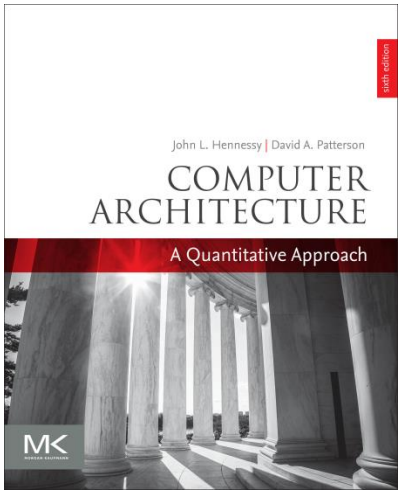
CESE 4085 Modern Computer Architecture

Lectures 3 & 4: Memory Hierarchy Design



Chapter 2

Memory Hierarchy Design

Overlap with other courses

- Self-contained story
- Depth of the material
- We perform checks with CE course – it may seem the course covered everything, but it mentioned many things superficially in order to draw a complete picture

Lecture Overview

Lecture 3:

- Memory Hierarchy
- Caches (intro, characteristics)
- Average Memory Access Time (AMAT)
- Caches – Reduce Miss rate
- Caches – Reduce Miss penalty

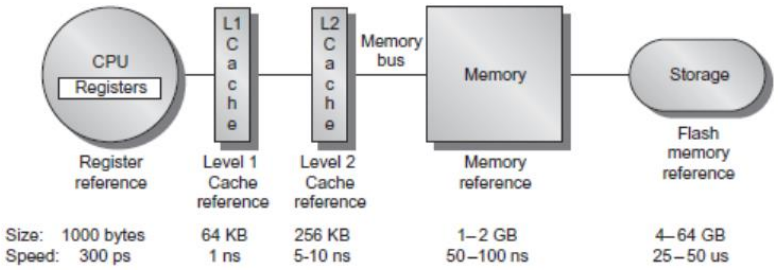
Lecture 4:

- Caches – Reduce Hit time
- SRAM vs. DRAM
- Other technologies
- Virtual Memory and Virtual Machines

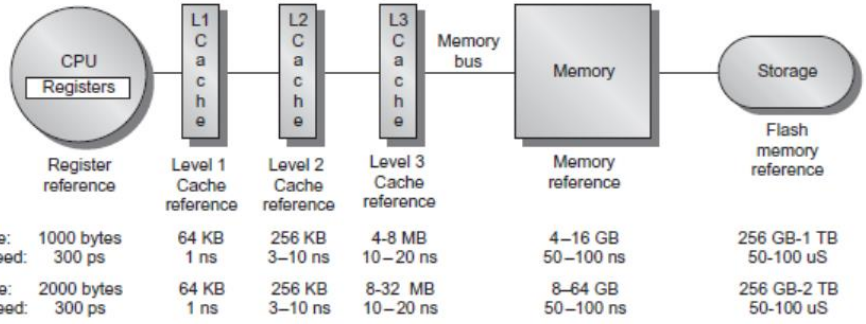
Introduction

- 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** ensures that nearly all references can be found in smaller memories
  - Gives the allusion of a large, fast memory being presented to the processor

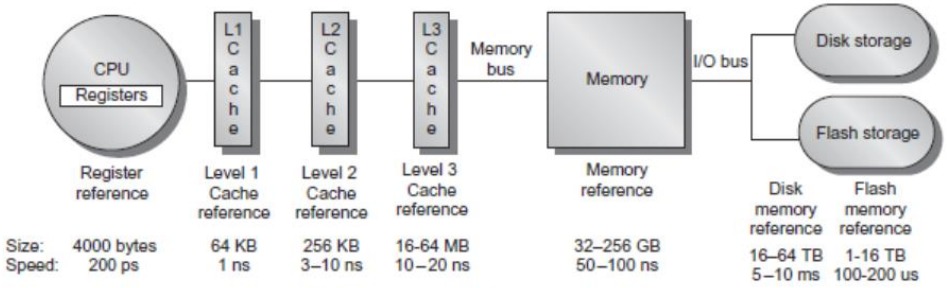
Memory Hierarchy



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

Memory Hierarchy Design

Memory hierarchy design becomes more crucial with recent multi-core processors:

- 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/second
    - = 409.6 GB/s!
  - DRAM bandwidth is only 8% of this (34.1 GB/s)
  - Requires:
    - Multi-port, pipelined caches
    - Two levels of cache per core
    - Shared third-level cache on chip
    - (now alsp witnessing L4 caches in same package)

Performance and Power

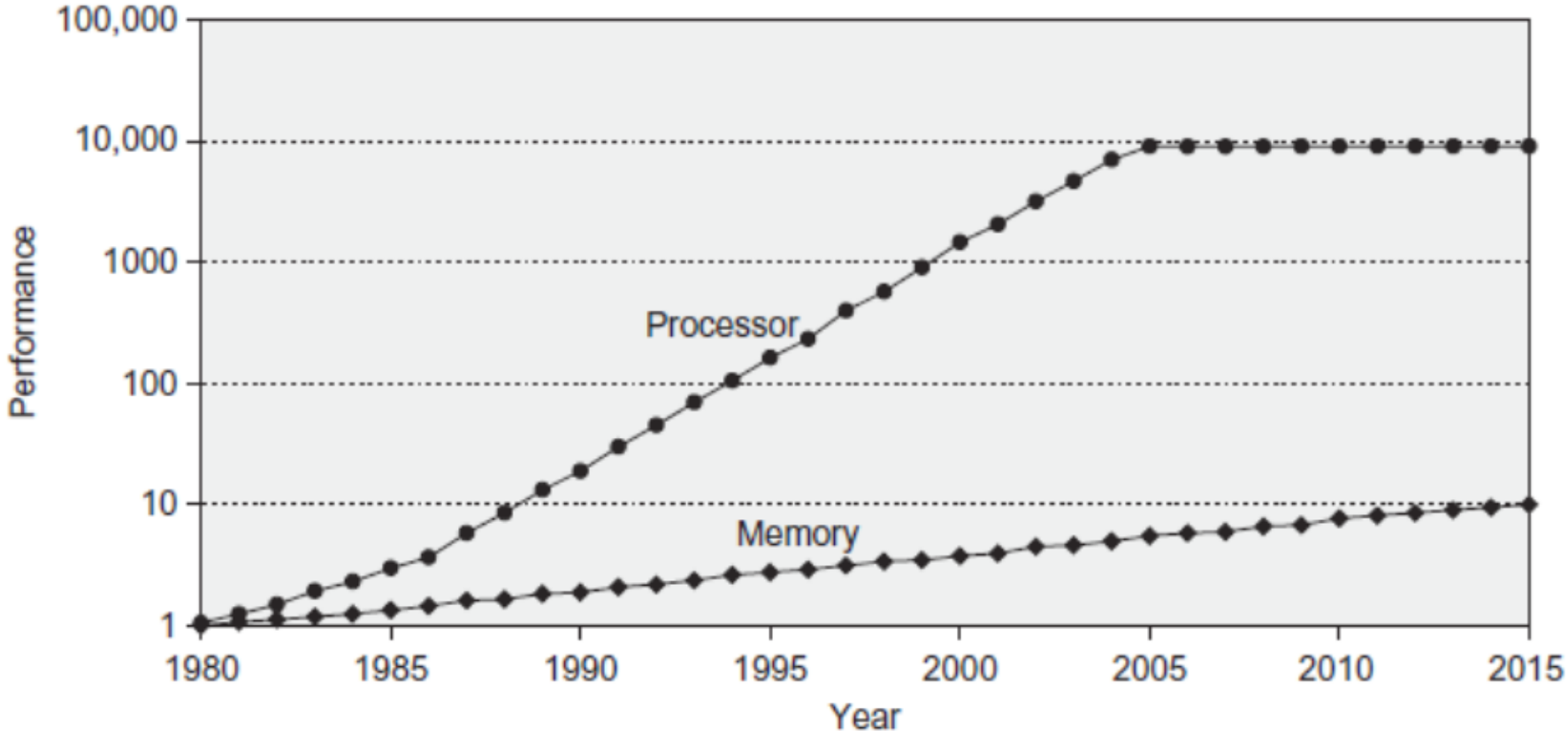
High-end microprocessors have >10 MB on-chip cache

- Consumes large amount of area and power budget (remember static power!)
- updated #s:
  - AMD Epyc 9184X, 16 cores, 768 MB of L3 cache
  - IBM z14, 10 cores, 128MB of on-chip L3 cache
    - & 672MB L4 cache per 60 cores (multi-chip)

Question:

- End of Dennard Scaling led to multicore processors, what was the other side-effect?

