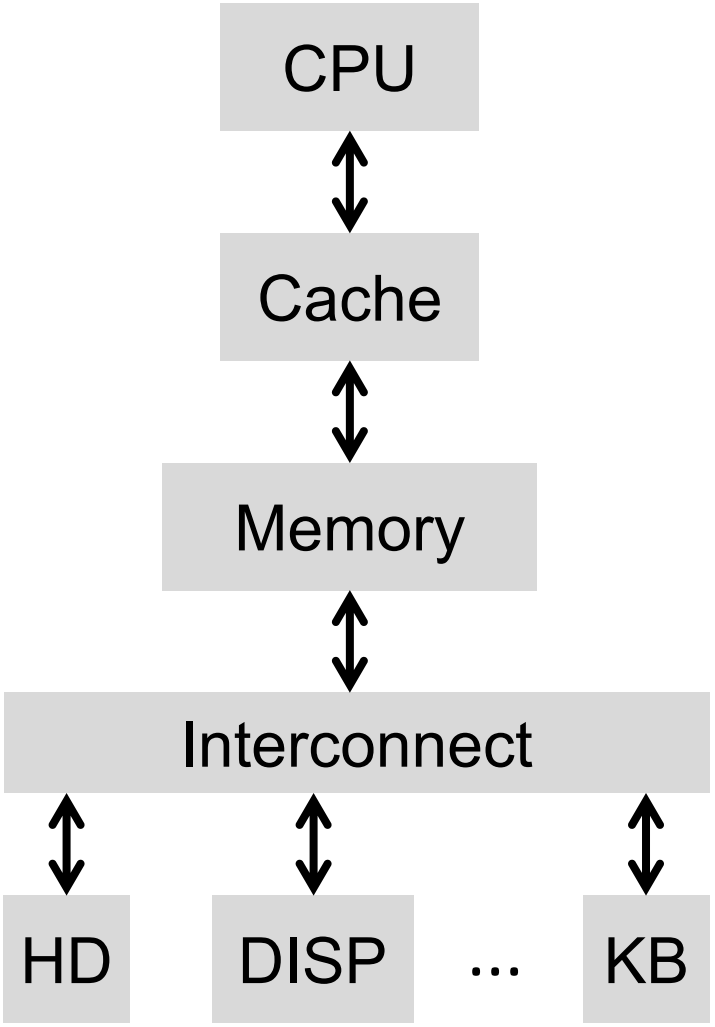
EEL 6764 Principles of Computer Architecture
Memory Hierarchy Design

Instructor: Hao Zheng
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620
Email: haozheng@usf.edu
Phone: (813)974-4757
Fax: (813)974-5456

A Simplified View of Computers



Reading

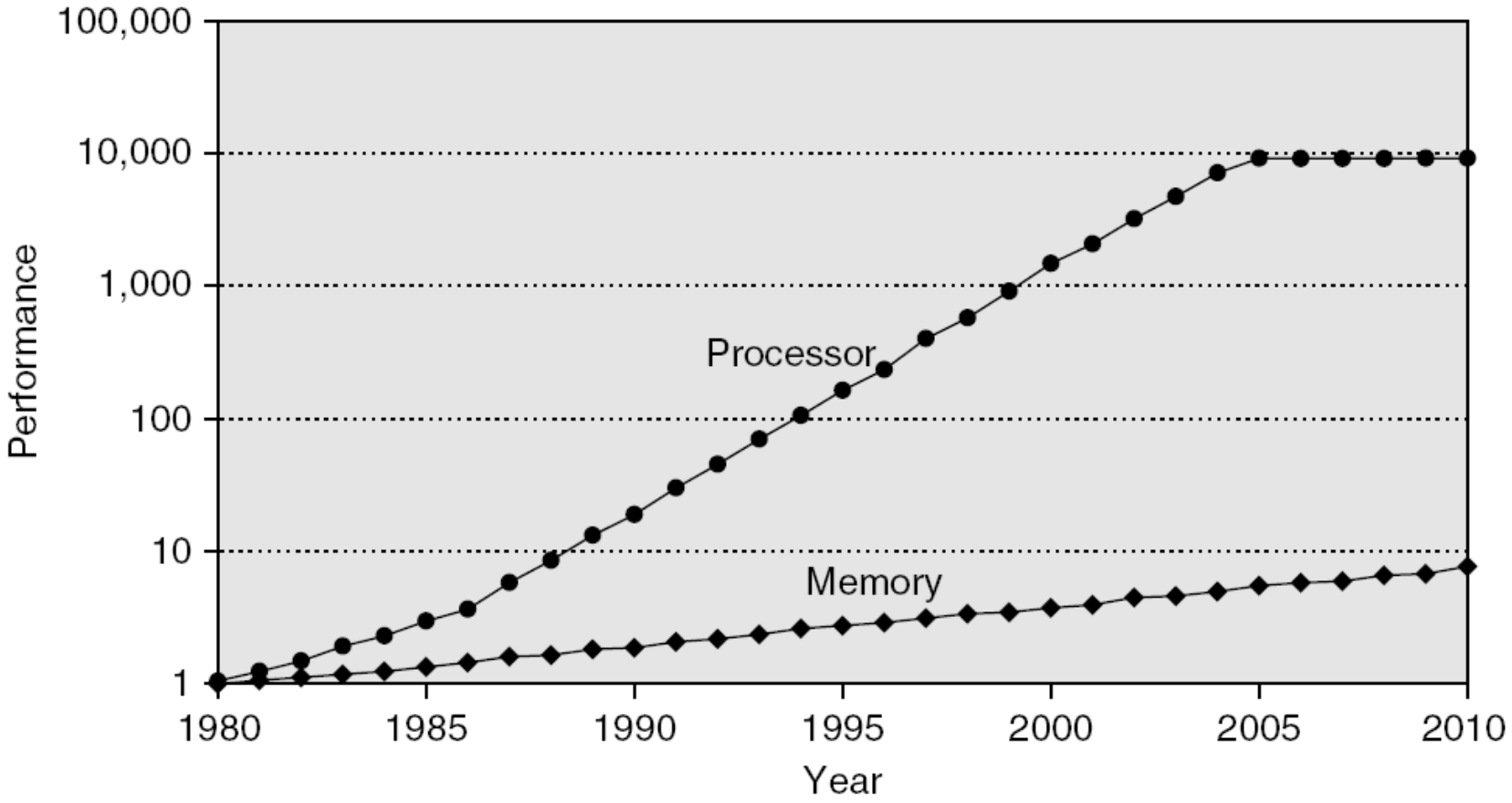
- Computer Architecture: A Quantitative Approach
  → Chapter 2, Appendix B
- Computer Organization and Design: The Hardware/Software Interface
  → Chapter 5

Memory Technology – Overview

- Static RAM (SRAM)
  → 0.5ns – 2ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  → 20ns – 30ns, $10 – $50 per GB
- Magnetic disk
  → 5ms – 20ms, $0.20 – $2 per GB

- Ideal memory
  → Access time of SRAM
  → Capacity and cost/GB of disk

The "Memory Wall"



Processor mem accesses/sec vs *DRAM accesses/sec*

The "Memory Wall" – A Multi-Core Case

- Aggregate peak bandwidth grows with # cores:
  - → Intel Core i7 can generate two references per core per clock
  - → Four cores and 3.2 GHz clock

    25.6 billion 64-bit data references/second +

    12.8 billion 128-bit instruction references

    = 409.6 GB/s!
- DRAM bandwidth is only 6% of this (25 GB/s)
- Requires:
  - **Multi-port, pipelined caches**
  - **Two levels of cache per core**
  - **Shared third-level cache on chip**

Principle of Locality – Review

- Programs often access a small proportion of their address space at any time
- Temporal locality
  - → Items accessed recently are likely to be accessed again soon
  - → e.g., instructions in a loop, induction variables
- Spatial locality
  - → Items near those accessed recently are likely to be accessed soon
  - → E.g., sequential instruction access, array data

Principle of Locality – Review

- Identify Temporal and spatial locality

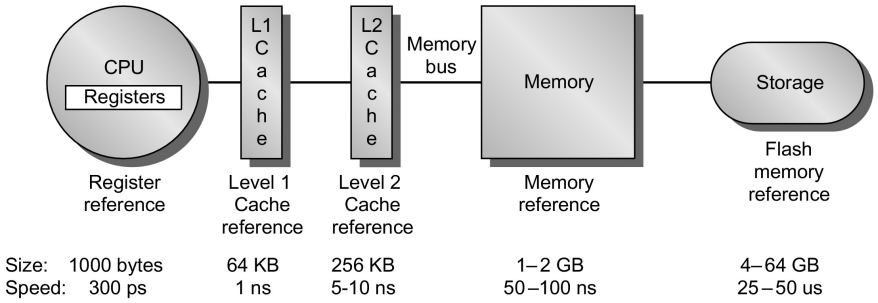
```
int sum = 0;
int x[1000];

for (int c = 0; c < 1000; c++) {
   sum += x[c];
   x[c] = 0;
}
```

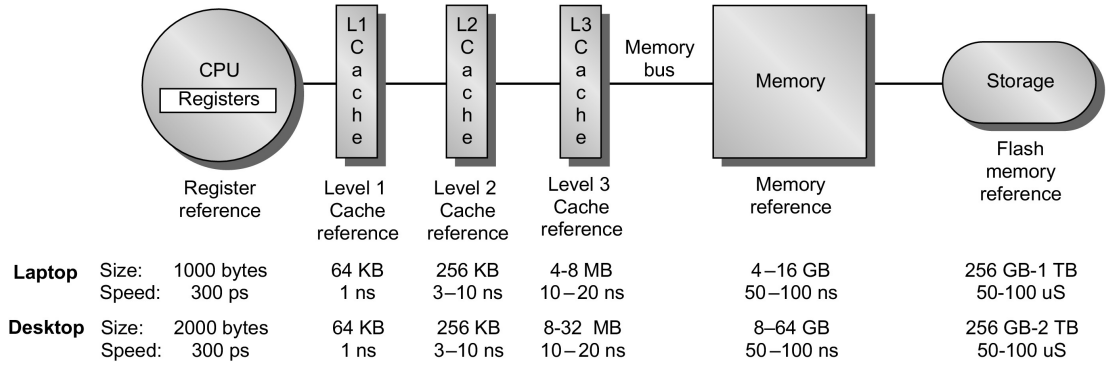
Memory Hierarchy

- Programmers want unlimited amounts of memory with low latency
- Fast memory technology is more expensive per bit than slower memory
- Solution: organize memory system into a hierarchy
  - → Entire addressable memory space available in largest, slowest memory
  - → Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - → Gives the illusion of a large, fast memory being presented to the processor

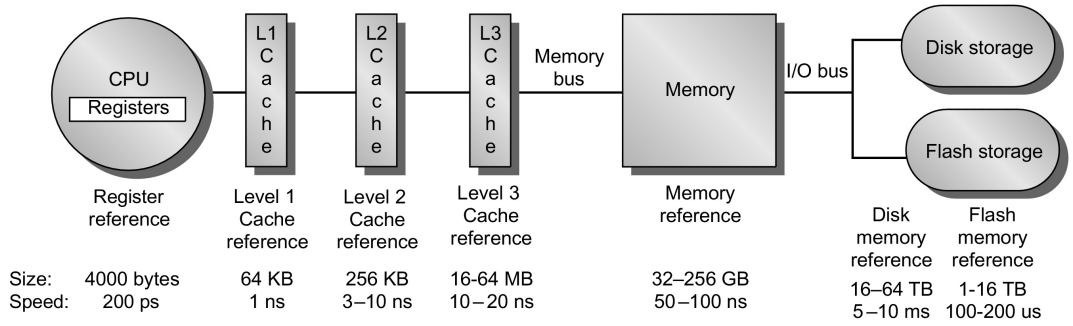
Memory Hierarchy



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

Energy Consumptions

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

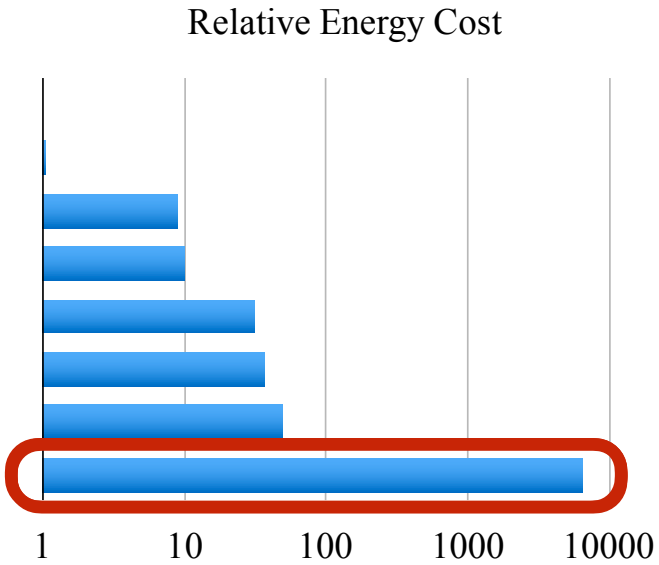


Figure 1: Energy table for 45nm CMOS process. Memory access is 2-3 orders of magnitude more energy expensive than arithmetic operations.

Song Han, FPGA'17 talk, "Deep Learning – Tutorial and Recent Trends"

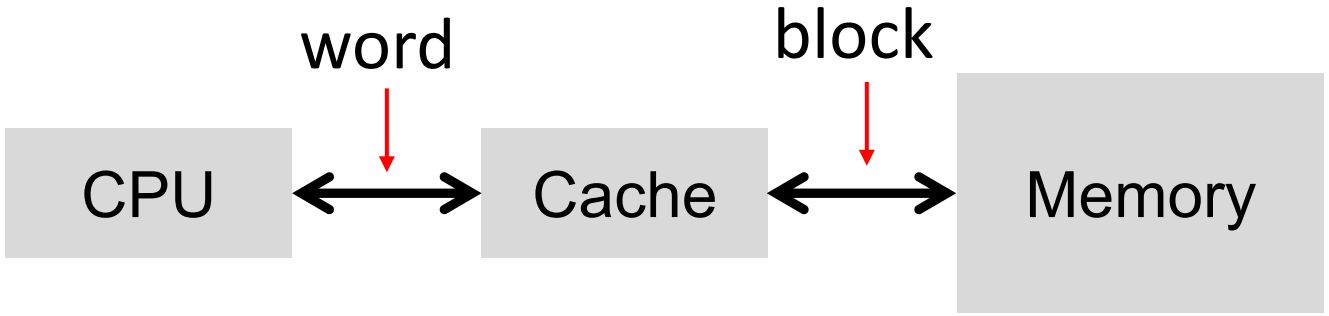
Memory Hierarchy Questions

- **Block placement** - where Can a block be placed in the upper level?
- **Block identification** – how to find a block in the upper level?
- **Block replacement** - which block should be replaced on a miss?
- **Write strategy** – how to handle writes?

Cache

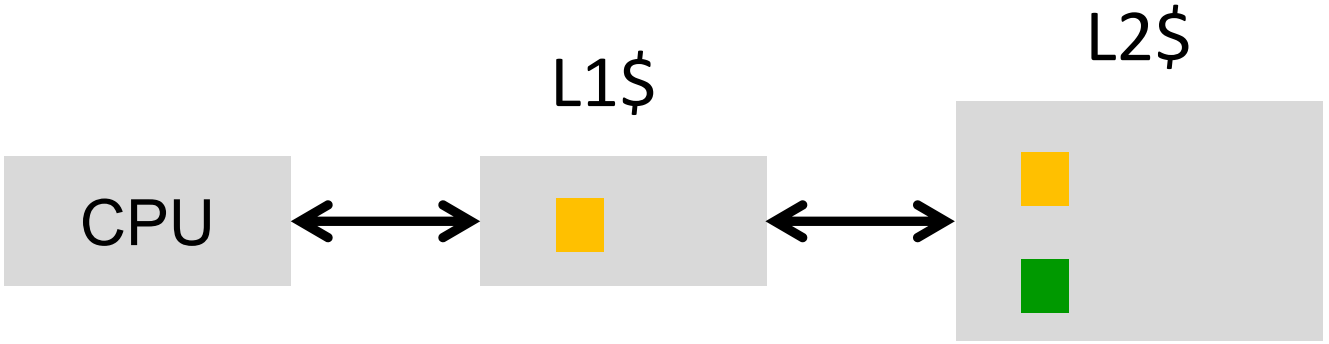
Basics

- When a word is found in cache --> cache hit.
- When a word is not found in the cache, a *miss* occurs:
  - → Fetch word from lower level in hierarchy, requiring a higher latency reference
  - → Lower level may be another cache or the main memory
  - → Also fetch the other words contained within the **block**
    - ➤ Takes advantage of spatial locality
  - → Place block into cache in any location within its ***set***, determined by address

