CPEN 411: Computer Architecture

Slide Set #11: Caches

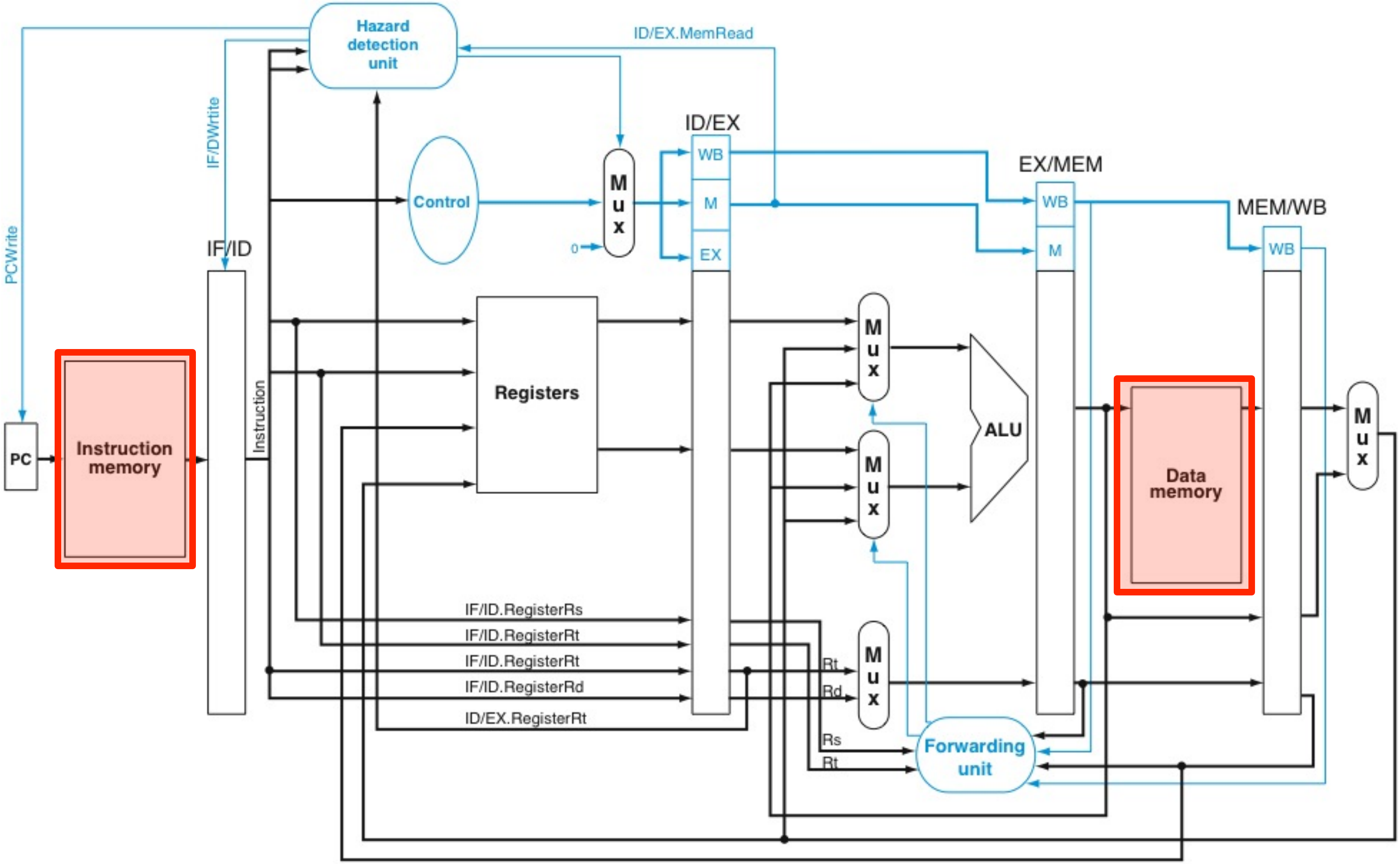
Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

Learning Objectives

- After this lecture you should be able to:
  - Explain the motivation for using caches
  - Describe levels of the memory hierarchy
  - Define simple cache terminology
  - Explain operation of several types of cache
  - Calculate relationship between cache parameters
  - Evaluate which accesses will hit or miss out of a sequence of accesses
  - Evaluate performance impact of caches using a simple analytical model
  - Apply similar reasoning as used to develop caches to solve novel architecture problems.

How to Design “Memory” Blocks?



$$Execution\ Time = IC \times CPI \times CycleTime$$

CPI = 1 + “average stall cycles per instruction”

So, longer memory access time increases stalls and consequently execution time

Store instructions and data electronically so we can access them at high speed.

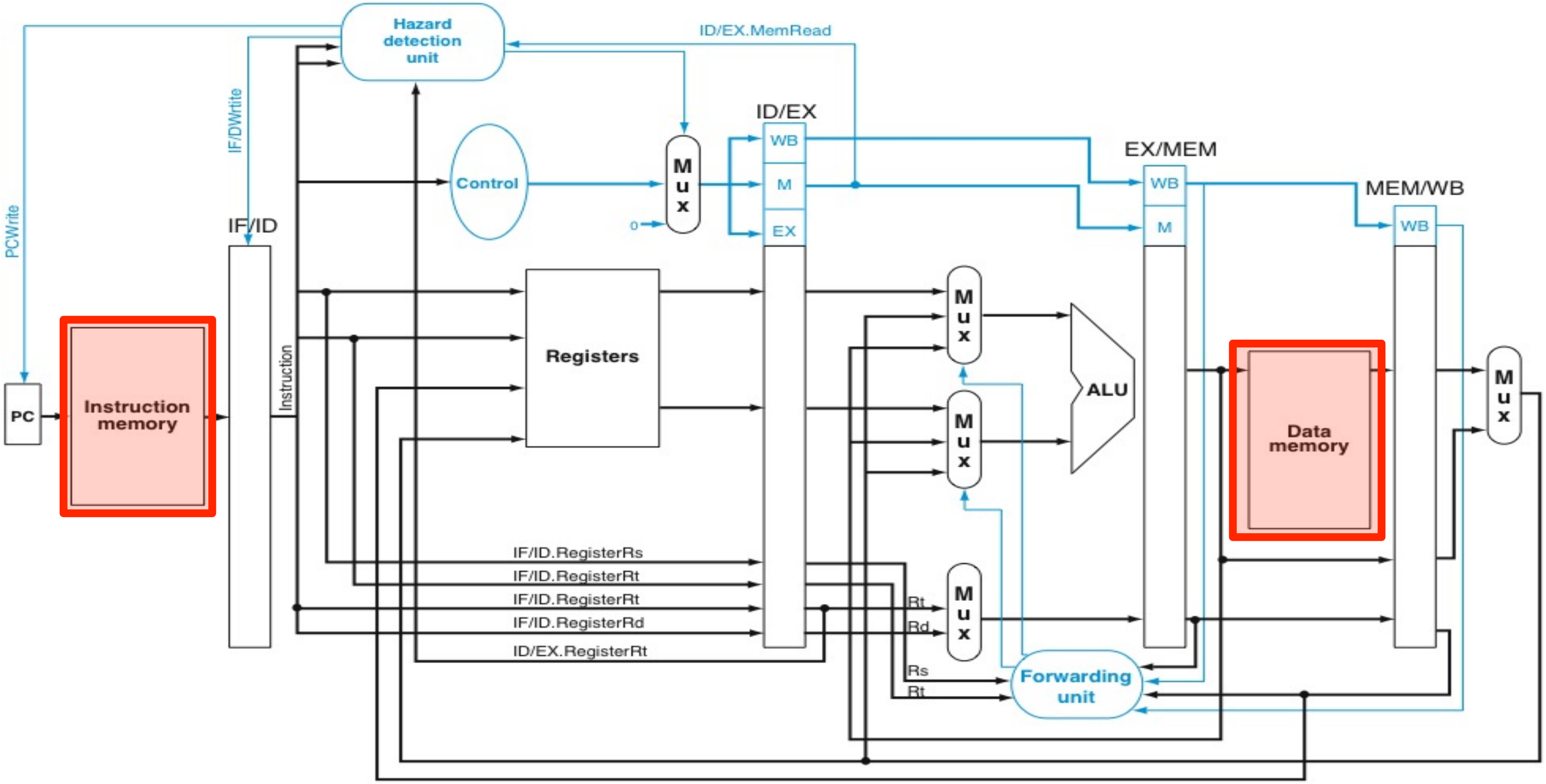
Read and write data at given address.

Available technology?

Magnetic tape?