Memory Performance Gap



Intermezzo (caches)

Understand the basics of caches

Identify all the parameters of caches
- Organization (mapping, associativity, size)
- Performance (average memory access time (AMAT))

Propose improvements with regard to parameters

Investigate (potential) effect of improvement – e.g., execution time

Investigate (potential) cost of improvement

Complicating factors:
- Interplay between all parameters and other CPU optimizations
- Different applications could produce different results

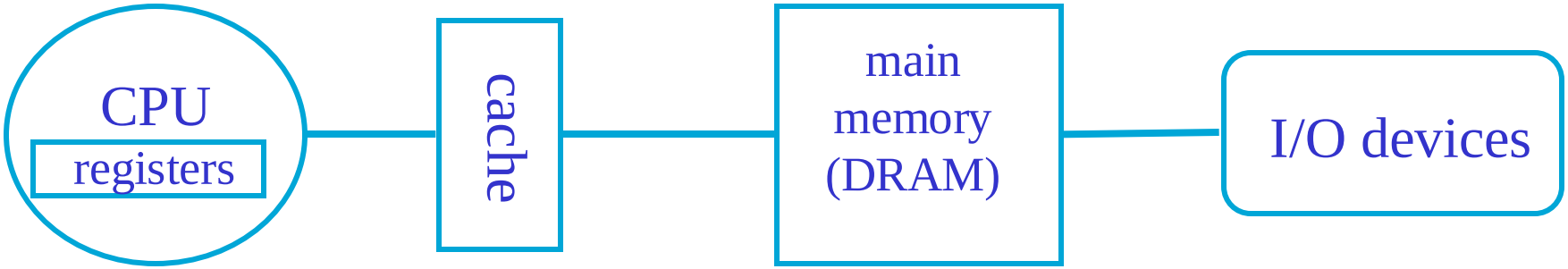
Cache

Literature:
Cache line = cache block

A cache is just a smaller memory

Four memory hierarchy questions:

- Where can a block be placed in a cache level? (**block placement**)
- How is a block found? (**block identification**)
- Which block should be replaced when data do not fit? (**block replacement**)
- What happens on a write? (**write strategy**)



Memory Hierarchy Basics

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
  - block address MOD number of sets in cache

Memory Hierarchy Basics

n sets => *n-way set associative*

- *Direct-mapped cache =>* one block per set
- *Fully associative* => one set

Writing to cache --> two "strategies"

- *Write-through*
  - Immediately update lower levels of hierarchy
- *Write-back*
  - Only update lower levels of hierarchy when an updated block is replaced

Both strategies use ***write buffer*** to make writes asynchronous

From table to cache

Memory Hierarchy Basics

Miss rate

Fraction of cache access that result in a miss

Causes of misses

Compulsory

First reference to a block – would also miss in an infinite cache. Another name = cold start miss.

Capacity

Blocks discarded and later retrieved – would also miss in a fully-associative cache with perfect replacement.

Conflict

Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache

Coherence

Miss caused by cache coherence (only in multiprocessors)

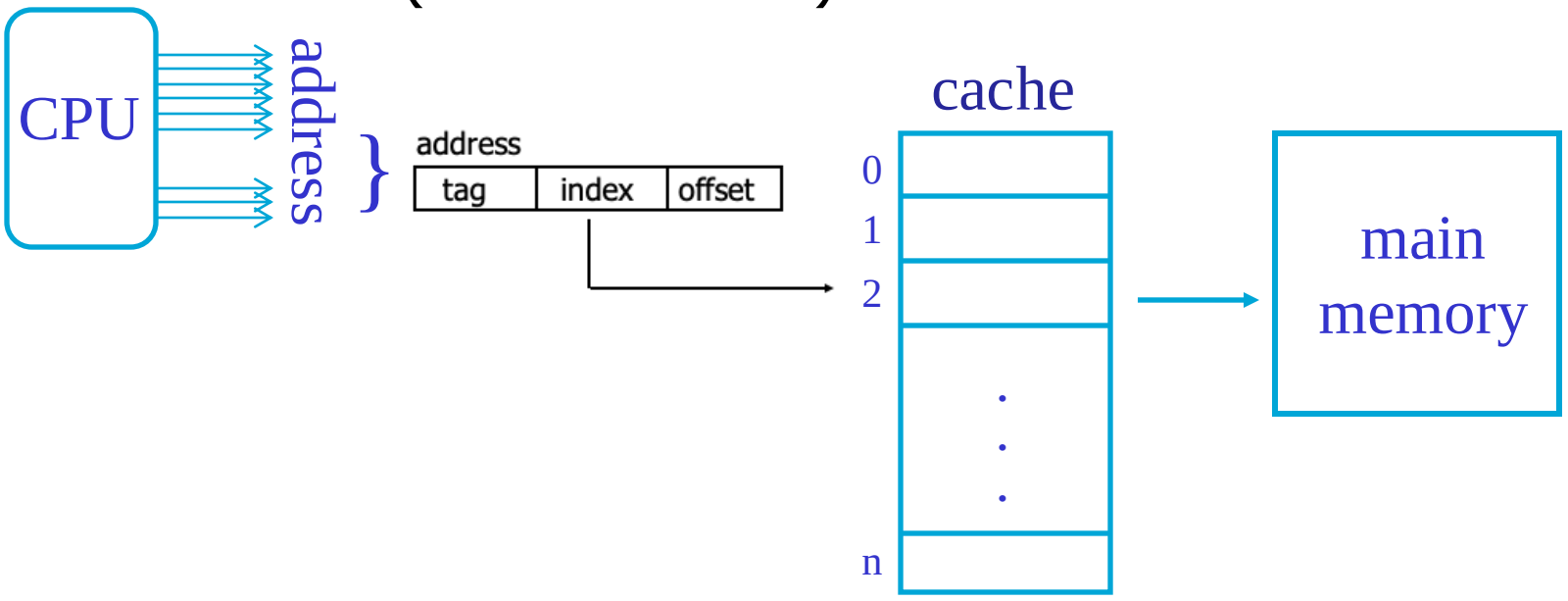
Block Placement (I)

A cache is just a smaller memory

not all program footprint fits in the cache

cache address ($addr) = some bits of data address

2 address locations may point to the same cache block
=> conflict (or: contention)