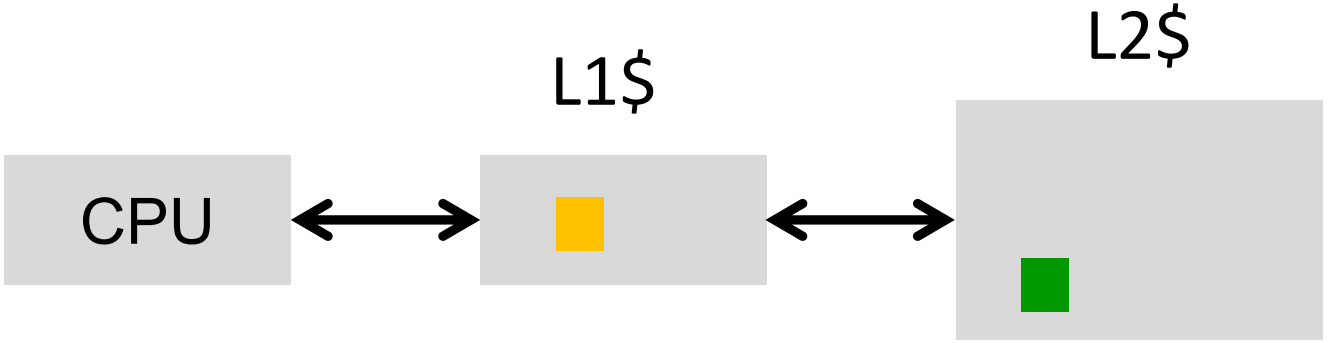


Basics

- Inclusive cache

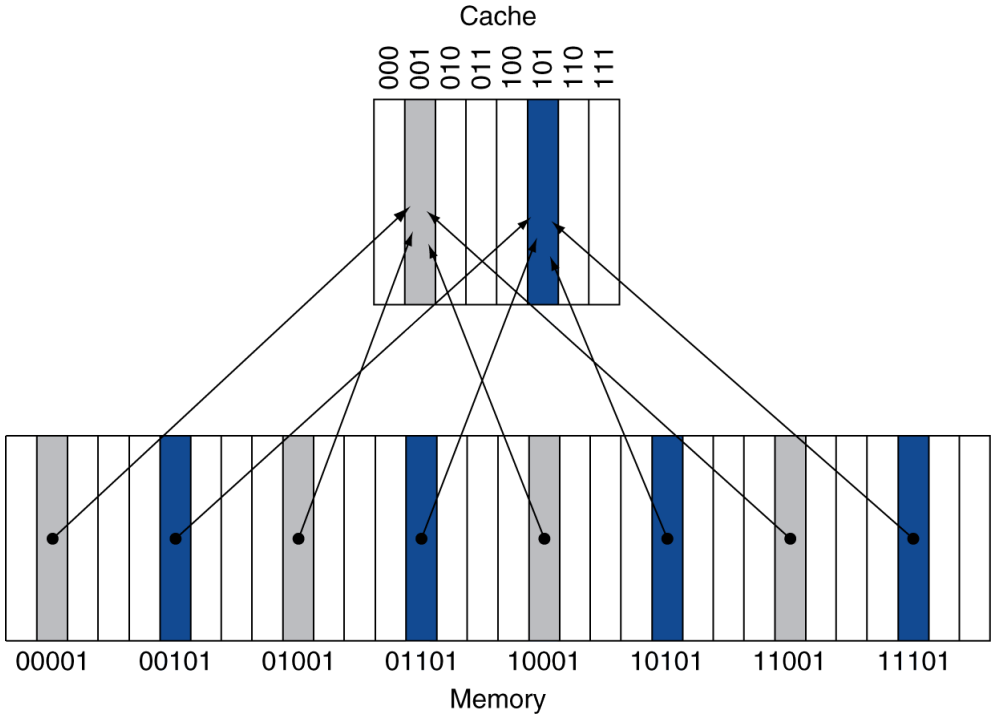


- Exclusive cache

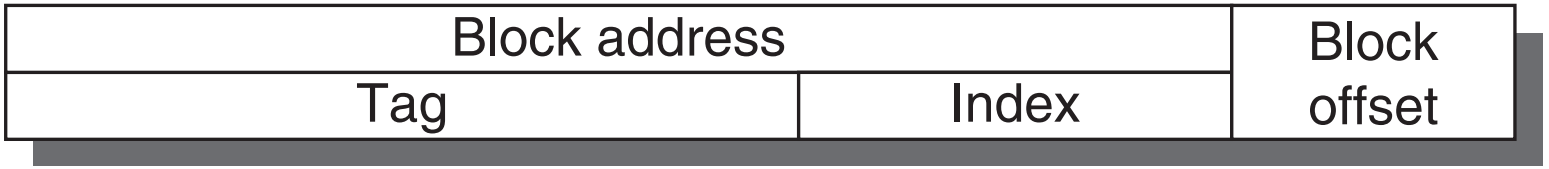


Direct Mapped Cache

- Only one choice **(Block address) MOD (#Blocks in cache)**



- *#Blocks is a power of 2*
- *Use low-order address bits to access bytes in a block*



Tags and Valid Bits

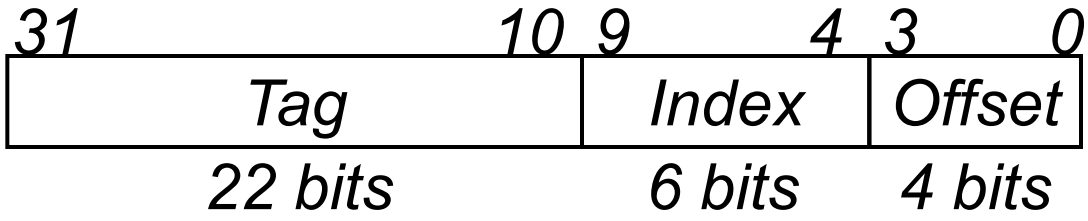
- One cache line ←Multiple memory blocks
- How do we know which particular memory block is stored in a cache location?
  - → Store **tag** as well as the data
  - → Tag is the high-order bits of a block address
- What if there is no data in a location?
  - → **Valid** bit: 1 = present, 0 = not present
  - → Initially 0

Example: Larger Block Size

- Cache: 64 blocks, 16 bytes/block
  - → To what block number does address 1200 (decimal) map?
  - → Assume 32-bit address

Example: Larger Block Size

- Cache: 64 blocks, 16 bytes/block
  → To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Cache block index = 75 modulo 64 = 11



Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Gray area is what's actually included in cache

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	**1**	**10**	**Mem[10110]**
111	0		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	0		
001	0		
010	**1**	**11**	**Mem[11010]**
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

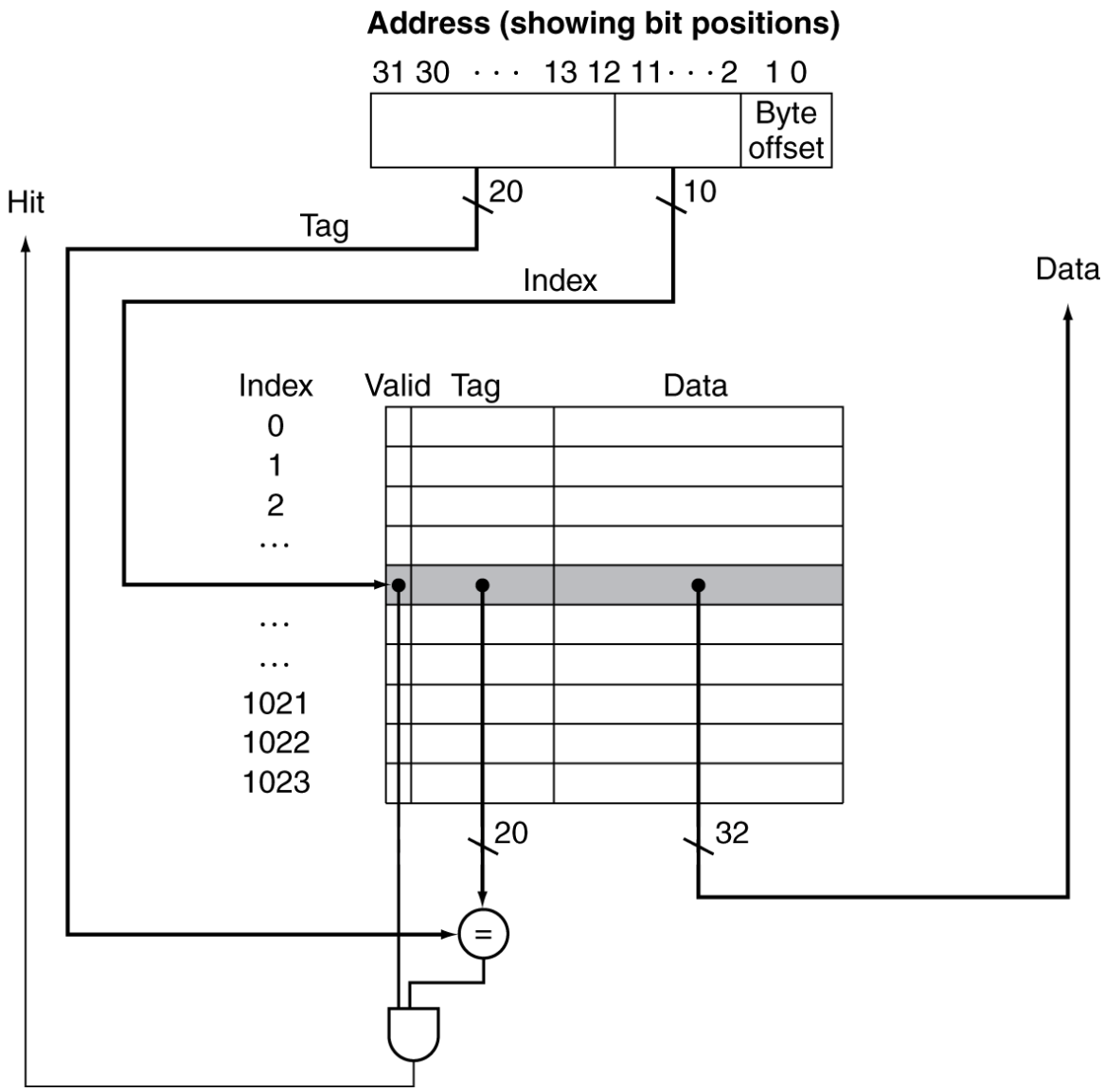
Index	V	Tag	Data
000	**1**	**10**	**Mem[10000]**
001	0		
010	1	11	Mem[11010]
011	**1**	**00**	**Mem[00011]**
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	**1**	**10**	**Mem[10010]**
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Address Subdivision



Cache Misses

- On cache *hit*, CPU proceeds normally
- On cache *miss*
  - → Stall the CPU
  - → Fetch a block from next level of mem. hierarchy
  - → Instruction cache miss
    - ➤ Restart instruction fetch
  - → Data cache miss
    - ➤ Complete data access
- Note that speculative and multithreaded processors may execute other instructions during a miss
  - → Reduces performance impact of misses

Cache Misses

- *Miss rate*
  - → Fraction of cache access that result in a miss
- *Miss Penalty* – time to access lower level memory
- Causes of misses
  - → *Compulsory*
    - ➤ First reference to a block
  - → *Capacity*
    - ➤ Blocks discarded and later retrieved
  - → *Conflict*
    - ➤ Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache
    - ➤ Only happen in direct-mapped or set associative caches

Measuring Cache Performance

CPU time = (CPU cycles + mem stall cycles) * cycle time

→ CPU cycles = instruction cycles + cache hit time

→ Memory stall cycles from cache misses

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$= \text{IC} \times \underbrace{\frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}}_{\text{Stall cycles per instruction}}$$

Cache Performance – Example

- Given
  - → I-cache miss rate = 2%
  - → D-cache miss rate = 4%
  - → Miss penalty = 100 cycles
  - → Base CPI (ideal cache) = 2
  - → Load & stores are 36% of instructions
- Miss cycles per instruction
  - → I-cache: 0.02 × 100 = 2
  - → D-cache: 0.36 × 0.04 × 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - → Ideal CPU is 5.44/2 =2.72 times faster

Cache Performance - Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)

$$AMAT = Hit\ time + Miss\ rate \times Miss\ penalty$$

- Example
  - → CPU with 1ns clock, hit time = 2 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - → AMAT = 2 + 0.05 × 20 = 3 cycles

Question: why not AMAT = hit time * (1-Miss rate) + Miss rate * Miss penalty?

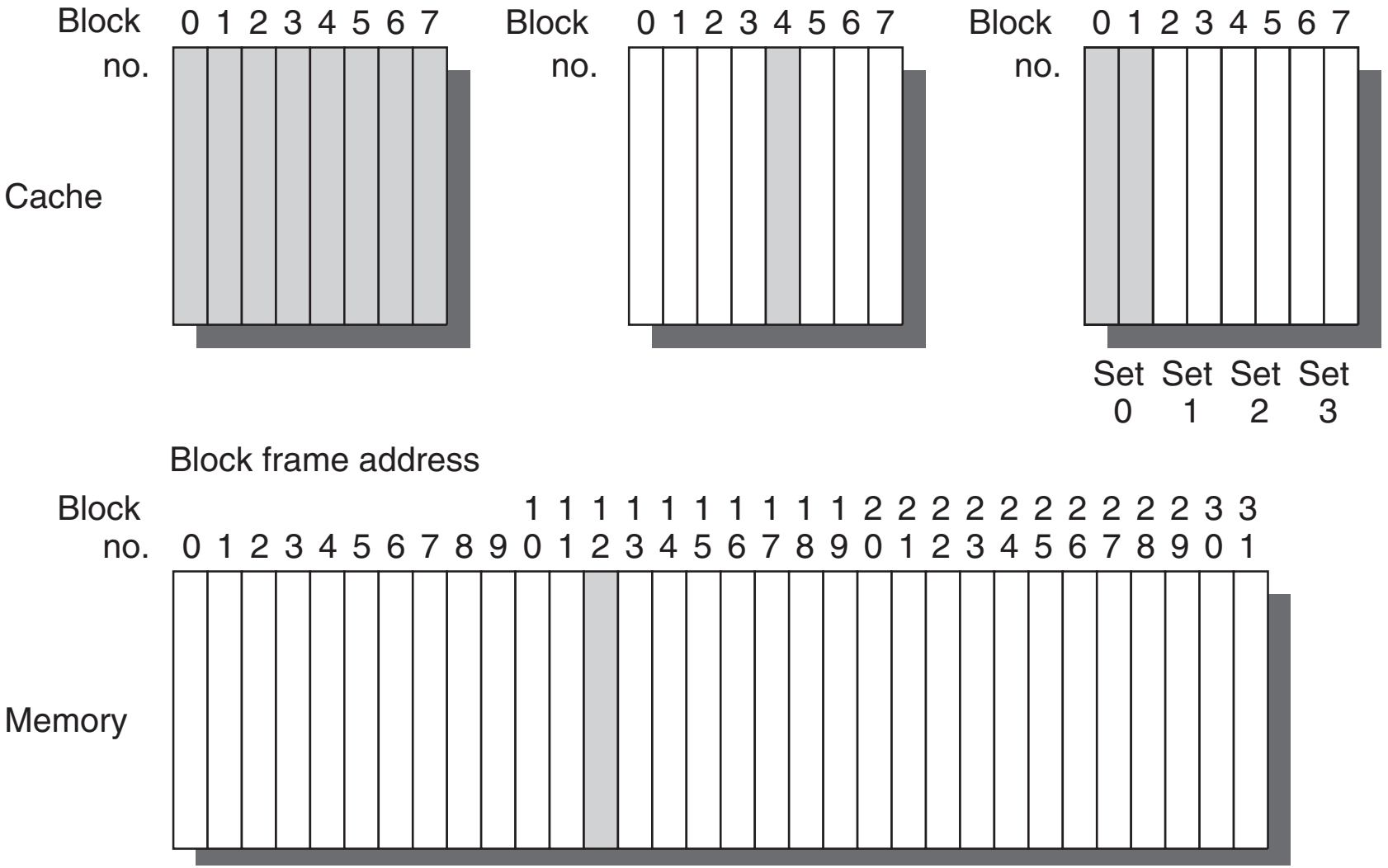
Performance Summary

- As CPU performance increased
  - → Miss penalty becomes more significant
- Increasing clock rate, and decreasing base CPI
  - → Memory stalls account for more CPU cycles
  - → Greater proportion of time spent on memory stalls
- Can’t neglect cache behavior when evaluating system performance
  - → What does Amdahl’s law tell us?

Associative Caches

- Reduce *conflict* misses
- *Fully associative*
  - → Allow a given block to go in any cache entry
  - → Requires all entries to be searched at once
  - → Comparator per entry – expensive (area)
- *n-way set associative*
  - → Each set contains *n* entries
  - → Block number determines which set
    - ➤ (Block address) MOD (#Sets in cache)
  - → Search all entries in a given set at once
  - → *n* comparators (less expensive)
  - → Direct-mapped = 1-way associative

Associative Cache Example



Associativity Example

- Compare 4-block caches
  - → Direct mapped, 2-way set associative, fully associative
  - → Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

5 misses (their types?)

Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Spectrum of Associativity

- For a cache with 8 entries

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

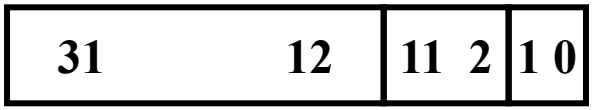
Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Size of Tags vs Associativity

- Address size = 32 bit
- Data/word size = 32 bit
- Cache size = 16KB
- **Direct mapping**
  - → Tag bits = 18×2^{12}
  - → Comparators = 1
- **4-way set-associative**
  - → Tag bits= $4 \times 2^{10} \times 20 = 20 \times 2^{12}$
  - → Comparators = 4
- **Fully associative**
  - → Tag bits = 30×2^{12}
  - → Comparators = 2^{12}

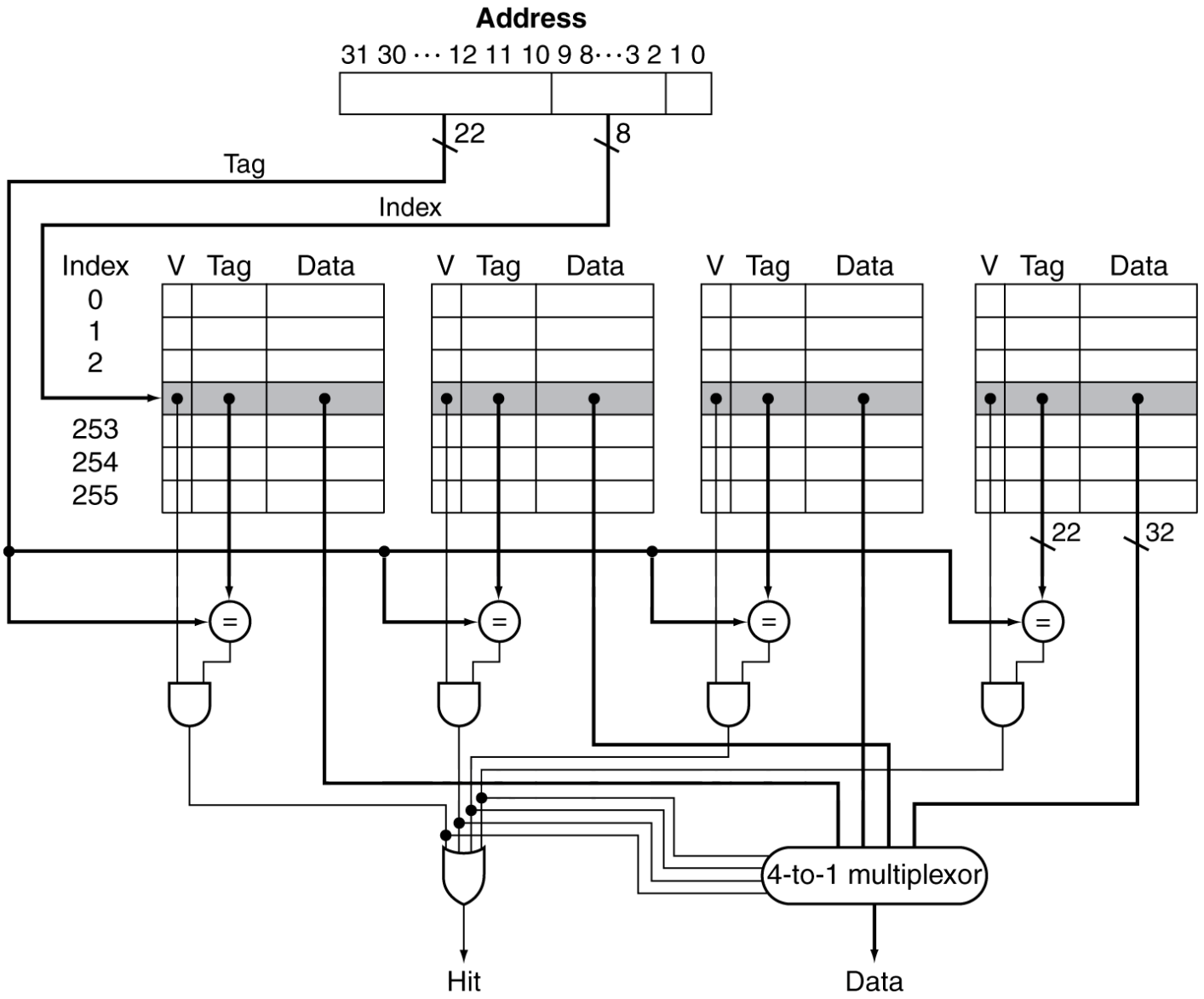
$$\#\text{blocks} = 2^{12}$$



Size of Tags vs Associativity

- Increasing associativity requires
  - → More tag bits per cache block
  - → More comparators, each of which is more complex
- The choice among direct, set-associative and fully-associative mapping in any memory hierarchy will depend on
  - → cost of miss vs cost of implementing associativity, both in time and in extra hardware

