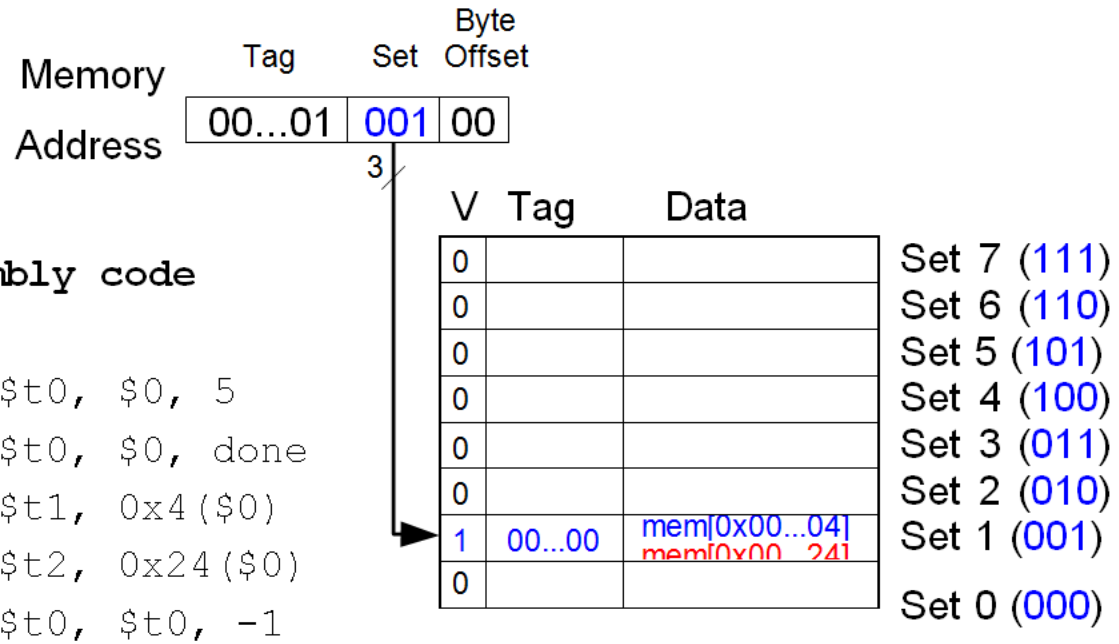
```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```



Miss Rate = ?

Cache mapat direct – conflict miss

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

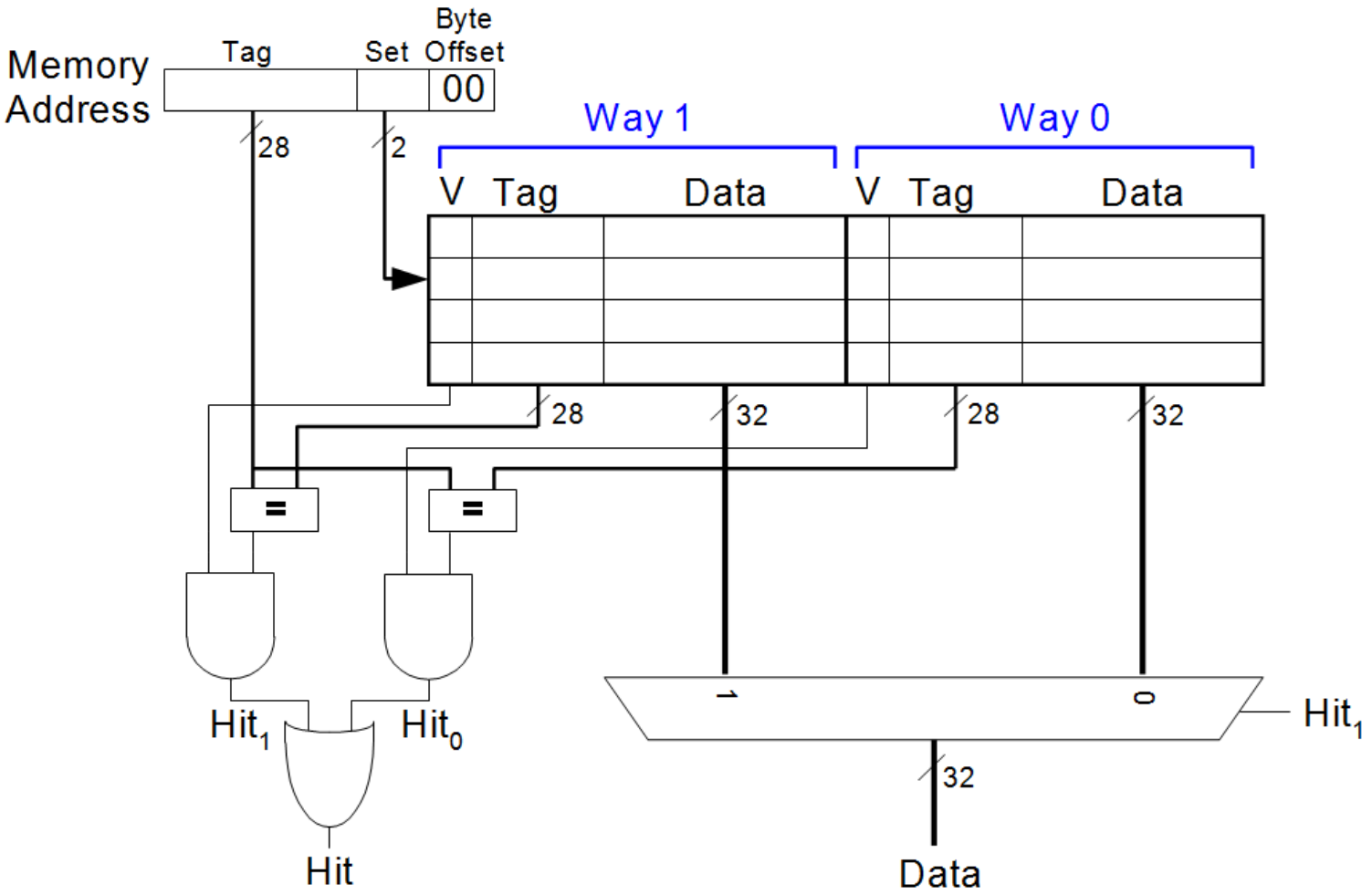


Miss Rate = 10/10

= 100%

Conflict Misses

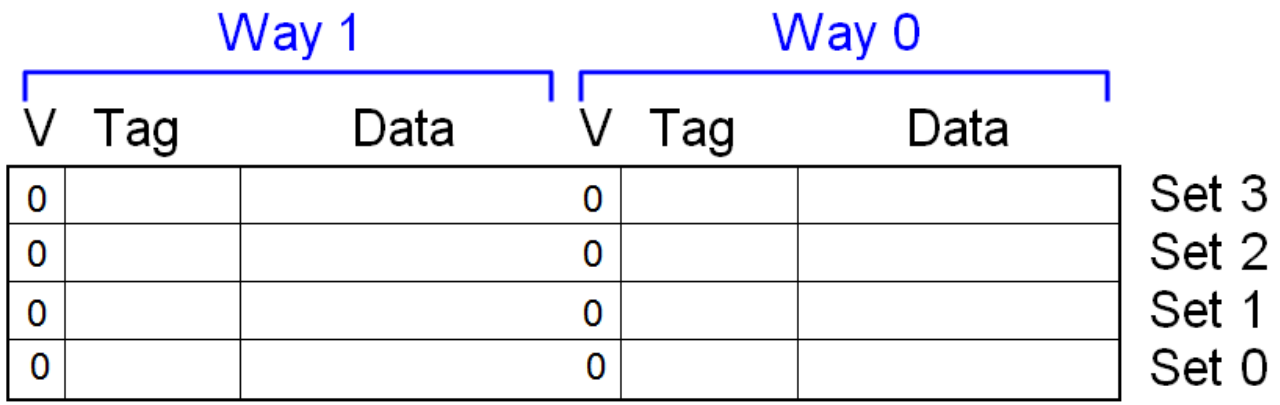
N-Way Set Associative Cache



N-Way Set Associative Cache

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = ?



N-Way Set Associative Cache

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

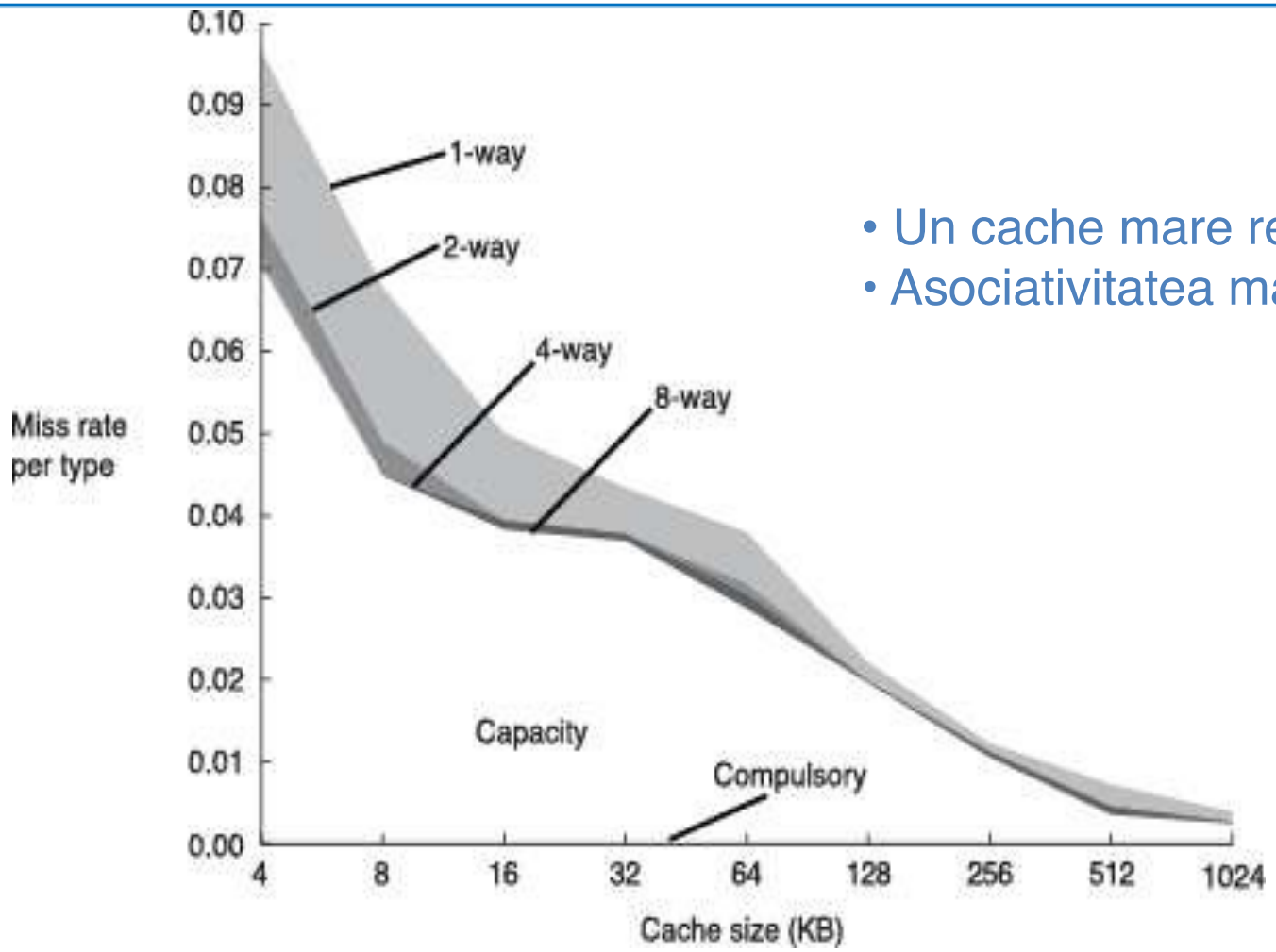
Miss Rate = 2/10

= 20%

Associativity reduces conflict misses

	Way 1			Way 0		
	V	Tag	Data	V	Tag	Data
Set 3	0			0		
Set 2	0			0		
Set 1	1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
Set 0	0			0		

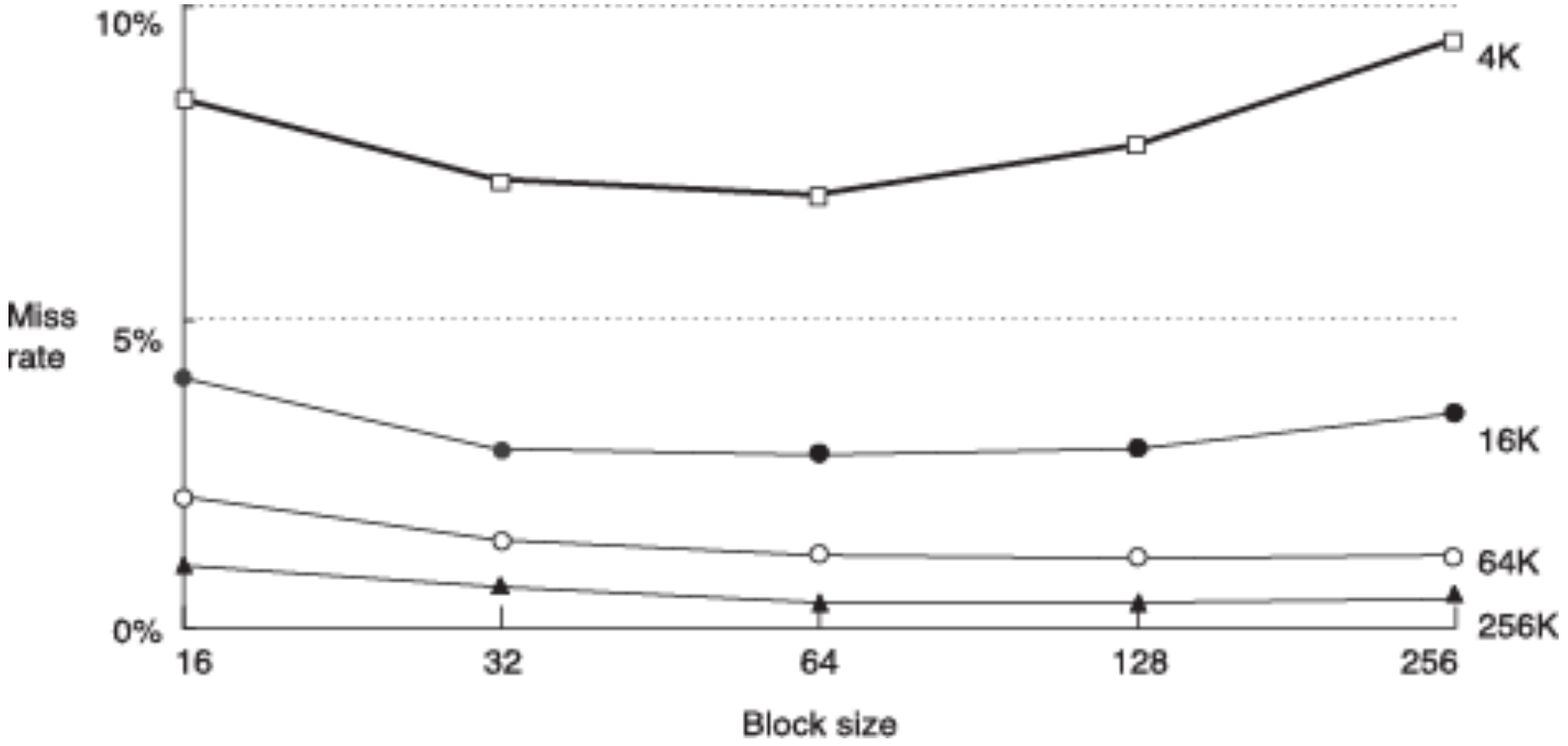
Tendințe miss-rate pentru N-way cache



- Un cache mare reduce rata de miss de capacitate
- Asociativitatea mai mare reduce rata de miss de conflict

Din Patterson & Hennessy, *Computer Architecture: A Quantitative Approach,* 2011

Tendințe miss-rate pentru N-way cache



- Blocurile mari reduc rata compulsory miss
- Dar, măresc rata conflict miss

Algoritmi pentru write-back

- Cache hit:
  - ***write through***: orice modificare în cache este actualizată automat și în RAM
    - De obicei mai mult trafic, dar duce la o implementare mai simplă a b.a. și a memoriei cache
  - ***write back***: scrie numai în cache, memoria este actualizată doar când linia este invalidată
