Chapter 2 (cont)

Instructor: Josep Torrellas

CS433

Improving Cache Performance

$$\text{Average mem access time} = \underset{\text{speed up}}{\text{hit time}} + \underset{\text{reduce}}{\text{miss rate}} * \underset{\text{reduce}}{\text{miss penalty}}$$

1. Reduce miss penalty: multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches
2. Reduce miss rate: larger block size, larger cache size, higher associativity, way prediction and pseudoassociativity, compiler optimizations
3. Reduce miss rate/penalty via parallelism: non-blocking caches, prefetching
4. Reduce the hit time: small and simple caches, avoiding address translation, pipelined cache access, and trace caches

Types of Misses

1. Compulsory : First access
2. Capacity : cache cannot contain all blocks
3. Conflict : too many blocks map into same set

How to compute them ?

How to Remove Misses

- Conflict : Increase associativity → expensive in H/W
  → slow processor clock
- Capacity : Enlarge cache → expensive
  → slow
- Compulsory : Enlarge line (block) size
  → may increase conflict

Larger Block Size

- Relation to miss rate
  - fewer compulsory misses (spatial locality)
  - increase conflict and capacity misses
- Relation to Miss penalty
  - increase it : miss penalty = a + b * c

    b = block size

$$\left.\begin{array}{l}\Rightarrow \text{high latency and high} \\ \quad \text{bandwidth}\end{array}\right\} \text{longer blocks}$$

usually 32 - 64 blocks

Larger Caches

Higher Associativity

- 8- way ≅ fully associative
- 2:1 cache rule of thumb :

  dir mapped size N ≅ 2-way set associative size N/2
- Increased assoc : + decreases misses

  - increases hit time

Second Level Caches

It is like making the cache faster and bigger at the same time .

$\rightarrow$ primary cache : small to match CPU speeds

$\rightarrow$secondary cache ; large to capture most accesses

$$\text{Avg} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} * \underline{\text{miss penalty}_{L1}}$$

$$\downarrow$$

$$\text{hit time}_{L2} + \underline{\text{miss rate}_{L2}} * \text{miss penalty}_{L2}$$

$$\downarrow$$

Of the left overs! “Local Miss rate”

For the secondary cache :

- Local miss rate : $\dfrac{\#\text{ misses L2}}{\#\text{ accesses L2}}$ high !
- Global Miss rate : $\dfrac{\#\text{ misses L2}}{\#\text{ accesses by CPU}} \longrightarrow$ more intuitive

Issues in Secondary Cache

- Does it lower the AMAT portion of the CPI ?
- How much does it cost ?

Note: SLC much larger than FLC (else miss rate no change)

- can use higher associativity for SLC
- can use longer blocks in SLC (not conflicts)
- Multilevel inclusion property :
  - desirable because of consistency
    - w/ I/o
    - w/ other caches in an MP

$\Big\}$ only check SLC

Optimizations

Ten advanced optimizations.