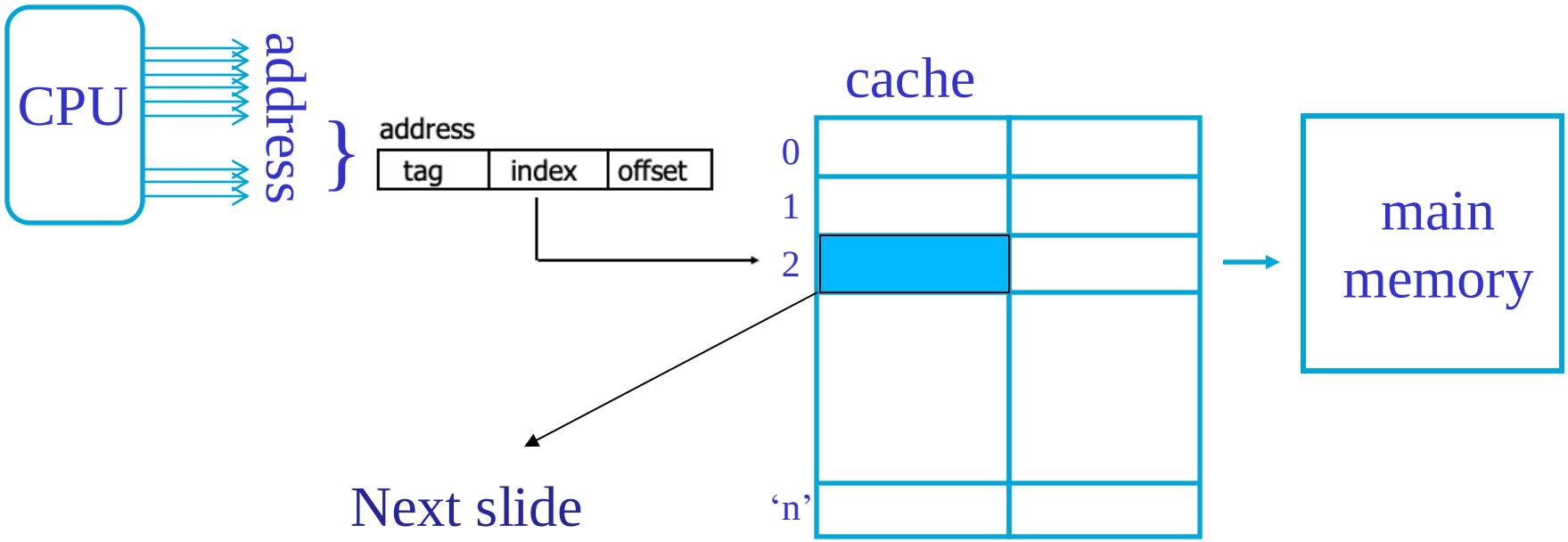


Block Placement (II)

2 address location may point to the same cache block => contention

a solution: associativity: more blocks have the same $addr (= index)

NOTE: usually not looking for doubling cache size, therefore, "height n" is reduced



Block Placement (III)

block address		(block)
tag	index	offset

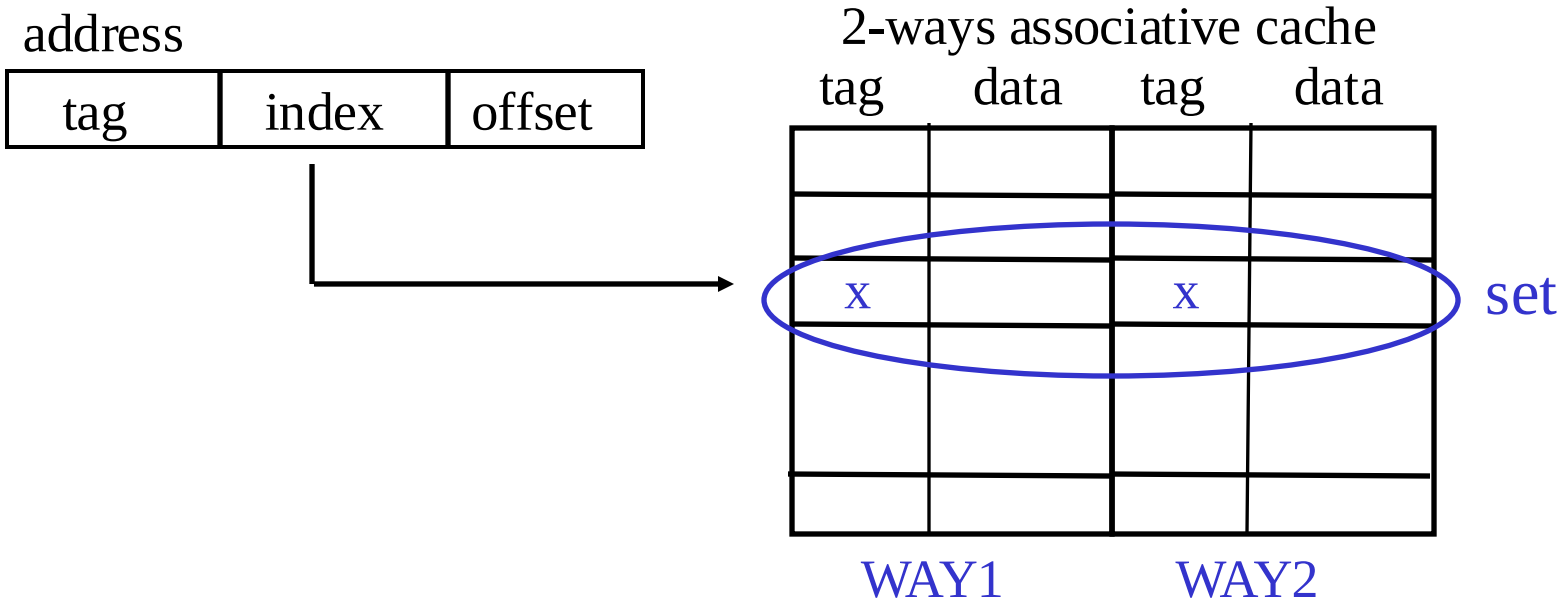
3 portions of an address:

block address = <tag, index>

the index addresses a cache set

the tag is compared to the tag stored in the cache

block offset – identifies the byte inside a cache block



Block Placement (IV)

block address		(block)
tag	index	offset

Useful formulas:

cache size = Nsets x associativity x block size

block address = (byte) address DIV block size in bytes

cache index = block address MOD Nsets

If block size and nsets are powers of two, DIV and MOD can be performed efficiently

DIV = most significant bits of the address

MOD = least significant bits of block address

Example 4-byte blocks:

Byte address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Block address	0				1				2				3				4	

Block Placement (V)

Direct-mapped cache

- number of sets (Nsets) = number of blocks in cache
- i.e., "index" points to a 'set' (=block) within cache
- "index" is 2log(Nsets)

Fully associative cache

- number of sets (Nsets) = 1
- Block can be placed anywhere
- Index portion of address consists of 0 bits

Set-associative cache

- block can be placed anywhere in the set
- *for reference, check figure 2 slides back and next slide*
- If there are **n** blocks in a set, cache is called n-way set associative

Direct Mapped and Associative Caches

Direct mapped

Index	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2-way set associative

Index	Tag	Data	Tag	Data
0				
1				
2				
3				

set

Fully associative cache

Tag	Data	T	D	T	D	T	D	T	D	T	D	T	D	T	D

Block Identification

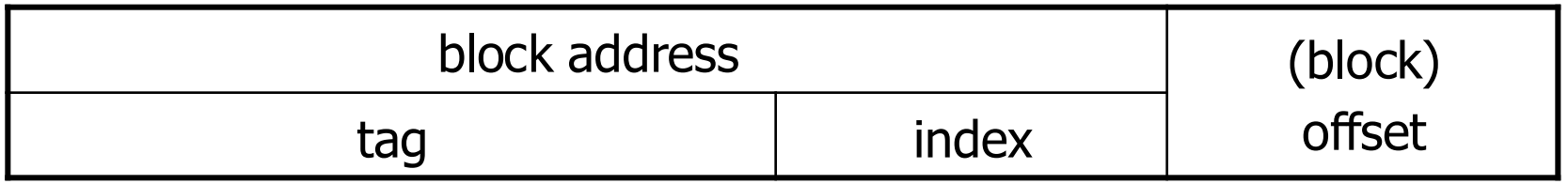
How is a block found in cache set?