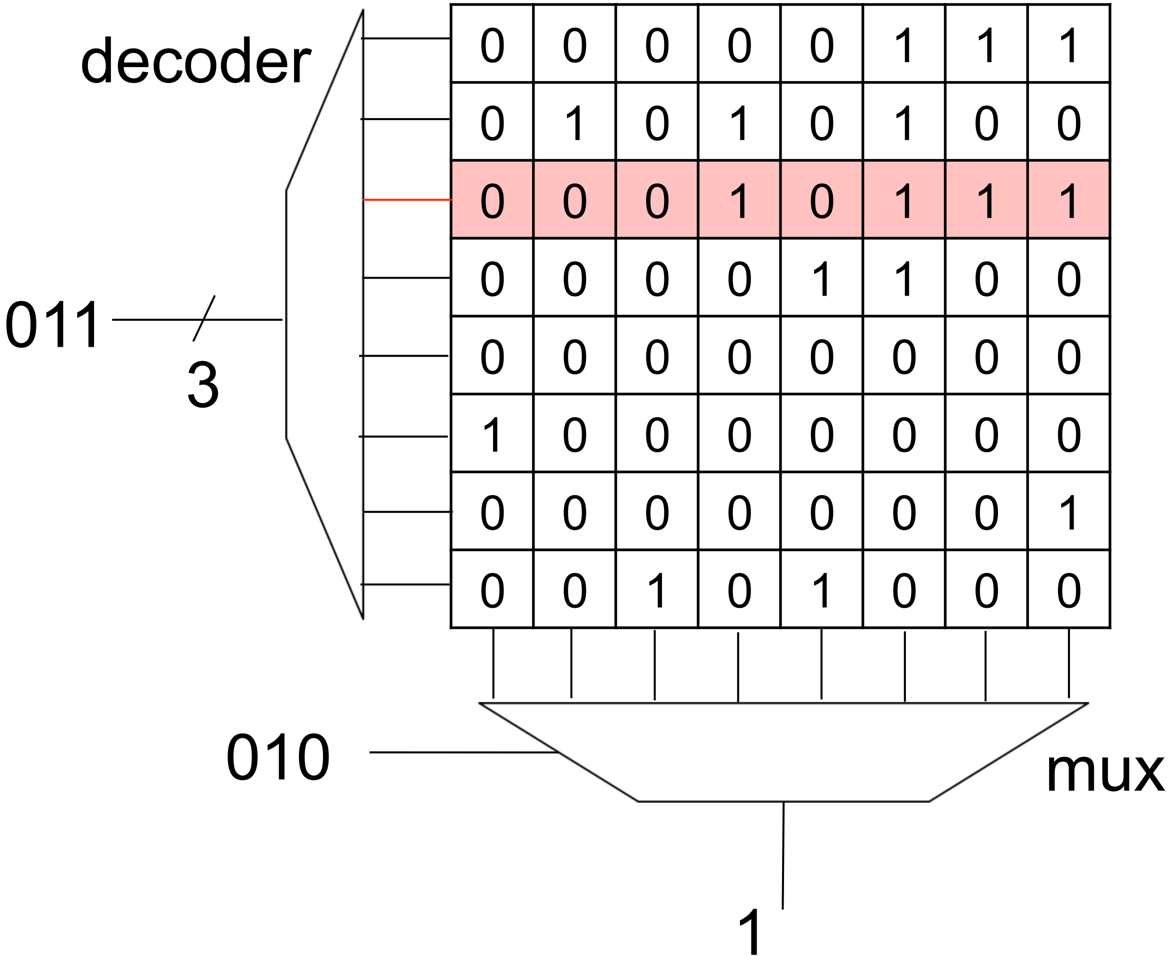
E.g., HP Ultrium. Sequential access 1TB/hour. 3TB cartridge costs $32.95 @ Office Depot (93 GB per $)

Tape: lowest cost per bit… low bandwidth, instructions & data not accessed sequentially



Random Access Memory

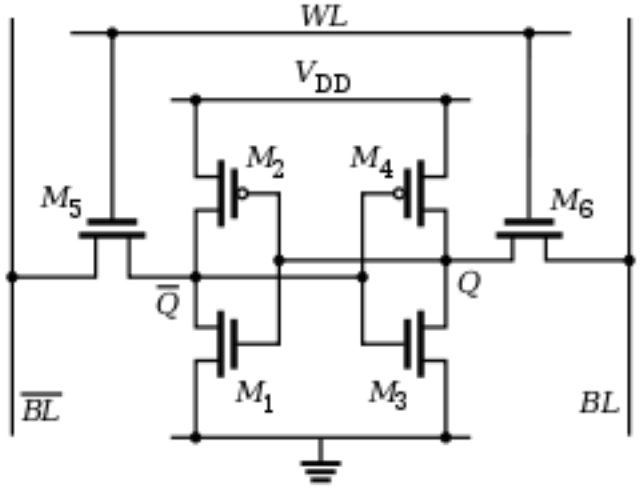
64x1-bit RAM: value at address “011010” = 1



0	0	0	0	0	1	1	1
0	1	0	1	0	1	0	0
0	0	0	1	0	1	1	1
0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0

Static RAM

Single ported SRAM uses six transistors to store a single bit.



SRAM Tradeoffs

- Small SRAM is very fast

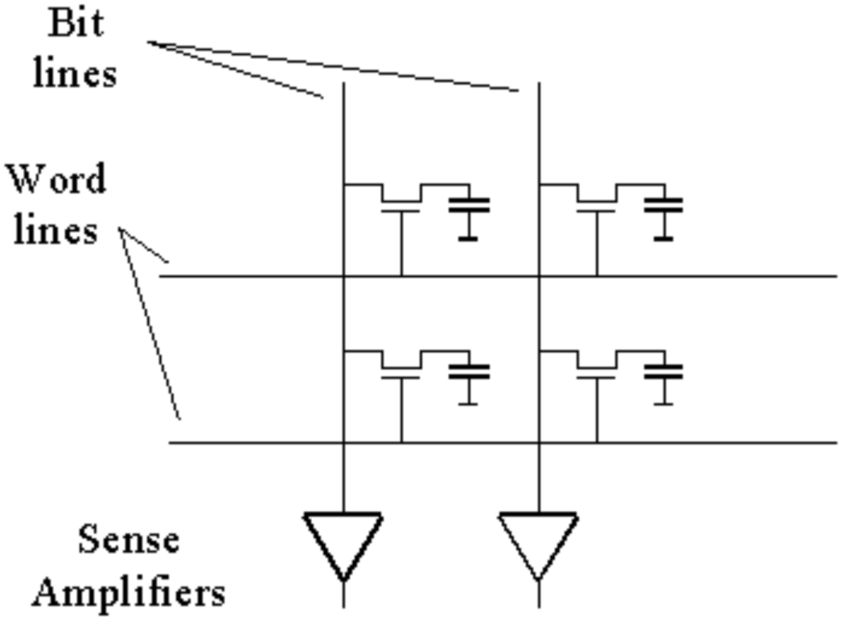
  ~200 picosecond access time for 4096 byte SRAM at 32 nm (single cycle access 5GHz)

- Large SRAM much slower

  ~5 nanosecond access time for 32 MB SRAM at 32 nm (25 cycle access at 5 GHz)

Dynamic RAM

DRAM uses single transistor and capacitor to store a bit.



DRAM Tradeoffs

- Cheaper than SRAM but more expensive than disk or tape: E.g., 16GB Corsair DDR3-2000 for $243.13 on shop.ca ($15 per GB --1395 times more per bit versus tape).

- Longer access time than SRAM

  DDR3-2000 minimum latency is 9 ns

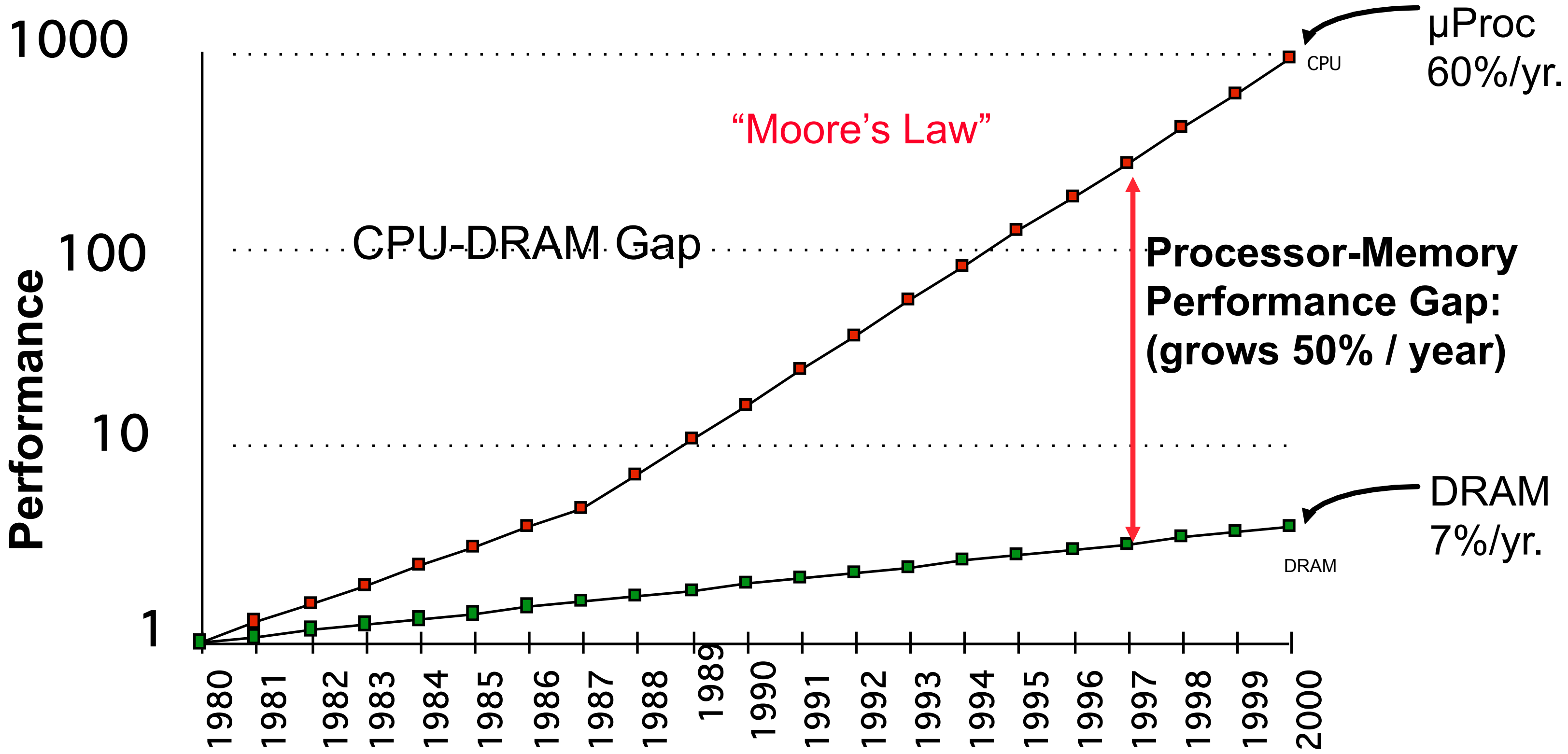
Fundamental Problem

Faster memory is smaller.

At this point, we know how to design a very fast computer that uses a very small memory.