1. Reducing Hit Time with small/simple caches

→ Often determines the average memory access time

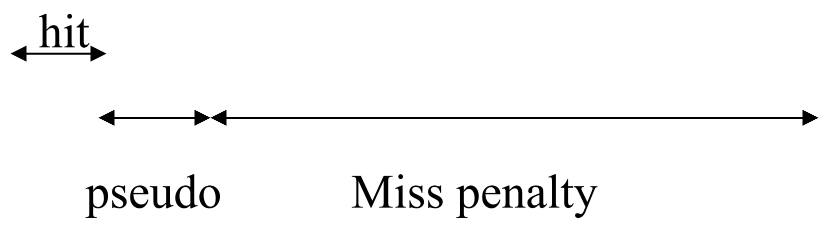
→ the smaller , the faster

→ needs to fit on chip

→ use dir-mapped cache

2. Way Prediction or Pseudo-assoc caches

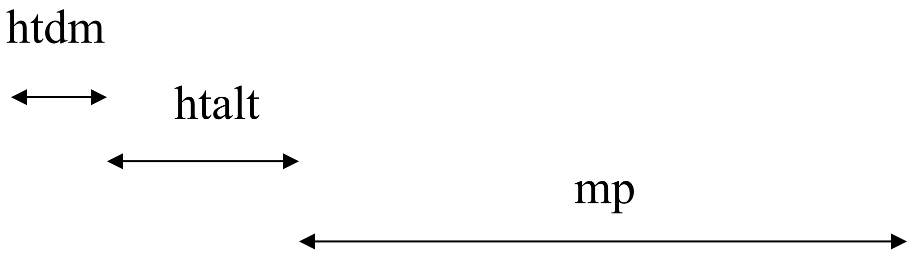
- hit is same
- if miss , before going to next level of mem hierarchy, another cache entry is checked (e.g invert the most significant bit of the index field)
- there is a $\left.\begin{array}{r}\text{fast}\\ \text{slow}\end{array}\right\}$ hit time $\begin{array}{l}\to \text{regular hit}\\ \to \text{pseudo hit}\end{array}$
- when pseudo hit $\longrightarrow$ swap entries



} Complicates pipelined CPU

Example for pseudo-assoc caches :

1) if



Avg = htdm + mr1 *mp = htdm + mr1 (htalt+mr2*mp)

Avg = htdm + mr1 *htalt + mr1*mr2*mp

More Optimizations

3) Pipelined cache access to increase cache bandwidth

4. Non Blocking Caches

- Suppose a dynamically scheduled processor → out of order completion : processor continues fetching instructions while waiting for the data cache to supply missed data

- Non blocking or lookup free cache : data supplied
  → hit under a miss
  → miss under a miss

What if miss on a line that is being requested?

