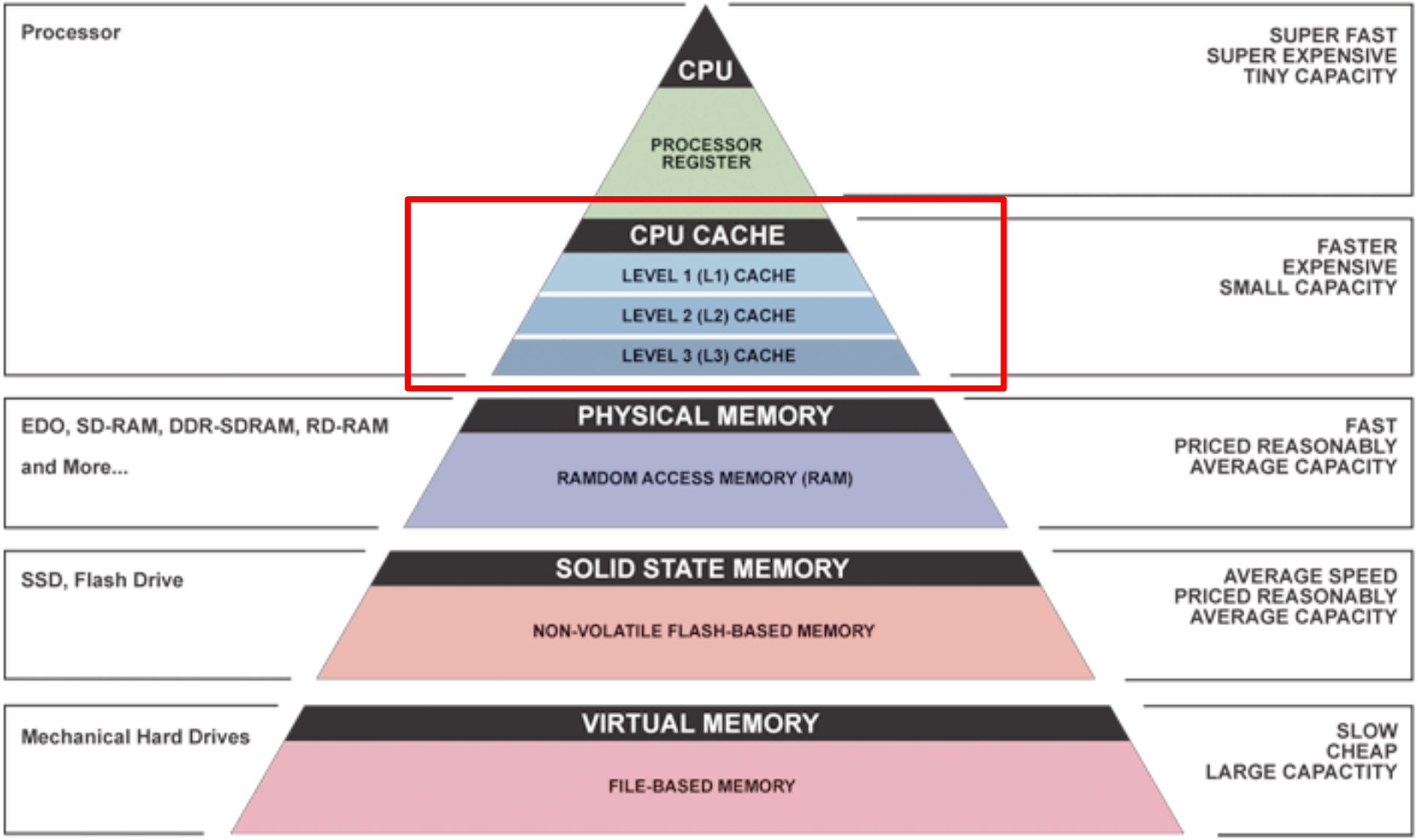
CS 61C: Great Ideas in Computer Architecture

Multilevel Caches, Cache Questions

Instructor: Alan Christopher

Great Idea #3: Principle of Locality/ Memory Hierarchy



Review of Last Lecture (1/2)

- Direct-Mapped Caches:
  - Use hash function to determine location for block
    - Each block maps into a single row
    - (*block* address) modulo (# of *slots* in the cache)
- N-way Set Associative Caches:
  - Split slots into sets of size N, map into set
    - (*block* address) modulo (# of *sets* in the cache)
- TIO breakdown of memory address
  - Index field is result of hash function (which *set*)
  - Tag field is identifier (which block is currently in slot)
  - Offset field indexes into block

Review of Last Lecture (2/2)

- Cache performance measured in CPI_{stall} and AMAT
  - Parameters that matter: HT, MR, MP
- The 3 Cs of cache misses
  - Compulsory, capacity, and conflict

Question: How many total bits are stored in the following cache?