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
  - → Prefer non-valid entry, if there is one
  - → Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - → Choose the one unused for the longest time
    - ➤ Simple for 2-way, manageable for 4-way, too hard otherwise
    - ➤ FIFO approximates LRU.
- Random
  - → Gives approximately the same performance as LRU for high associativity

Write Policy – Write-Through

- Update cache and memory
- Easier to implement
- Writes take longer – wait for mem update to complete
  - → e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - ➤ Effective CPI = 1 + 0.1×100 = 11
- Solution: **write buffer**
  - → Holds data waiting to be written to memory
  - → CPU continues immediately
    - ➤ Only stalls on write if write buffer is already full
  - → Write buffer is freed when a write to memory is finished

Write Policy – Write-Back

- Just update the block in cache
  - → Keep track of whether each block is dirty - dirty bits
  - → Cache and memory would be inconsistent
- When a dirty block is replaced
  - → Write it back to memory
- Write speed is faster
  - → One memory update for multiple writes.
  - → Power saving.
- Write buffer can also be used

Write Allocation

- What should happen on a write miss?
  - → Write allocate
  - → No write allocate
- Alternatives for write-through
  - → Allocate on miss: fetch the block
  - → Write around: don’t fetch the block
- For write-back
  - → Usually fetch the block
  - → Act like read misses

Write Miss Policies – Example

write M[100]
write M[100]
read M[200]
write M[200]
write M[100]

- Number of misses if the following write policy is used.
  → No-write-allocate
  → Write-allocate

Cache Performance – Example

Perfect CPI = 1.6,
cycle time = 0.35ns
Hit time(DM) = 1 cycle
Miss rate (DM) = 0.021
Hit time (2-way) = 1.35 cycles
Miss rate (2-way) = 0.019
Miss penalty = 65 ns
Avg mem. req/inst = 1.4

Which cache is better?

Basic Cache Optimizations

Cache Performance – Review

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Any optimization should consider its impact on all three factors

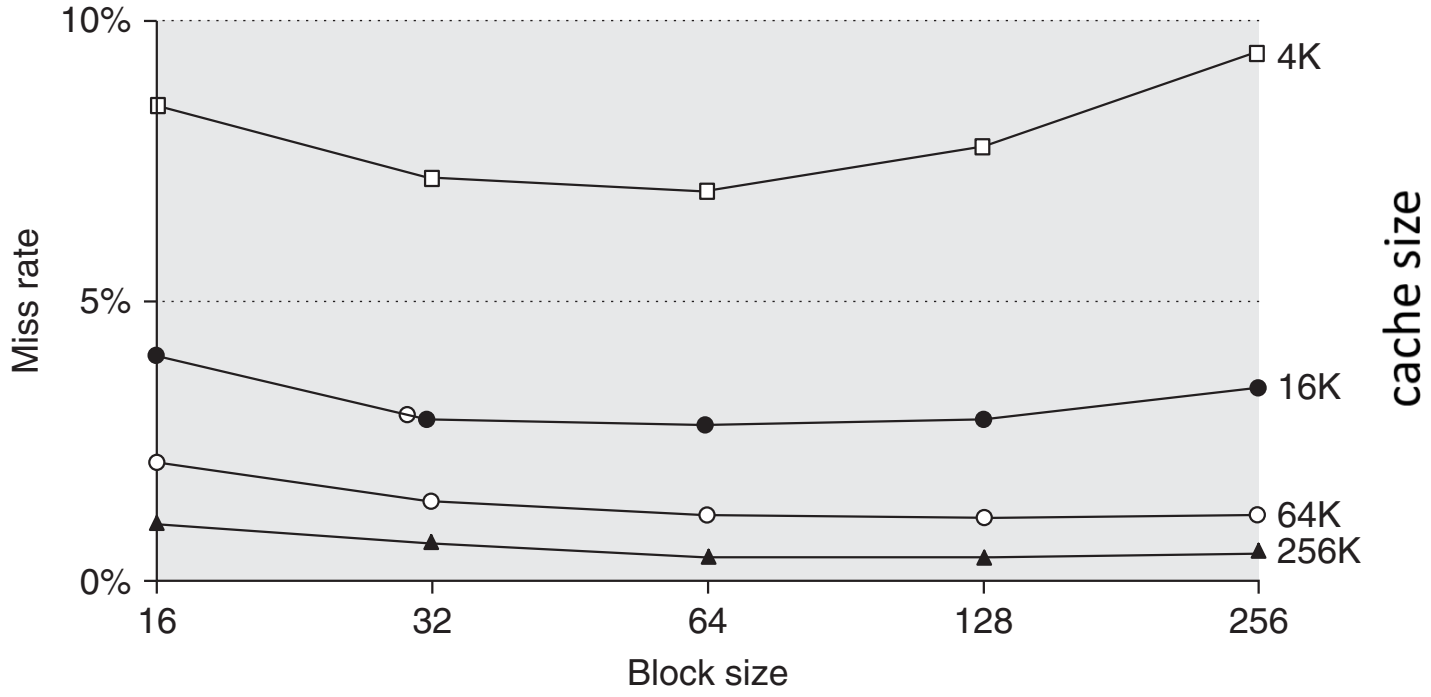
- Hit time
- Miss rate
- Miss penalty

Six Basic Cache Optimizations

- Larger block size
  - → Reduces compulsory misses
  - → Increases capacity and conflict misses, increases miss penalty
- Larger total cache capacity to reduce miss rate
  - → Increases hit time, increases power consumption
- Higher associativity
  - → Reduces conflict misses
  - → Increases hit time, increases power consumption
- Multi-level cache to reduce miss penalty
  - → Reduces overall memory access time
- Giving priority to read misses over writes
  - → Reduces miss penalty
- Avoiding address translation in cache indexing
  - → Reduces hit time

Optimization 1 – Larger Block Size

- Reduce compulsory misses due to spatial locality
- May increase conflict/capacity misses
- Increase miss penalty



Optimization 1 – Larger Block Size

- Reduce compulsory misses due to spatial locality
- May increase conflict/capacity misses
- Increase miss penalty

Block size	Miss penalty	Cache size			
		4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

AMAT

Optimization 1 – Block Size Selection

- Determined by lower level memory
- High latency and high bandwidth – larger block size
- Low latency and low bandwidth – small block size

Optimization 2 – Larger Cache

- Reduce capacity misses
- May increases hit time
- Increases power consumption

Optimization 3 – Higher Associativity

- Reduces conflict misses
- Increases hit time
- Increases power consumption

	Associativity			
Cache size (KB)	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Optimization 3 – Higher Associativity

- Higher Associativity -> higher hit time
- Larger miss penalty rewards higher associativity
- Example:

Assume 4KB cache

AMAT(1-way) = 1 + 0.098 * miss penalty

AMAT(2-way) = 1.36 + 0.076 * miss penalty

Consider

1. miss penalty = 25 cycles
2. miss penalty = 100 cycles

Optimization 4 – Multilevel Cache

- L1 cache – small to keep hit time fast
- L2 cache – capture as many L1 misses & lower miss penalty
- Local miss rates – L1/L2 miss rates
- Global miss rates – miss rate(L1) * miss rate(L2)
- AMAT = hit(L1) + miss rate(L1) * miss penalty(L1)
  → miss penalty(L1) = hit(L2) + miss rate(L2) * miss penalty(L2)
- Mem Stall cycles/inst = miss/inst(L1) * hit(L2) +
  misses/inst(L2) * miss penalty(L2)

Optimization 4 – Multilevel Cache

1000 mem accesses -> L1 misses = 40, L2 misses = 20

Q1: Local/global miss rates?

Hit(L1) = 1 cycle
Miss penalty(L2) = 200 cycles,
Hit(L2) = 10 cycles
Avg mem/inst = 1.5

Q2: AMAT?

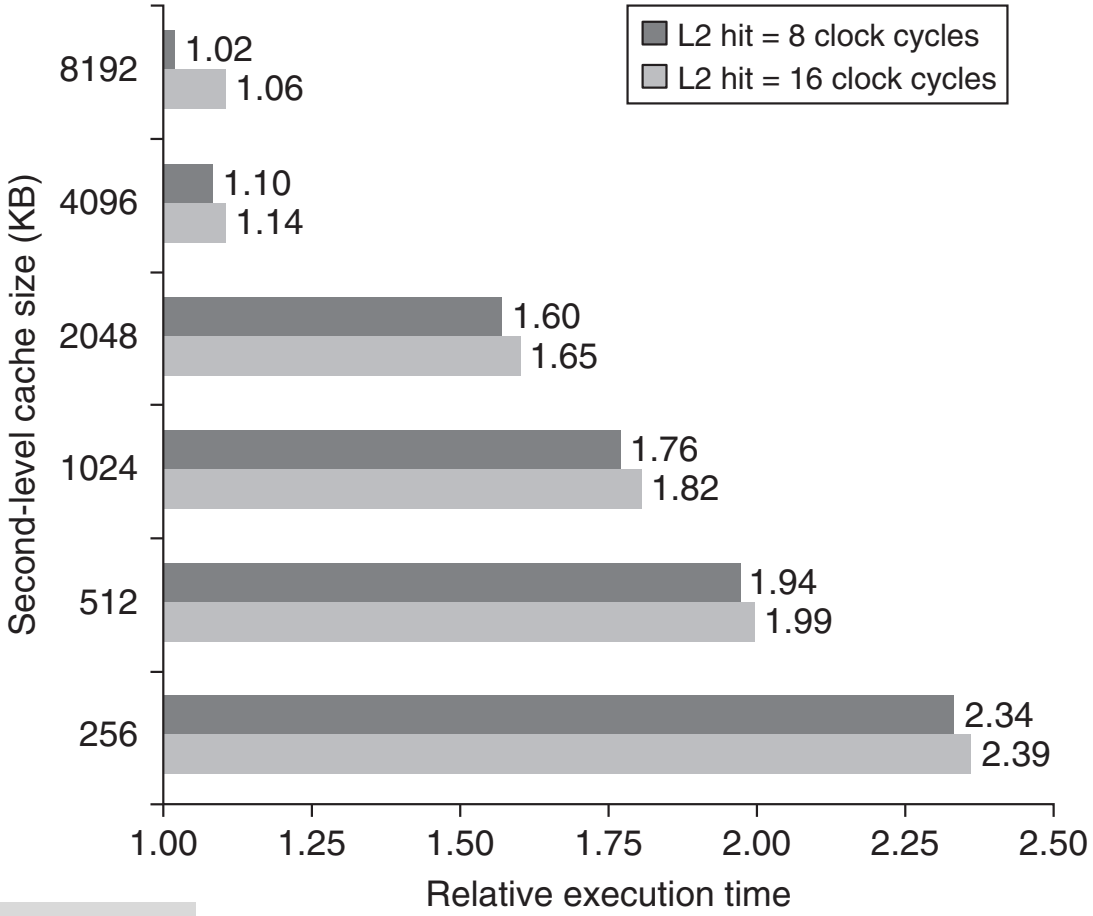
Q3: Avg stalls/inst?

Optimization 4 – Multilevel Cache

- L1 hit time affects CPU speed
  → Small and fast L1
- Speed of L2 affects L1 miss penalty
  → Large L2 with higher associativity

Optimization 4 – Multilevel Cache

- Impact of L2 cache size on execution speed



Optimization 5 – Giving Priority to Read Misses over Writes

- Reduces miss penalty. See example below.

```
SW R3, 512(R0)     ;M[512] ← R3     (cache index 0)
LW R1, 1024(R0)    ;R1 ← M[1024]    (cache index 0)
LW R2, 512(R0)     ;R2 ← M[512]     (cache index 0)
```

- Write-through cache with write buffers suffers from RAW conflicts with main memory reads on cache misses:
  - → Write buffer holds updated data needed for the read.
  - → Alt #1 – wait for the write buffer to empty, increasing read miss penalty (in old MIPS 1000 by 50%).
  - → Alt #2 – Check write buffer contents before a read; if no conflicts, let the memory read go first.

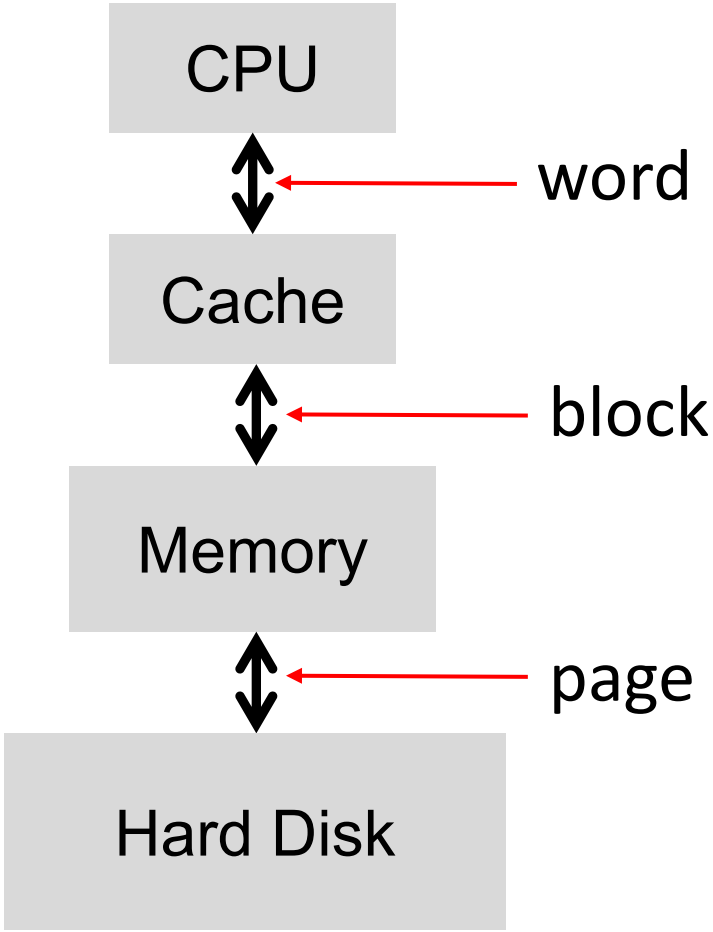
Optimization 5 – Giving Priority to Read Misses over Writes

- Reduces miss penalty. See example below.

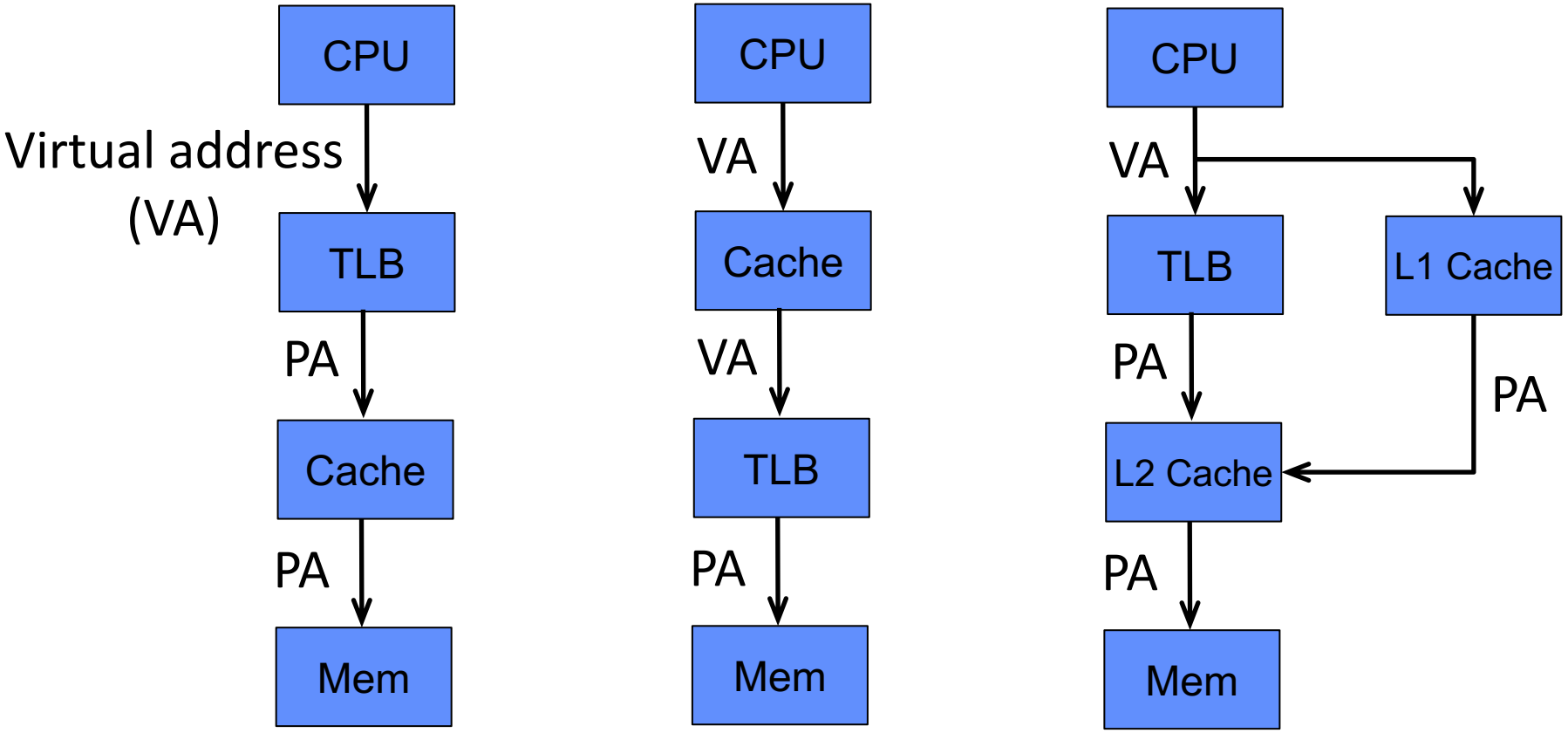
```
SW R3, 512(R0)     ;M[512] ← R3     (cache index 0)
LW R1, 1024(R0)    ;R1 ← M[1024]    (cache index 0)
LW R2, 512(R0)     ;R2 ← M[512]     (cache index 0)
```

- In a write-back cache, suppose a read miss causes a dirty block to be replaced
  - → Alt #1 – write the dirty block to memory first, then read memory.
  - → Alt #2 – copy the dirty block to a write buffer, read the new block, then write the dirty to memory.