    - Fiecare linie are un flag de ”dirty” care marchează liniile ce trebuiesc actualizate în RAM – micșorează traficul
    - Trebuie să permită 0, 1 sau 2 accese la memorie pentru fiecare load/store
- Cache miss:
  - ***no write allocate***: scrie doar înapoi în RAM
  - ***write allocate*** (aka fetch on write): face fetch în cache
- Combinații întâlnite:
  - write through, fără write allocate
  - write back împreună cu write allocate

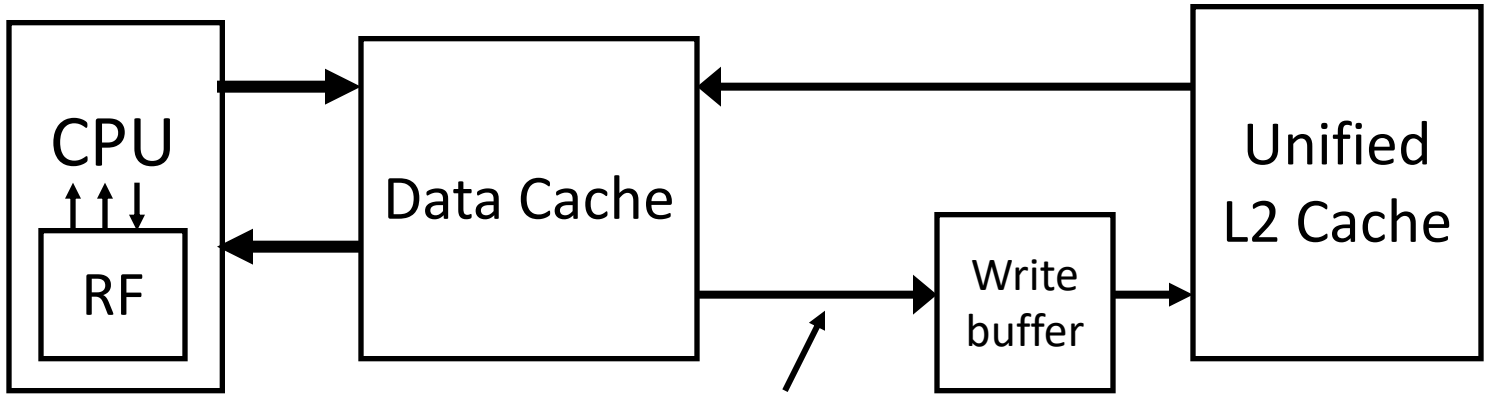
Reducerea timpului la Write Hit

Problemă: Scrierile durează doi cicli, unul pentru verificarea tag-urilor și altul pentru scrierea datelor la un hit.

Soluții:

- Proiectăm un circuit RAM care să permită o citire și o scriere într-un singur ciclu de ceas, reface vechea valoare dacă avem un tag miss
- Fully-associative (CAM Tag) caches: Linia este activată doar dacă avem un hit
- Pipelined writes: Ține datele ce trebuiesc scrise într-un singur buffer, în afara memoriei cache. Scrie datele în timpul ciclului de tag check

Write Buffering pentru reducerea penalizărilor la Read



Evicted dirty lines for write back cache
OR
All writes in write through cache

Procesorul nu este blocat la o scriere și un read miss poate să acceseze mai repede decât un write memoria RAM

Problema: Buffer-ul pentru write poate să conțină o valoare actualizată a locației cerute de un read miss

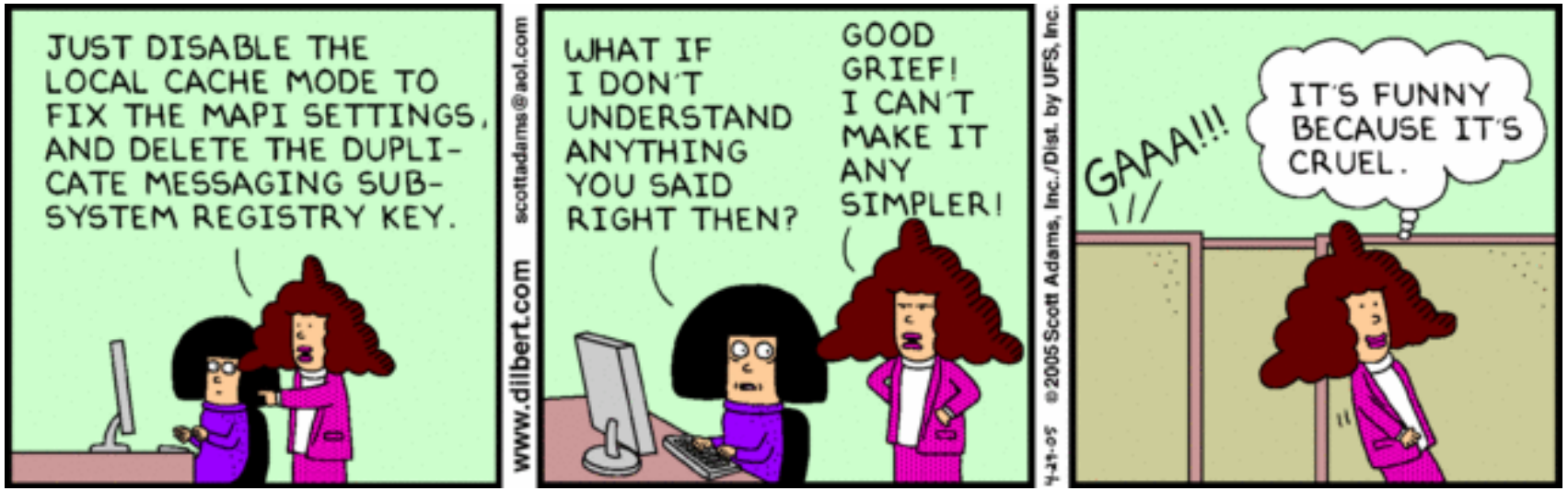
Soluție simplă: la un read miss, așteaptă să se golească write buffer.

Soluție mai rapidă: Verifică adresele din write buffer și compară-le cu cele ale liniilor pentru care avem read miss. Dacă nu coincid, lasă read miss să acceseze memoria RAM; dacă nu, întoarce valoarea conținută în write buffer.

