



《计算机系统结构》课程直播

2020. 3.19

请将ZOOM名称改为“姓名”；

听不到声音请及时调试声音设备，可以下课后补签到

本次课讲授内容

1. Cache miss classification
2. Cache Performance Equation
3. Cache Performance Techniques

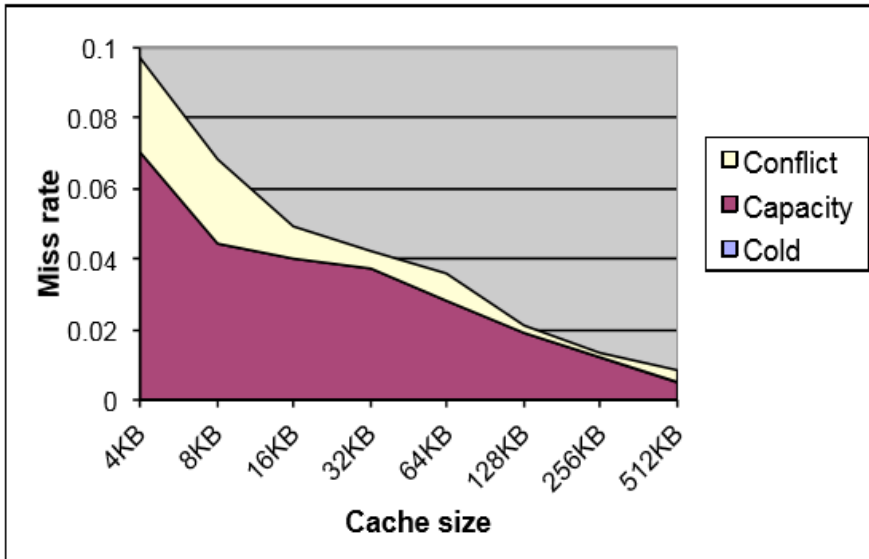
From : H&P Computer Architecture

Cache miss classification

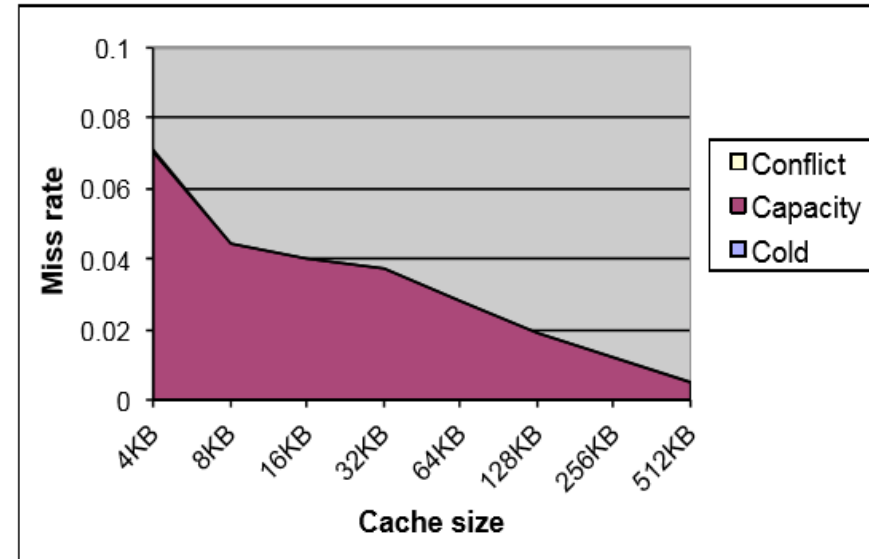
- Cache miss classification: the “three C’s”
- **Compulsory misses** (or cold misses):
 - when a block is accessed for the first time
- **Capacity misses**
 - when a block is not in the cache because it was evicted because the cache was full
- **Conflict misses**
 - when a block is not in the cache because 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 miss Vs. Cache Size

Direct mapped



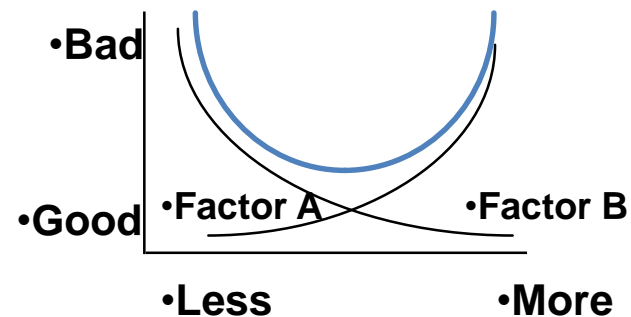
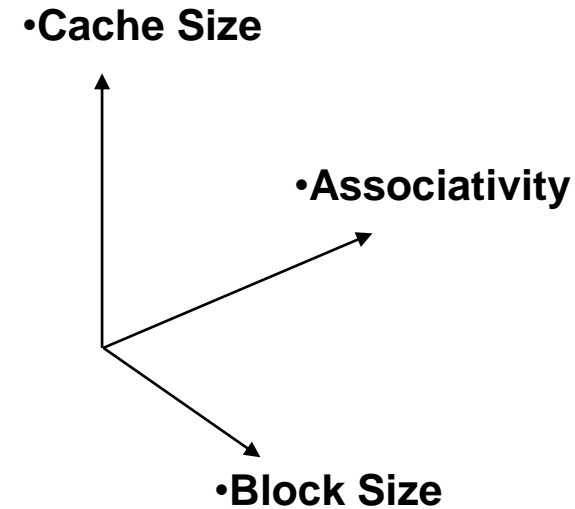
4-way set associative



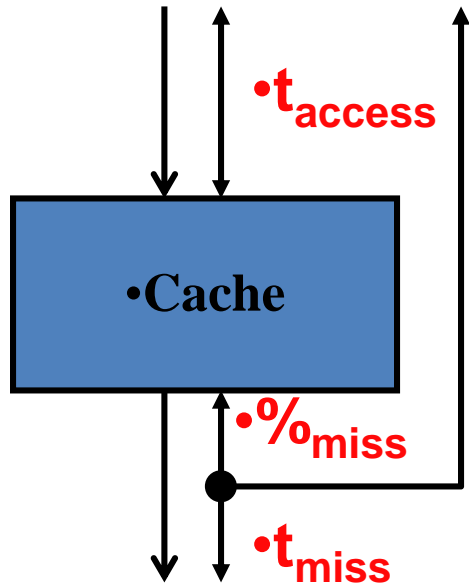
- Miss rates are very small in practice (caching is effective!)
- Miss rates decrease significantly with cache size
 - Rule of thumb: miss rates change in proportion to $\sqrt{\text{cache size}}$
 - e.g., 2x cache $\rightarrow \sqrt{2}$ fewer misses
- Miss rates decrease with set-associativity because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