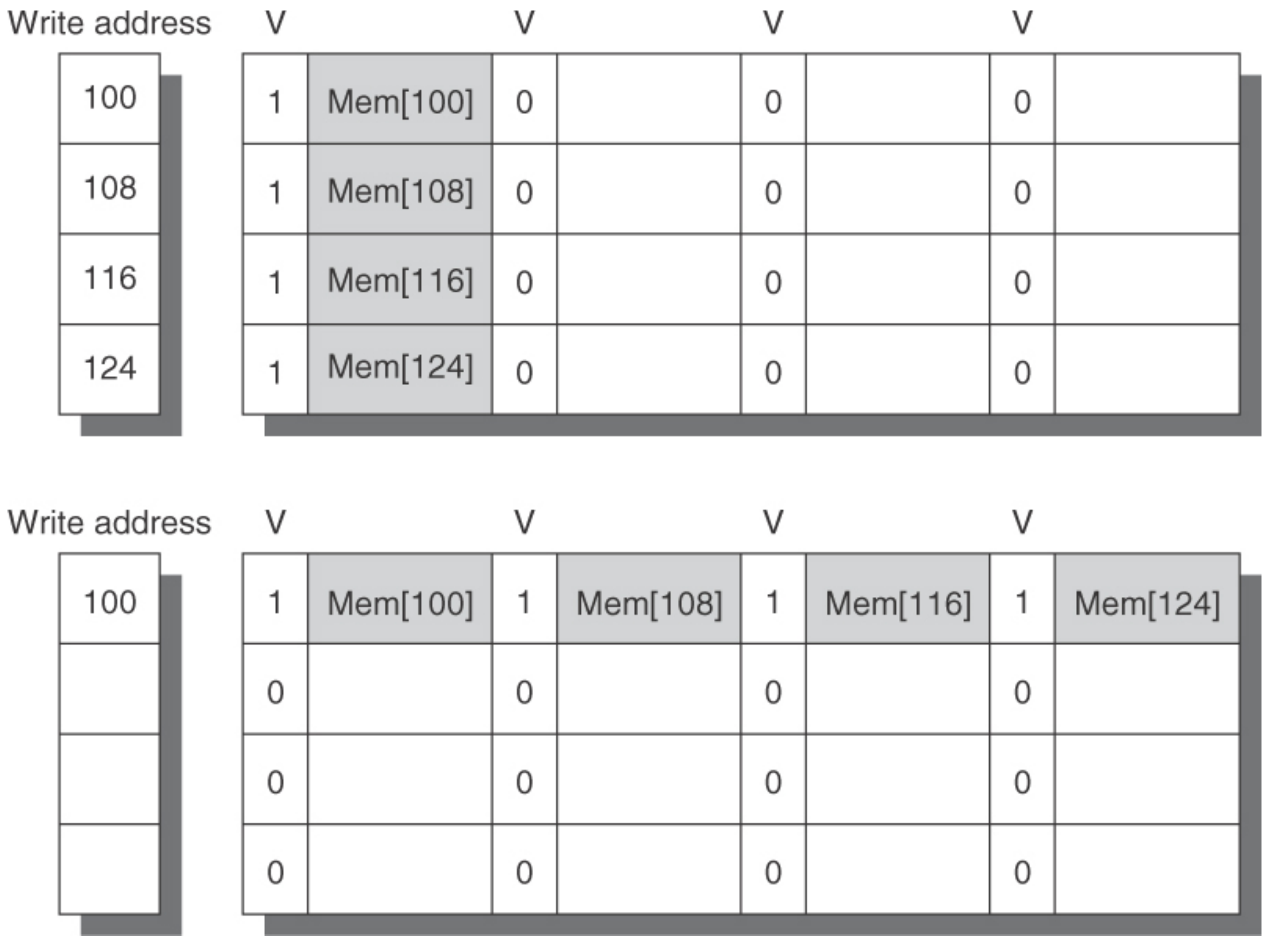
5. Multibanked caches

6. Early Restart and Critical Word First

ER: As soon as the requested word arrives , continue

CWF(or wrapped first) : get the requested word first and continue

7. Merging Write Buffer



Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Figure 2.7

Two Additional Optimizations:

A. Give priority to read misses over writes

Write through cache :

→ writes stacked in the write buffer

→ read miss: check if no conflict w/any write in buffer

if not , bypass writes.

Write back cache :

→ read min displacing a dirty line

→ move the dirty line to a buffer

→ issue the read miss access

→ write the line back later

B. Victim Cache

Add a small, fully assoc cache between a cache and its refill path.

Contains blocks that are dissociated from cache

- Checked on a miss
- If found , swap entries
- 1-5 entries . Fully assoc ;
- good for small , direct mapped caches

8. Compiler Optimizations to reduce miss rate

A. Code reorganization - procedure level

- basic block level

- B. Data reorganization → improve spatial/temporal locality of data. For example, loop interchange:

```
for(j=0;j<100;j=j+1) {
    for(I=0;I<5000;I=I+1){
     x[i][j] = 2 * x [i][j];
    }
}
for(I=0;I<5000;I=I+1){
  for (j=0;j<100;j=j+1) {
     x[i][j] = 2 * x [i][j];
  }
}
```

Blocking

- Improve temporal locality
- One array accessed in row major; another one in column major
- Instead of operating on entire rows/columns,operate on submatrices or blocks
- Goal: maximize reuse of data loaded into the cache before data is replaced

```
for(i=0;i<n;i=i+1)
    for(j=0;j<n;j=j+1) {
      r = 0;
      for(k=0;k<n;k=k+1)
         r=r+y[i][k]*z[k][j];
      x[i][j] = r;
}
```

