# Memory System II

# Recap: Cache Performance

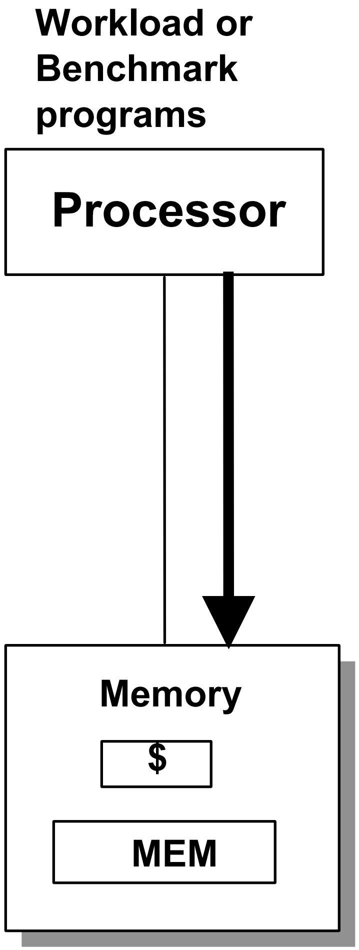
**Execution_Time =**
**Instruction_Count × Cycle_Time ×**
**(ideal CPI + Memory_Stalls/Inst + Other_Stalls/Inst)**

**Memory_Stalls/Inst =**
**Instruction Miss Rate × Instruction Miss Penalty +**
**Loads/Inst × Load Miss Rate × Load Miss Penalty +**
**Stores/Inst × Store Miss Rate × Store Miss Penalty**

**Average Memory Access time (AMAT) =**

$(\text{Hit Rate}_{L1} \times \text{Hit Time}_{L1}) + (\text{Miss Rate}_{L1} \times \text{Miss Time}_{L1})$

# The Art of Memory System Design



Workload or Benchmark programs

Processor

Memory

$

MEM

reference stream

<op, addr>, <op, addr>, <op, addr>, <op, addr>, . . .

op: i-fetch, read, write

*Optimize the memory system organization to minimize the average memory access time for typical workloads*

# Cache Organizations: Cache Lines

- **What Is A Cache Line?**

  **Data is hauled into the cache from memory in “chunks”.**
  - If you ask for 4 bytes of data, you’ll get the whole line (32/64/128 bytes)
  - Locality of reference says you’ll need that data anyway
  - Incur the cost only once rather than each time you ask for a piece of data.

- **How Is The Cache Laid Out?**

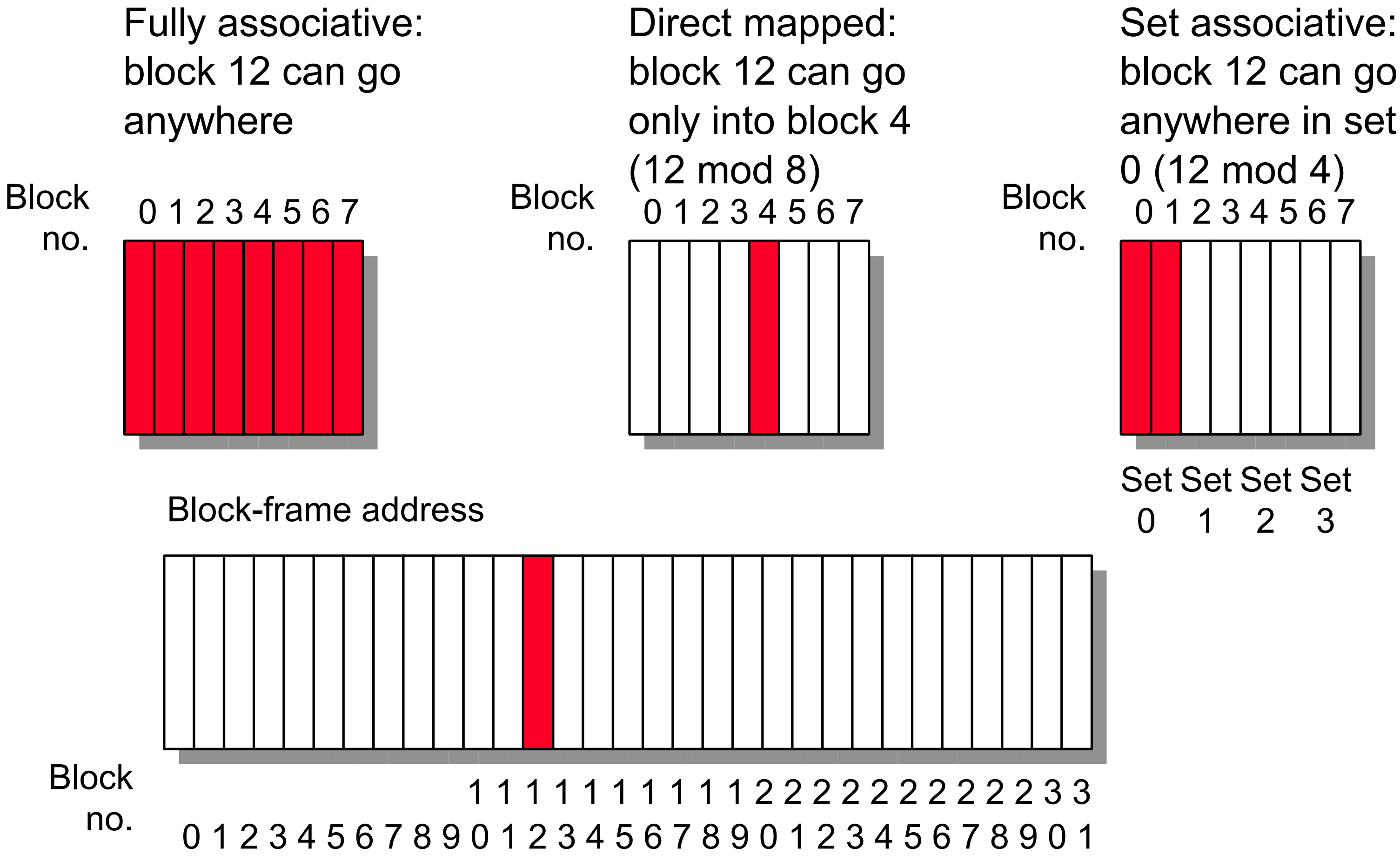
  **The cache is made up of a number of cache lines.**
  - The Level 1 Data Cache of a Pentium P4 Xeon processor contains 8K bytes.
  - The cache lines are each 64 Bytes.
  - This gives 8192 bytes / 64 bytes = 128 cache lines.

- **How Does The Cache Manage the Cache Lines?**

  **Associativity describes how the data is stored in the cache.**
  - Direct Mapped (Associativity == 1 ) means each line has its own slot. (Analogy: Each person gets their own mailbox.)
  - X-way Associativity means X cache lines share a slot. (All the “A’s” share a mailbox, but it’s a bigger mailbox.)
  - Fully associative means all cache lines share the same possible places. (All the letters are put into one giant mailbox.)

# Where can a block be placed in the Cache?

- **Block 12 placed in 8 block cache:**
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets



Fully associative: block 12 can go anywhere

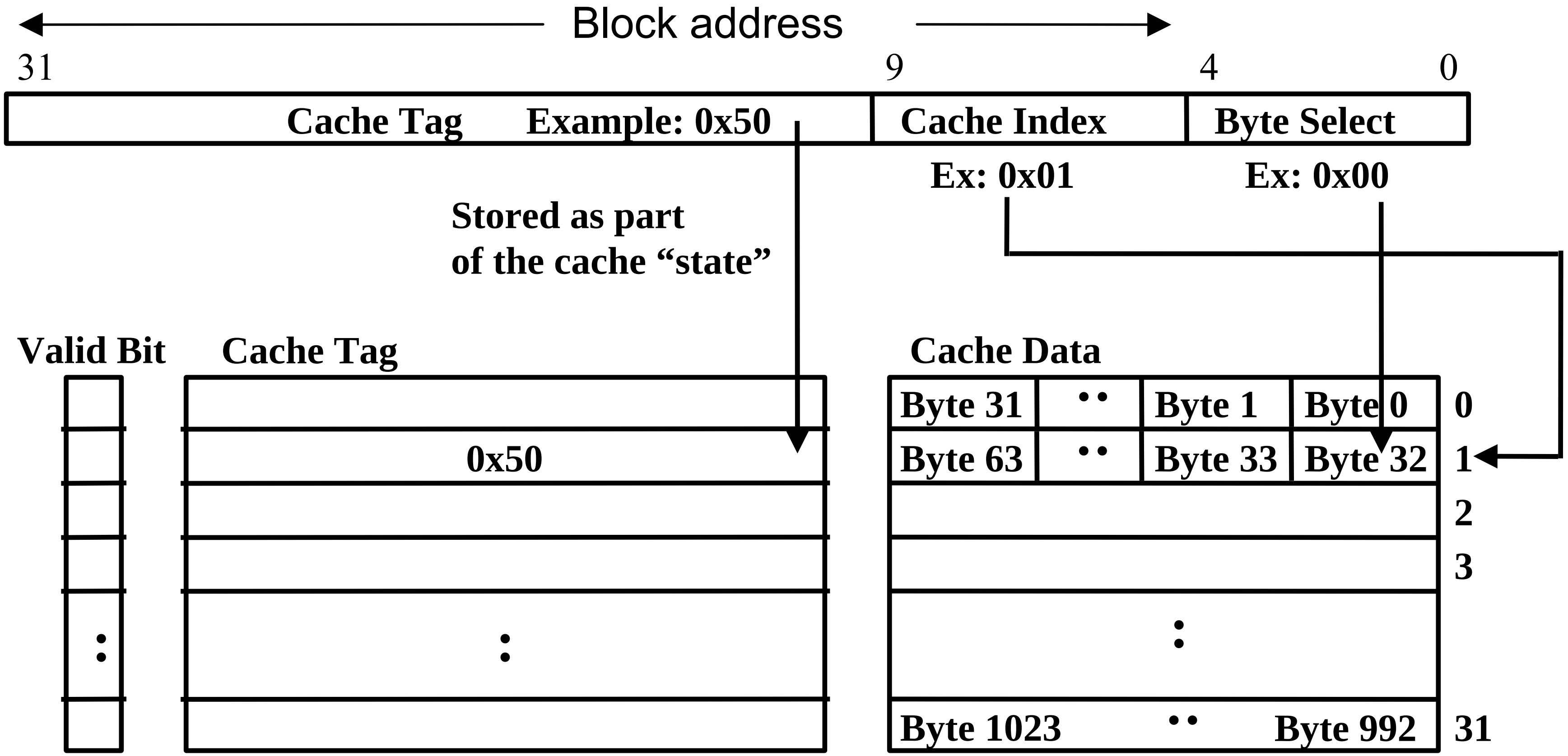
Direct mapped: block 12 can go only into block 4 (12 mod 8)

Set associative: block 12 can go anywhere in set 0 (12 mod 4)

Block-frame address

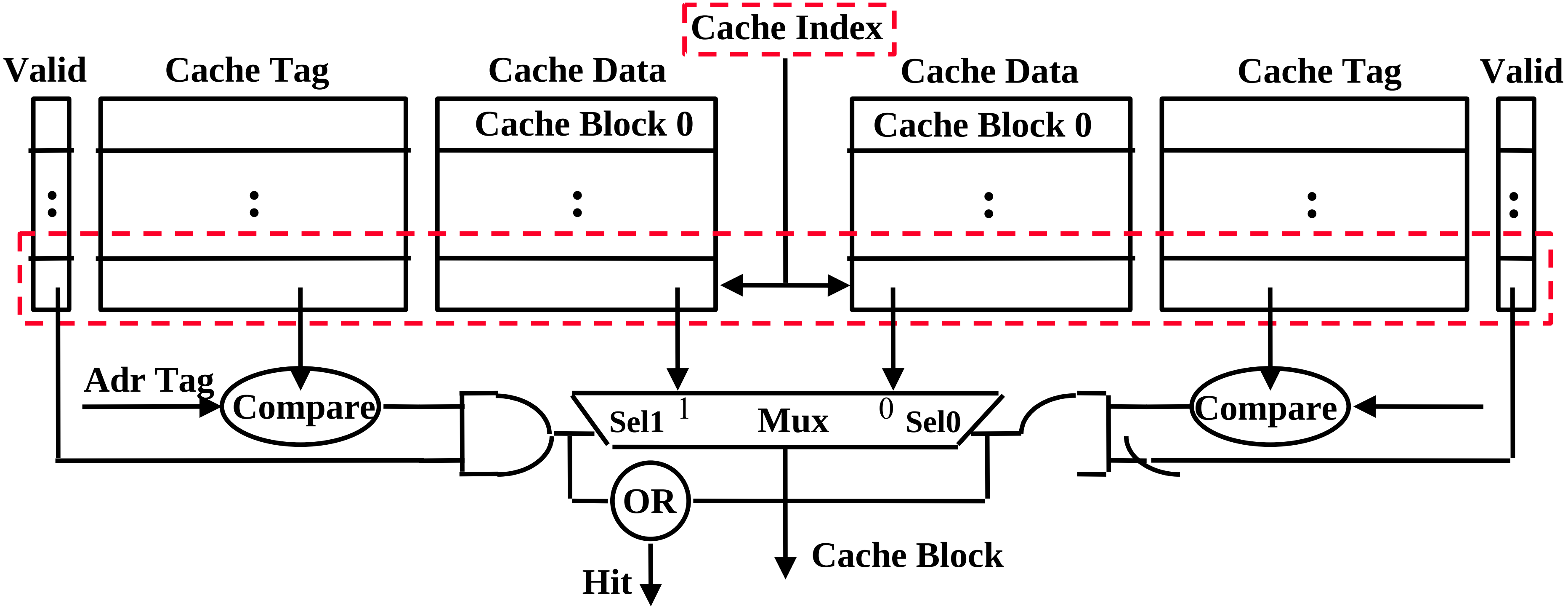
# Cache Organizations I : Direct–Mapped Cache

- **For a $2^N$ byte cache:**
  - 1 KB Direct–Mapped Cache with 32 Byte Lines
  - The upper-most (32 - *N*) bits are always the Cache Tag
  - The lowest *M* bits are the Byte Select (Block Size = $2^M$)
  - One cache miss, pull in complete “Cache Block” (or “Cache Line”)



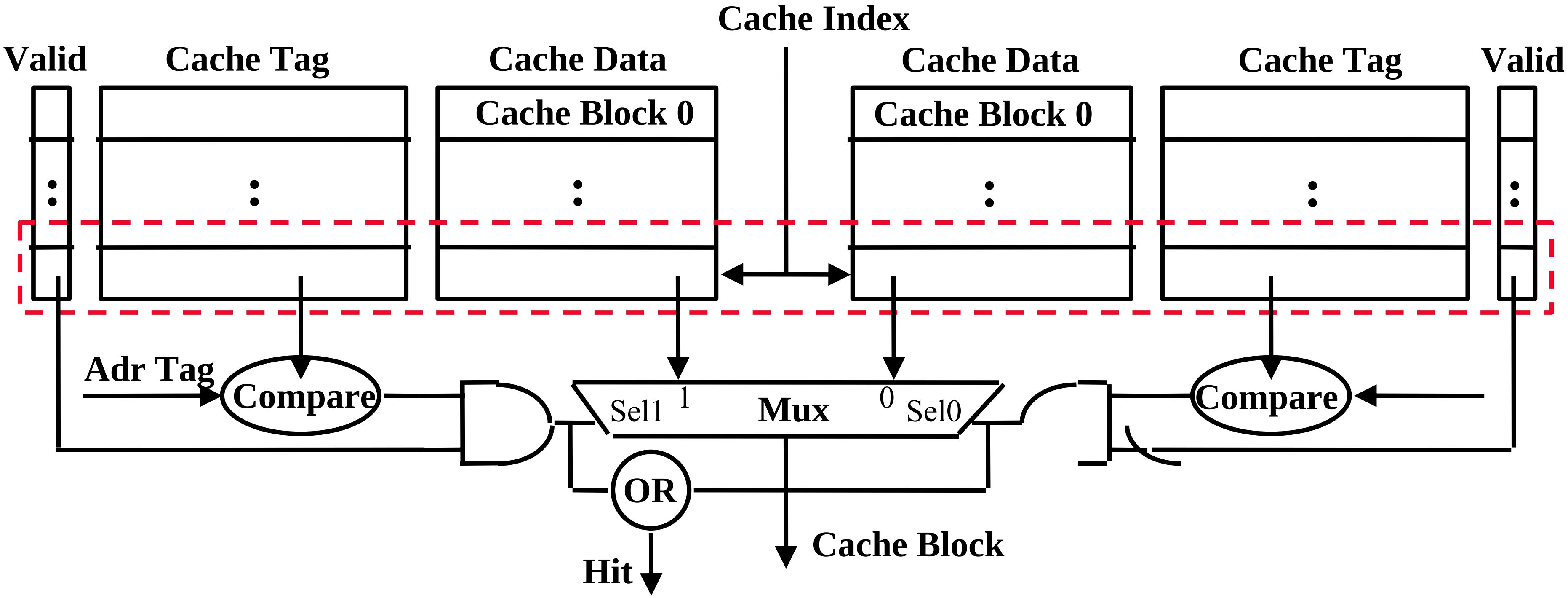
# Cache Organizations II : Set–Associative Cache

