Cache Performance

- Memory system and processor performance:

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock time} \longrightarrow \text{CPU performance eqn.}$$

$$\text{CPI} = \text{CPI}_{\text{ld/st}} \times \frac{\text{IC}_{\text{ld/st}}}{\text{IC}} + \text{CPI}_{\text{others}} \times \frac{\text{IC}_{\text{others}}}{\text{IC}}$$

$$\text{CPI}_{\text{ld/st}} = \text{Pipeline time} + \text{Average memory access time}$$

$$\text{Avg. mem. time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \longrightarrow \text{Memory performance eqn.}$$

- Improving memory hierarchy performance:
  - Decrease hit time
  - Decrease miss rate
  - Decrease miss penalty

Cache Performance – example problem

Assume we have a computer where the CPI is 1 when all memory accesses hit in the cache. Data accesses (ld/st) represent 50% of all instructions. If the miss penalty is 25 clocks and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

[H&P 5th ed, B.1]

Answer First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}\\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle}\\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25\\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term (1 + 0.5) represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle}\\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{Clock cycle}}\\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster.

Reducing Cache Miss Rates

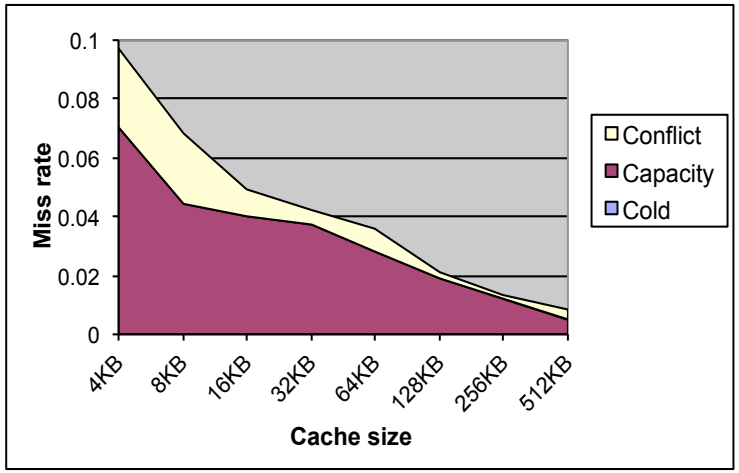
Cache miss classification: the “three C’s”

- Compulsory misses (or cold misses): when a block is accessed for the first time
- Capacity misses: when a block is not in the cache because it was evicted because the cache was full
- Conflict misses: when a block is not in the cache because it was evicted because the cache set was full

