Lecture 14-15: September 8, 2021
Computer Architecture and Organization-II
Biplab K Sikdar

Reducing miss rate

0.3.7 Compiler controlled prefetching

Compiler inserts prefetch instructions.

Let consider following loop

```
for (i=0; i<30; i++)
    sum = sum + A[i];
```

During execution, every miss for 'A' stalls CPU.

Solution to this problem can be prefetching of A[i] well in advance.

Code is revised by compiler to avoid miss rate

```
for (i=0; i<30; i++)
    prefetch(A[i+8]);
    sum = sum + A[i];
```

Each cache line is requested 8 iterations prior to its actual use.

Assumption is - miss penalty is equivalent to execution of 8 instructions.

Delay for prefetching is insignificant compared to waiting time for cache miss.

If a block consists 4 A[i]s, then compiler controlled efficient modified code can be

```
for (i=0; i<30; i++)
    if ((i%4) = = 0) prefetch(A[i+8]);
    sum = sum + A[i];
```

0.3.8 Compiler optimizations

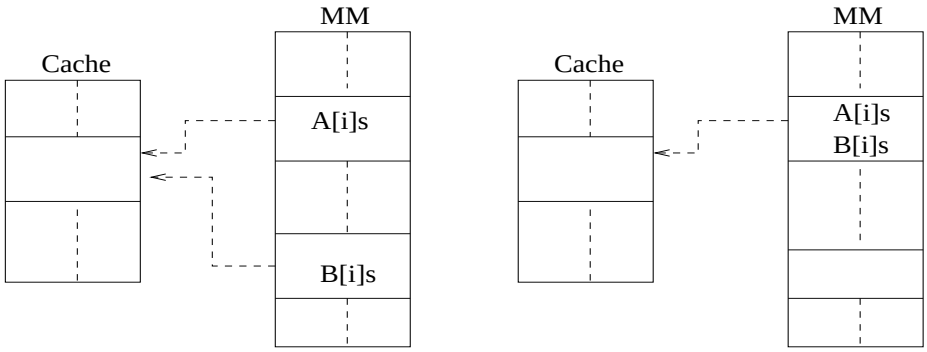
Following schemes reduce capacity miss as well as conflict miss.

a. Merging arrays Let consider A[1:n] and B[1:n]. Execution of Algorithm 0.1

Algorithm 0.1
MergingArrays
⋮
for i=1 to 100 do
  C[i] = B[i] + A[i]
⋮

results in number of misses.

Access to A[i] and then B[i] may generate conflict misses (Figure 22(a)).



a) Before merging a) After merging

Figure 22: Merging of arrays

However, a structure can be defined for A[1:n] and B[1:n] as

begin structure
  array A[1:n];
  array B[1:n];
end structure

This improves spatial locality (Figure 22(b)).

b. Loop interchange It reorients sequence of memory accesses for nested loops.

Interchange of loops depends on arrangement of data items in memory.

Let consider

for(j=0; j <10; j++)
    for(i=0; i <20; i++)
        x[i][j] = a[i] + b[j];

Let maximum 10 blocks of x can be in the cache. Cache system is direct mapped.

Elements of x are stored in row major order.

10 elements per block. That is, 200 elements form 20 blocks - B_0, B_1, B_2, $\cdots$, B_{19}.

Issues related to the access of a and b are not considered.

References for the addresses of $x[i][j]$ in first loop is

x[0][0]
x[1][0]
x[2][0]
$\vdots$
x[19][0]

System issued requests for the blocks as follow

x[0][0], x[0][1], x[0][2], $\cdots$, x[0][9] .....block B_0
x[1][0], x[1][1], x[1][2], $\cdots$, x[1][9] .....block B_1
x[2][0], x[2][1], x[2][2], $\cdots$, x[2][9] .....block B_2
$\vdots$ $\vdots$
x[19][0], x[19][1], x[19][2], $\cdots$, x[19][9] .....block B_{19}

There are 10 compulsory misses for first 10 references

x[0][0], that is for B_0
x[1][0], that is for B_1
$\vdots$
x[9][0], that is for B_9.

For x[10][0] to x[19][0] -that is for B_{10} to B_{19}, there are also 10 compulsory misses.

Due to conflict in cache, B_0 to B_9 are replaced.

B_0 is replaced by B_{10}, B_1 by B_{11} and so on.