- 4-way SA cache, random replacement
- Cache size 1 KiB, Block size 16 B
- Write-back
- 16-bit address space

(B) $2^6 \times (2^7 + 2^3 + 2^1) = 8.625$ Kib

(G) $2^4 \times (2^7 + 2^3 + 2^0) = 2.140625$ Kib

(P) $2^4 \times (2^7 + 2^3 + 2^1) = 2.15625$ Kib

(Y) $\mathbf{2^4 \times (2^7 + 6 + 2^1) = 2.125}$ **Kib**

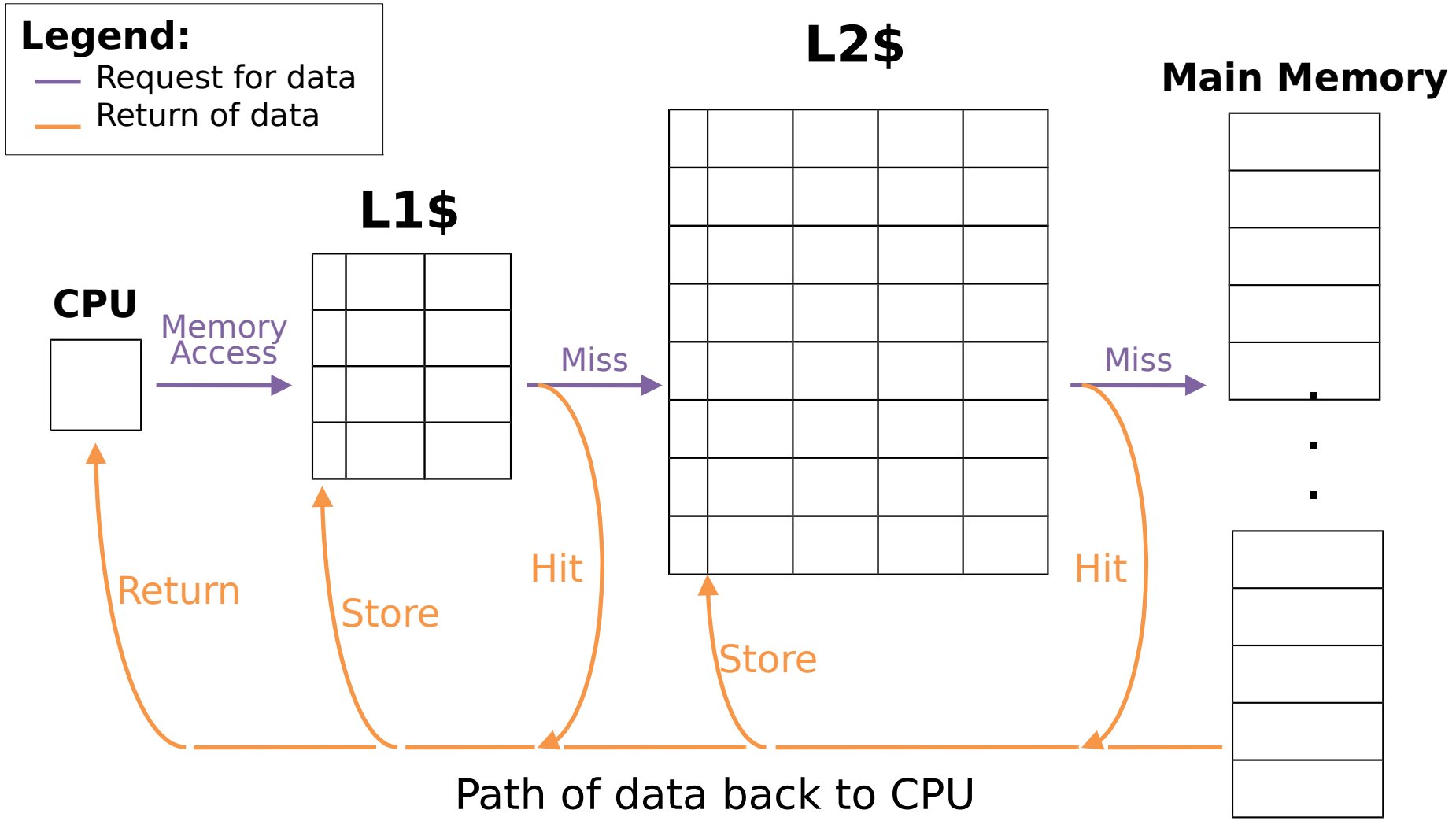
Agenda

- Multilevel Caches
- Administrivia
- Improving Cache Performance
- Anatomy of a Cache Question
- Example Cache Questions

Multiple Cache Levels

- With advancing technology, have more room on die for bigger L1 caches and for L2 (and in some cases even L3) cache
  - Normally lower-level caches are *unified* (i.e. holds both instructions and data)
- Multilevel caching is a way to reduce miss penalty
- So what does this look like?

