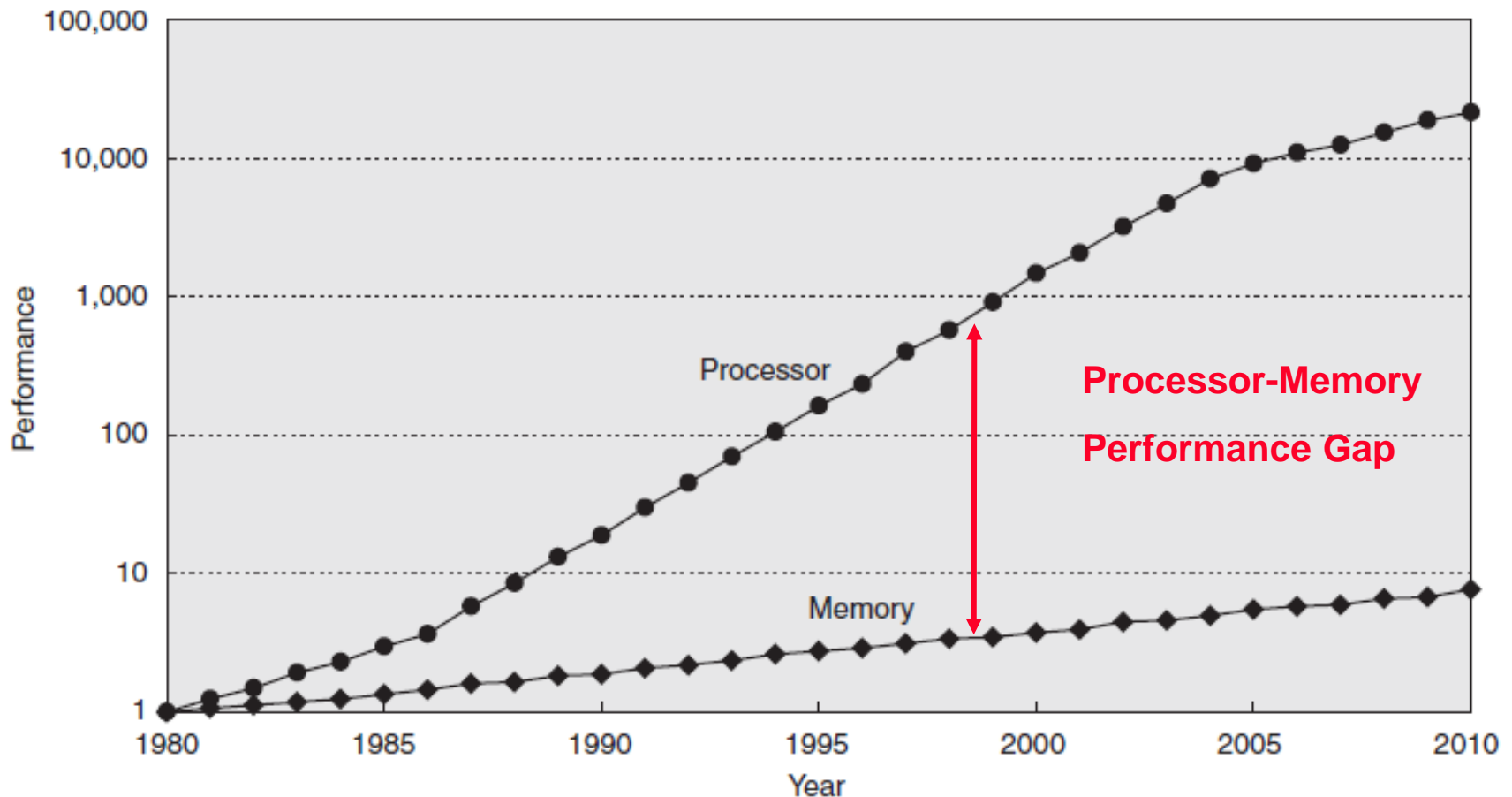


Memory Hierarchy

Why Memory Hierarchy?



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

- The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency

- The processor line assumes a 1.25 improvement per year until 1986, and a 1.52 improvement until 2004, and a 1.20 improvement thereafter

Levels of the Memory Hierarchy

Capacity
Access Time
Cost

Staging
Xfer Unit

CPU Registers
100s Bytes
0.3-0.5 ns

Registers

Instr. Operands

prog./compiler
1-8 bytes

Upper Level

L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
\$1000s/ GByte

L1 Cache

Blocks

cache cntl
32-64 bytes

faster

L2 Cache

Blocks

cache cntl
64-128 bytes

Main Memory
G Bytes
80ns- 200ns
~ \$100/ GByte

Memory

Pages

OS
4K-8K bytes

dynamic random access memory

Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ \$1 / GByte

Disk

Files

user/operator
Mbytes

Larger

Tape
infinite
sec-min
~\$1 / GByte

Tape

Lower Level

Memory Hierarchy: Apple iMac G5

Managed
by compiler

Managed
by hardware

Managed by OS,
hardware,
application



iMac G5
1.6 GHz

07	Reg	L1 Inst	L1 Data	L2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency Cycles, Time	1, 0.6 ns	3, 1.9 ns	3, 1.9 ns	11, 6.9 ns	88, 55 ns	10 ⁷ , 12 ms

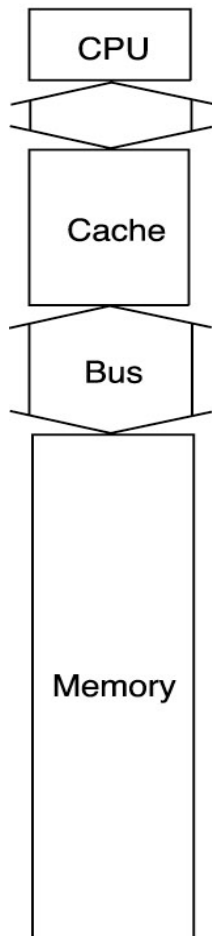
Goal: Illusion of large, fast, cheap memory

Let programs address a memory space that scales
to the disk size, at a speed that is usually as fast as
register access

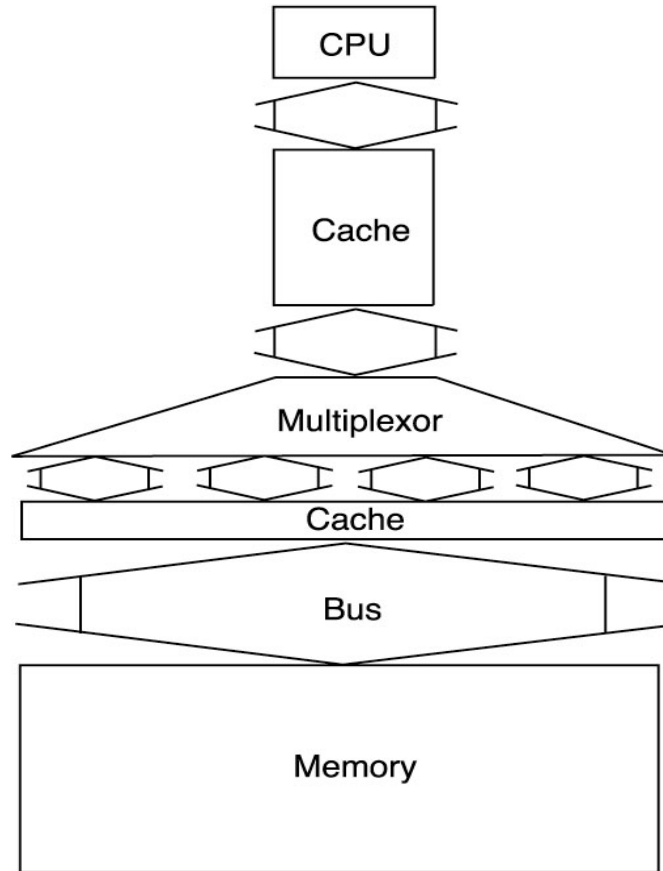


Achieving higher memory bandwidth

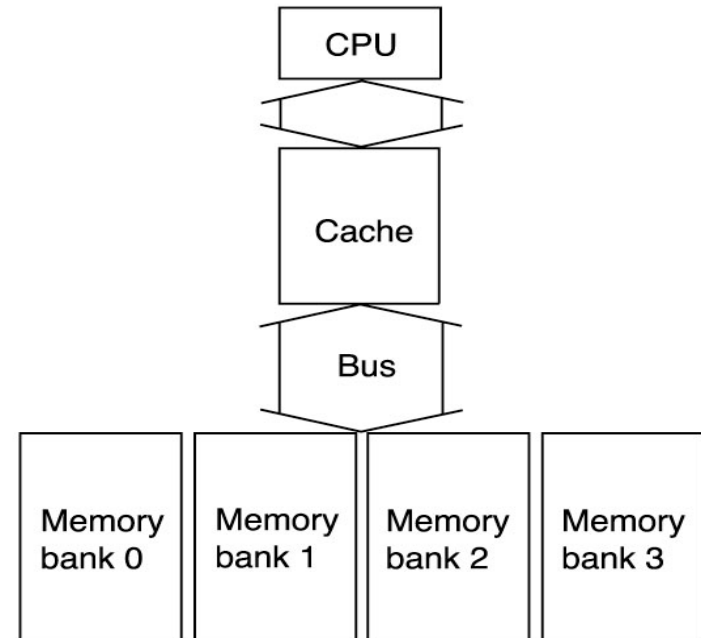
(a) One-word-wide memory organization



(b) Wide memory organization



(c) Interleaved memory organization



Achieving higher memory bandwidth

- Suppose

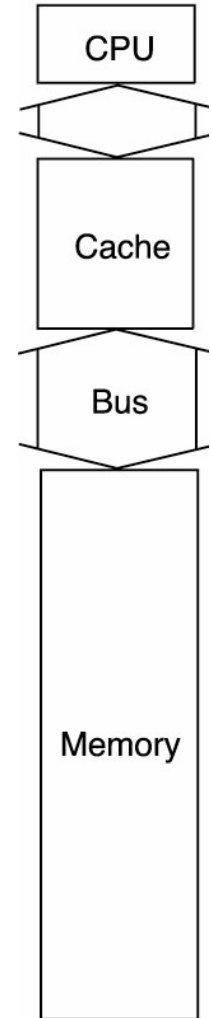
- 1 clock cycle to send the address
- 15 clock cycles for each DRAM access initiated
- 1 clock cycle to send a word of data

- If we have a cache block of four words and a one-word-wide bank of DRAMs, the miss penalty would be

$$1 + 4 \times 15 + 4 \times 1 = 65$$

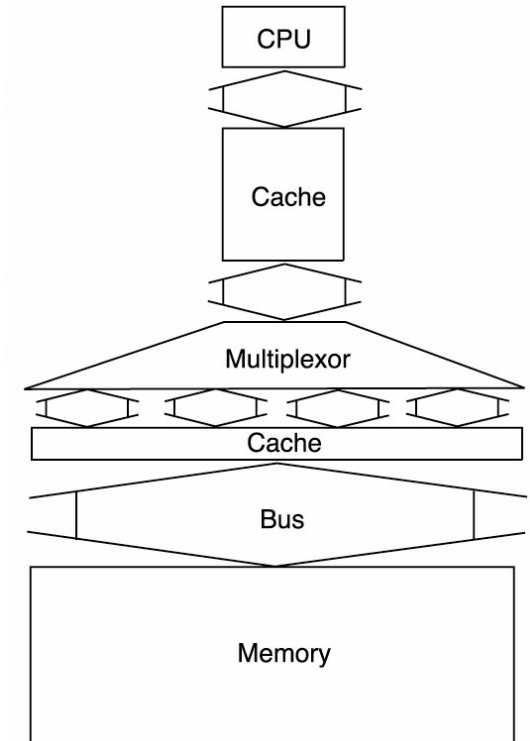
- Number of bytes transferred per clock cycle for a single miss is $4 \text{ (words)} \times 4 \text{ (bytes/word)} / 65 = 0.246$

- The memory is one-word-wide and all the accesses are made sequentially.



Achieving higher memory bandwidth

- Suppose
 - 1 clock cycle to **send the address**
 - 15 clock cycles for each **DRAM access initiated**
 - 1 clock cycle to **send a word of data**
- Increase the bandwidth to memory by **widening the memory and the buses** between the processor and memory
- Allow **parallel access** to all the words of the block
- A four-word-wide memory, the miss penalty is $1 + (4/4) \times 15 + (4/4) \times 1 = 17$ clock cycles
- A two-word-wide memory, the miss penalty is $1 + (4/2) \times 15 + (4/2) \times 1 = 33$ clock cycles



Achieving higher memory bandwidth

- Suppose

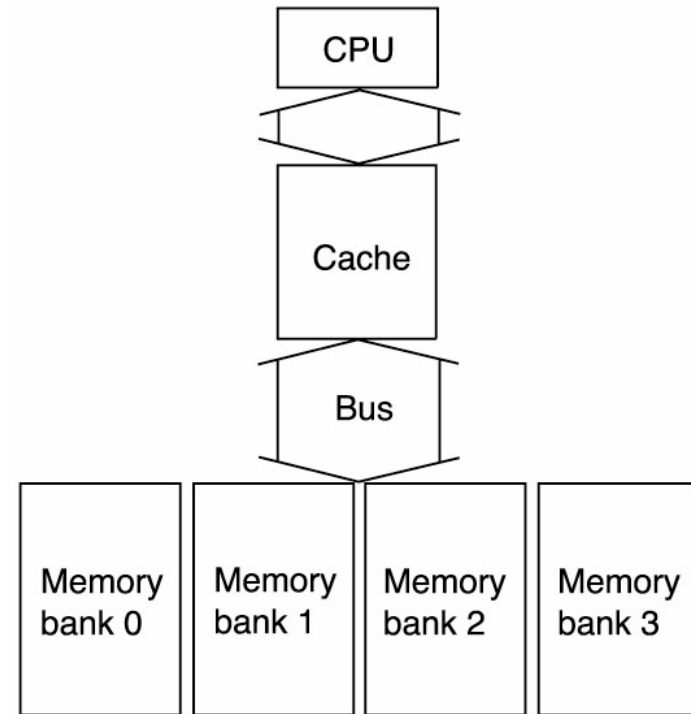
- 1 clock cycle to **send the address**
- 15 clock cycles for each **DRAM access initiated**
- 1 clock cycle to **send a word of data**

- Increase the bandwidth to memory by **widening the memory but not the interconnection bus**

- **Still need to pay a cost to transmit each word, but we can avoid paying the cost of access latency more than once**

- **Interleaving:** memory chips organized in banks read/write multiple words in one access time instead of reading/writing single words each time

$$1 + 1 \times 15 + 4 \times 1 = 20 \text{ clock cycles}$$



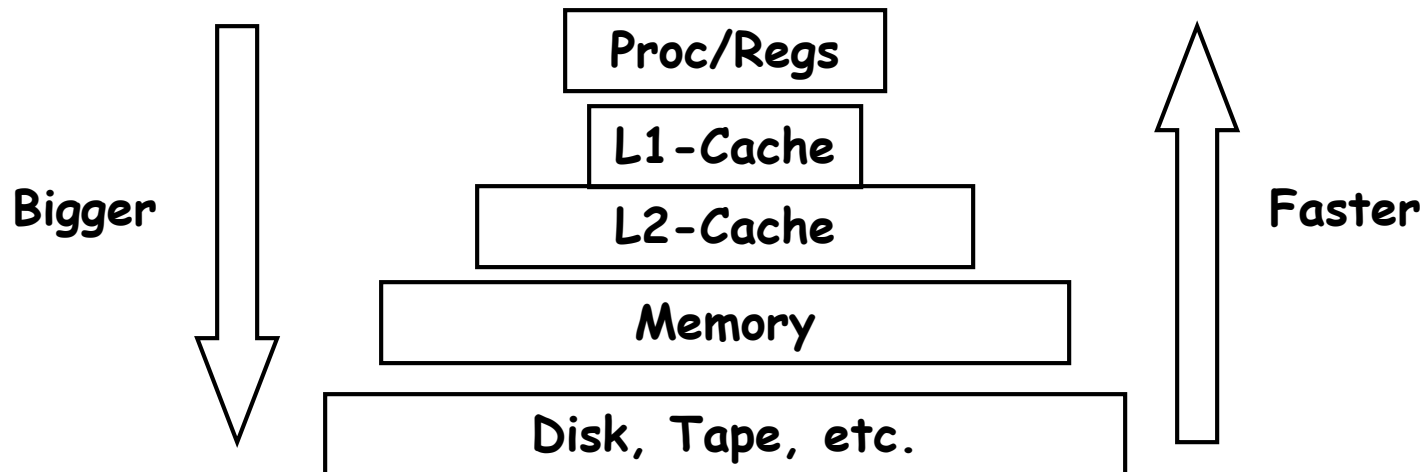
Caches

The Principle of Locality

- **The Principle of Locality:**
 - Program access a relatively small portion of the address space at any instant of time.
- **Two Different Types of Locality:**
 - **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- **Last 15 years, HW relied on locality for speed**