- Each block has a tag
- Tag of the address of the requested data is **compared to** the tag(s) of the block(s) stored in the set
- No need to compare index or block offset

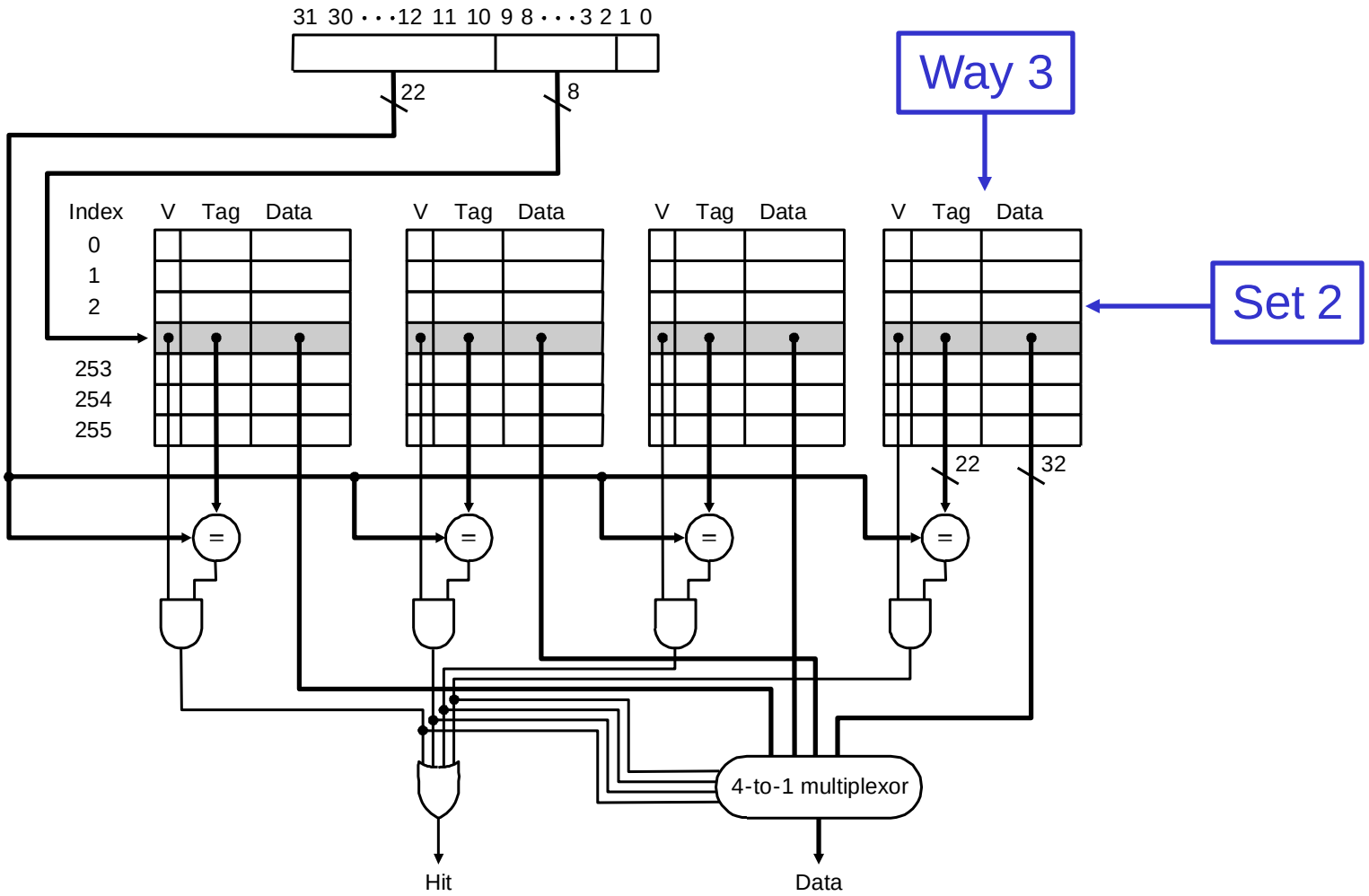
Increasing associativity shrinks index, expands tag

Each block also has valid bit

- indicates if block contains valid data
- needed for start-up, flushing cache after context switch



Block identification in a 4-way set associative cache



Block Replacement

No choice for Direct Mapped

Set Associative or Fully Associative:

- Random: replace a random block in the set
- FIFO (First In First Out): replace the block that has been brought in the longest ago
- LRU (Least Recently Used): replace the block that has not been used for the longest time

Quick Summary:

- Direct-mapped vs. set-associative caches
- Access to caches via address broken down to (tag,index,offset)

Now what? (Keep in mind, cache services the CPU)

- What happens during a read → hit/miss
- What happens during a write (← next couple of slides)

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.

wait for lower-level memory? No, use **write buffers!** *(see next slide for more details)*

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.

at replacement, is block clean (not written) or **dirty** (written)?

Pros and Cons of each?

WT: read misses cannot result in writes (to main mem)

WB: read misses do result in writes (to main mem)

WB: repeated writes to same location do not result in (unnecessary) accesses to main memory → WT: …

Write stall in write through caches

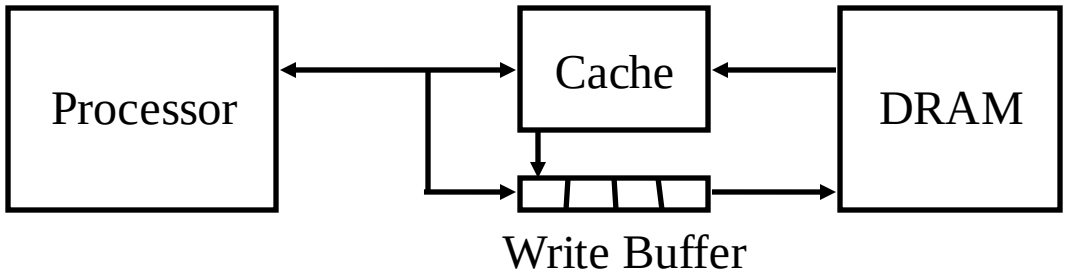
When the CPU must wait for writes to complete during write through, the CPU is said to **write stall**

Common optimization:

Write buffer which allows the processor to continue as soon as the data are written to the buffer, thereby overlapping processor execution with memory updating.

However, write stalls can occur even with a write buffer (when buffer is full).

Write Buffer for Write Through



A Write Buffer is needed between the Cache and Memory

- Processor: writes data into the cache and the write buffer memory controller: write contents of the buffer to memory