Multilevel Cache Diagram



Multilevel Cache AMAT

- AMAT = L1 HT + L1 MR × L1 MP
  - Now L1 MP depends on other cache levels
- L1 MP = L2 HT + L2 MR × L2 MP
  - If more levels, then continue this chain
  - (i.e. $MP_i = HT_{i+1} + MR_{i+1} \times MP_{i+1}$)
  - Final MP is main memory access time
- For two levels:

  AMAT = L1 HT + L1 MR × (L2 HT + L2 MR × L2 MP)

Multilevel Cache AMAT Example

- **Processor specs:** 1 cycle L1 HT, 2% L1 MR, 5 cycle L2 HT, 5% L2 MR, 100 cycle main memory HT
  - Here assuming unified L1$
- Without L2$:

$$AMAT_1 = 1 + 0.02 \times 100 = 3$$

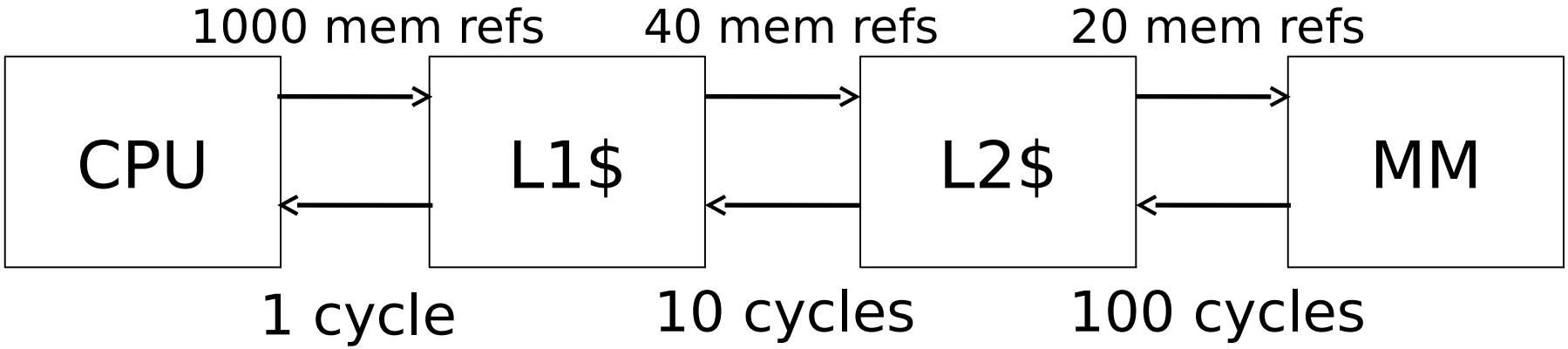
  - With L2$:

$$AMAT_2 = 1 + 0.02 \times (5 + 0.05 \times 100) = 1.2$$

Local vs. Global Miss Rates

- *Local miss rate:* Fraction of references to one level of a cache that miss
  - e.g. L2$ local MR = L2$ misses/L1$ misses
  - Specific to level of caching (as used in AMAT)
- *Global miss rate:* Fraction of all references that miss in all levels of a multilevel cache
  - Property of the overall memory hierarchy
  - Global MR is the product of all local MRs
    - Start at Global MR = Ln misses/L1 accesses and expand
    - So by definition, *global MR ≤ any local MR*

Memory Hierarchy with Two Cache Levels



- For every 1000 CPU to memory references
  - 40 will miss in L1$; what is the local MR? 0.04
  - 20 will miss in L2$; what is the local MR? 0.5
  - Global miss rate? 0.02

Rewriting Performance

- For a two level cache, we know:

$$MR_{global} = L1\ MR \times L2\ MR$$

- AMAT:
  - $AMAT = L1\ HT + L1\ MR \times (L2\ HT + L2\ MR \times L2\ MP)$
  - $= L1\ HT + L1\ MR \times L2\ HT + MR_{global} \times L2\ MP$
- CPI:
  - $CPI_{stall} = CPI_{base} + (L1\ MR \times L2\ HT + MR_{global} \times L2\ MP) \times \frac{Accesses}{Instr}$