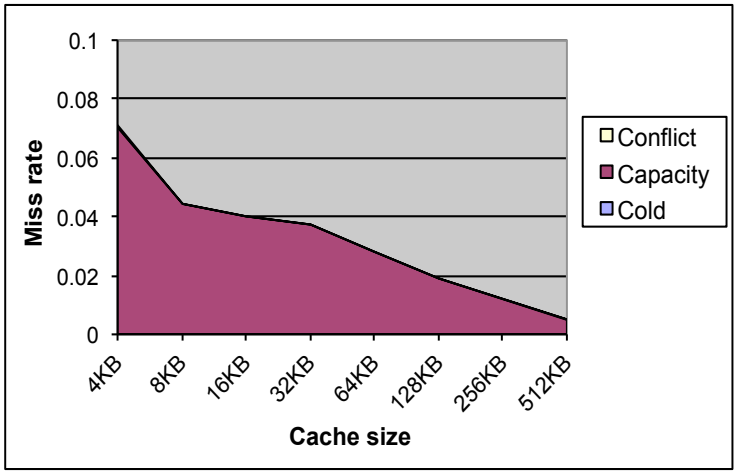
Cache Misses vs. Cache Size

H&P Fig. 5.15

Direct mapped



4-way set associative



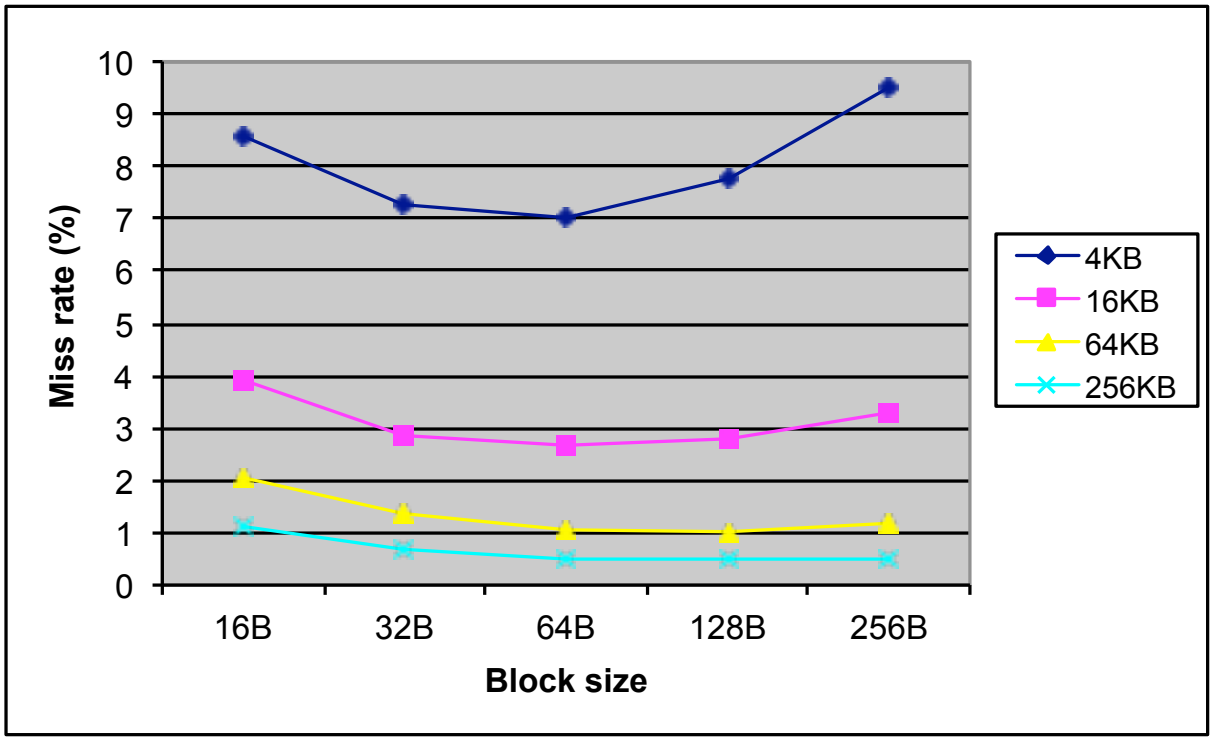
- Miss rates are very small in practice (caching is effective!)
- Miss rates decrease significantly with cache size
- Miss rates decrease with set-associativity because of reduction in conflict misses

Reducing Cold Miss Rates

Technique 1: Large block size

- Principle of locality → other data in the block are likely to be used soon
- Reduce cold miss rate
- May increase conflict and capacity miss rate for the same cache size (fewer blocks in cache)
- Increase miss penalty because more data has be brought in each time
- Uses more memory bandwidth

Cache Misses vs. Block Size



H&P
Fig. 5.16

- Small caches are very sensitive to block size
- Very large blocks (> 128B) never beneficial

Reducing Cold Miss Rates

Technique 2: Prefetching

- Idea: bring into the cache (or a special buffer) ahead of time data or instructions that are likely to be used soon
- Reduce cold misses
- Uses more memory bandwidth
- May increase conflict and capacity miss rates (unless prefetch buffer is used)
- Does not increase miss penalty (prefetch is handled after main cache access is completed)

Prefetching

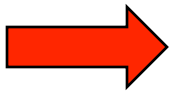
- Hardware prefetching: hardware automatically prefetches cache blocks on a cache miss
  - No need for extra prefetching instructions in the program
  - Effective for regular accesses, such as instructions
  - E.g., next blocks prefetching, stride prefetching

- Software prefetching: compiler inserts instructions at proper places in the code to prefetch
  - Requires new IS instructions for prefetching (nonbinding prefetch)
  - Adds instructions to compute the prefetching addresses and to perform the prefetch itself (prefetch overhead)
  - E.g., data prefetching in loops, linked list prefetching

Software Prefetching

- E.g., prefetching in loops: Brings the next required block, two iterations ahead of time (assuming each element of x is 4-bytes long and the block has 64 bytes).

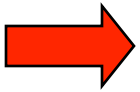
```
for (i=0; i<=999; i++) {
  x[i] = x[i] + s;
}
```



```
for (i=0; i<=999; i++) {
  if (i%16 == 0)
    prefetch(x[i+32]);
  x[i] = x[i] + s;
}
```

- E.g, linked-list prefetching: Brings the next object in the list

```
while (student) {
  student->mark = rand();
  student = student->next;
}
```