The Cache Design Space

- Several interacting dimensions
 - cache size
 - block size
 - associativity
 - replacement policy
 - write-through vs write-back
 - write allocation
- The optimal choice is a compromise
 - depends on access characteristics
 - workload
 - use (I-cache, D-cache, TLB)
 - depends on technology / cost
- Simplicity often wins



Cache Performance Equation



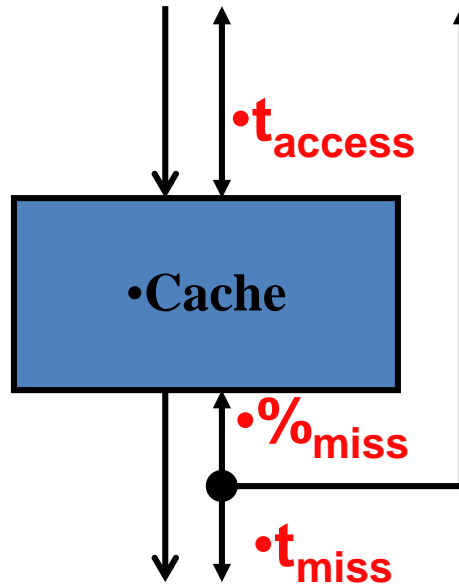
- For a cache
 - **Access**: read or write to cache
 - **Hit**: desired data found in cache
 - **Miss**: desired data not found in cache
 - Must get from another component
 - No notion of “miss” in register file
 - **Fill**: action of placing data into cache
 - **%_{miss}** (miss-rate): #misses / #accesses
 - **t_{access}**: time to check cache. If hit, we're done.
 - **t_{miss}**: time to read data into cache

- Performance metric: average access time

$$t_{avg} = t_{access} + (\%_{miss} * t_{miss})$$

Measuring Cache Performance

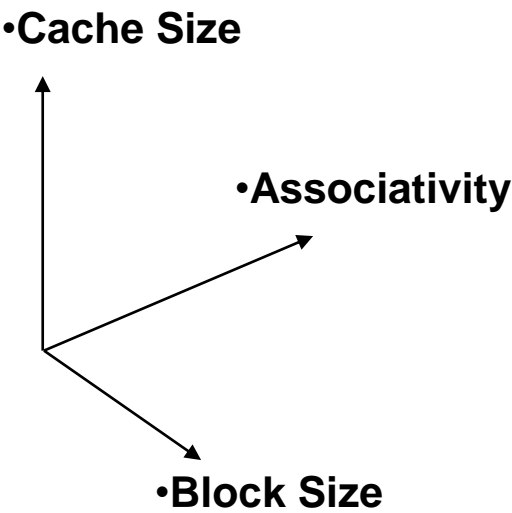
- Performance metric: average access time
- $t_{avg} = t_{access} + (\%_{miss} * t_{miss})$
- Improving memory hierarchy performance:
 - Decrease hit time: t_{access}
 - Decrease miss rate: $\%_{miss}$
 - Decrease miss penalty: t_{miss}



Cache Performance Techniques

technique	Miss rate	Miss Penalty	Hit time	Complexity
small and simple caches				
large block size				
high associativity				

$t_{avg} = \text{Hit time} + (\text{Miss rate} * \text{Miss Penalty})$



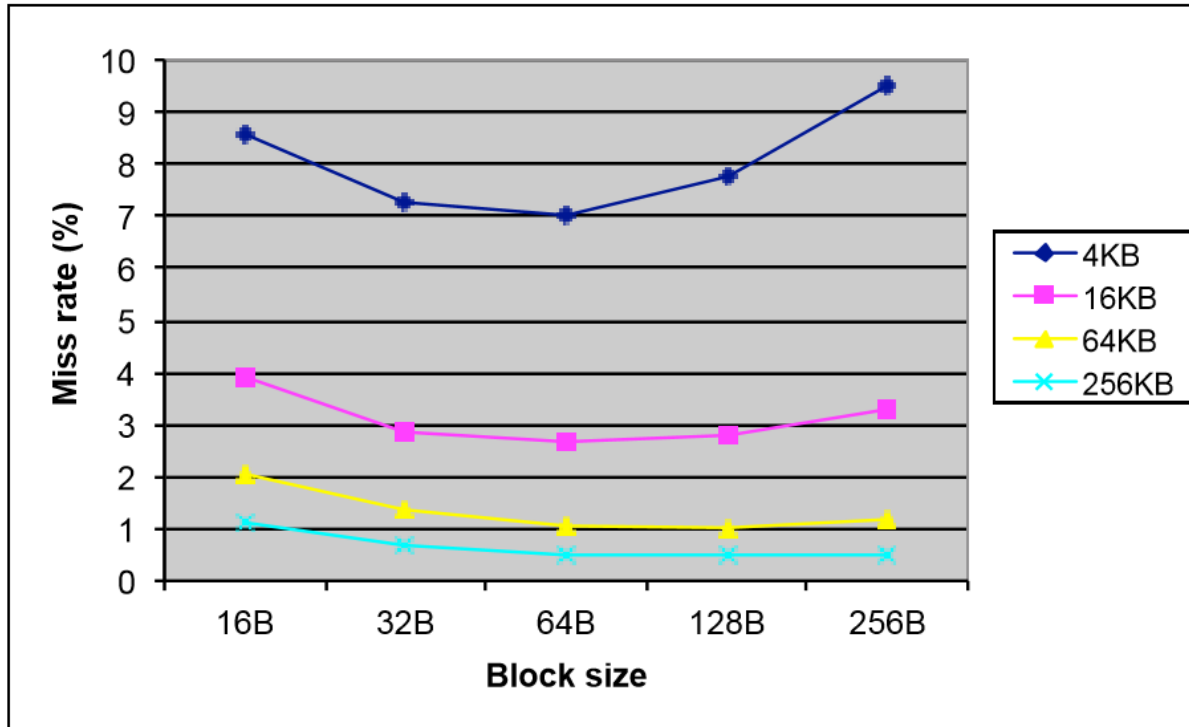
Cache Performance Techniques

technique	Miss rate	Miss Penalty	Hit time	Complexity
small and simple caches	☹		☺	☺