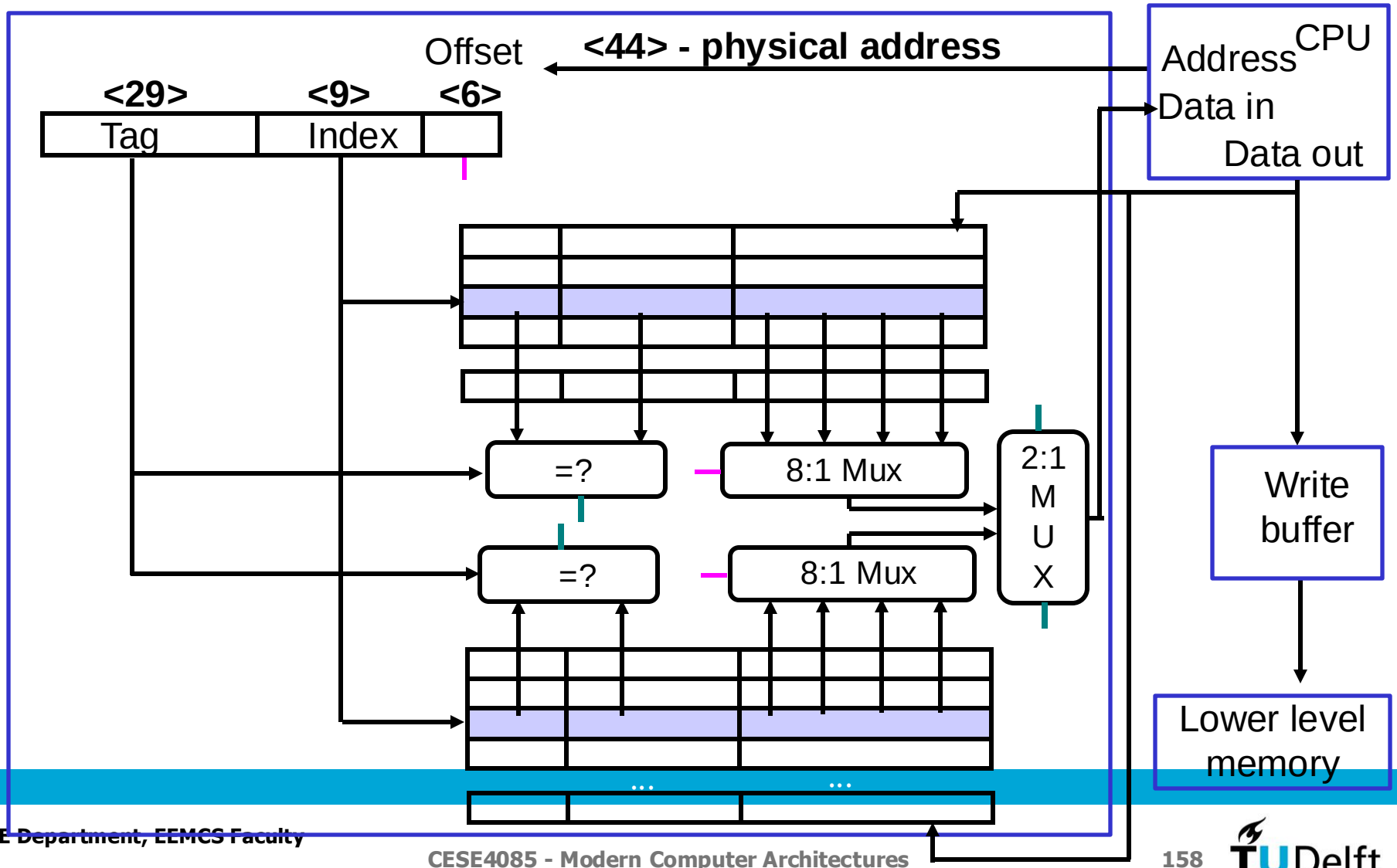
Write buffer is just a FIFO:

- Typical number of entries: 4
- Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle

Memory system designer's nightmare:

- Store frequency (w.r.t. time) -> 1 / DRAM write cycle
- Write buffer saturation

An Example: The Alpha 21264 Data Cache (64KB, 64-byte blocks, 2-way SA)



Cache Performance

Hit Time = time to find and retrieve data from cache
Hit Rate = % of requests found in cache
Miss Rate = 1 - Hit Rate
Miss Penalty = time to retrieve data on a miss

Average memory access time (AMAT):

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

Out-of-order complication: HitTime & MissPenalty vary
some accesses are overlapped (HitTime=Total HitTime–Overlapped HitTime)
some instructions do not retire (miss-predicted branches) –
NOTE: AMAT eq. becomes quite complex for OoO execution, check appendix for more details!

How to Improve Cache Performance?

Cache optimizations:

- Reduce miss rate (exploiting locality)
- Reduce miss penalty (using parallelism)
- Reduce hit time (using parallelism)

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

Classifying Misses: the 3 Cs

The 3 Cs:

Compulsory — The first access to a block is always a miss. Also called cold start misses.
(Misses in an infinite cache)

Capacity — If cache cannot contain all blocks needed, capacity misses will occur due to blocks being discarded and later retrieved.
(Misses in Fully Associative Cache with optimal replacement)

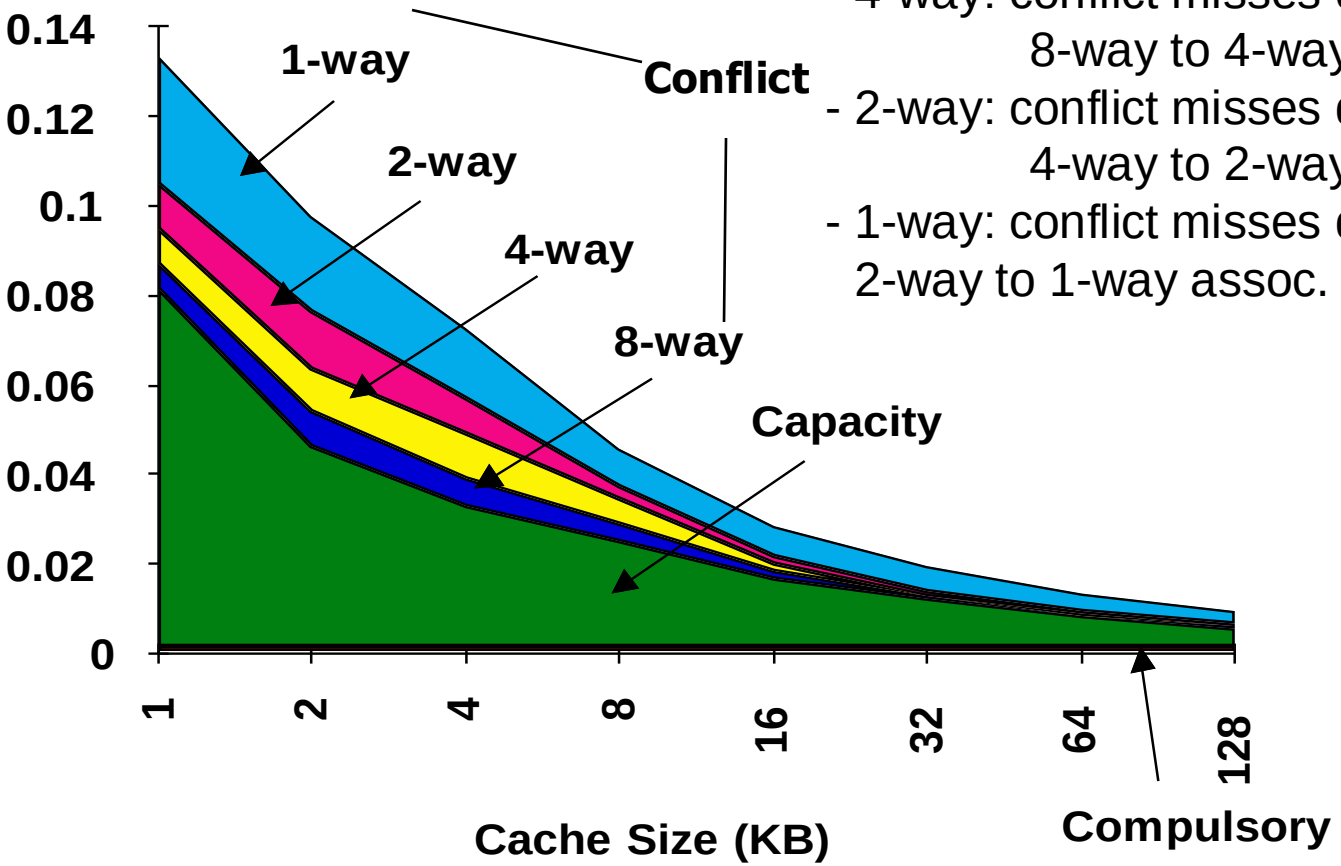
Conflict — Misses occurring because several blocks map to same set. Also called collision misses.
(All other misses)

More recent, 4th "C":