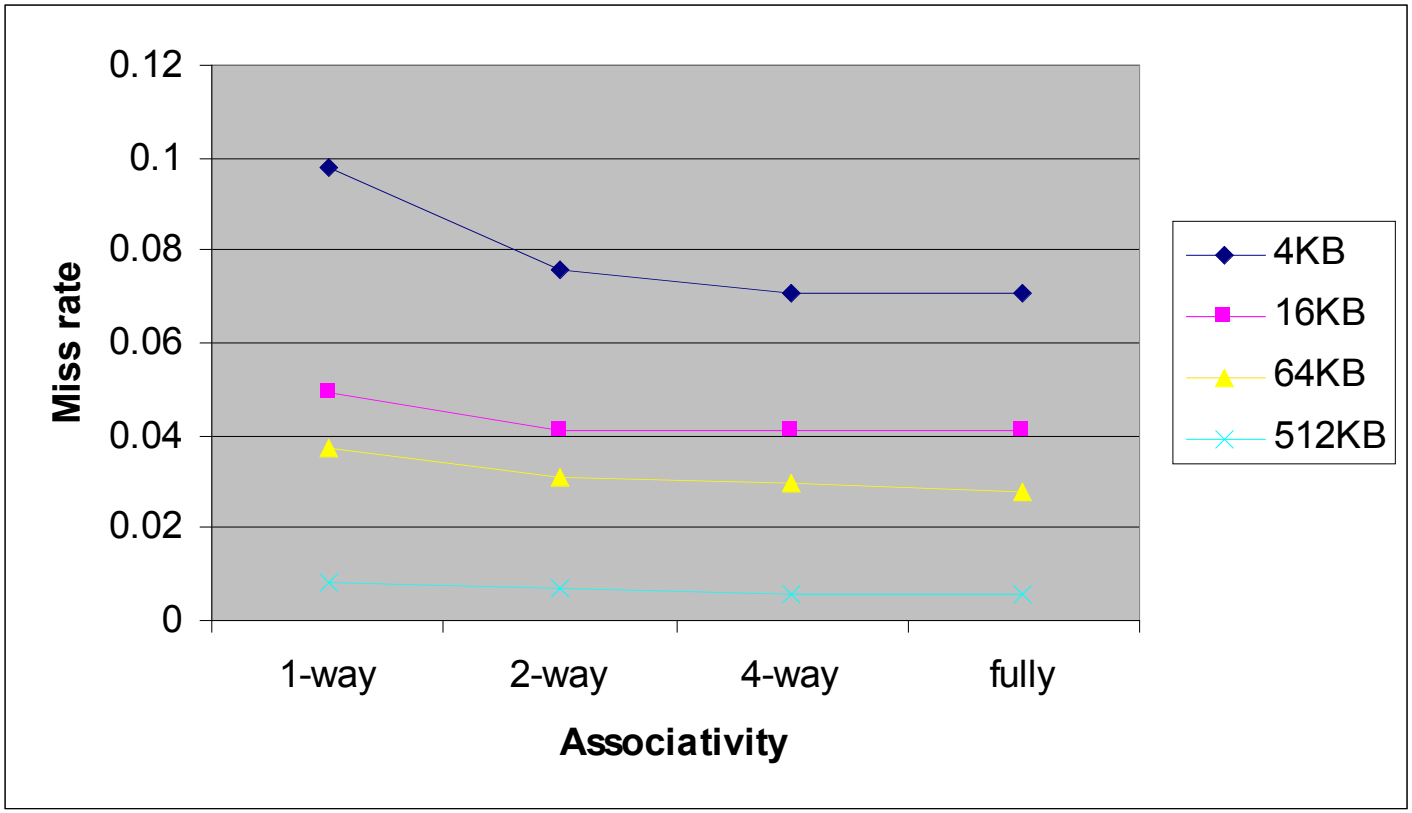
```
while (student) {
  prefetch(student->next);
  student->mark = rand();
  student=student->next;
}
```

Reducing Conflict Miss Rates

Technique 3: High associativity caches

- More options for block placement → fewer conflicts
- Reduce conflict miss rate
- May increase hit access time because tag match takes longer
- May increase miss penalty because replacement policy is more involved

Cache Misses vs. Associativity

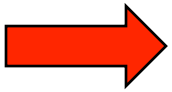


- Small caches are very sensitive to associativity
- In all cases more associativity decreases miss rate, but little difference between 4-way and fully associative

Technique 4: Compiler optimizations

- E.g., merging arrays: improves spatial locality if the fields are used together for the same index

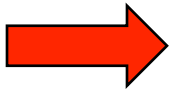
```
int val[size];
int key[size];
```



```
struct merge {
  int val;
  int key;
};
Struct merge merged_array[size];
```

- E.g., loop fusion: improves temporal locality

```
for (i=0; i<1000; i++)
  A[i] = A[i]+1;
for (i=0; i<1000; i++)
  B[i] = B[i]+A[i];
```



```
for (i=0; i<1000; i++) {
  A[i] = A[i]+1;
  B[i] = B[i]+A[i];
}
```

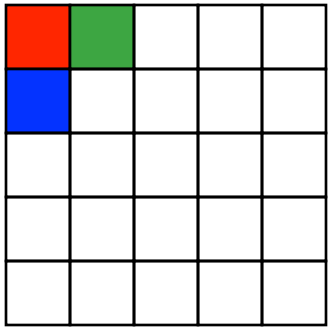
Reducing Conflict Miss Rates

- E.g., blocking: change row-major and column-major array distributions to block distribution to improve spatial and temporal locality

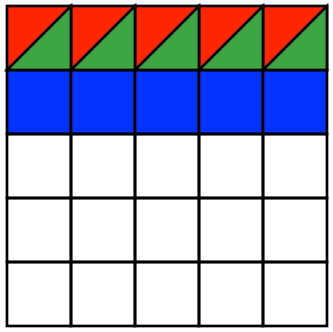
```
for (i=0; i<5; i++)
  for (j=0; j<5; j++) {
    r=0;
    for (k=0; k<5; k++) {
      r=r+y[i][k]*z[k][j];
    x[i][j]=r;
    }
```

(matrix multiplication x=y*z)

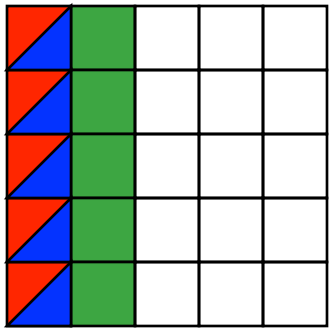
x:



y:



z:



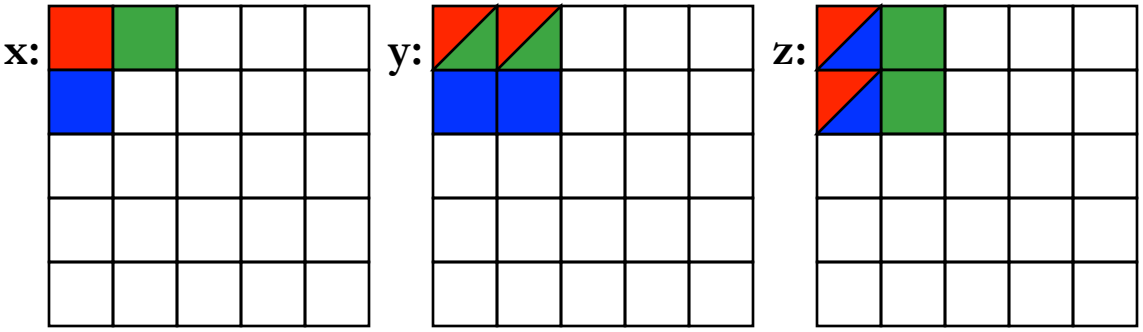
```
i=0;j=0;0<k<5
i=0;j=1;0<k<5
      ...
i=1;j=0;0<k<5
```

Poor temporal locality

Poor spatial and temporal locality

Reducing Conflict Miss Rates – Loop Blocking or Tiling

```
for (jj = 0; jj < 5; jj = jj+2)
  for (kk = 0; kk < 5; kk = kk+2)
    for (i = 0; i < 5; i++)
      for (j = jj; j < min(jj+2-1,5); j++)
      { r = 0;
        for (k = kk; k < min(kk+2-1,5); k++)
          r = r + y[i][k]*z[k][j];
        x[i][j]= x[i][j] + r;
      }
```



jj=0;kk=0;i=0;j=0;0<k<1

jj=0;kk=0;i=0;j=1;0<k<1

jj=0;kk=0;i=1;j=0;0<k<1

Better temporal locality

Cache Performance II

- Memory system and processor performance:

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock time} \longrightarrow \text{CPU performance eqn.}$$

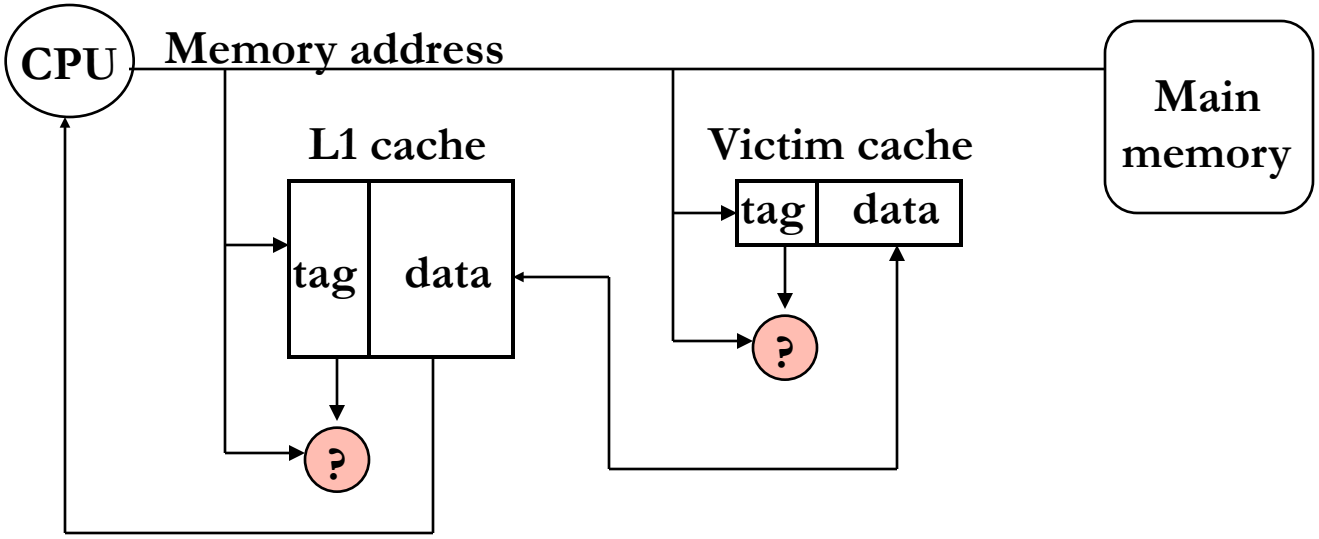
$$\text{Avg. mem. time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \longrightarrow \text{Memory performance eqn.}$$

- Improving memory hierarchy performance:
  - Decrease hit time
  - Decrease miss rate (block size, prefetching, associativity, compiler)
  - Decrease miss penalty