Design Considerations

- L1$ focuses on *low hit time* (fast access)
  - minimize HT to achieve shorter clock cycle
  - L1 MP significantly reduced by presence of L2$, so can be smaller/faster even with higher MR
  - e.g. smaller $ (fewer rows)
- L2$, L3$ focus on *low miss rate*
  - As much as possible avoid reaching to main memory (heavy penalty)
  - e.g. larger $ with larger block sizes (same # rows)

Multilevel Cache Practice (1/3)

- **Processor specs:**
  - CPI_{base} of 2
  - 100 cycle miss penalty to main memory
  - 25 cycle miss penalty to unified L2$
  - 36% of instructions are load/stores
  - 2% L1 I$ miss rate; 4% L1 D$ miss rate
  - 0.5% **global** Unified L2$ miss rate
- What is CPI_{stall} with and without the L2$?

Multilevel Cache Practice (2/3)

- **Notes:**
  - Both L1 I$ and L1 D$ send misses to L2$
  - What does the global L2$ MR mean?
    - MR to main memory
    - Since there are 2 L1$s, implies L2$ has 2 different local MRs depending on source of access
  - Use CPI_{stall} formula shown at bottom of slide 13
    - Remember that CPI is cumulative

Multilevel Cache Practice (3/3)

- **Without L2$:**

  $CPI_{stall} = 2 + 1 \times 0.02 \times 100 + 0.36 \times 0.04 \times 100$

  $CPI_{stall} = 5.44$

- **With L2$:** Instr Fetch Load/Store

  $CPI_{stall} = 2 +$ 1x.02×25 + .36×.04×25 L1

  \+ 1x.005×100 + .36×.005×100 L2

  = 3.54

Agenda

- Multilevel Caches
- Administrivia
- Improving Cache Performance
- Anatomy of a Cache Question
- Example Cache Questions

Administrivia

- HW4 due next Tuesday
  - Due after midterm, but on midterm material
- Project 1 due Friday
  - But no slip days can be used
  - Late means 0 credit
- Mid-Semester Survey
  - Short survey to complete as part of Lab 6

Agenda

- Multilevel Caches
- Administrivia
- Improving Cache Performance
- Anatomy of a Cache Question
- Example Cache Questions

Improving Cache Performance (1/2)

1) Reduce the Hit Time of the cache
   - Smaller cache (less to search/check)
   - Smaller blocks (faster to return selected data)

2) Reduce the Miss Rate
   - Bigger cache (capacity)
   - Larger blocks (compulsory & spatial locality)
   - Increased associativity (conflict)

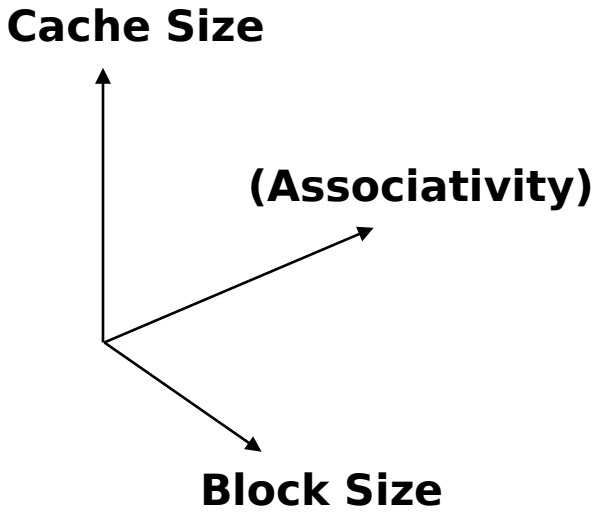
Improving Cache Performance (2/2)

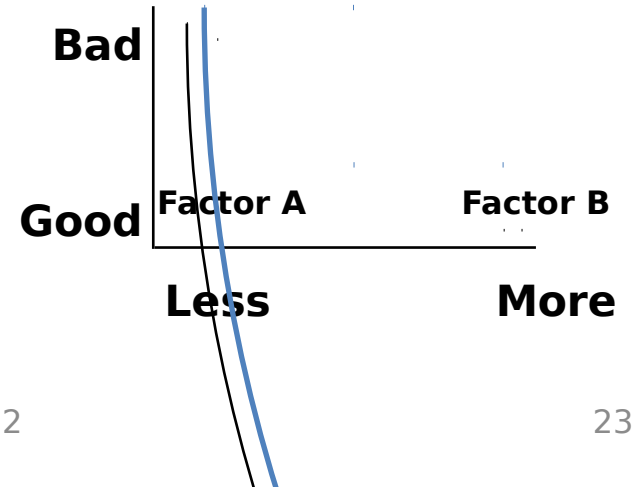
3) Reduce the Miss Penalty
   - Smaller blocks (less to move)
   - Use multiple cache levels
   - Use a *write buffer*
     - Can also check on read miss (may get lucky)