During second iteration of outer loop, when x[0][1] is referred,

B_0 is needed and results in a miss.

Effectively, for all 200 references to x- there are 200 misses.

Miss reduction by loop interchange Following is the desired solution

```
for(i=0; i <20; i++)
    for(j=0; j <10; j++)
        x[i][j]= a[i] + a[j];
```

Now, the references for addresses of $x[i][j]$ in first loop is

$$x[0][0]$$
$$x[0][1]$$
$$x[0][2]$$
$$\vdots$$
$$x[0][9]$$

That is, block reference sequence is B_0, B_0, ... 10 times.

That is, there is only 1 miss.

In next iteration of outer loop (i = 1), reference sequence is B_1, B_1, ... 10 times.

It results in only 1 miss.

Effectively, for all 200 references for x - total cache misses is 20.

c. Loop fusion Consider execution of following program segment

```
        ⋮
for i=1 to 100 do
  C[i] = B[i] + A[i];
for i=1 to 100 do
  D[i] = B[i] * 2×A[i];
        ⋮
```

Two elements of each array (A/B/C/D) form a block.

Let check only number of misses for A. For A, there are 50 blocks.

Let cache can accommodate less than 50 blocks of A at a time.

Cache replacement policy is first in first out.

Therefore, there are 50 misses during execution of first loop

As well as 50 misses in second loop for A.

In total 100 misses for A.

But, if loops are fused to a single one

```
        ⋮
for i=1 to 100 do
begin
  C[i] = B[i] + A[i];
  D[i] = B[i] * 2×A[i];
end
        ⋮
```

Number of misses for A is reduced to 50.

Loop fusion reduces miss rate by improving temporal locality.

d. Blocking I skip.

0.4 Reducing Miss Penalty

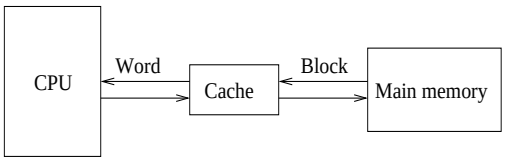


Figure 23: Cache main memory data transfer

miss penalty = time to transfer a block from the main memory to the cache + time to deliver the block to CPU from the cache.

The fact

Memory 'read' operation dominates over the memory 'write'.

In average, about 7% of overall memory traffic is 'write'.

If only data traffic is considered, then about 25% of that is 'write'.

For instruction, almost there is no write access, all are the read access.

If CPU tries to update block B, not in cache, it is a write miss.

B in memory is then copied to cache. Penalty is read miss penalty.

Write miss penalty is counted only when block B is copied from cache to memory.

(during write-through or in write-back)

Therefore, more attentions are given to reduce cache read miss penalty.

0.4.1 Priority to read misses over writes

Access time $t_{AR} < t_{AW}$.

That is, read miss penalty is less in comparison to write miss penalty.

Therefore,

priority is given to read misses over writes.

This scheme is effective but requires extra hardware.

0.4.2 Sub-block placement

Miss penalty for large block B_i is avoided - divide B_i into sub-blocks B_{i1}, B_{i2}, $\cdots$.

When CPU refers item x of B_i, not in cache - it is a read miss for B_i.

Let x belongs to sub-block B_{i2}.

Then in sub-block placement scheme, only B_{i2} is transferred to cache (Figure 24(a)).

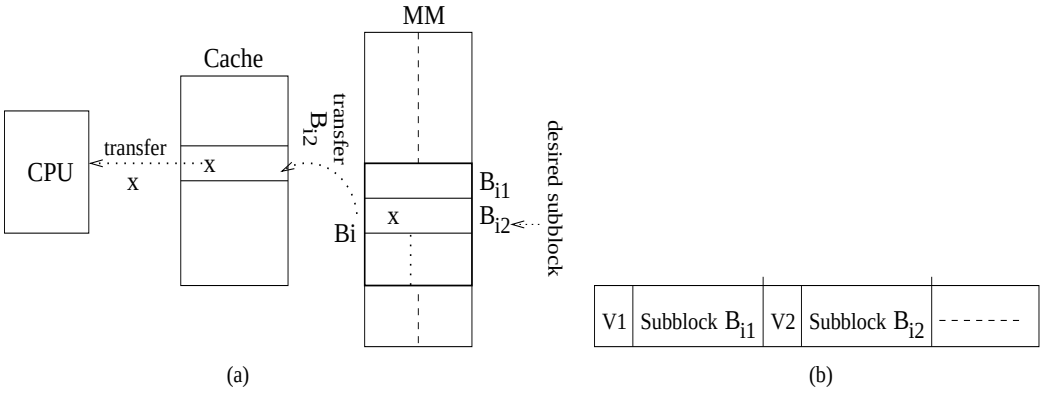


Figure 24: Sub-block placement

This can be most effective when a cache block is to be copied in memory.