- Small caches are compact and have short wire spans
 - Wires are slow
- Direct mapped caches have only one tag to compare and comparison can be done in parallel with the fetch of the data

Cache Performance Techniques

technique	Miss rate	Miss Penalty	Hit time	Complexity
large block size	😊	😞		😊

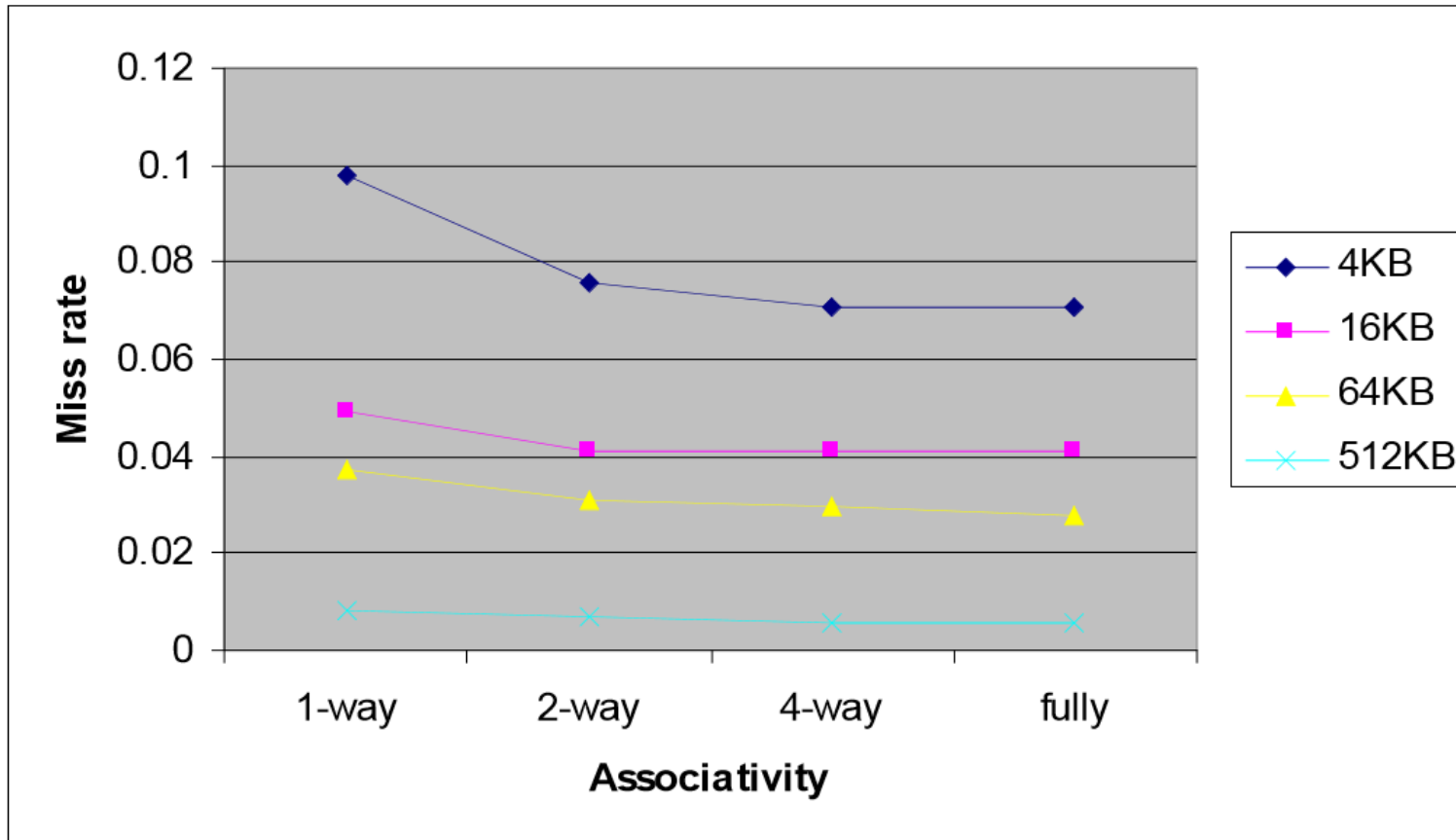


Small caches are very sensitive to block size

- Given capacity, increase **block size**
 - Exploit **spatial locality**
 - 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
 - Very large blocks (> 128B) never beneficial
 - 64B is a common choice

Cache Performance Techniques

technique	Miss rate	Miss Penalty	Hit time	Complexity
high associativity	😊		😞	😞

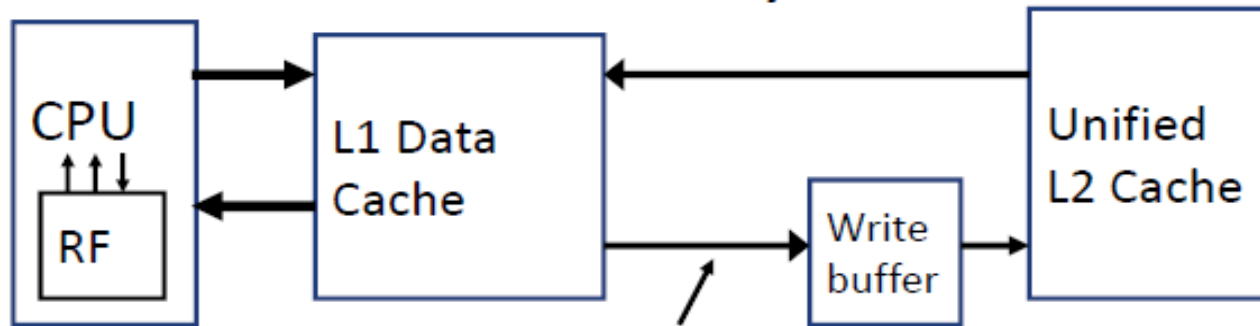


- Small caches are very sensitive to associativity
- In all cases more associativity decreases miss rate, **but little difference between 4-way and fully associative**