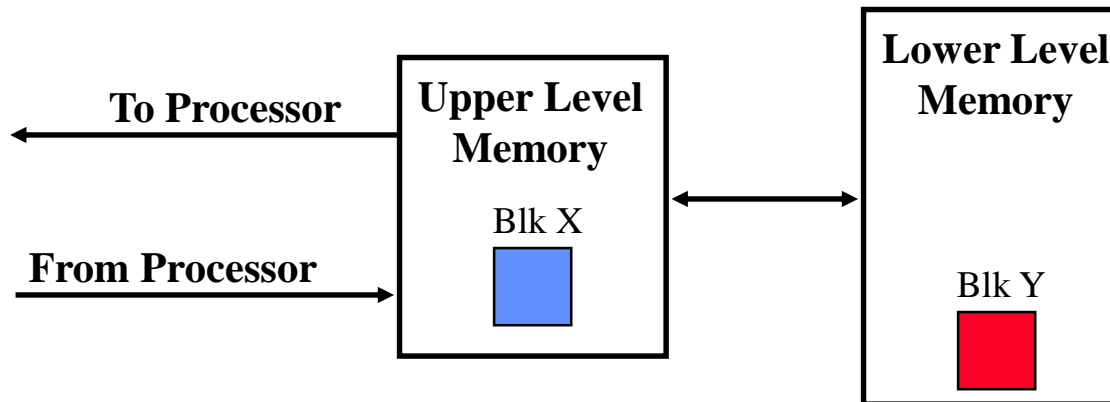
What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- Exploits spacial and temporal locality
- In computer architecture, almost everything is a cache!
 - Registers: a cache on variables
 - First-level cache: a cache on second-level cache
 - Second-level cache: a cache on memory
 - Memory: a cache on disk (virtual memory)
 - TLB: a cache on page table
 - Branch-prediction: a cache on prediction information
 - BTB: a cache of branch target



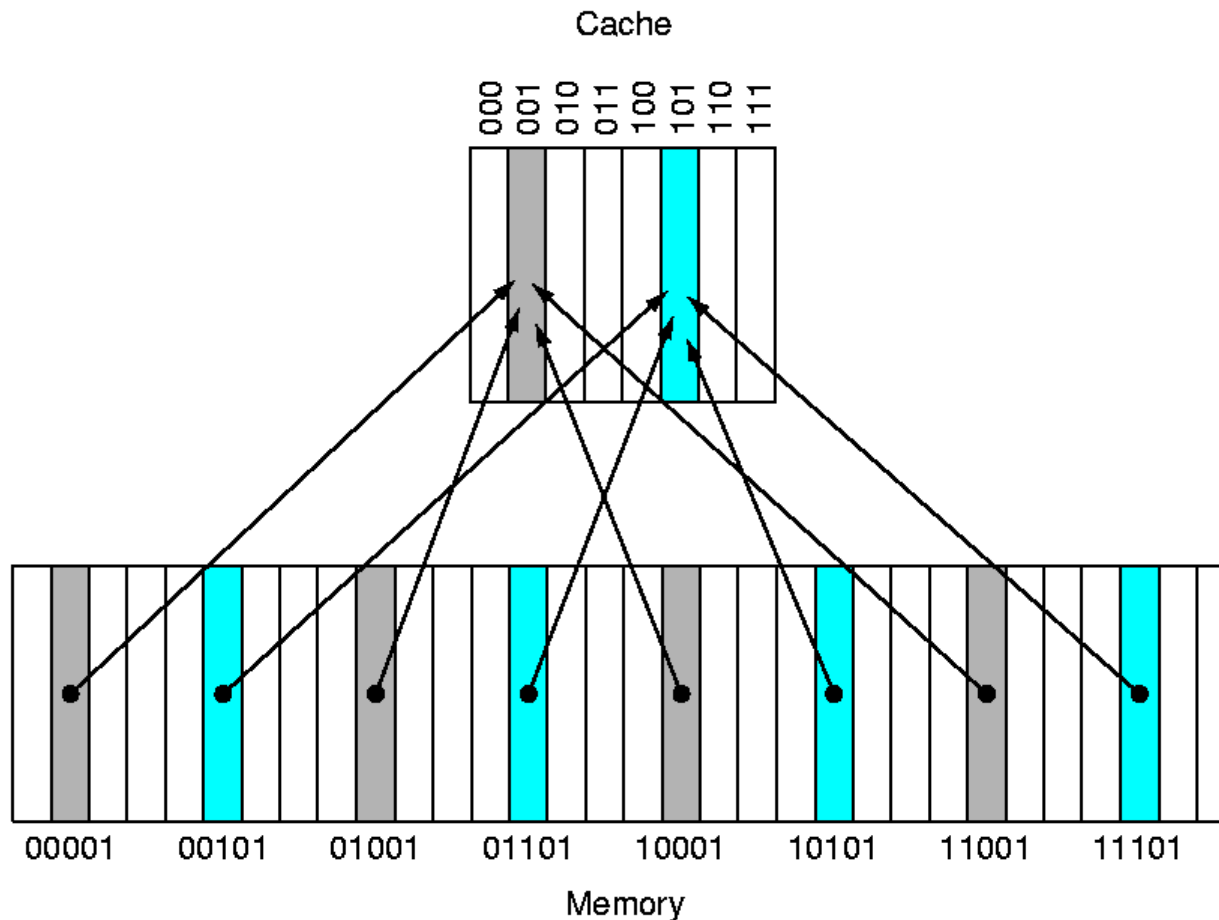
Review: Terminology

- **Hit**: data appears in some block in this level
 - **Hit Rate**: the fraction of memory access found in this level
 - **Hit Time**: Time to access the upper level which consists of
RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
 - **Miss Rate** = $1 - (\text{Hit Rate})$
 - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block to the processor
- **Hit Time** << **Miss Penalty**

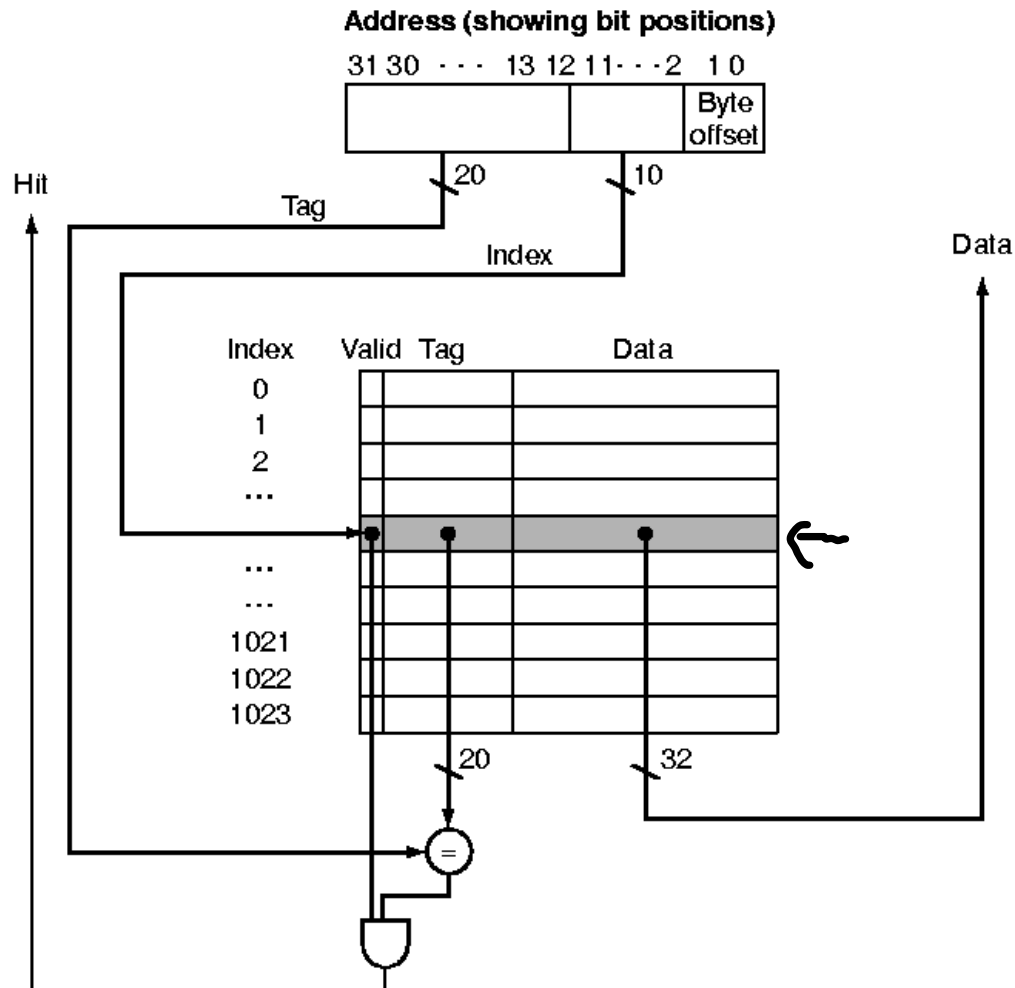


Review: The Basics of Caches

A direct-mapped cache with eight entries



A 4KB direct map cache using one word block



4 Questions for Memory Hierarchy

- 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)

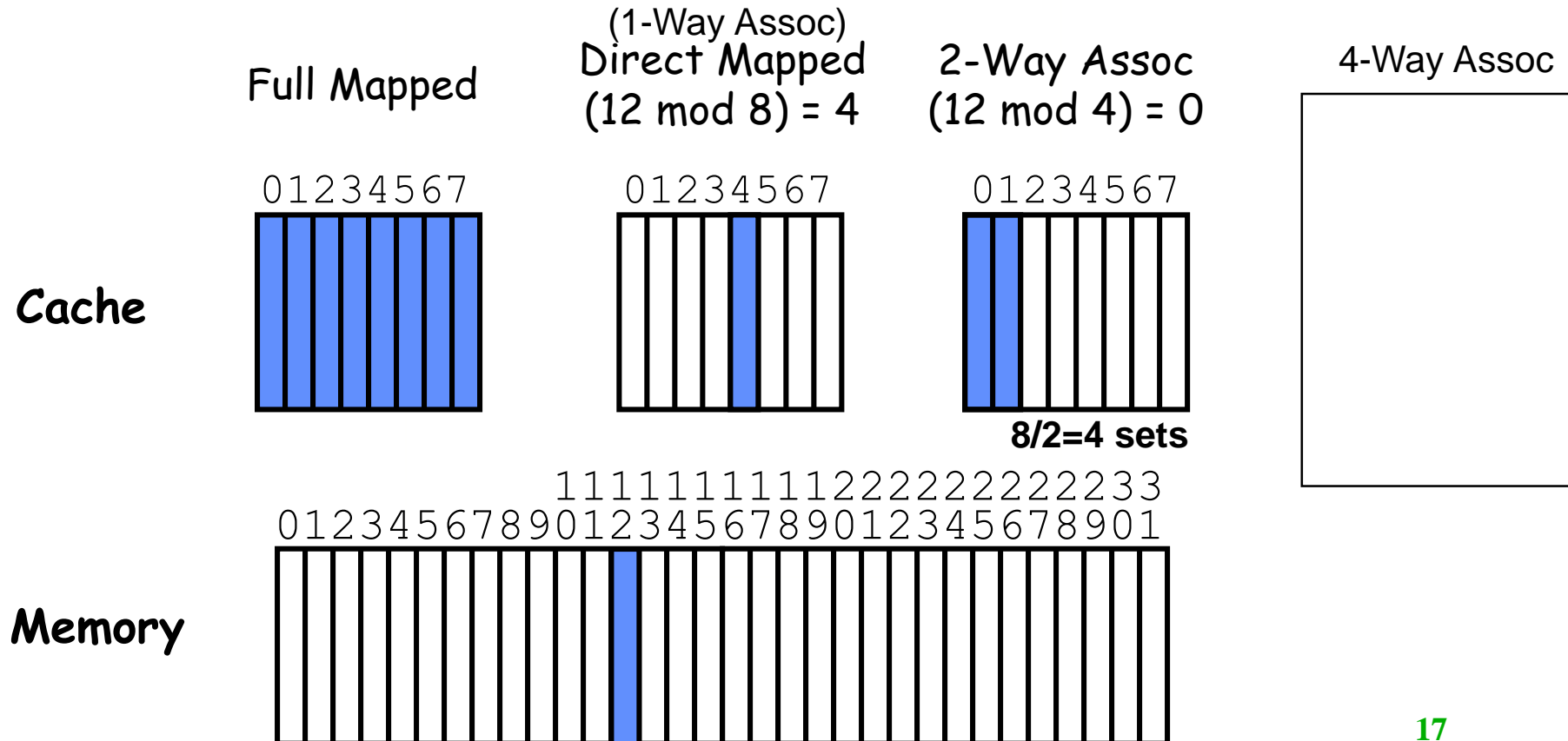
Block Placement

- **Q1: Where can a block be placed in the upper level?**
 - **Fully Associative,**
 - **Set Associative,**
 - **Direct Mapped**

(Block address) MOD (Number of Sets in Cache)