Optimization 6 – Avoid Address Translation during Cache Indexing to Reduce Hit Time



Optimization 6 – Avoid Address Translation during Cache Indexing to Reduce Hit Time

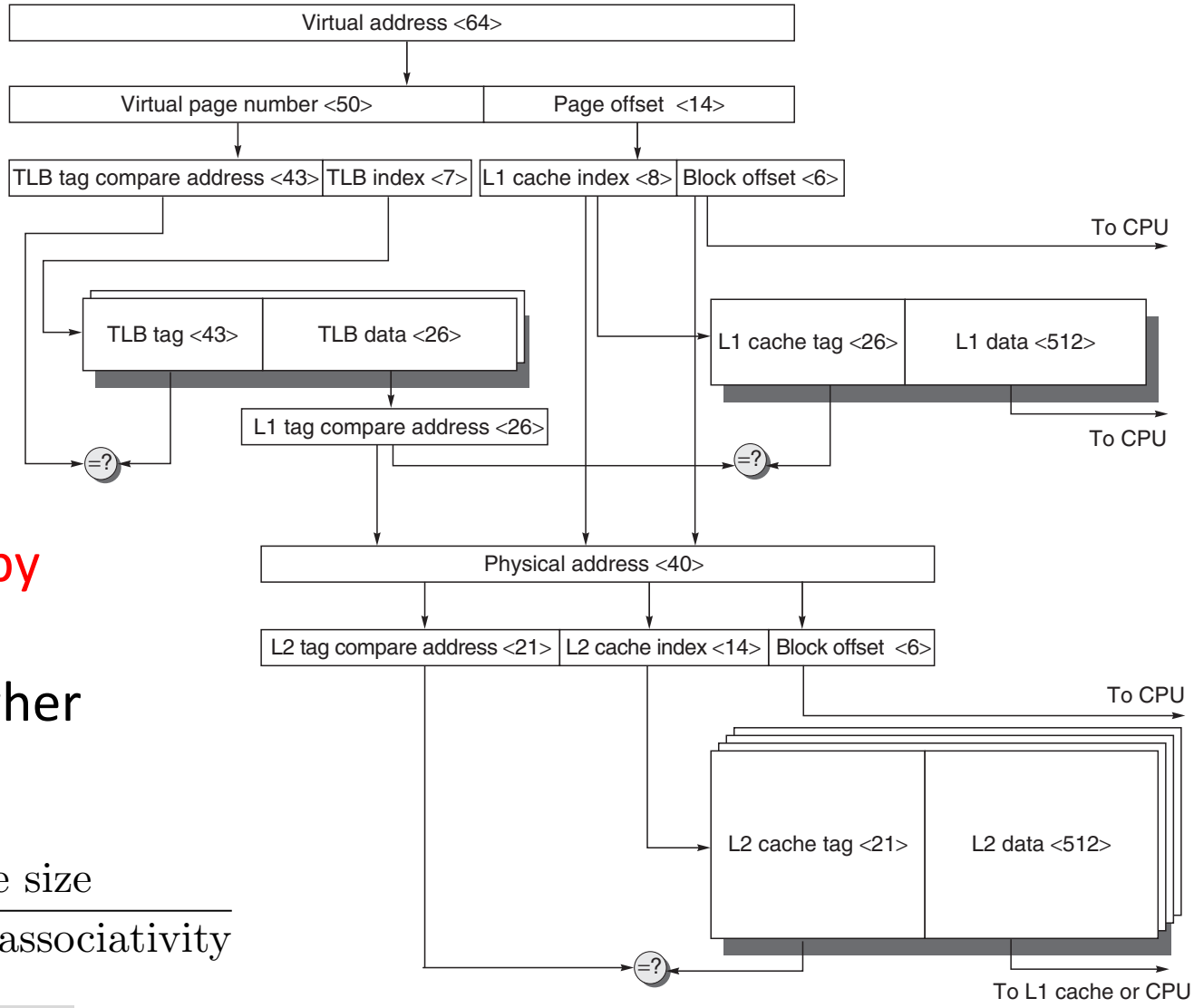


Conventional organization

Virtually addressed cache

Virtually addressed physically tagged

Optimization 6 – Avoid Address Translation during Cache Indexing to Reduce Hit Time



- Cache size limited by the page size.
- Increase size by higher associativity.

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

Advanced Cache Optimizations

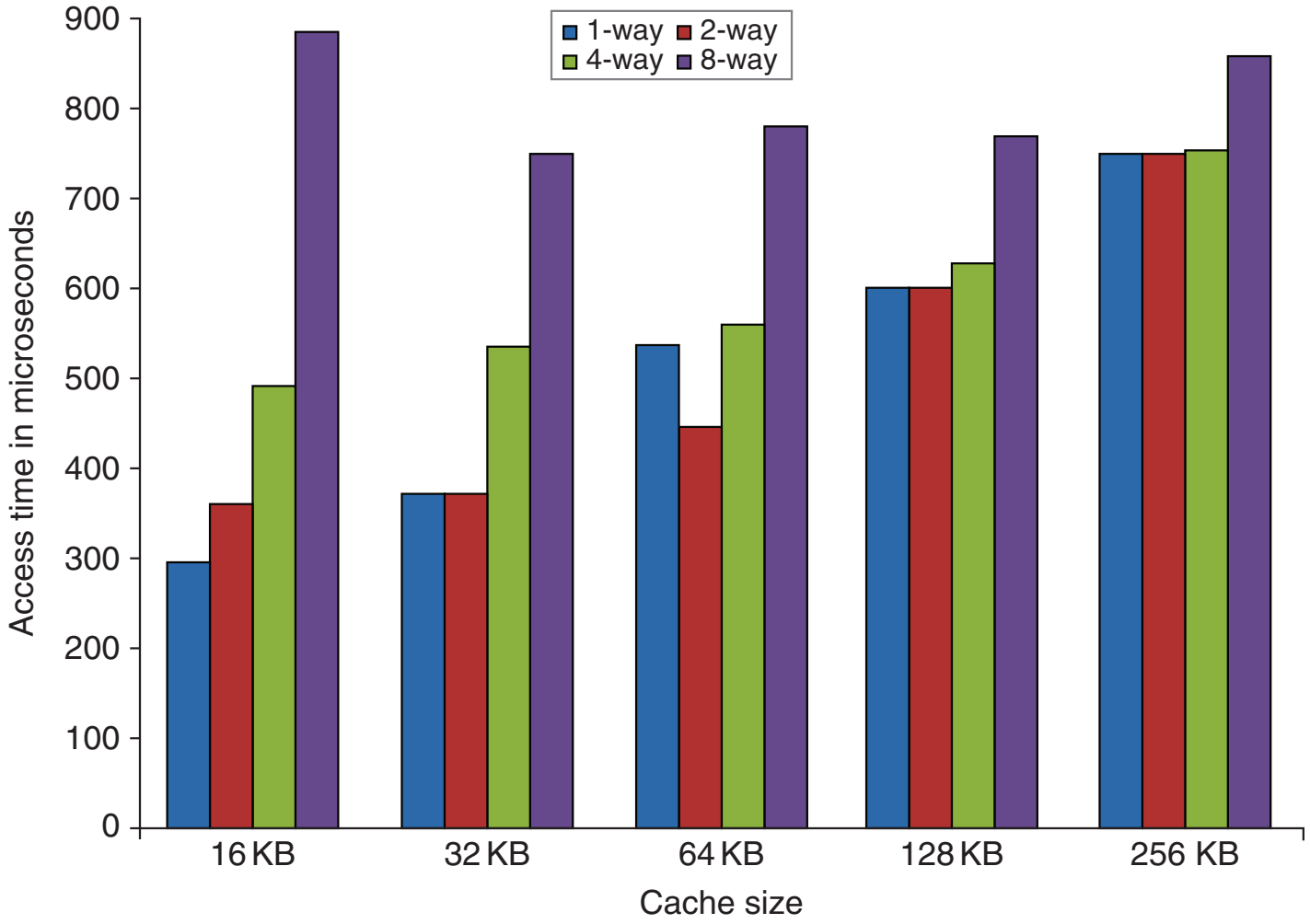
Basic Ideas

- Reduce hit time, miss rate, and miss penalty
- Increase cache bandwidth
- Reduce miss rate/penalty via parallelism

Opt 1 – Small and Simple L1 Caches

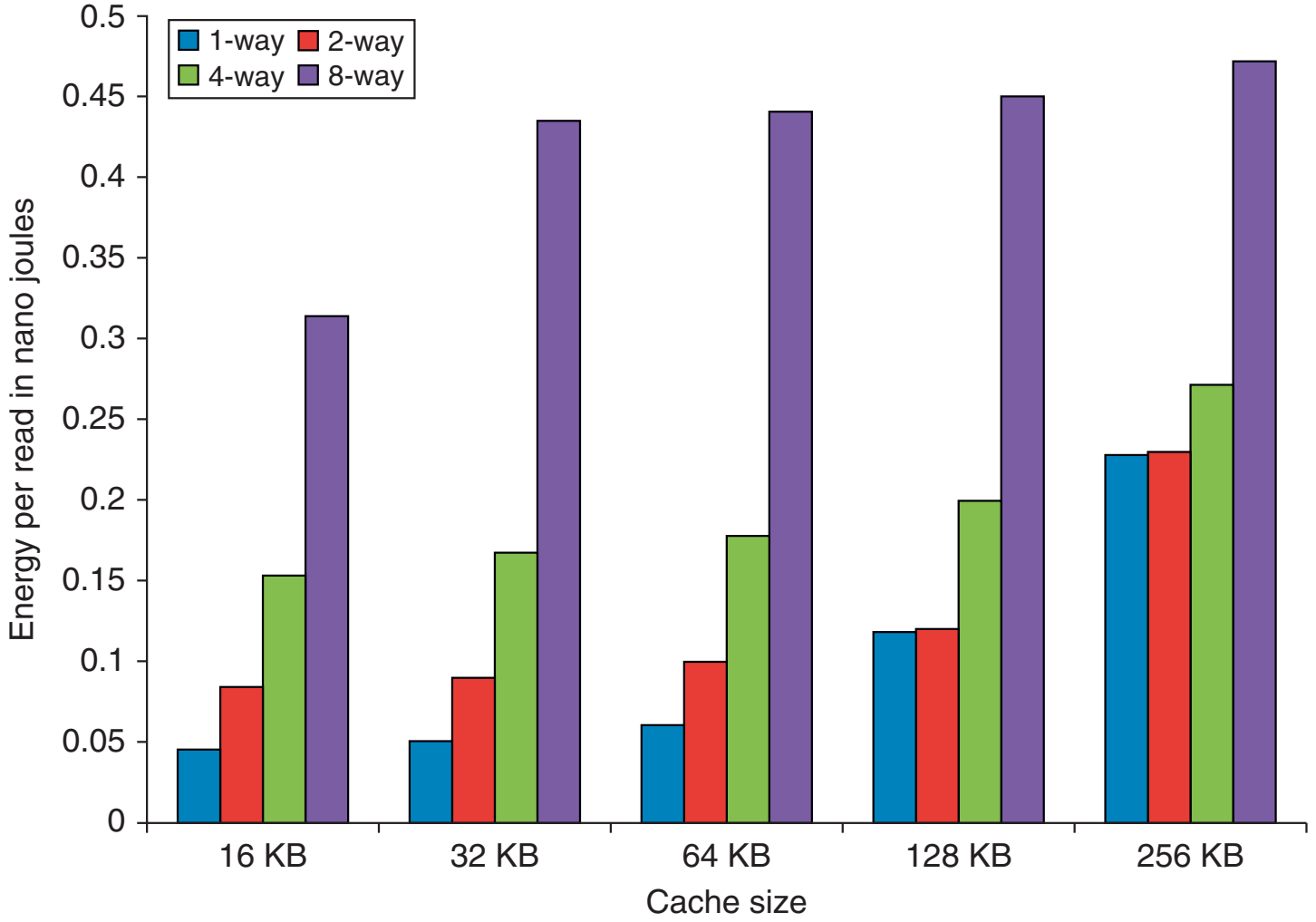
- Critical timing path in cache hit:
  - → addressing tag memory, then
  - → comparing tags, then
  - → selecting correct set
- Direct-mapped caches can overlap tag comparison and transmission of data
- Lower associativity reduces power because fewer cache lines are accessed. However,
  - → Associativity used to increase size of virtually indexed cache
  - → Higher associativity reduces conflict misses due to multithreading

Opt 1 – L1 Size and Associativity



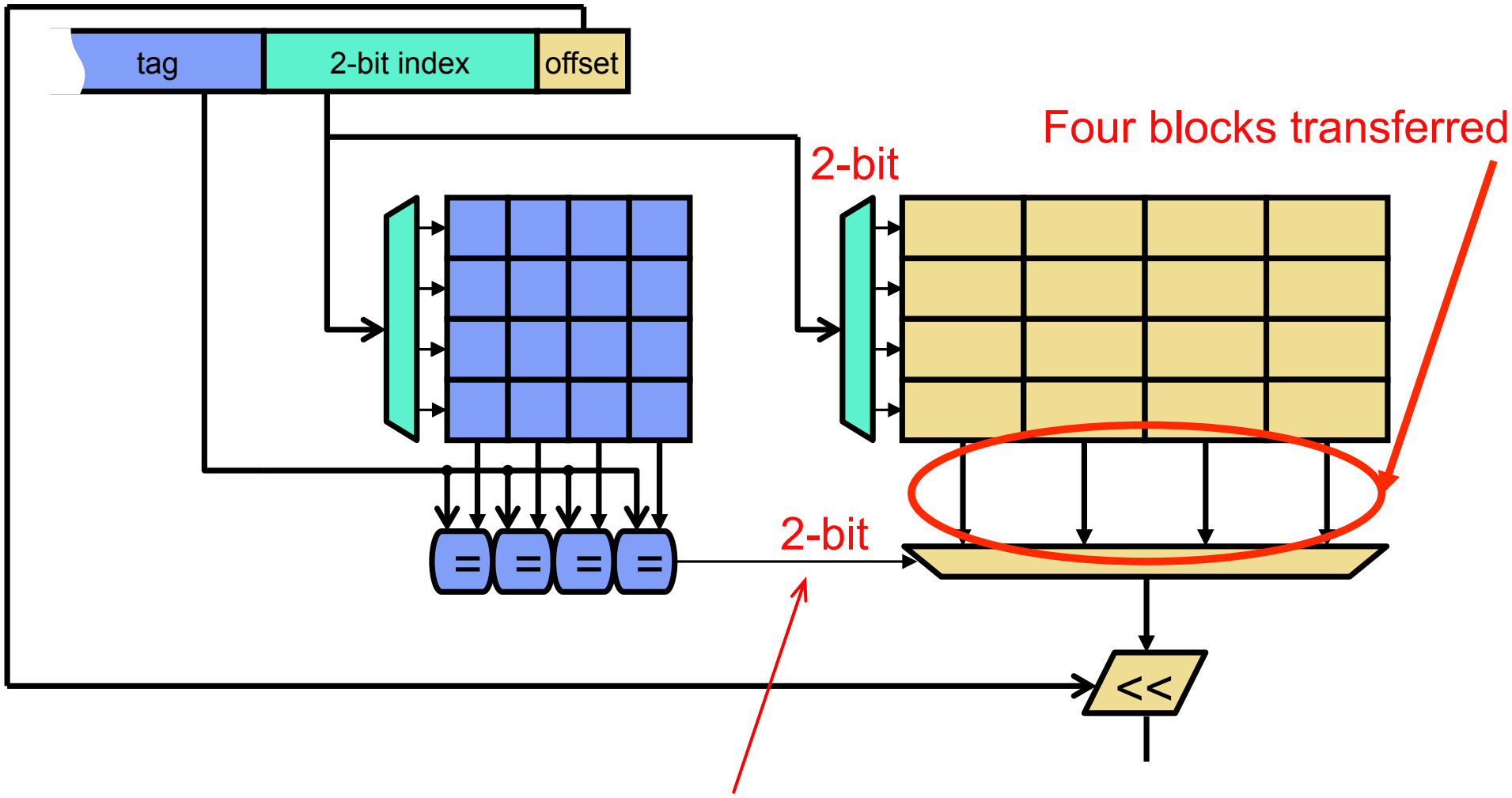
Access time increases as size & associativity increases