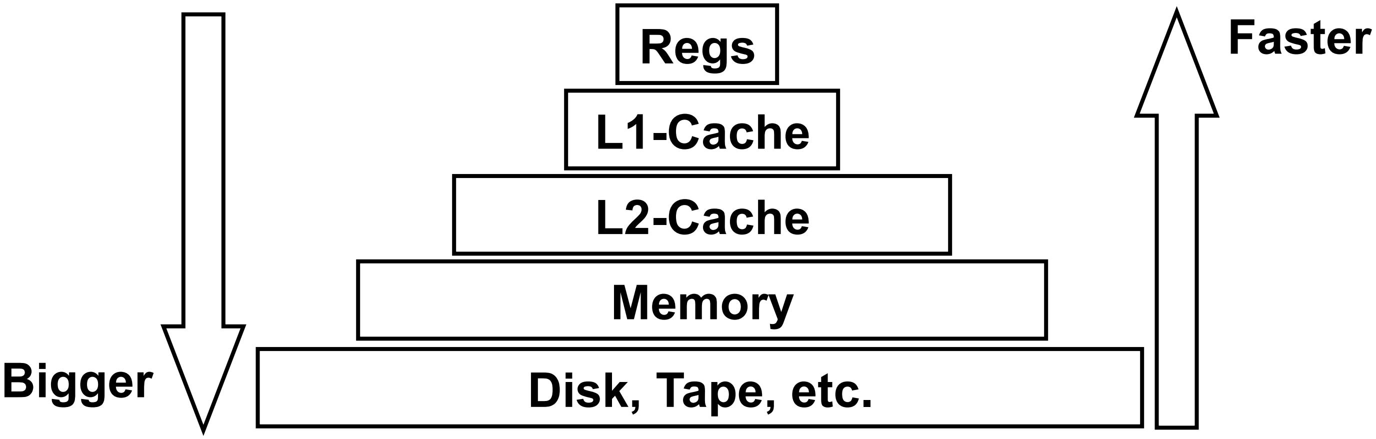
How can we design a fast computer that can use large memory?

The Memory Wall



- 1980: no cache in µproc; 1995 2-level cache on chip (1989 first Intel µproc with a cache on chip)

Solution: Memory Hierarchy



What do real programs do?

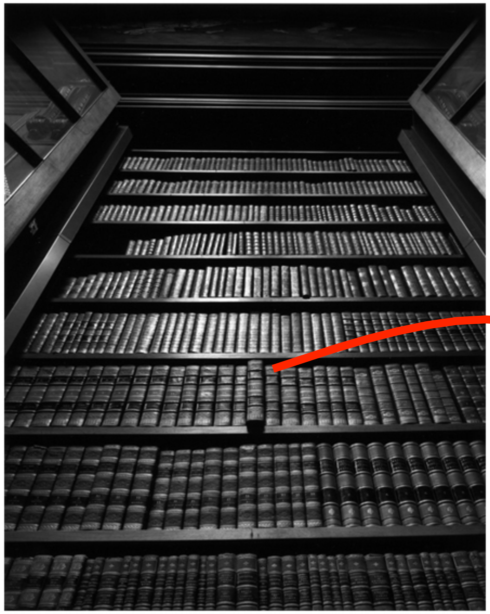
Consider “SAXPY” loop:

```
void saxpy( float a, float *X, float *Y, unsigned N )
{
   for( i=0; i < N; i++ )
        Y[i] = a*X[i] + Y[i];
}
```

Two observations:

1. Read then write Y[i] (temporal locality)
2. Read X[0], then X[1], then X[2] (spatial locality)

A Silly Locality Analogy



Fast RAM: Is book I want in backpack?

Slow RAM: Is book in UBC library?

Disk: inter-library loan (or chapters.ca)

Temporal Locality (analogy):
If use book once, likely to use it again soon.

Spatial Locality (analogy):
If last time went to library borrowed book on “mystery novels” next time likely get another book from same section.

Temporal Locality

Temporal Locality: After accessing single location once the program accesses it again.

Example: `Y[i] = a*X[i] + Y[i];`

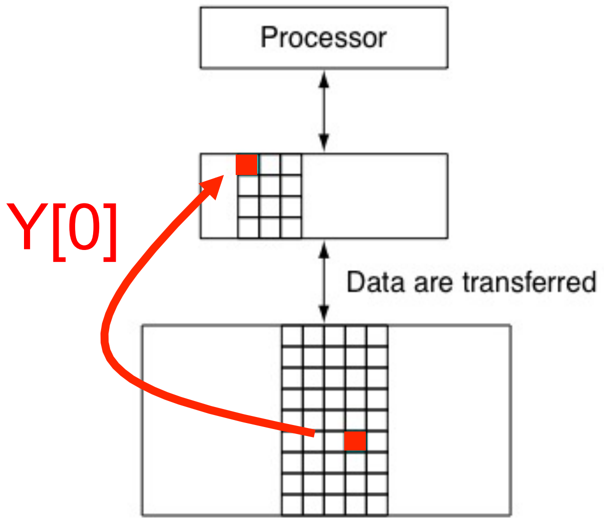
```
1:     L.S     F1,0(R1)      ; read X[i]
2:     MUL.S   F2,F1,F0      ; a*X[i]
3:     L.S     F3,0(R2)      ; read Y[i]
4:     ADD.S   F4,F2,F3      ; a*X[i]+Y[i]
5:     S.S     F4,0(R2)      ; write Y[i]
```

line 3: read `Y[i]`

line 5: write `Y[i]` (note: same location)

Exploiting Temporal Locality

Idea: When accessing location (address) make copy of data in small fast memory. If there is temporal locality program will try to access the same data again soon.



Y[0]

Processor

Data are transferred

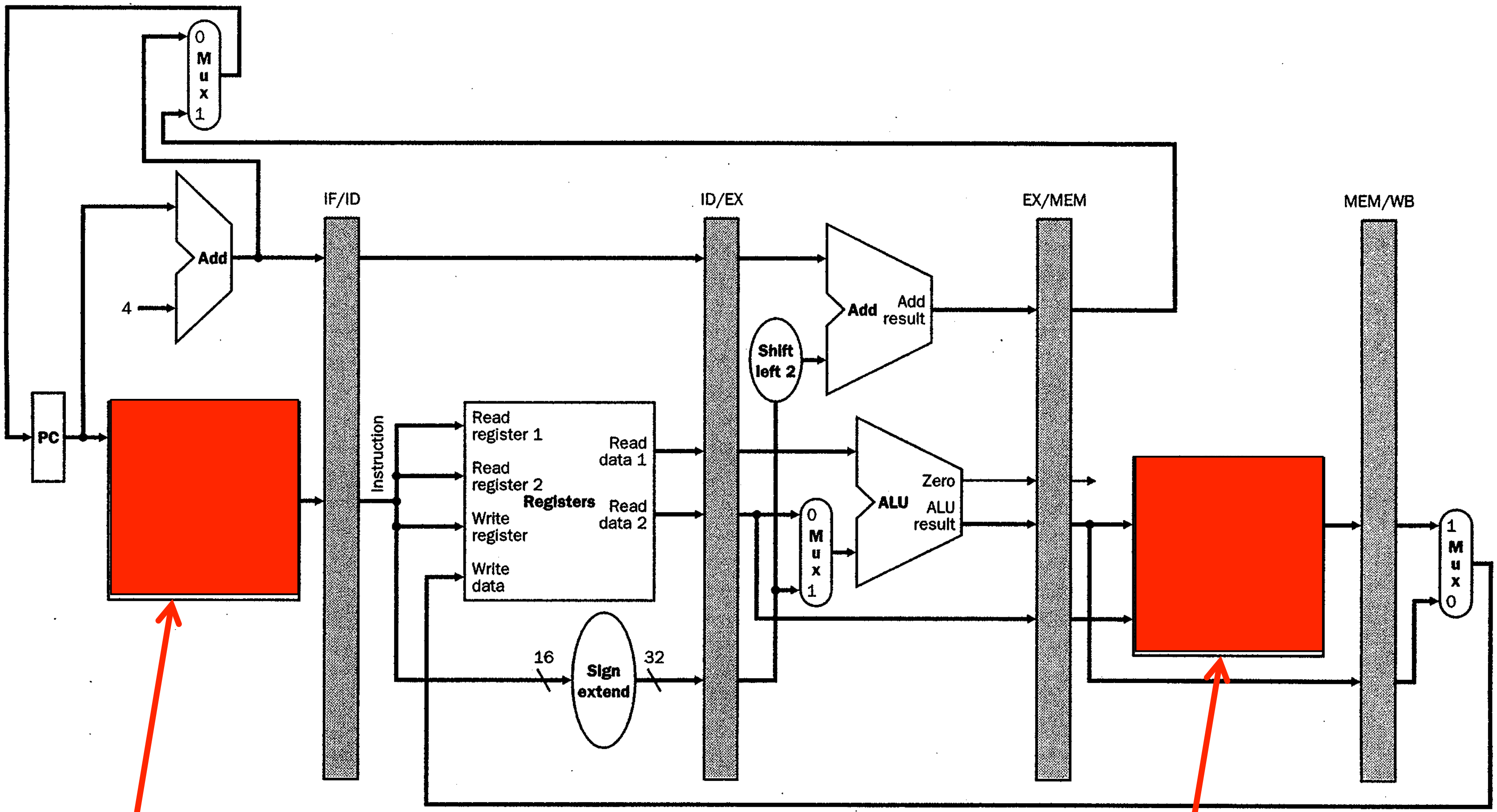
Small fast memory

Large slow memory

How is data transferred?

- Option 1: Software moves data
  - Small fast RAM called a “scratchpad” memory
  - Used by IBM Cell processor and recent GPUs
  - Typically, the programmer must decide when to move data between slow big memory and fast small memory. Add extra code to move data.
  - Benefits: Programmer can optimize what to put in fast memory and when.
  - Drawbacks: More work for programmer
- Option 2: Hardware managed cache
  - Most CPUs take this approach. Our focus.

First level caches in typical pipeline



Instruction Cache

First level caches (separate to avoid structural hazard)

Data Cache

Hardware managed cache design

- We will have hardware automatically copy data to “cache” when program accesses the location holding that data.
- Problem: Need mechanism for hardware to find data next time program accesses it.
- Solution: Save address (“tag”) in cache along with copy of the data at that address.