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

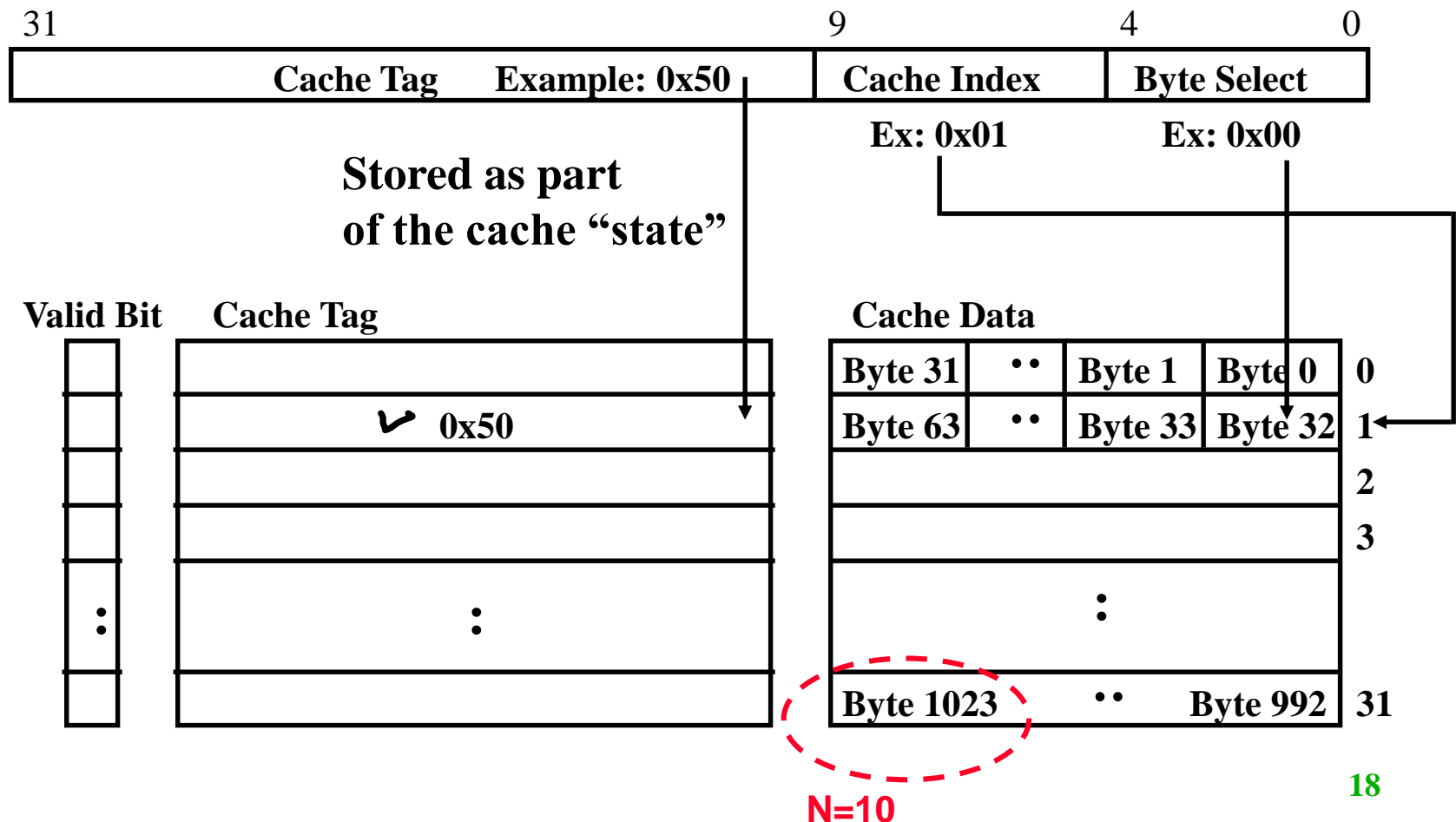


Direct Mapped Cache: 1 KB , 32B blocks

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

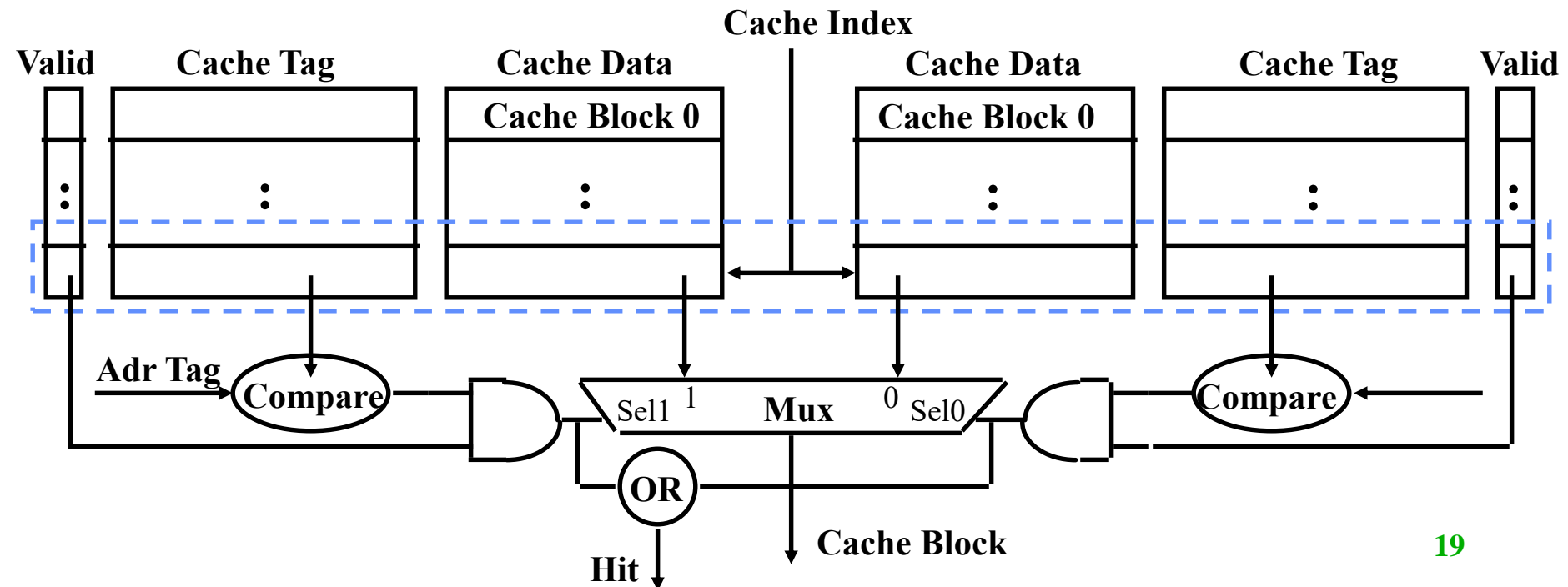
- For a 2^N byte cache:

- The uppermost **(32-N)** bits are always the **Cache Tag**
- The lowest **M** bits are the **Byte Select (Block Size = 2^M)**



Set Associative Cache

- **N-way set associative:** N entries for each Cache Index
 - N direct mapped caches operates in parallel
- **Example: Two-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

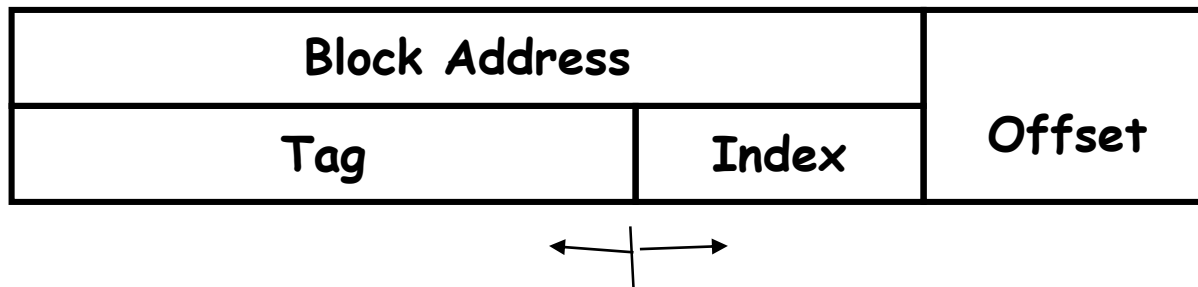


4 Questions for Memory Hierarchy

- 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)

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

- Index identifies set
- Increasing associativity shrinks index, expands tag



$$\text{Cache size} = \text{Associativity} \times 2^{\text{index_size}} \times 2^{\text{offset_size}}$$

Exercise

- A 512KB two way set associative cache
- Each block is 64B
- The length of the address is 32 bits.
- How many bits are used for tag, index, and byte offset, respectively?

4 Questions for Memory Hierarchy

- 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)

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
 - Random
 - LRU (Least Recently Used)

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
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%

Q3: After a cache read miss, if there are no empty cache blocks, which block should be removed from the cache?

The Least Recently Used (LRU) block? Appealing, but hard to implement for high associativity

A randomly chosen block? Easy to implement, how well does it work?

Miss Rate for 2-way Set Associative Cache

Size	Random	LRU
16 KB	5.7%	5.2%
64 KB	2.0%	1.9%
256 KB	1.17%	1.15%

Also, try other LRU approx.

4 Questions for Memory Hierarchy

- 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)

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 repeated writes to same location
- **What about miss on a write?**
 - **write_allocate** or not

Q4: What happens on a write?

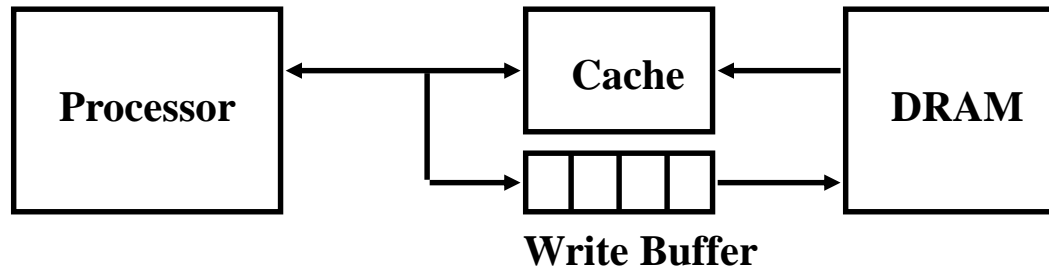
	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Additional option -- let writes to an un-cached address allocate a new cache line (“**write-allocate**”).

- **Write allocate (fetch on write):** The block is loaded on a write miss
 - Usually, **write-back cache** use **write allocate** (hoping subsequent writes to that block will be captured by the cache)
- **No write allocate (write around):** The block is modified in the lower level and not loaded into the cache
 - **Write through cache** often use **no-write allocate** (since subsequent writes to that block will still have to go to memory)

Write Buffer for Write Through

WT always combined with **write buffers** so that we don't have to wait for lower level memory



- **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 \ll (1 / DRAM write cycle)

Cache performance

- **Miss-oriented Approach to Memory Access:**

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

- **Separating out Memory component entirely**

- AMAT = Average Memory Access Time

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

$$AMAT = HitTime + MissRate \times MissPenalty$$

$$= (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) + \\ (HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data})$$

Impact on Performance

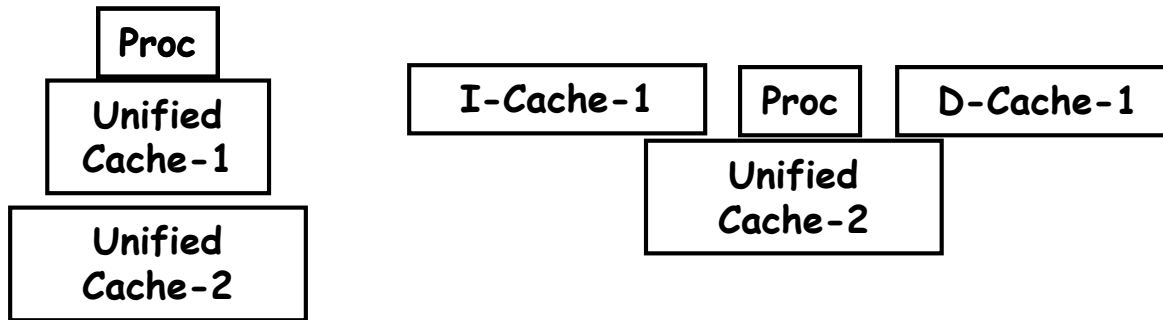
- Suppose a processor executes at
 - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 2.1
 - 50% arith/logic, 30% ld/st, 20% control
- Suppose that
 - 10% of memory ops get 50 cycle miss penalty
 - 1% of instructions get 50 cycle miss penalty
- $\text{CPI} = \text{ideal CPI} + \text{avg. stalls per instruction}$
 $= 2.1(\text{cycles/ins})$
 $+ 0.30 (\text{DataMem/ins}) \times 0.10 (\text{miss/DataMem}) \times 50(\text{cycle/miss})$
 $+ 1 (\text{InstMem/ins}) \times 0.01 (\text{miss/InstMem}) \times 50 (\text{cycle/miss})$
 $= (2.1 + 1.5 + .5) \text{ cycle/ins} = 4.1$
- Average Memory Access Clock Cycles=
 $(1/1.3) \times [1 + 0.01 \times 50] + (0.3/1.3) \times [1 + 0.1 \times 50] = 2.54$

1.3 memory references per instruction

Assume hit time: 1 clock cycle

Example: Harvard Architecture

- **Unified vs Separate** Instruction & Data (Harvard)



- **Statistics :**
 - 16KB Instruction & Data: Inst miss rate= **0.64%** , Data miss rate= **6.47%**
 - 32KB unified: Aggregate miss rate= **1.99%**
- **Which is better (ignore L2 cache)?**
 - Assume 33% data ops \Rightarrow **75%** accesses from instructions (1.0/1.33)
 - **hit time=1 cycle, miss penalty=50 cycles**
 - Note that *data* hit has **1** stall for unified cache (**only one port**)

$$AMAT_{\text{Harvard}} = 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) = 2.05$$

$$AMAT_{\text{Unified}} = 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) = 2.24$$

Impact of Cache

Example:

- Miss penalty 50 clock cycle
- All instructions normally take 2.0 clock cycles (ignoring memory stalls)
- Miss rate 2%
- Average 1.33 memory reference per instruction

CPU time with cache = $IC \cdot (2.0 + 1.33 \cdot 2\% \cdot 50) \cdot \text{clock cycle time} = 3.33 \cdot IC \cdot CC$

CPU time without cache = $IC \cdot (2.0 + 1.33 \cdot 50) \cdot \text{clock cycle time} = 68.5 \cdot IC \cdot CC$

- The **lower the $CPI_{\text{execution}}$** , the **higher the relative impact** of a fixed number of **cache miss clock cycles**
- For CPU with **higher clock rates**, a larger number of clock **cycles per miss** and hence **the memory portion of CPI** is higher

Performance Examples

- CPI with perfect cache is 2.0
- 1.3 memory references per instruction
- Size of both caches is 64KB, block size 32 bytes
- Cache 1: direct map
 - Clock cycle time: 2ns
 - Miss rate: 1.4%
- Cache 2: 2-way set associative
 - Clock cycle time: 2ns*1.1
 - Miss rate: 1%
- Cache miss penalty: 70ns
- Assume hit time: 1 clock cycle

$$AMAT = HitTime + MissRate \times MissPenalty$$

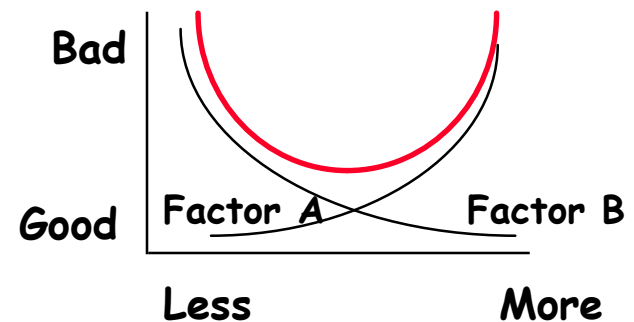
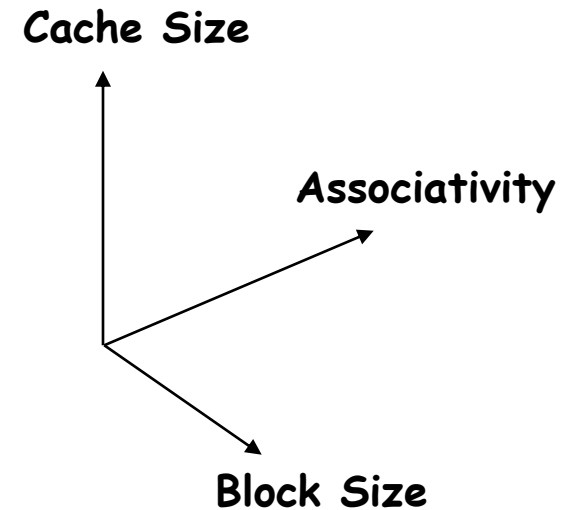
Calculate the **Memory Access time** and then the **Performance (CPU time)**

Answer

- $AMAT1 = 1 \times 2 + 1.4\% \times 70 = 2.98 \text{ ns}$
- $AMAT2 = 1 \times 2 \times 1.1 + 1\% \times 70 = 2.90 \text{ ns}$
- $CPU \text{ time } 1 = IC \times (2 \times 2 + 1.3 \times 1.4\% \times 70) = 5.27 \times IC$
- $CPU \text{ time } 2 = IC \times (2 \times 2 \times 1.1 + 1.3 \times 1\% \times 70) = 5.31 \times IC$

The Cache Design Space

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



Improving Cache Performance

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

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

Reducing Misses

- Classifying Misses: 3 Cs

- **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called **cold start misses** or **first reference misses**.

(Misses in even an Infinite Cache)

- **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, **capacity misses** will occur due to blocks being discarded and later retrieved.