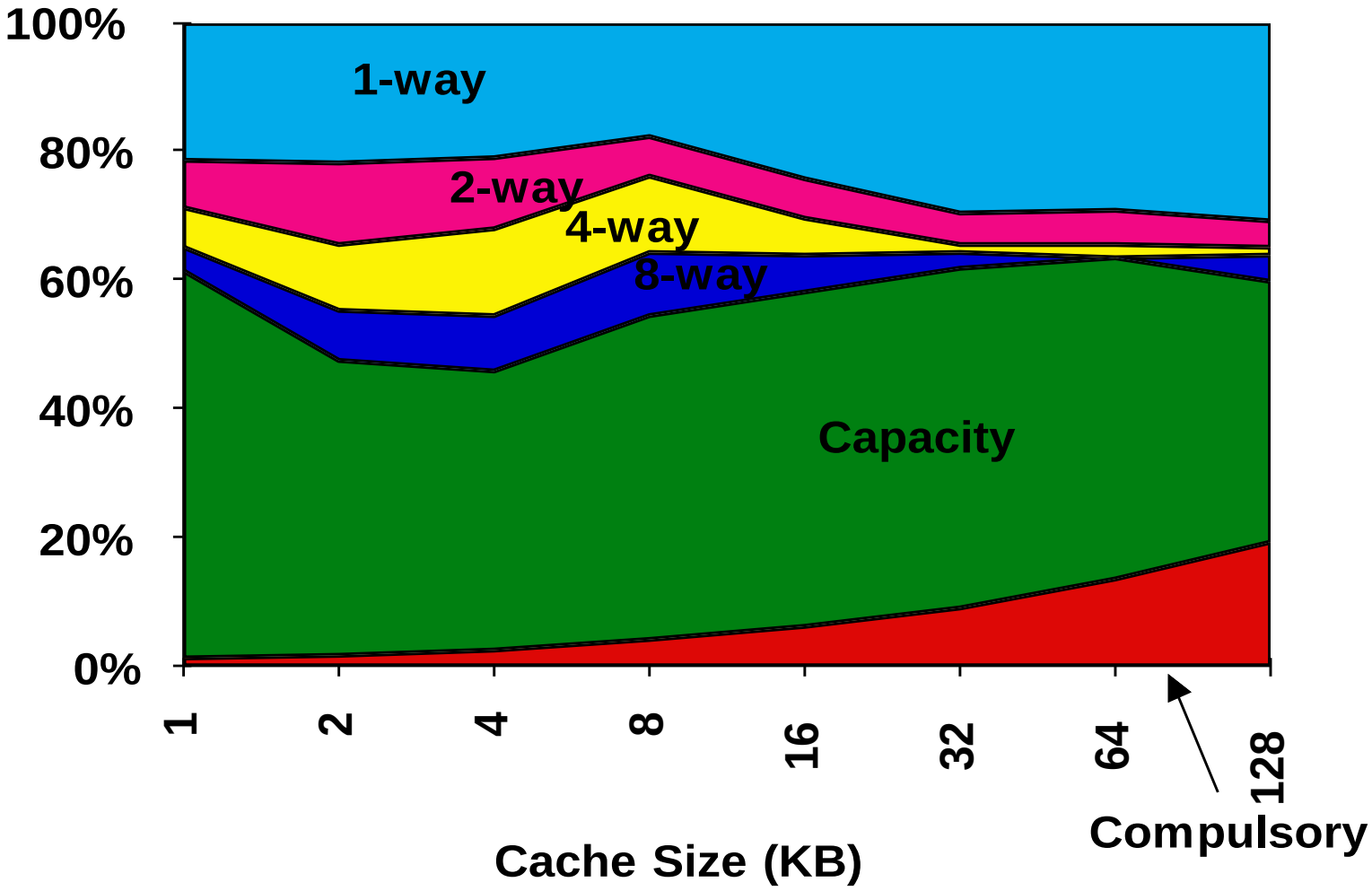
Coherence — Misses caused by cache coherence (only for multiprocessors).

3Cs Absolute Miss Rate (SPEC92)

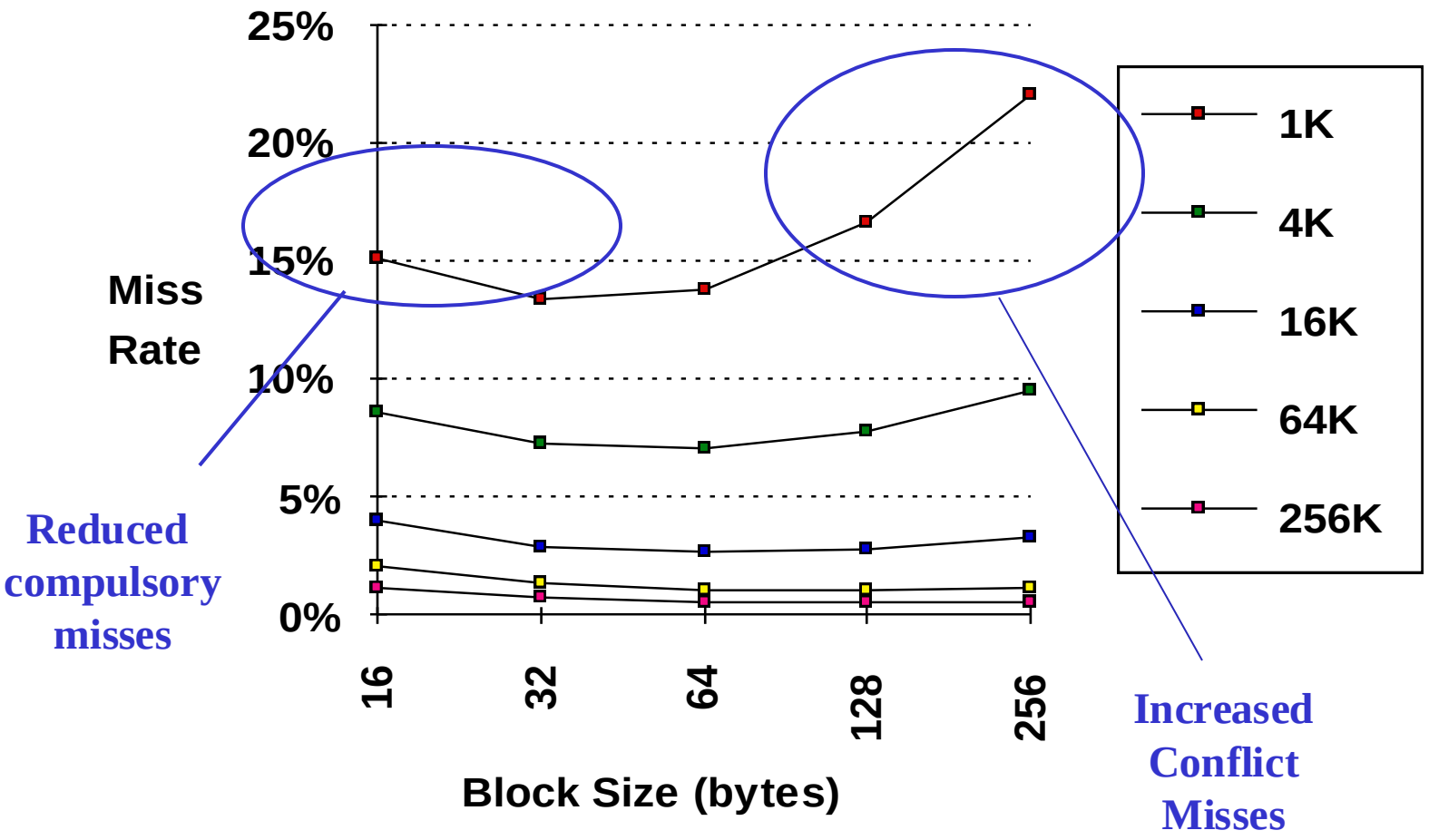
- 8-way: conflict misses due to going from fully associative to 8-way assoc.
- 4-way: conflict misses due to going from 8-way to 4-way assoc.
- 2-way: conflict misses due to going from 4-way to 2-way assoc.
- 1-way: conflict misses due to going from 2-way to 1-way assoc. (direct mapped)