Reducerea întârzierilor de parcurgere prin indexarea pe sub-blocuri

- **Problemă**: Etichetele pentru o linie sunt prea mari -> un overhead prea mare
  - Soluție simplă: Linii mai mari; va duce automat la creșterea penalizărilor ce apar la un miss.
- **Soluție**: Sub-block placement (aka sector cache)
  - Se adaugă un bit de validitate pentru unități mai mici de o linie, denumite sub-blocuri
  - La miss – citește doar un sub-bloc
  - *Dacă avem tag match, este cuvântul respectiv în cache?*

100	1		1		1		1	
300	1		1		0		0	
204	0		1		0		1	

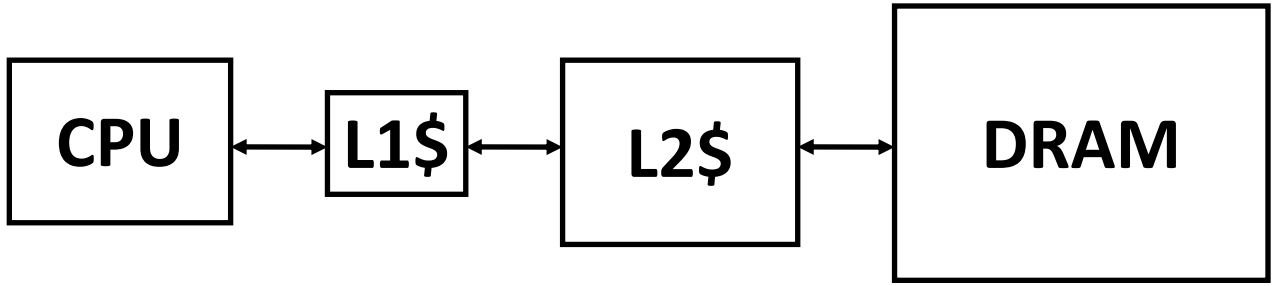


http://dilbert.com/strips/comic/2005-04-29/

Memoria Cache pe mai multe niveluri

Problemă: O memorie nu poate să fie și mare și rapidă

Soluție: Creștem dimensiunea cache cu fiecare nivel



Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions

Prezența unui L2 cache influențează design-ul L1 cache

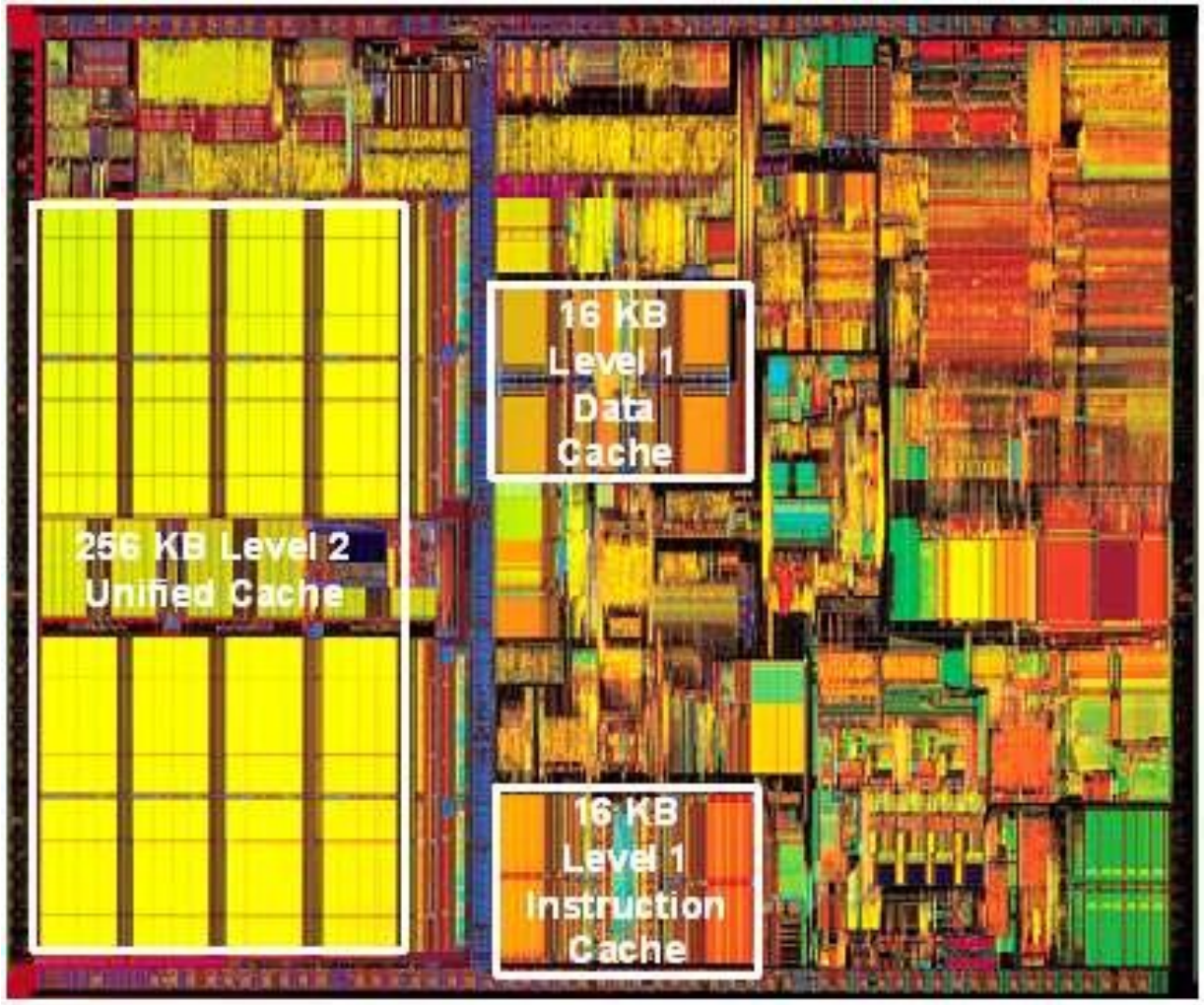
- Folosim un cache L1 dacă avem și un L2
  - Dimensiunea lui L1 va fi probabil mai mică -> va avea un miss rate mai mare, dar reducem drastic timpul de hit
  - Existența unui cache L2 reduce penalizarea la miss pentru L1
  - Reduce consumul mediu de energie pe acces
- Putem folosi L1 write-through (simplu de implementat) cu L2 on-chip
  - Cache L2 write-back L2 absoarbe traficul pentru write, nu accesează resurse din afara chip-ului
  - Simplifică problemele de coerență