(Misses in Fully Associative Size X Cache)

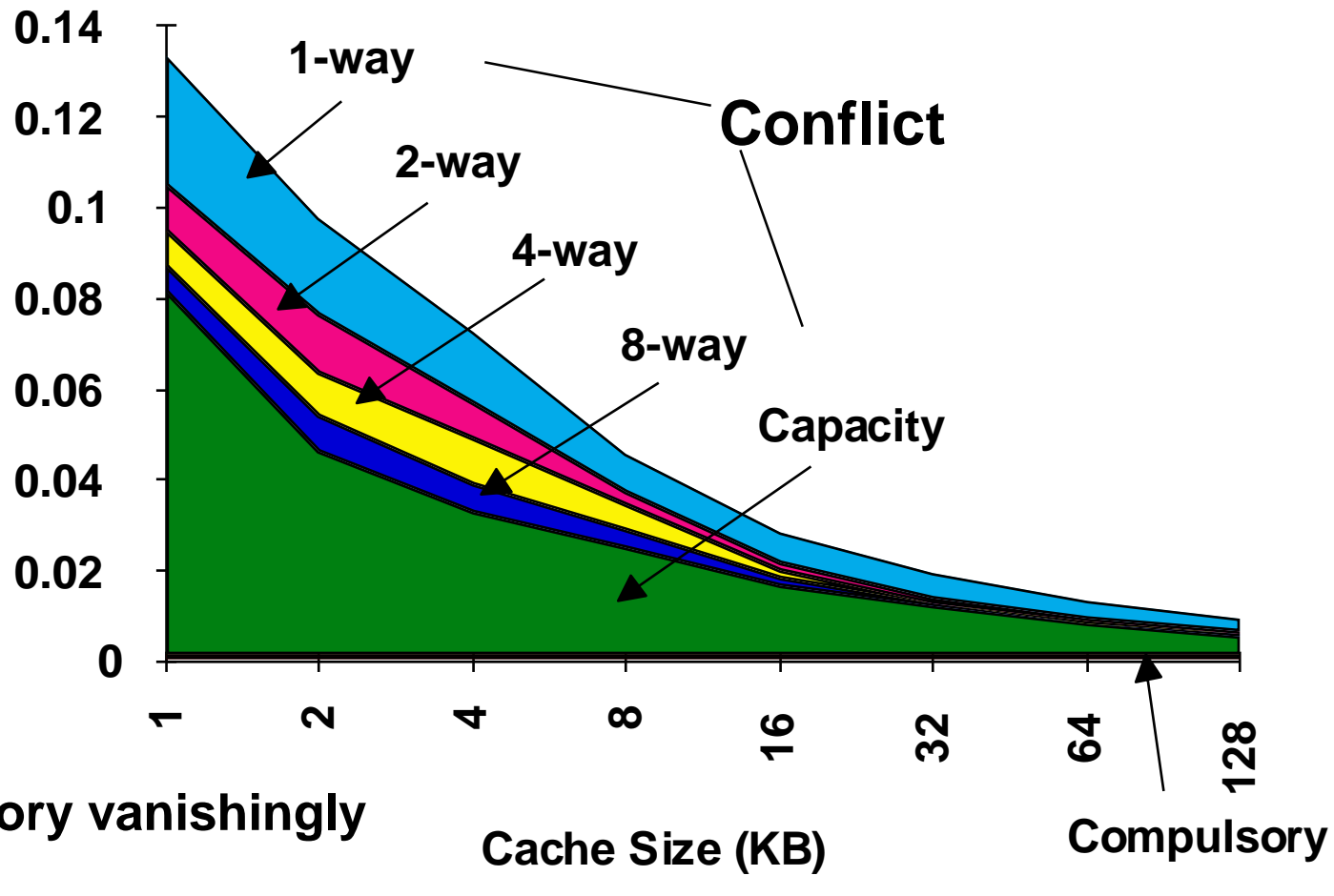
- **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called **collision misses** or **interference misses**.

(Misses in N-way Associative, Size X Cache)

- More recent, 4th “C”:

- **Coherence** - Misses caused by cache coherence.

3Cs Absolute Miss Rate (SPEC92)

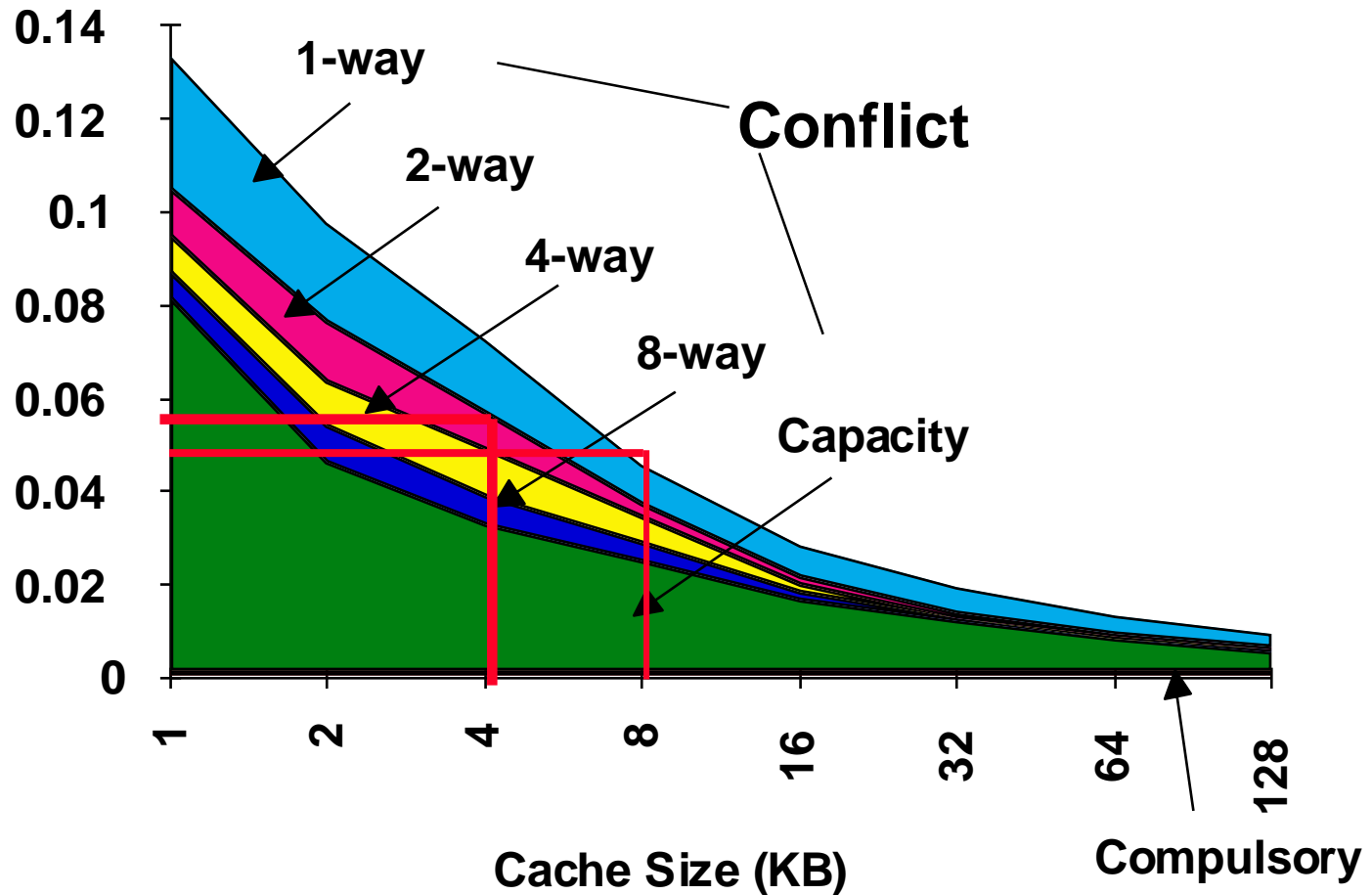


Classify Cache Misses, How?

- **(1) Infinite cache, fully associative**
 - Compulsory misses
- **(2) Finite cache, fully associative**
 - Compulsory misses + Capacity misses
- **(3) Finite cache, limited associativity**
 - Compulsory misses + Capacity misses + Conflict misses

2:1 Cache Rule

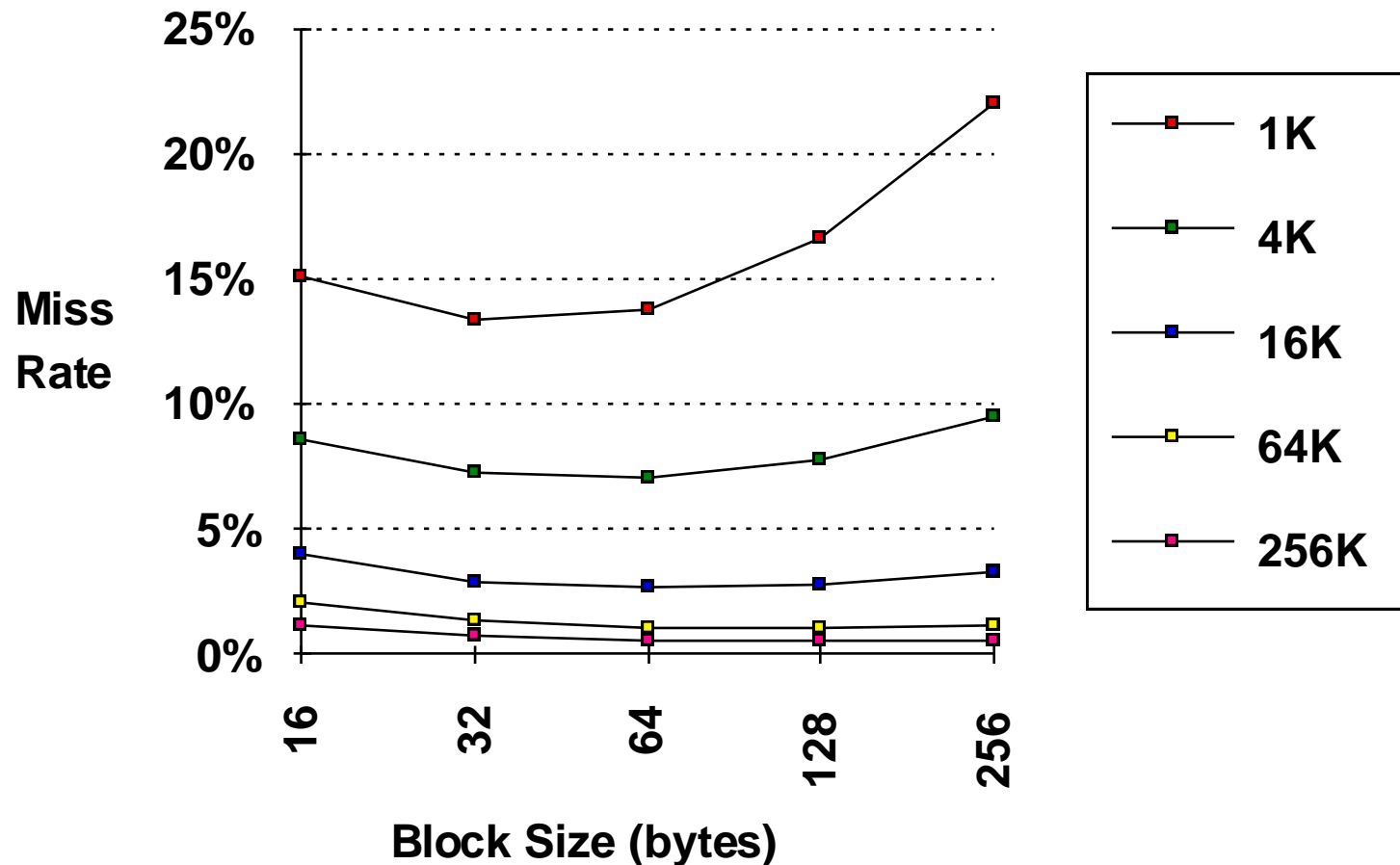
miss rate 1-way associative cache size X
~ miss rate 2-way associative cache size $X/2$



How to Reduce Misses?

- 3 Cs: **Compulsory, Capacity, Conflict**
- In all cases, assume total cache size not changed:
- What happens if:
 - 1) Change Block Size:
Which of 3Cs is obviously affected? **Compulsory**
 - 2) Change Associativity:
Which of 3Cs is obviously affected? **Conflict**
 - 3) Change Algorithm / Compiler:
Which of 3Cs is obviously affected? **3Cs**

1. Reduce Misses via Larger Block Size



Block size	Cache size				
	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Larger Block Size

Compulsory Miss ↓

However,

Miss Penalty ↑

FIGURE 5.12 Actual miss rate versus block size for five different-sized caches in Figure 5.11. Note that for a 1-KB cache, 64-byte, 128-byte, and 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses.

Block size	Miss penalty	Cache size				
		1K	4K	16K	64K	256K
16	42	7.321	4.599	2.655	1.857	1.458
32	44	6.870	4.186	2.263	1.594	1.308
64	48	7.605	4.360	2.267	1.509	1.245
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

If the cache is small:

Conflict Miss ↑

Capacity Miss ↑

FIGURE 5.13 Average memory access time versus block size for five different-sized caches in Figure 5.11. The smallest average time per cache size is boldfaced.

2. Reduce Misses via Higher Associativity

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

Example:

Avg. Memory Access Time vs. Miss Rate

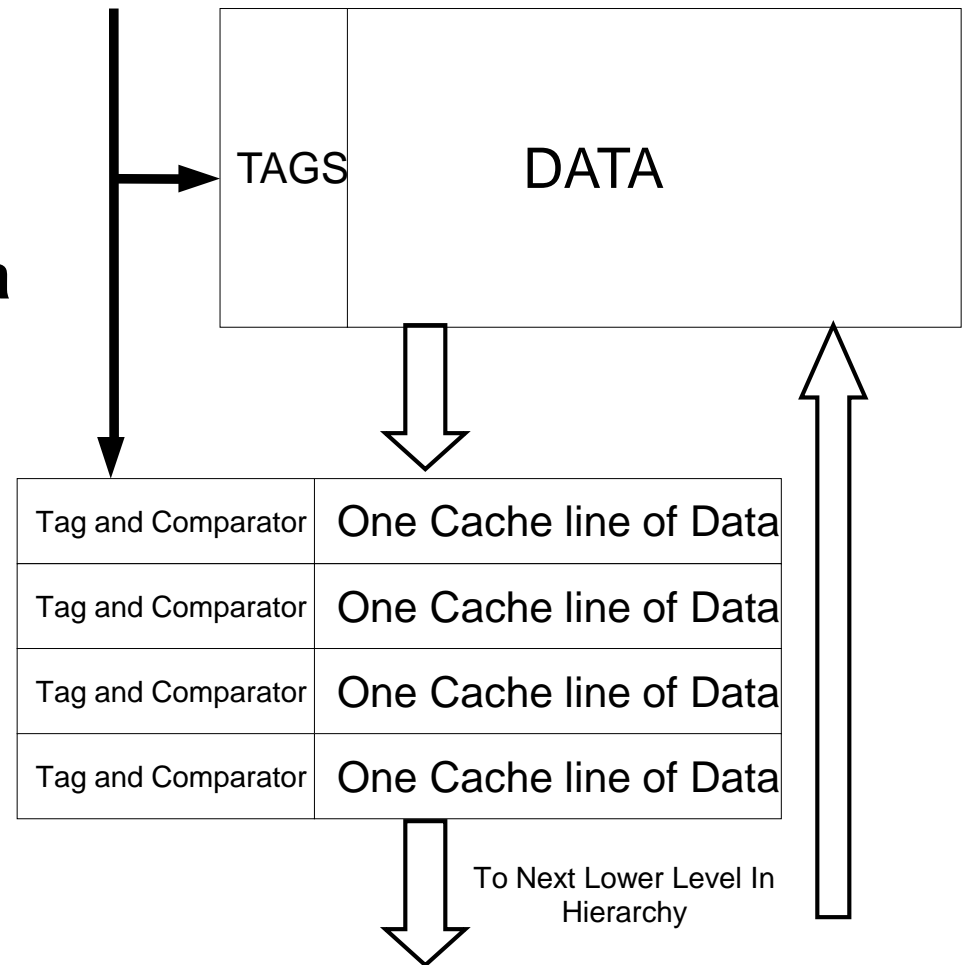
- Assume CCT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

