Announcements

- Previous lecture
  - Caches

Recap: Memory Hierarchy Issues

- Block size: smallest unit that is managed at each level
  - E.g., 64B for cache lines, 4KB for memory pages
- Block placement: Where can a block be placed?
  - E.g., direct mapped, set associative, fully associative
- Block identification: How can a block be found?
  - E.g., hardware tag matching, OS page table
- Block replacement: Which block should be replaced?
  - E.g., Random, Least recently used (LRU), Not recently used (NRU)
- Write strategy: What happens on a write?
  - E.g., write-through, write-back, write-allocate
- Inclusivity: whether next lower level contains all the data found in the current level
  - Inclusive, exclusive

Announcements

- Previous lecture
  - Caches
- This lecture
  - Cache Performance.
- Tutorials happening this week & next

Cache Performance

- Memory system and processor performance:

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

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

$$\text{CPI}_{ld/st} = \text{Average memory access time (AMAT)}$$

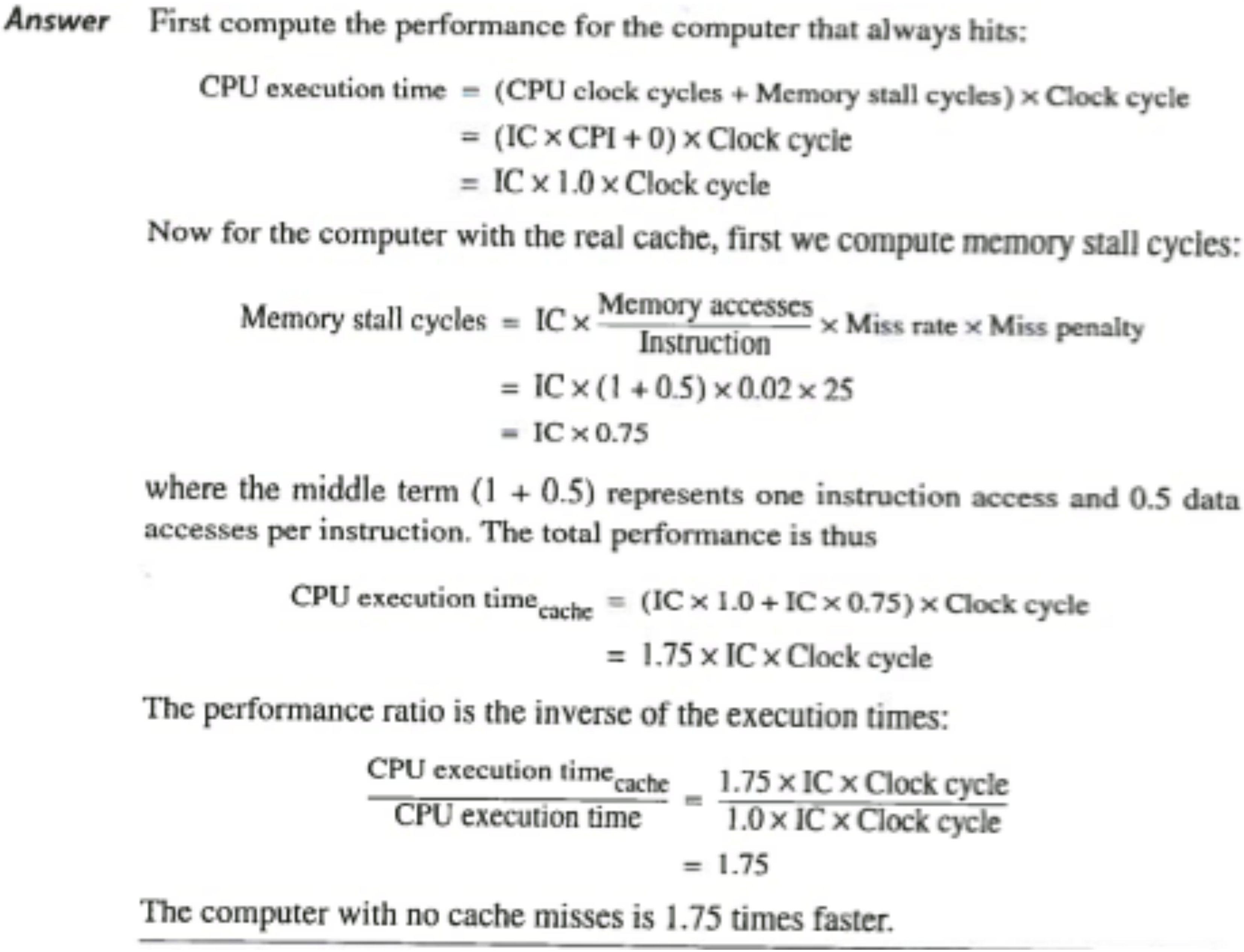
$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \longrightarrow \text{Memory performance eqn.}$$

- 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

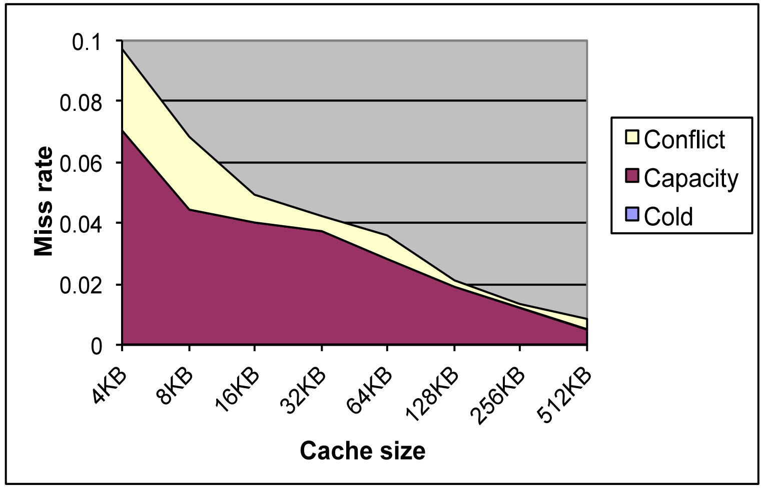
Cache miss classification: the “three C’s”