3Cs Relative Miss Rate



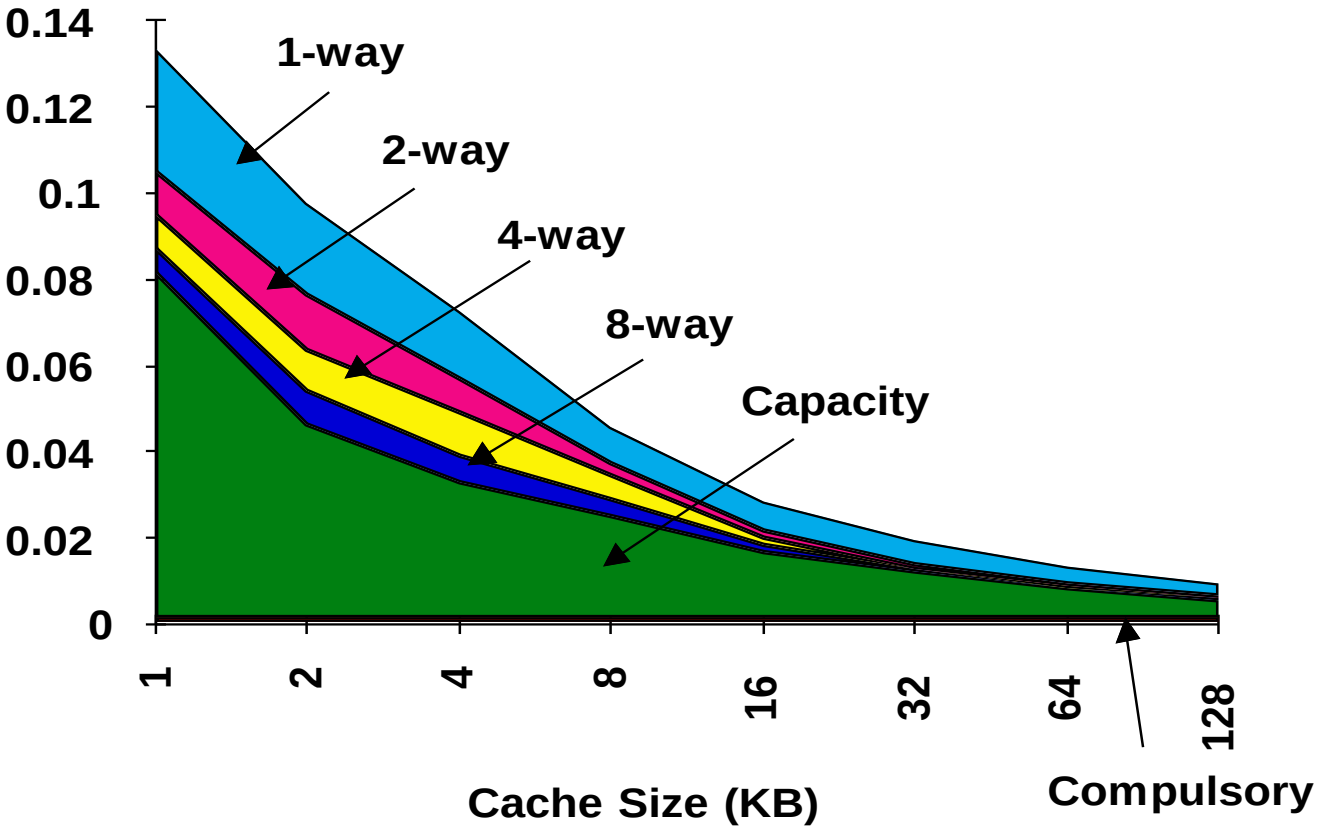
1st Miss Rate Reduction Technique: Larger Block Size



2nd Miss Rate Reduction Technique: Larger Caches

Pro: Reduces Capacity misses

Cons: Higher cost (P4 data $ smaller than P3 data $), Longer hit time



3rd Miss Rate Reduction Technique: Higher Associativity

Pro: reduces conflict misses (previous figure)

Two rules of thumb

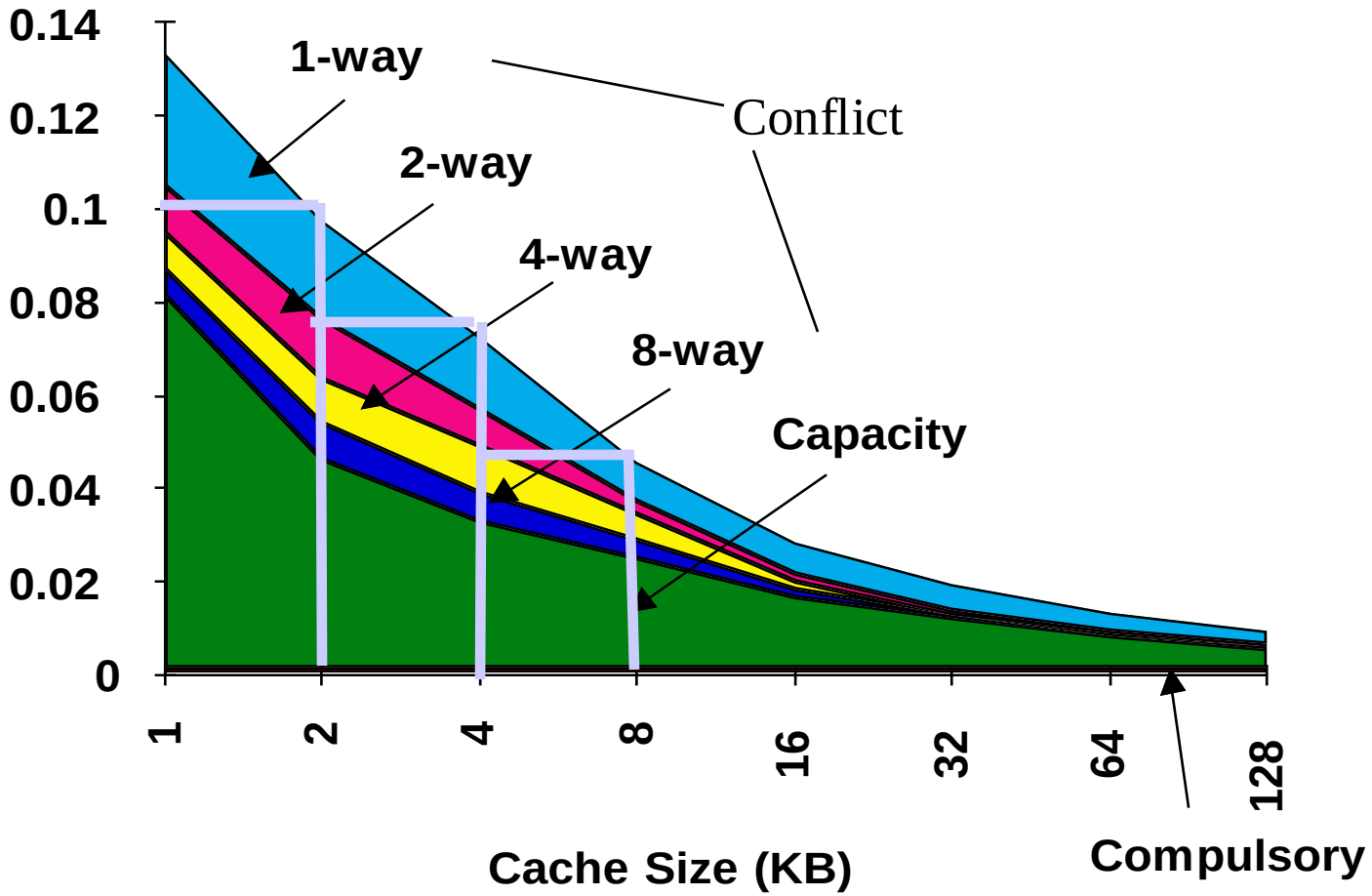
- 8-way set-associative almost as effective in reducing misses as fully-associative cache of the same size
- 2:1 Cache Rule: Miss Rate DM cache of size N = Miss Rate 2-way cache of size N/2

Beware: Execution time is the final measure!