Improving Cache Performance

1. Reduce the hit time

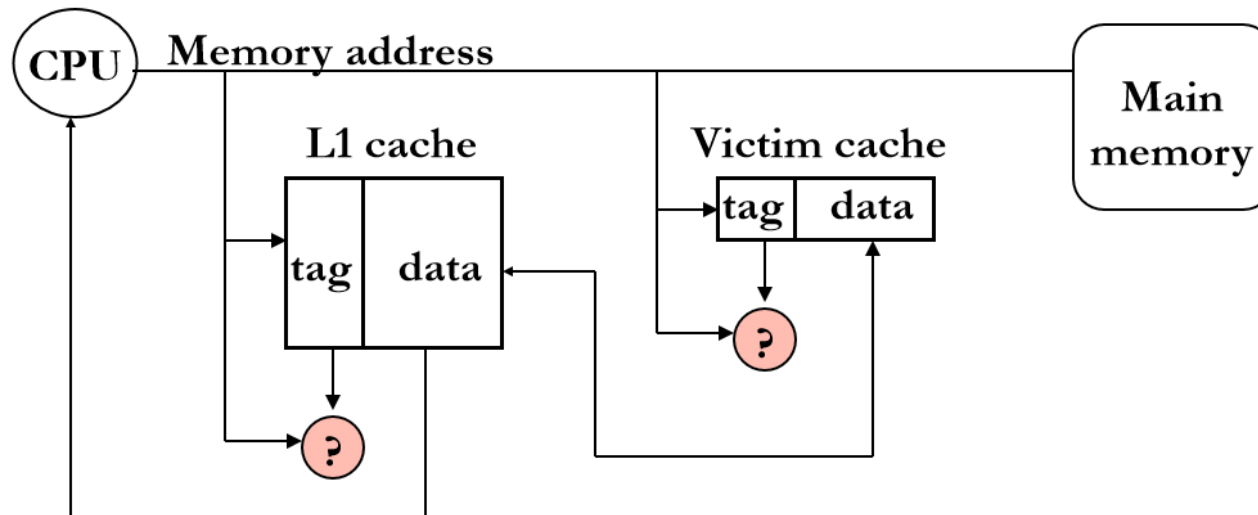
- ✓ smaller&simple cache
- ✓ direct mapped cache
- ✓ smaller blocks
- for writes
 - write to **write buffer**



Improving Cache Performance

2. Reduce the miss rate

- bigger cache (reduce capacity miss)
- more flexible placement (increase associativity, reduce conflict)
- larger blocks (16 to 64 bytes typical, reduce Compulsory miss)
- victim cache
 - small buffer holding most recently discarded blocks from cache



Improving Cache Performance

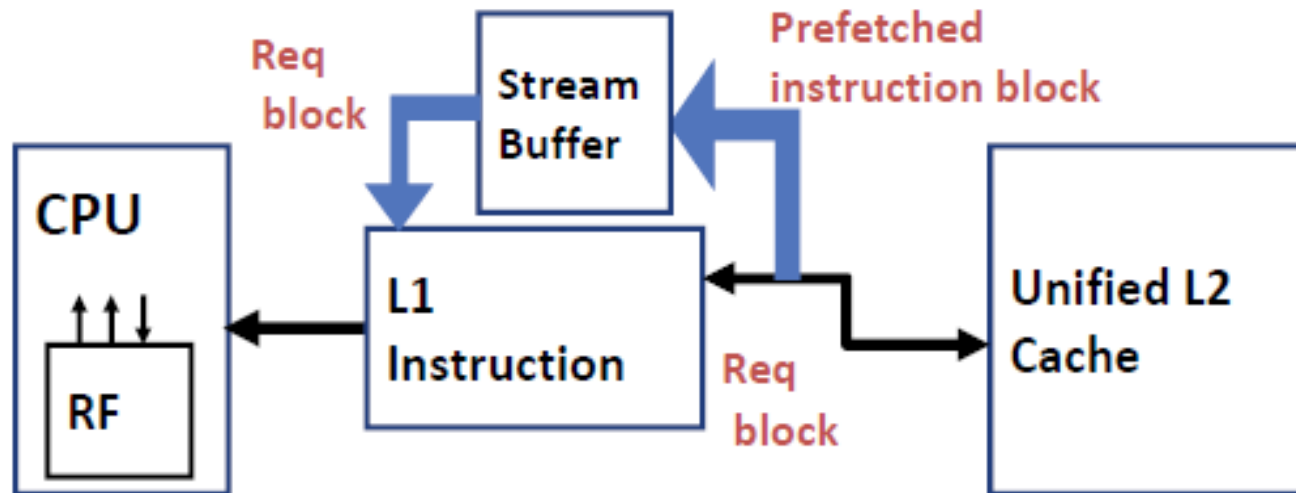
2. Reduce the miss rate (continue)

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)
 - E.g., data prefetching in loops, linked list prefetching