Example

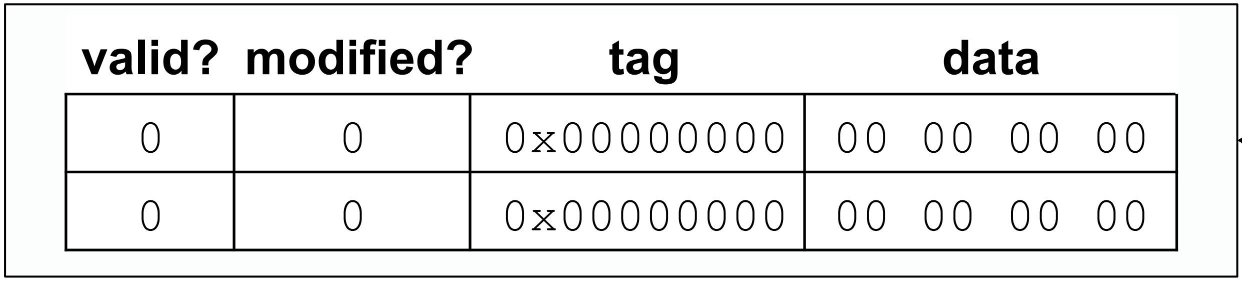
Show cache contents when executing following code on “fully associative” data cache with two 4 byte blocks.

```
L.S      F3,0(R2)   ; read Y[i]
ADD.S    F4,F2,F3   ; a*X[i]+Y[i]
S.S      F4,0(R2)   ; write Y[i]
```

Initially: Regs[R2]=0x8, Regs[F2]=78.2, memory state:

Address	**Memory contents**	**Description**
0x0000000c	C4 7A 00 00	Y[3] = -1000.0
0x00000008	40 2D F8 54	Y[2] = 2.718282
0x00000004	40 49 0F DB	Y[1] = 3.141593
0x00000000	42 28 00 00	Y[0] = 42.0

Example, continued…



valid?	modified?	tag	data
0	0	0x00000000	00 00 00 00
0	0	0x00000000	00 00 00 00

cache holding two 4-byte blocks

- Each row in cache contains four fields:
  - "data" is holds copy of memory data
  - "tag" indicates address associated with data
  - "modified" indicates if data in cache was written to since copying from memory
  - "valid" indicates the row contains valid information.

Example, continued

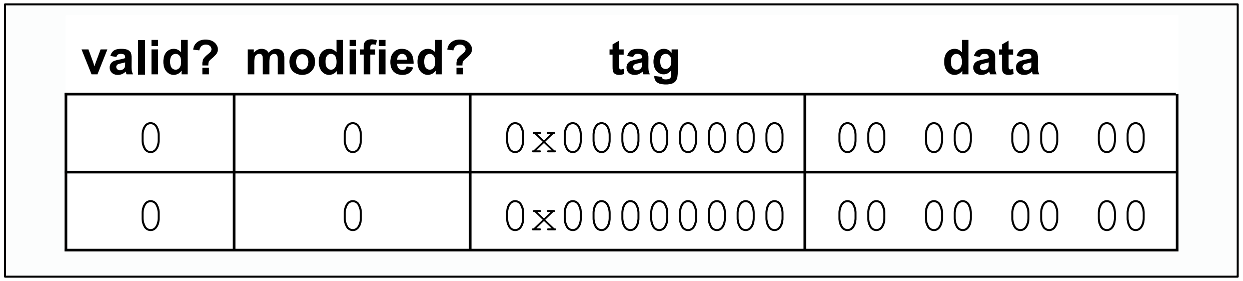
Executing:

```
        L.S   F3,0(R2)   ; read Y[i]
```

Step 1: compute effective address of memory access

0(R2) = 0x00000008

Step 2: Check if 0x00000008 already in cache:



valid?	modified?	tag	data
0	0	0x00000000	00 00 00 00
0	0	0x00000000	00 00 00 00

Answer = No. “Cache miss”. (If yes, “cache hit”)

Example, continued

Executing:

```
L.S   F3,0(R2)   ; read Y[i]
```

Step 3: Copy data into cache (put in first invalid row)

valid?	modified?	tag	data
1	0	0x00000008	40 2D F8 F4
0	0	0x00000000	00 00 00 00

Address	**Memory contents**	**Description**
0x0000000c	C4 7A 00 00	Y[3] = -1000.0
0x00000008	40 2D F8 54	Y[2] = 2.718282
0x00000004	40 49 0F DB	Y[1] = 3.141593
0x00000000	42 28 00 00	Y[0] = 42.0

Example, continued

Cache after executing

```
ADD.S F4,F2,F3; Regs[F4]=80.918 (0x42A1D629)
S.S   F4,0(R2); write Y[i]
```

valid?	modified?	tag	data
1	1	0x00000008	42 A1 D6 29
0	0	0x00000000	00 00 00 00

Address	Memory contents	Description
0x0000000c	C4 7A 00 00	Y[3] = -1000.0
0x00000008	40 2D F8 54	Y[2] = 2.718282
0x00000004	40 49 0F DB	Y[1] = 3.141593
0x00000000	42 28 00 00	Y[0] = 42.0

Question

- Consider the following series of memory references (byte addresses, in decimal): 2, 3, 2, 8, 10, and 2.

  Assume fully associative cache with 8 one-byte blocks that are initially empty, label each reference in the list as a hit or a miss, report the total number of hits and show the final contents of the cache.

  You can ignore “modified” and “data” fields.

Solution

2 m, 3 m, 2 h, 8 m, 10 m, 2 h

Valid	Tag
1	2
1	3
1	8
1	10
0	-
0	-
0	-
0	-

Problem with this design

Where do we put data if cache is full?

Return to this question after considering spatial locality.

Spatial Locality

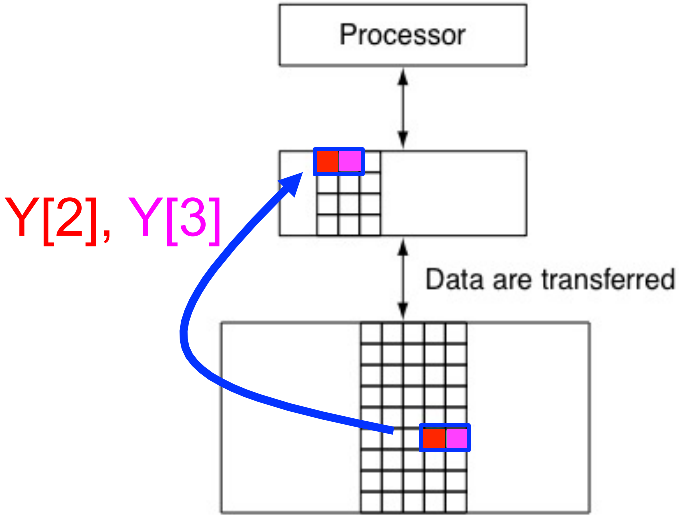
Definition: After accessing a location program accesses nearby location.

Example:

```
Y[2] = a*X[2] + Y[2];
Y[3] = a*X[3] + Y[3];
```

Program reads `Y[2]`, then `Y[3].`

Exploiting Spatial Locality



When program accesses Y[2] transfer both Y[2] and Y[3].

In general, transfer a “block” of data (or instructions) at a time. Block size typically 32 to 128 bytes.

In figure at left have a block containing Y[2] and Y[3] but more typical 64 byte block can hold sixteen 4-byte values.

Cache Blocks (Cache Lines)

Want to move nearby data at same time to exploit spatial locality. Divide up memory into blocks that are power of two multiples of word size. Block starts at address aligned to block size.

memory address   memory contents

```
0xfffffff8   00 00 00 00 00 00 00 00
0xfffffff0   00 00 00 00 00 00 00 00
    …                   …
0x00000010   42 3C E7 98 01 00 70 95
0x00000008   40 2D F8 54 C4 7A 00 00
0x00000000   42 28 00 00 40 49 0F DB
```

8 byte block. First byte at 0x00000008 contains "40". Last byte at 0x0000000f contains "00" ("big endian" byte order).

Copy entire block at once.

Modified Cache Design

- Increase data field size to hold block of size 8 bytes. Now cache can exploit both spatial and temporal locality.

valid?	modified?	tag	data
0	0	0x00000000	00 00 00 00 00 00 00 00
0	0	0x00000000	00 00 00 00 00 00 00 00

Block Size Impact on Tag

- Blocks start at address aligned to block size.

- This means lower $\log_2$(block size) bits of block address are always zero.

- Would be a waste of transistors if we explicitly store these zero values.

Question

For a fully associative cache with 16 byte blocks what is the tag in hex for a block with starting address 0x00008020?

Example, revisited

Regs[R2] = 8 implies tag=0x0000001 -- why?

In binary `0x00000008 = 0000 … 0000000001000`

Drop least significant 3 bits since using 8 byte blocks

Cache after executing

```
L.S   F3,0(R2) ; read Y[i]
```

valid?	modified?	tag	data
1	0	0x00000001	40 2D F8 54 C4 7A 00 00
0	0	0x00000000	00 00 00 00 00 00 00 00

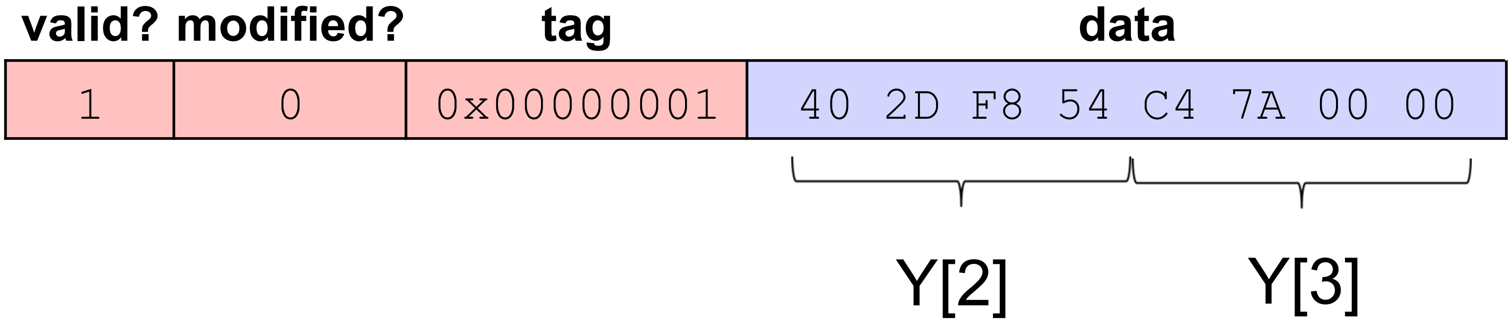
copy 8 byte block to cache

memory address	memory contents
0x00000010	42 3C E7 98 01 00 70 95
0x00000008	40 2D F8 54 C4 7A 00 00
0x00000000	42 28 00 00 40 49 0F DB

4 byte value written to Regs[F3] by L.S is 0x402DF854

Multiple words in block

- For the SAXPY loop 8 byte block contains two elements – e.g., Y[2] and Y[3]:



valid?	modified?	tag	data
1	0	0x00000001	40 2D F8 54 C4 7A 00 00

- How to find specific data (e.g., Y[2] or Y[3]) accessed by load/store instruction?