The Cache Design Space

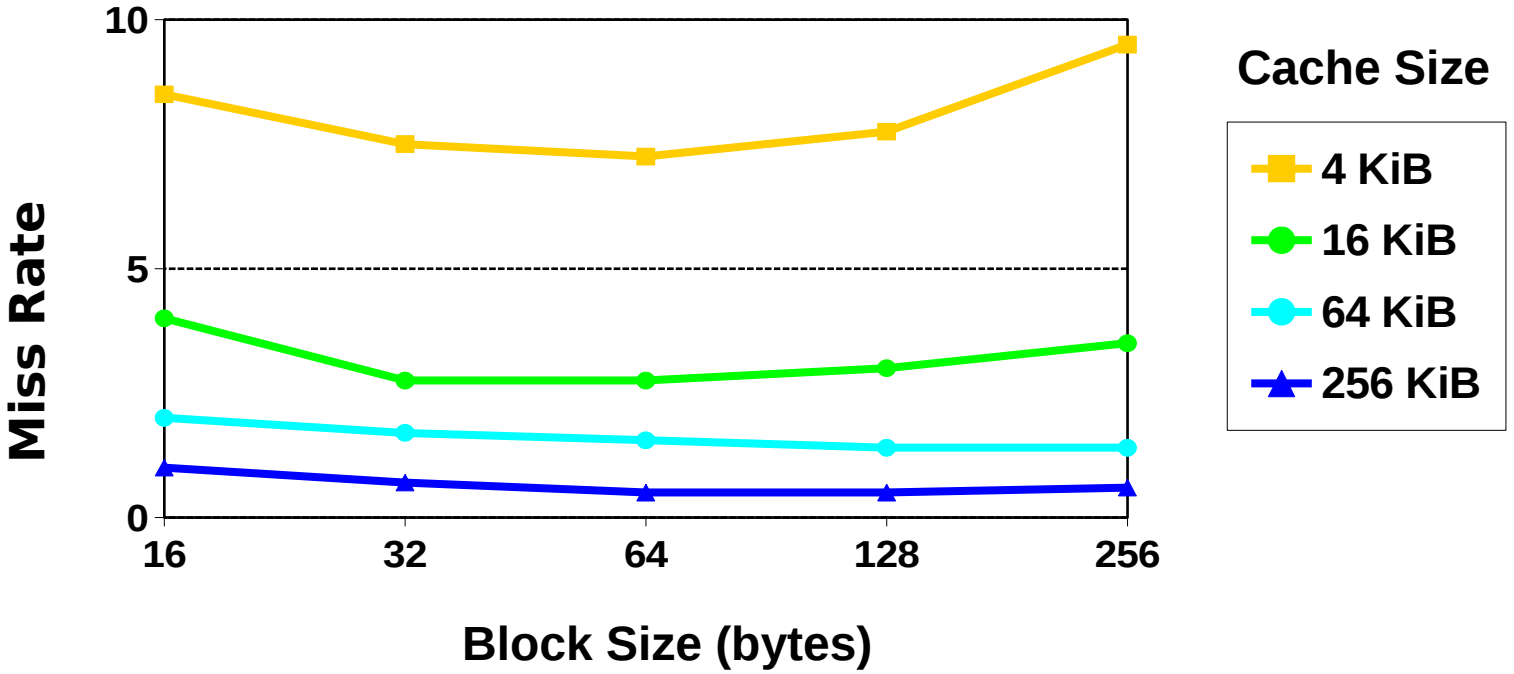
Several interacting dimensions

- Cache parameters:
  - Cache size, Block size, Associativity
- Policy choices:
  - Write-through vs. write-back
  - Write allocation vs. no-write allocation
  - Replacement policy
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload and use (I$, D$)
  - Depends on technology / cost
- Simplicity often wins