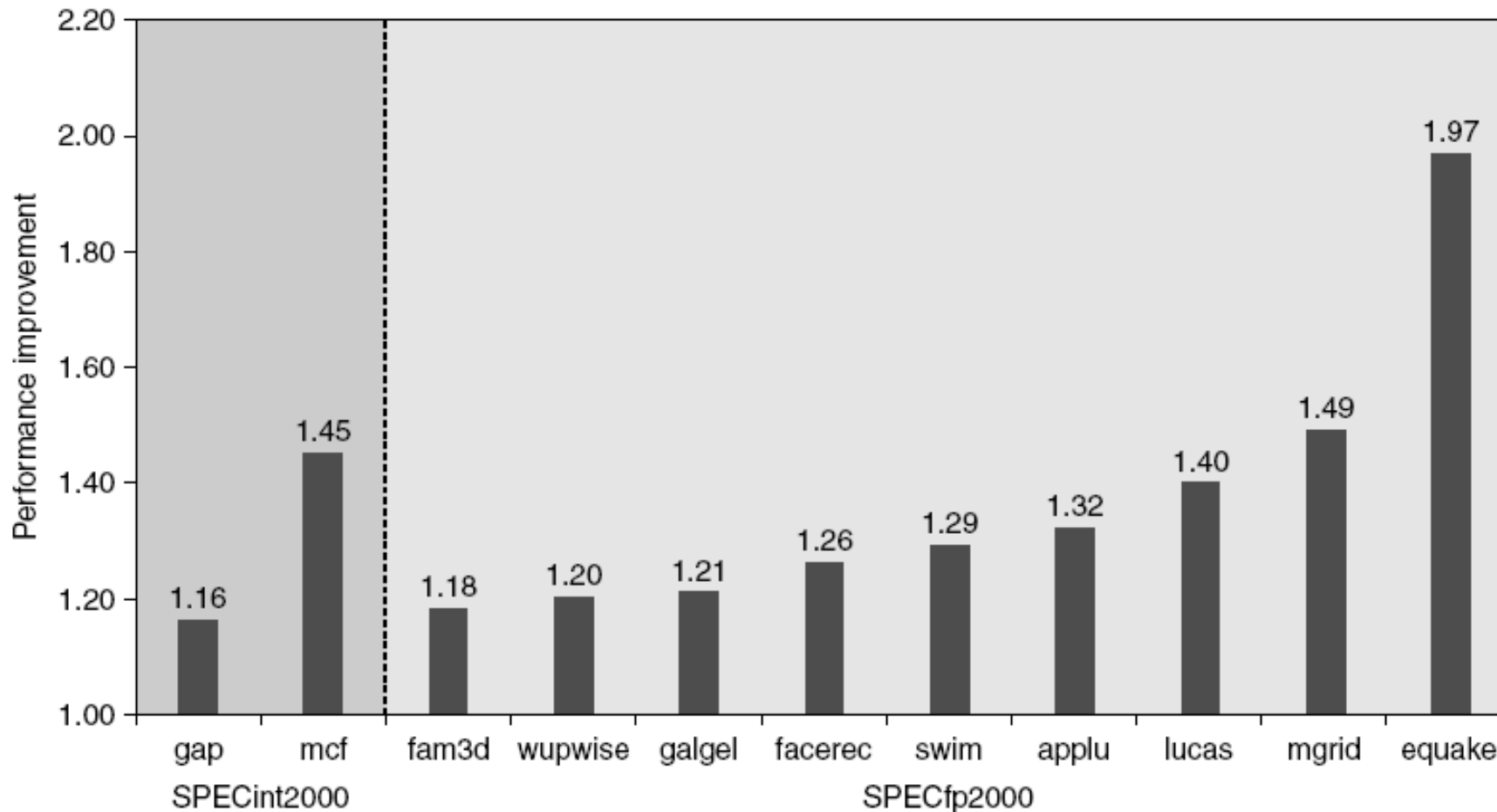
Hardware Instruction Prefetching

- Instruction prefetch in Alpha AXP 21064
 - Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
 - If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)
 - Strided prefetch: if observe sequence of accesses to block b, b+N, b+2N, then prefetch b+3N etc.



Hardware Prefetching

Fetch two blocks on miss (include next sequential block)



•Pentium 4 Pre-fetching

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).

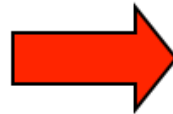
```
for (i=0; i<=999; i++) {  
    x[i] = x[i] + s;  
}
```



```
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;  
}
```