- 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
  - Conflict misses only exist in direct-mapped or set-associative caches
  - In a fully associative cache, all non-compulsory misses are capacity misses

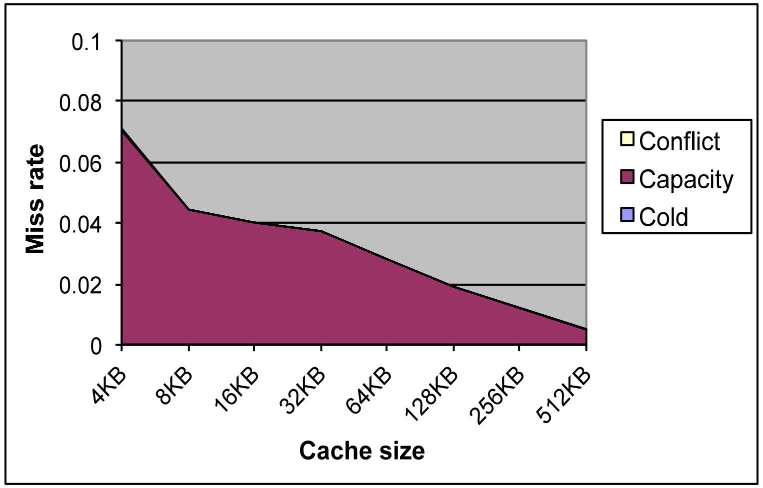
Cache Misses vs. Cache Size

H&P
Fig. 5.15

Direct mapped



4-way set associative



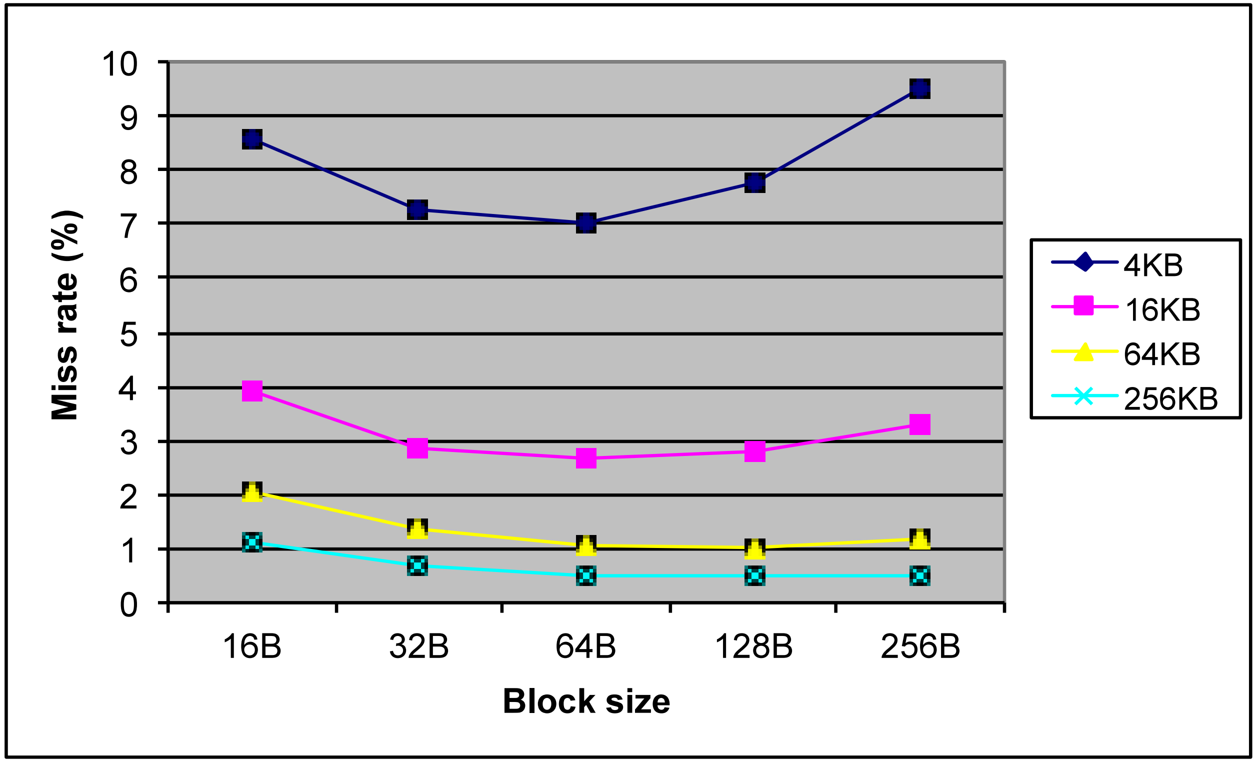
- Miss rates are very small in practice (caching is effective!)
- Miss rates decrease significantly with cache size
  - Rule of thumb: miss rates change in proportion to √ of cache size e.g., 2x cache → √2 fewer misses
- Miss rates decrease with set-associativity because of reduction in conflict misses

Reducing Cold Miss Rates

Technique 1: Large block size

- Principle of spatial locality → other data in the block 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 to 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
- 64B is a sweet spot → common choice in today's processors

Reducing Cold Miss Rates

Technique 2: Prefetching

- Idea: bring into the cache ahead of time data or instructions that are likely to be used soon
- Can reduce cold misses (also capacity misses)
- Uses more memory bandwidth
- Does not typically increase miss penalty (prefetch is generally handled after main cache access is completed)
- May increase conflict and capacity miss rates by displacing useful blocks (cache pollution)
  - Can use a prefetch buffer to avoid polluting the cache