Here, only dirty sub-blocks are copied to memory.

There is valid bit V_j for sub-block j (B_{ij}) of block B_i (Figure 24(b)).

(for conventional design, there is one valid bit per block B_i)

For replacing B_i, V_1, V_2, .. are checked to identify sub-blocks to be copied to MM.

All sub-blocks of a block share a single tag.

Sub-block placement is very effective but it requires additional hardware.

Hardware adds delay.

0.4.3 Early restart and critical word first

Early restart

For a miss, while reading B_i from MM,

As soon as requested word/item x is arrived at cache, x is transferred to CPU.

CPU continues execution with x.

It does not wait for transfer of whole B_i (Figure 25(a)).

Rest part of B_i is transferred simultaneously with CPU execution.

It reduces the miss penalty on an average 50%.

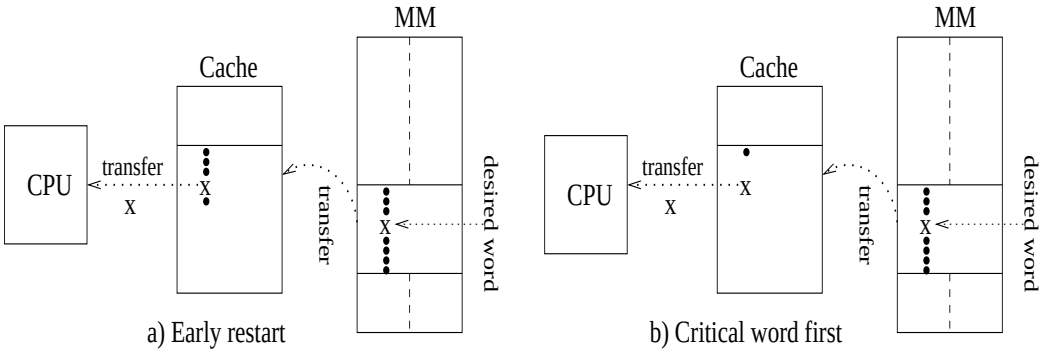


Figure 25: Early restart and critical word first

Critical word first

The system finds requested word/item x first from MM.

It is then sent to CPU.

CPU continues with x.

Rest of the block is transferred simultaneously with CPU execution (Figure 25(b)).

Early restart and critical word first are effective when block size is very large.

These are not so costly to implement. Does not require additional hardware.

0.4.4 Nonblocking caches

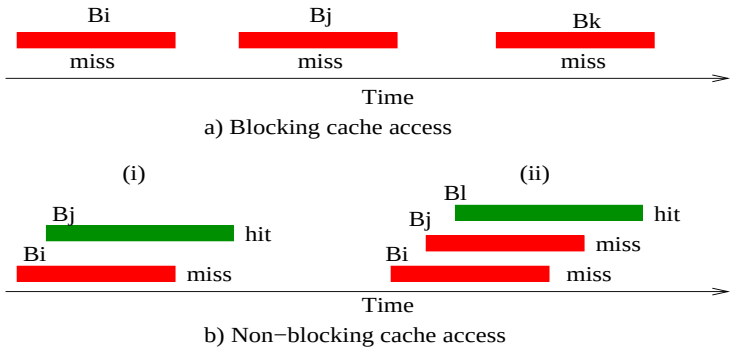


Figure 26: Blocking/non-blocking cache

Conventional cache systems can handle only one request at a time.

If request for a data block B_i is placed and it is a miss,

Then cache waits for B_i to be transferred and wastes stall cycles.

It can not accept other requests -that is, cache is blocked (Figure 26(a)).

Nonblocking cache:

It targets to reduce stalls on misses.

It allows processor to do useful work even in presence of cache miss (Figure 26(b)).

When a miss occurs for data block B_i, request for B_i is placed in a queue.

This miss does not block next cache references that do not need B_i.

That is, a request for block B_j (next to request for B_i) is accepted for service.

This is essential for an out-of-order super scalar processing.