  - Simplifică procesul de error recovery pentru L1 (poate să folosească doar biți de paritate și să reîncarce din L2 atunci când este detectată o eroare de paritate la citirea din L1)

Politica de incluziune

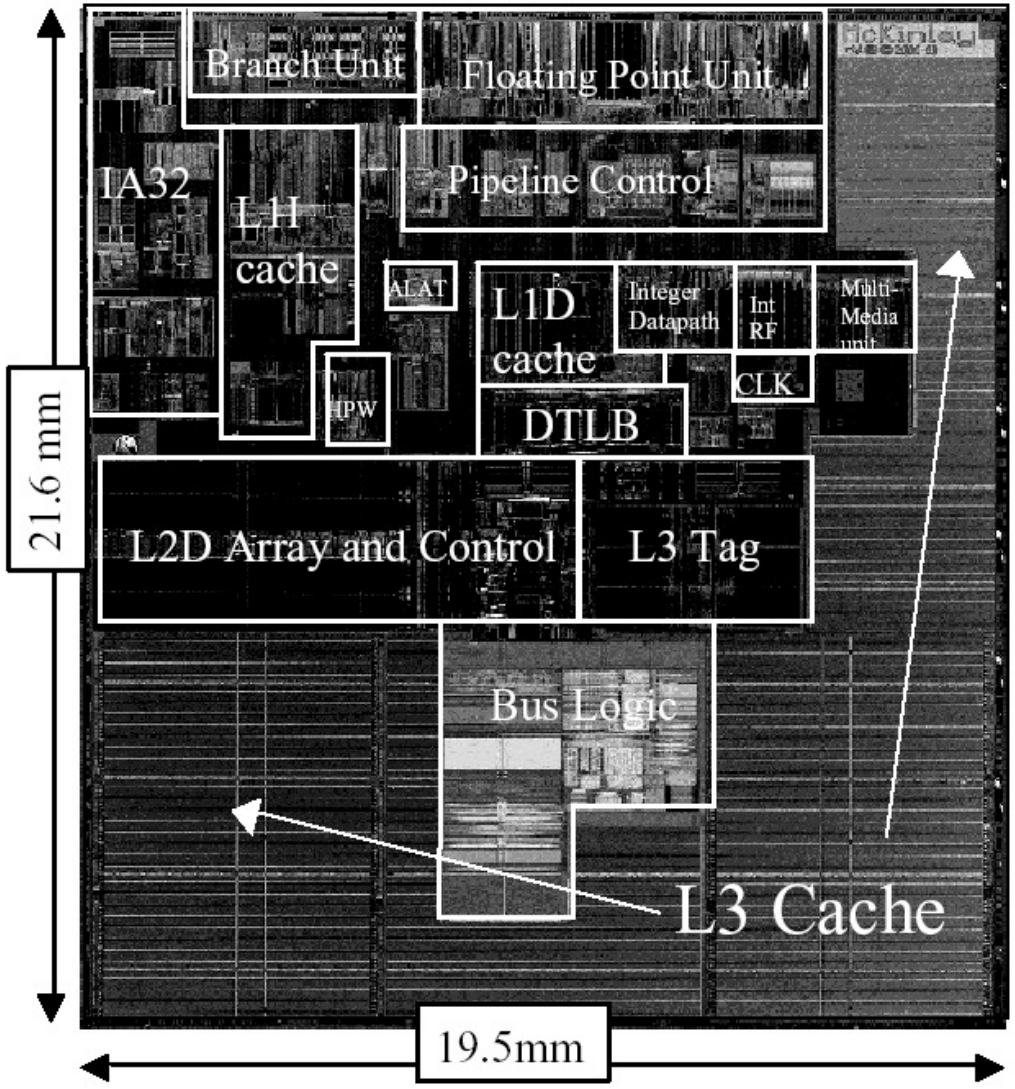
- Inclusive multilevel cache:
  - Cache-ul interior conține liniile care sunt prezente și în cache-ul exterior
  - Menținerea coerenței se face doar prin accese la cache-ul exterior (snooping)
- *Exclusive* multilevel caches:
  - Cache-ul interior conține liniile care nu sunt prezente în cel exterior
  - Interschimbăm liniile din cache interior-exterior la un miss
  - Folosit la AMD Athlon cu 64KB primary și 256KB secondary cache

Care variantă este mai bună?

Intel Pentium III die



Itanium-2 On-Chip Caches
(Intel/HP, 2002)



Level 1: 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

Level 2: 256KB, 4-way s.a, 128B line, quad-port (4 load or 4 store), five cycle latency

Level 3: 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

Power 7 On-Chip Caches [IBM 2009]

32KB L1 I$/core

32KB L1 D$/core

3-cycle latency

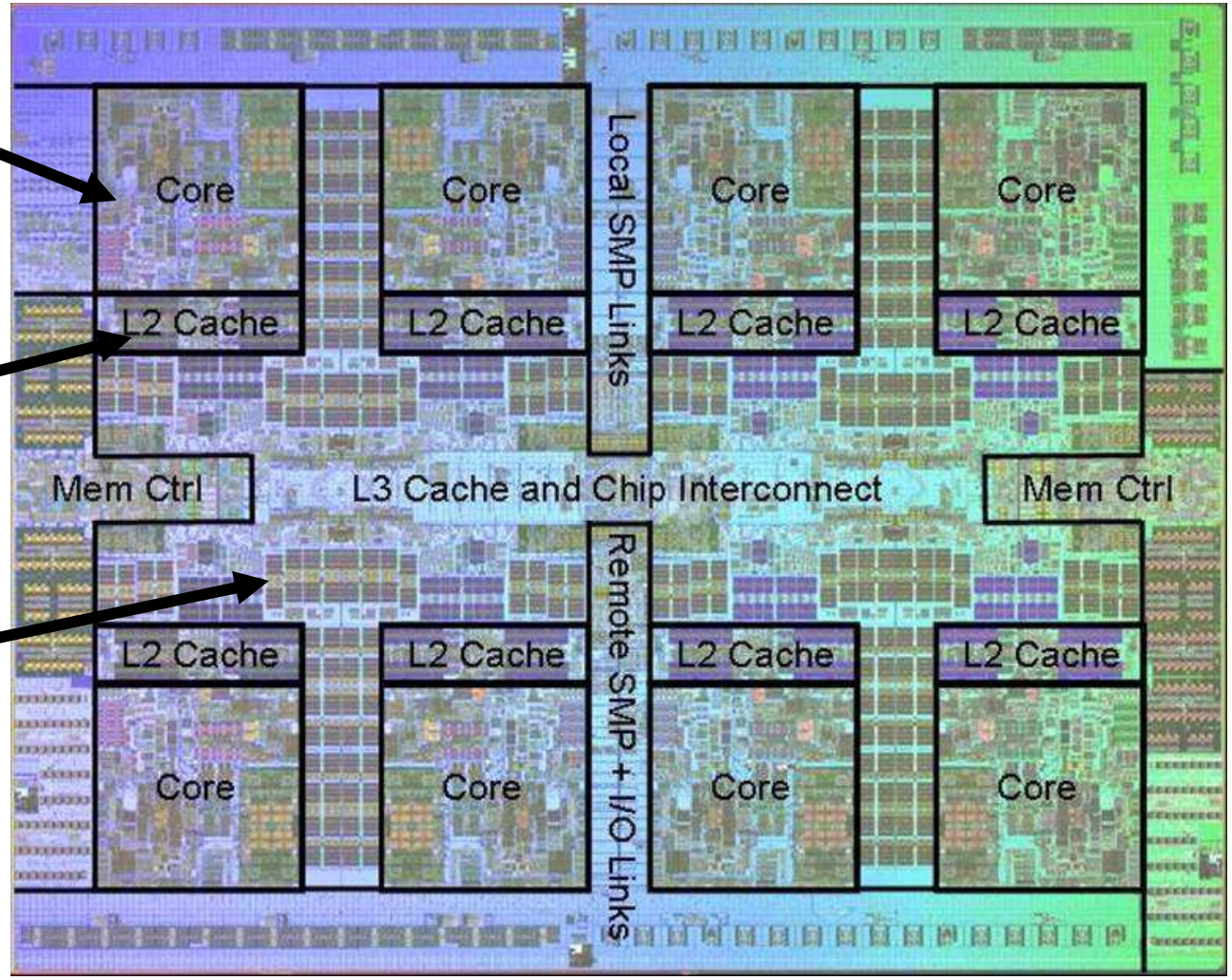
256KB Unified L2$/core

8-cycle latency

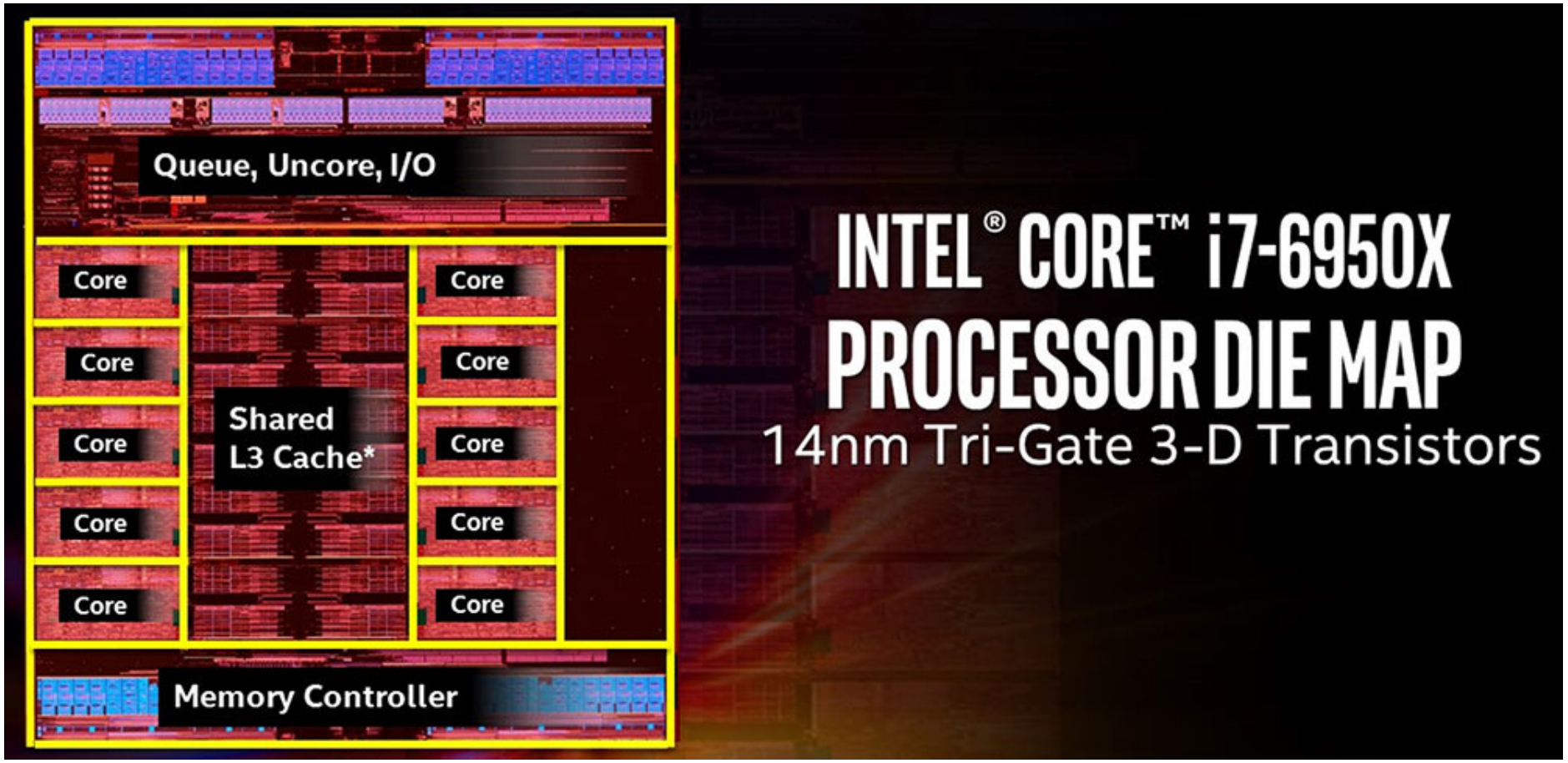
32MB Unified Shared L3$

Embedded DRAM (eDRAM)

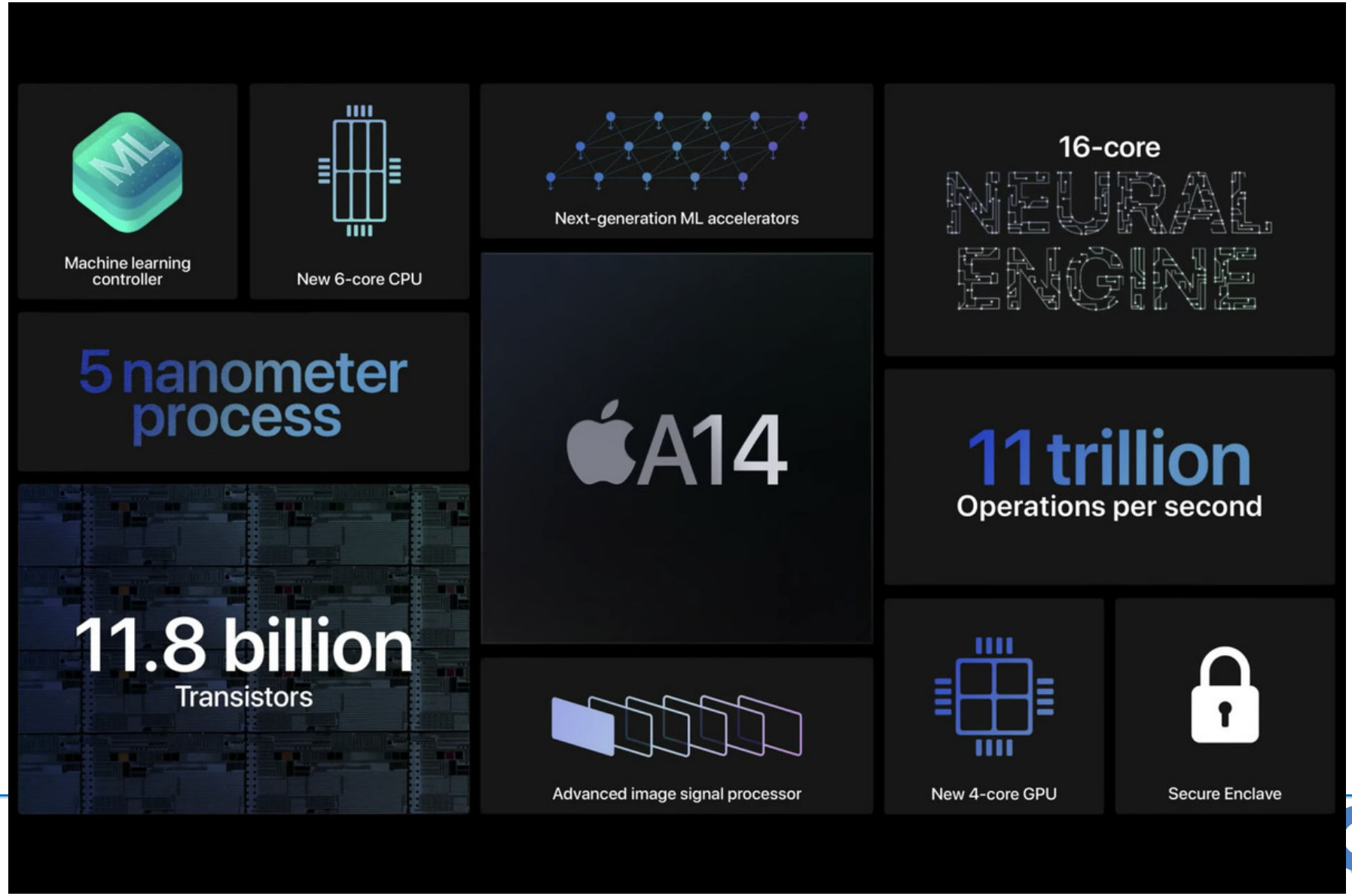
25-cycle latency to local slice



Core i7-6950X [Intel 2016]



Apple A14 [2020]



z196 Mainframe Caches [IBM 2010]

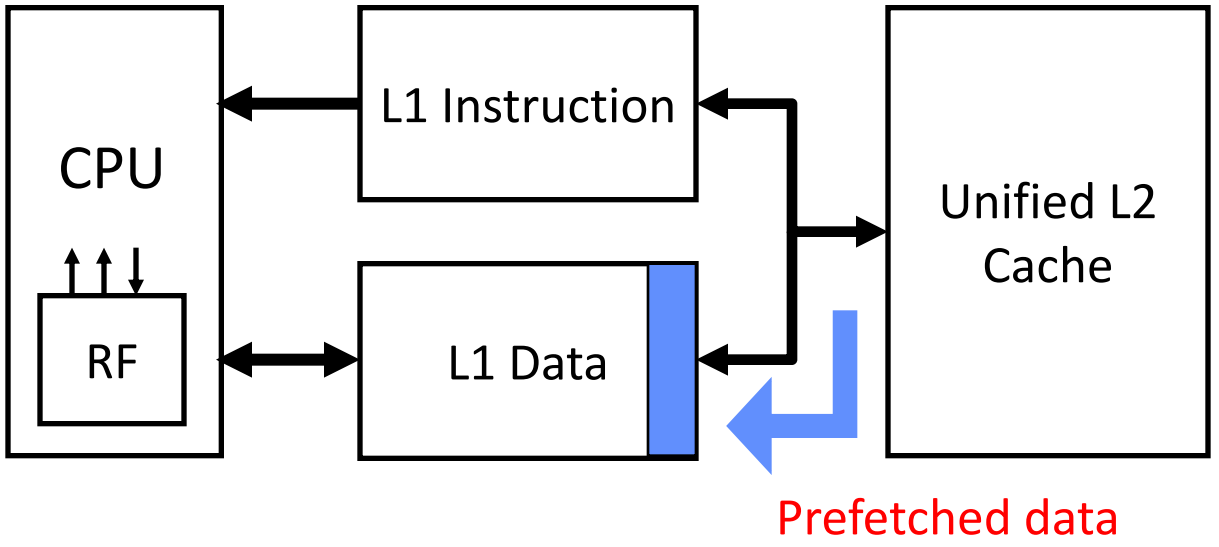
- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-$/core + 128KB D-$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)