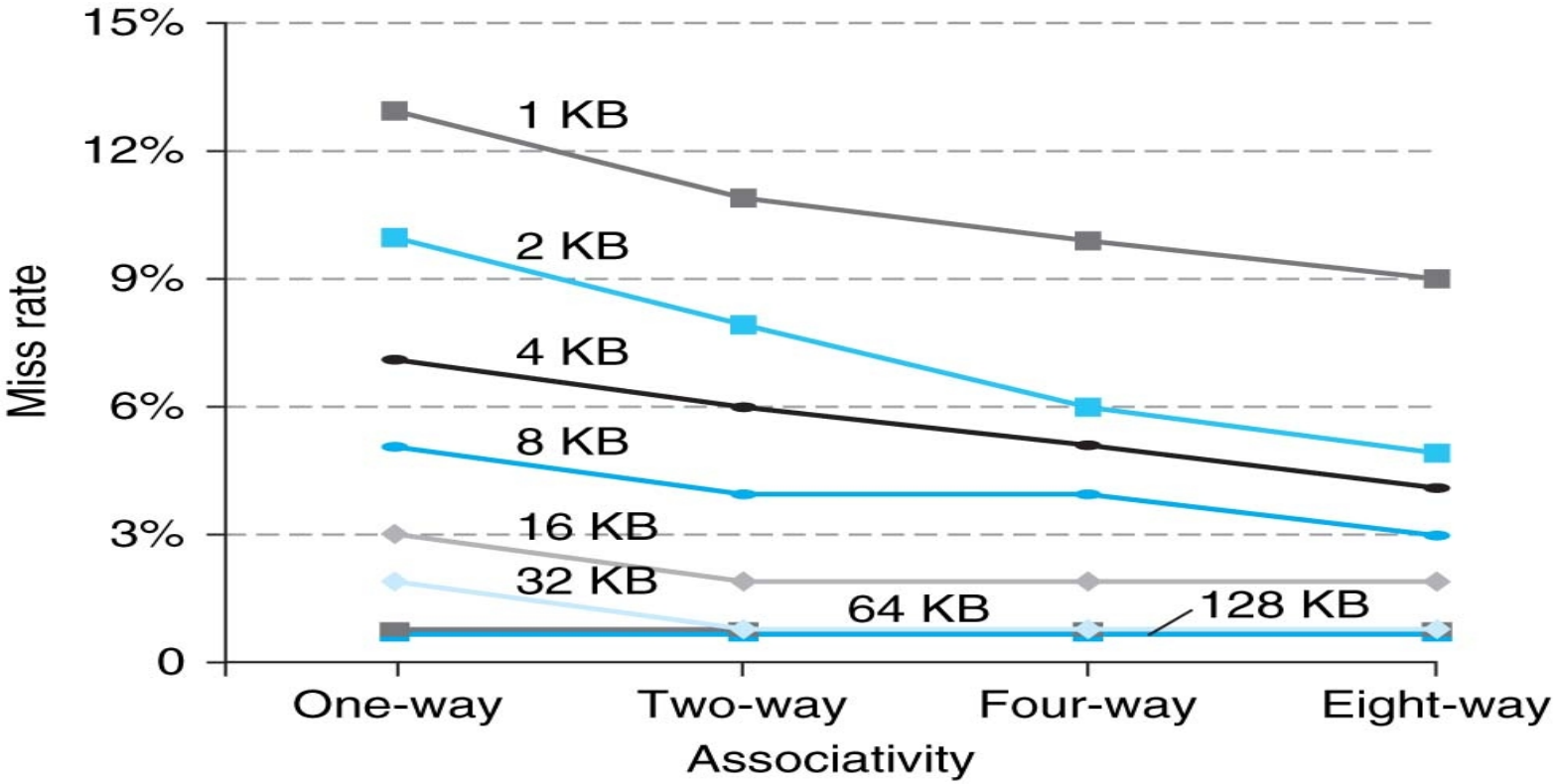


Effect of Block and Cache Sizes on Miss Rate



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing conflict misses)

Benefits of Set-Associative Caches



- Consider cost of a miss vs. cost of implementation
- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Agenda

- Multilevel Caches
- Administrivia
- Improving Cache Performance
- Anatomy of a Cache Question
- Example Cache Questions

Anatomy of a Cache Question

- Cache questions come in a few flavors:
  - TIO Breakdown
  - For fixed cache parameters, analyze the performance of the given code/sequence
  - For fixed cache parameters, find best/worst case scenarios
  - For given code/sequence, how does changing your cache parameters affect performance?
  - AMAT/CPI

The Cache

- What are the important cache parameters?
  - Must figure these out from problem description
  - Address size, cache size, block size, associativity, replacement policy
  - Solve for TIO breakdown, # of sets, set size
- Are there multiple levels?
  - Mostly applies to AMAT/CPI questions
- What starts in the cache?
  - Not always specified (best/worst case)

Code: Arrays

- Elements stored sequentially in memory
  - Ideal for spatial locality
  - Different arrays not necessarily next to each other
- Remember to account for data size!
  - char is 1 byte, int is (often) 4 bytes
- Pay attention to access pattern
  - Touch *all* elements (e.g. shift, sum)
  - Touch *some* elements (e.g. histogram, stride)
  - How many times do we touch each element?