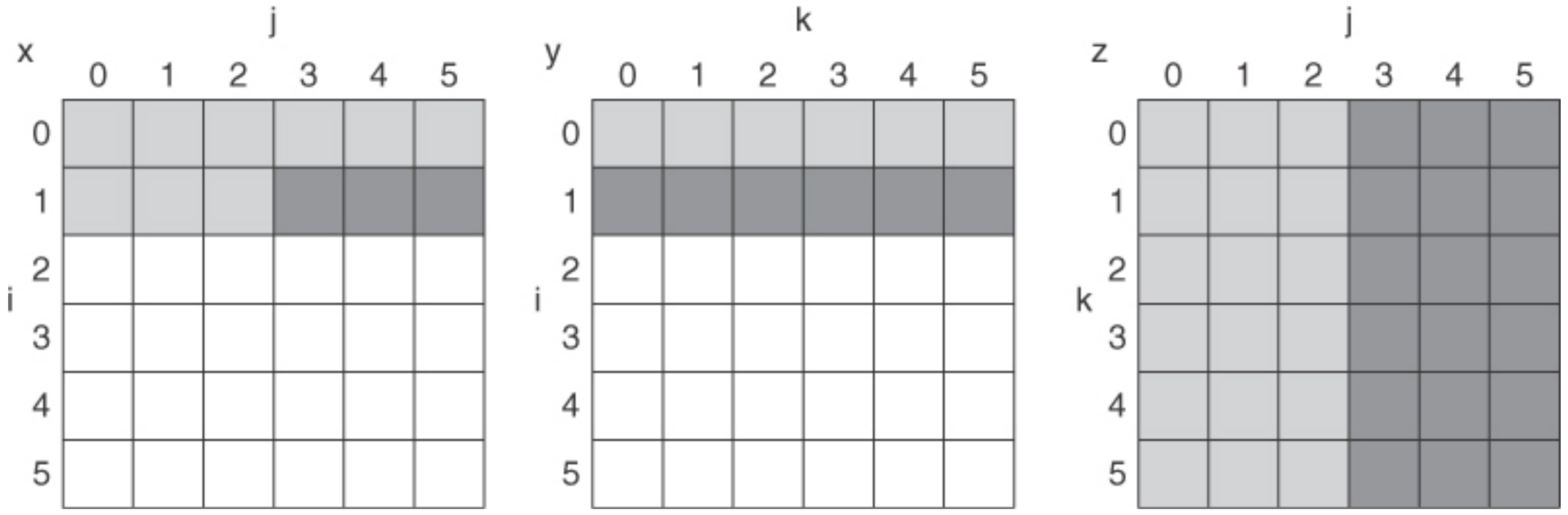


Figure 2.8

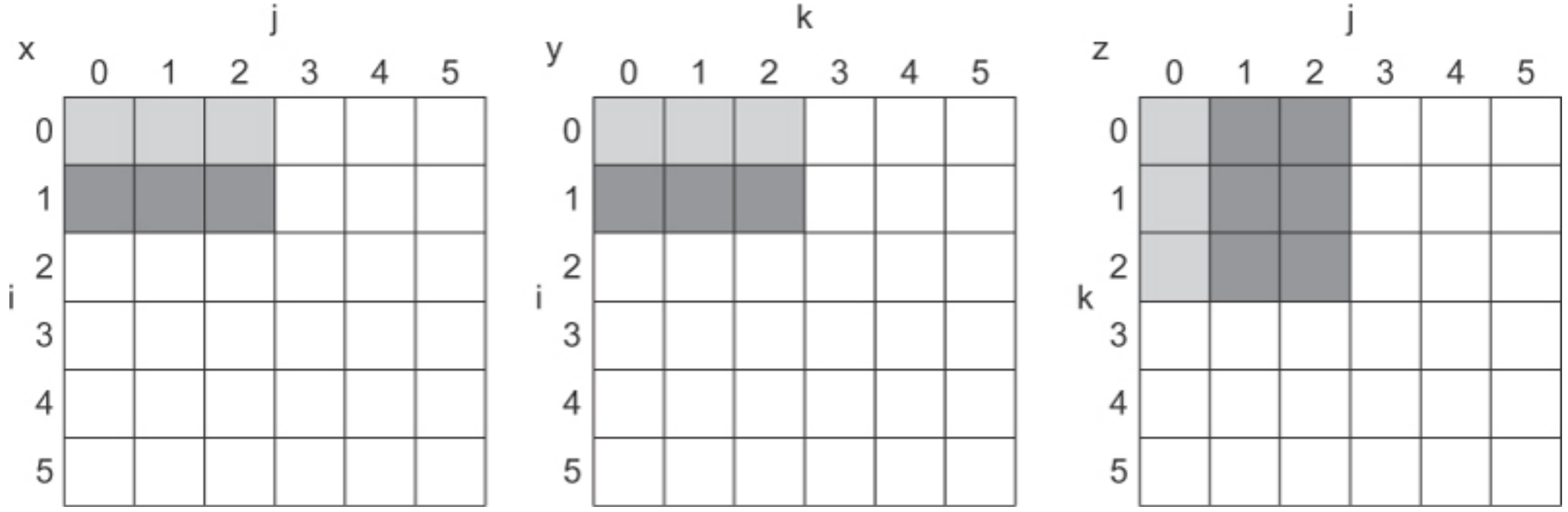


Figure 2.9

Suppose that z overflows the cache →compute on a submatrix of B*B; B= blocking factor;

```
for(jj = 0; jj<n; jj= jj + BB)
    for(kk=0;kk<n;kk=kk+BB)
            for(I = 0;I<n;I=I+1)
                for(j=jj;j<min(jj=B-1,N);j=j+1){
                    r = 0;
                    for(k=kk;k<min(kk+B-1;n);k=k+1) {
                        r=r+y[I][k]*z[k][j];
                    }
                    x[I][j]=x[I][j] + r;
            }
    }
}
```

9. Hardware Prefetching of I,D

- Prefetch : access items before they are needed and deposit them into caches or external buffers
- I prefetching: e.g. fetch next block on a miss or

  on access. The prefetched block goes to a "stream buffer" (or cache)
- D prefetching : same idea

  could have several stream buffers to capture several localities
- careful uses BWdth

10. Compiler Controlled Prefetching

- Compiler inserts prefetch instructions
- Register prefetch : into a reg. (+ cache)
- Cache prefetch : into the cache
- Can be → faulting : causes an exception if protection violation

  → non faulting : turns to No op if it would cause an exception
- Needs a " non blocking " or " lockup free " cache:

  cache can be accessed while there is a prefetch / miss pending.

Example

8 KB dir mapped cache with 16 B blocks

Each element of “ a” and ‘b” is 8 byte long

3r,100c ↙   ↘ 101r,3c

```
for(i=0;i<3;i=i+1)
    for(j=0;j<100;j=j+1)
        a[i][j]= b[j][0] * b[j+1][0]
```

a: even value miss; odd hit (spatial loc) → 150 misses

b: no spatial loc ; temp loc ;

suppose no conflicts , miss 101 times

TOTAL= 251 misses

Prefetching

- Usually works in loops
- can be combined with loop unrolling

  software pipelining
- pbm: has overhead

- Simplifications: 1) not worry about first few misses,
  2) not a faulting pref
- Split so that first loop pref a,b
  second " " a
- assume long latency of miss $\rightarrow$ prefetch 7 iterations ahead

```
for(j=0;j<100;j=j+1) {
    prefetch(b[j+8][0]);
    prefetch(a[0][j+7]);
    a[0][j] = b[j][0]*b[j+1][0];
}
for(I=1;I<3;I=I+1){
  for (j=0;j<100;j=j+1) {
      prefetch(a[I][j+7]);
      a[I][j]=b[j][0]*b[j+1][0];
  }
}
```

We are prefetching a[0][7] - a[0][99]
a[1][7] - a[1][99]
a[2][7] - a[2][99]
b[8][0] - b[100][0]

only left with:

8 misses for b b[0][0]….b[7][0]

12 misses for a: a[0][0] a[0][2] a[0][4] a[0][6]
a[1][0] a[1][2] a[1][4] a[1][6]
a[2][0] a[2][2] a[2][4] a[2][6]

so executing 400 instructions

avoiding 231 misses