Opt 1 – L1 Size and Associativity



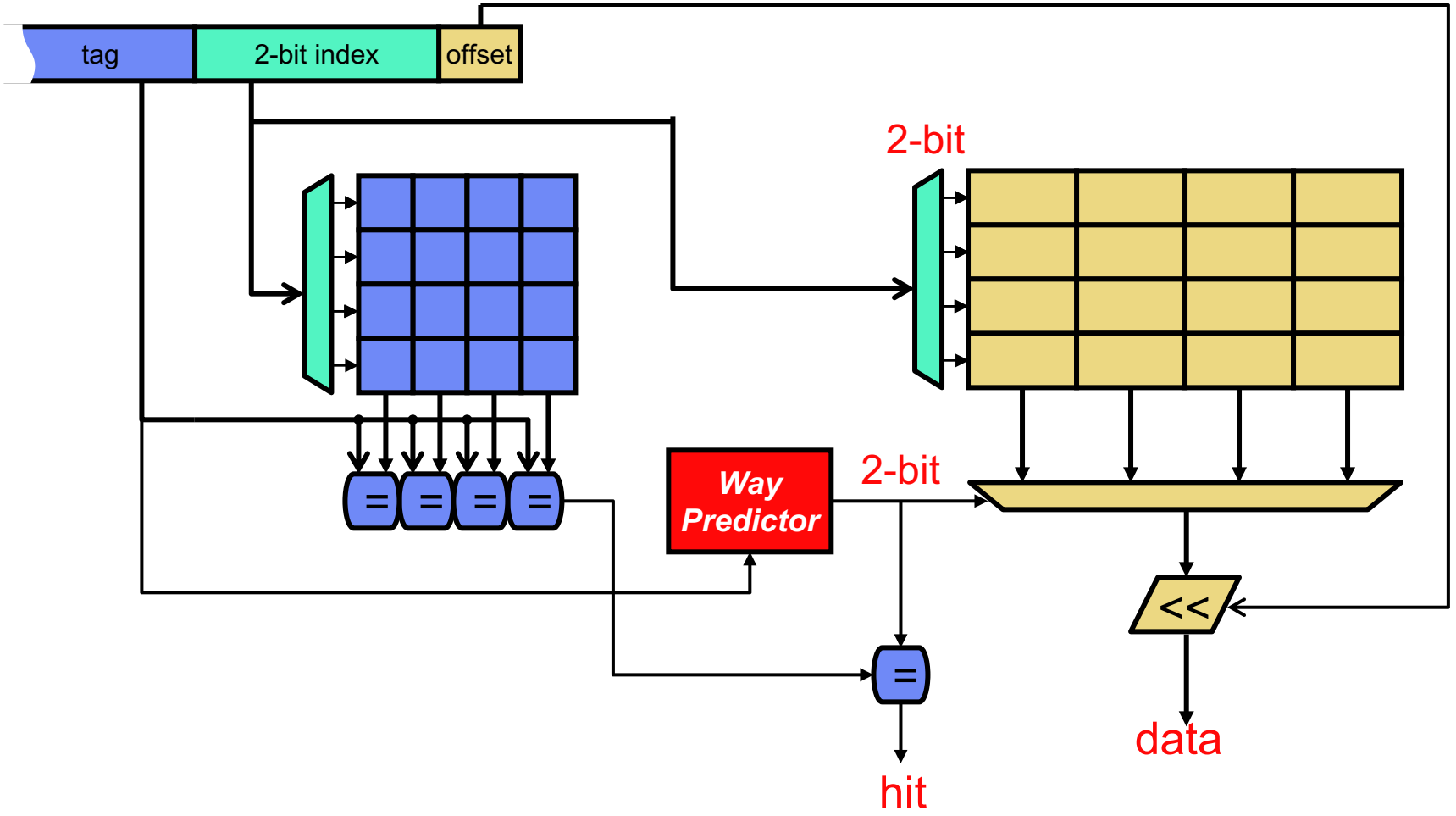
Energy per read increases as size & associativity increases

Opt 2 – Way Prediction



Way prediction set them before tag comparison is done.

Opt 2 – Way Prediction



Opt 2 – Way Prediction

- Block associated with prediction (low order tag) bits.
  - → Predicts the next block to be accessed – what locality?
  - → Multiplexer could be set early to select the predicted block, only a single tag comparison
  - → A miss results in checking the other blocks
- Prediction accuracy
  - → > 90% for two-way
  - → > 80% for four-way
  - → I-cache has better accuracy than D-cache
  - → First used on MIPS R10000 in mid-90s
- Extend to predict block as *way selection*
  - → Intends to save power consumption.
  - → Increases mis-prediction penalty

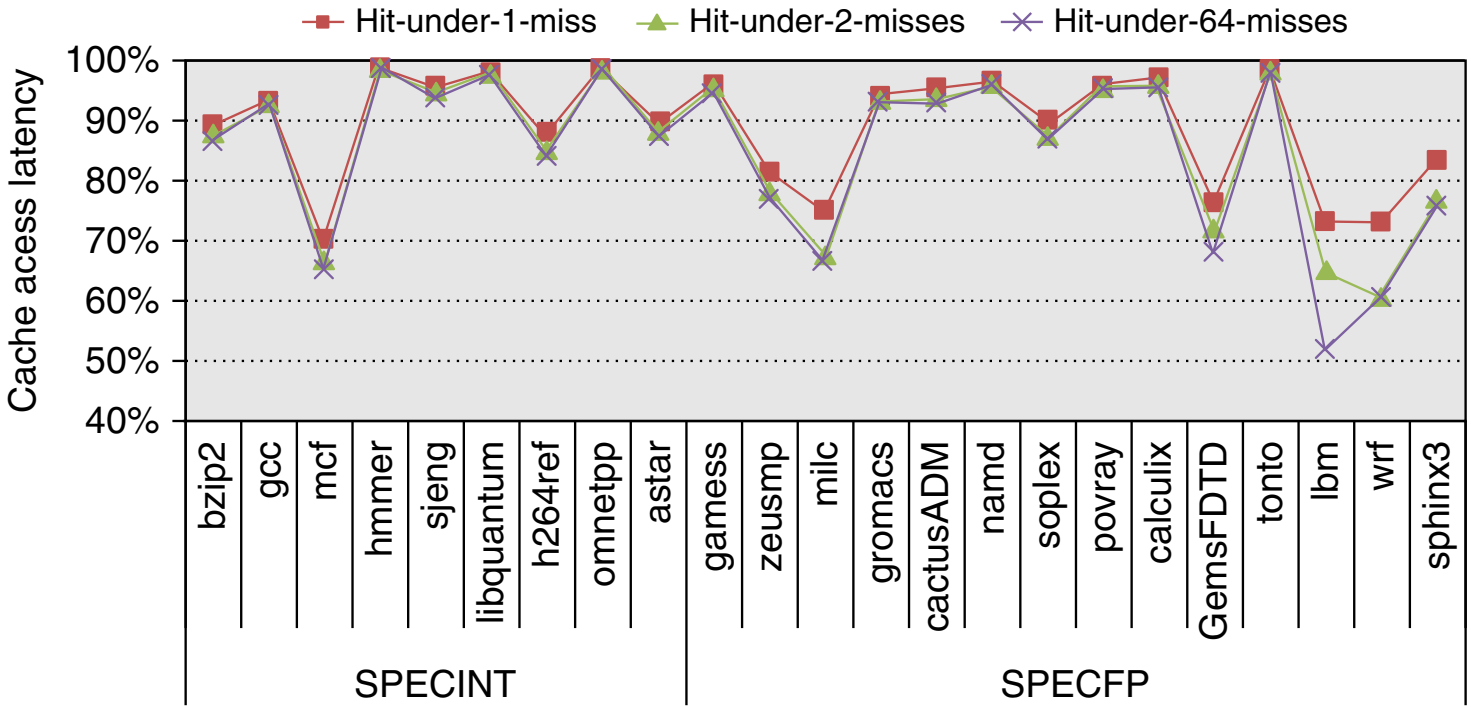
Opt 3 – Pipelining Cache

- Pipeline cache access to improve bandwidth
  - → Faster clock cycle, but slow hit time
  - → Examples:
    - ➤ Pentium: 1 cycle
    - ➤ Pentium Pro – Pentium III: 2 cycles
    - ➤ Pentium 4 – Core i7: 4 cycles

- Increases branch mis-prediction penalty
- Makes it easier to increase associativity
  - → In associative cache, tag compare and data output are serialized

Opt 4 – Nonblocking Caches

- Allow data cache to service hits during a miss
  - → Reduces effective miss penalty
  - → "Hit under miss"
- Extended to "Hit under multiple miss"
  - → L2 must support this



Opt 5 – Multibanked Caches

- Organize cache as independent banks to support simultaneous access
  - → ARM Cortex-A8 supports 1-4 banks for L2
  - → Intel i7 supports 4 banks for L1 and 8 banks for L2
- Interleave banks according to block address
  - → Mapping banks by **index mod #banks**

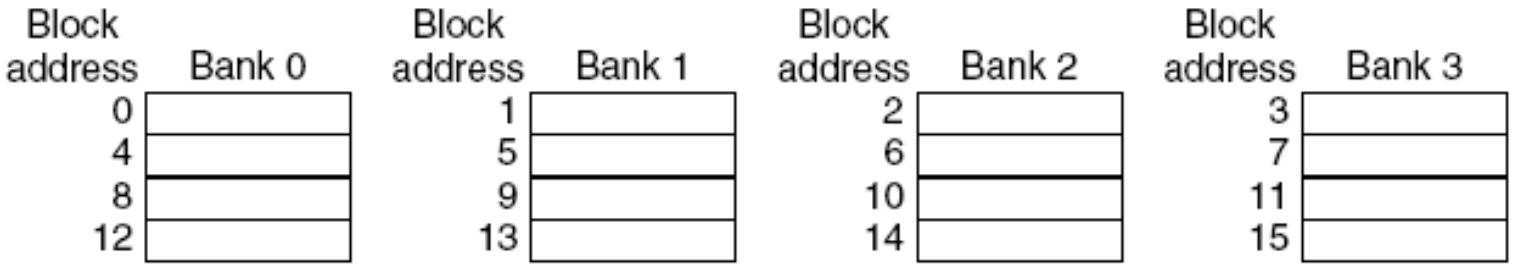


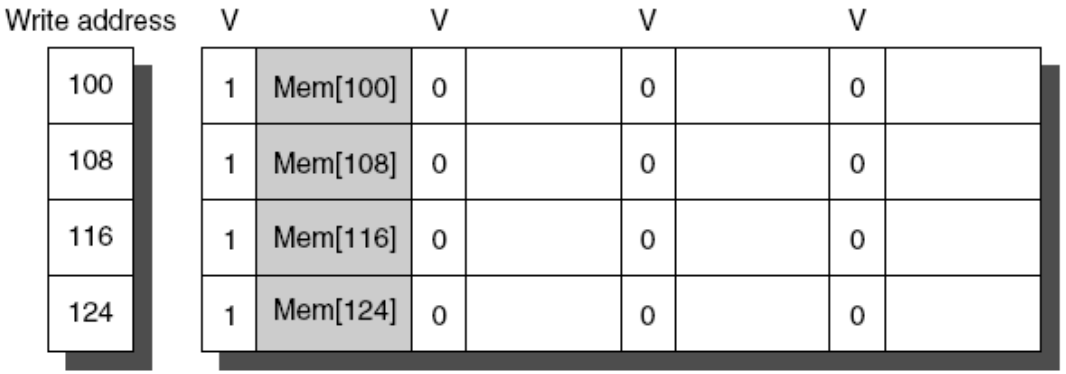
Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Opt 6 – Critical Word First, Early Restart

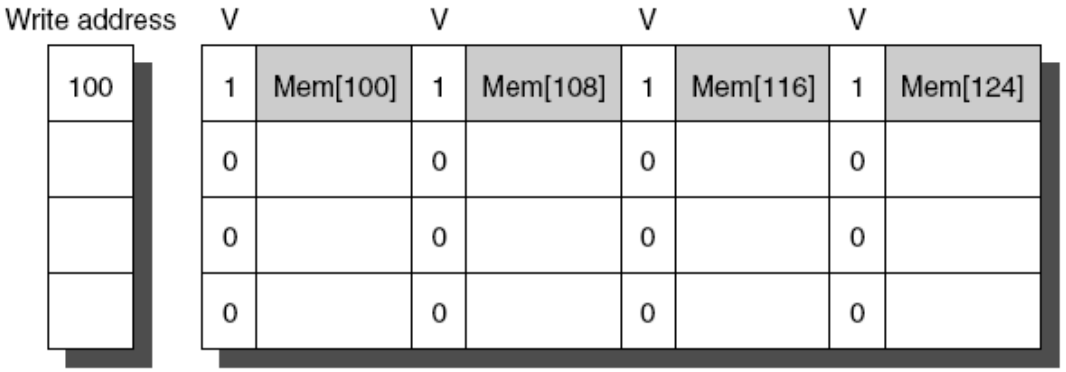
- Processor needs to one word in a block
- *Critical word first*
  - → Request missed word from memory first
  - → Send it to the processor as soon as it arrives
- *Early restart*
  - → Request words in normal order
  - → Send missed work to the processor as soon as it arrives
- Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched
  - → More benefits if block size is larger

Opt 7 – Merging Write Buffer

- When storing to a block that is already pending in the write buffer, update write buffer
- Reduces stalls due to full write buffer
- Do not apply to I/O addresses



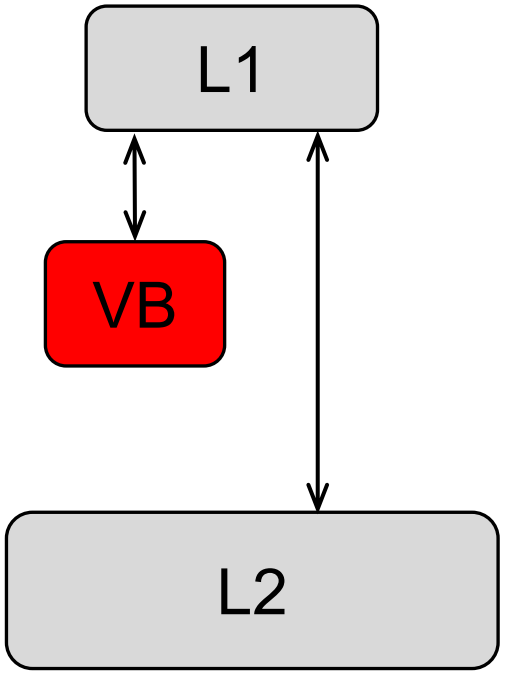
w/o write merging



w write merging

Victim Buffer

- A small fully associative cache between L1 and L2
  → Shared by all sets in L1
- Reduce conflict misses
- On L1 miss, check VB.
  → Hit -> place block back in L1
- Very effective in practice



Opt 8 – Compiler Optimizations

- Gap between CPU and memory requires SW developer to look at memory hierarchy

```
/* before */
for (j = 0; j < 100; j++)
    for (i = 0; i < 100; i++)
        x[i][j] = 2*x[i][j];
```

Opt 8 – Compiler Optimizations

- Loop Interchange
  - → Swap nested loops to access memory in sequential order
  - → Expose spatial locality

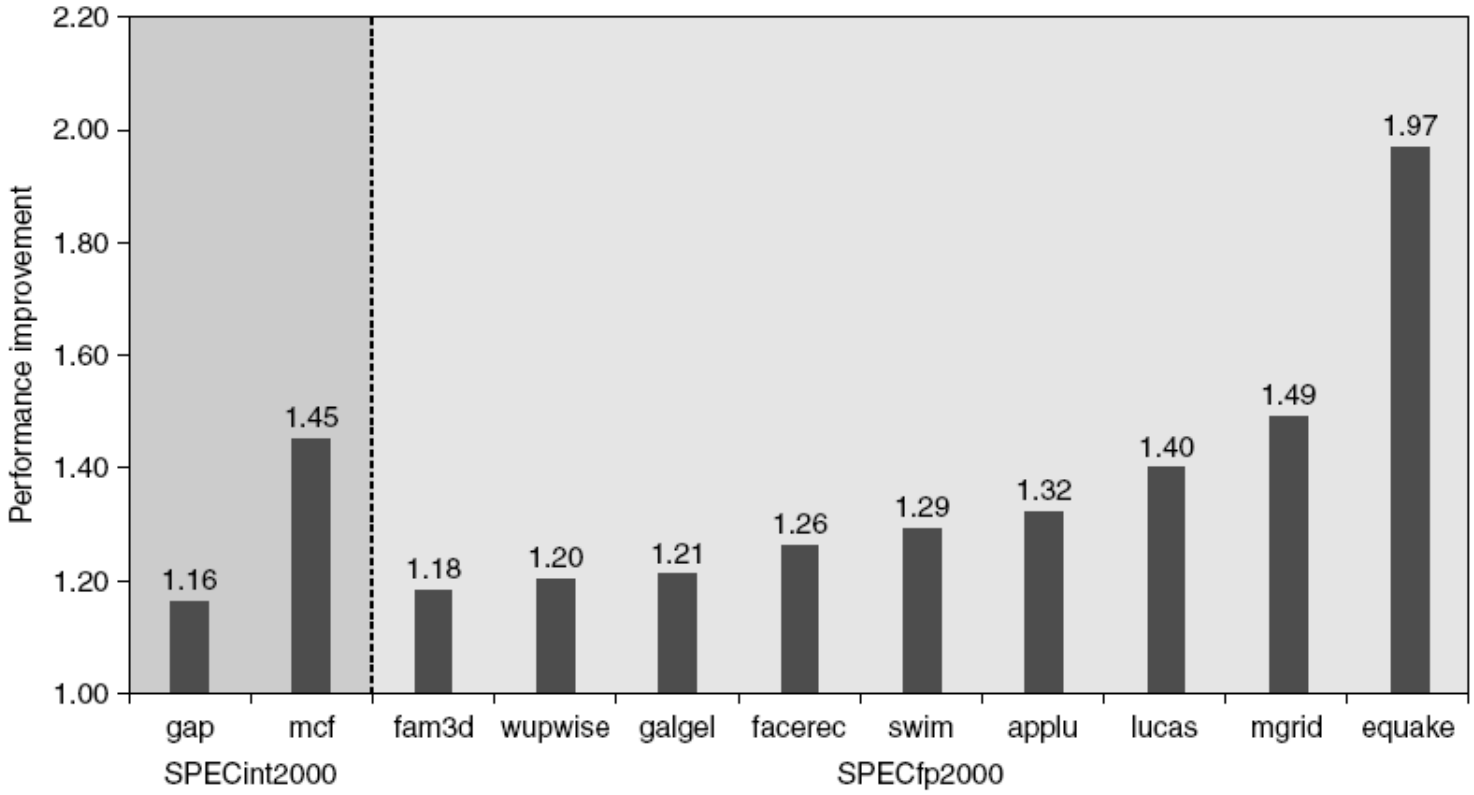
```
/* after*/
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        x[i][j] = 2*x[i][j];
```

Opt 8 – Compiler Optimizations

- Blocking
  - → Instead of accessing entire rows or columns, subdivide matrices into blocks
  - → Requires more memory accesses but improves temporal locality of accesses

Opt 9 – Hardware Prefetching

- Fetch two blocks on miss (include next sequential block)
- Can hurt power if prefetched data are not used.