Code: Linked Lists/Structs

- Nodes stored separately in memory
  - Addresses of nodes may be very different
  - Type and ordering of linking is important
- Remember to account for size/ordering of struct elements
- Pay attention to access pattern
  - Generally must start from “head”
  - How many struct elements are touched?

Access Patterns

- How many hits within a single block once it is loaded into cache?
- Will block still be in cache when you revisit its elements?
- Are there special/edge cases to consider?
  - Usually edge of block boundary or edge of cache size boundary

Best/Worst Case

- Set Associative (including Fully Associative)
  - Best: All accessed blocks fit into cache/set
  - Worst: Block is replaced before you access again
- Direct-Mapped
  - Best: Accessed blocks go into different slots
  - Worst: Constant conflict misses

Technology Break

Agenda

- Multilevel Caches
- Administrivia
- Improving Cache Performance
- Anatomy of a Cache Question
- Example Cache Questions

Example 1 (Sp07 Final)

a) 1 GiB address space, 100 cycles to go to memory. Fill in the following table:

	L1	L2
Cache Size	32 KiB	512 KiB
Block Size	8 B	32 B
Associativity	4-way	Direct-mapped
Hit Time	1 cycle	33 cycles
Miss Rate	10%	2%
Write Policy	Write-through	Write-through
Replacement Policy	LRU	n/a
Tag	17	11
Index	10	14
Offset	3	5
AMAT	AMAT L1 = 1 + 0.1 * 35 = 4.5	AMAT L2 = 33 + 0.02 * 100 = 35

Example 1 (Sp07 Final)

Only use L1$: **C** = 32 KiB, **K** = 8 B, **N** = 4, LRU, write-through

char A[] is block aligned and SIZE = 32 MiB

```
char *A = (char *) malloc (SIZE*sizeof(char));
/* number of STRETCHes */
for (i = 0; i < (SIZE/STRETCH); i++) {
   /* go up to STRETCH */
   for(j=0;j<STRETCH;j++)    sum  += A[i*STRETCH+j];
   /* down from STRETCH */
   for(j=STRETCH-1;j>=0;j--) prod += A[i*STRETCH+j];
}
```

- 2nd inner for loop hits same indices as 1st inner for loop, but in reverse order
- Always traverse full SIZE, regardless of STRETCH

Example 1 (Sp07 Final)

Only use L1$: **C** = 32 KiB, **K** = 8 B, **N** = 4, LRU, write-through

char A[] is block aligned and SIZE = 32 MiB

```
char *A = (char *) malloc (SIZE*sizeof(char));
for (i = 0; i < (SIZE/STRETCH); i++) {
   for(j=0;j<STRETCH;j++)    sum  += A[i*STRETCH+j];
   for(j=STRETCH-1;j>=0;j++) prod += A[i*STRETCH+j];
}
```

b) As we double our STRETCH from 1 to 2 to 4 (…etc), we notice the number of cache misses doesn't change! What is the largest value of STRETCH *before* cache misses changes? (Use IEC)

32 KiB, when STRETCH exactly equals **C**

Example 1 (Sp07 Final)

Only use L1$: **C** = 32 KiB, **K** = 8 B, **N** = 4, LRU, write-through

char A[] is block aligned and SIZE = 32 MiB

```
char *A = (char *) malloc (SIZE*sizeof(char));
for (i = 0; i < (SIZE/STRETCH); i++) {
    for(j=0;j<STRETCH;j++)    sum  += A[i*STRETCH+j];
    for(j=STRETCH-1;j>=0;j++) prod += A[i*STRETCH+j];
}
```

c) If we double our STRETCH from (b), what is the ratio of cache *hits* to *misses*?

Now STRETCH = 64 KiB. Moving sequentially by byte, so each block for entire 1st inner loop has 1 miss and 7 hits (7:1). Upper half of STRETCH lives in cache, so first half of 2nd inner loop is 8 hits/block (8:0). Second half is as before (7:1).

Example 1 (Sp07 Final)

Only use L1$: **C** = 32 KiB, **K** = 8 B, **N** = 4, LRU, write-through

char A[] is block aligned and SIZE = 32 MiB

```
char *A = (char *) malloc (SIZE*sizeof(char));
for (i = 0; i < (SIZE/STRETCH); i++) {
   for(j=0;j<STRETCH;j++)    sum  += A[i*STRETCH+j];
   for(j=STRETCH-1;j>=0;j++) prod += A[i*STRETCH+j];
}
```

c) If we double our STRETCH from (b), what is the ratio of cache *hits* to *misses*?