Solution: Block Offset

- Recall, block starts at address aligned to $\log_2$(block size) and data stored within block in same order as in memory.

- So, we can compute location of word in block by using lower $\log_2$(block size) bits of address of word.

Solution: Block Offset

- Example block offset calculation (8 byte block)

address (hex)		address (binary)
0x00000008	=	0000 … 000000001|000|
0x0000000C	=	0000 … 000000001|010|

valid?	modified?	tag	data
1	0	0x00000001	40 2D F8 54 C4 7A 00 00

Block Offset = 0

Block Offset = 4

Question

For a cache with 16 byte blocks what is the block offset in hex for a 4 byte float stored in memory at starting address 0x00008028?

Example, revisited

Regs[R2] = 0x8 => Tag=0x1, Block Offset=0

Cache after executing

```
    ADD.S F4,F2,F3          ; Regs[F4] = 0x42A1D629
    S.S   F4,0(R2)          ; write Y[i]
```

valid?	modified?	tag	data
1	1	0x00000001	42 A1 D6 29 C4 7A 00 00
0	0	0x00000000	00 00 00 00 00 00 00 00

What if cache is full?

- So far we assumed we “fill” blocks into cache starting at first available (“invalid”) row and when accessing cache do parallel search for tag match in all rows (“fully associative”)

- If cache full (no blocks are “invalid”), need to make space for new block by “evicting” a “valid” block from cache.

Which block gets evicted?

- Which is best block to evict? Recall we want to keep useful data in cache. So, we need to consider locality.
- Provably optimal: evict block that will be accessed again furthest in future.
- Problem: Hardware does not know future. Need prediction mechanism.

How to predict?

- Goal: Want to predict which block will be accessed furthest in future.
- Observation: For many programs, the longer a block has not been accessed, the less likely it is that we will access it soon.
- Can quantify this using “reuse distance”.

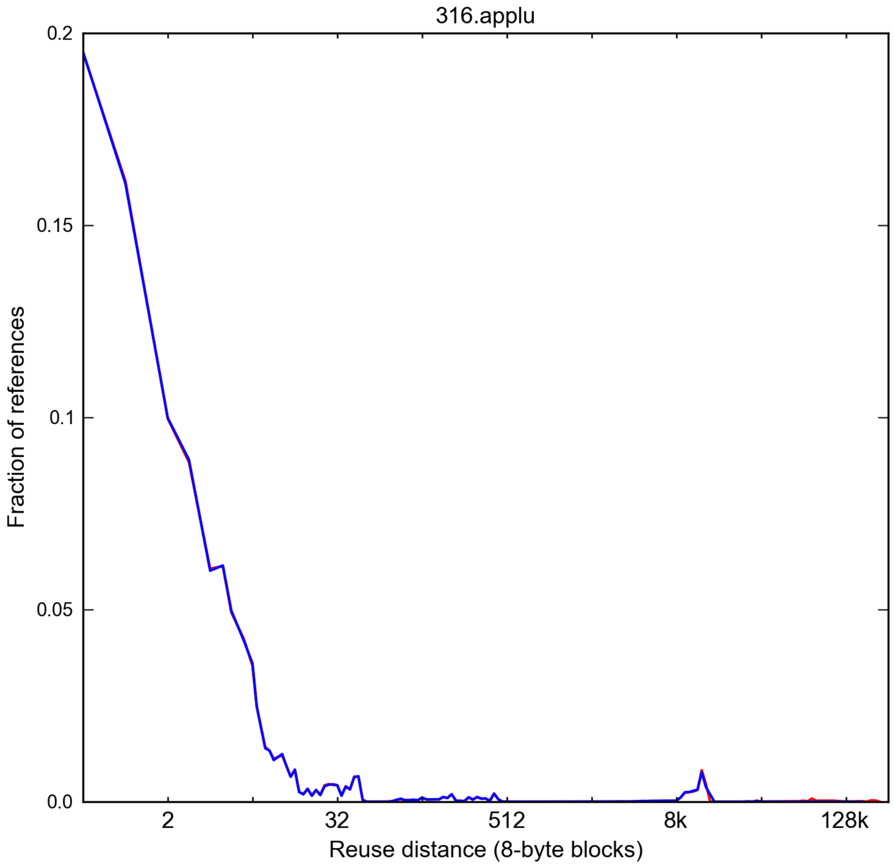
Reuse Distance (Definition)

- Consider sequence of memory accesses to blocks A, B, C, D, and E:

ABBCBCCDBBEDBBBA

- Between two accesses to block A made 14 accesses to four unique blocks (B, C, D, E)
- Reuse distance for A is four.

Reuse distance measurement



Typically access recently used data most frequently.

Replacement Policy

- The algorithm used to decide which block to evict is called a “replacement policy”.

- Observation on last slide suggests good replacement policy is to evict the “least recently used” (LRU) block.

Least Recently Used

- LRU Implementation?
  - Naïve: add timestamp field to row updated each time block accessed. To evict, compare all timestamps to find smallest.
  - In practice try to limit search to small number of blocks (e.g., “set associative” later in slides). With two blocks can use bit to indicate which one most recently used.
  - Patents on approximate LRU (pseudo-LRU).
- Random & FIFO get similar hit rates to LRU.
- Recently, replacement policy very active area of innovation due to multicore and “big data”.

Question

- Assume fully associative cache with 8 one-byte blocks. Consider the following series of memory references (byte addresses, in decimal): 2, 3, 2, 8, 10, and 2. After this sequence has been observed, what is the tag of the least recently used block?

Problem with Fully Associative

- Comparing all tags in parallel is expensive (time, area and energy).
- Can avoid multiple tag comparisons if block can only be stored in single row in cache. This is called “direct mapped”. Then, only one tag needs to be checked.
- Also makes replacement decision easy (only one choice of what to evict).

Direct Mapped Cache

- Given start address of block need way for hardware to select single row of cache to place block in.

- Start of block is aligned so least significant $\log_2$(block size) bits in address are zero. Use hash on remaining address bits to select row.

- Typically, use next $\log_2$(number of rows) least significant bits of block start address to select row.

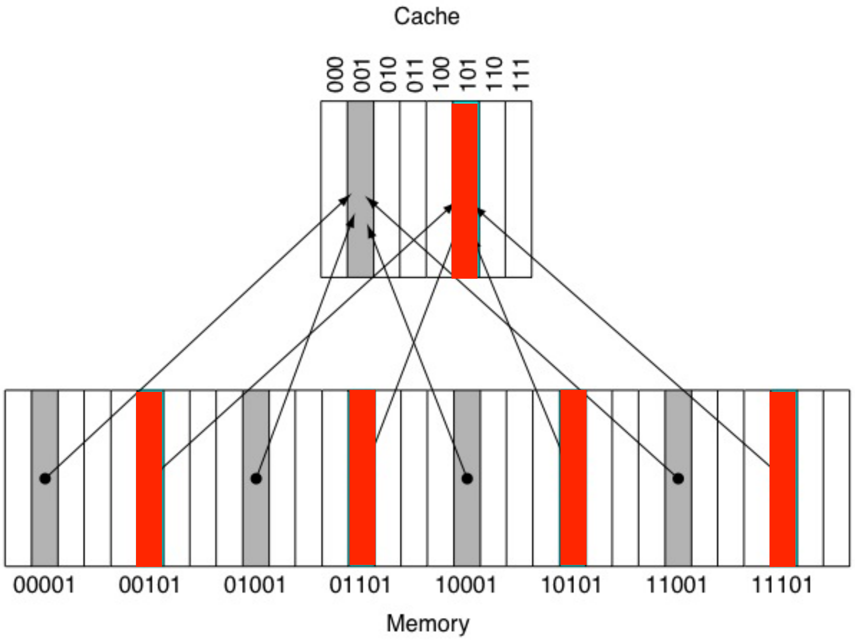
Direct Mapped Cache

“Block Address” is starting address of block with $\log_2$(block size) least significant bits truncated (removed).

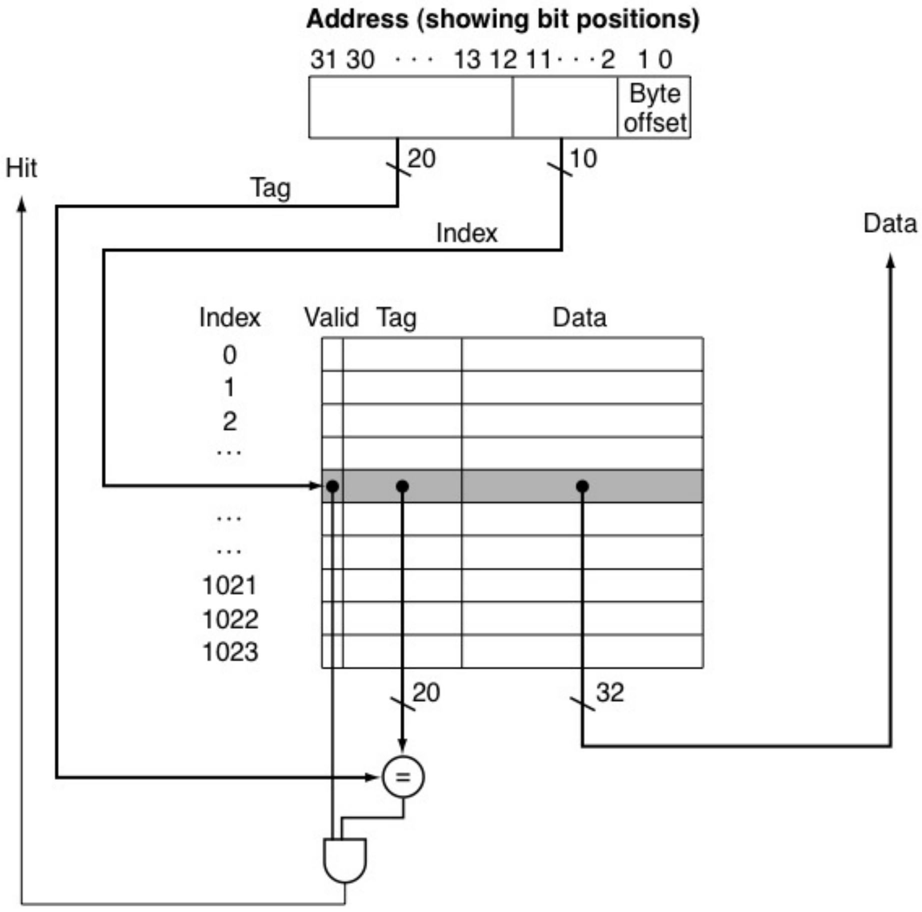
Address:

Block Address	Block Offset

Index = (Block Address) modulo (Number of Blocks)



Direct Mapped Cache



- 4-byte blocks, so block offset is 2-bits.
- 1024 rows so use $\log_2(1024)=10$-bits of address to select row. These bits are called "index".