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 trigger prefetches
  - Requires ISA support (nonbinding prefetch instruction)
  - 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;
}
```

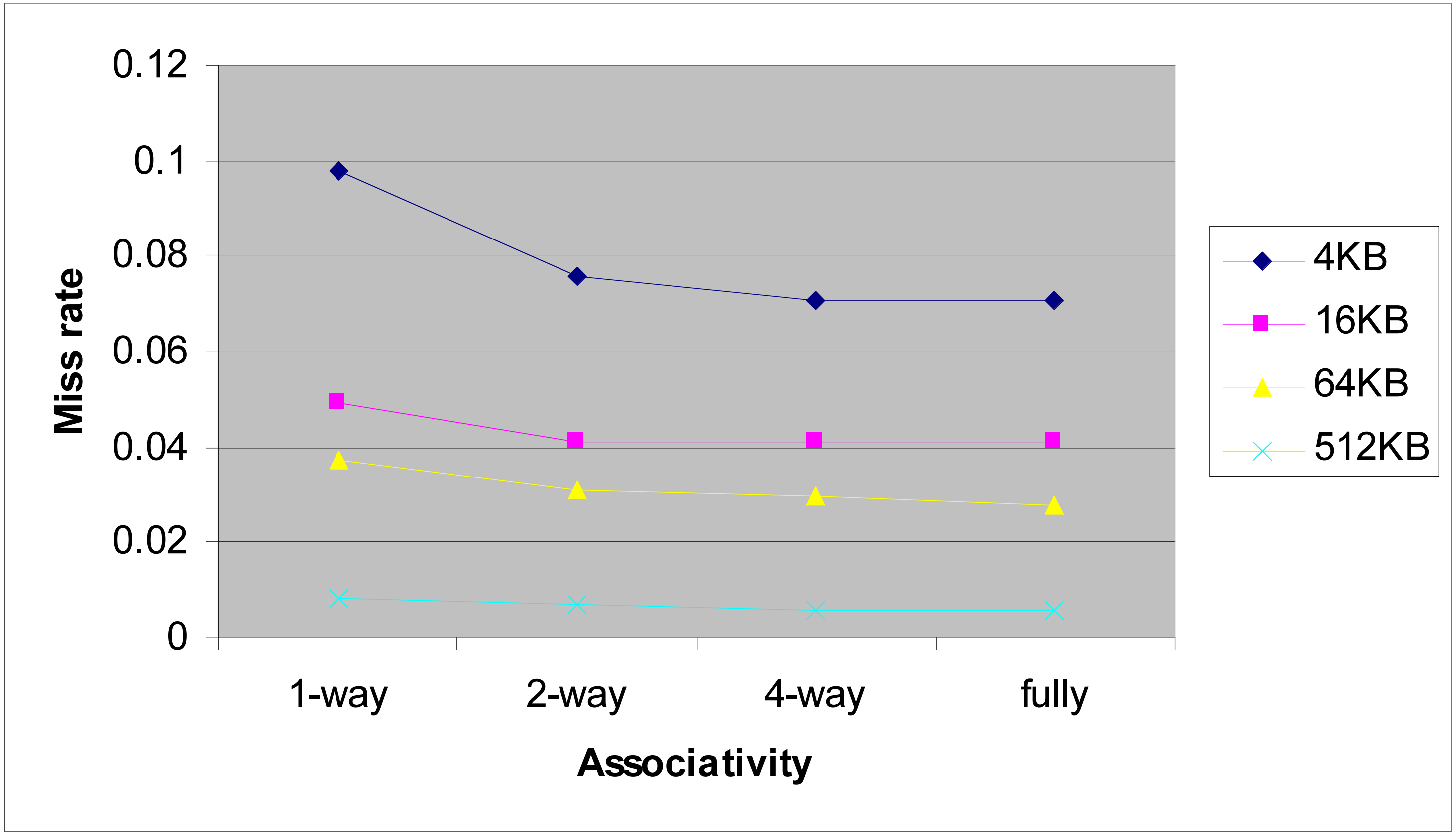
```
while (student) {
  prefetch(student->next);
  student->mark = rand();
  student=student->next;
}
```

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

Reducing Miss Rates

Technique 4: Compiler optimizations

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

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

```
struct valkey{
  int val;
  int key;
};
Struct valkey 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];
```

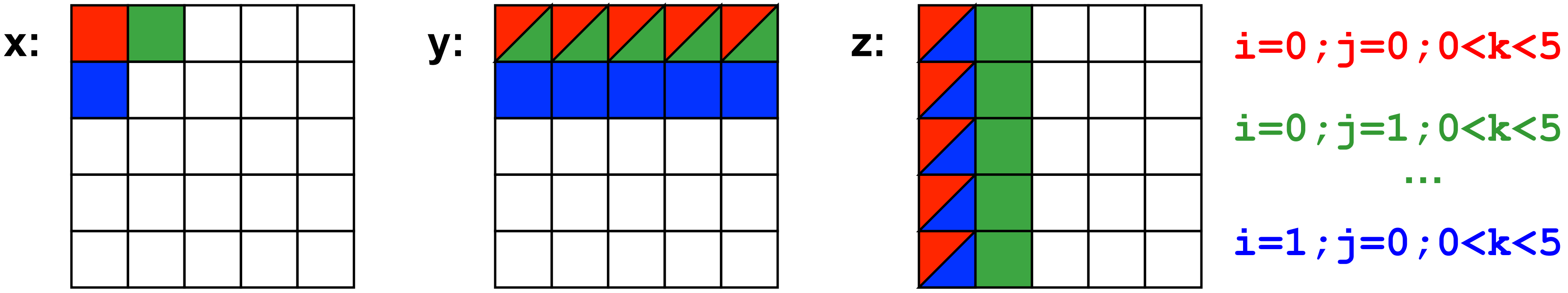
```
for (i=0; i<1000; i++) {
  A[i] = A[i]+1;
  B[i] = B[i]+A[i];
}
```

Reducing 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;
    }
```