Considering the equal-sized chunks of half of each inner for loop, we have loop 1 1st (7:1), loop 1 2nd (7:1), loop 2 1st (8:0), and loop 2 2nd (7:1).

7+7+8+7:1+1+0+1 = 29:3

Questions?

Example 2 (Sp13 Final)

32-bit MIPS, 4 GiB memory, single L1$ of size ***C*** with block size ***K*** (***C*** ≥ ***K*** and a power of 2).

A, **B** are arrays in different places of memory of equal size ***n*** (power of 2 and a [natural #] multiple of ***C***), block aligned.

Array data size is 1 byte

```
// sizeof(uint8_t) = 1
SwapLeft(uint8_t *A, uint8_t *B, int n) {
    uint8_t tmp;
    for (int i = 0; i < n; i++) {
        tmp = A[i];
        A[i] = B[i];
        B[i] = tmp;
    }
}
```

Do ***n*** times:

← Read A[i]

← Read B[i], Write A[i]

← Write B[i]

Example 2 (Sp13 Final)

32-bit MIPS, 4 GiB memory, single L1$ of size ***C*** with block size ***K*** (***C*** ≥ ***K*** and a power of 2).

A, **B** are arrays in different places of memory of equal size **n** (power of 2 and a [natural #] multiple of ***C***), block aligned.

a) If the cache is direct-mapped and the *best* hit:miss ratio is "H:1", what is the block size in bytes?

Best case is A[i] and B[i] DON'T map to same slot.

Use every value of i ϵ [0,n) only once.

Rd A, Rd B, Wr A, Wr B --> Miss, Miss, Hit, Hit (1st time)

--> Hit, Hit, Hit, Hit (***K***-1 times in block)

Per block:

4*(***K***-1)+2:2 = 4***K***-2:2 = 2***K***-1:1 = H:1 --> ***K*** = (H+1)/2

Example 2 (Sp13 Final)

32-bit MIPS, 4 GiB memory, single L1$ of size ***C*** with block size ***K*** (***C*** ≥ ***K*** and a power of 2).

A, **B** are arrays in different places of memory of equal size **n** (power of 2 and a [natural #] multiple of ***C***), block aligned.

b) What is the *worst* hit:miss ratio?

Worst case is A[i] and B[i] map to same slot (conflict).

Rd A, Rd B, Wr A, Wr B --> Miss, Miss, Miss, Miss (all times)

because blocks keep replacing each other

0:1 (or 0:<anything>)

Example 2 (Sp13 Final)

c) Fill in code for SwapRight so that it does the same thing as SwapLeft but improves the (b) hit:miss ratio.

```
SwapRight(uint8_t *A, uint8_t *B, int n) {
    uint8_t tmpA, tmpB;
    for (int i = 0; i < n; i++) {
        tmpA = A[i];      ← Read A[i]
        tmpB = B[i];      ← Read B[i]
        B[i] = tmpA;      ← Write B[i]
        A[i] = tmpB;      ← Write A[i]
    }
}
```

Example 2 (Sp13 Final)

32-bit MIPS, 4 GiB memory, single L1$ of size ***C*** with block size ***K*** (***C*** ≥ ***K*** and a power of 2).

A, **B** are arrays in different places of memory of equal size **n** (power of 2 and a [natural #] multiple of ***C***), block aligned.

d) What is the *worst* hit:miss ratio for SwapRight?

Worst case is A[i] and B[i] map to same slot (conflict).

Rd A, Rd B, Wr B, Wr A --> Miss, Miss, Hit, Miss (1st time)

--> Hit, Miss, Hit, Miss (***K***-1 times)

Per block:

(***K***-1)*2+1:(***K***-1)*2+3 = 2***K***-1:2***K***+1

Example 2 (Sp13 Final)

e) Change the cache to be **2-way set-associative**. Cache size ***C***, block size ***K***. What is the *worst* hit:miss ratio for SwapLeft with the following replacement policies?

- LRU and an empty cache

  Even if A[i] and B[i] map to same set, they can both co-exist.

  Rd A, Rd B, Wr A, Wr B --> Miss, Miss, Hit, Hit (1st time)

  --> Hit, Hit, Hit, Hit (***K***-1 times in block)

  So 2***K***-1:1 (from part (a))

- MRU and a full cache

  Because cache is full, acts just like direct-mapped.

  So 0:<anything> (same as part (b))

Summary

- Multilevel caches reduce *miss penalty*
  - Local vs. global miss rate
  - Optimize first level to be fast (low HT)
  - Optimize lower levels to not miss (minimize MP)
- Cache performance depends heavily on cache design (there are many choices)
  - Effects of parameters and policies
  - Cost vs. effectiveness
- Cache problems are hard!