Improving Cache Performance

2. Reduce the miss rate (continue)

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];  
}
```

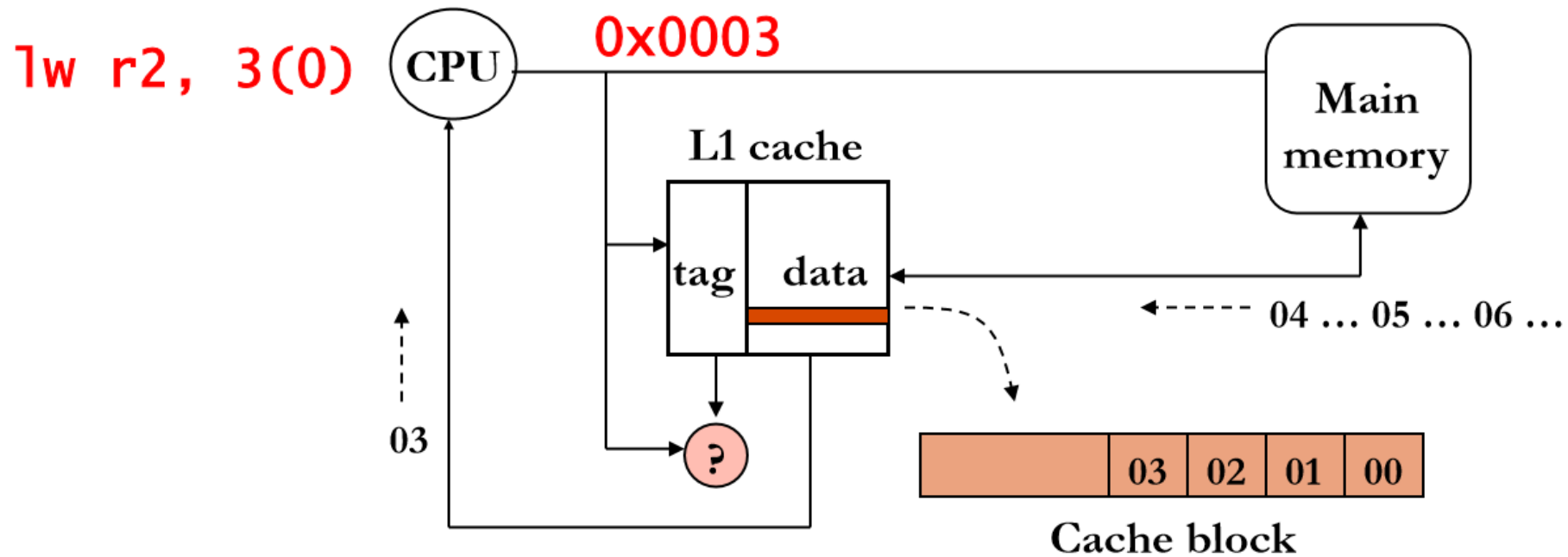
Improving Cache Performance

3. Reduce the miss penalty

- smaller blocks
- use a **write buffer** to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check write buffer (and/or **victim cache**) on read miss
- for large blocks fetch:
 - **early restart and critical word first**
- **giving priority to reads over writes**
- **Non blocking cache**
- use **multiple cache levels** – L2 cache not tied to CPU clock rate

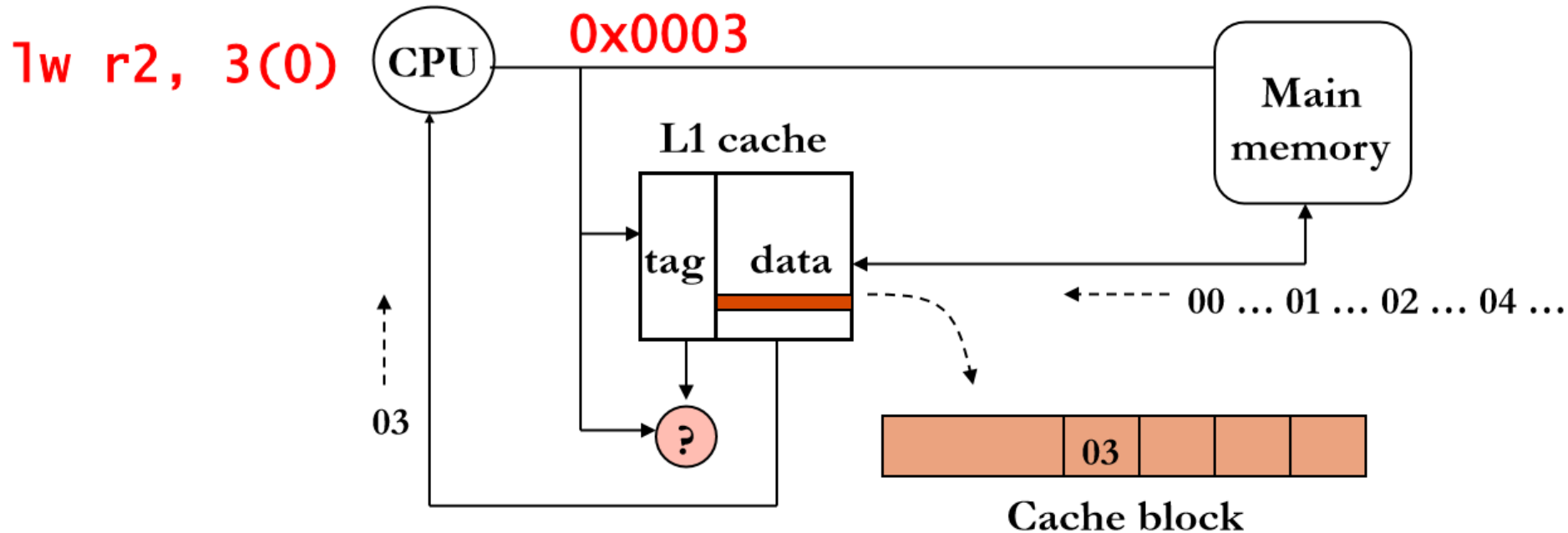
Early restart

- Early restart
 - Request words in normal order
 - Send missed word to the processor as soon as it arrives



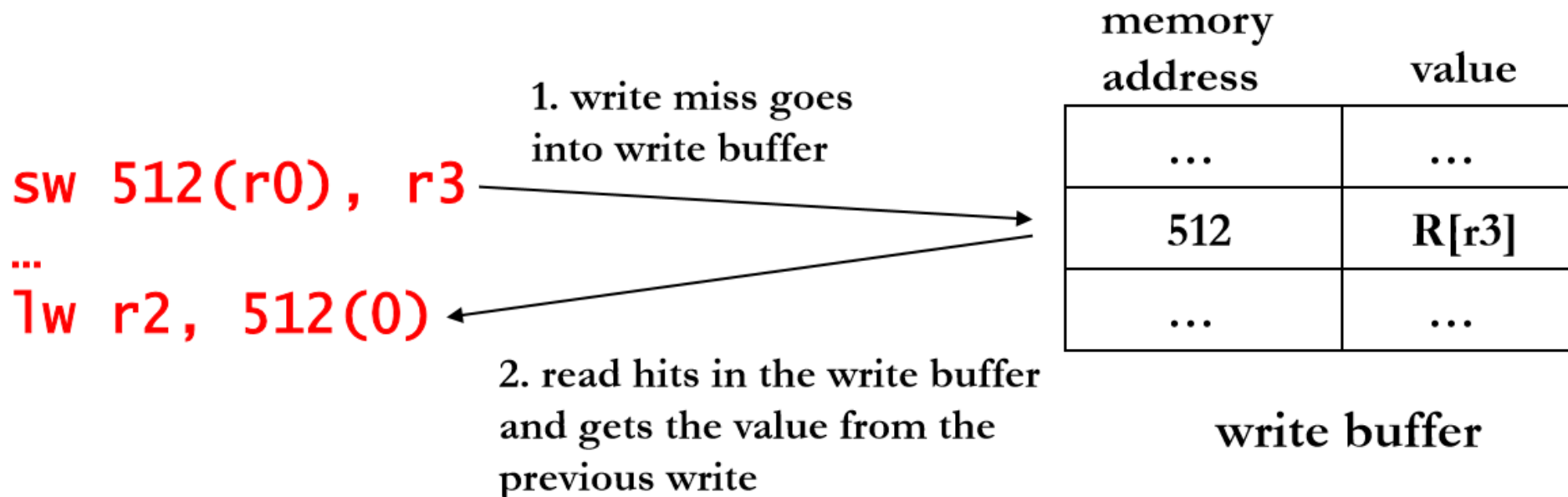
Critical word first

- Critical word first
 - Request missed word from memory first
 - Send it to the processor as soon as it arrives