Reducing Cache Miss Penalty

Technique 1: Victim caches

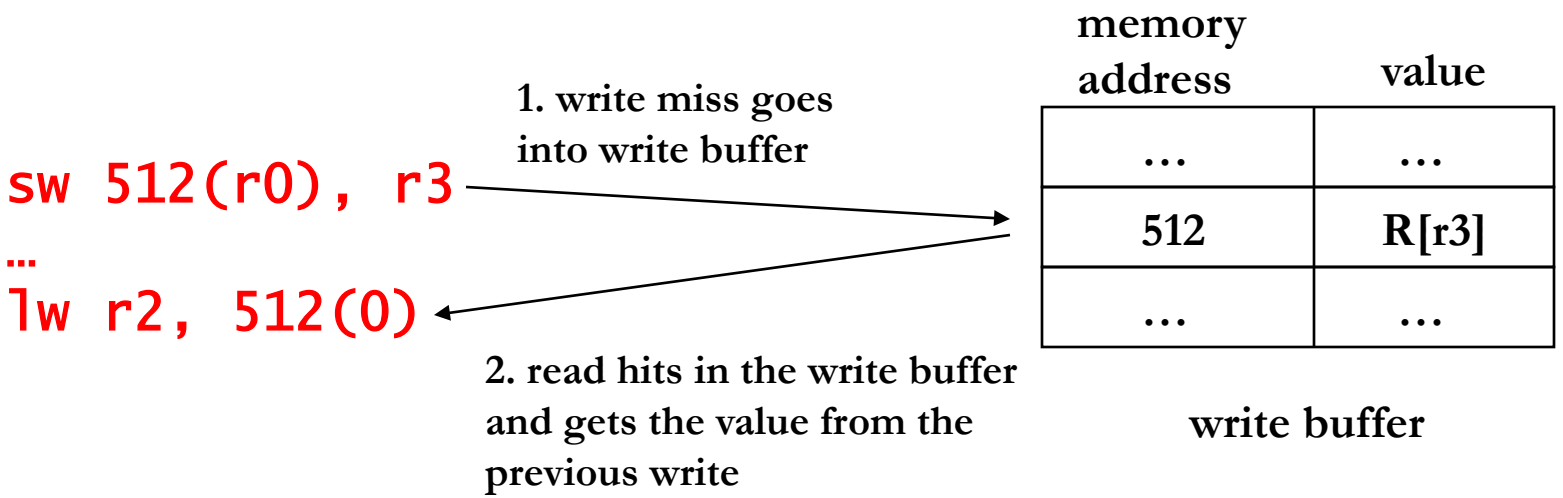
- (Can also considered to reduce miss rate)
- Very small cache used to capture evicted lines from cache
- In case of cache miss the data may be found quickly in the victim cache
- Access victim cache in series or in parallel with main cache. Trade-off?
- Replacement policy is more involved



Reducing Cache Miss Penalty

Technique 2: giving priority to reads over writes

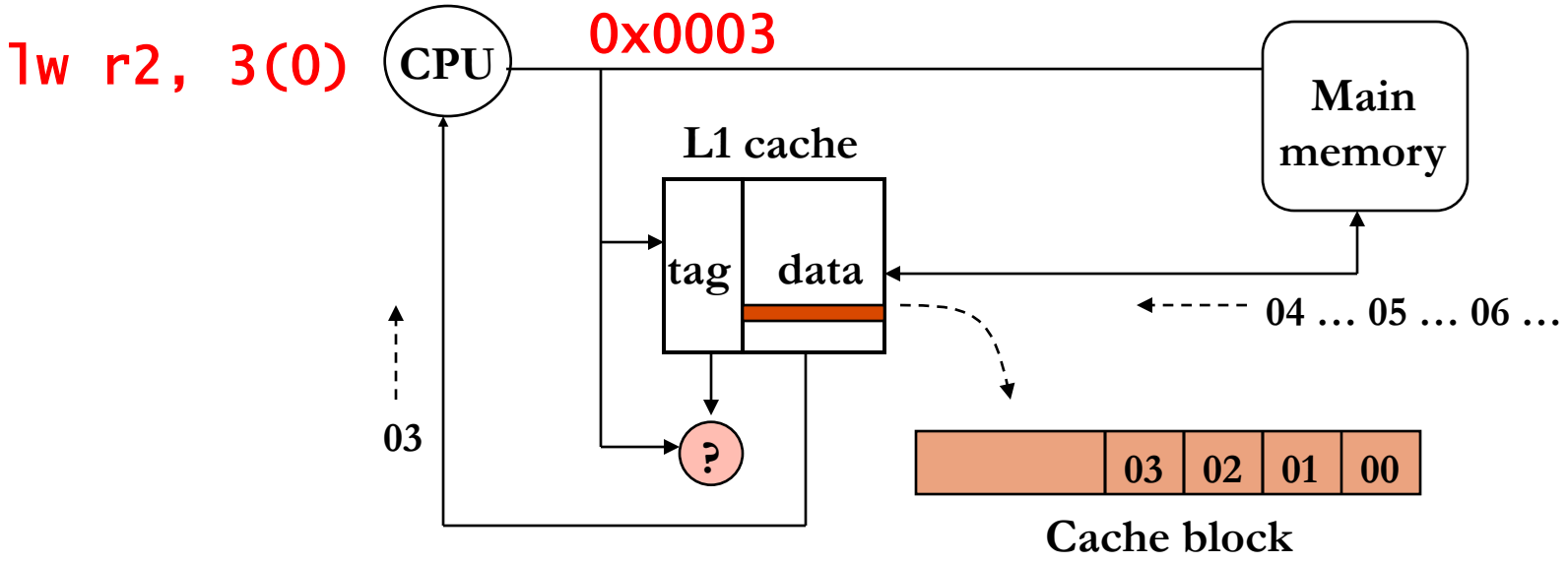
- The value of a read (load instruction) is likely to be used soon, while a write does not affect the processor
- Idea: place write misses in a write buffer, and let read misses overtake writes
- Reads to the same memory address of a pending write in the buffer now become hits in the buffer:



Reducing Cache Miss Penalty

Technique 3: early restart and critical word first

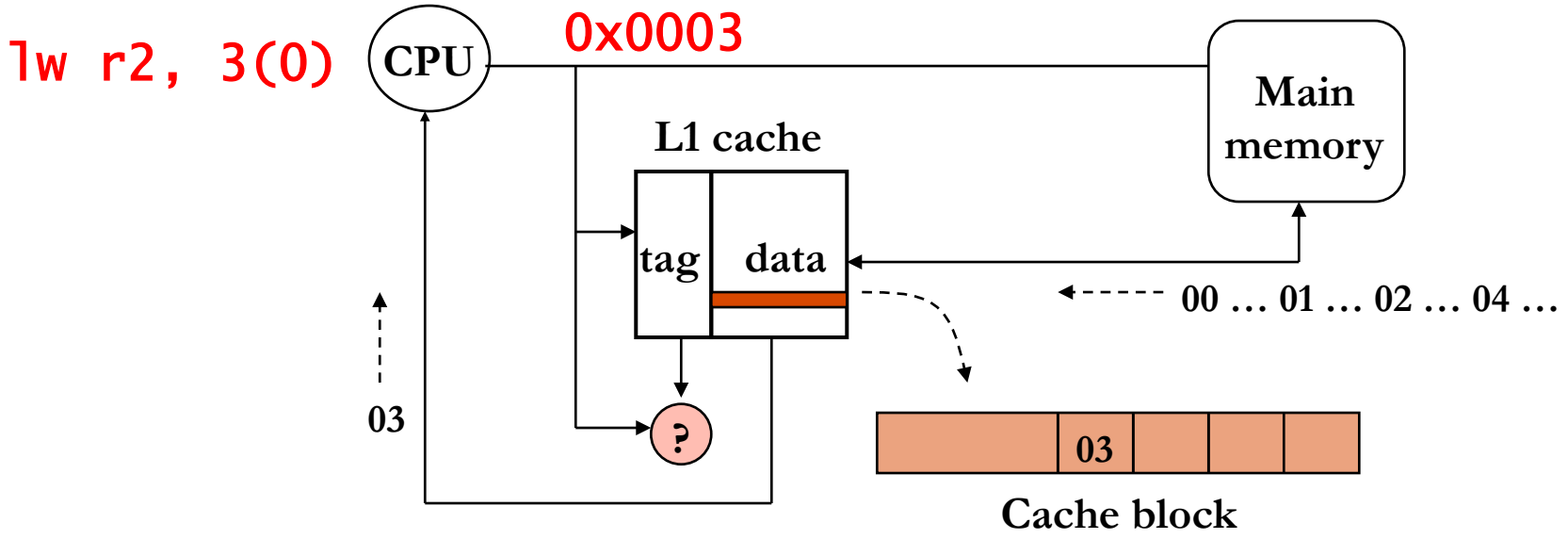
- On a read miss processor will need just the loaded word (or byte) very soon, but processor has to wait until the whole block is brought into the cache
- Early restart: as soon as the requested word arrives in the cache, send it to the processor and then continue reading the rest of the block into the cache



Reducing Cache Miss Penalty

Technique 3: early restart and critical word first

- On a read miss processor will need just the loaded word (or byte) very soon, but processor has to wait until the whole block is brought into the cache
- Early restart: as soon as the requested word arrives in the cache, send it to the processor and then continue reading the rest of the block into the cache
- Critical word first: get the requested word first from the memory, send it asap to the processor and then continue reading the rest of the block into the cache