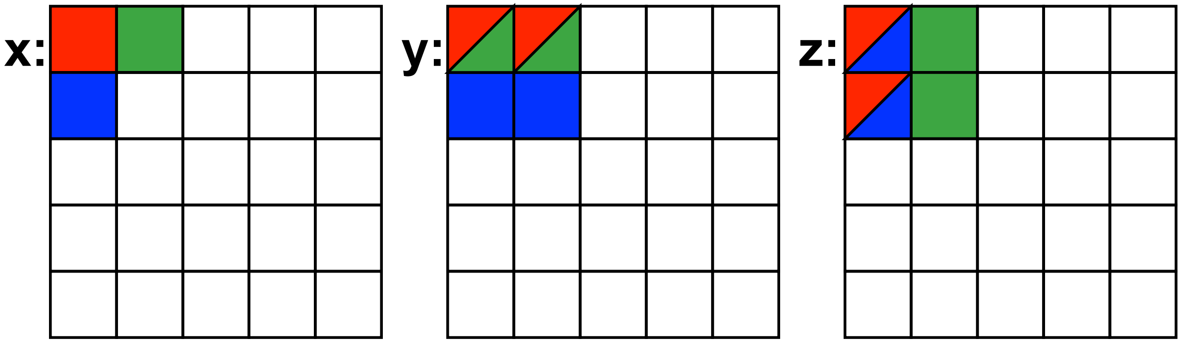
// matrix multiplication



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:

CPU time = IC x CPI x Clock time → CPU performance eqn.

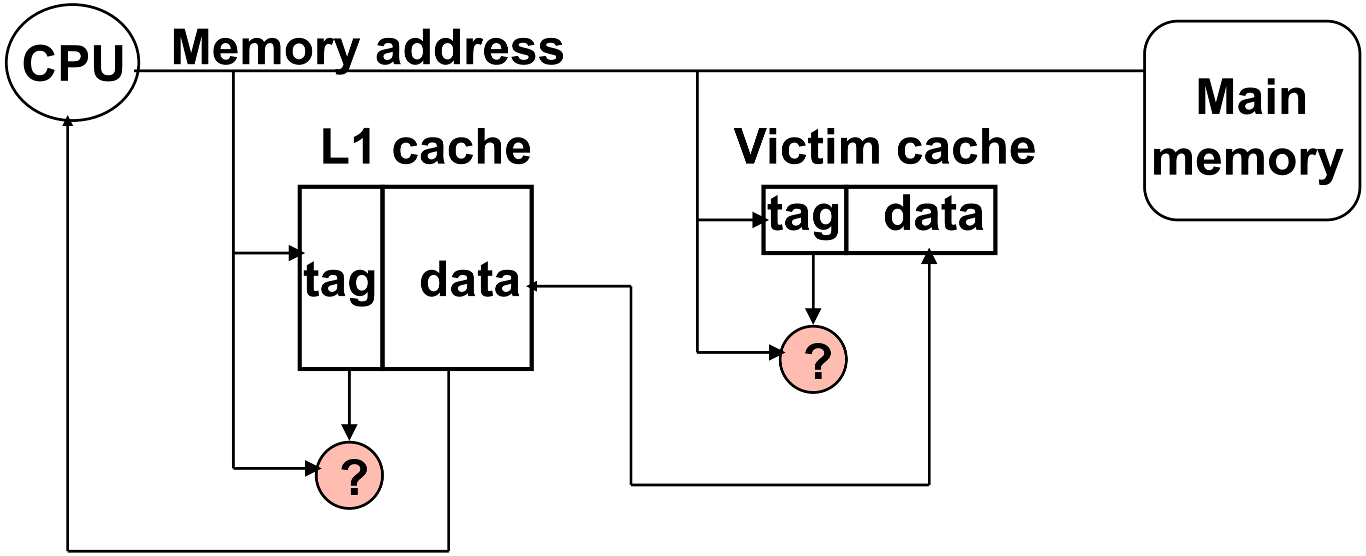
Avg. mem. time = Hit time + Miss rate x Miss penalty → Memory performance eqn.

- 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

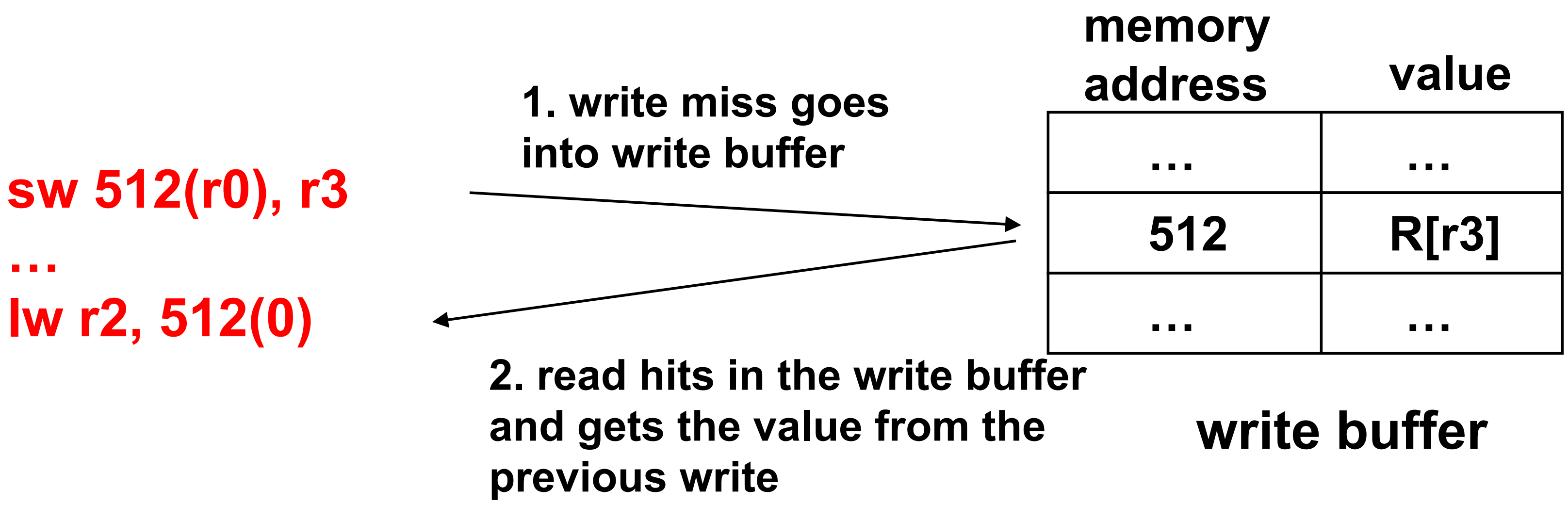
- (Can also be considered to reduce miss rate)
- Very small cache used to capture evicted lines from cache
  - Targets conflict misses
- In case of cache miss the data may be found quickly in the victim cache
- Typically 8-32 entries, fully-associative
- Access victim cache in series or in parallel with main cache: trade-off?