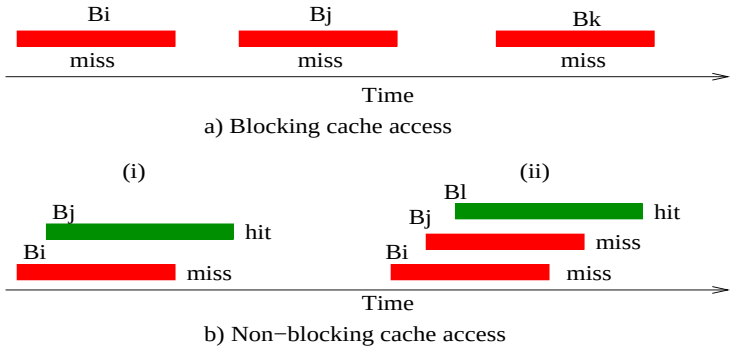
Request for B_j can be either a hit (B_i/B_j of Figure 26(b)i) or miss (Figure 26(b)ii)).

If it is a miss, then it is also placed in the queue.

Non-blocking cache is also known as lockup-free cache.

It requires multi-bank memories.

The *hit under miss*:



The *hit under multiple miss* or *miss under miss* lower the effective miss penalty.

It allows overlapping of multiple misses.

Miss status holding register (MSHR) tracks status of outstanding cache misses.

On a miss for B_i, system allocates an MSHR entry to track request.

If all MSHRs are occupied, then access to cache is blocked (as in blocking cache).

0.4.5 Reduced miss penalty for write

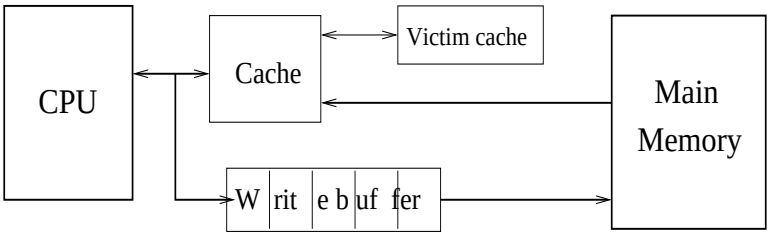


Figure 27: Write buffer

In Write-through, every write operation is a write miss.

To reduce miss penalty, a buffer called *write buffer* (Figure 27) is provided.

Copy of block B_i, modified in cache, is written to buffer.

Write buffer size is determined from simulation run on benchmarks.

Write to buffer, however, complicates memory access.

Read miss for x in B_i, updated x (block B_i) may be in write buffer.

What can be done

Option a: Read miss for x (block B_i) waits until write buffer is empty.

On read miss, write buffer is written first to MM. Then B_i is transferred to cache.

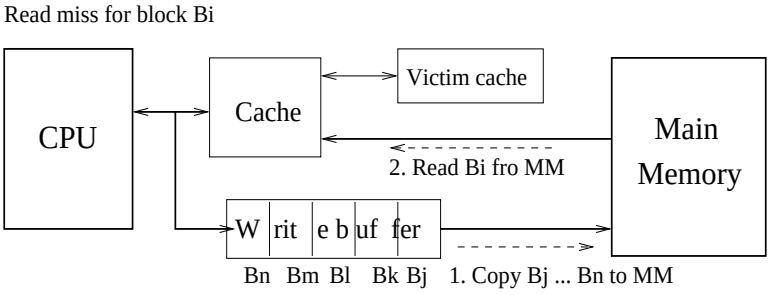


Figure 28: Manage write buffer - option (a)

This leads to larger miss penalty (Figure 28).

What can be done

Option b: On read miss, first check write buffer.

If B_i is not in buffer, normal miss action can continue (Figure 29(i)).

Otherwise, B_i is taken to cache from buffer (Figure 29(ii)).