Prefetching

- Speculează care vor fi următoarele accese la instrucțiuni și date și încarcă acele înregistrări în cache(uri)
  - Accesele la instrucțiuni sunt mai ușor de prezis decât accesele la date
- Variante de prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- Care tipuri de miss-uri sunt afectate de prefetching?

Probleme generate de prefetching

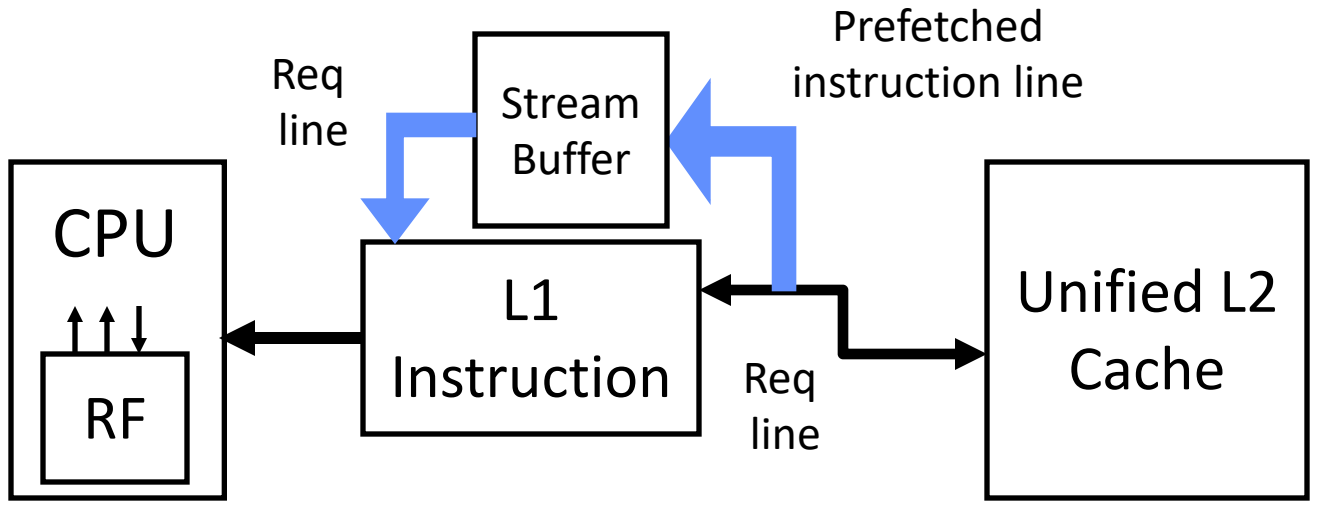
- Utilitate – ar trebui să mărească rata de hit
- Punctualitate – nu trebuie să ajungă prea târziu (dar nici prea devreme)
- Poluarea cache-ului și a lățimii de bandă



Hardware Instruction Prefetching

Prefetch al instrucțiunilor la Alpha AXP 21064

- Fetch la două linii pentru un miss; linia cerută (i) și linia imediat următoare (i+1)
- Linia cerută plasată în cache, următoarea linie într-un buffer special - instruction stream buffer
- Dacă miss în cache dar hit în stream buffer, mută linia din stream buffer în cache și prefetch la următoarea linie (i+2)



Hardware Data Prefetching

- Prefetch-on-miss:
  - Prefetch la b + 1 dacă am miss b
- One-Block Lookahead (OBL)
  - Initializează prefetch pentru blocul b + 1 când accesăm blocul b
  - De ce e diferit față de dublarea dimensiunii blocurilor?
  - Poate fi extins la N-block lookahead
- Strided prefetch
  - Dacă există o secvență de accese de tipul b, b+N, b+2N, atunci prefetch la linia b+3N etc.

- Exemplu: IBM Power 5 [2003] implementează opt căi independente de strided prefetch per procesor și face prefetch cu 12 linii înainte pentru un acces dat

Software Prefetching