Reducing Cache Miss Penalty

Technique 2: giving priority to reads over writes

- The value of a a load is likely to be used soon (i.e., dependent inst may stall), while writes are “fire-and-forget”
  - Insight: writes are “off the critical path” and their latency doesn’t usually matter. Thus, don’t stall for writes!
- Idea: place write misses in a write buffer, and let read misses overtake writes
  - Flush the writes from the write buffer when pipeline is idle or when buffer full
- Reads to the memory address of a pending write in the buffer now become hits in the buffer:



```
sw 512(r0), r3
…
lw r2, 512(0)
```

1. write miss goes into write buffer

2. read hits in the write buffer and gets the value from the previous write

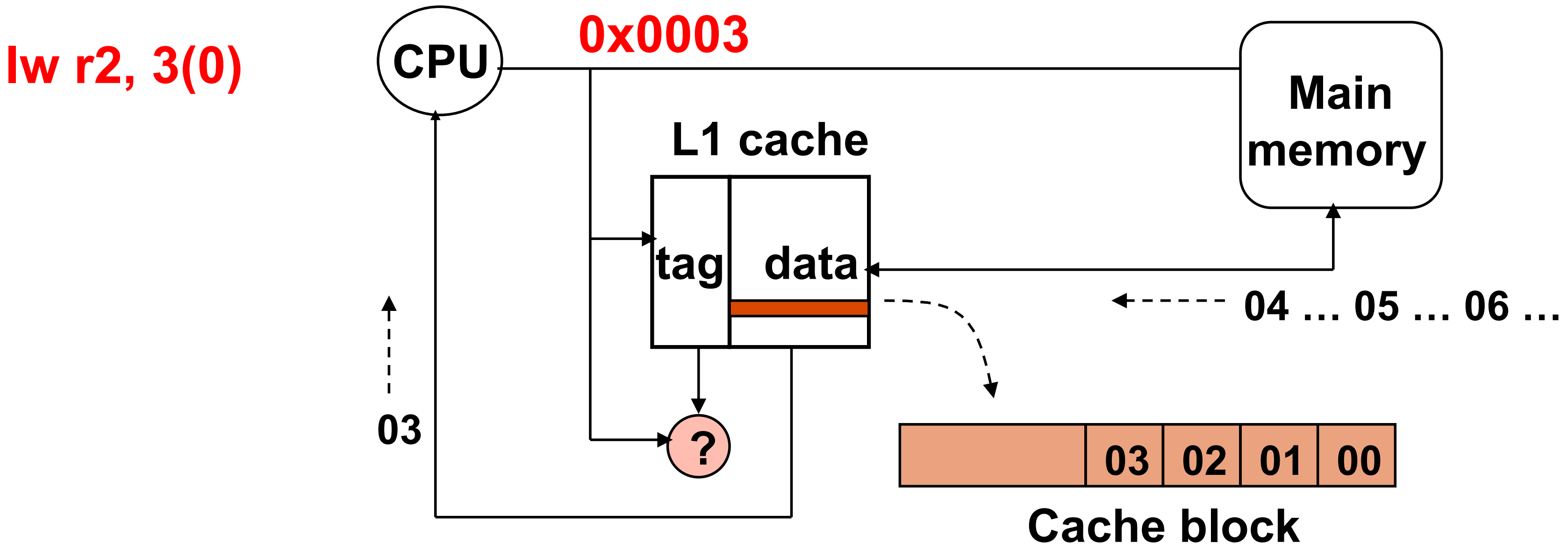
memory address	value
…	…
512	R[r3]
…	…

write buffer

Reducing Cache Miss Penalty

Technique 3: early restart and critical word first

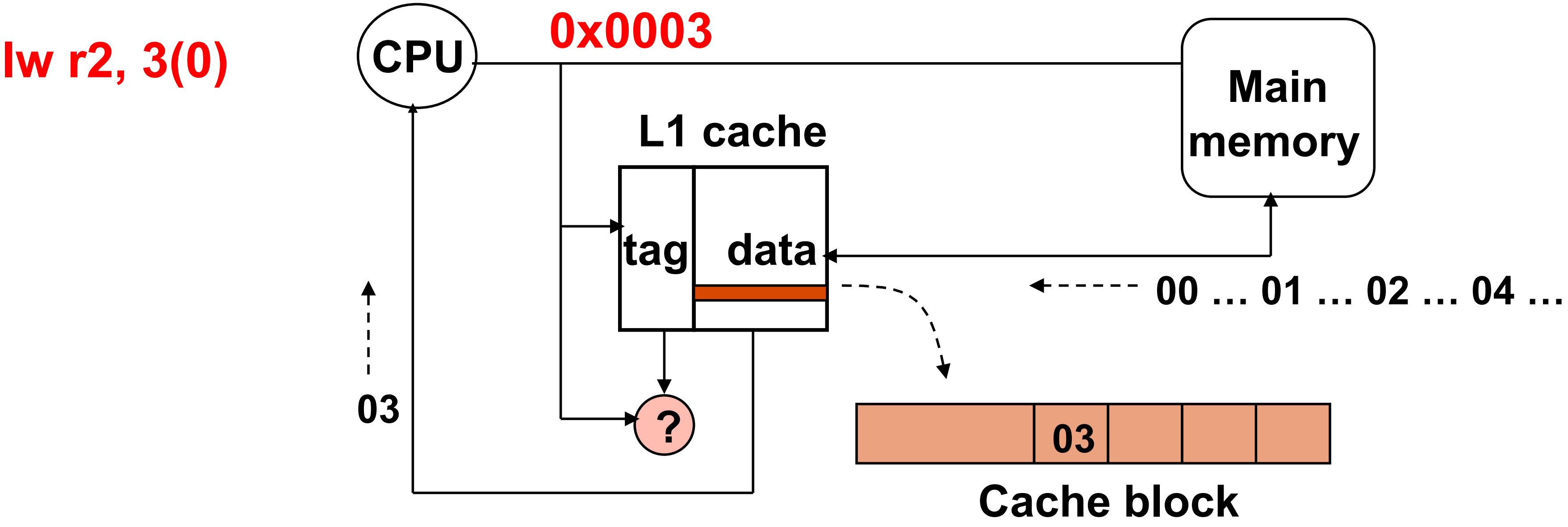
- On a read miss, processor needs just the requested word (or byte)
  - but processor must wait until the whole block is brought into the cache
- Early restart: as soon as the requested word arrives in the cache, send it to the processor
  - Meanwhile, continue reading the rest of the block into the cache