- *N*-way Set−Associative: *N* entries for each Cache Index
  - – *N* direct−mapped caches operating in parallel.
- *Example*: 2−way set associative cache:
  - – Cache Index selects a “set” from the cache
  - – The two tags in the set are compared to the input in parallel
  - – Data is selected based on the tag result



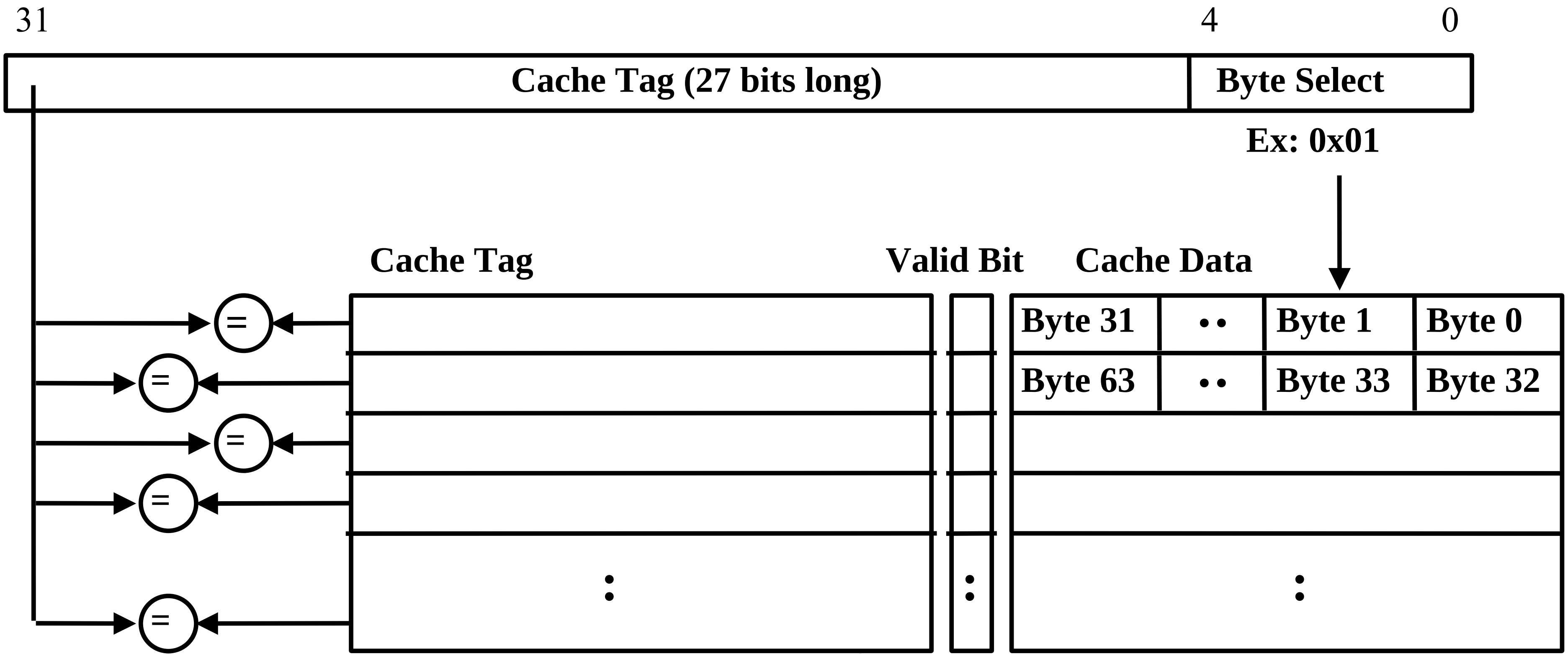
# Disadvantage of Set Associative Cache

- ***N*–way Set Associative Cache *vs*. Direct Mapped Cache:**
  - – *N* comparators *vs*. 1 ;
  - – Extra MUX delay for the data ;
  - – Data comes AFTER Hit/Miss decision and set selection.
- **In a direct–mapped cache, Cache Block is available BEFORE Hit/Miss:**
  - *– Possible to assume a hit and continue. Recover later if miss.*



# Cache Organizations III : Fully–Associative Cache

- **Fully–Associative Cache:**
  - – Forget about the Cache Index.
  - – Compare the Cache Tags of *all cache entries in parallel*.
  - – *Example*: Block Size = 32 B blocks, we need *N* 27-bit comparators.
- **By definition: Conflict Miss = 0 for a fully associative cache**



# The Memory Addresses We'll Be Asking For

Here's a number of addresses. We'll be asking for the data at these addresses and see what happens to the cache when we do so.

| Address | Tag | Set | Off-set | Result |
|---|---|---|---|---|
| | 31 … 9 | 8 … 5 | 4 … 0 | |
| 1090 | 00000000000000000000010 | 0010 | 00010 | Miss |
| | 31 … 9 | 8 … 5 | 4 … 0 | |
| 1440 | 00000000000000000000010 | 1101 | 00000 | Miss |
| | 31 … 9 | 8 … 5 | 4 … 0 | |
| 5000 | xxxxxxxxxxxxxxxxxxxxxxx | xxxx | 01000 | |
| | 31 … 9 | 8 … 5 | 4 … 0 | |
| 1470 | xxxxxxxxxxxxxxxxxxxxxxx | xxxx | xxxxx | |

Cache:

1. Is Direct Mapped
2. Contains 512 bytes.
3. Has 16 sets.
4. Each set can hold 32 bytes – 1 cache line.

# Here's the Cache We'll Be Touching

Initially the cache is empty.

Cache:
1. Is Direct Mapped
2. Contains 512 bytes.
3. Has 16 sets.
4. Each set can hold 32 bytes – 1 cache line.

| Set Address | V | Tag | Data<br>(Can hold a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | N | | |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | N | | |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | N | | |
| 13 (1101) | N | | |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to READ data from address 1090 = 010|0010|00010

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data<br>(Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000….10 | Data from memory loc. 1088 - 1119 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | N | | |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | N | | |
| 13 (1101) | N | | |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to READ data from address 1440 = 010|1101|00000

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data (Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000….10 | Data from memory loc. 1088 - 1119 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | N | | |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | N | | |
| 13 (1101) | Y | 00000….10 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to READ data from address 5000
= 1001|1100|01000

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data<br>(Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000…….10 | Data from memory loc. 1088 - 1119 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | N | | |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | Y | 00000….1001 | Data from memory loc. 4992 - 5023 |
| 13 (1101) | Y | 00000…0010 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to READ data from address 1470 = 0010|1101|11110

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data<br>(Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000…….10 | Data from memory loc. 1088 - 1119 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | N | | |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | Y | 00000….1001 | Data from memory loc. 4992 - 5023 |
| 13 (1101) | Y | 00000….0010 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to READ data from address 1600 = 0011|0010|00000

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data<br>(Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000….0011 | Data from memory loc. 1600 - 1631 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | N | | |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | Y | 00000….1001 | Data from memory loc. 4992 - 5023 |
| 13 (1101) | Y | 00000…00010 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to WRITE data from address 256
= 0000|1000|00000

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data (Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000….0011 | Data from memory loc. 1600 - 1631 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | Y | 00000….0000 | Data from memory loc. 256 - 287 |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | Y | 00000….1001 | Data from memory loc. 4992 - 5023 |
| 13 (1101) | Y | 00000…00010 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to WRITE data from address 1620
= 0011|0010|10100

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data<br>(Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000….0011 | Data from memory loc. 1600 - 1631 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | Y | 00000….0000 | Data from memory loc. 256 - 287 |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | Y | 00000….1001 | Data from memory loc. 4992 - 5023 |
| 13 (1101) | Y | 00000…00010 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Doing Some Cache Action

We want to WRITE data from address 1099 = 0010|0010|01011

| Add. | Tag | Set | Offset |
|---|---|---|---|
| 256 | 0000 | 1000 | 00000 |
| 512 | 0001 | 0000 | 00000 |
| 1024 | 0010 | 0000 | 00000 |
| 1090 | 0010 | 0010 | 00010 |
| 1099 | 0010 | 0010 | 01011 |
| 1440 | 0010 | 1101 | 00000 |
| 1470 | 0010 | 1101 | 11110 |
| 1600 | 0011 | 0010 | 00000 |
| 1620 | 0011 | 0010 | 10100 |
| 2048 | 0100 | 0000 | 00000 |
| 4096 | 1000 | 0000 | 00000 |
| 5000 | 1001 | 1100 | 01000 |

| Set Address | V | Tag | Data<br>(Always holds a 32-byte cache line.) |
|---|---|---|---|
| 0 (0000) | N | | |
| 1 (0001) | N | | |
| 2 (0010) | Y | 00000….0010 | Data from memory loc. 1088 - 1119 |
| 3 (0011) | N | | |
| 4 (0100) | N | | |
| 5 (0101) | N | | |
| 6 (0110) | N | | |
| 7 (0111) | N | | |
| 8 (1000) | Y | 00000….0000 | Data from memory loc. 256 - 287 |
| 9 (1001) | N | | |
| 10 (1010) | N | | |
| 11 (1011) | N | | |
| 12 (1100) | Y | 00000….1001 | Data from memory loc. 4992 - 5023 |
| 13 (1101) | Y | 00000…00010 | Data from memory loc. 1440 - 1471 |
| 14 (1110) | N | | |
| 15 (1111) | N | | |

# Sources of Cache Misses

1 **Compulsory (cold start or process migration, first reference): first access to a block.**

- "Cold" fact of life: not a whole lot you can do about it (pre-load?)
- *Note*: If you are going to run "billions" of instruction, Compulsory Misses are *insignificant*. If locality is low, compulsary misses reverse the benefit of even using a cache.

2 **Capacity:**

- Cache cannot contain *simultaneously* all blocks accessed by the program;
- Solution: increase cache size.

3 **Conflict (collision):**

- Multiple memory locations mapped to the *same cache* location;
- Solution 1: increase cache size;
- Solution 2: increase associativity.

4 **Coherence (Invalidation): other process (*e.g.*, I/O, other Processor) updates shared memory.**

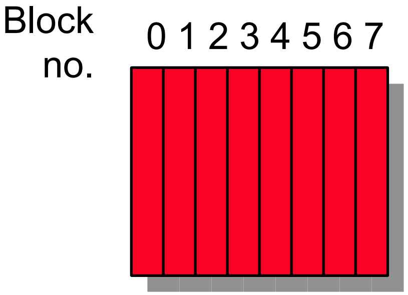
# Four Basic Questions for Caches and Memory Hierarchies

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

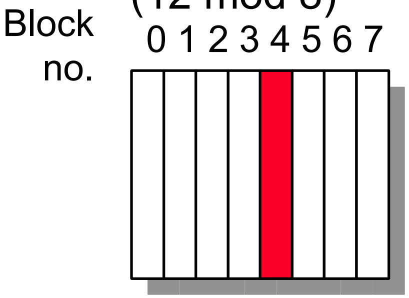
# Q1: Where can a block be placed in the upper level?

- **Block 12 placed in 8 block cache:**
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

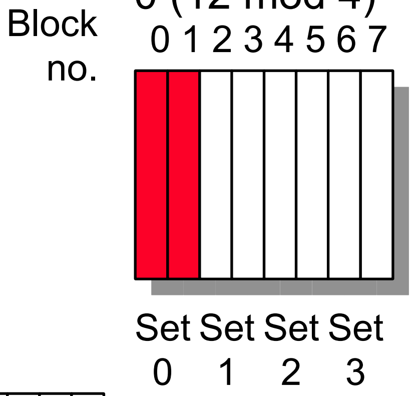
Fully associative:
block 12 can go
anywhere



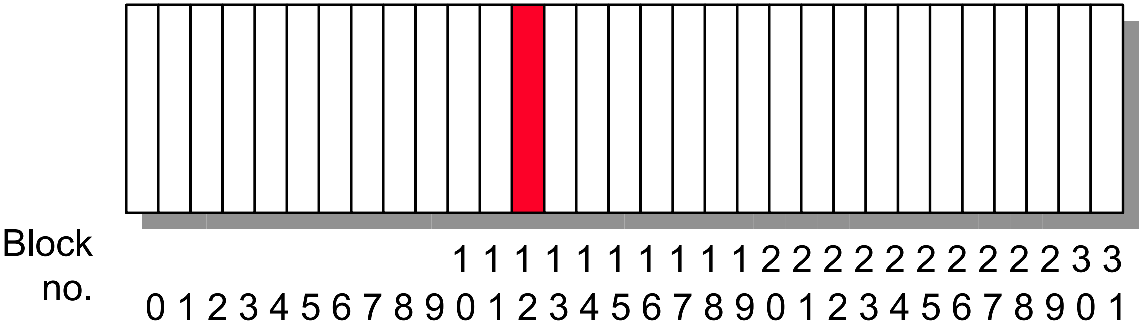
Direct mapped:
block 12 can go
only into block 4
(12 mod 8)



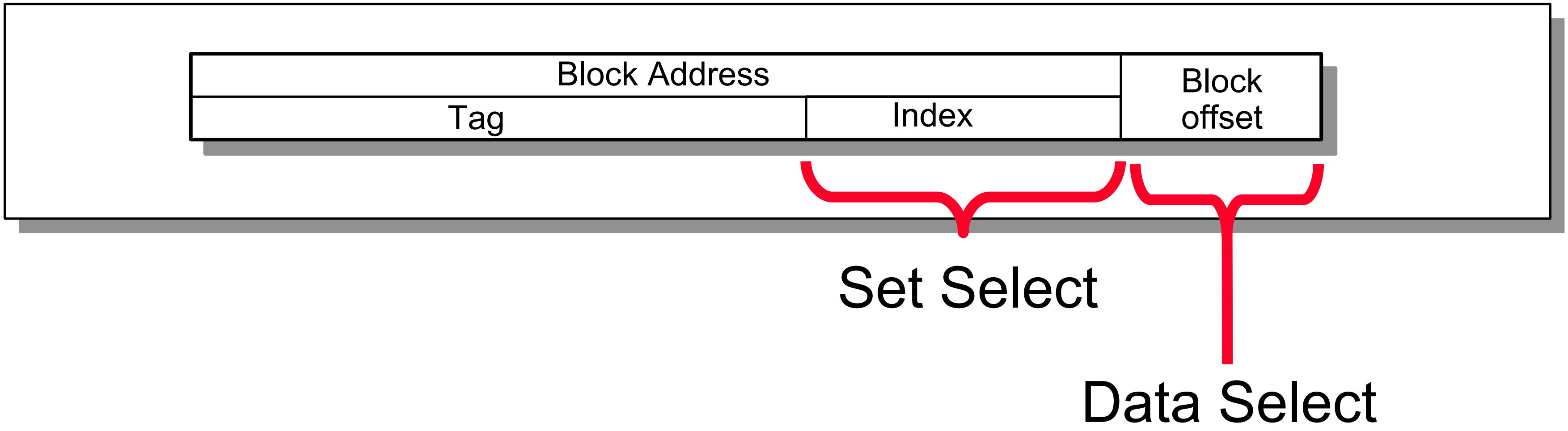
Set associative:
block 12 can go
anywhere in set
0 (12 mod 4)



Block-frame address



# Q2: How is a block found if it is in the upper level?



- **Direct indexing (using index and block offset), tag compares, or combination**
- **Increasing associativity shrinks index, expands tag**

# Q3: Which block should be replaced on a miss?

- **Easy for Direct Mapped**
  - One choice only.
- **Set Associative or Fully Associative:**
  - Random: easy to implement;
  - LRU (Least Recently Used): costlier.