Some results obtained on Pentium 4 w. Pre-fetching

Opt 10 – Compiler Prefetching

- Insert prefetch instructions before data is needed
- Non-faulting: prefetch doesn't cause exceptions

- Register prefetch
  → Loads data into register
- Cache prefetch
  → Loads data into cache

- Combine with loop unrolling and software pipelining

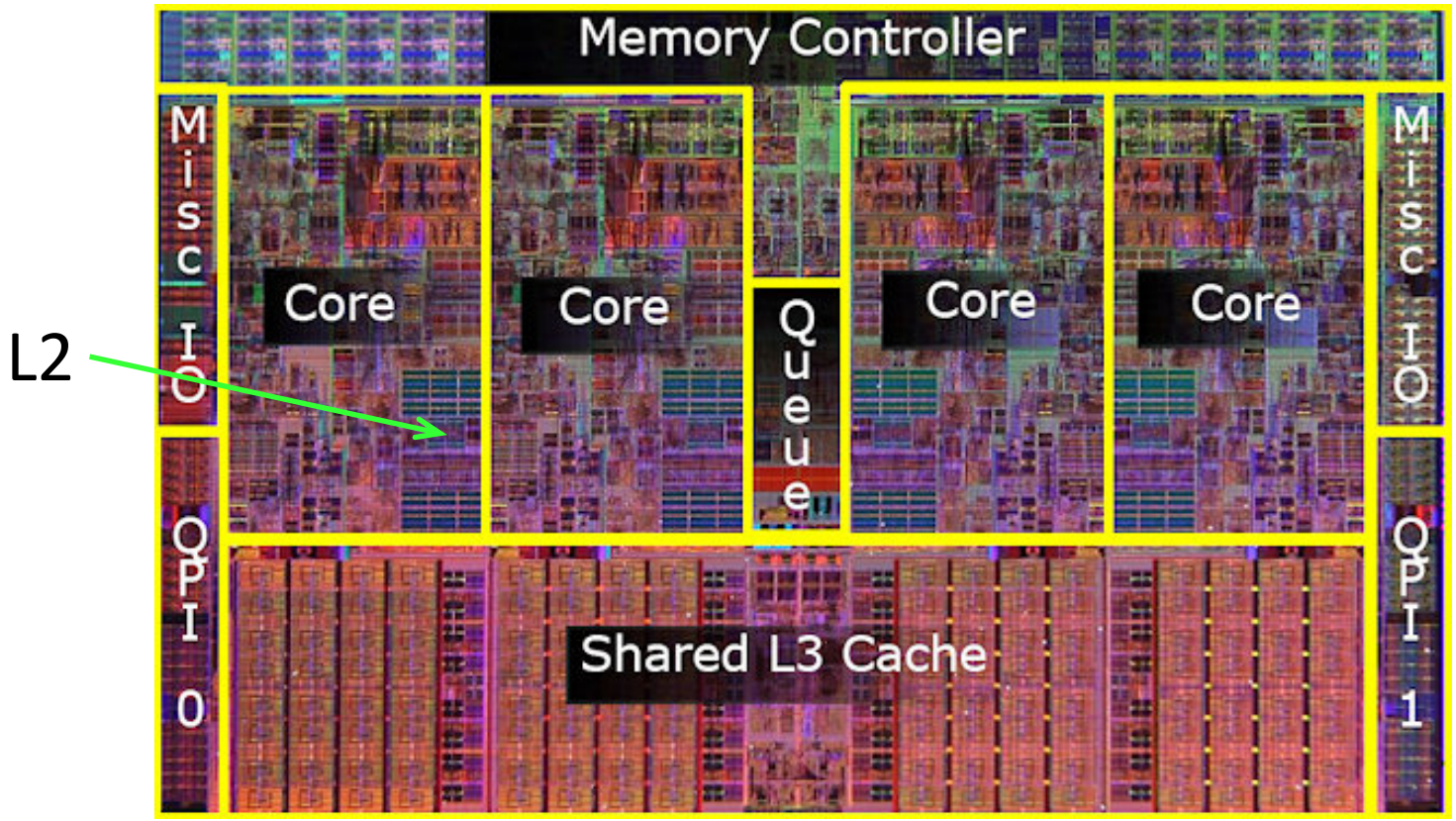
Cache - Summary

- A small and fast buffer between CPU and main memory
- Direct mapped cache
  → shorter hit time, higher miss rate, lower power consumption
- Associative cache
  → longer hit time, lower miss rate, higher power consumption
- Performance evaluation
  → AMAT – average memory access time: maybe misleading
  → CPI with stall cycles due to cache misses: more accurate

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Example: Intel Core I7 Cache



- L1 I$ - 32 KB, L1 D$ - 32KB 8-way set associative, private
- L2 – 256 KB, 8-way set associative, private
- L3 – 8 MB, 16-way set associative, shared

Virtual Memory

Why Virtual Memory

- Limited physical memory leads to complications
- Size of a program is larger than main memory size
- Memory demand of multiple programs is larger than main memory size
- *Observation: a program often needs to small part of memory during its execution*

  → Load what is needed into main memory!

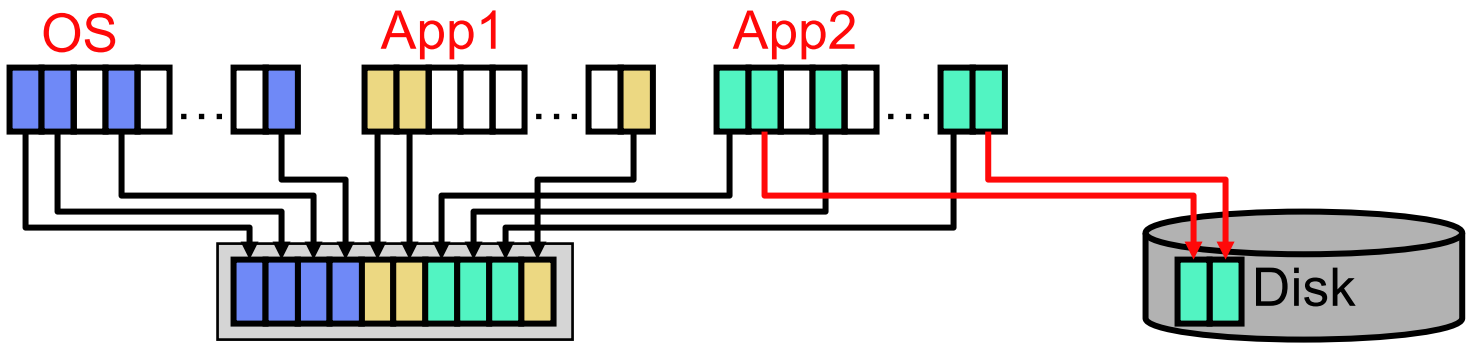
Why Virtual Memory

- Programs were divided into pieces and identified pieces that are mutually exclusive
- These pieces were loaded and unloaded under user program control during execution
- Calls between procedures in different modules was lead to overlaying of one module with the other
- Used to be done by hand

  → Significant burden on programmers

Basics of Virtual Memory

- Programs use virtual addresses (VA)
- Memory uses physical addresses (PA)
- VA -> PA at the page granularity
  - → pages can be anywhere in memory
  - → or in disk



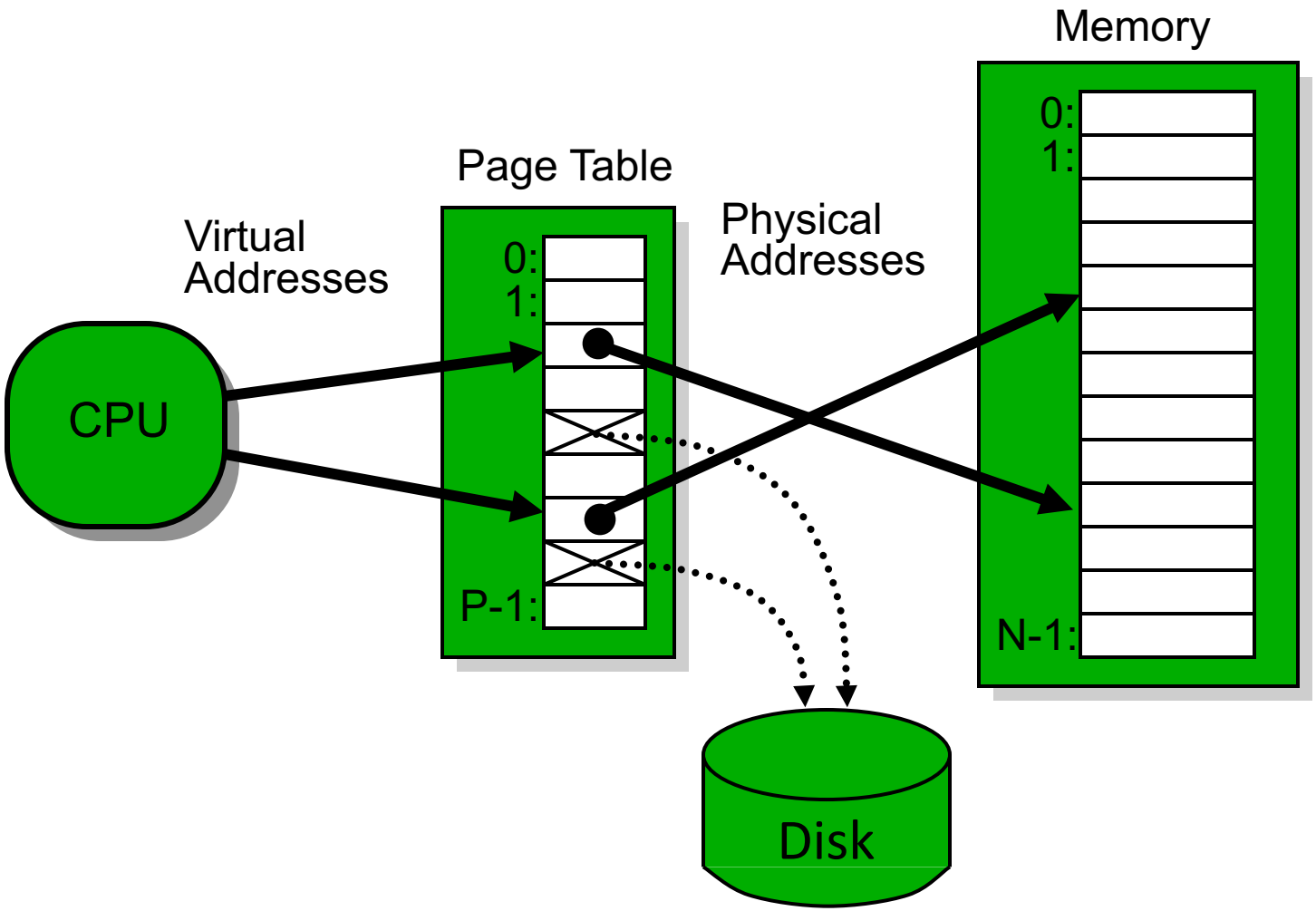
Basics of Virtual Memory

- Use physical DRAM as cache for disk
  - → Address space of a process can exceed physical memory size
  - → Sum of address spaces of multiple processes can exceed physical memory
- Simplify memory management
  - → Multiple processes resident in main memory
    - ➤ Each process with its own address space
  - → Only “active” code and data is actually in memory
    - ➤ Allocate more memory to process as needed
- Provide protection
  - → One process can’t interfere with another
    - ➤ Because they operate in different address spaces
  - → User process cannot access privileged information
    - ➤ Different sections of address space have different permissions

Basic Issues

- Shares same basic concepts of cache memory but different terminology
- Issues
  - → Mapping: translation of virtual to physical address
  - → Management: controlled sharing and protection. Protection in multi-programming environment
- Mapping techniques
  - → Paging (demand paging)
  - → Segmentation

A Simplified View of Virtual Memory



Cache vs Virtual Memory

Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	100–200 clock cycles
Miss penalty	8–200 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–160 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45-bit physical address to 14–20-bit cache address	32–64-bit virtual address to 25–45-bit physical address

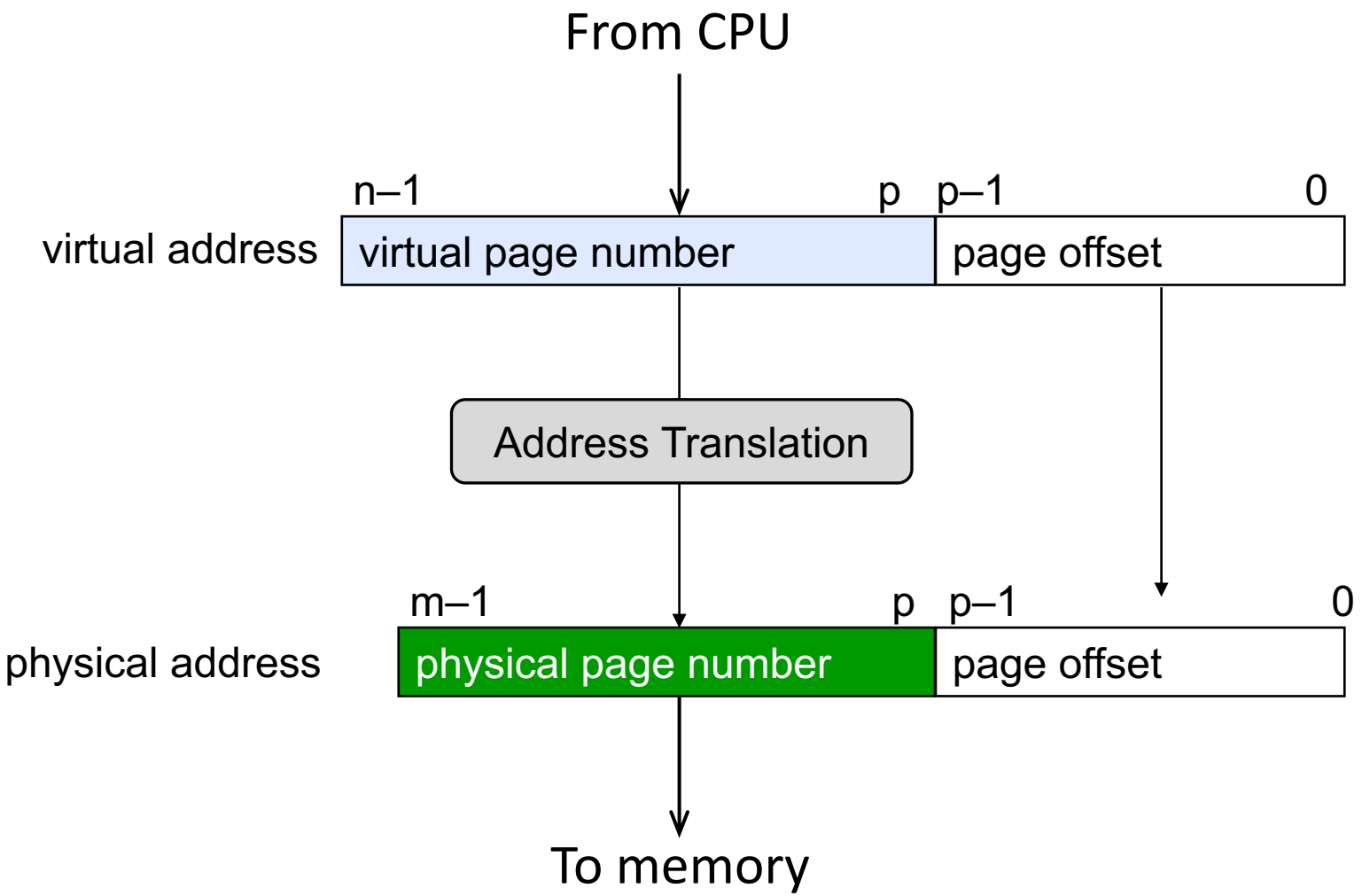
Cache vs Virtual Memory

- Terminology
  - → Block → *page*
  - → Cache miss → *page fault*
- Replacement on cache memory misses by hardware whereas virtual memory replacement is by OS
- Size of processor address determines size of VM whereas cache size is independent of address size
- VM can have fixed or variable size blocks
  - → page vs segmentation: they both have pros and cons

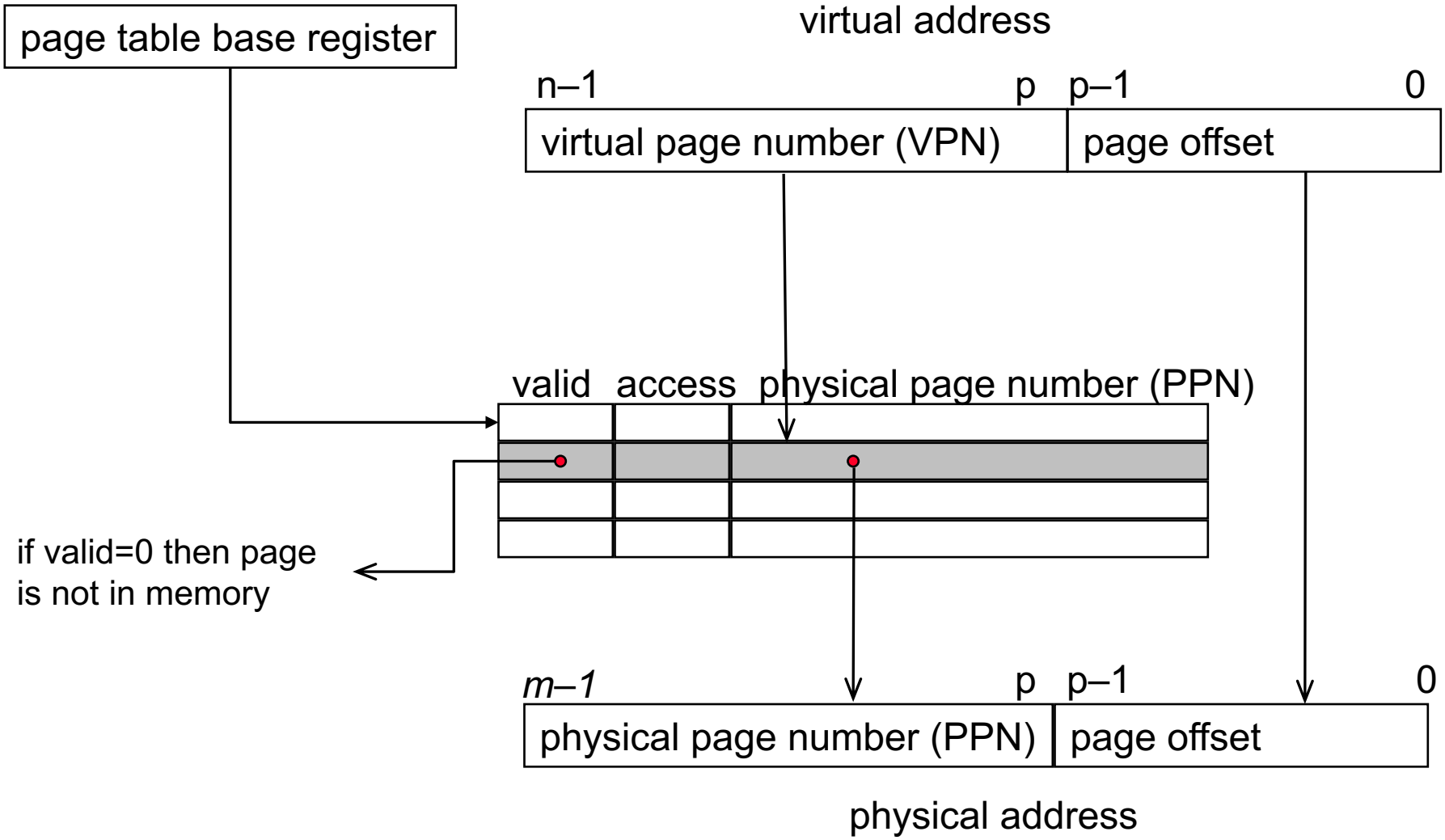
Virtual Memory Design Issues

- Page size should be large enough to try to amortize high access time
  → Typical size: 4KB – 16KB
- Reducing page fault rate is important
  → Fully associative placement of pages in memory
- Page faults not handled by hardware
  → OS can afford to use clever algorithm for page replacement to reduce page fault rate
- Write through approach is too expensive
  → Write back approach is always used