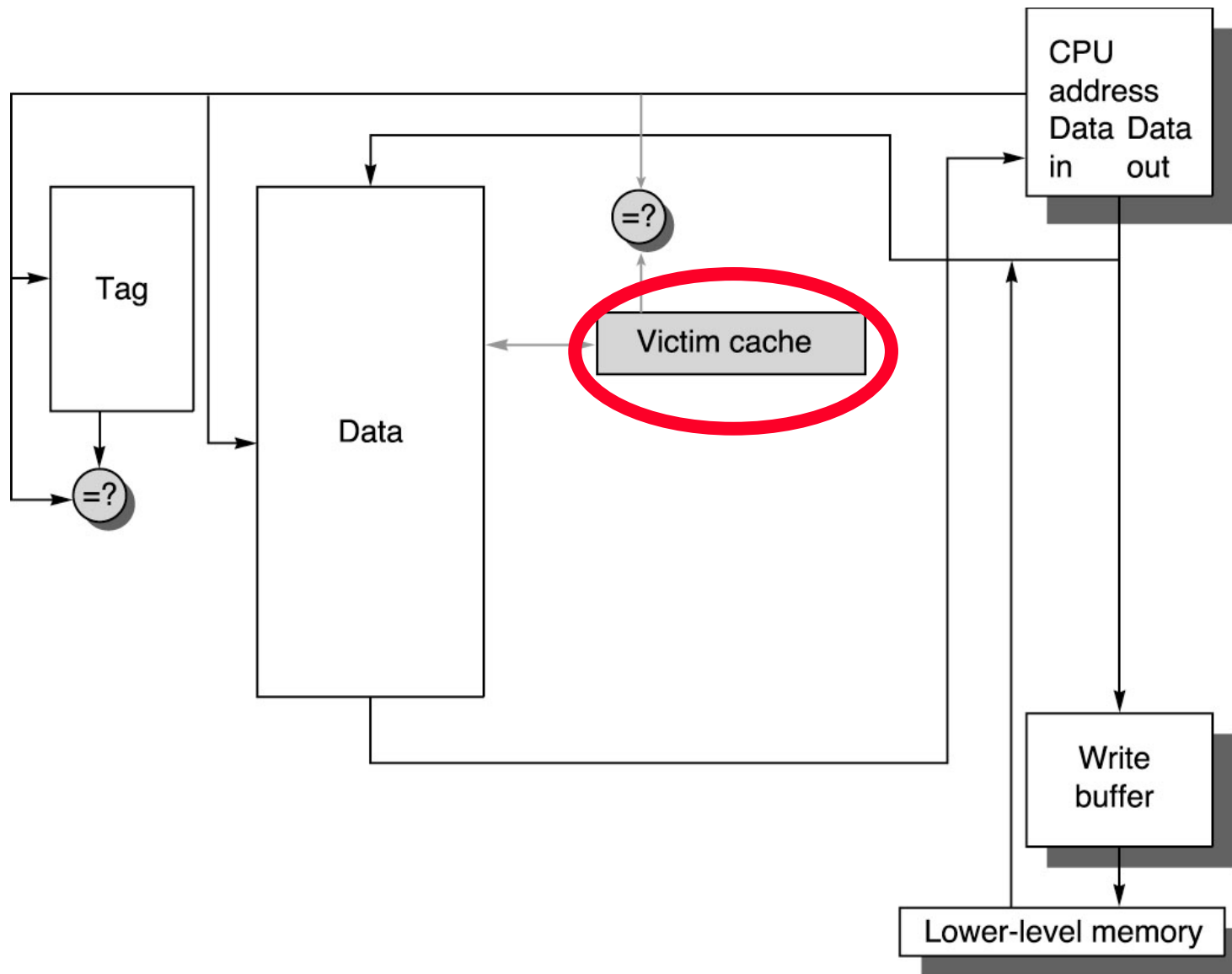
Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

(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**
- **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 via “Pseudo-Associativity”

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)



- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
 - Better for caches not tied directly to processor (L2)
 - Used in MIPS R1000 L2 cache, similar in UltraSPARC

5. Reducing Misses by Hardware Prefetching of Instructions & Data

6. Reducing Misses by Software Prefetching Data

- **E.g., Instruction Prefetching**
 - Alpha 21064 fetches 2 blocks on a miss
 - Extra block placed in “**stream buffer**”
 - » **Instruction stream buffer**
 - » **Data stream buffer**
 - On miss check stream buffer
- **Works with data blocks too:**
 - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 stream buffers got 43%
 - Palacharla & Kessler [1994] for scientific programs for 8 stream buffers 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

7. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Reorder procedures in memory so as to reduce misses
- Data
 - **Merging Arrays**: improve spatial locality by single array of compound elements vs. 2 arrays
 - **Loop Interchange**: change nesting of loops to access data in order stored in memory
 - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
 - **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows


Merging Arrays Example

```
/* Before: 2 sequential arrays */  
int val[SIZE];  
int key[SIZE];  
  
/* After: 1 array of structures */  
struct merge {  
    int val;  
    int key;  
};  
struct merge merged_array[SIZE];
```

- Some programs reference multiple arrays in the same dimension with the same indices. (these accesses will interfere with each other)
- Reducing conflicts by combining independent matrices into a single compound array so that each cache block can contain the desired elements; improve spatial locality

Loop Interchange Example

```
/* Before */  
for (k = 0; k < 100; k = k+1)  
    for (j = 0; j < 100; j = j+1)  
        for (i = 0; i < 5000; i = i+1)  
            x[i][j] = 2 * x[i][j];  
  
/* After */  
for (k = 0; k < 100; k = k+1)  
    for (i = 0; i < 5000; i = i+1)  
        for (j = 0; j < 100; j = j+1)  
            x[i][j] = 2 * x[i][j];
```



Row major
or column
major

- Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

- 2 misses per access to a & c vs. one miss per access; improve spatial locality

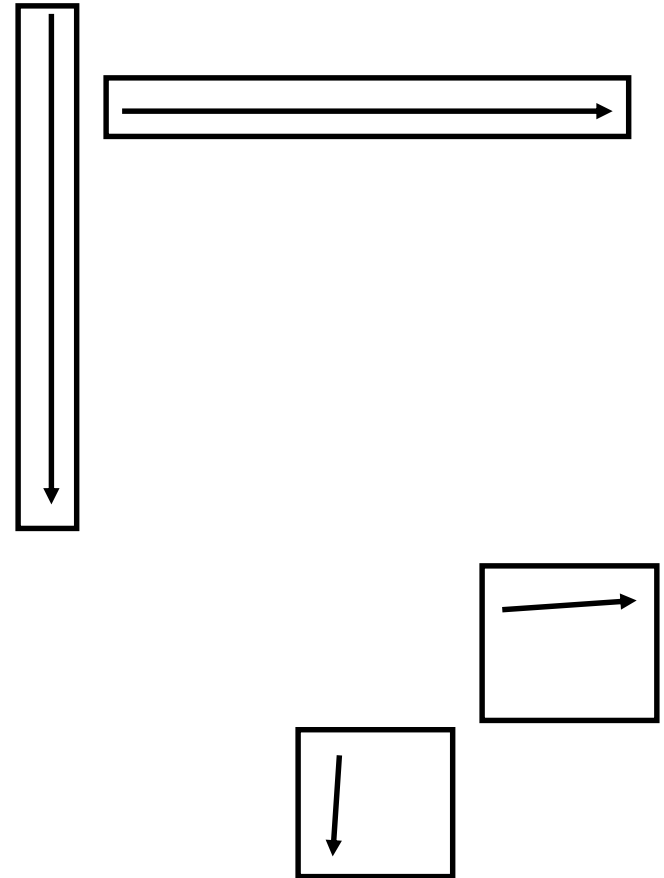
Blocking Example

```
/* Before */  
for (i = 0; i < N; i = i+1)  
  for (j = 0; j < N; j = j+1)  
    {r = 0;  
      for (k = 0; k < N; k = k+1) {  
        r = r + y[i][k]*z[k][j];}  
      x[i][j] = r;  
    };
```

- **Two Inner Loops:**

- Read all NxN elements of z[]
- Read N elements of 1 row of y[] repeatedly
- Write N elements of 1 row of x[]

- **Idea: compute on BxB submatrix that fits**

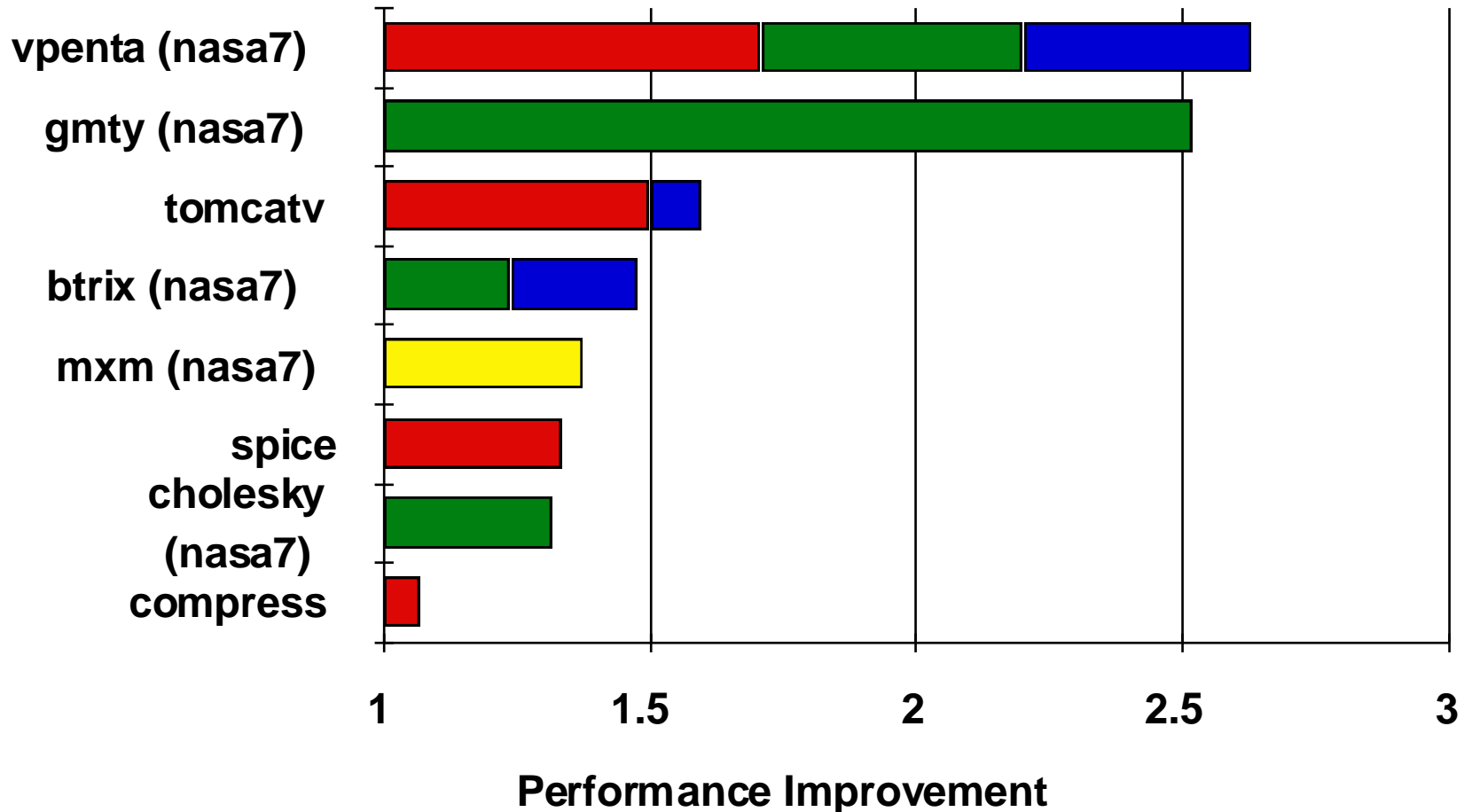


Blocking Example

```
/* After */  
for (jj = 0; jj < N; jj = jj+B)  
for (kk = 0; kk < N; kk = kk+B)  
for (i = 0; i < N; i = i+1)  
    for (j = jj; j < min(jj+B-1,N); j = j+1)  
        {r = 0;  
            for (k = kk; k < min(kk+B-1,N); k = k+1) {  
                r = r + y[i][k]*z[k][j];}  
            x[i][j] = x[i][j] + r;  
        };
```

- B called ***Blocking Factor***

Summary of Compiler Optimizations to Reduce Cache Misses



merged arrays loop interchange loop fusion blocking

Summary on reducing miss rate

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times Miss\ penalty \right) \times Clock\ cycle\ time$$

- **3 Cs: Compulsory, Capacity, Conflict**
 1. Reduce Misses via Larger Block Size
 2. Reduce Misses via Higher Associativity
 3. Reducing Misses via Victim Cache
 4. Reducing Misses via Pseudo-Associativity
 5. Reducing Misses by HW Prefetching Instr, Data
 6. Reducing Misses by SW Prefetching Data
 7. Reducing Misses by Compiler Optimizations
- Remember the danger of concentrating on just one parameter when evaluating performance

Example 1

1. Assume it takes **1 extra cycle** if the instruction misses in the cache is found in the Pre-fetch buffer. Suppose that the **pre-fetch miss rate is 30%**, instruction cache **miss rate 2%**, **hit time takes 2 cycles**, and **miss penalty takes 50 cycles**. What is the **effective miss rate** using **instruction prefetching**?

Answer 1

- **AMAT_prefetch**
= HitTime + MissRate * PrefetchHitRate*1 +
MissRate*(1-PrefetchHitRate)*MissPenalty
= 2 + 2%*70%*1 + 2%*30%*50 =
2+0.014+0.3=2.314

$$\text{AMAT} = \text{HitTime} + \text{EffectiveMissRate} * \text{MissPenalty}$$

$$2.314 = 2 + \text{EffectiveMissRate} * 50$$

$$\text{EffectiveMissRate} = 0.628\%$$

Example 2

- 2. Assume it takes **2 extra cycles** to find the entry in the alternative cache using pseudo-associative scheme. **Hit time takes 2 cycles**, and **miss penalty takes 50 cycles**. Suppose that **miss rate** of a 16K direct map cache is 2%, miss rate of a 16K 2-way set associative cache is 1.2%, Compute the **average memory access time of a 16K pseudo set associative cache**.