**Associativity:**

| | 2-way | | 4-way | | 8-way | |
|---|---|---|---|---|---|---|
| **Size** | **LRU** | **Random** | **LRU** | **Random** | **LRU** | **Random** |
| 16 KB | 5.2% | 5.7% | 4.7% | 5.3% | 4.4% | 5.0% |
| 64 KB | 1.9% | 2.0% | 1.5% | 1.7% | 1.4% | 1.5% |
| 256 KB | 1.15% | 1.17% | 1.13% | 1.13% | 1.12% | 1.12% |

# Replacement algorithm

- **Least Recently Used (approximation of LRU)**
  - Time is divided into time window of length W seconds
  - Each page P has a usage bit (U(P)) and a counter (C(P))
  - Initially U(P)=0 at start of each window
  - Operations:
    - A reference to P leads U(p)=1 and C(p)=C(p)+1
    - P is candidate for replacement if there is P with U(p)=0
    - Else (all U(p) are 1) Find P : C(P)= is least.
  - Note: (1) each page requires a counter that must be incremented on every reference, (2) requires setting up proper value of W for correct operation, (3) what is a procedure to increase or decrease the value of W!

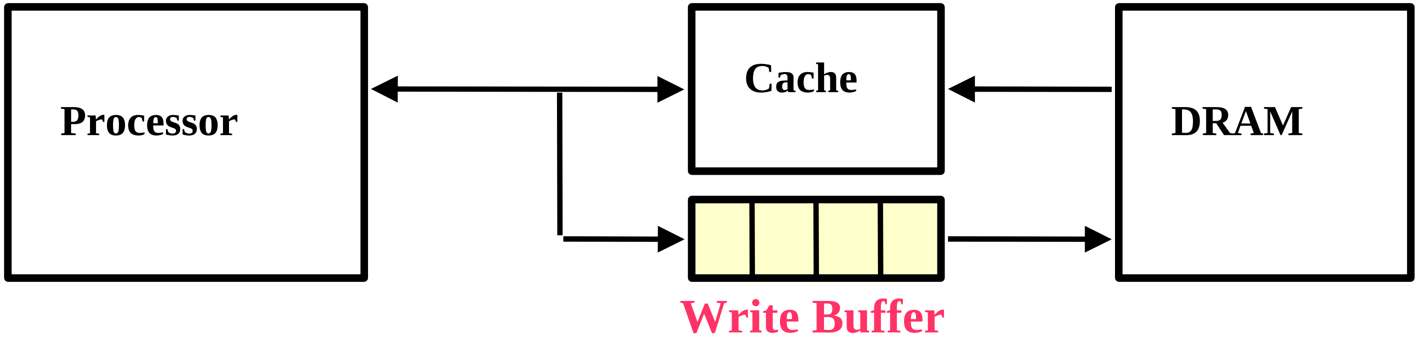
# Replacement algorithm

- **Working Set Algorithm (approximation of LRU)**
  - Time is divided into time window of length W seconds
  - Each page P has a counter (C(P))
  - Initially C(P)=0 at  start of each window
  - A high frequency clock is used to increment the counter of each page at each clock
  - Operations:
    - A reference to P leads C(p)=0
    - P is candidate for replacement if its C(P) is highest
  - Note: (1) each page requires a counter that must be incremented on every clock not every reference, (2) less overhead at each reference as compared to previous approach, (3) requires setting up proper value of W for correct operation, (3) what is a procedure to increase or decrease the value of W!

# Q4: What happens on a write?

- *Write through*—The information is written to *both* the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written *only* to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
- WT always combined with write buffers so that don’t wait for lower level memory

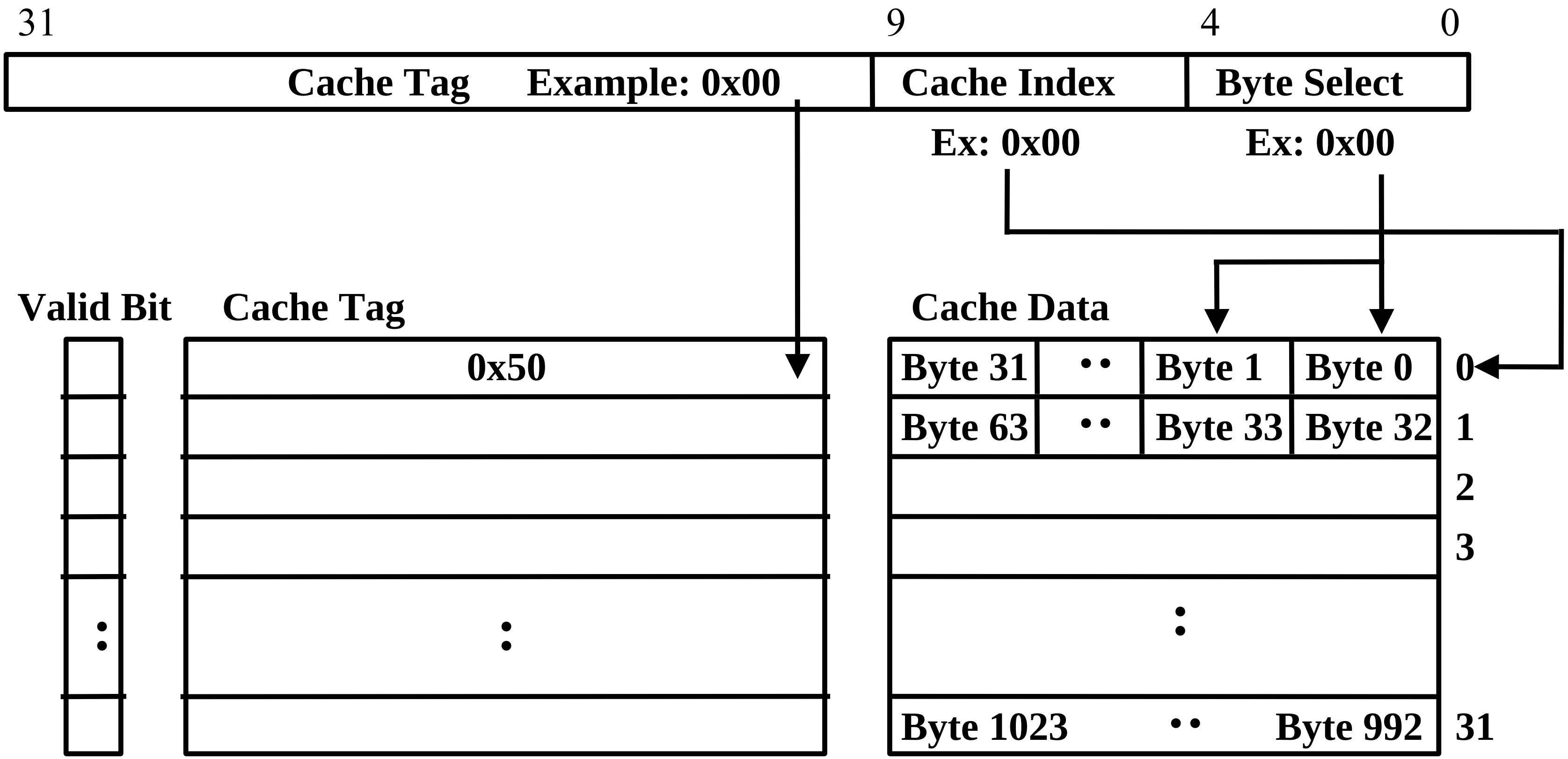
# Write Buffer for Write Through



- **A Write Buffer is needed between the Cache and Memory**
  - Processor: writes data into the cache *and* the write buffer;
  - Memory controller: write contents of the buffer to memory.
- **Write buffer is just a FIFO:**
  - Typical number of entries: 4;
  - Must handle bursts of writes;
  - Works fine if: store frequency (w.r.t. time) << 1 / DRAM write cycle. (Why ?)

# Write-miss Policy: Write Allocate *vs.* Not Allocate

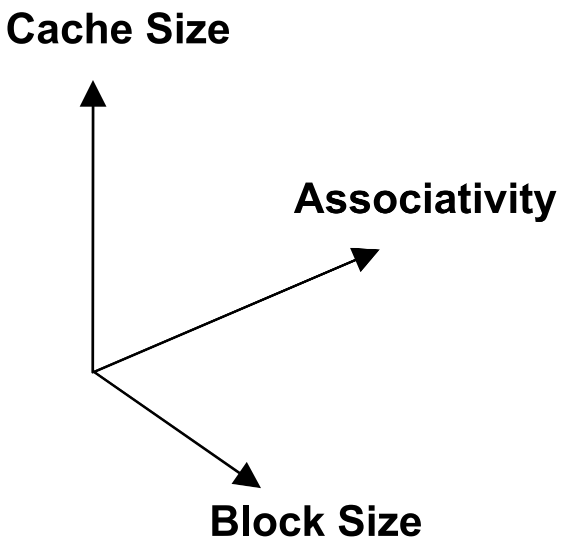
- **Assume: a 16-bit (sub-block) write to memory location 0x0 and causes a miss. Do we allocate space in cache and possibly read in the block?**
  - Yes: → Write Allocate
  - No: → Not Write Allocate (Usually we do this).



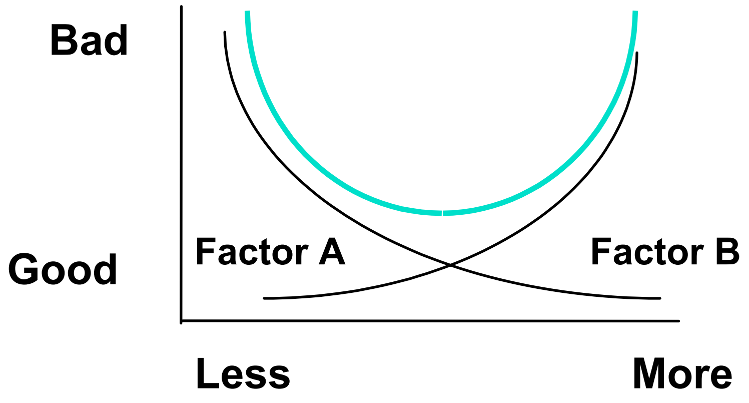
# Summary #1/ 2:

- **The Principle of Locality:**
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- **Three (+1) Major Categories of Cache Misses:**
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity.
    Nightmare Scenario: ping pong effect!
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices
- **Cache Design Space**
  - total size, block size, associativity
  - replacement policy
  - write-hit policy (write-through, write-back)
  - write-miss policy

# Summary #2 / 2: The Cache Design Space

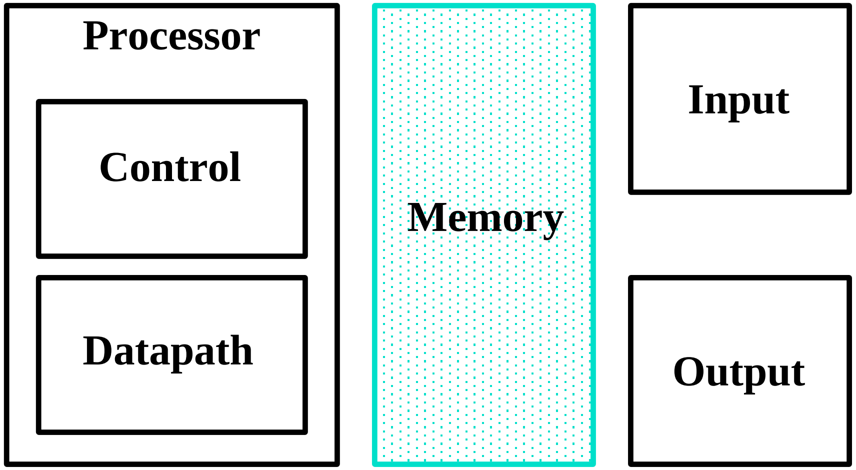


- **Several *interacting* dimensions**
  - cache size,
  - block size,
  - associativity,
  - replacement policy,
  - write-through *vs*. write-back,
  - write allocation policy.
- **The optimal choice is a compromise**
  - depends on access characteristics
    - workload,
    - use (I-cache, D-cache,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



# The Big Picture: Where are We Now?

The Five Classic Components of a Computer

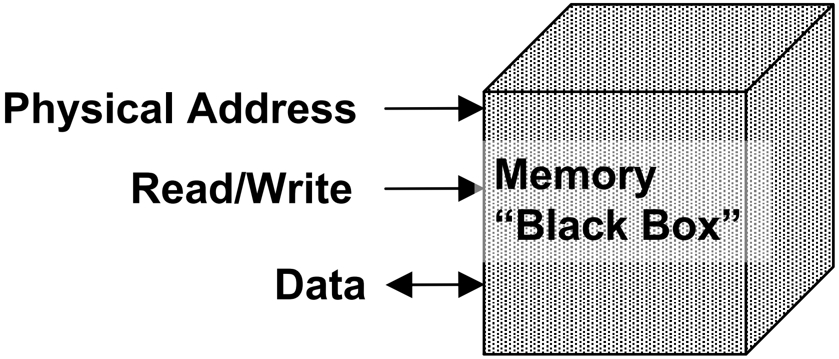


What happens next?

– How to improve cache performance.

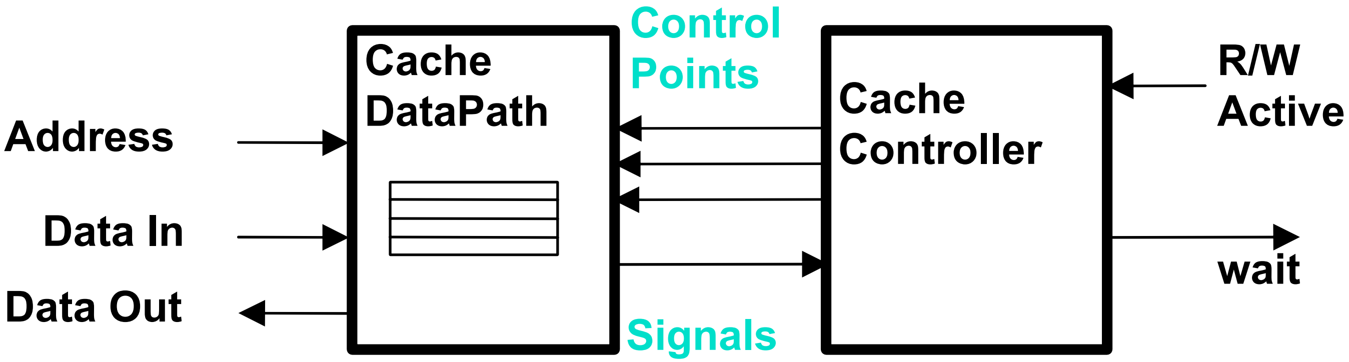
# How Do you *Design* a Memory System?