Problem: Conflict Misses

- Some “rows” (blocks) in direct mapped cache used more than others.

- Data that index to these blocks may get evicted before reused even though there was enough space in cache to keep them.

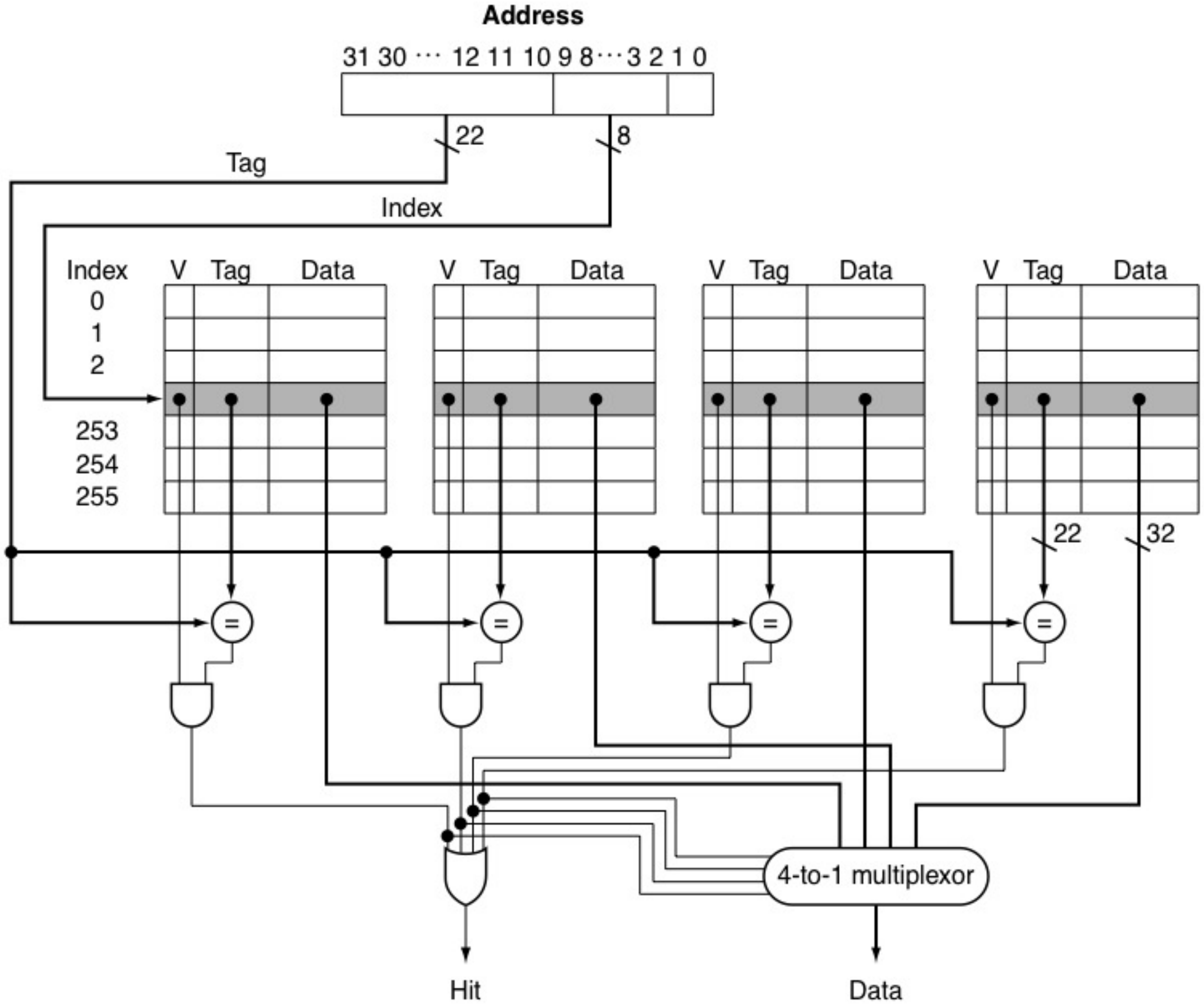
- Would like to have flexibility of fully associative without cost of parallel comparison of all tags in cache.

Set Associative Cache

- Index selects row containing **set of blocks**.

- Search tags of all blocks in this set in parallel.

Set Associative Cache



Question

- Consider the following series of memory references (byte addresses, in decimal): 2, 3, 2, 8, 10, and 2. Assuming a two-way set associative cache with 2-byte blocks and total size of 16 bytes that are initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache.

Solution

2 , 3 , 2 , 8 , 10 , 2

Cache set	Address
000	8
001	
010	2
011	3
100	
101	
110	
111	

Direct-mapped cache
1 byte blocks
8 bytes total

2 , 3 , 2 , 8 , 10 , 2

Cache Set	Address
00	{ [8,9] , - }
01	{ [2,3] , [10,11] }
10	{ - , - }
11	{ - , - }

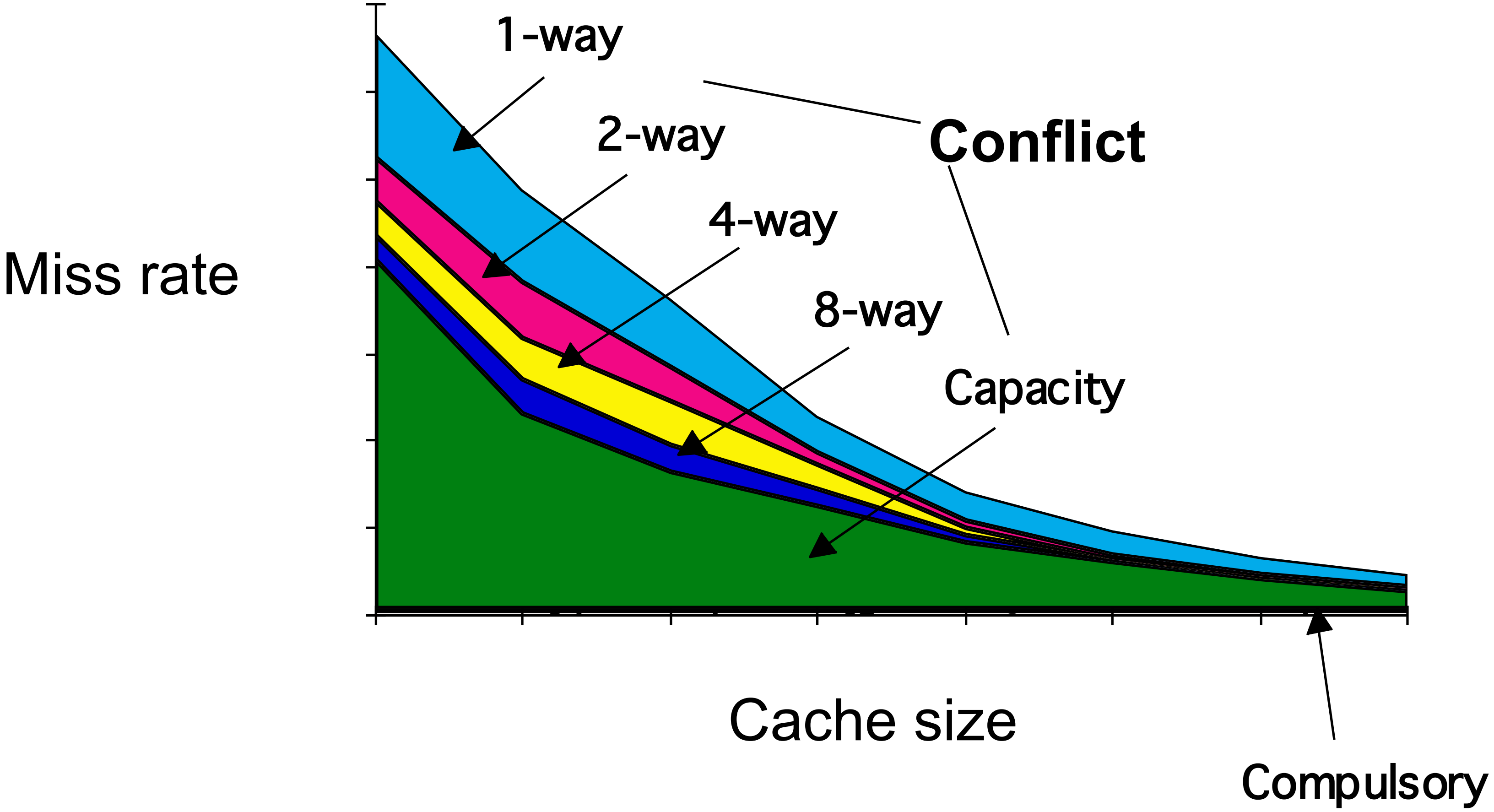
2-way set associative
2 byte blocks
16 bytes total