Giving priority to reads over 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:



Non blocking cache

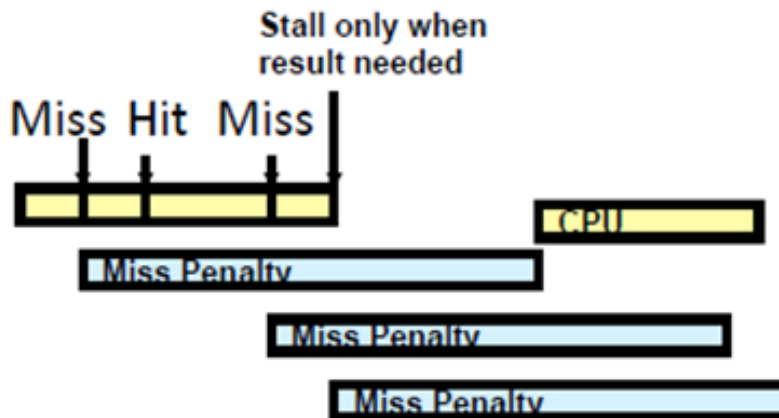
- Cache can service multiple hits while waiting on a miss:
 - “Hit under miss”
 - “Hit under multiple miss”
- L2 must support this (core i7 , ARM A8)
- In general, processors can hide L1 miss penalty but not L2 miss penalty



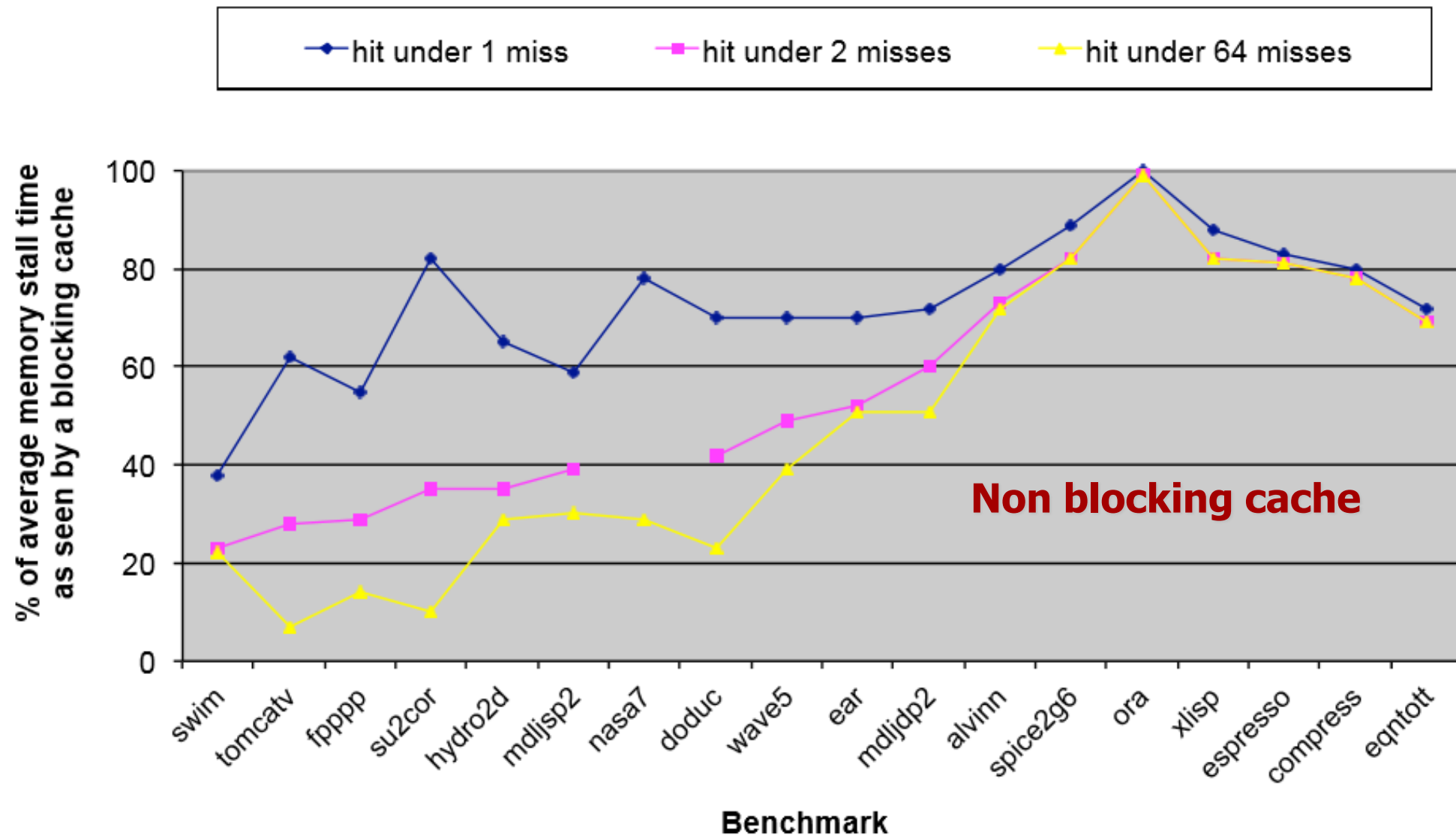
Stall CPU on miss



Hit under miss



Multiple out-standing misses



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

Designing a Cache Hierarchy

- For any memory component: t_{access} vs. $\%_{\text{miss}}$ tradeoff
- Upper components (I\$, D\$) emphasize low t_{access}
 - Frequent access $\rightarrow t_{\text{access}}$ important
 - t_{miss} is not bad $\rightarrow \%_{\text{miss}}$ less important
 - Lower capacity and lower associativity (to reduce t_{access})
 - Small-medium block-size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{\text{miss}}$