- **Set of Operations that must be supported**
  - read: data <= Mem[Physical Address]
  - write: Mem[Physical Address] <= Data



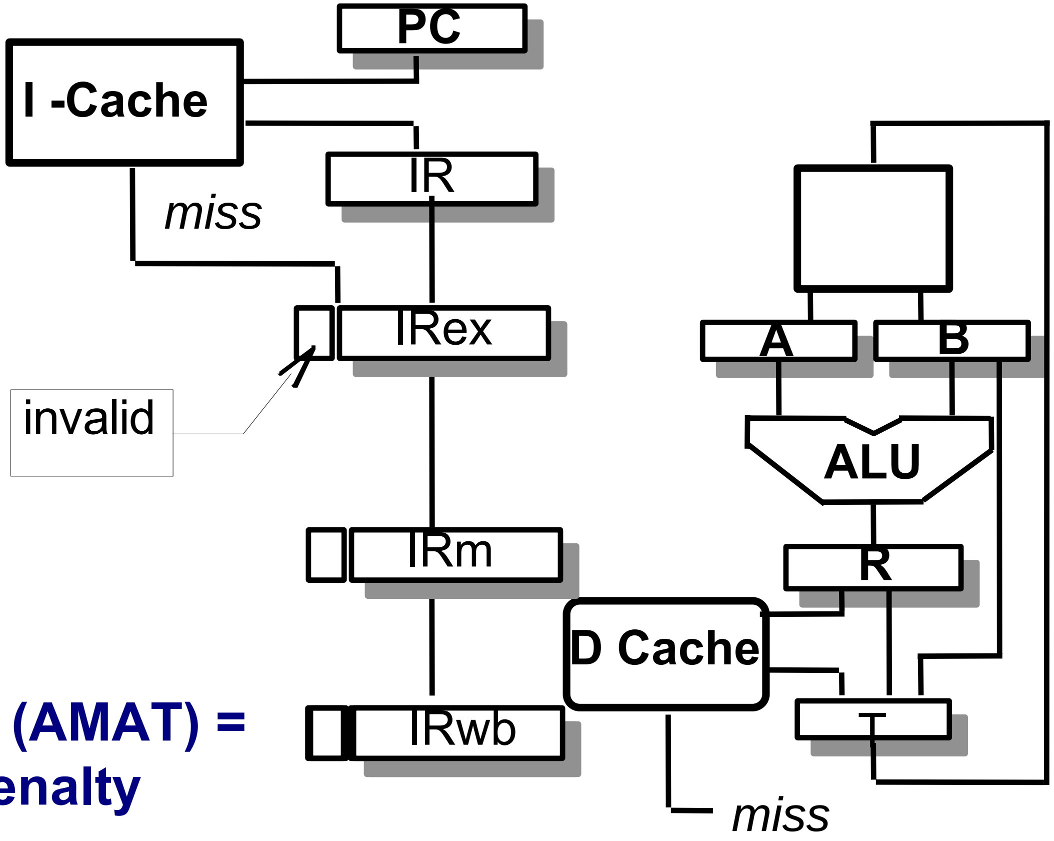
**Inside it has:**
**Tag-Data Storage,**
**Muxes,**
**Comparators, . . .**

- **Determine the internal register transfers**
- **Design the Datapath**
- **Design the Cache Controller**



# Impact on Cycle Time

**Cache Hit Time:**
**directly tied to clock rate**
**increases with cache size**
**increases with associativity**



**Average Memory Access time (AMAT) =**
**Hit Time + Miss Rate × Miss Penalty**

**Time = IC × CT × (ideal CPI + memory stalls)**

# Improving Cache Performance: 3 general options

**Average Memory Access time =**

**Hit Time + (Miss Rate x Miss Penalty) =**

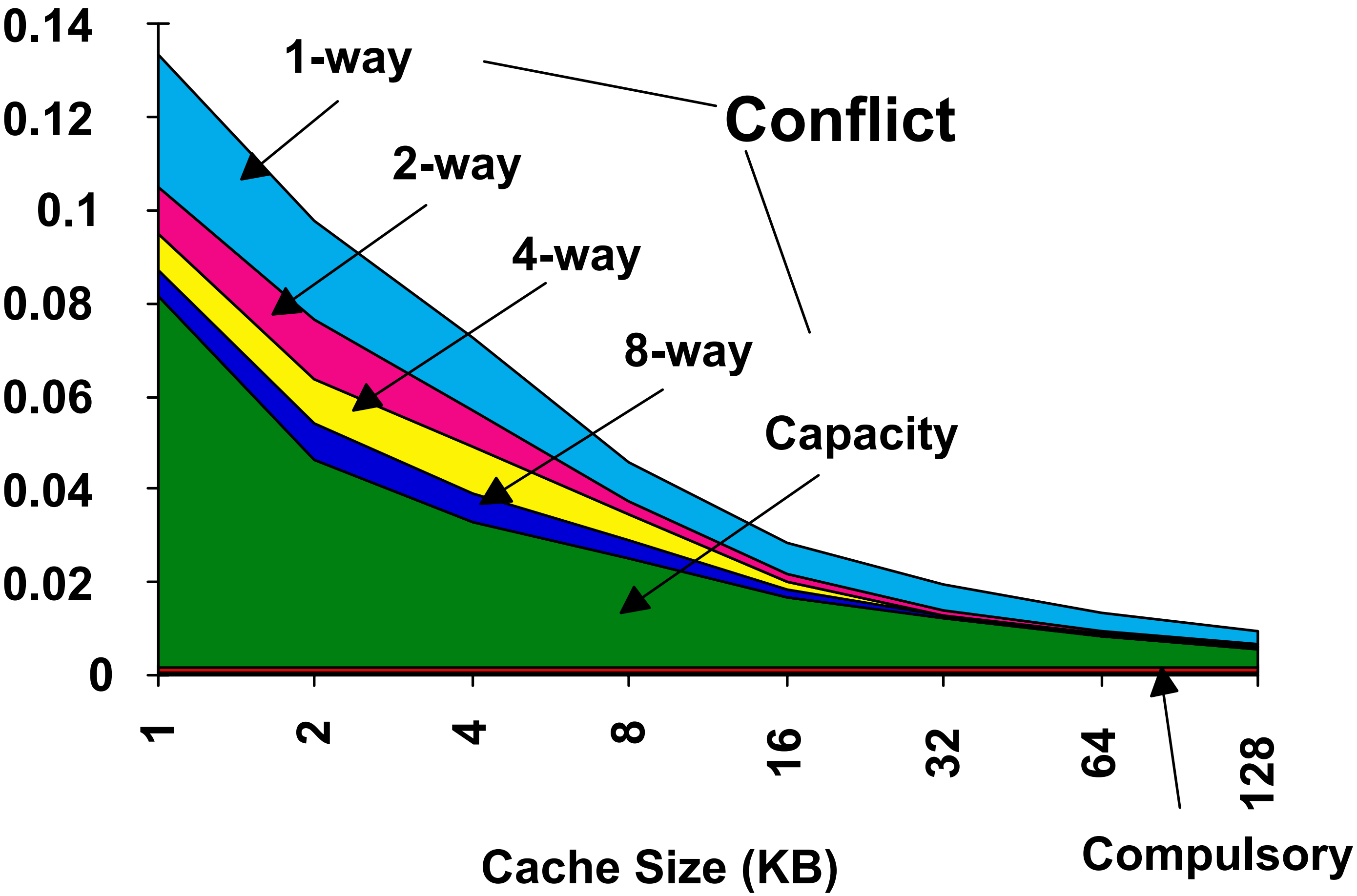
**(Hit Rate x Hit Time) + (Miss Rate x Miss Time)**

- **Options to reduce Average Memory Access Time == AMAT:**
  - 1. Reduce the miss rate,
  - 2. Reduce the miss penalty, or
  - 3. Reduce the time to hit in the cache.

# Improving Cache Performance

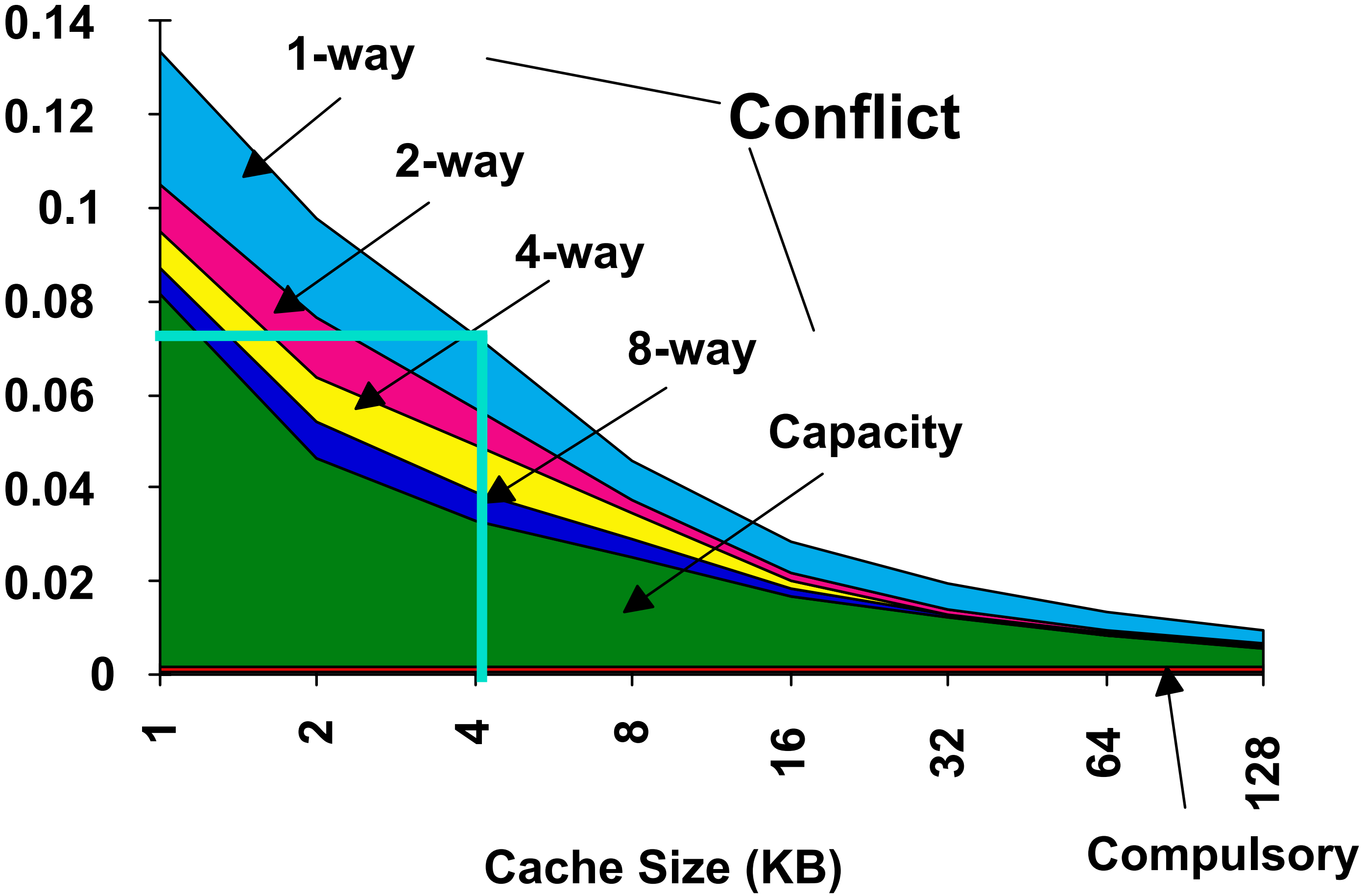
1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