Answer 2

- $AMAT_pseudo$
 $= HitTime_pseudo$
 $+ MissRate_pseudo * MissPenalty_pseudo$
 $= HitTime + \text{AltHitRate} * 2$
 $+ MissRate_2way * MissPenalty$
 $= 2 + (MissRate_1way - MissRate_2way) * 2$
 $+ 1.2\% * 50$
 $= 2 + (2\% - 1.2\%) * 2 + 1.2\% * 50 = 2 + 0.016 + 0.6 = 2.616$

Improving Cache Performance

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times \text{Miss rate} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

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

Reducing Cache Miss Penalty:

1. Read Priority over Write on Miss

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times Miss\ rate \times Miss\ penalty \right) \times Clock\ cycle\ time$$

- **Write through with write buffer**

- If simply wait for write buffer to empty, might increase read miss penalty
- Check write buffer contents before read; if not conflicts, let the memory access continue

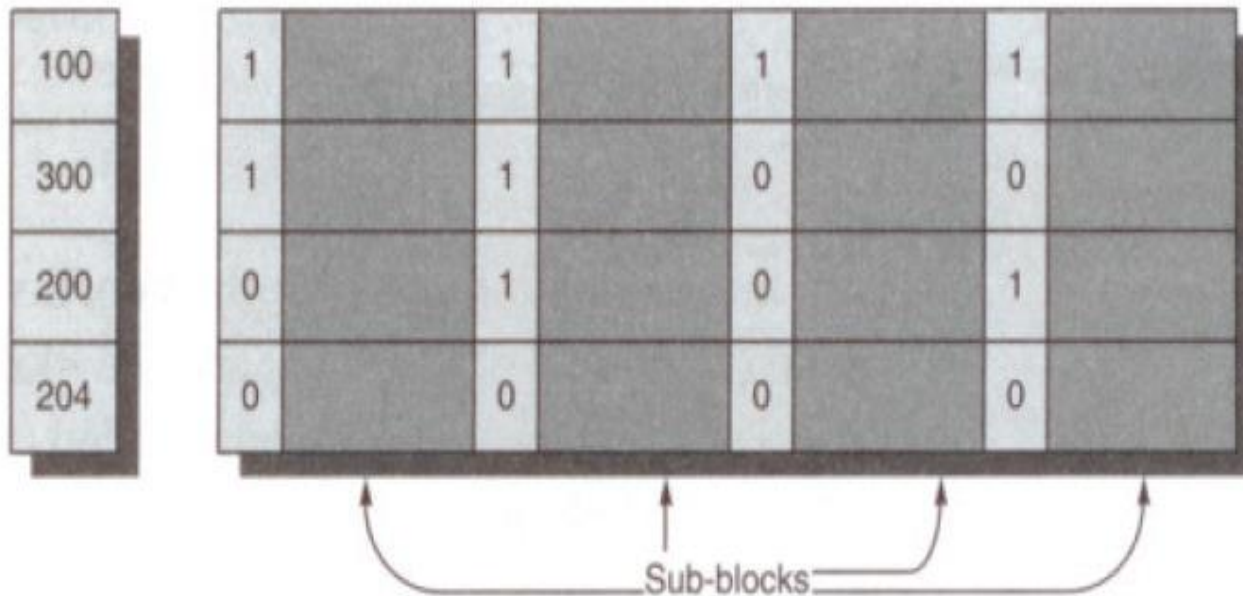
- **Write Back?**

- Read miss replaces dirty block
- Normal: Write dirty block to memory, then do the read
- **Instead:** Copy the dirty block to a write buffer, then do the read, and then do the write
- CPU stall less since restarts as soon as read is done

slow

Reduce Miss Penalty: 2. Sub-block Placement

- Don't have to load full block on a miss
- Have valid bits per sub-block to indicate valid



Reduce Miss Penalty: 3. Early Restart and Critical Word First

- **Do not wait for full block to be loaded before restarting CPU**
 - **Early restart :**
 - » as soon as 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**

Reduce Miss Penalty: 4. Non-blocking Caches to reduce stalls on misses

- **Non-blocking cache** or **lockup-free cache** allow data cache to continue to supply cache hits during a miss
 - out-of-order execution
 - requires multi-bank memories
- “**hit under miss**” reduces the effective miss penalty by working during miss
- “**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 multiple memory banks (otherwise cannot support)
 - Pentium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC

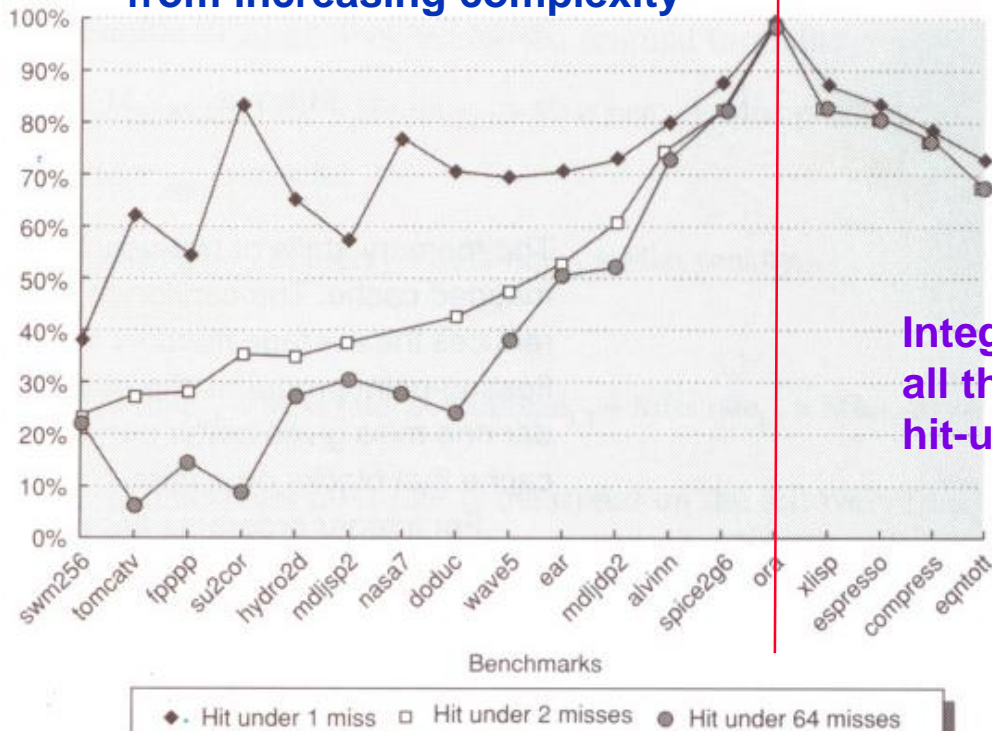
FP Programs benefit more
from increasing complexity

The data was collected for
an 8-KB direct-mapped data
cache with 32-byte blocks
and a 16 cycle miss penalty

Ratio of the
average
memory
stall time

Ratio of the
Average
Memory
Stall Time

(Compared
with
blocking
cache)



Integer Programs get almost
all the benefit from a simple
hit-under-one-miss scheme

- Average of FP programs:
 - Hit under 1 miss: 76%, 2 misses: 51%, 64 misses: 39%
- Average of integer programs:
 - Hit under 1 miss: 81%, 2 misses: 78%, 64 misses: 78%

Example

8-KB direct-mapped data cache with 32-byte blocks and a 16 cycle miss penalty

- For the cache described in the previous slide, **which is more important**
 - A. Two-way set associativity
 - B. Hit under one miss
 - Average of FP programs:
 - Hit under 1 miss: 76%
 - Average of integer programs:
 - Hit under 1 miss: 81%
- for **Floating Point** and **Integer** Programs?

- Assume the following miss rate for 8-KB cache
 - 11.4% for **FP** programs with a direct mapped cache
 - 10.7% for **FP** programs with a two-way associative cache
 - 7.4 % for **Int** programs with a direct mapped cache
 - 6.0 % for **Int** programs with a two-way associative cache

- Assume

Average memory stall time = miss rate x miss penalty

Answer of the Example

- **FP:**

- $\text{Miss_rate_DM} \times \text{Miss penalty} = 11.4\% \times 16 = 1.84$
- $\text{Miss rate_2way} \times \text{Miss penalty} = 10.7\% \times 16 = 1.71$
- Ratio of the Average Memory Stall Time (2 way / direct map) = $1.71/1.84 = 93\%$
- From the previous slide, the Ratio of the Average Memory Stall Time (hit-under-one-miss/blocking cache) = 76%
- **Supporting hit under miss gives better performance improvement (more important)**

- **Integer:**

- $\text{Miss_rate_DM} \times \text{Miss penalty} = 7.4\% \times 16 = 1.18$
- $\text{Miss rate_2way} \times \text{Miss penalty} = 6\% \times 16 = 0.96$
- Ratio of the Average Memory Stall Time (2 way / direct map) = $0.96/1.18 = 81\%$
- From the previous slide, the Ratio of the Average Memory Stall Time (hit-under-one-miss/blocking cache) = 81%
- **About the same performance improvement**
- **Which one will you choose?**

One potential advantage of hit-under-miss is that it won't increase hit time

Reduce Miss Penalty: 5. Add a second-level cache

- L2 Equations

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$$

- 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**
(global miss rate for L2 cache: $\text{Global miss rate} \times \text{Miss Rate}_{L1} \times \text{Local miss rate} \times \text{Miss Rate}_{L2}$)
- **Global Miss Rate is what matters**

Example of global and local miss rate

- In 1000 memory references, there are 50 misses in the first level and 20 misses in the second level
- Global miss rate for first level 5%
- Local miss rate for first level 5%
- Global miss rate for second level 2%
- Local miss rate for second level $20/50 = 40\%$

Example: Given the data below, what is the impact of second-level cache associativity on the miss penalty?

- **Two way set associativity increases hit time by 1 clock cycle**
- **Hit Time_L2 for direct mapped = 10 clock cycles**
- **Local Miss Rate_L2 for direct mapped = 25%**
- **Local Miss Rate_L2 for two-way set associative = 20%**
- **Miss Penalty_L2 = 50 clock cycles**

Answer

- For a direct mapped L2 cache, the L1 cache miss penalty is

$$10 + 25\% \times 50 = 22.5 \text{ clock cycles}$$

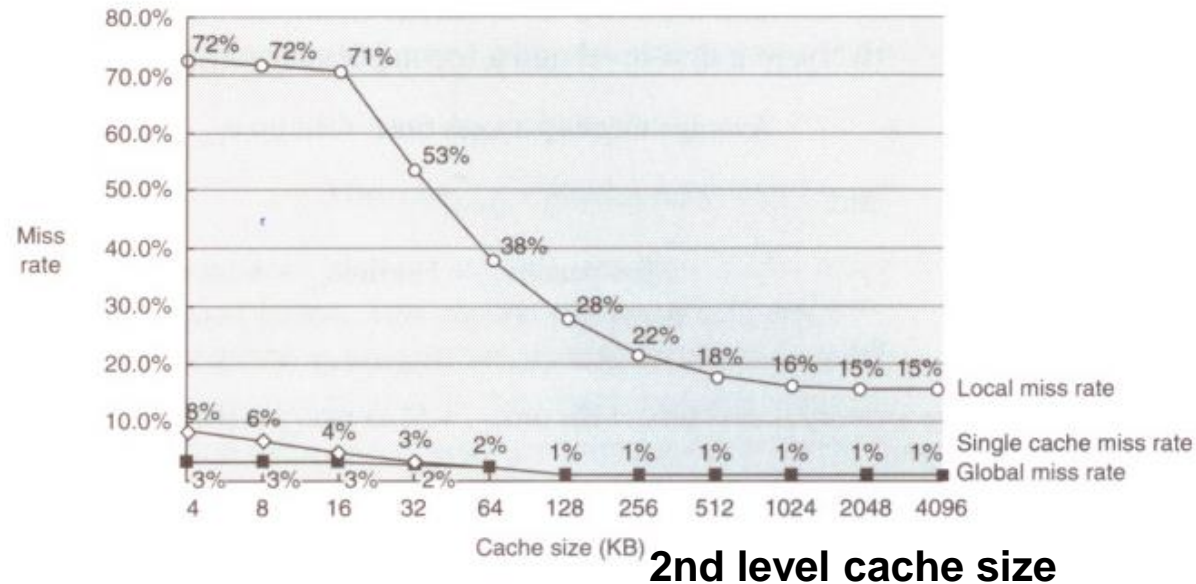
- For a 2-way L2 cache, the L1 cache miss penalty is

$$11 + 20\% \times 50 = 21 \text{ clock cycles}$$

- Conclusion: **Higher associativity or pseudo-associativity are worth considering** because
 - They have small impact on the second-level hit time
 - Much of the AMAT is due to misses in the second level cache

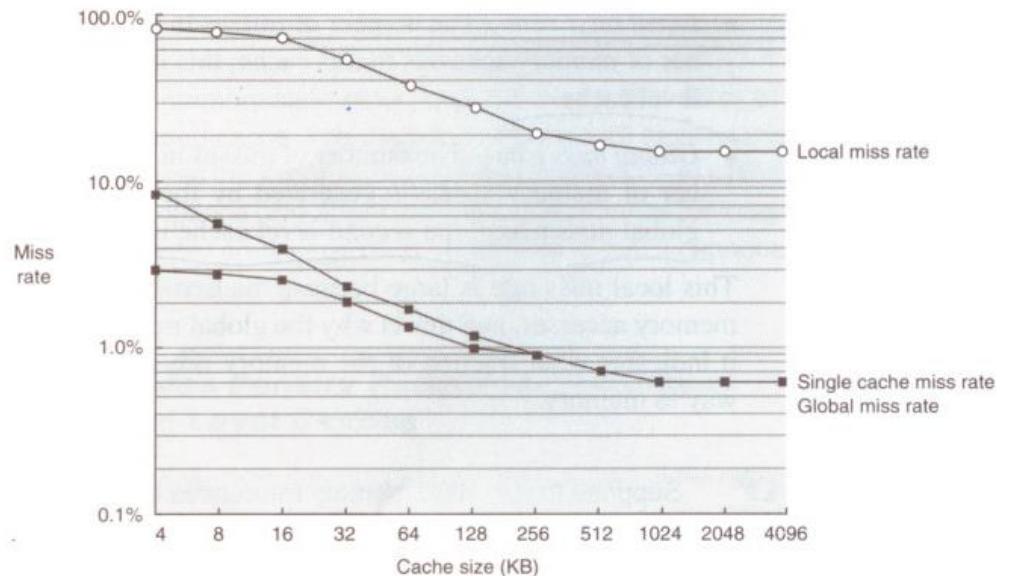
Comparing Local and Global Miss Rates

- 32 KByte 1st level cache;
- L2 >> L1
- L2 not tied to CPU clock cycle



Usually for L2 cache:

- Higher associativity or pseudo associativity
- Larger block size
- Multilevel inclusion property



Reducing Miss Penalty Summary

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{Memory\ accesses}{Instruction} \times Miss\ rate \times Miss\ penalty \right) \times Clock\ cycle\ time$$

- **Techniques**

- Read priority over write on miss
- Sub-block placement
- Early Restart and Critical Word First on miss
- Non-blocking Caches (Hit under Miss, Miss under Miss)
- Second Level Cache

- **Can be applied recursively to Multilevel Caches**

- Danger is that time to DRAM will grow with multiple levels in between

Cache Optimization Summary on miss rate and miss penalty

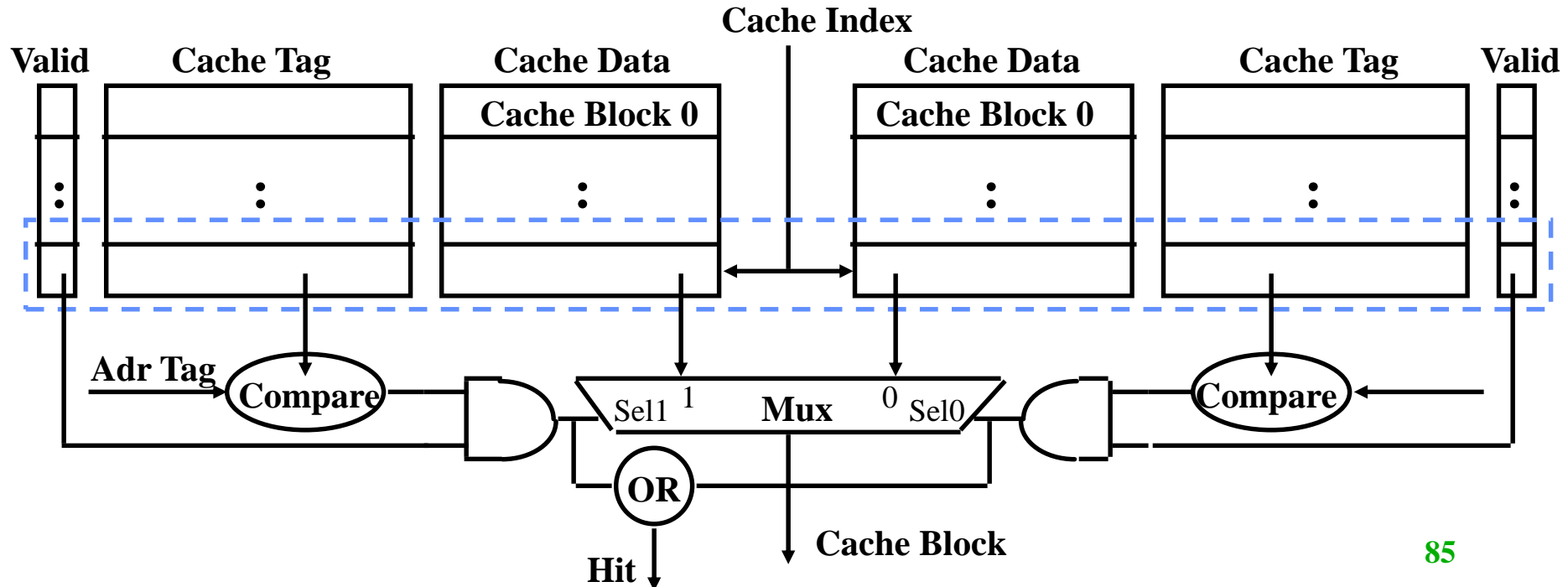
	<i>Technique</i>	<i>MR</i>	<i>MP</i>	<i>HT</i>	<i>Complexity</i>
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Sub-block placement		+		1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2