Virtual Memory Address Translation

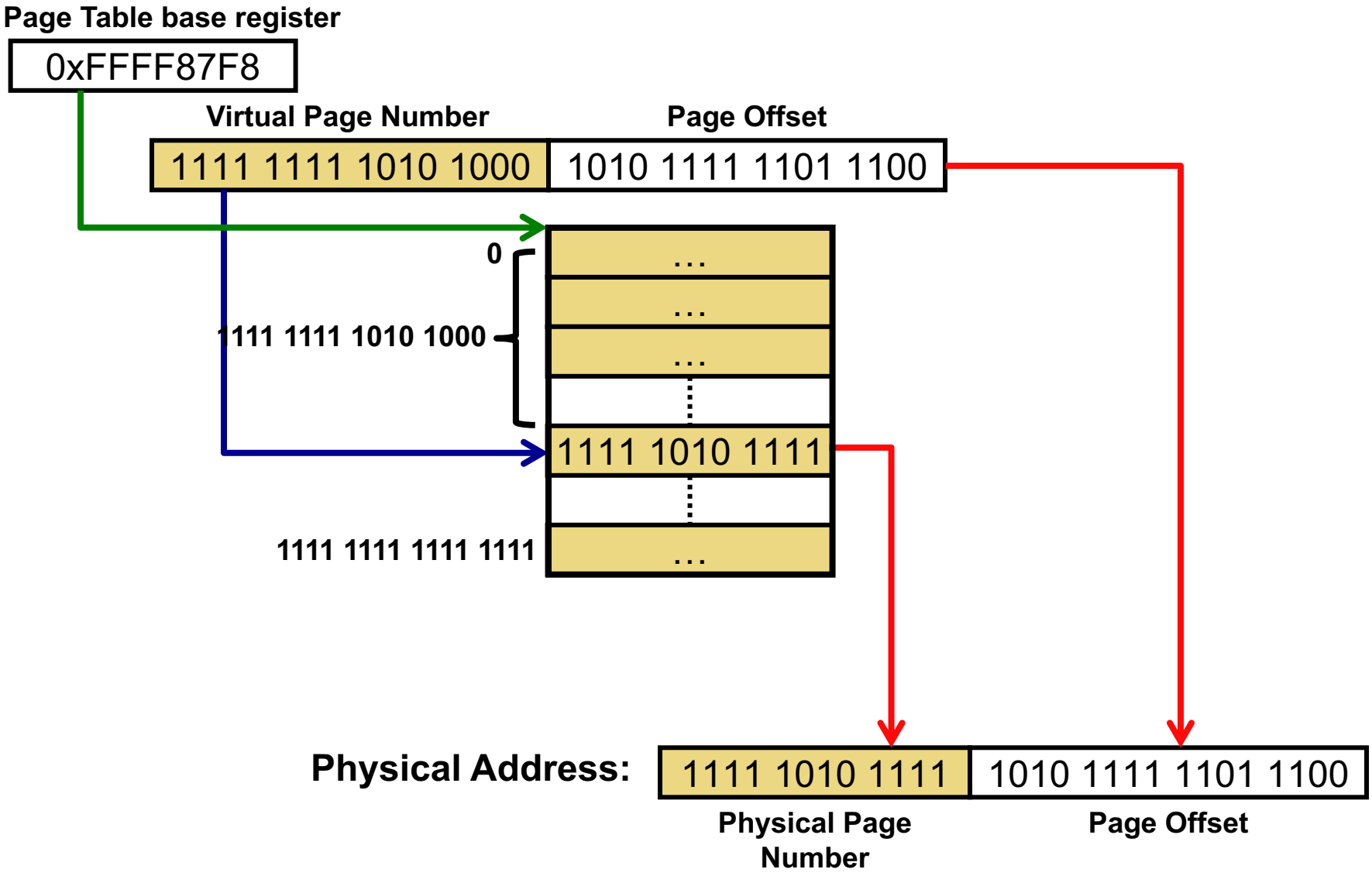


Address Translation



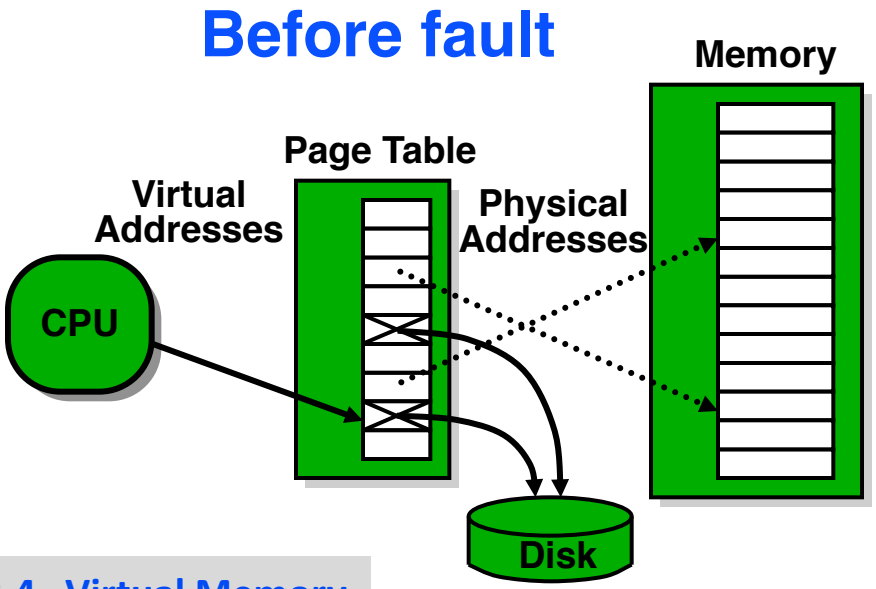
- Each process has its own page table.

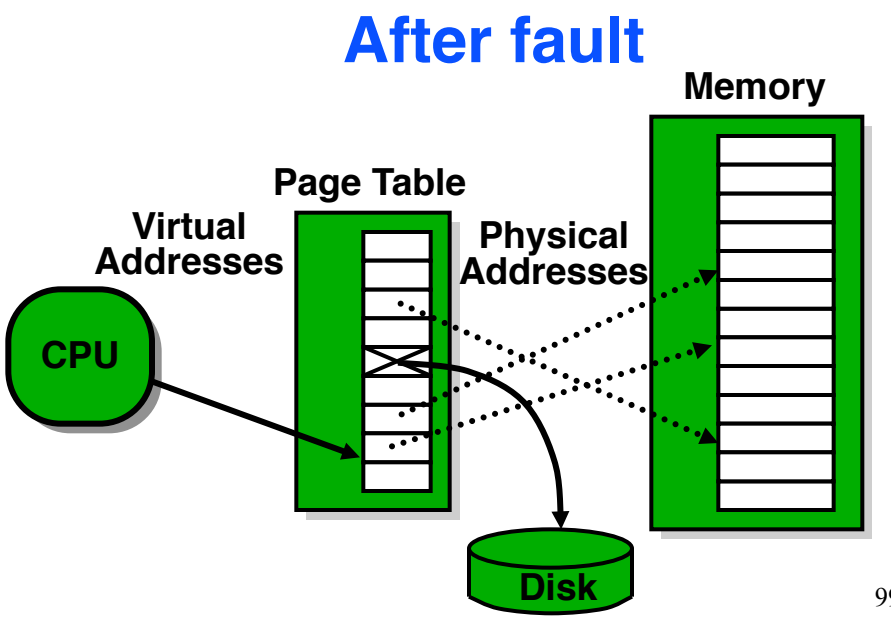
Address Translation - Example



Page Faults

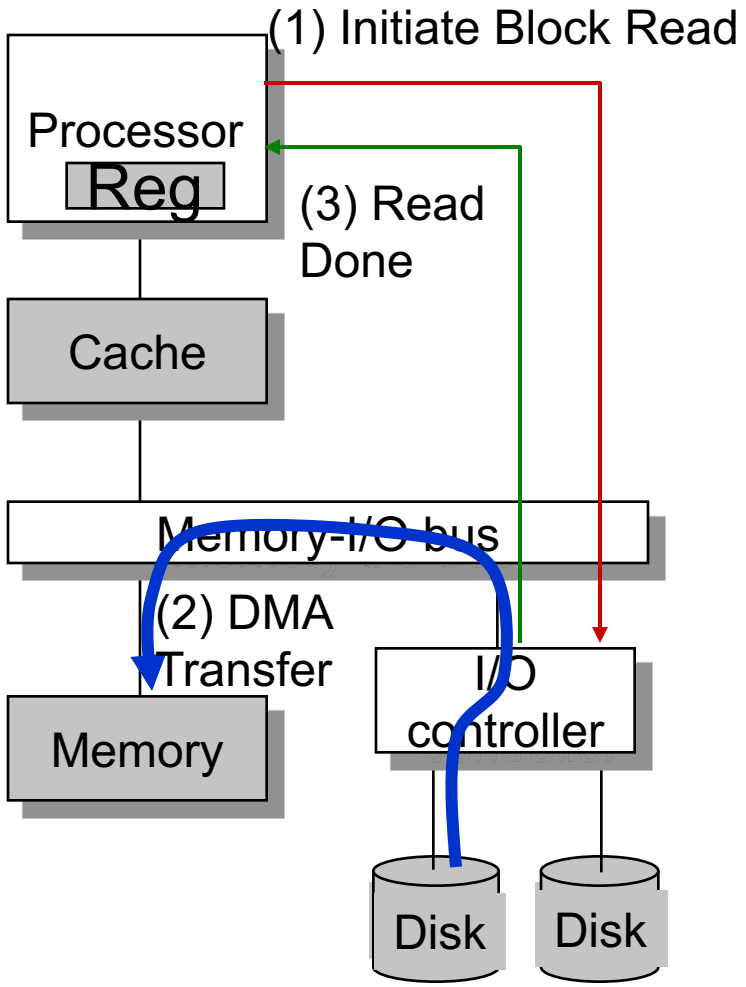
- What if object is on disk rather than in memory?
  - → Page table entry indicates virtual address not in memory
  - → OS exception handler invoked to move data from disk into memory – VM and Multiprogramming are symbiotic
    - ➤ Current process suspends, others can resume
    - ➤ OS has full control over placement, etc.





Servicing Page Faults

- Processor signals controller
  - → Read block of length P starting at disk address X and store starting at memory address Y
- Read occurs
  - → Direct Memory Access (DMA)
  - → Under control of I/O controller
- I/O controller signals completion
  - → Interrupt processor
  - → OS resumes suspended process



Page Replacement

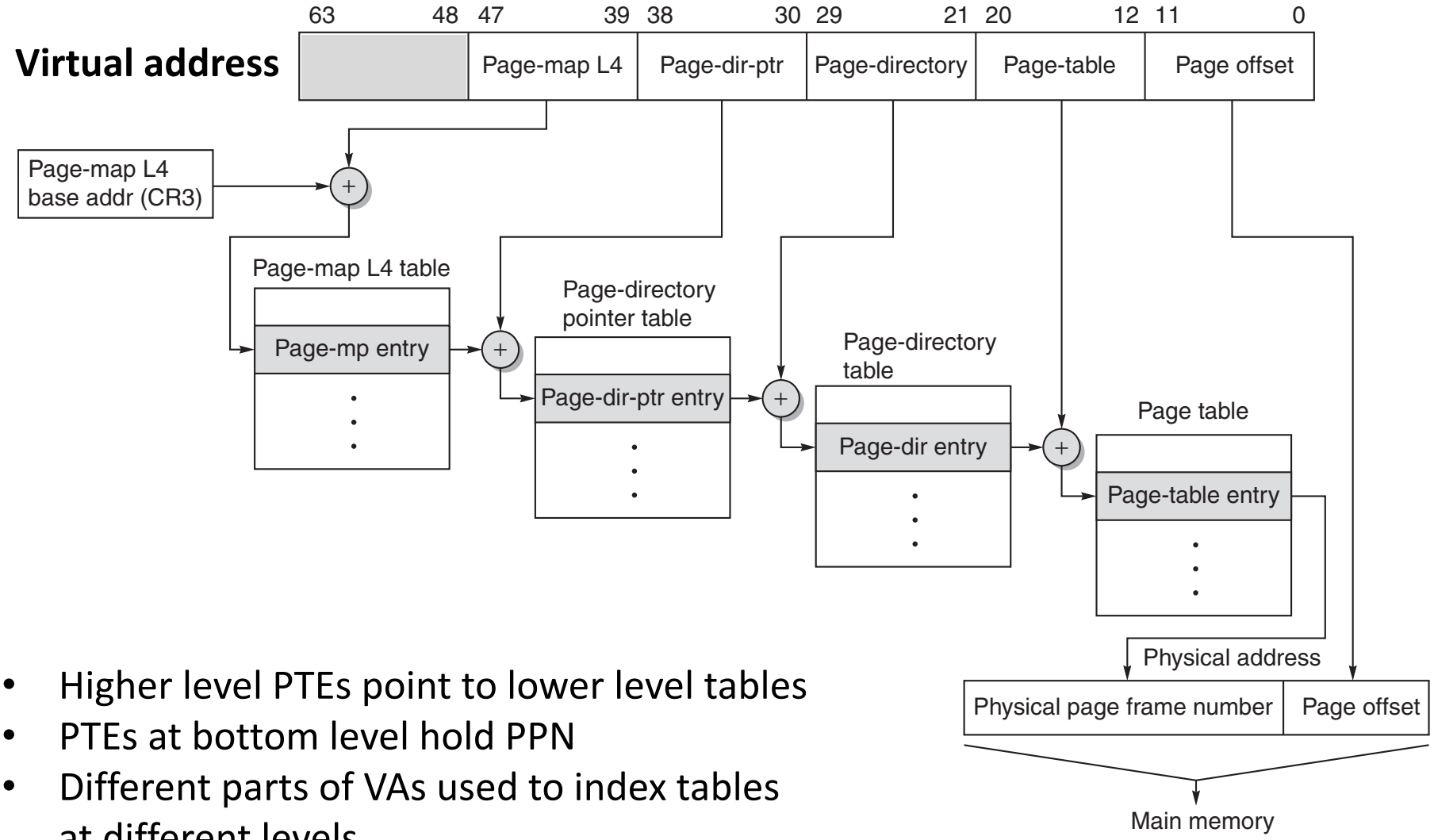
- When there are no available free pages to handle a fault we must find a page to replace.
- This is determined by the *page replacement algorithm*.
- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.
- LRU is the most popular replacement policy.
  - → Each page is associated with a *use/reference* bit.
  - → It is set whenever a page is accessed.
  - → OS periodically clears these bits.

Question: Page size and Page Table Size

- Assume 32-bit virtual address
  - → Page table size if page size is 4K?
  - → Page table size if page size is 16K?

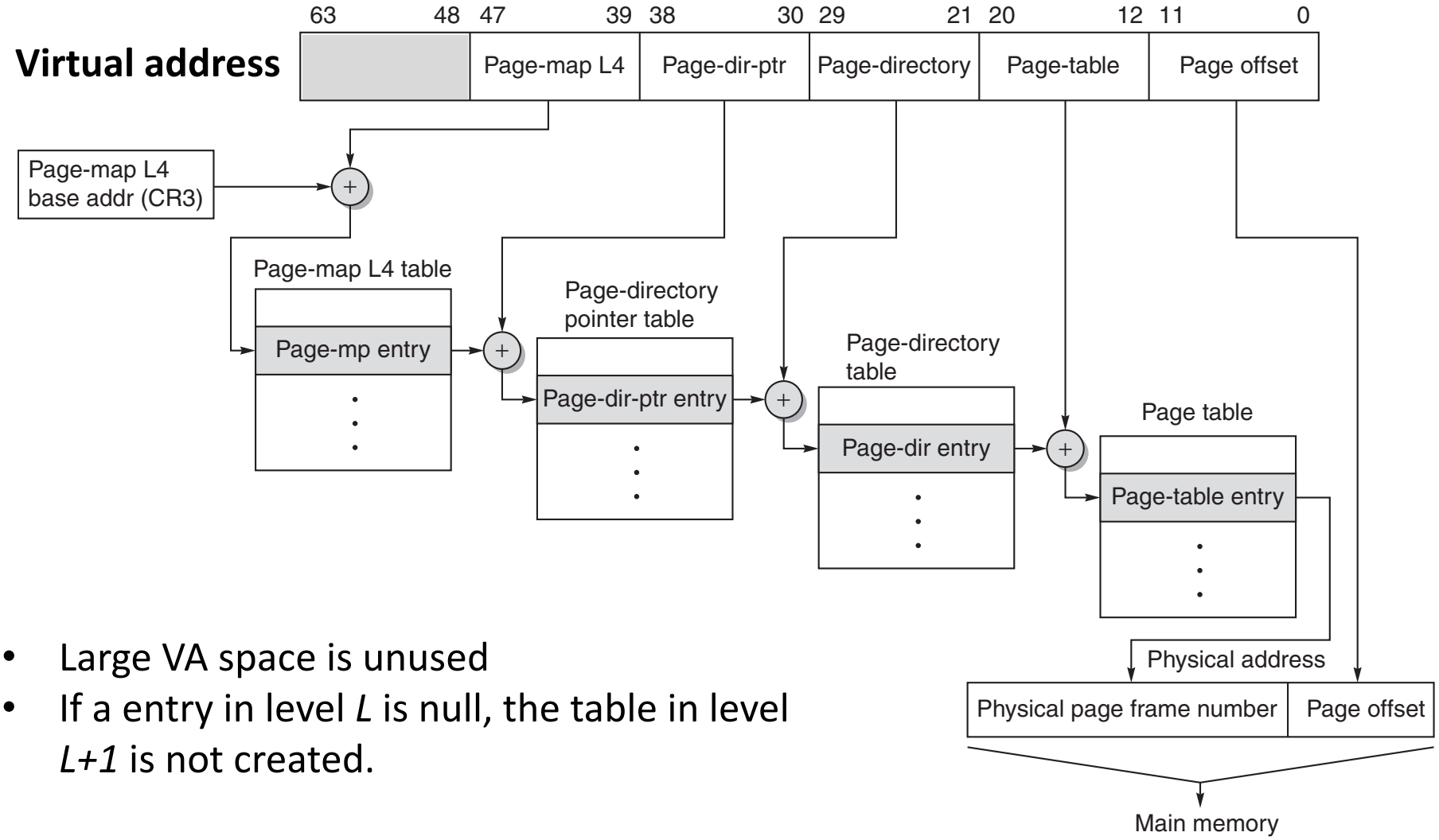
- What about 64-bit virtual address?