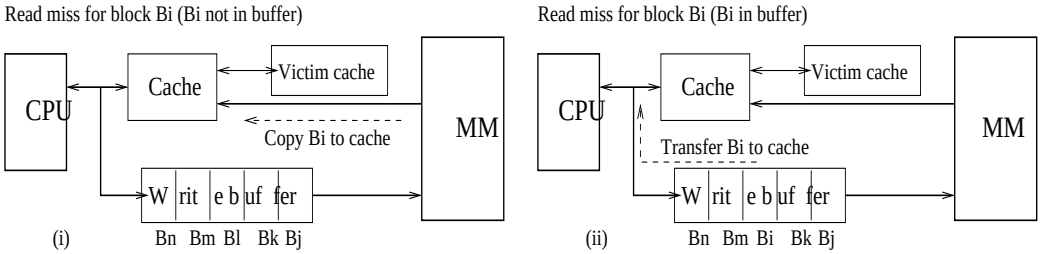


Figure 29: Manage write buffer - option (b)

In write-back When write-back,

A dirty block B_i may be transferred from cache on cache read miss for block B_j (when B_i and B_j have same cache index -that is, the conflict miss).

Normal implementation is - transfer B_i to main memory and then read B_j.

In a system with write buffer, for such an event,

Dirty block B_i is first transferred to write buffer and then B_j is read from MM.

Finally, B_i is transferred to memory (Figure 30). Read (for B_j) gets higher priority.

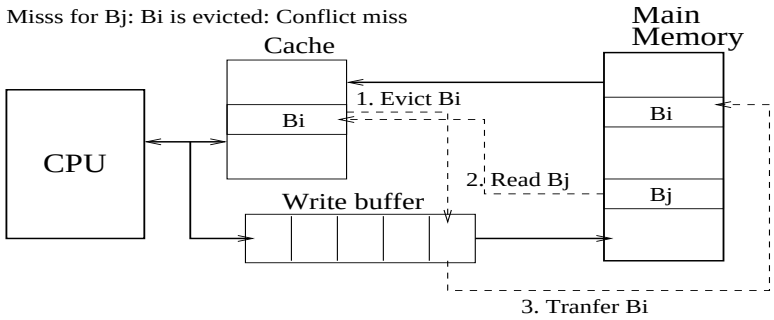


Figure 30: Managing write buffer in write-back

0.4.6 Two-level caches

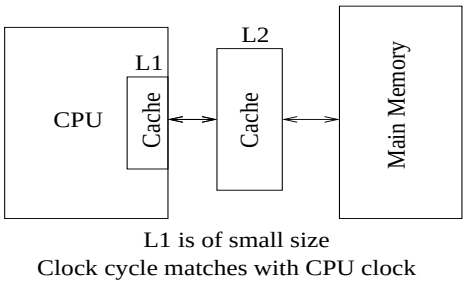


Figure 31: Two level cache

Two cache levels L_1 and L_2 (Figure 31).

L_1's (on-chip) access speed matches with CPU speed.

Average memory access time,

$$\text{AMAT} = \textit{hit ratio } L_1 \times \textit{hit time } L_1 + \textit{miss rate } L_1 \times \textit{miss penalty } L_1$$

where,

$$\textit{miss penalty } L_1 = \textit{hit ratio } L_2 \times \textit{hit time } L_2 + \textit{miss rate } L_2 \times \textit{miss penalty } L_2$$

therefore,

$$\text{AMAT} = \textit{hit ratio } L_1 \times \textit{hit time } L_1 + \textit{miss rate } L_1 \times (\textit{hit ratio } L_2 \times \textit{hit time } L_2 + \textit{miss rate } L_2 \times \textit{miss penalty } L_2)$$

Performance of two-level cache system.

$$Missrate_{local} = \frac{\text{Number of misses in the cache}}{\text{Total number of memory accesses to this cache}}$$

$$Missrate_{global} = \frac{\text{Number of misses in the cache}}{\text{Total number of memory accesses by the CPU}}$$

Global miss rate is more useful measure.

For L_1, global and local miss rates are same.

If number of requests from CPU is (N_1+N_2),

where L_1 supports N_1 requests and N_2 requests are misses at L_1,

then local and global miss rate at L_1 is

$$\frac{N_2}{N_1+N_2}.$$

Now, if L_2 can not support N'$_2$ requests, then the local miss rate at L_2 is

$$\frac{N_2'}{N_2}.$$

Its global miss rate, however, is

$$\frac{N_2'}{N_1+N_2}.$$

Example 0.1 Let total number of memory references is = 10000 and number of misses at L_1 and L_2 are 200 and 40 respectively. Then

(i) Miss rate (local or global) at L_1 is $\frac{200}{10000}$ = 0.02.

(ii) Missrate$_{local}$ at L_2 is $\frac{40}{200}$ = 0.20.

(iii) Missrate$_{global}$ at L_2 is $\frac{40}{10000}$ = 0.004.