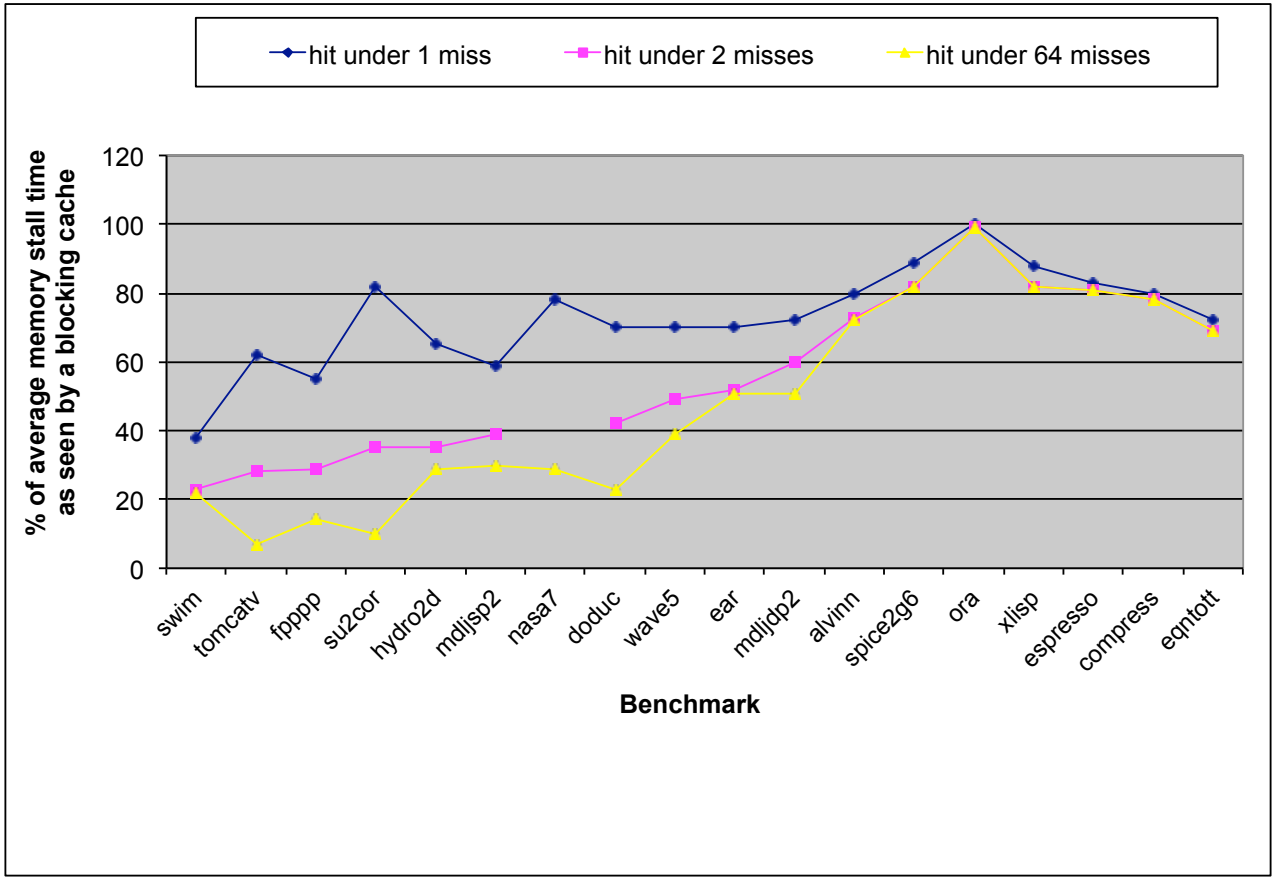


Reducing Cache Miss Penalty

Technique 4: non-blocking (or lockup-free) caches

- Dynamic scheduling (Tomasulo's): ALU instructions can overtake a cache miss instruction
- Non-blocking caches: other memory instructions can also overtake a cache miss instruction
  - Cache can service multiple hits while waiting on a miss: "hit under miss"
  - More aggressive: cache can service multiple hits while waiting on multiple misses: "miss under miss" or "hit under multiple misses"
- Cache and memory must be able to service multiple requests concurrently
- Must keep track of multiple outstanding memory operations
- Increased hardware complexity

Non-blocking Caches



H&P
Fig. 5.23

- Significant improvement from small degree of outstanding memory operations
- Some applications benefit from large degrees

Reducing Cache Miss Penalty

Technique 4: second level caches (L2)

- Gap between main memory and L1 cache speeds is increasing
- L2 makes main memory appear to be faster if it captures most of the L1 cache misses
  - L1 miss penalty becomes L2 hit access time if hit in L2
  - L1 miss penalty higher if miss in L2
- L2 considerations:
  - 256KB – 1MB capacity
  - 10 – 20 cycles access time
  - Higher associativity (16-32 ways) possible. Why?
  - Higher miss rate than L1. Why?
- L3 caches now standard on laptop/desktop/server processors
  - 30+ cycle access time
  - 2-20+ MB capacity

Second Level Caches

- Memory subsystem performance:

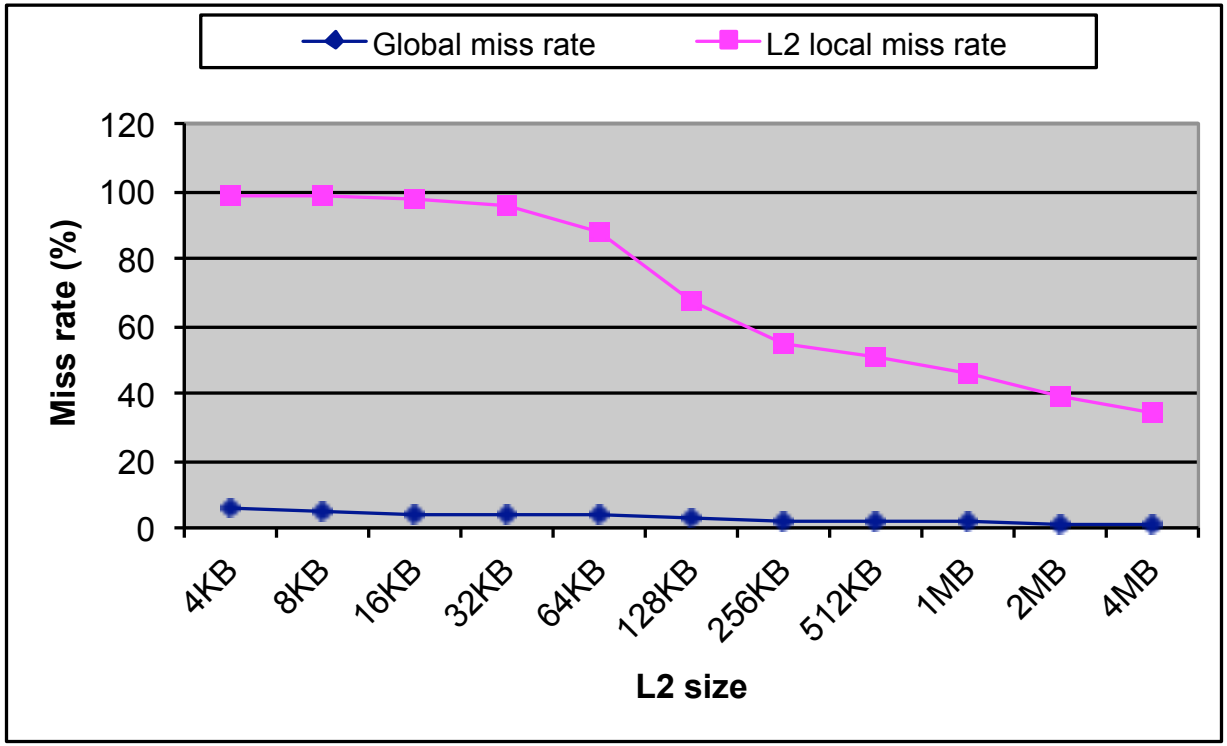
$$\text{Avg. mem. time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