# 3Cs Absolute Miss Rate (SPEC92)

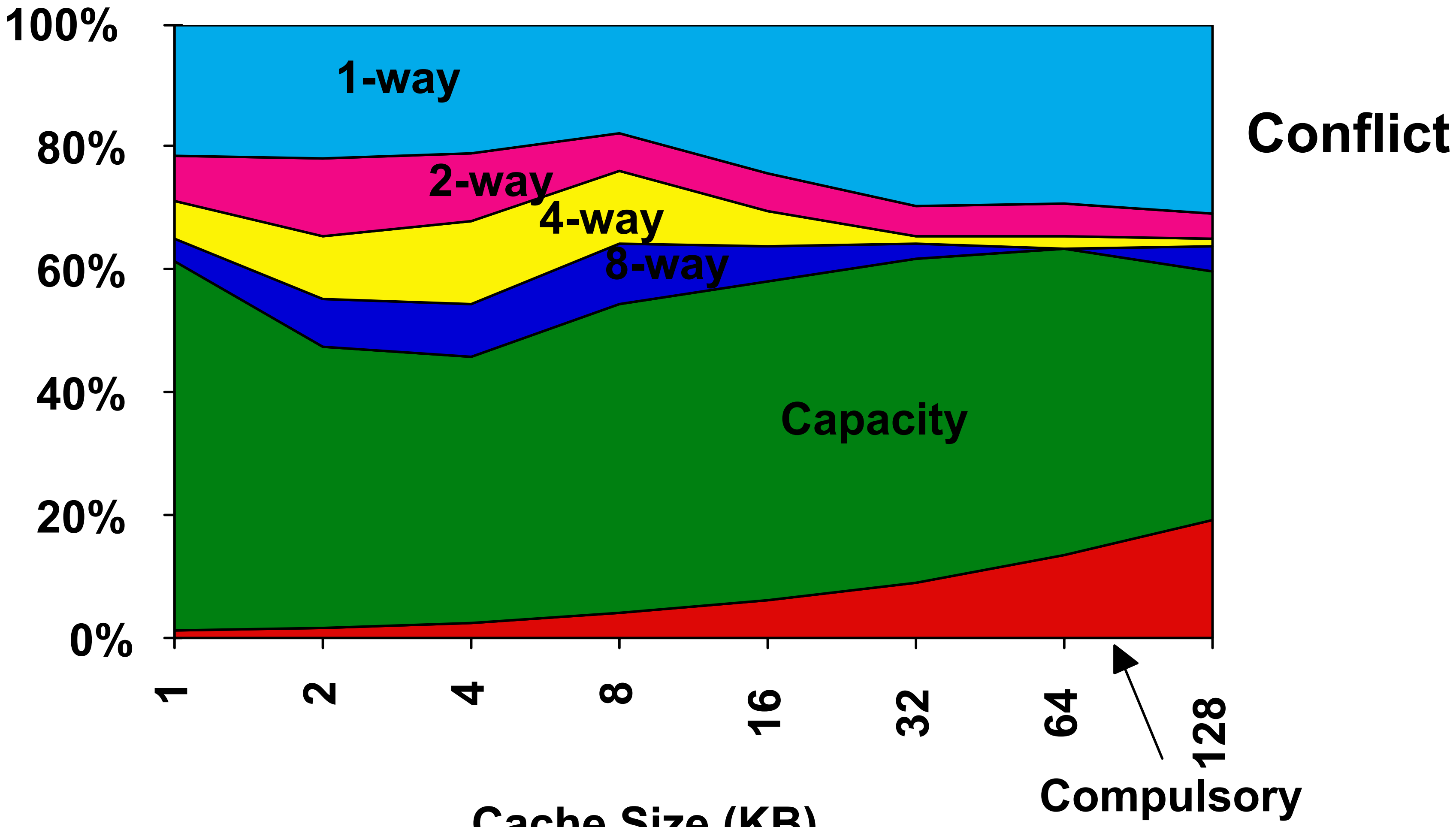


# 2:1 Cache Rule

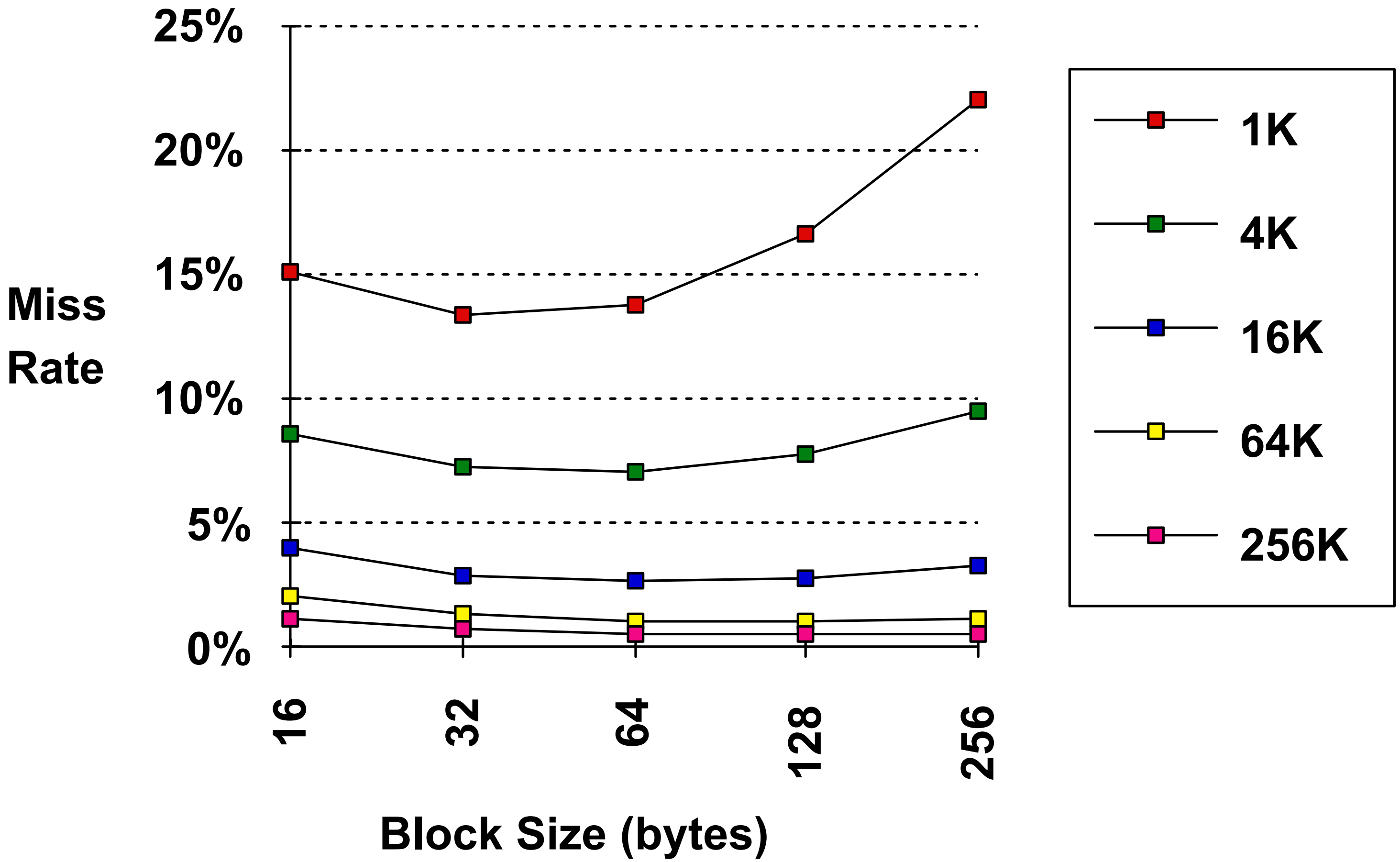
**miss rate 1-way associative cache size *X***
**= miss rate 2-way associative cache size *X*/2**



# 3Cs Relative Miss Rate



# 1. Reduce Misses via Larger Block Size



# 2. Reduce Misses via Higher Associativity

- **2:1 Cache Rule**:
  - Miss Rate DM cache size *M* ≅ Miss Rate 2-way cache size *N*/2
- **Beware: Execution time is only *final* measure!**
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way *vs*. 1-way:
    - external cache +10%,
    - internal + 2% .

# Avg. Memory Access Time *vs.* Miss Rate

- **Assume CCT =**

  1.10 for 2-way,

  1.12 for 4-way,

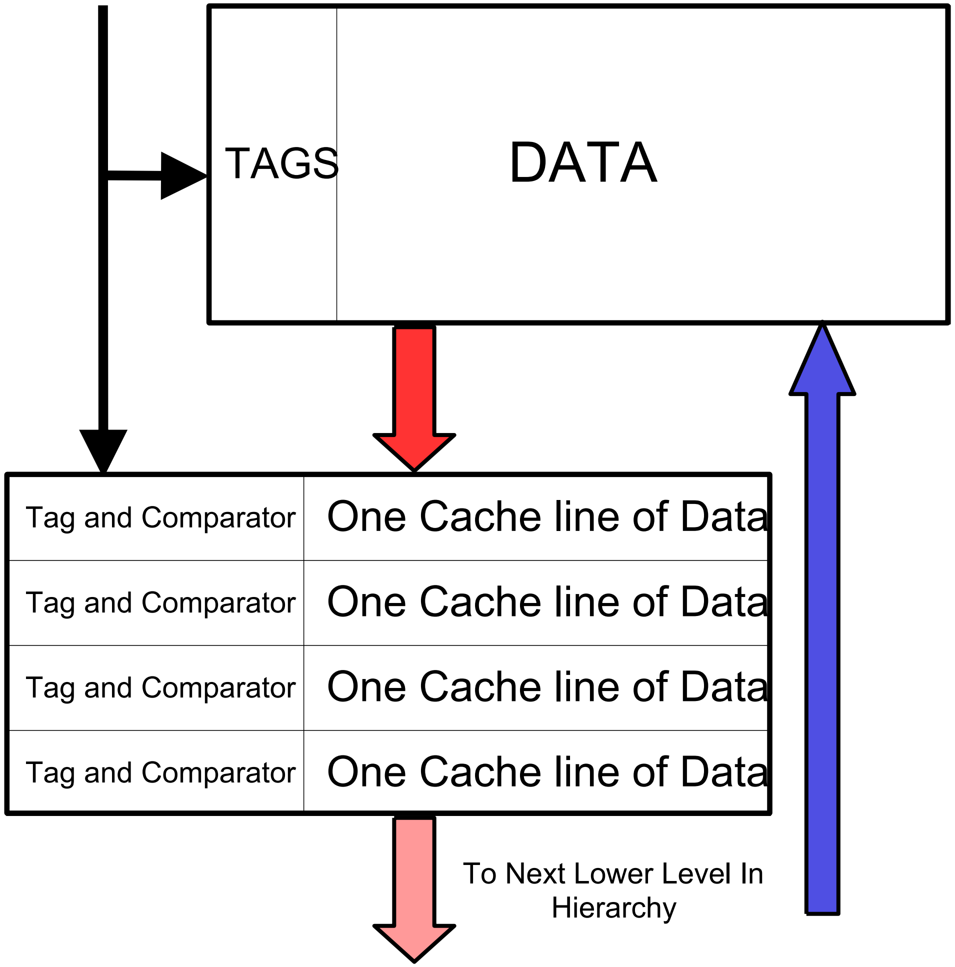
  1.14 for 8-way *vs.* CCT direct mapped

| **Cache Size** | | **Associativity** | | |
|---|---|---|---|---|
| **(KB)** | **1-way** | **2-way** | **4-way** | **8-way** |
| **1** | **2.33** | **2.15** | **2.07** | **2.01** |
| **2** | **1.98** | **1.86** | **1.76** | **1.68** |
| **4** | **1.72** | **1.67** | **1.61** | **1.53** |
| **8** | **1.46** | **1.48** | **1.47** | **1.43** |
| **16** | **1.29** | **1.32** | **1.32** | **1.32** |
| **32** | **1.20** | **1.24** | **1.25** | **1.27** |
| **64** | **1.14** | **1.20** | **1.21** | **1.23** |
| **128** | **1.10** | **1.17** | **1.18** | **1.20** |

**(Red means A.M.A.T. not improved by more associativity)**

# 3. Reducing Misses via a “Victim Cache”

- How to combine fast hit time of direct mapped, yet still avoid conflict misses?
- Add buffer to place data discarded from cache.
  - Fully-Associative Write buffer.
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines



# 4. Reducing Misses by Hardware Prefetching

- ***e.g.*, Instruction Prefetching:**
  - Alpha 21064 fetches 2 blocks on a miss;
  - Extra block placed in "stream buffer";
  - On miss check stream buffer.
- **Works with data blocks too H/W Data Prefetching:**
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from
    2 64KB, 4-way set associative caches
- **Prefetching relies on having extra memory bandwidth that can be used without penalty;**
  - Could (and will) reduce performance if done indiscriminantly!!!

# 5. Reducing Misses by Software Prefetching Data

- **Data Prefetching**
  - Pre–load data into register (HP PA-RISC loads).
  - Cache Pre–fetch: load into cache (MIPS IV, PowerPC, SPARC *v*. 9).
  - Special prefetching instructions cannot cause faults; a form of *speculative execution.*

- **Issuing Prefetch Instructions takes time**
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

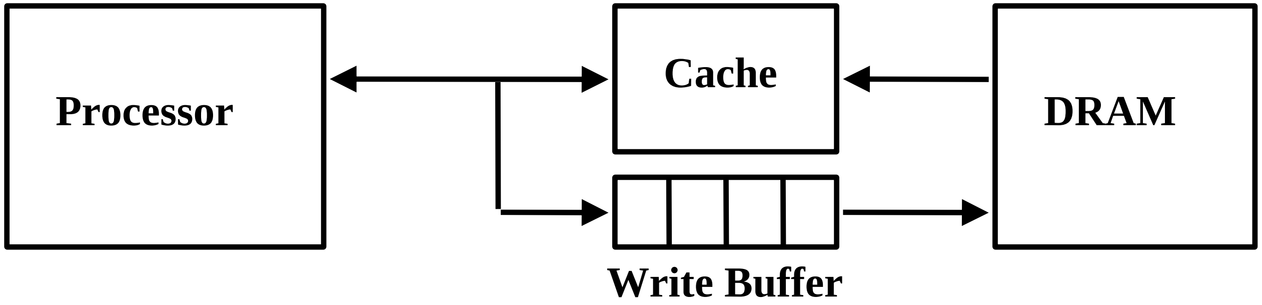
# Improving Cache Performance (*Cont'd*)

1. Reduce the miss rate,
2. *Reduce the miss penalty*, or
3. Reduce the time to hit in the cache.

# 0. Reducing Miss Penalty: Faster DRAM / Interface

- **New DRAM Technologies**
  - RAMBUS - same initial latency, but much higher bandwidth;
  - Synchronous DRAM (uses clock);
  - TMJ-RAM (Tunneling magnetic-junction RAM) from IBM??
  - Merged DRAM/Logic – IRAM project here at Berkeley.
- **Better BUS interfaces.**
- **CRAY Technique: only use SRAM (!).**

# 1. Reducing Miss Penalty: Read Priority over Write on Miss



- **A Write Buffer is needed between the Cache and Memory**
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
  - Must handle burst behavior as well!

# 2. Reduce Miss Penalty: Early Restart and Critical Word First

- **Don't wait for full block to be loaded before restarting CPU**
  - *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - *Critical Word First*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called *wrapped fetch* and *requested word first*
- **Generally useful only in large blocks.**
- **Spatial locality a problem; tend to want next sequential word, so not clear if benefit by early restart.**
  - Beneficial for frequent access to different words within the same window of time (when spatial locality is low.)

# 3. Reduce Penalty: Non-blocking Caches

- ***Non-blocking cache*** **or** ***lockup-free cache*** **allow data cache to continue to supply cache hits during a miss:**
  - requires F/E bits on registers, or out-of-order execution;
  - requires multi-bank memories.
- **"*hit under miss*" reduces the effective miss penalty by working during miss *vs*. ignoring CPU requests**
- **"*hit under multiple miss*" or "*miss under miss*" may further lower the effective miss penalty by overlapping multiple misses**
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses;
  - Requires muliple memory banks (otherwise cannot support);
  - Penium Pro allows 4 outstanding memory misses.

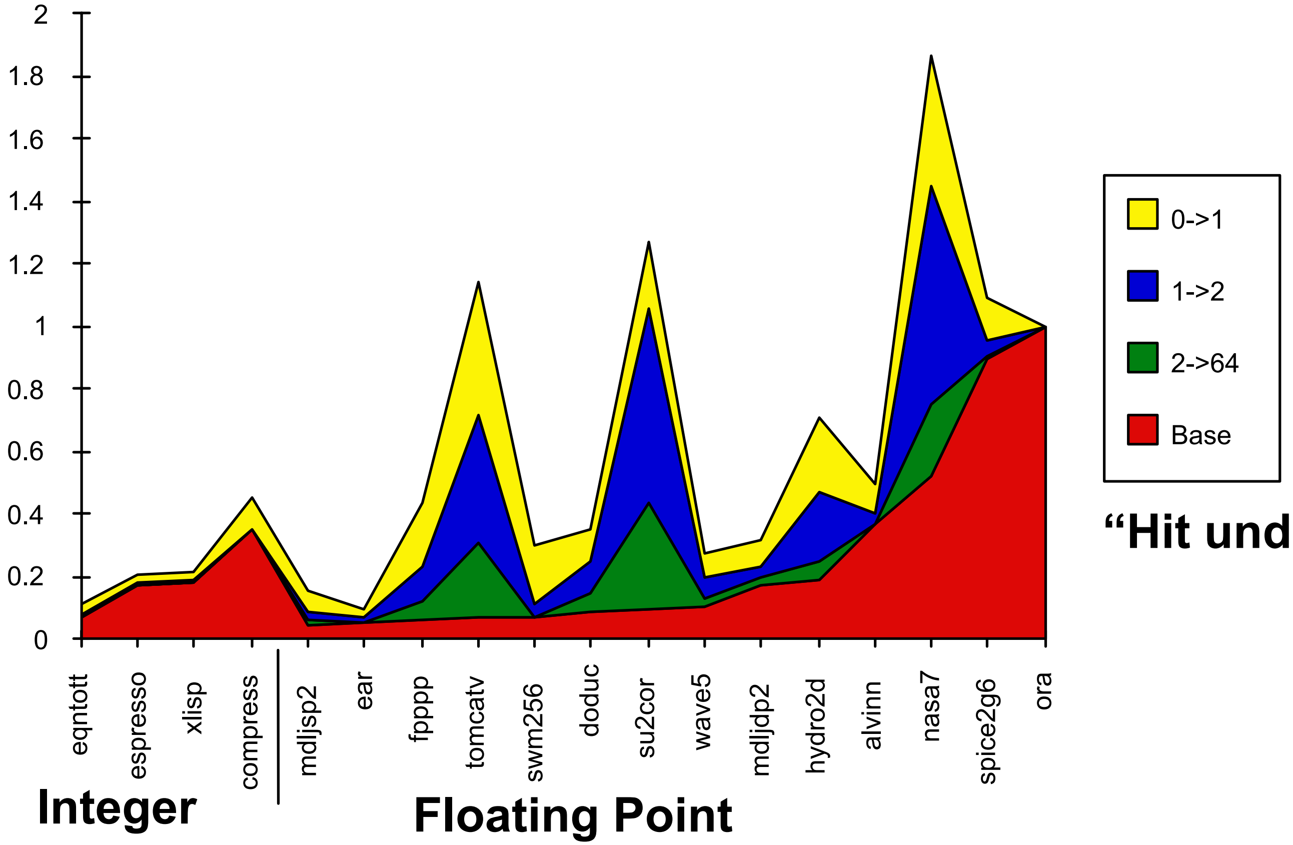
# What happens on a Cache miss?