 - Infrequent access $\rightarrow t_{\text{access}}$ less important
 - t_{miss} is bad $\rightarrow \%_{\text{miss}}$ important
 - High capacity, associativity, and block size (to reduce $\%_{\text{miss}}$)

Memory Hierarchy Parameters

Parameter	I\$/D\$	L2	L3	Main Memory
t_{access}	2ns	10ns	30ns	100ns
t_{miss}	10ns	30ns	100ns	10ms (10Mns)
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	2-8	4–16	4-16	NA

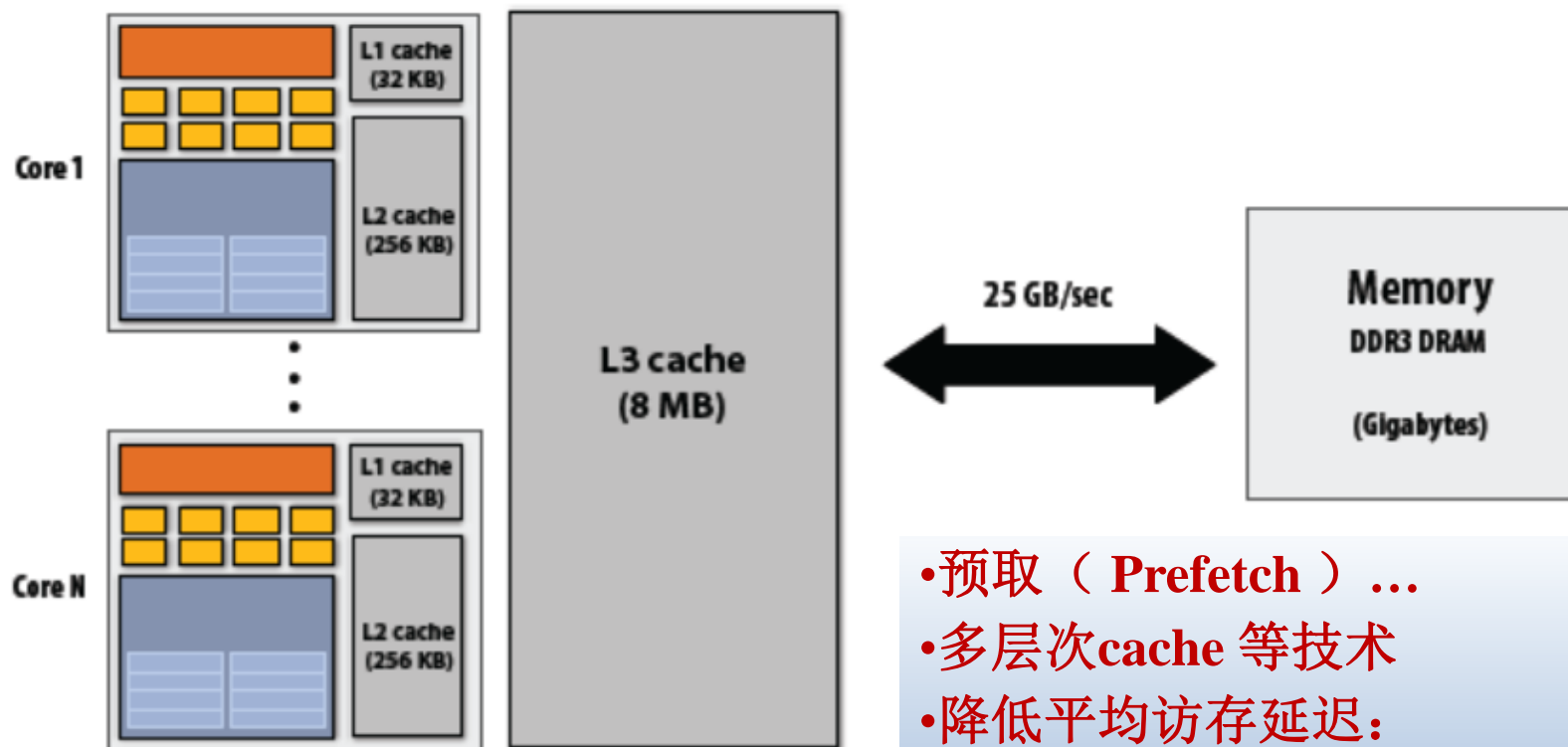
- Some other design parameters
 - Split vs. unified insns/data
 - Inclusion vs. exclusion vs. nothing

Cache Performance Techniques Summary

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			😊	😞

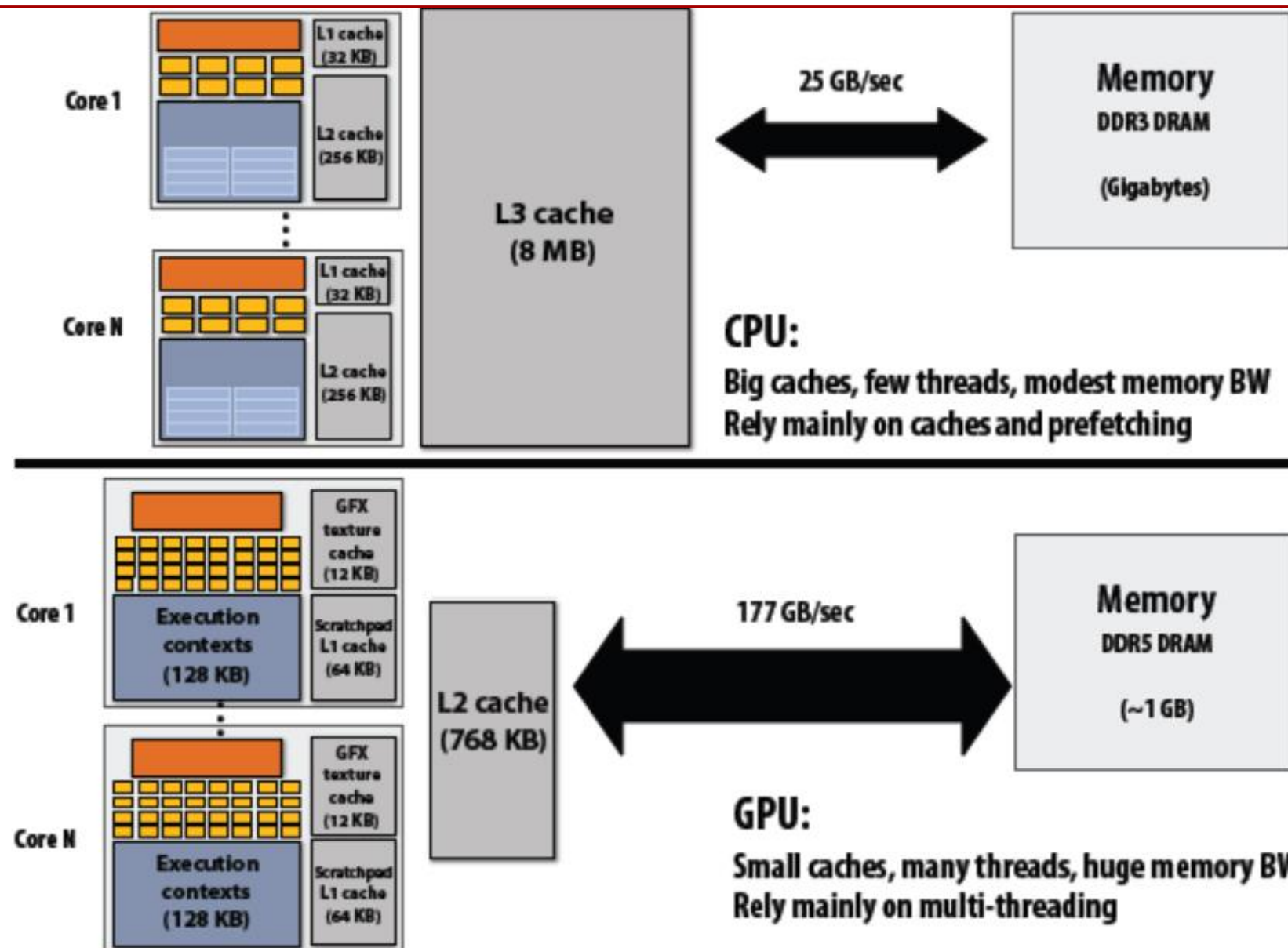
Discussion

CPU Memory hierarchies : 多层次cache



- 预取 (Prefetch) ...
- 多层次cache 等技术
- 降低平均访存延迟:
- Memory access times ~ 100 cycles
- Cache access times ~ 2-30 cycles

CPU vs. GPU memory hierarchies, Why?



Summary

1. Cache Performance Equation
2. Cache Performance Techniques
3. GPU cache and CPU cache

再见