Reducing Cache Miss Penalty

Technique 3: early restart and critical word first

- On a read miss, processor needs just the requested word (or byte)
    - but processor must wait until the whole block is brought into the cache
- Critical word first: get the requested word first from the memory and immediately send it to the processor
    - Meanwhile, continue reading the rest of the block into the cache

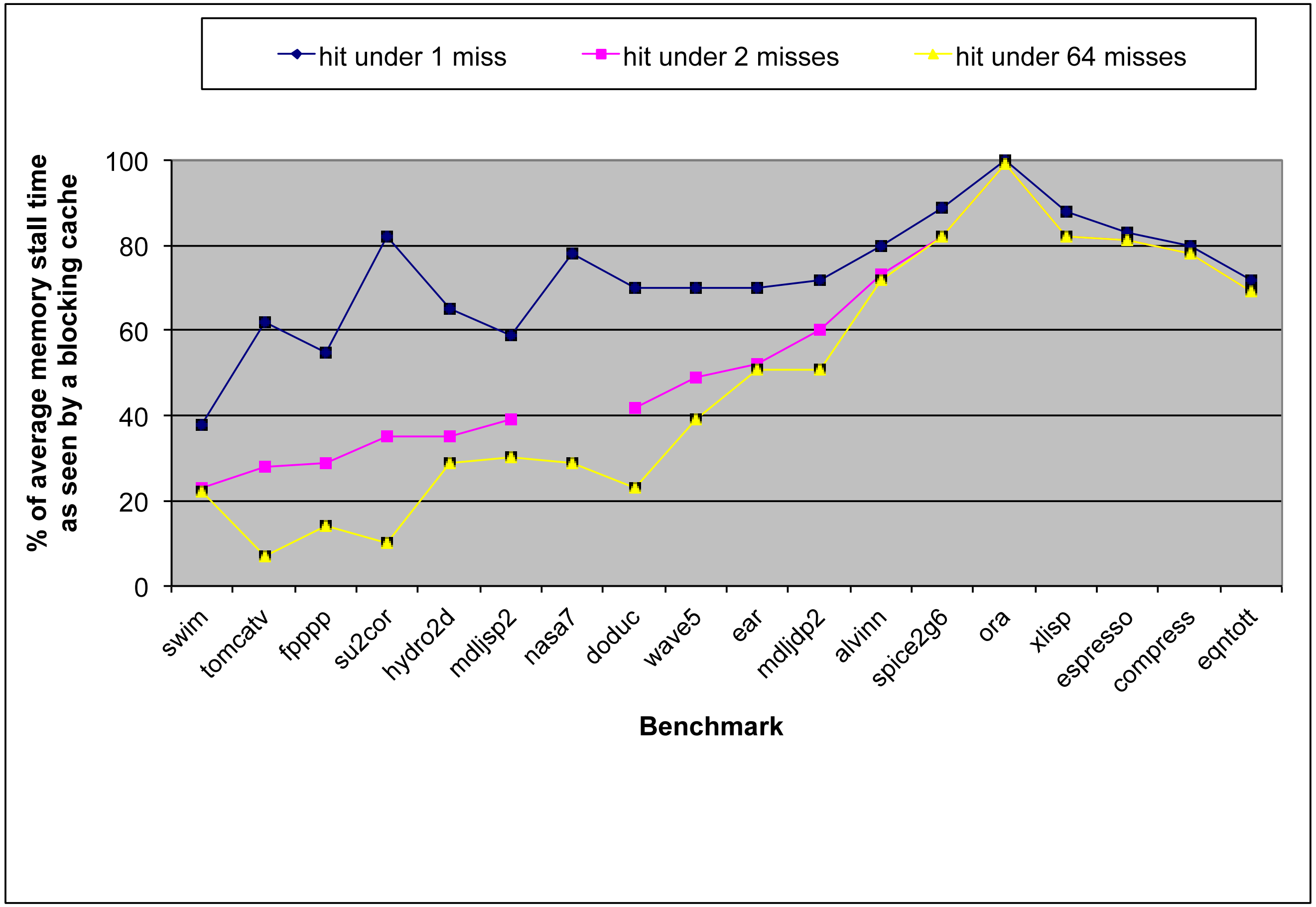


Reducing Cache Miss Penalty

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

- Non-blocking caches: other memory instructions can 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
  - Particularly valuable in dynamically scheduled (out-of-order) processors
- Must keep track of multiple outstanding memory operations
  - New hardware structure: Miss Status Handler Registers (MSHRs)
    - Address of a block being waited on
    - Capability to merge multiple requests to the same block
    - Destination register

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 5: 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 – 4MB capacity (last level of cache in smartphones & tablets)
  - ~10-20 cycles access time
  - Higher associativity (e.g., 8-16 ways) possible. Why?
  - Higher miss rate than L1. Why?
- L3 caches are common on laptop/desktop/server processors
  - 30+ cycle access time
  - 2-20+ MB capacity
  - Very high associativity (16-32 ways)

Second Level Caches

- Memory subsystem performance:

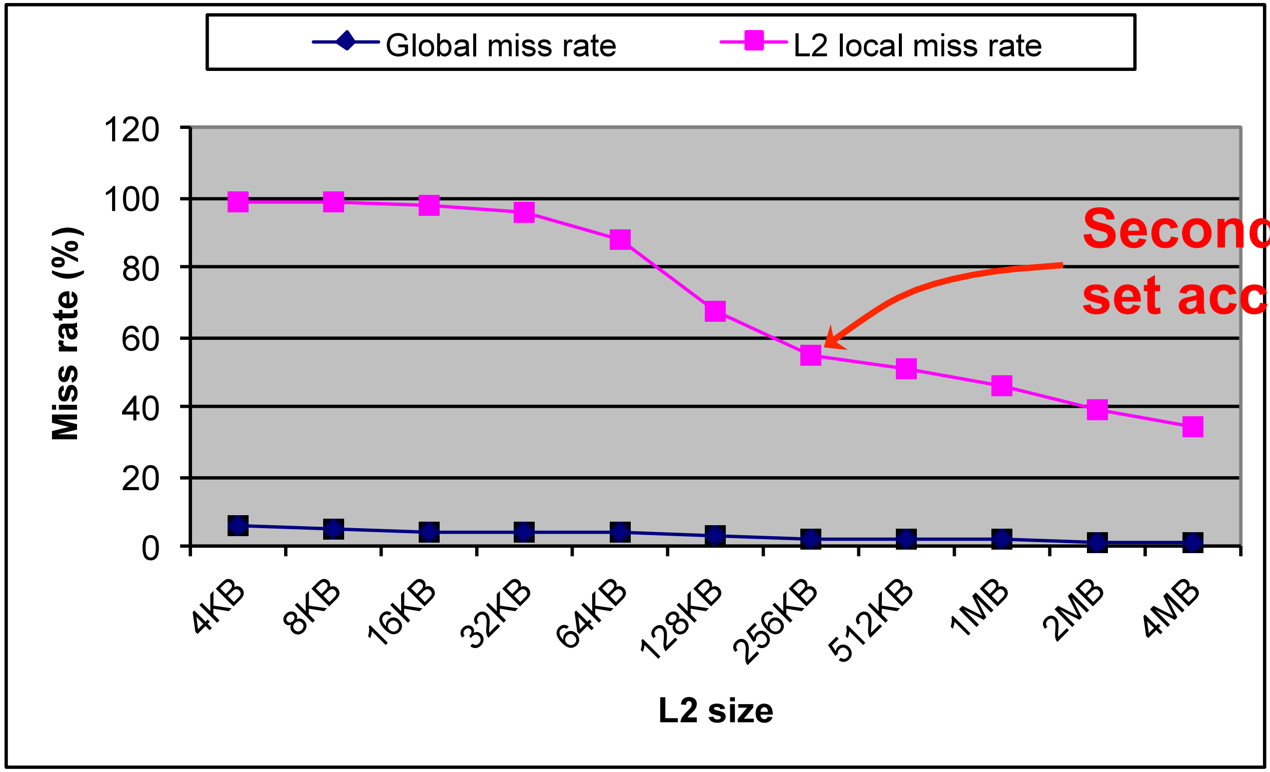
$$\textbf{Avg. mem. time} = \textbf{Hit time}_{L1} + \textbf{Miss rate}_{L1} \times \textbf{Miss penalty}_{L1}$$

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

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

- 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 cache hierarchy

Cache Misses vs. L2 size



Secondary working set accommodated

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:

CPU time = IC x CPI x Clock time → CPU performance eqn.

Avg. mem. time = Hit time + Miss rate x Miss penalty → 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

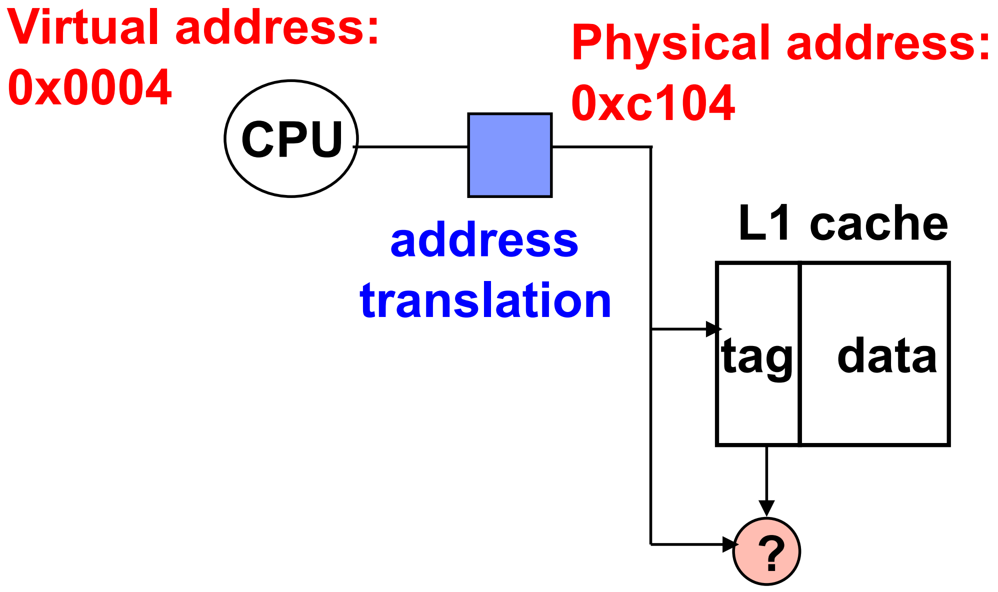
Technique 1: small and simple caches

- Small caches are compact → have short wire spans
  - Wires are slow
- 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-addressed caches