Multi-Level Page Table



- Higher level PTEs point to lower level tables
- PTEs at bottom level hold PPN
- Different parts of VAs used to index tables at different levels

Multi-Level Page Table

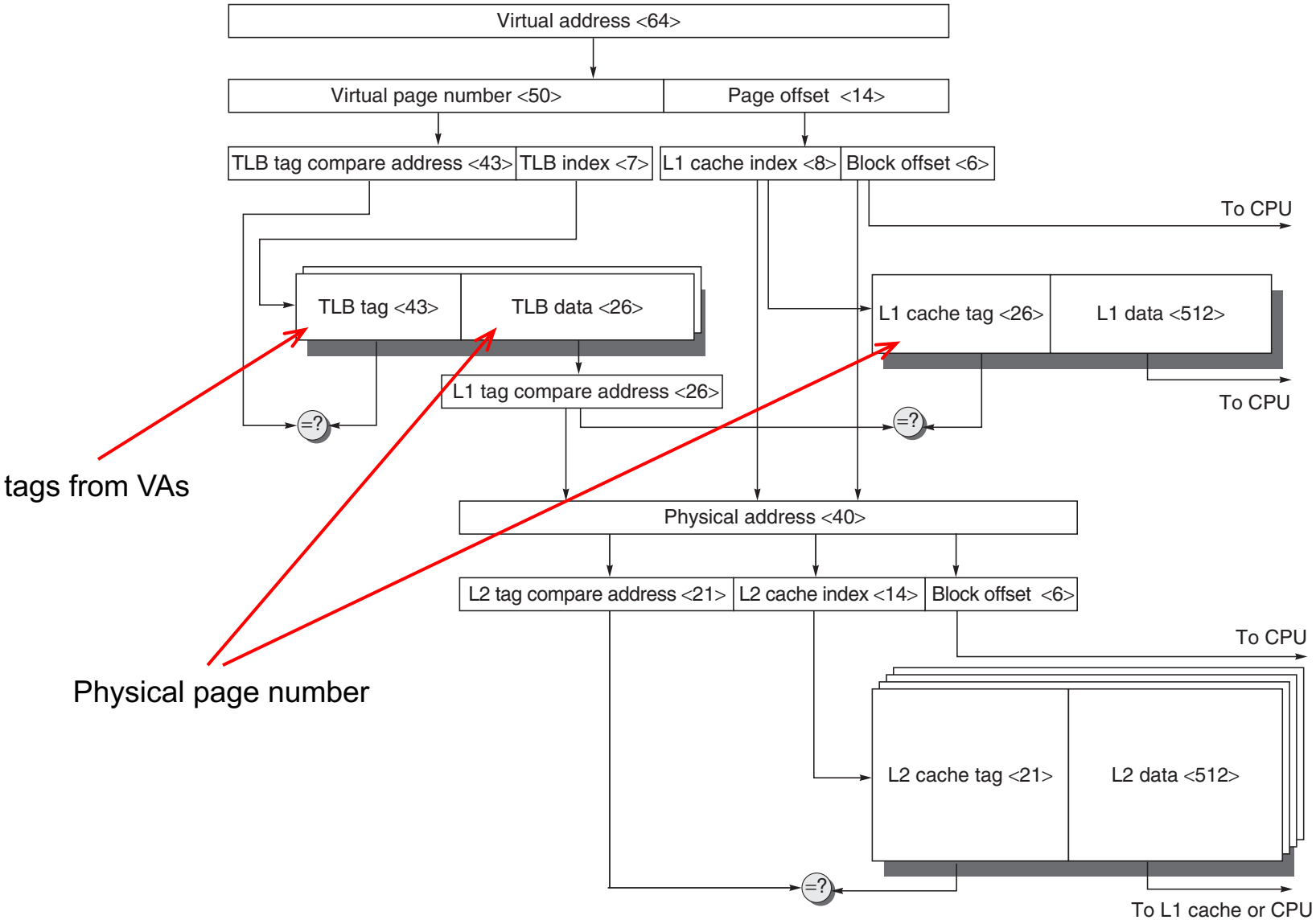


- Large VA space is unused
- If a entry in level *L* is null, the table in level *L+1* is not created.

Selecting Page Size

- The larger the page size, the smaller page table
  → A page table can occupy a lot of space
- Larger page size -> larger L1 cache
- Larger page size -> less TLB misses, faster xlation
- More efficient to xfer larger pages from 2nd storage
- Large page size
  → Wasted storage
  → Wasted IO bandwidth
  → Slower process start up

Fast Address Translation – TLB



Virtual Memory – Protection

- Virtual memory and multiprogramming
  - → Multiple processes sharing processor and physical memory
- Protection via virtual memory
  - → Keeps processes in their own memory space
- Role of architecture:
  - → Provide user mode and supervisor mode
  - → Protect certain aspects of CPU state: PC, register, etc
  - → Provide mechanisms for switching between user mode and supervisor mode
  - → Provide mechanisms to limit memory accesses
  - → Provide TLB to translate addresses

Page Table Entries

- Address – physical page number
- Valid/present bit
- Modified/dirty bit
- Reference bit
  - → For LRU
- Protection bits – Access right field defining allowable accesses
  - → E.g., read only, read-write, execute only
  - → Typically support multiple protection modes (kernel vs user)

Summary

- OS virtualizes memory and IO devices
  → Each process has an illusion of private CPU and memory
- Virtual memory
  → Arbitrarily large memory, isolation/protection, inter-process communication
  → Reduce page table size
  → Translation buffers – cache for page table
  → Manage TLB misses and page faults

Memory Technology

Memory Technology

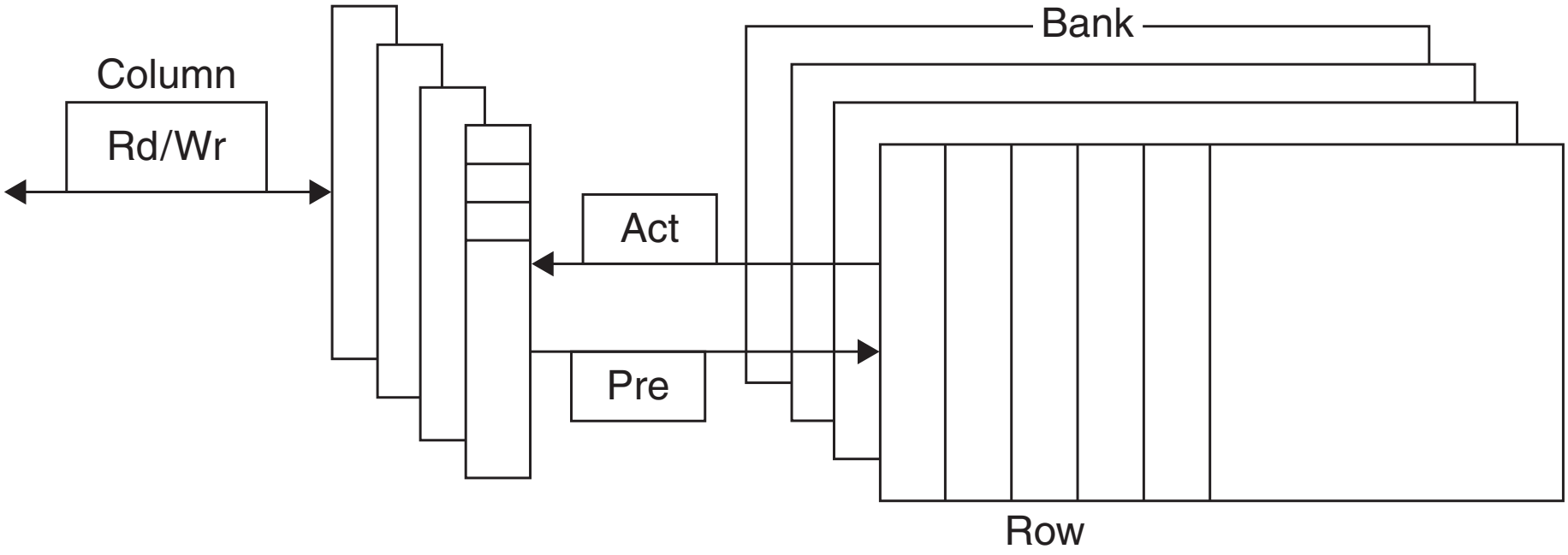
- Performance metrics
  - → Latency is concern of cache
  - → Bandwidth is concern of multiprocessors and I/O
  - → Access time
    - ➤ Time between read request and when desired word arrives
  - → Cycle time
    - ➤ Minimum time between unrelated requests to memory
- DRAM used for main memory, SRAM used for cache

Memory Technology

- SRAM
  - → Requires low power to retain bit
  - → Requires 6 transistors/bit

- DRAM
  - → Must be re-written after being read
  - → Must also be periodically refreshed
    - ➤ Every ~ 8 ms
    - ➤ Each row can be refreshed simultaneously
  - → One transistor + one capacitor/bit
  - → Address lines are multiplexed:
    - ➤ Upper half of address: row access strobe (RAS)
    - ➤ Lower half of address: column access strobe (CAS)

Organization of DRAM



- Each DRAM organized row x column
- Larger memory contains banks of DRAM chips

Memory Technology

- Amdahl:
  - → Memory capacity should grow linearly with processor speed
  - → Unfortunately, memory capacity and speed has not kept pace with processors

- Some optimizations:
  - → Multiple accesses to same row
  - → Synchronous DRAM
    - ➤ Added clock to DRAM interface
    - ➤ Burst mode with critical word first
  - → Wider interfaces
  - → Double data rate (DDR)
  - → Multiple banks on each DRAM device

Memory Optimizations

Production year	Chip size	DRAM Type	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
			Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	DRAM	180	150	75	250
1983	256K bit	DRAM	150	120	50	220
1986	1M bit	DRAM	120	100	25	190
1989	4M bit	DRAM	100	80	20	165
1992	16M bit	DRAM	80	60	15	120
1996	64M bit	SDRAM	70	50	12	110
1998	128M bit	SDRAM	70	50	10	100
2000	256M bit	DDR1	65	45	7	90
2002	512M bit	DDR1	60	40	5	80
2004	1G bit	DDR2	55	35	5	70
2006	2G bit	DDR2	50	30	2.5	60
2010	4G bit	DDR3	36	28	1	37
2012	8G bit	DDR3	30	24	0.5	31

Figure 2.13 Times of fast and slow DRAMs vary with each generation. (Cycle time is defined on page 95.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs. The introduction of various burst transfer modes in the mid-1990s and SDRAMs in the late 1990s has significantly complicated the calculation of access time for blocks of data; we discuss this later in this section when we talk about SDRAM access time and power. The DDR4 designs are due for introduction in mid- to late 2012. We discuss these various forms of DRAMs in the next few pages.

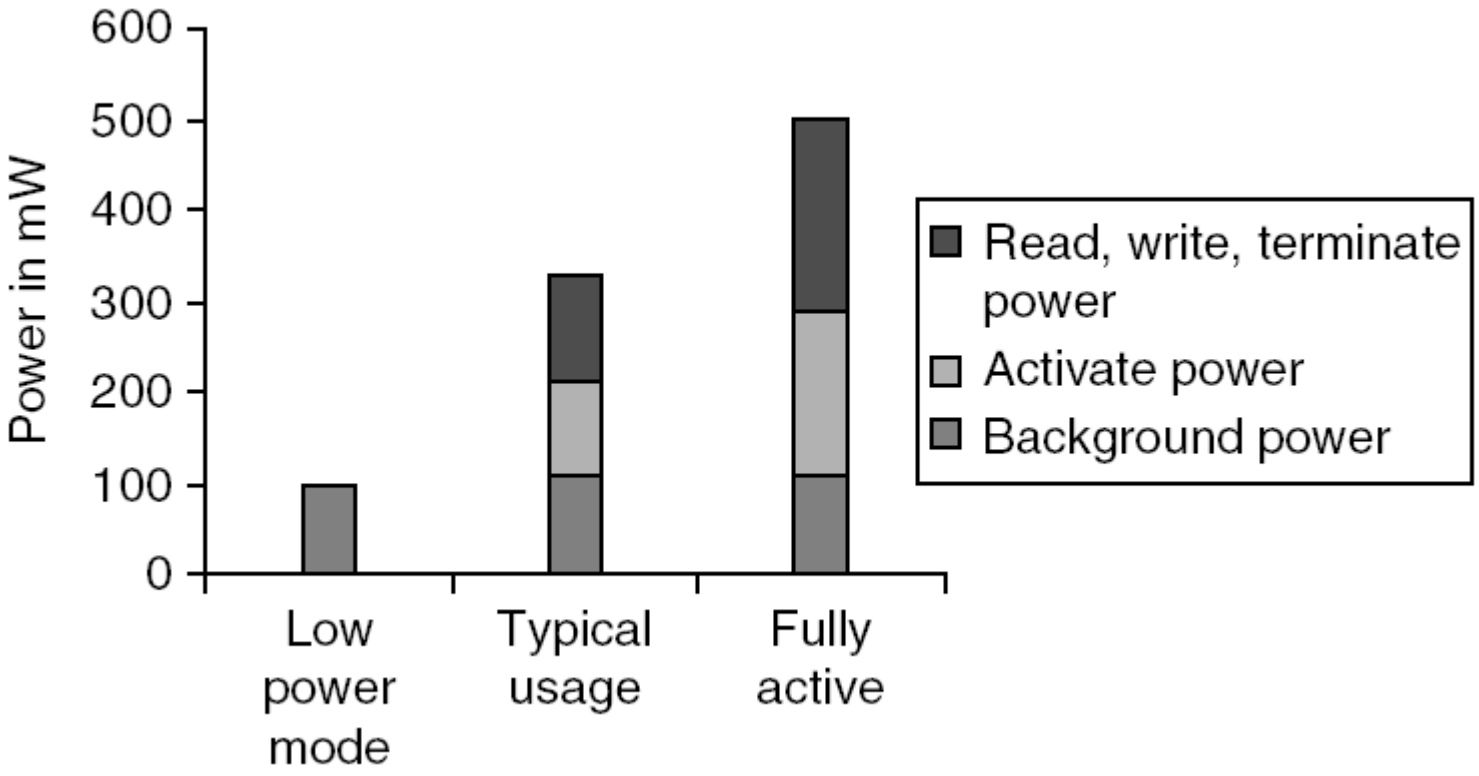
Memory Optimizations

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1066–1600	2133–3200	DDR4-3200	17,056–25,600	PC25600

Figure 2.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2010. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles as four numbers, which are specified by the DDR standard. For example, DDR3-2000 CL 9 has latencies of 9-9-9-28. What does this mean? With a 1 ns clock (clock cycle is one-half the transfer rate), this indicate 9 ns for row to columns address (RAS time), 9 ns for column access to data (CAS time), and a minimum read time of 28 ns. Closing the row takes 9 ns for precharge but happens only when the reads from that row are finished. In burst mode, transfers occur on every clock on both edges, when the first RAS and CAS times have elapsed. Furthermore, the precharge in not needed until the entire row is read. DDR4 will be produced in 2012 and is expected to reach clock rates of 1600 MHz in 2014, when DDR5 is expected to take over. The exercises explore these details further.

Memory Power Consumption

- Reducing power in SDRAMs:
  - → Lower voltage 1.8v@DDR2 -> 1.2v@DDR4
  - → Low power mode (ignores clock, continues to refresh)



Flash Memory

- Type of EEPROM
- Must be erased (in blocks) before being overwritten
- Non volatile
- Limited number of write cycles
- Cheaper than SDRAM, more expensive than disk
- Slower than SRAM, faster than disk
- No moving parts – ideal for PMD

Memory Dependability

- Memory is susceptible to cosmic rays
- *Soft errors*: dynamic errors
  → Detected and fixed by error correcting codes (ECC)
- *Hard errors*: permanent errors
  → Use sparse rows to replace defective rows

- Solutions:
  → Error correcting code – correct signal error
  → Chipkill: a RAID-like error recovery technique using redundancy

Backup

Virtual Machines

- Supports isolation and security
- Sharing a computer among many unrelated users
- Enabled by raw speed of processors, making the overhead more acceptable

- Allows different ISAs and operating systems to be presented to user programs
  - → "System Virtual Machines"
  - → SVM software is called "virtual machine monitor" or "hypervisor"
  - → Individual virtual machines run under the monitor are called "guest VMs"

Impact of VMs on Virtual Memory

- Each guest OS maintains its own set of page tables
  - → VMM adds a level of memory between physical and virtual memory called “real memory”
  - → VMM maintains shadow page table that maps guest virtual addresses to physical addresses
    - ➤ Requires VMM to detect guest’s changes to its own page table
    - ➤ Occurs naturally if accessing the page table pointer is a privileged operation