Reducing Hit time

- **Hit time is critical because it affects the clock cycle time**
- **A fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything**
- **Fast hit time via small and simple cache**

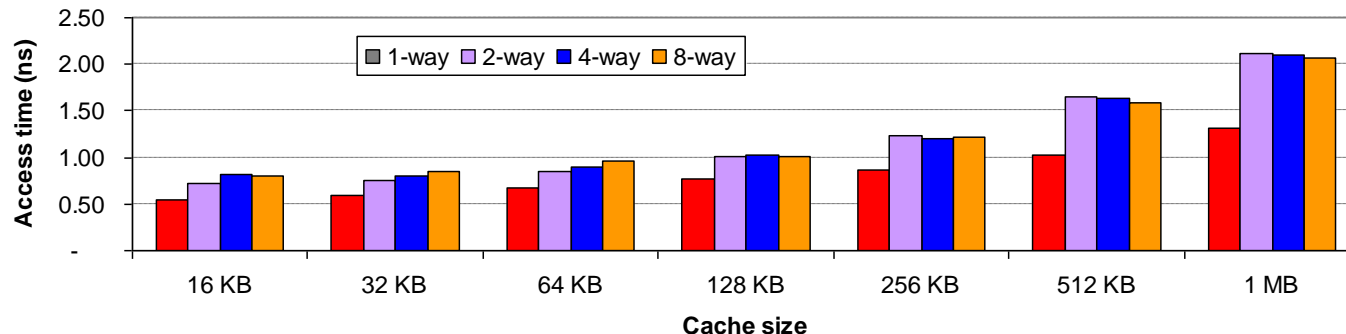
Disadvantage of Set Associative Cache

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



Fast Hit time via Small and Simple Caches

- Index tag memory and then compare takes time
- \Rightarrow **Small** cache can help hit time since smaller memory takes less time to index
 - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
 - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- **Simple** \Rightarrow direct mapping
 - Can overlap tag check with data transmission
- Access time estimate for 90 nm using CACTI model 4.0
 - Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches



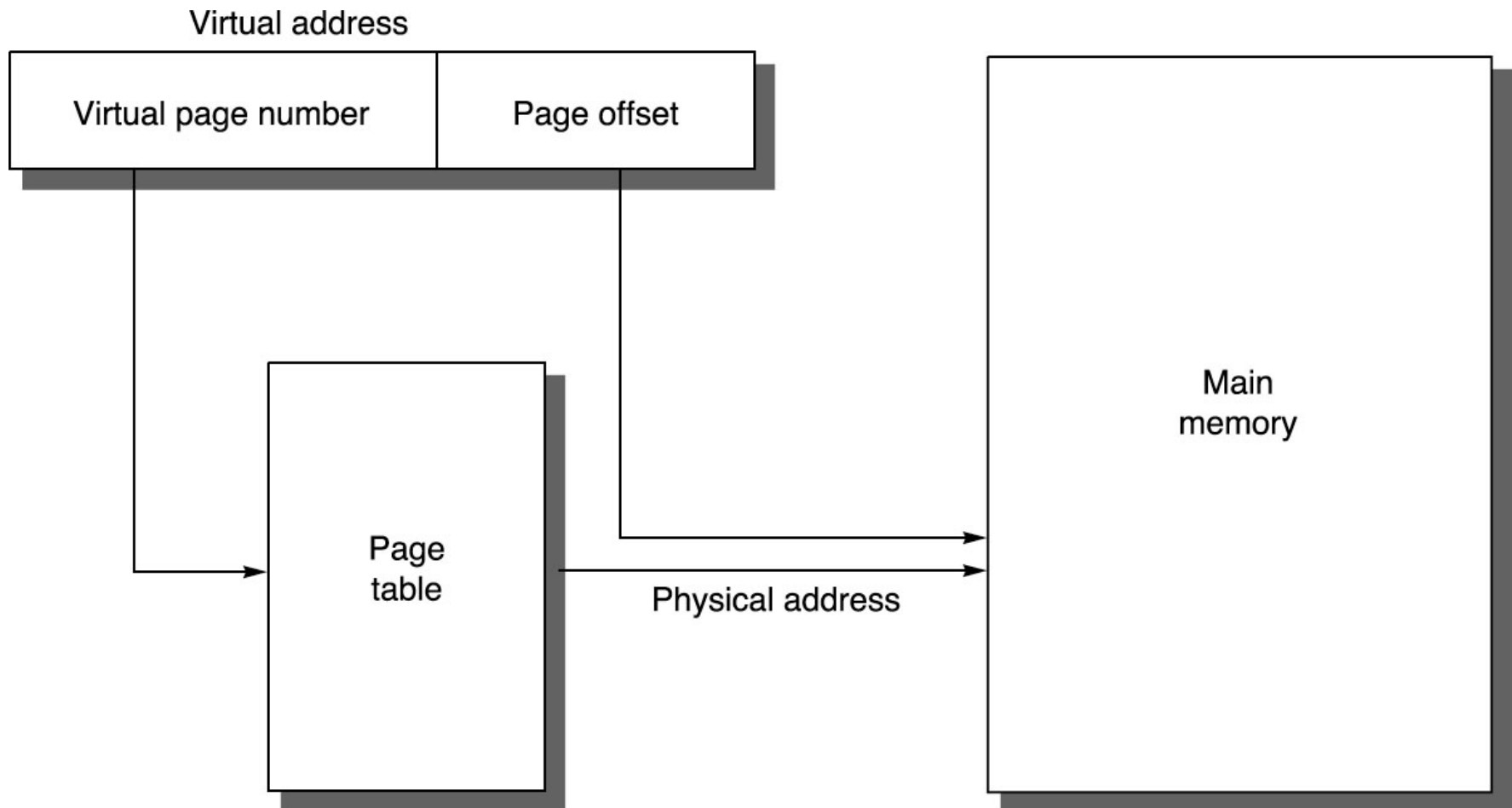
Virtual Memory

For Virtual Memory

- **Q1: Where can a block be placed in the main memory?**
- **Q2: How is a block found if it is in the main memory?**
- **Q3: Which block should be replaced on a miss (page fault)?**
- **Q4: What happens on a write?**

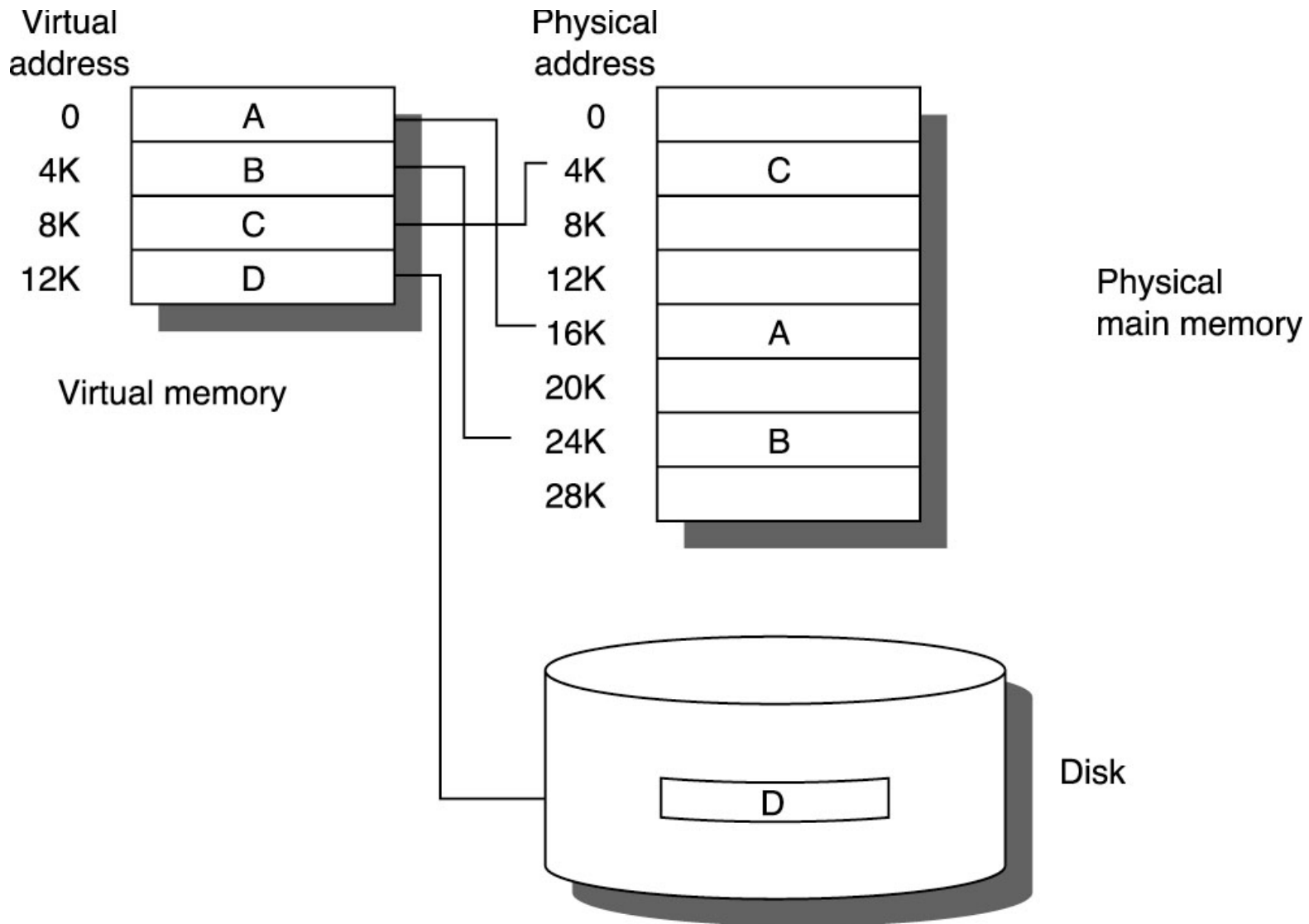
For Virtual Memory

- Q1: Where can a block be placed in the main memory?
(placement)
 - Fully Associative (Due to the exorbitant miss penalty)
- Q2: How is a block found if it is in the upper level?
(identification)
 - Page Table
 - Translation look-aside buffer to reduce address translation time
- Q3: Which block should be replaced on a miss?
(replacement)
 - LRU (provide use bit or reference bit)
- Q4: What happens on a write?
(Write strategy)
 - Write Back

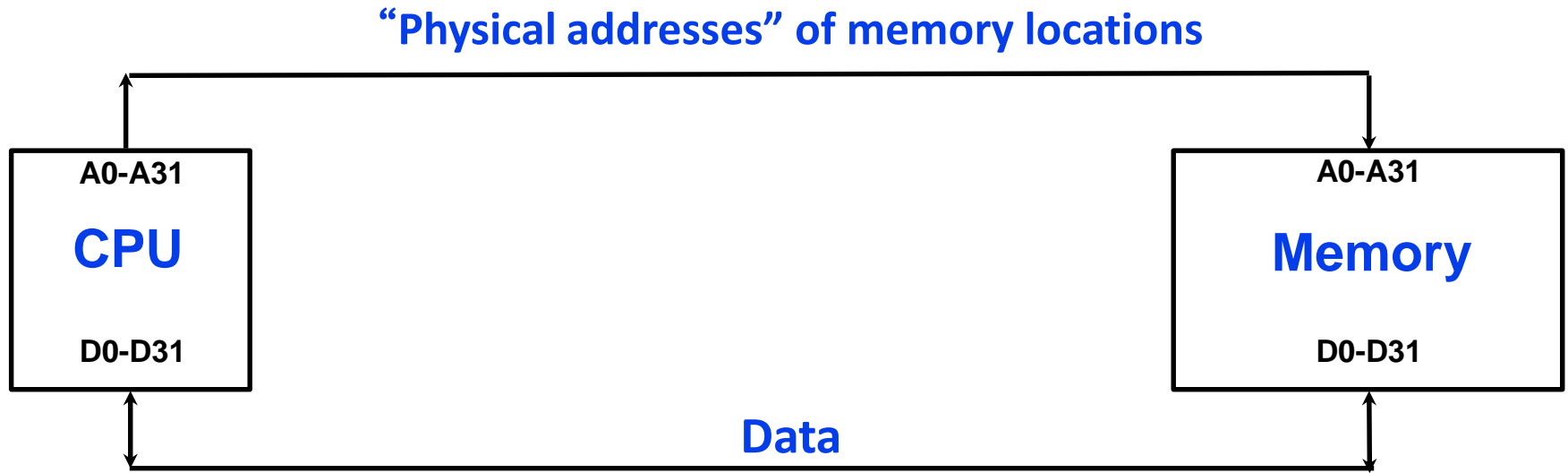


Page size usually large:

- Smaller **page table**, save memory, reduce TLB misses
- Transferring larger pages to/from secondary storage is more efficient



The Limits of Physical Addressing

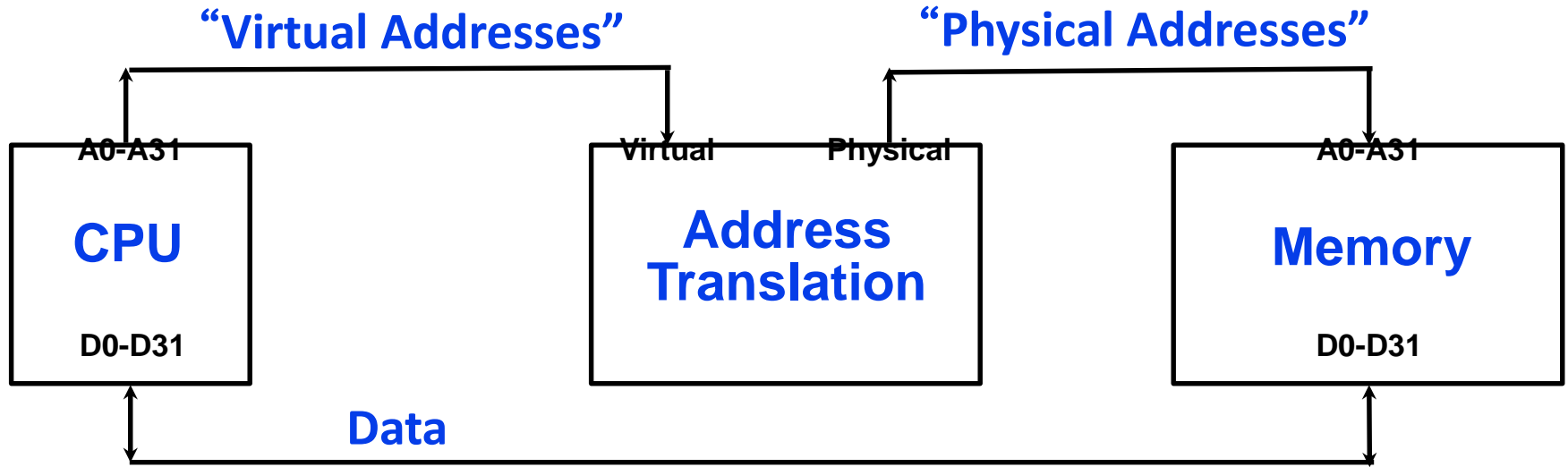


**All programs share one address space:
The **physical** address space**

**Machine language programs must be
aware of the machine organization**

**No way to prevent a program from
accessing **any machine resource****

Solution: Add a Layer of Indirection



User programs run in an standardized
virtual address space

Address Translation hardware
managed by the operating system (OS)
maps virtual address to physical memory

Hardware supports "modern" OS features:
Protection, Translation, Sharing

Three Advantages of Virtual Memory

- Translation:

- Program can be given **consistent view** of memory, even though physical memory is scrambled
- Makes **multithreading** reasonable (now used a lot!)
- Only the most important part of program (“**Working Set**”) must be in **physical memory**.
- Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.