The 3 C's

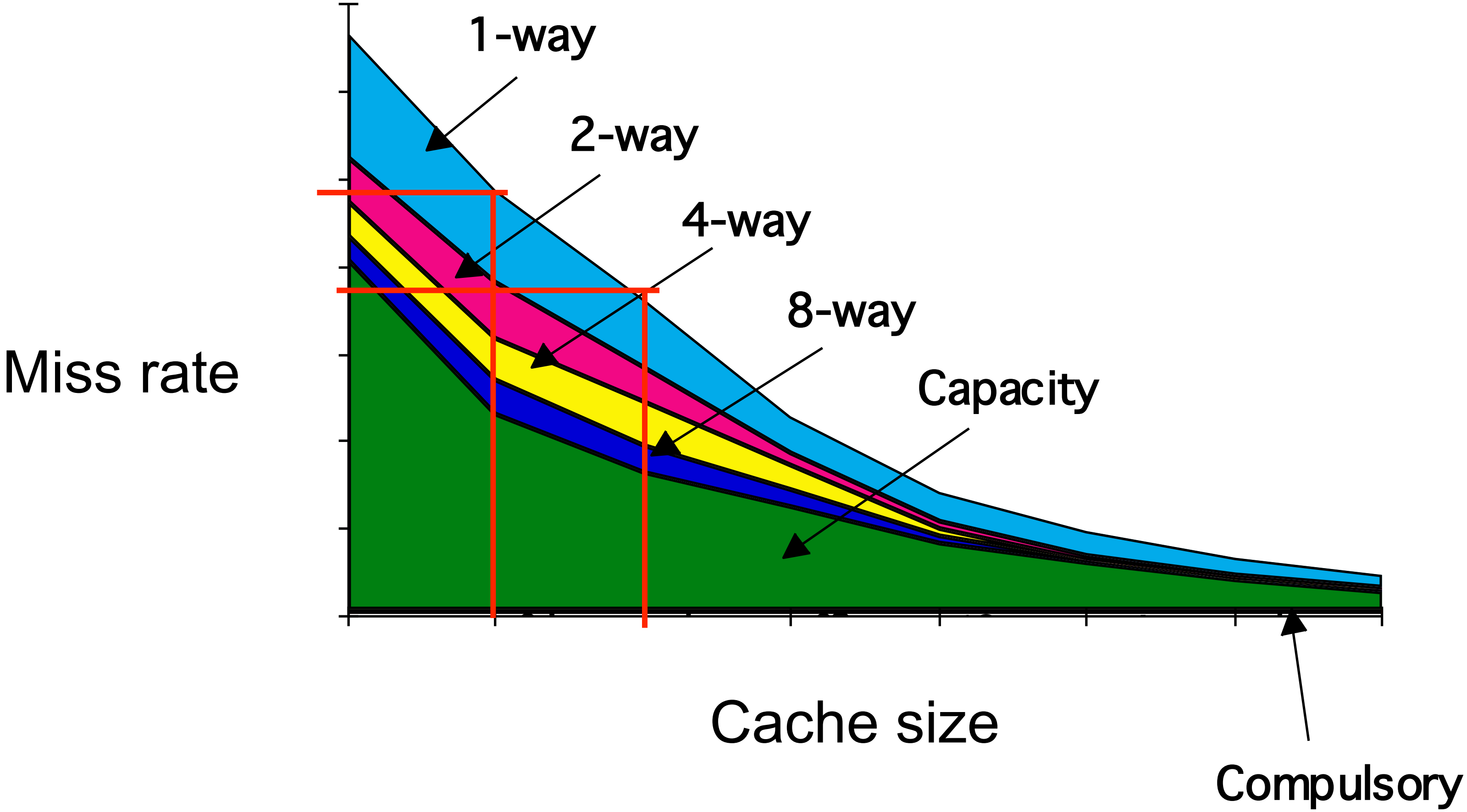
- Cold Start Misses
  - First time we access block of data, we have a miss => can't avoid this by making cache bigger or changing associativity.
- Capacity Misses
  - Difference in cache misses suffered by a fully associative cache (typically assume LRU replacement) and an infinite size cache
- Conflict Misses
  - Difference in cache misses suffered in cache being analyzed (associative or direct mapped cache) and a fully associative cache with same capacity and block size. (So, fully associative has no conflict misses.)

- NOTE: Typically, one does not actually try to decide whether a specific cache miss is a conflict or capacity miss (just measure total number of capacity and/or conflict misses). One reason: it is possible for miss rate fully assoc. > miss rate direct mapped.

3Cs Absolute Miss Rate (SPEC92)



Impact of Cache Size



Factor by which miss rate decreases is less than increase in cache size (here double cache size 2KB to 4KB, reduce miss rate ~ 20%);

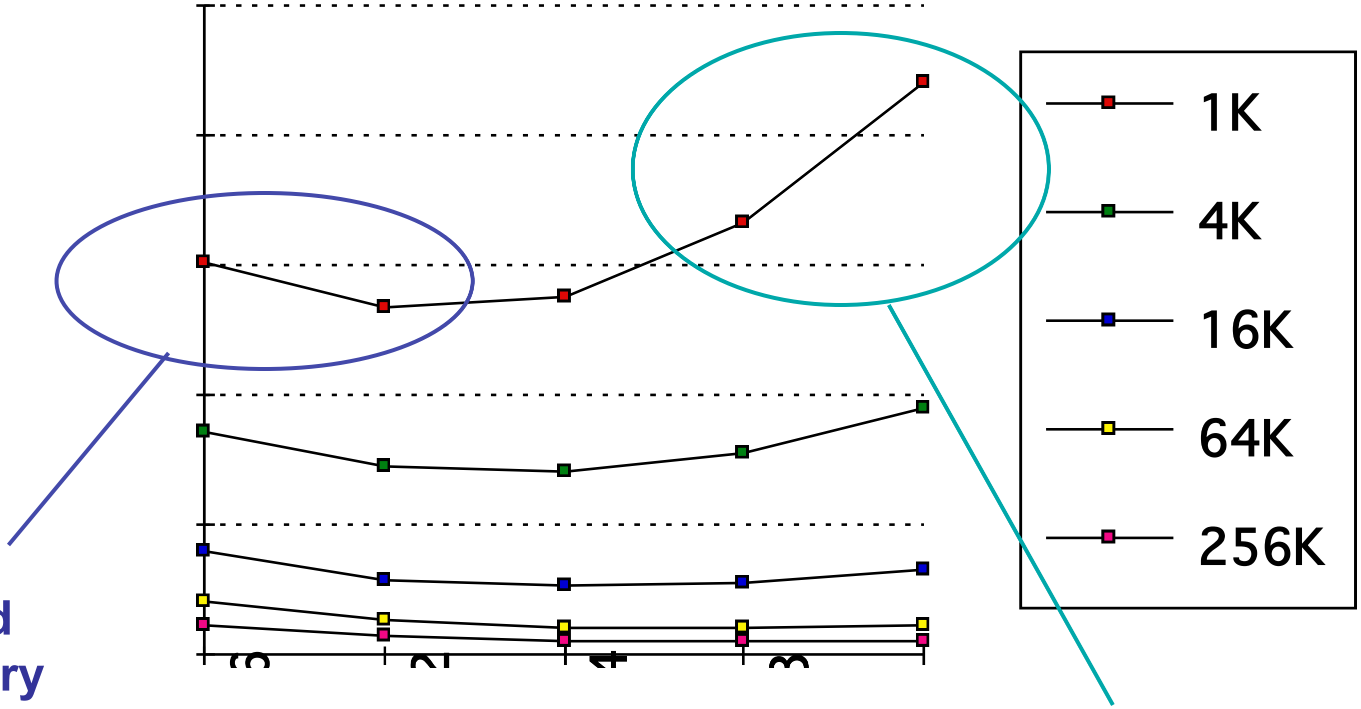
Increase in capacity

Data from H&P 4th edition Figure C.8
(SPEC 2000, 8-way, LRU, 64B blocks)

capacity	miss rate	change (ratio)
4	0.071	
8	0.044	0.620
16	0.041	0.932
32	0.037	0.902
64	0.029	0.784
128	0.019	0.655
256	0.012	0.632
512	0.006	0.500
	average	**0.718**
	1/sqrt(2)	**0.707**

Old “rule of thumb” (very approximate, as data above shows)
factor by which miss rate decreases $\propto$ 1/sqrt(increase in capacity)

Larger Block Size (fixed size & assoc)