- **For in-order pipeline, 2 options:**
  - Freeze pipeline in Mem stage (popular early on: Sparc, R4000)

```
IF   ID   EX   Mem stall stall stall … stall Mem   Wr
     IF   ID   EX   stall stall stall … stall stall Ex Wr
```

  - Use Full/Empty bits in registers + MSHR queue
    - MSHR = “Miss Status/Handler Registers” (Kroft)
      Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.
      - Per cache-line: keep info about memory address.
      - For each word: register (if any) that is waiting for result.
      - Used to “merge” multiple requests to one memory line
    - New load creates MSHR entry and sets destination register to “Empty”.  Load is “released” from pipeline.
    - Attempt to use register before result returns causes instruction to block in decode stage.
    - Limited “out-of-order” execution with respect to loads.
      Popular with in-order superscalar architectures.
- **Out-of-order pipelines already have this functionality built in… (load queues, etc).**

# Value of Hit Under Miss for SPEC

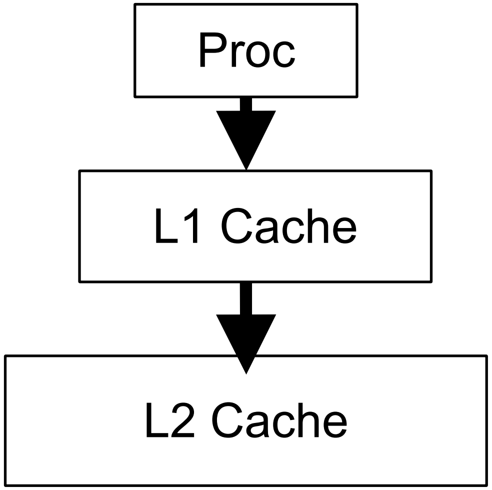


**0->1**
**1->2**
**2->64**
**Base**

**“Hit under n Misses”**

- **FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26**
- **Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19**
- **8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss**

# 4. Reduce Penalty: Second-Level Cache



- **L2 Equations**

**AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ × Miss Penalty$_{L1}$**

**Miss Penalty$_{L1}$ = Hit Time$_{L2}$ + Miss Rate$_{L2}$ × Miss Penalty$_{L2}$**

**AMAT = Hit Time$_{L1}$ +**

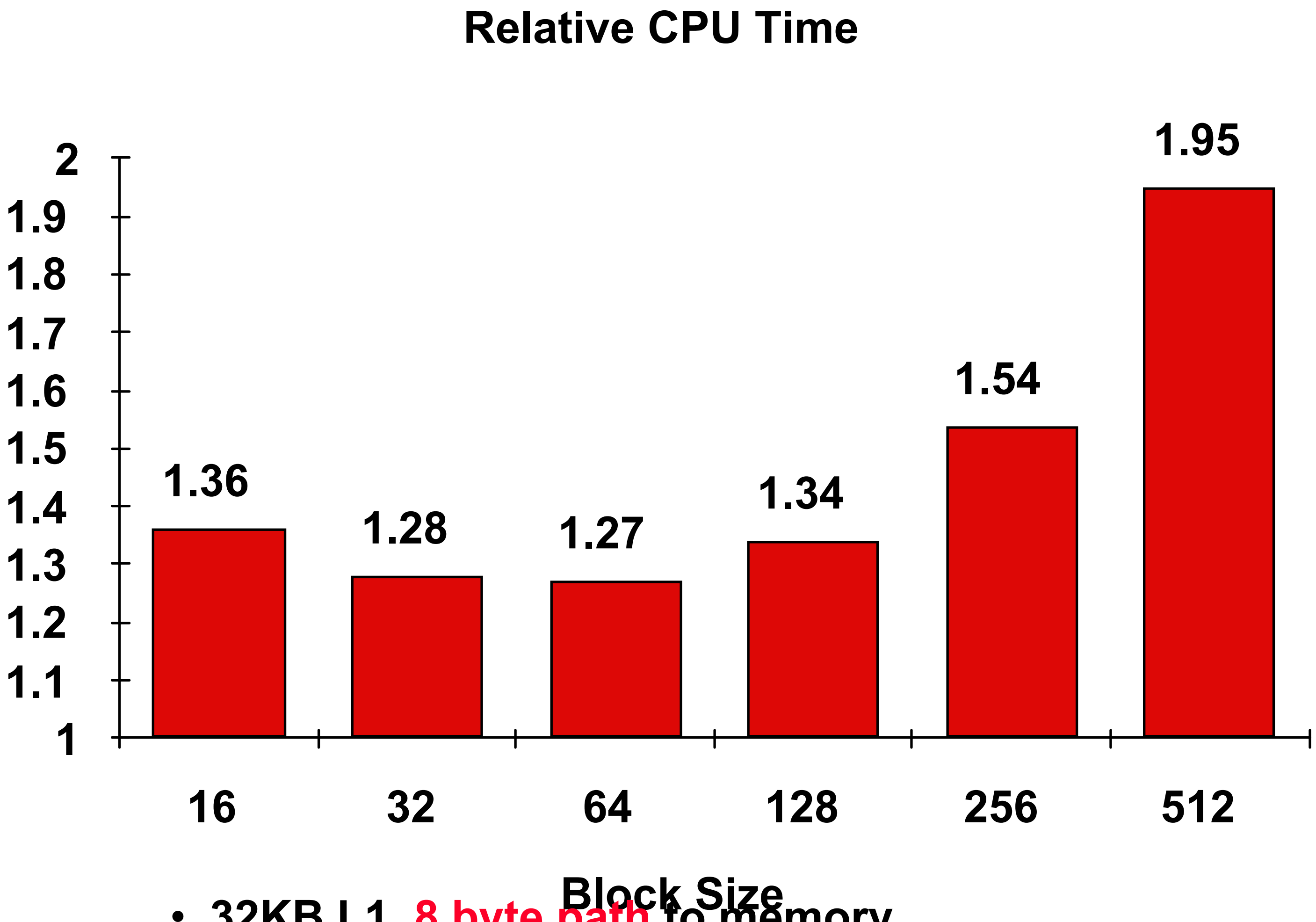
**Miss Rate$_{L1}$ × ( Hit Time$_{L2}$ + Miss Rate$_{L2}$ × Miss Penalty$_{L2}$ )**

- **Definitions:**
  - *Local miss rate*— misses in this cache divided by the total number of memory accesses *to this cache* ( Miss rate$_{L2}$ )
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses *generated by the CPU* ( Miss Rate$_{L1}$ × Miss Rate$_{L2}$ )
  - Global Miss Rate is what matters

# Reducing Misses: which apply to L2 Cache?

- **Reducing Miss Rate**
  1 Reduce Misses via Larger Block Size
  2 Reduce Conflict Misses via Higher Associativity
  3 Reducing Conflict Misses via Victim Cache
  4 Reducing Misses by HW Prefetching Instr, Data
  5 Reducing Misses by SW Prefetching Data
  6 Reducing Capacity/Conf. Misses by Compiler Optimizations

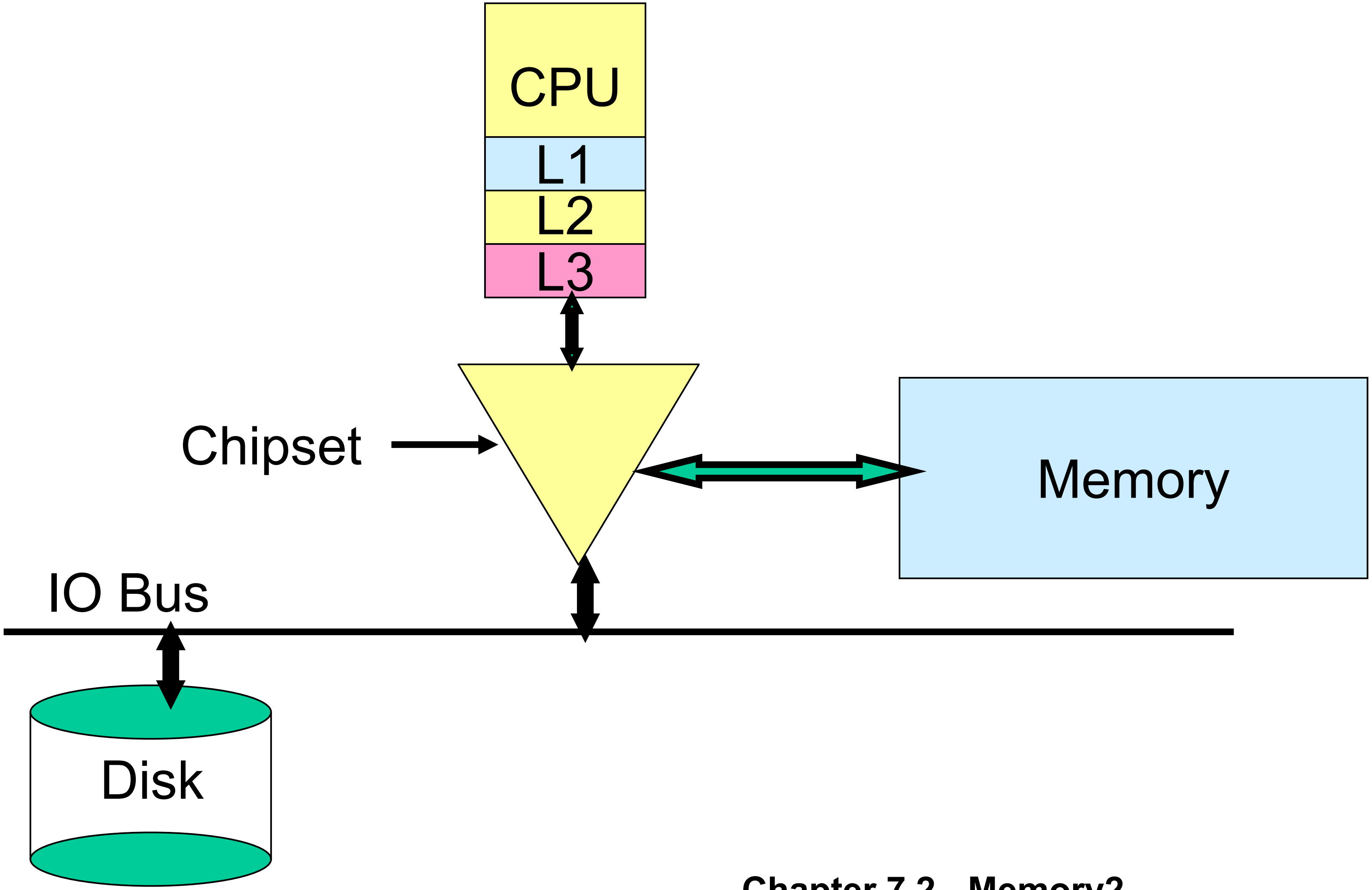
# L2 cache block size & A.M.A.T.



- 32KB L1, 8 byte path to memory

# Putting It All Together

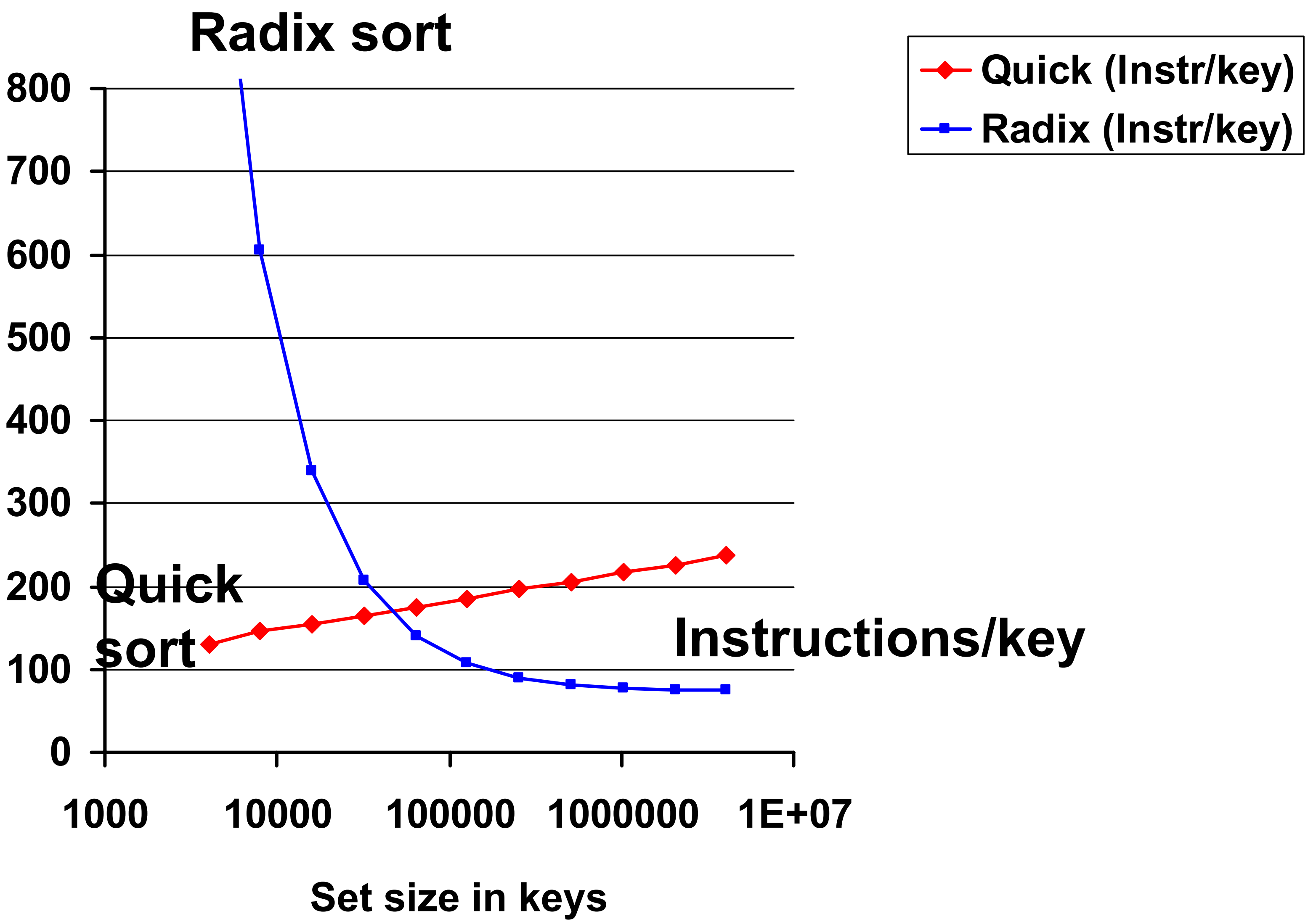
**What Have We Done This Semester?**



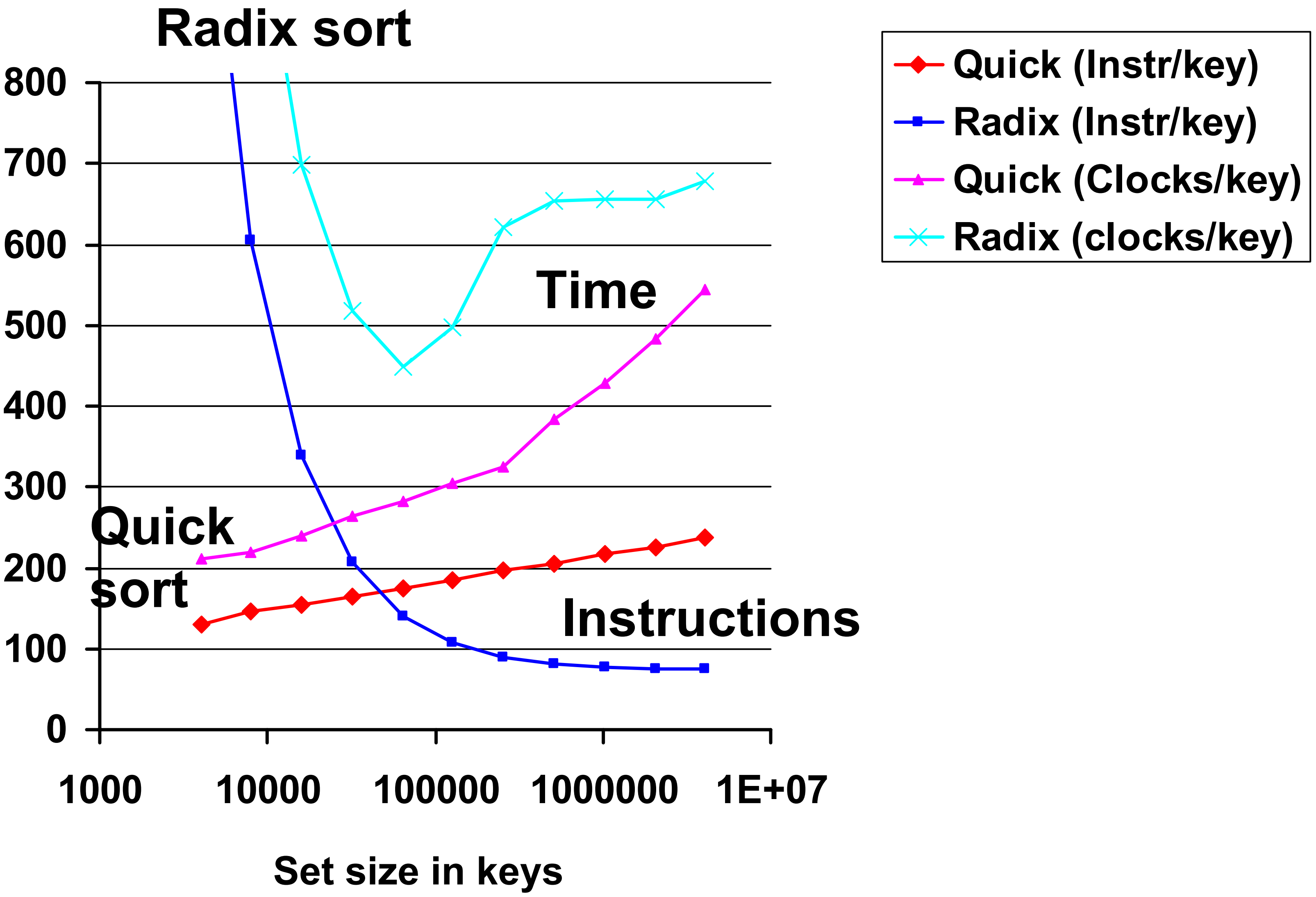
# Impact of Memory Hierarchy on Algorithms