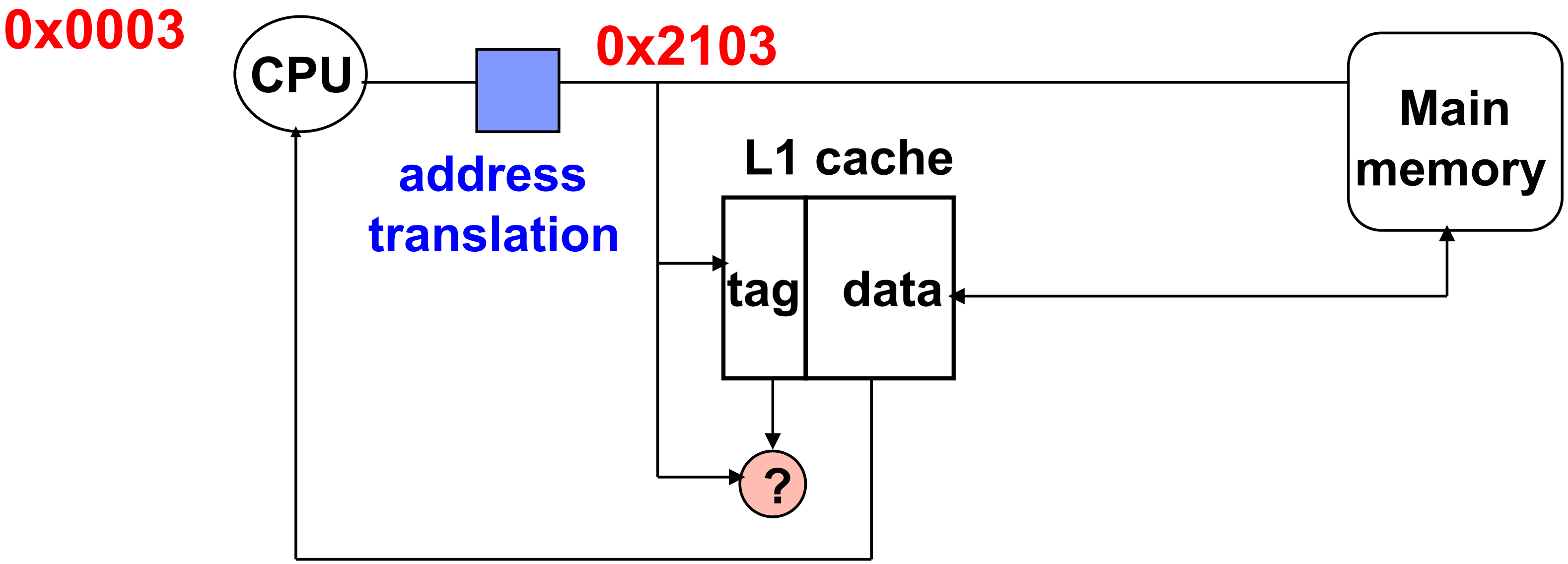
- Programs use virtual addresses for data, while main memory uses physical addresses → addresses from processor must be translated at some point



Reducing Cache Hit Time

Technique 2: virtual address caches

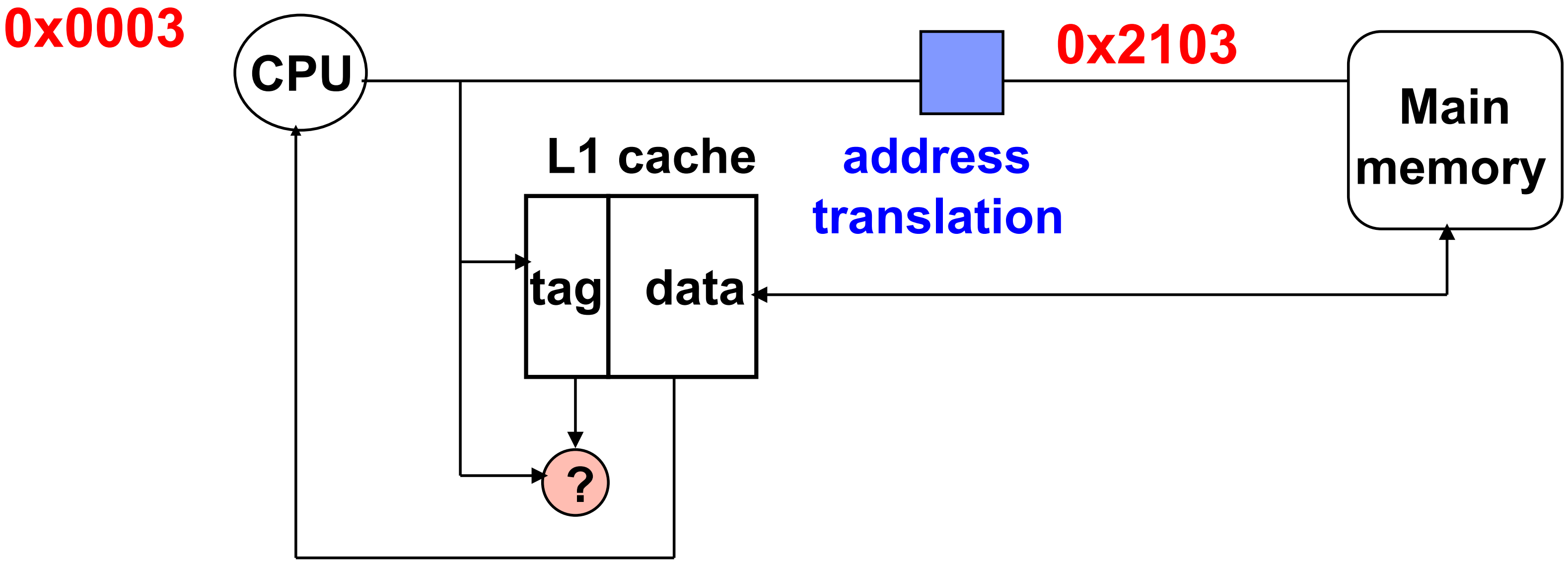
- Programs use virtual addresses for data, while main memory uses physical addresses → addresses from processor must be translated at some point
- Option 1: physical address caches → perform address translation before cache access
  - Hit time is increased to accommodate translation



Reducing Cache Hit Time

Technique 2: virtual address caches

- Option 2: virtual address caches → perform address translation after cache access if miss
  - Hit time does not include translation



Discussed in “Virtual Memory” lecture

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-addressed caches			☺	☹