```
for(i=0; i < N; i++) {
    prefetch( &a[i + 1] );
    prefetch( &b[i + 1] );
    SUM = SUM + a[i] * b[i];
 }
```

Software Prefetching Issues

- Cea mai mare problemă este timing-ul, nu predictibilitatea
  - Dacă faci prefetch foarte aproape de momentul în care ai nevoie de date, s-ar putea să fie prea târziu
  - Prefetch prea devreme – poluezi cache-ul
  - Estimăm cât de mult timp e nevoie pentru a aduce datele în L1 pentru a face un prefetch exact
  - *De ce e greu de făcut asta?*

```
for(i=0; i < N; i++) {
    prefetch( &a[i + P] );
    prefetch( &b[i + P] );
    SUM = SUM + a[i] * b[i];
 }
```

Trebuie să luăm în considerare costul instrucțiunilor de prefetch

Optimizări de compilator

- Restructurarea codului afectează secvența de acces la date
  - Grupăm accesele de date împreună pentru a îmbunătăți localitatea spațială
  - Re-aranjăm accesele la date pentru a îmbunătăți localitatea temporală
- Prevenim intrările nedorite de date în cache
  - Folositor pentru variabilele care vor fi accesate o singură dată înainte de a fi înlocuite
  - Necesită un mecanism prin care software-ul să spună hardware-ului să nu facă data caching (“no-allocate” instruction hints sau page table bits)
- Invalidează datele care nu vor fi folosite niciodată
  - Un stream de date exploatează localitatea spațială, dar nu și cea temporală
  - Înlocuiește în locațiile ”moarte” din cache

Loop Interchange