- Protection:

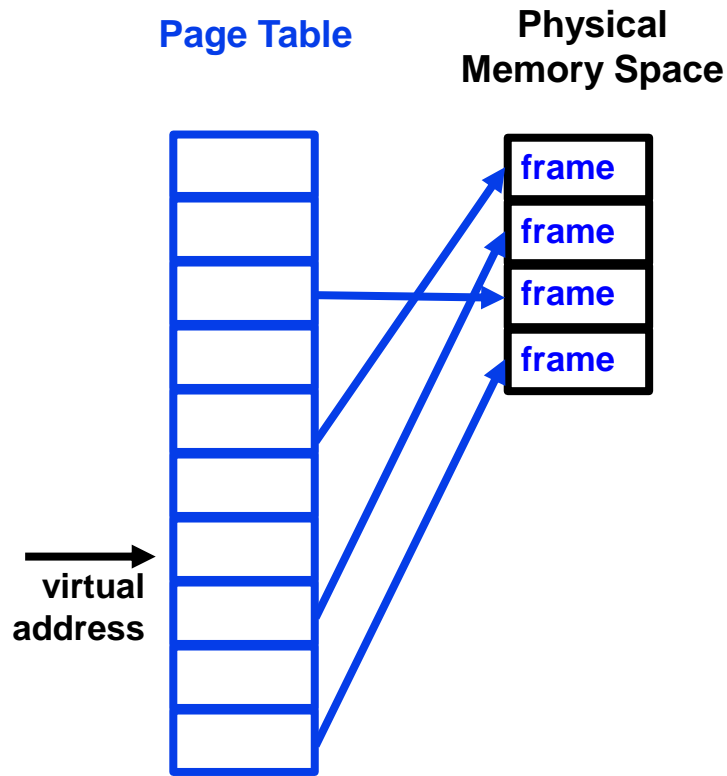
$$\text{Base} \leq \text{Address} \leq \text{Bound}$$

- Different threads (or processes) protected from each other.
- Different pages can be given special behavior
 - » (**Read Only, Invisible to user programs, etc**).
- Kernel data protected from User programs
- Very important for protection from malicious programs

- Sharing:

- Can map same physical page to multiple users (“Shared memory”)

Page tables encode virtual address spaces



OS manages
the page
table for
each address
space ID

A virtual address space
is divided into blocks
of memory called **pages**

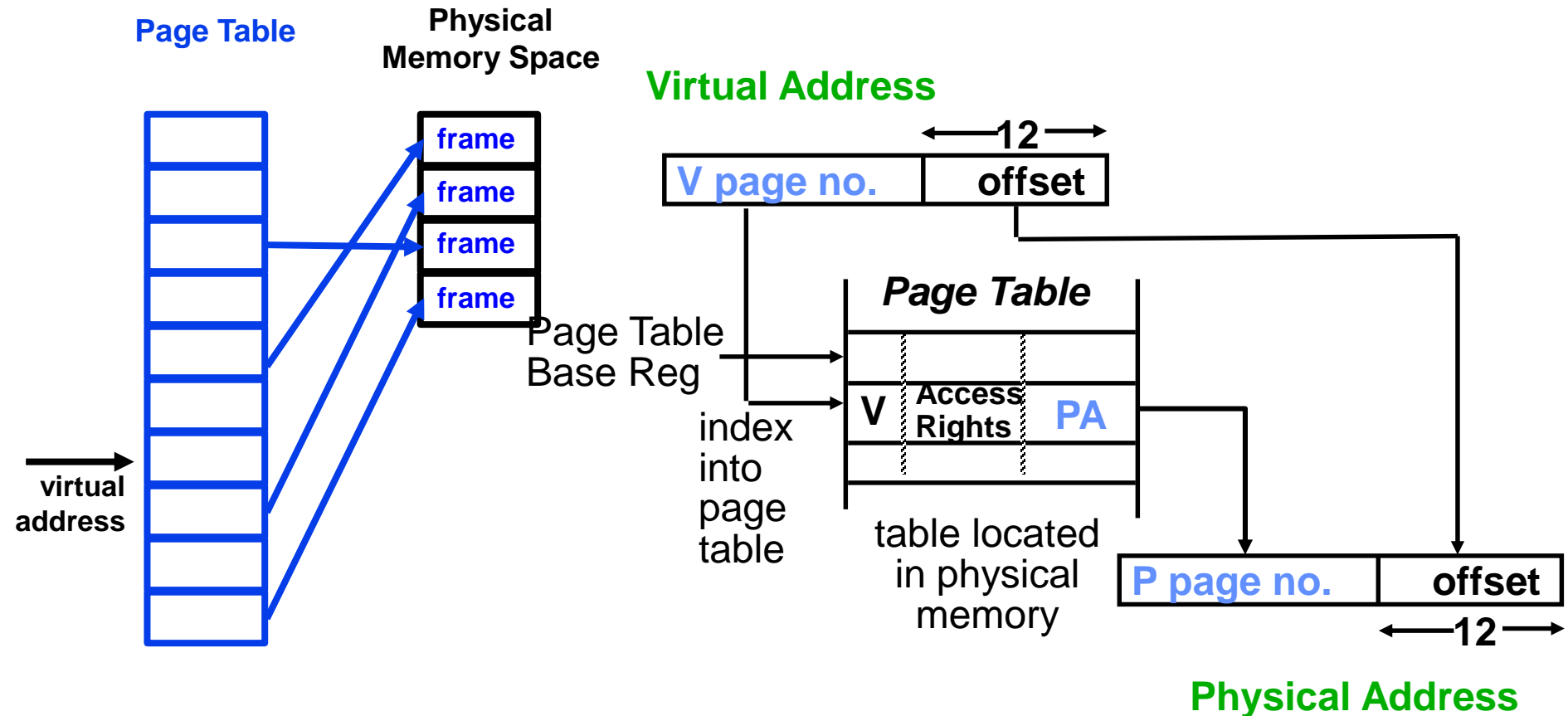
A machine
usually supports
pages of a few
sizes
(MIPS R4000):

Page Size
4 Kbytes
16 Kbytes
64 Kbytes
256 Kbytes
1 Mbyte
4 Mbytes
16 Mbytes

A page table is indexed by a
virtual address

A valid page table entry codes **physical
memory “frame”** address for the page

Details of Page Table



- Page table maps virtual page numbers to physical frames (“PTE” = Page Table Entry)
- Virtual memory => treat memory \approx cache for disk

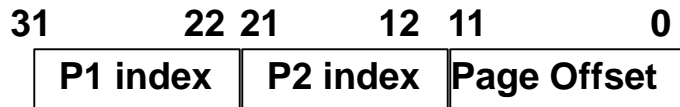
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

Each process needs its own address space!

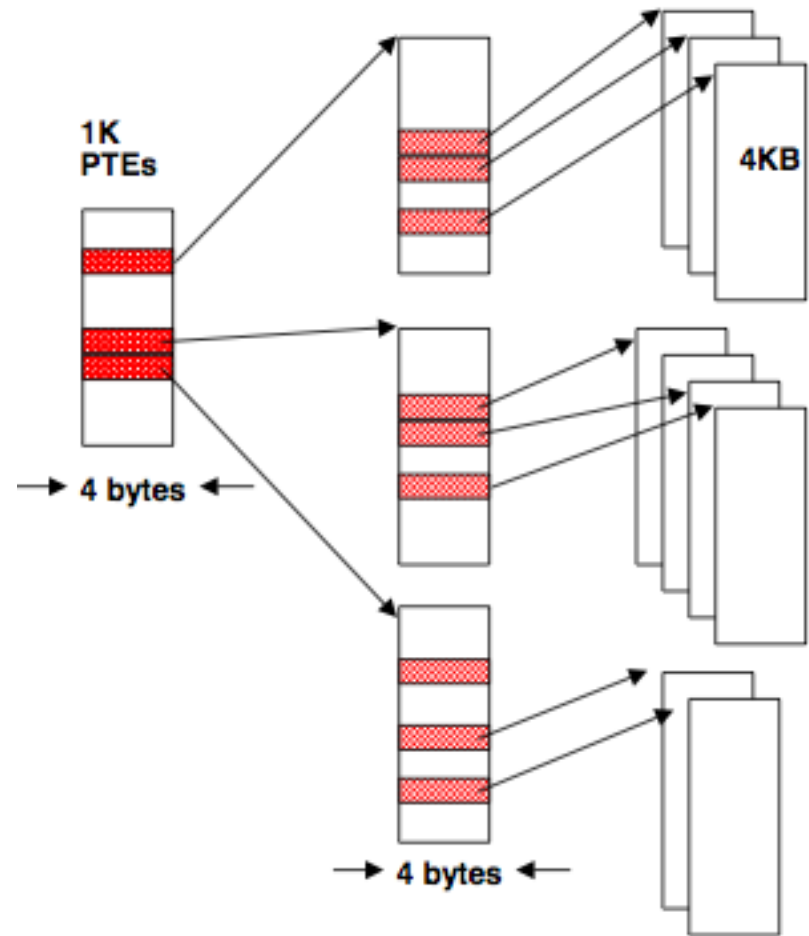
Two-level Page Tables

32 bit virtual address



Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated



VM and Disk: Page replacement policy

Page Table

dirty	used	
1	0	...
1	0	
0	1	
1	1	
0	0	

Dirty bit: page
written.

Used bit: set to
1 on any reference

Set of all pages
in Memory

Tail pointer:
Clear the used
bit in the
page table

Head pointer

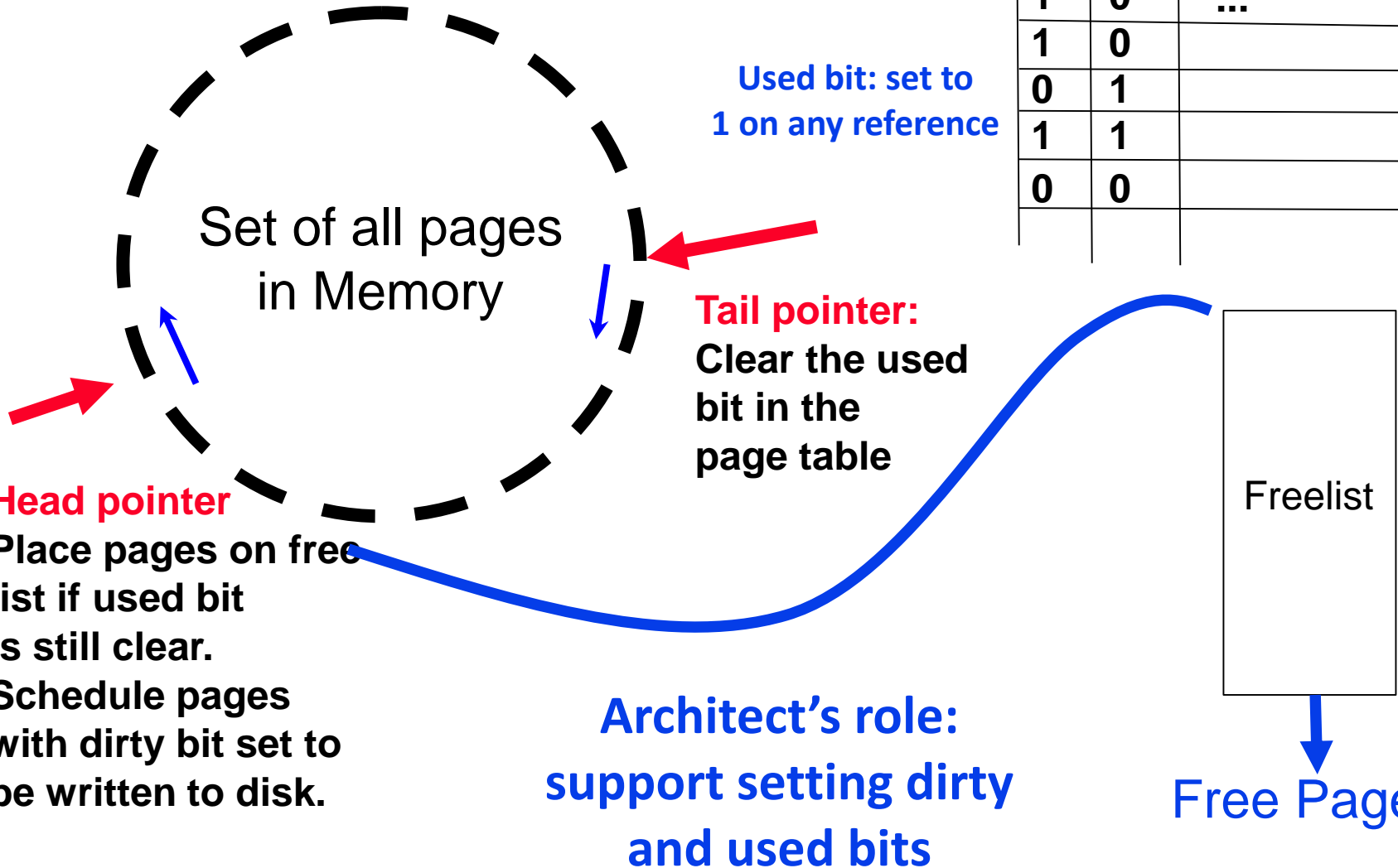
Place pages on free
list if used bit
is still clear.

Schedule pages
with dirty bit set to
be written to disk.

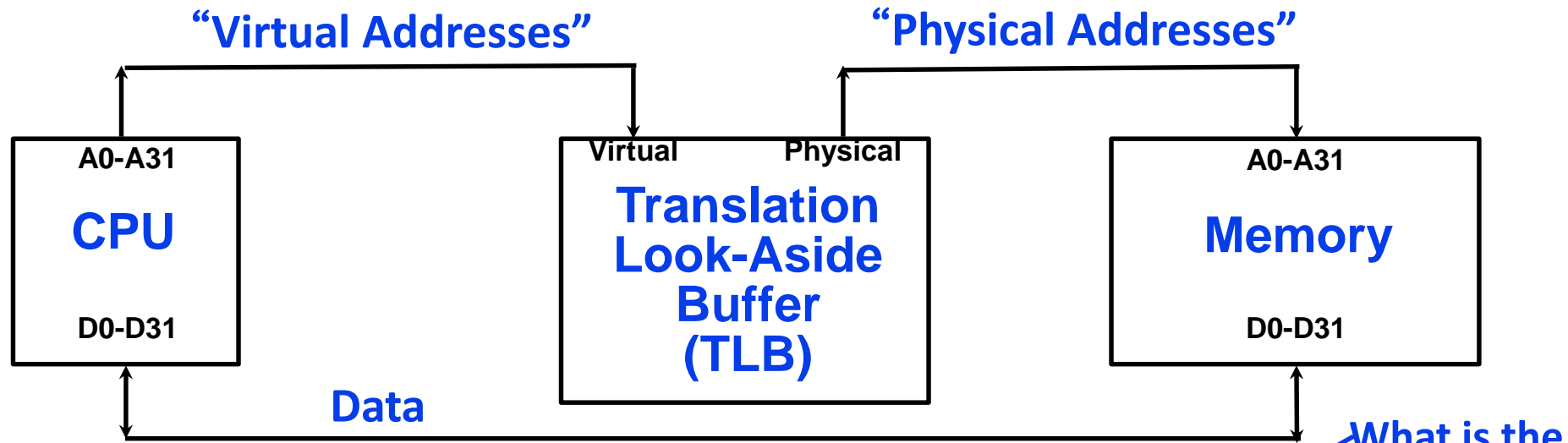
Architect's role:
support setting dirty
and used bits

Freelist

Free Pages



MIPS Address Translation: How does it work?



Translation Look-Aside Buffer (TLB)

A small fully-associative cache of mappings from virtual to physical addresses

What is the table of mappings that it caches?

TLB also contains protection bits for virtual address

Fast common case: Virtual address is in TLB, process has permission to read/write it.

The TLB caches page table entries

TLB caches page table entries.