- **Today CPU time is a function of (ops, cache misses) vs. just f(ops):**
  **What does this mean to Compilers, Data structures, Algorithms?**
- **“The Influence of Caches on the Performance of Sorting” by A. LaMarca and R.E. Ladner. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, January, 1997, 370-379.**
- **Quicksort: fastest comparison based sorting algorithm when all keys fit in memory**
- **Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys**
- **For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000**

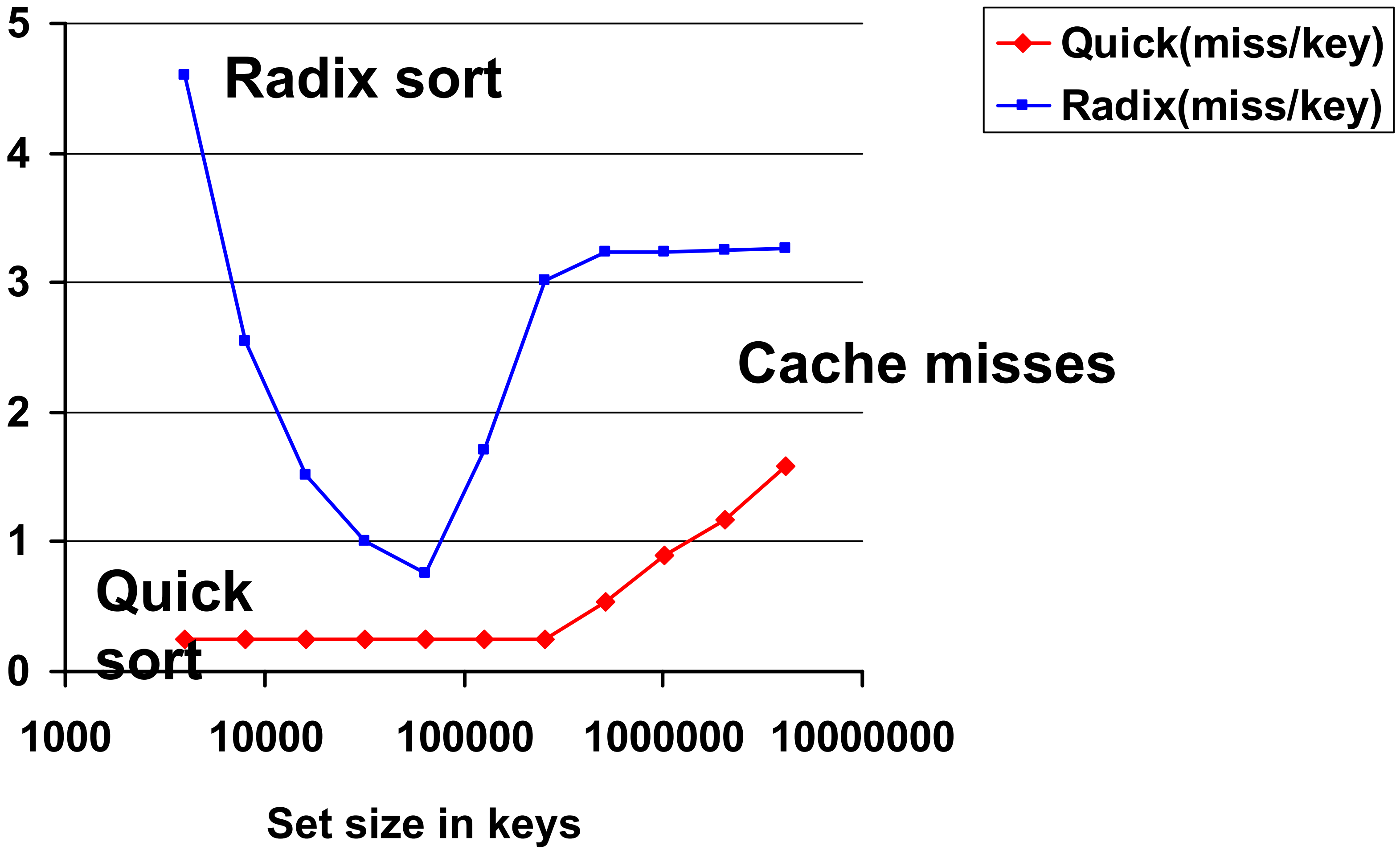
# Quicksort vs. Radix as vary number keys: Instructions



# Quicksort vs. Radix as vary number keys: Instrs & Time



# Quicksort vs. Radix as vary number keys: Cache misses



**What is proper approach to fast algorithms?**

# Recall: Levels of the Memory Hierarchy

*Capacity*
*Access Time*
*Cost*

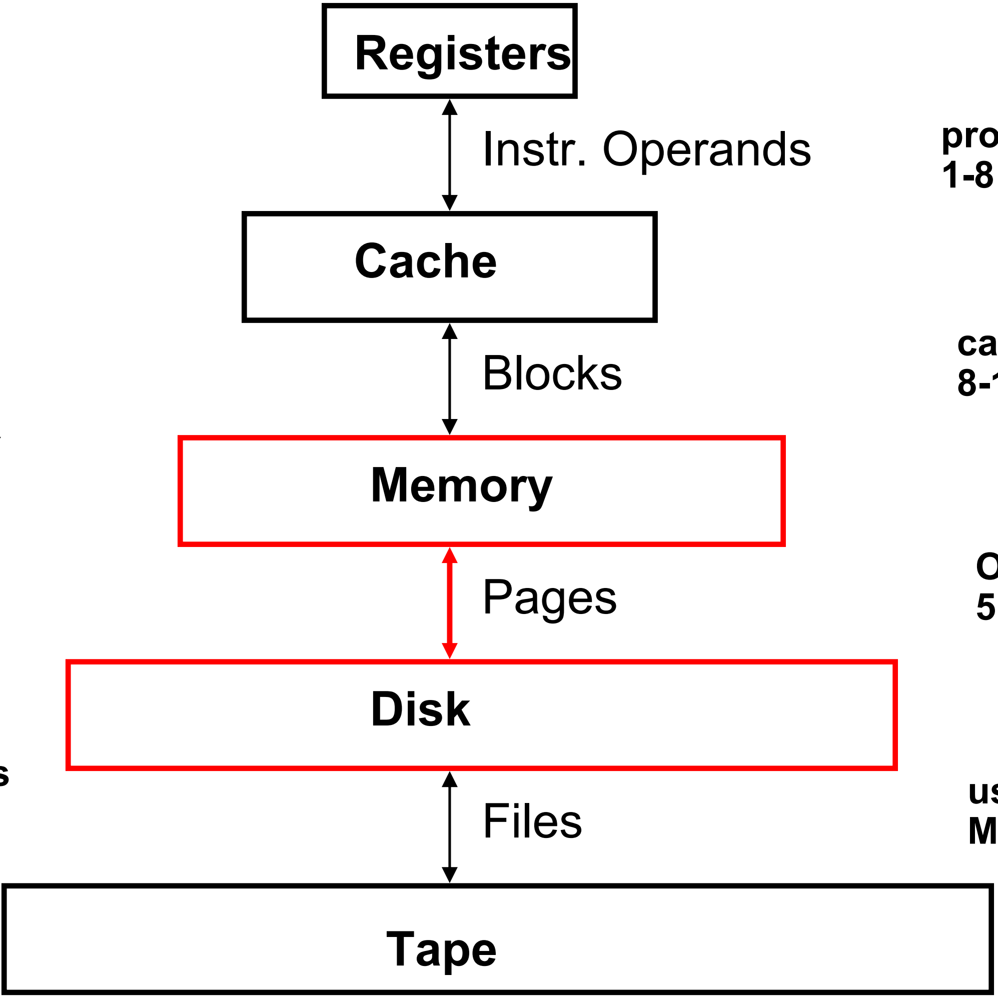
*CPU Registers*
**100s Bytes**
**<10s ns**

*Cache*
**K Bytes**
**10-100 ns**
**$.01-.001/bit**

*Main Memory*
**M Bytes**
**100ns-1us**
**$.01-.001**

*Disk*
**G Bytes**
**ms**
**$10^{-3}$ - $10^{-4}$ cents**

*Tape*
**infinite**
**sec-min**
**$10^{-6}$**



Upper Level

*Staging*
*Xfer Unit*

faster

**prog./compiler**
**1-8 bytes**

**cache cntl**
**8-128 bytes**

**OS**
**512-4K bytes**

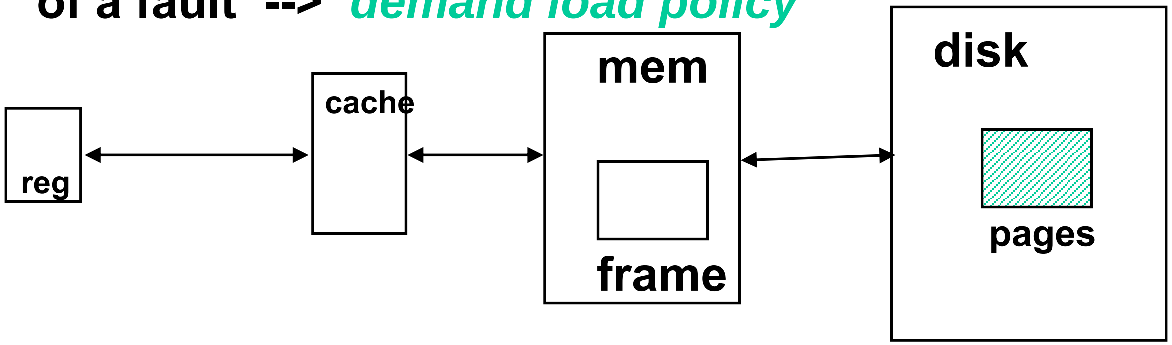
**user/operator**
**Mbytes**

Larger

Lower Level

# Basic Issues in Virtual Memory System Design

- **size of information blocks that are transferred from secondary to main storage (M)**
- **block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy***
- **which region of M is to hold the new block --> *placement policy***
- **missing item fetched from secondary memory only on the occurrence of a fault --> *demand load policy***



**<u>Paging Organization</u>**

**virtual and physical address space partitioned into blocks of equal size**

***page frames***

***pages***

# Address Map

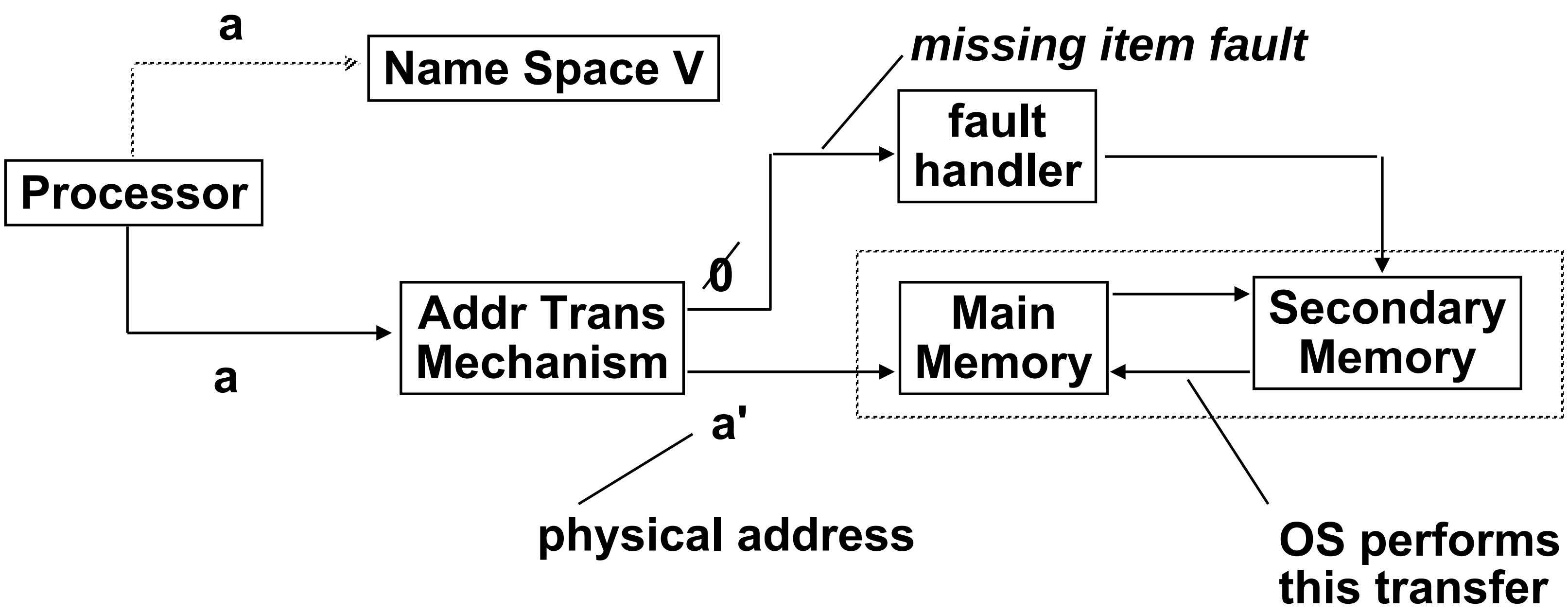
V = {0, 1, . . . , n - 1}   virtual address space
M = {0, 1, . . . , m - 1}  physical address space

n > m

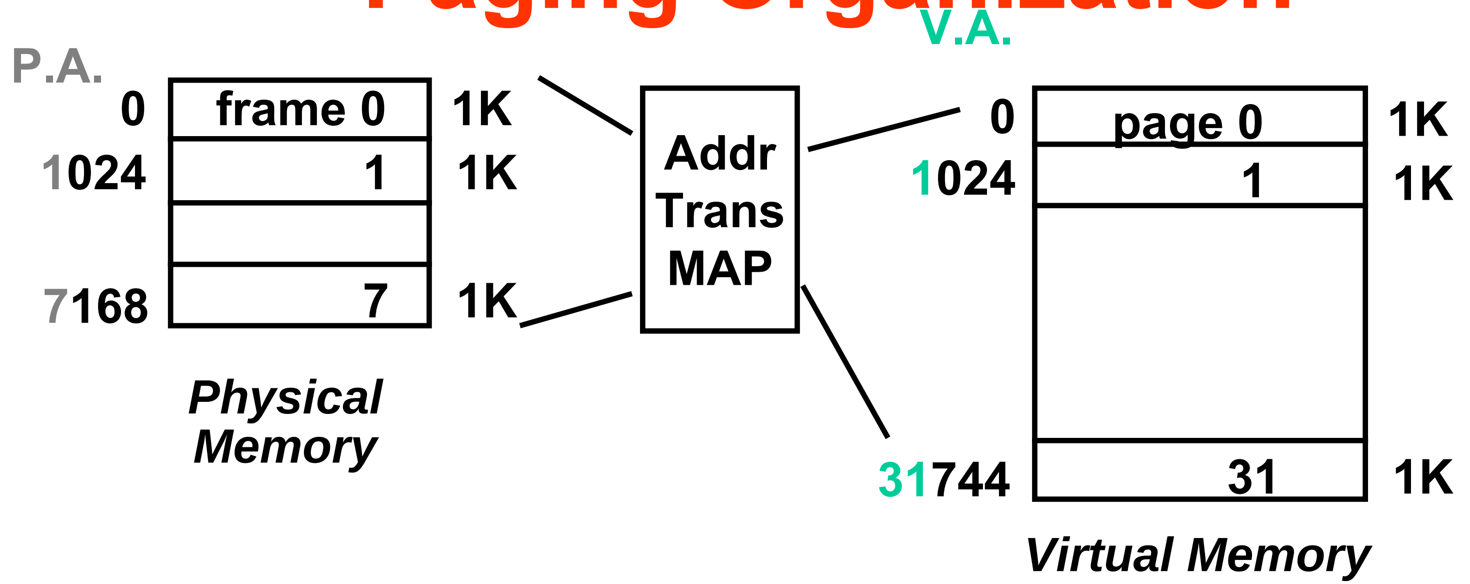
MAP:  V --> M  U  {$\emptyset$}  address mapping function