```
for(j=0; j < N; j++) {
    for(i=0; i < M; i++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

```
for(i=0; i < M; i++) {
    for(j=0; j < N; j++) {
        x[i][j] = 2 * x[i][j];
    }
}
```

Ce fel de localitate exploatează schimbarea de sus?

Loop Fusion

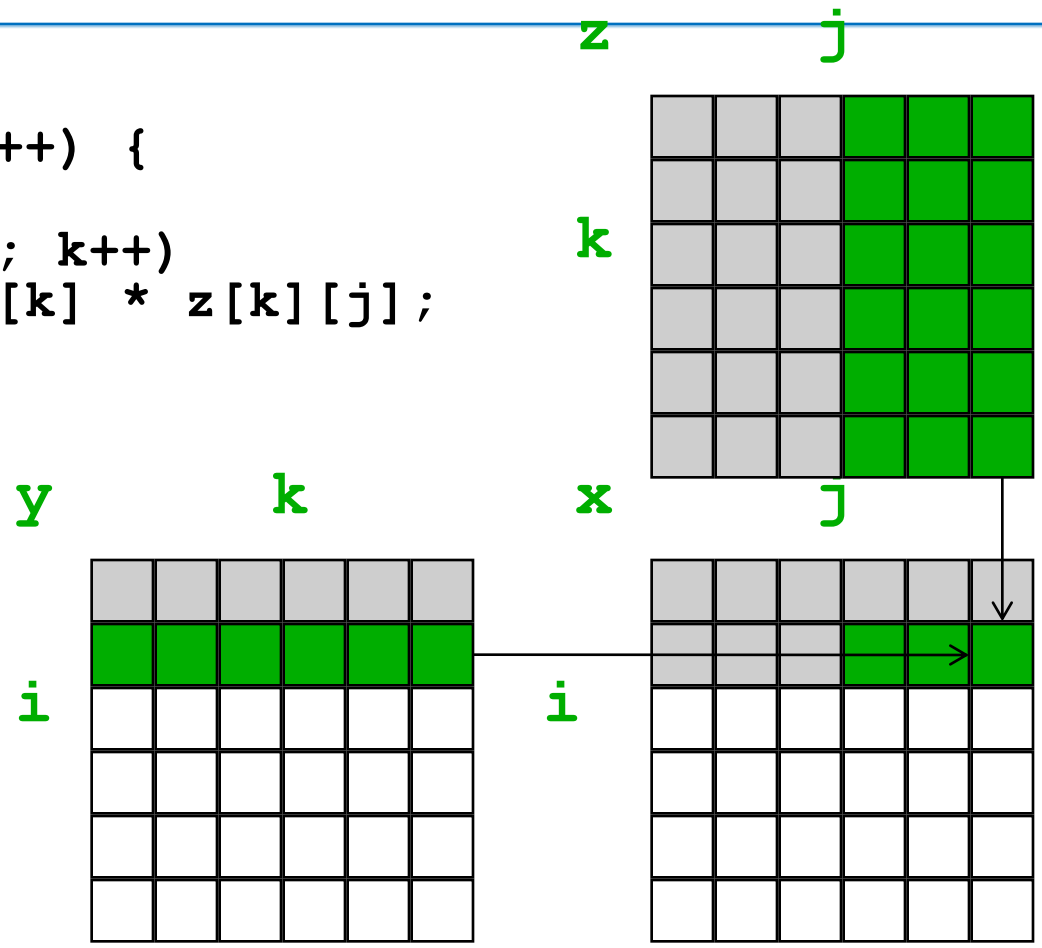
```
for(i=0; i < N; i++)
      a[i] = b[i] * c[i];

for(i=0; i < N; i++)
      d[i] = a[i] * c[i];
```

```
for(i=0; i < N; i++)
{
      a[i] = b[i] * c[i];
      d[i] = a[i] * c[i];
}
```

Dar schimbarea de sus?

Matrix Multiply, Naïve Code

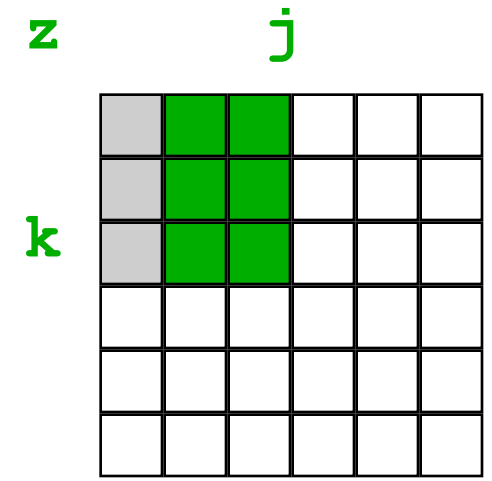
```
for(i=0; i < N; i++)
     for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
          r = r + y[i][k] * z[k][j];
        x[i][j] = r;
     }
```