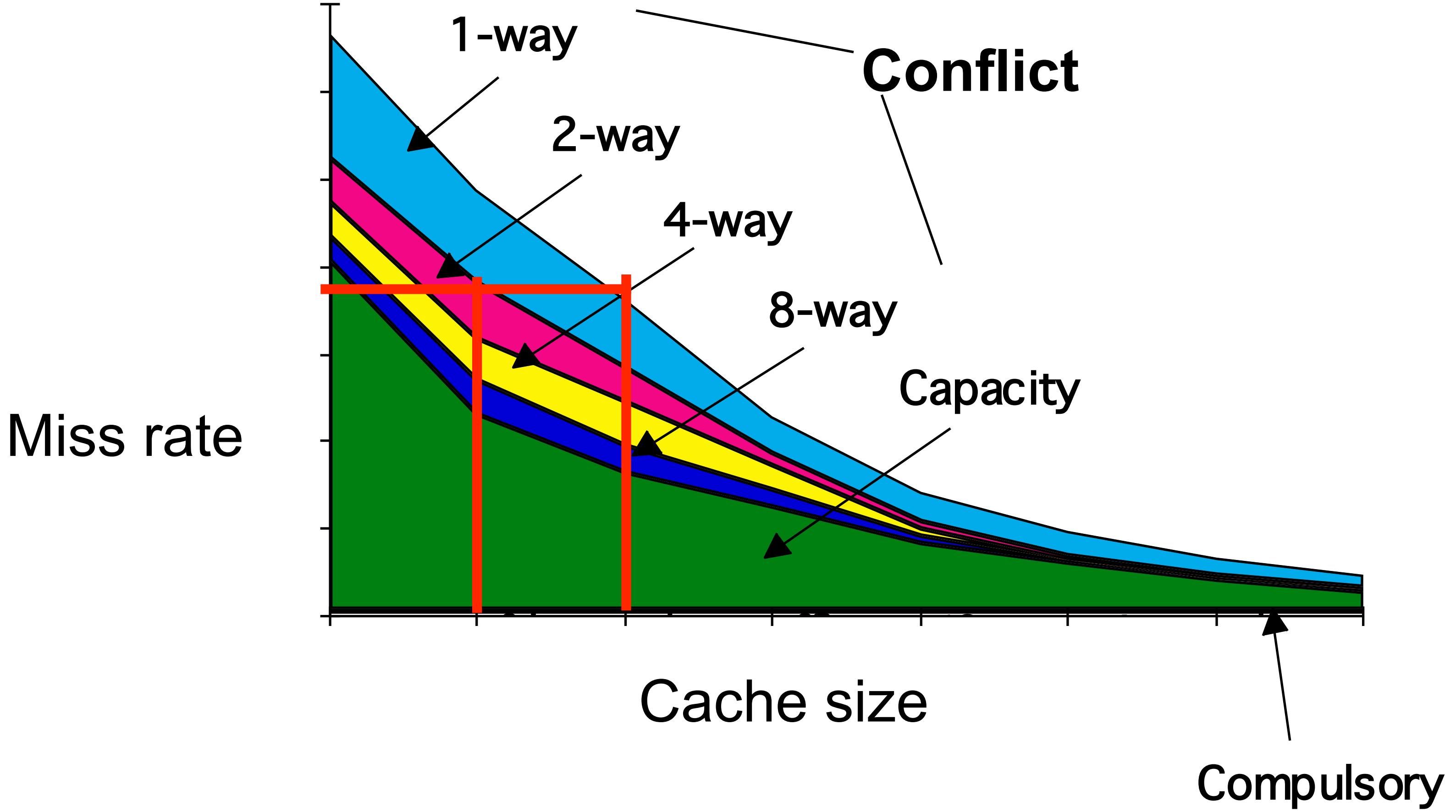


Reduced compulsory misses

Increased Conflict Misses

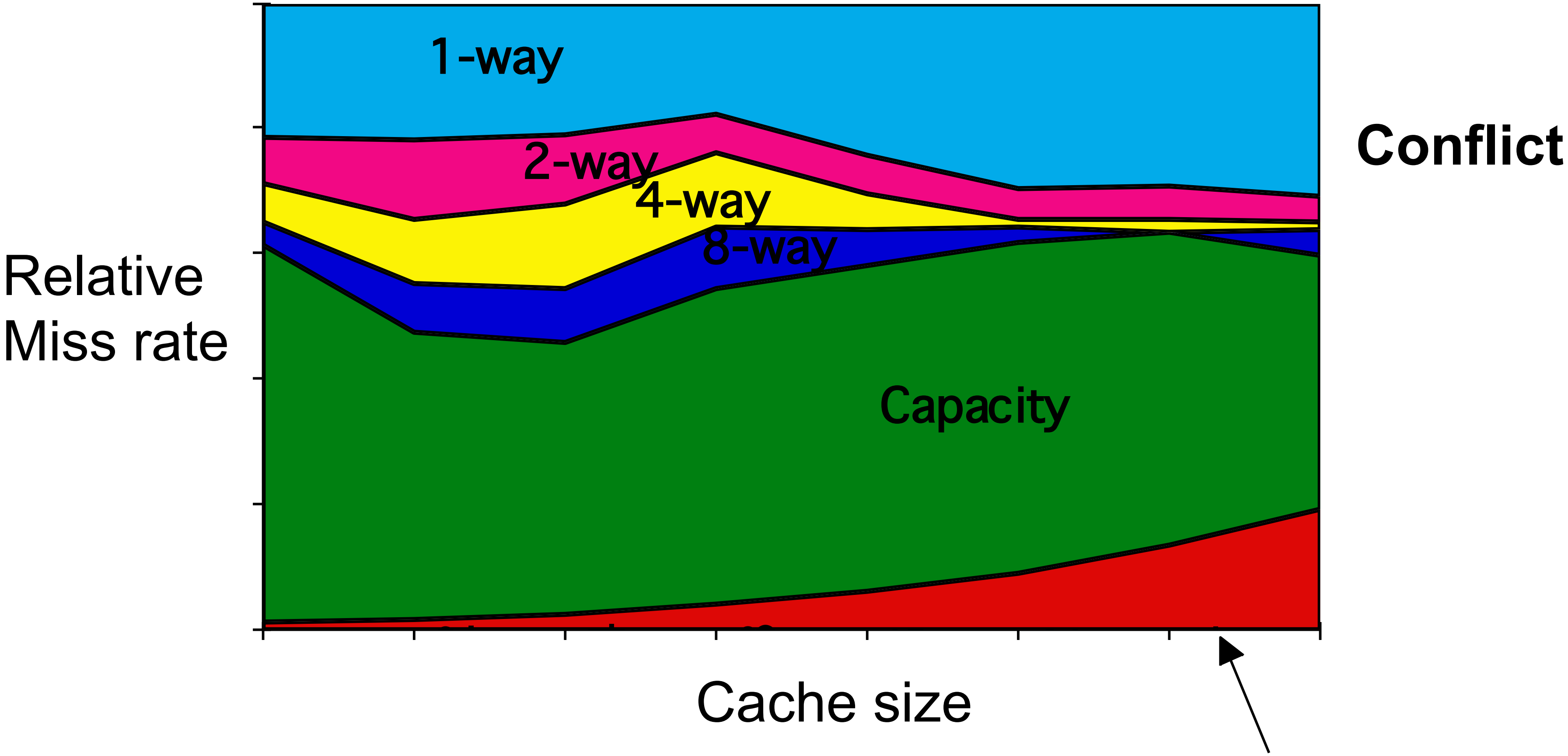
Also: Increase in block size may increase miss penalty.

Associativity



"2:1 cache rule of thumb" : miss rate of direct mapped cache of size N is about same as 2-way of size N/2

3Cs Relative Miss Rate



Flaws: for fixed block size, replacement policy
Good: insight => invention

Cache Performance Impact

(for a simple in-order processor with a "blocking cache")

- Miss-oriented Approach to Memory Access:

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

  - $CPI_{Execution}$ includes ALU and Memory instructions

- Separating out Memory component entirely
  - AMAT = Average Memory Access Time
  - CPI_{ALUOps} does not include memory instructions

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalty \\ &= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst} \right) + \\ &\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data} \right) \end{aligned}$$

Example Problem

- Assume CPI = 1.0 when all memory references hit in the cache. (Assume simple in-order pipeline.)
- Take into account both instruction and data references to memory.
- Only data accesses are loads and stores and these are 50% of instructions.
- Miss penalty is 25 clock cycles.
- Miss rate is 2%.
- How much faster is processor if all instructions hit in cache?

Solution

Computer that always hits:

$\text{CPU time}_{\text{always hits}}$ = (CPU cycles + Memory stall cycles) x cycle time
= (ICxCPI + 0) x cycle time
= IC x 1.0 x cycle time

Computer with real cache:

Memory stall cycles = IC x (Memory accesses/instruction) x miss rate
x miss penalty
= IC x (1 + 0.5) x 0.02 x 25
= IC x 0.75

$\text{CPU time}_{\text{cache}}$ = (IC x 1.0 + IC x 0.75) x cycle time
= 1.75 x IC x cycle time

Comparison: $\text{CPU time}_{\text{cache}}/\text{CPU time}_{\text{always hits}} = 1.75$

(So 75% speedup if always hit… 75% = 1.75 – 1)

What happens on a write? Cache Writing Policies

- Write-through
  - Store to both cache and memory on every write
    - Slow writes
  - Conservative: memory always holds valid data
  - Simple cache

- Write-back
  - Store to **cache only**
    - Fast writes
  - On write, mark data block in cache as **dirty**
    - **Need extra memory bit for each cache line (dirty bit)**
  - Memory may hold invalid/stale data (must fix!)
  - More complex cache

Write Misses

(how are they handled by hardware?)

- Writes can miss in cache… leads to a design choice (the design choice options are answers to question "how do we deal with write miss"). Two choices are:
  - **Write-allocate**
    - First, read cache block from memory into cache
    - Second, proceed with write
  - **No write-allocate**
    - Write to memory directly
    - Memory access is slow, CPU must wait
    - Improve performance using "write buffers" (see later)
- Most common design choices:
  - Writeback cache uses write-allocate
  - Write-through cache uses no write-allocate (keep it simple)

Counter-Intuitive Case 1: Load instruction may cause Writes

- Write-back cache problems
  - Over time, many cache blocks marked “dirty”
  - Eventually, must write dirty data back to memory
    - Usually, when cache block is evicted
    - (Also, if cache is “flushed” for OS context switch)
  - Called a “writeback” event

- Load may cause writes
  - Load instruction miss, may evict a dirty cache block
    - Write first: (old) dirty cache block writes to memory
    - Read second: (new) memory block loaded into cache
  - …but CPU is waiting for read result to do Load…
    - Many CPUs do the “read first”
    - Holds dirty cache block in special “write buffer” during read

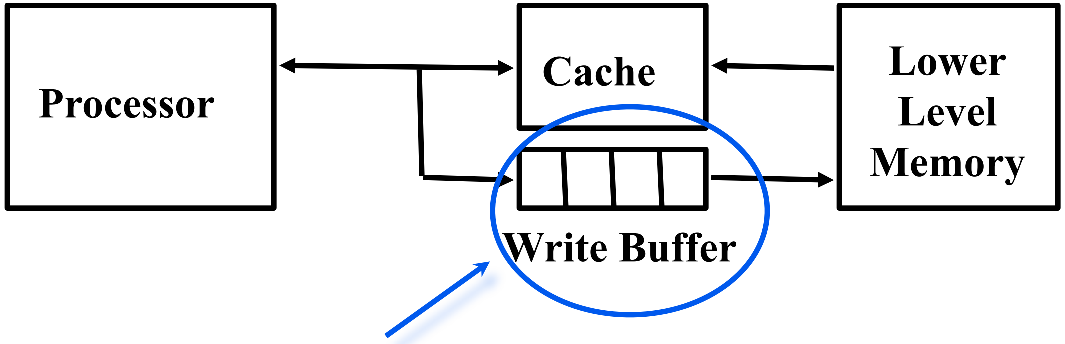
Counter-Intuitive Case 2: Store instruction may cause Reads

- Writes can miss in the cache!

- Store may cause Reads
  - Writeback cache with write-allocate policy
    - Store misses in cache
    - CPU performs READ from memory, puts data in cache
    - CPU performs write in cache, marks block dirty

- It can be even more complex
  - When store misses in cache above
    - Must evict existing cache block
    - If marked dirty, must first write back the cache block to memory
  - CPU performs read from memory
  - CPU performs write in cache

Write Buffers for Write-Through Caches



Holds data awaiting write-through to lower level memory

Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or send read after 1st checking write buffers.

5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
  5. Giving Reads Priority over Writes
     - E.g., Read complete before earlier writes in write buffer

Multiple Cache Levels

- Bigger Cache is slow
- Instead, use multiple levels of caches
  - L1 cache: (level 1 cache) smallest, fastest, in CPU datapath (eg, 8-64kB)
  - L2 cache: usually on-chip with CPU, bigger (eg, 64kB-1MB)
  - L3 cache: on-chip with CPU (eg >1MB), or off-chip (eg, >16MB!)
- Split Cache
  - Separate Instruction and Data Caches
  - Used in L1 cache, tightly coupled to datapath of CPU
  - Necessary for bandwidth: need to read **instruction and data** every cycle (I.e., prevent structural hazards)
- Unified (I + D) Cache
  - Slightly more effective than separate
    - Avoids sizing problem of determining best proportion between I-cache and D-cache
  - Used for L2, L3, etc caches