MAP(a)  =  a'  if data at virtual address a is present in physical address a'  and  a' in M

= $\emptyset$  if data at virtual address a is not present in M



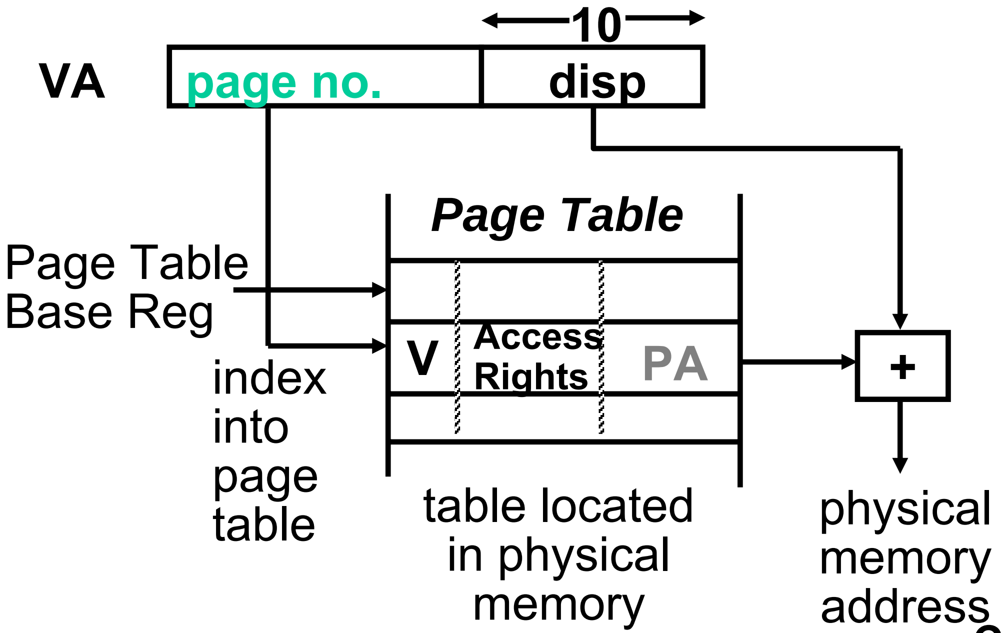
# Paging Organization



unit of mapping

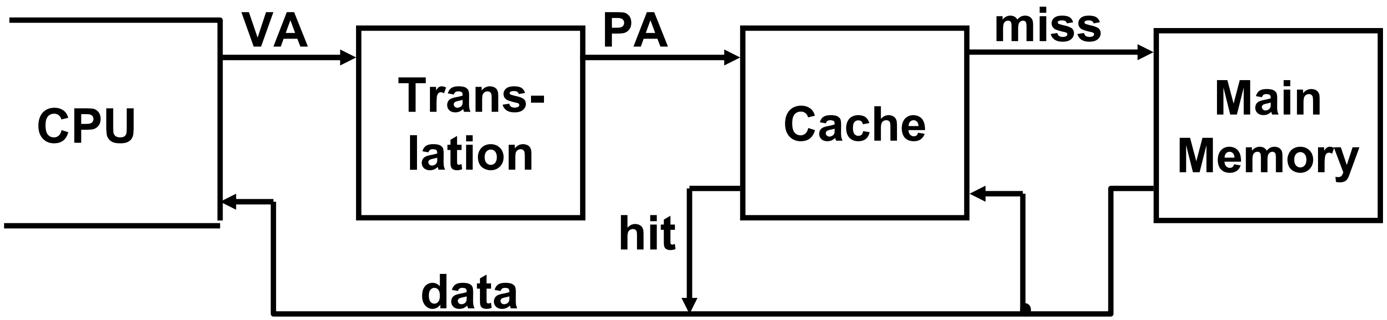
also unit of transfer from virtual to physical memory

## Address Mapping



actually, concatenation is more likely

# Virtual Address and a Cache



It takes an extra  memory access to translate VA to PA

This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible

ASIDE:  Why access cache with PA at all?  VA caches have a problem!
*synonym  / alias problem*:  two different virtual addresses map to same physical address  =>  two different cache entries holding data for the same physical address!

for update:  must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits; or

software enforced alias boundary: same lsb of VA &PA > cache size

# TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer or TLB*

| Virtual Address | Physical Address | Dirty | Ref | Valid | Access |
|---|---|---|---|---|---|
| | | | | | |

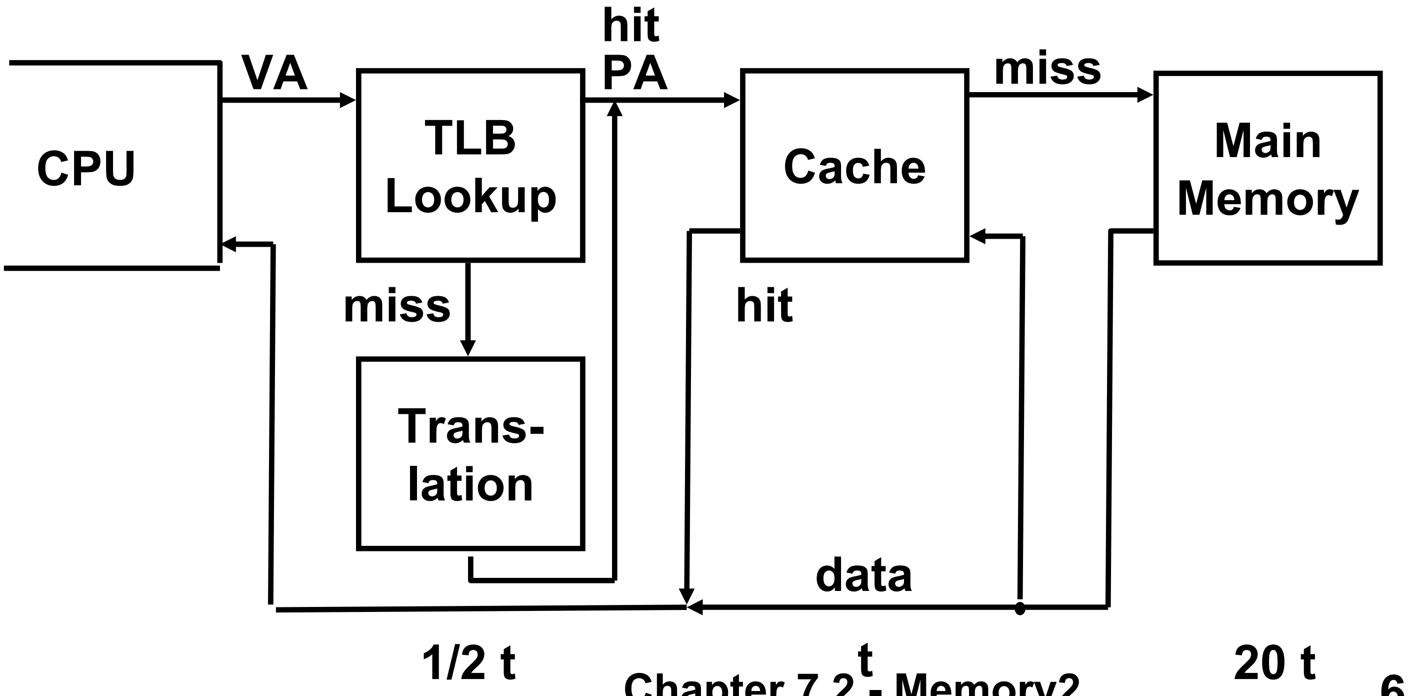
TLB access time comparable to cache access time (much less than main memory access time)

# Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.

*Translation with a TLB*



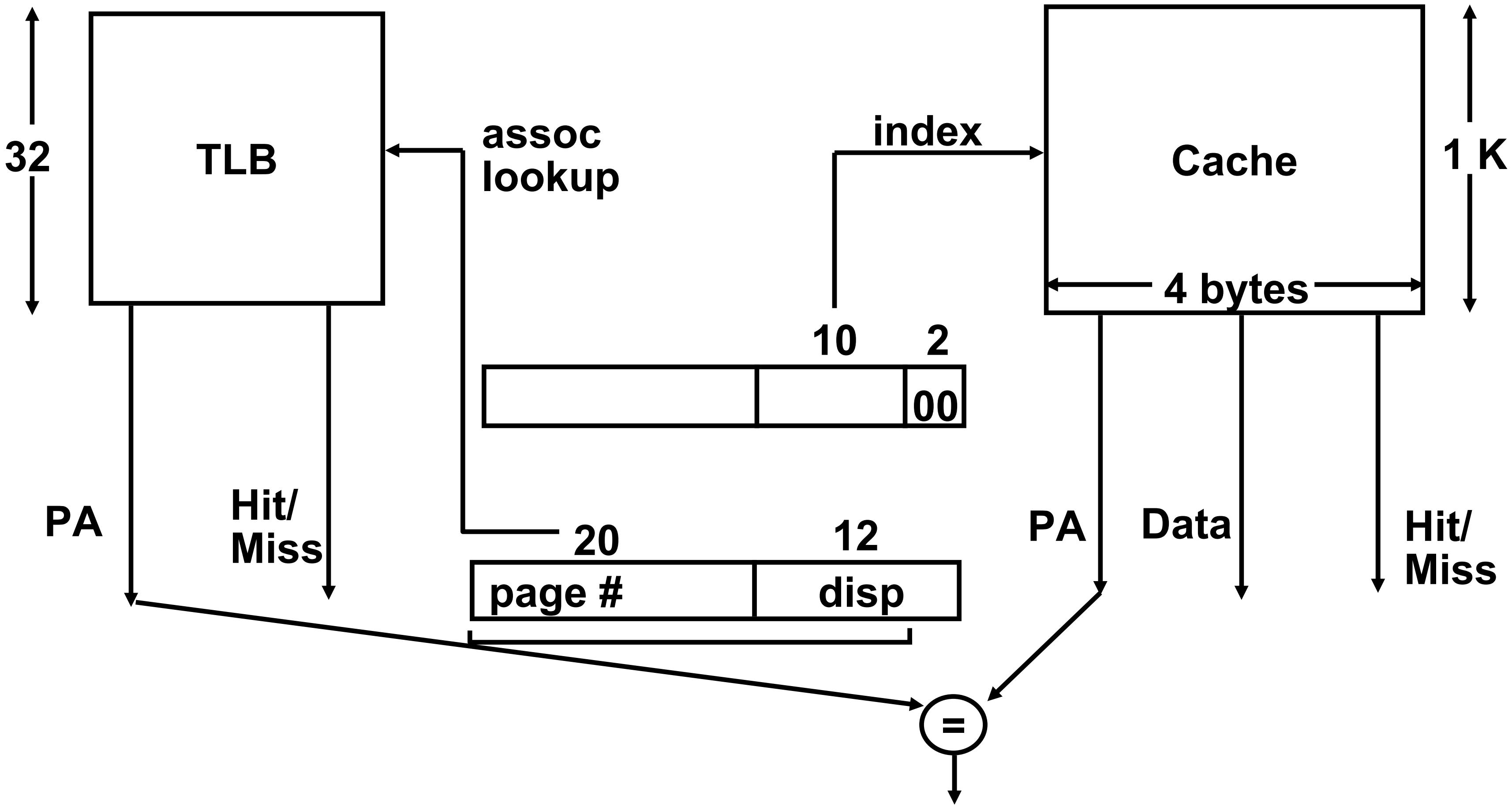
# Reducing Translation Time

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access

Works because high order bits of the VA are used to look in the TLB
while low order bits are used as index into cache

# Overlapped Cache & TLB Access



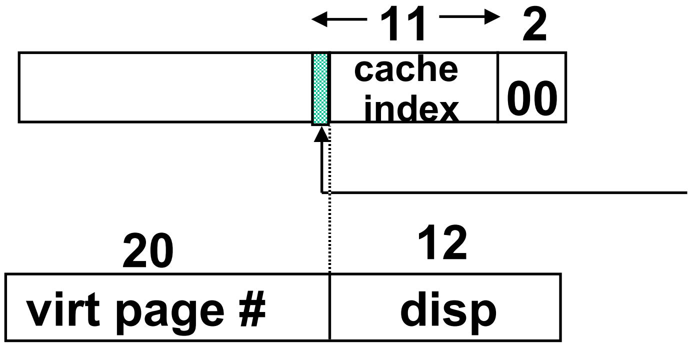
IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = PA)] and TLB hit THEN
    access memory with the PA from the TLB
ELSE do standard VA translation

# Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

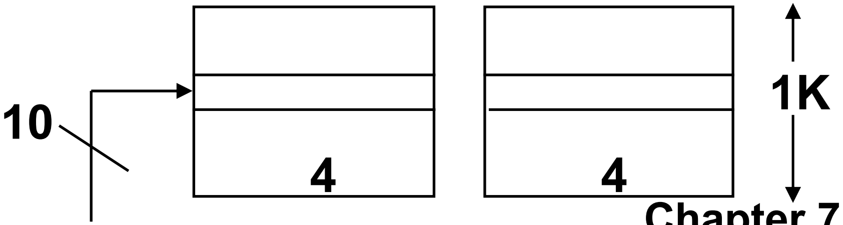
Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:



This bit is changed by VA translation, but is needed for cache lookup

Solutions:

go to 8K byte page sizes;
go to 2 way set associative cache; or
SW guarantee VA[13]=PA[13]



2 way set assoc cache

# Summary #1/ 4:

- **The Principle of Locality:**
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- **Three Major Categories of Cache Misses:**
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity.
    Nightmare Scenario: ping pong effect!
  - Capacity Misses: increase cache size
- **Cache Design Space**
  - total size, block size, associativity
  - replacement policy
  - write-hit policy (write-through, write-back)
  - write-miss policy