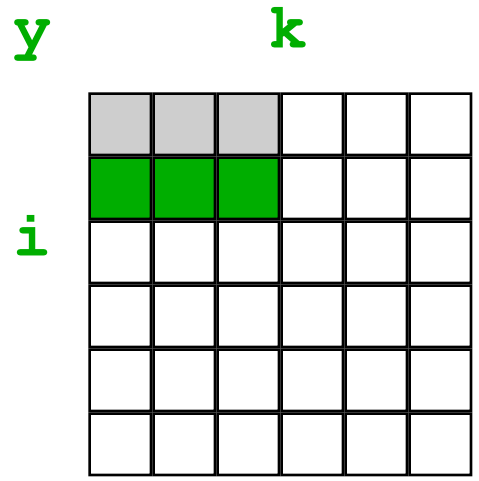


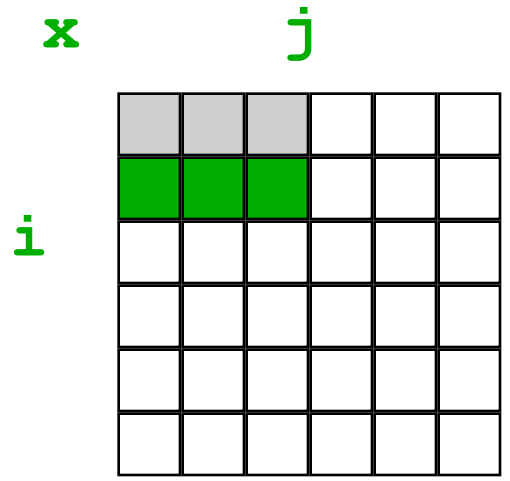
Not touched Old access New access

Matrix Multiply with Cache Tiling

```
for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
       for(i=0; i < N; i++)
           for(j=jj; j < min(jj+B,N); j++) {
              r = 0;
              for(k=kk; k < min(kk+B,N); k++)
                 r = r + y[i][k] * z[k][j];
              x[i][j] = x[i][j] + r;
           }
```







Ce tip de localitate este îmbunătățită?

Exemplul 1

Program A

```
struct DATA
{
    int a;
    int b;
    int c;
    int d;
};
DATA * pMyData;

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

Program B

```
struct DATA
{
    int a;
    int b;
};

DATA * pMyData;

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

~2X mai rapid!

Exemplul 2

Program C

```
struct DATA
{
    char a;
    int b;
    char c;
};

DATA * pMyData;

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

Program D

```
struct DATA
{
    int b;
    char a;
    char c;
};

DATA * pMyData;

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

60% mai rapid!

Exemplul 3

Program E	Program F

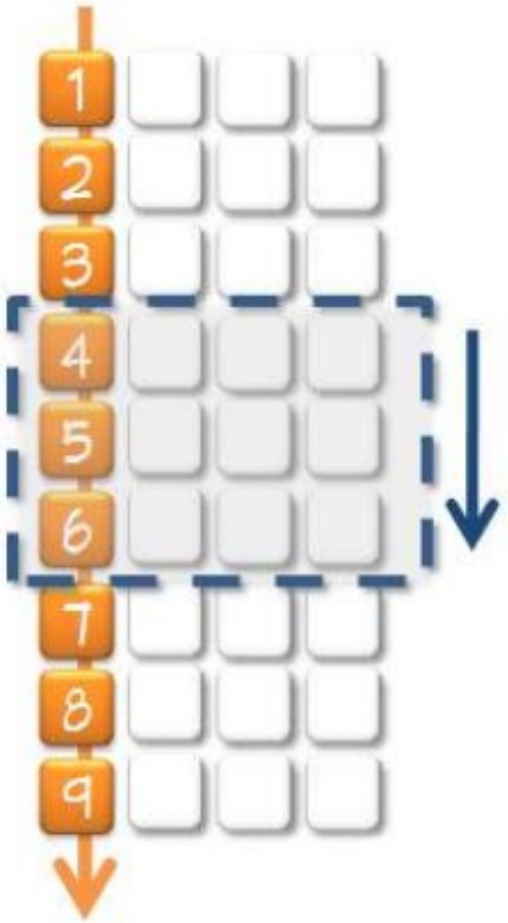
`char * p;`
 	
`p = new char[SIZE];`
 	
`for (long x=0; x<sRowSize; x++)`
`for (long y=0; y<nbRows; y++)`
`{`
`  p[x+y*sRowSize]++;`
`}`
	`char * p;`
	
	`p = new char[SIZE];`
	
	`for (long y=0; y<nbRows; y++)`
	`for (long x=0; x<sRowSize; x++)`
	`{`
	`  p[x+y*sRowSize]++;`
	`}`

de 4x mai rapid!

Exemplul 3

Program E

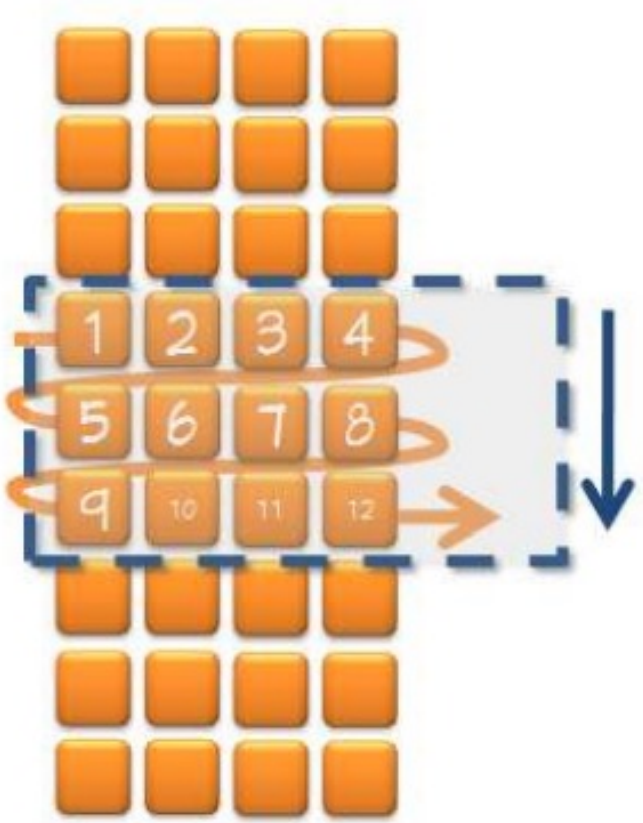
```
char * p;

p = new char[S

for (long x=0;
for (long y=0;
{
    p[x+y*sRow
}
```

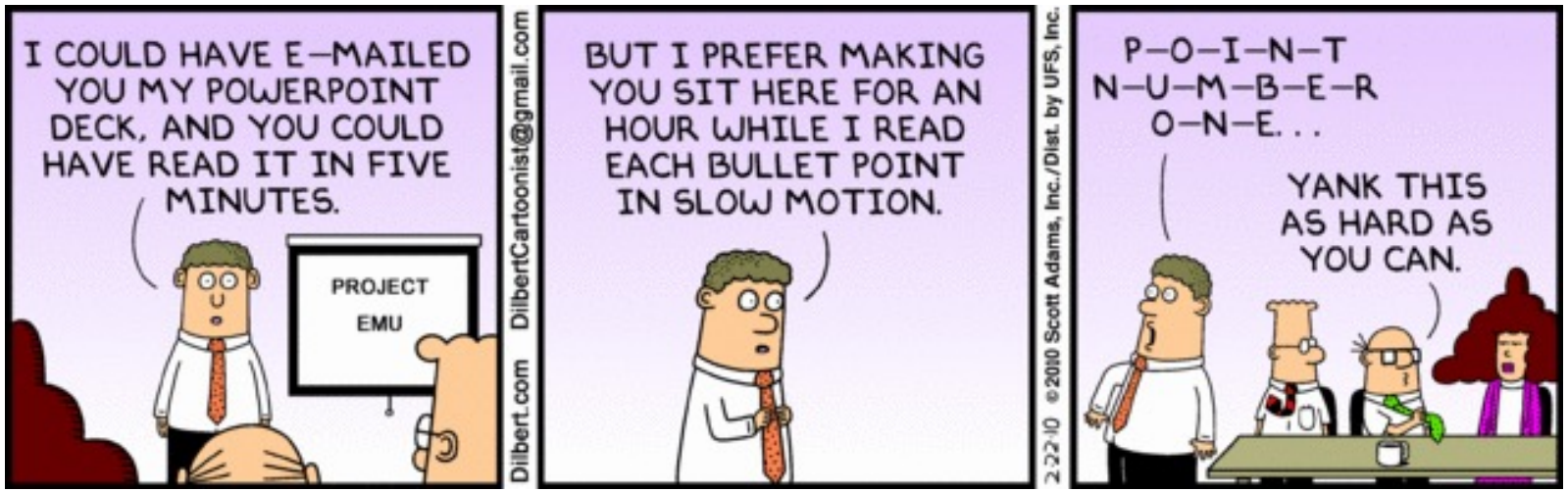


Program F

x mai
bid!

+)





http://dilbert.com/strips/comic/2010-02-22/

Acknowledgements

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

- MIT material derived from course 6.823
- UCB material derived from course CS252