$$\therefore \text{Avg. mem. time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

- Miss rates:
  - Local: the number of misses divided by the number of requests to the cache
    - E.g., Miss rate_{L1} and Miss rate_{L2} in the equations above
    - Usually not so small for lower level caches
  - Global: the number of misses divided by the total number of requests from the CPU
    - E.g, L2 global miss rate = Miss rate_{L1} x Miss rate_{L2}
    - Represents the aggregate effectiveness of the caches combined

Cache Misses vs. L2 size



H&P
Fig. 5.10

- L2 caches must be much bigger than L1
- Local miss rates for L2 are larger than for L1 and are not a good measure of overall performance

Cache Performance II

- Memory system and processor performance:

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock time} \longrightarrow \text{CPU performance eqn.}$$

$$\text{Avg. mem. time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \longrightarrow \text{Memory performance eqn.}$$

- Improving memory hierarchy performance:
  - Decrease hit time
  - Decrease miss rate (block size, prefetching, associativity, compiler)
  - Decrease miss penalty (victim caches, reads over writes, prioritize critical word, non-blocking caches, additional cache levels)

Reducing Cache Hit Time

Technique 1: small and simple caches

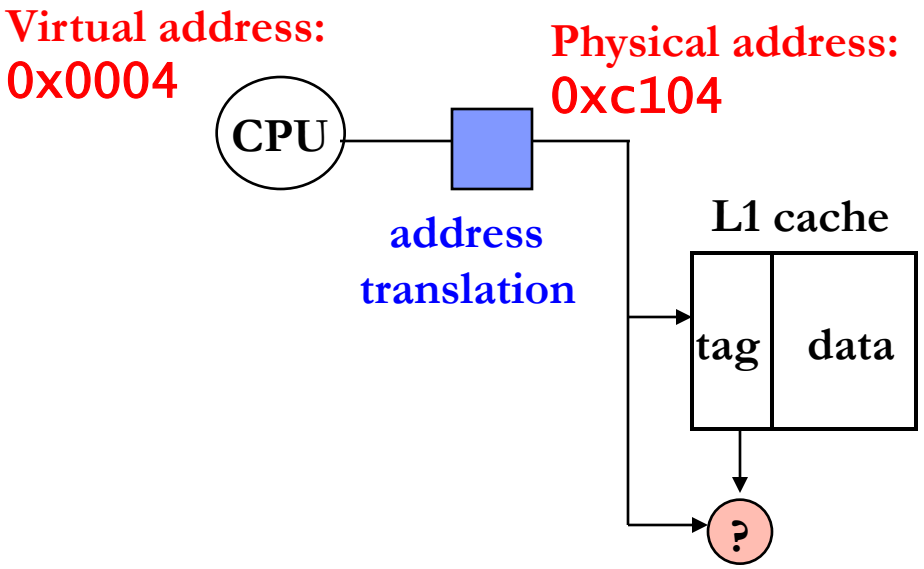
- Small caches fit on-chip → signals take a long time to go off-chip
- Low associativity caches have few tags to compare against the requested data
- Direct mapped caches have only one tag to compare and comparison can be done in parallel with the fetch of the data

Reducing Cache Hit Time

Technique 2: virtual address caches

- Programs use virtual addresses for data, while main memory uses physical addresses → addresses from processor must be translated at some point

Discussed in "Virtual Memory" lecture (next).



Cache Performance Techniques

technique	miss rate	miss penalty	hit time	complexity
large block size	☺	☹		☺
high associativity	☺		☹	☹
victim cache	☺	☺		☹
hardware prefetch	☺			☹
compiler prefetch	☺			☹
compiler optimizations	☺			☹
prioritisation of reads		☺		☹
critical word first		☺		☹
nonblocking caches		☺		☹
L2 caches		☺		☹
small and simple caches	☹		☺	☺
virtual caches			☺	☹