- Will cycle time increase?

2:1 Cache Rule

Miss rate 1-way set associative cache size N
= Miss rate 2-way set associative cache size N/2



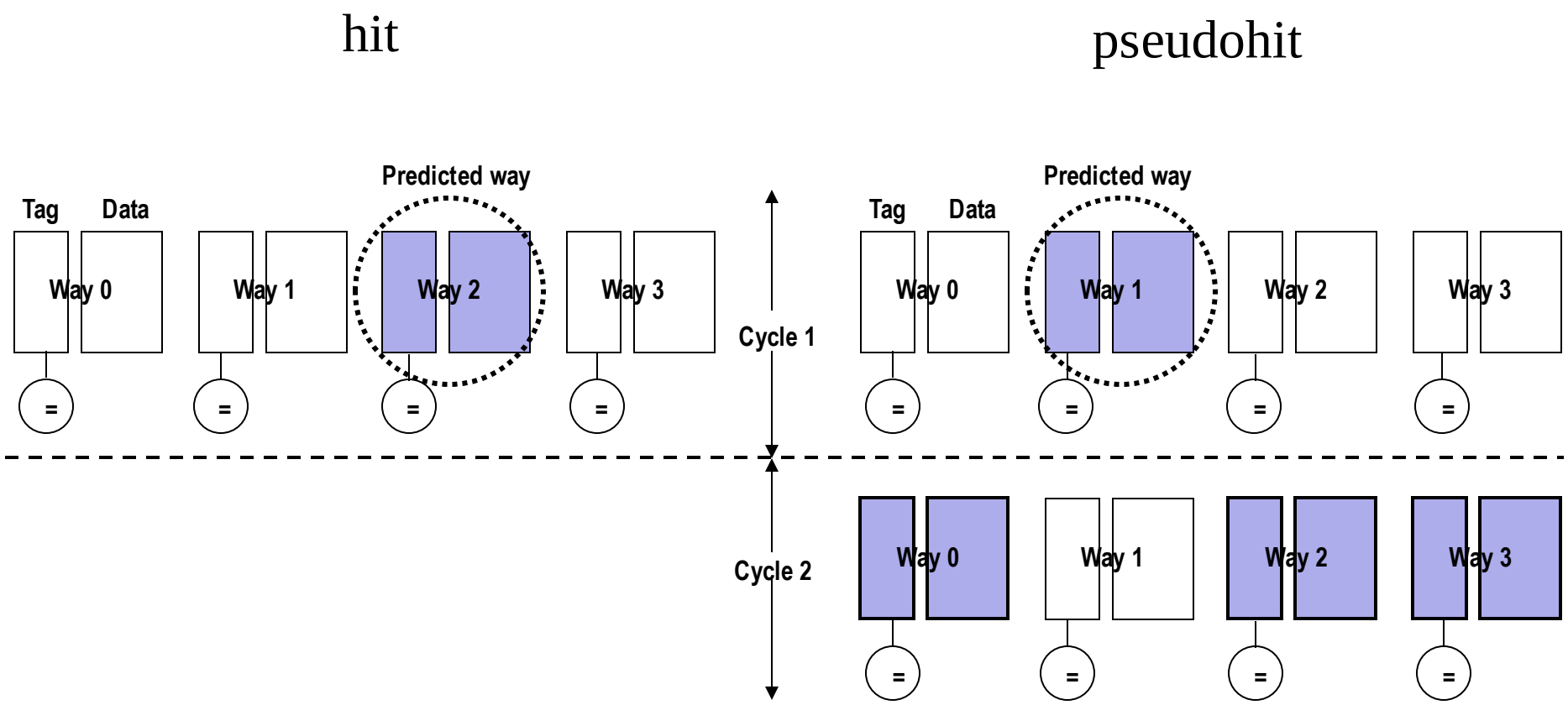
4th Miss Rate Reduction Technique: Way Prediction, Pseudo-Associative Cache

How to combine **fast hit time** of Direct Mapped and have the **lower conflict misses** of 2-way SA cache?

Way Prediction: extra bits are kept to predict the way or block within a set

- MUX is set early to select the desired block
- Only a single tag comparison is performed
- What if miss? => check the other blocks in the set
- Used in Alpha 21264 (1 bit per block in IC$)
  - **1 cc if predictor is correct, 3 cc if not**
  - Effectiveness: prediction accuracy is 85%
- Used in MIPS 4300 embedded processor to reduce power
  - In most cases, need to activate only one way

Way Prediction



5th Miss Rate Reduction Technique: Compiler Optimizations

Reduction comes from software (no hardware changes)

Instructions

- Reorder procedures in memory so as to reduce conflict misses
- Profiling to look at conflicts (using tools)

Data

- ***Merging Arrays***: improve spatial locality by single array of compound elements (Array of Structures (AoS)) 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

Motivation: some programs reference multiple arrays in the same dimension with the same indices at the same time => these accesses can interfere with each other, leading to conflict misses.

Solution: combine these independent matrices into a single compound array, so that a single cache block can contain the desired elements.

E.g.,:

Array1[i] && Array2[i] in single block (see next slide)

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
```

Reduces conflicts between val and key arrays and improves spatial locality

Loop Interchange

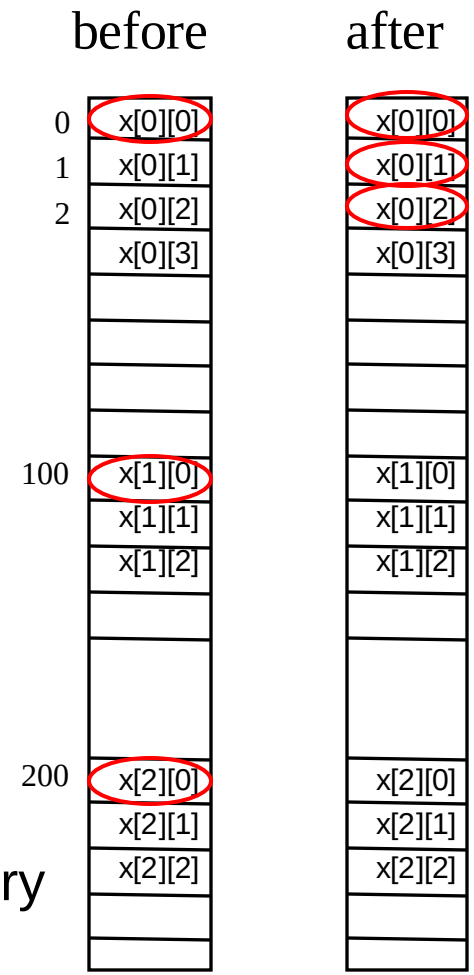
Motivation: some programs have nested loops that access data in nonsequential order

Solution: Simply exchanging the nesting of the loops can make the code access the data in the order it is stored => reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded

Loop Interchange Example

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



- Row major order in memory
- Sequential accesses instead of striding through memory every 100 words; improved spatial locality.

Loop Fusion

Some programs have separate sections of code that access with the same loops, performing different computations **on the common data**

Solution:
"Fuse" the code into a single loop =>
the data that are fetched into the cache can be used repeatedly before being swapped out => reducing misses via improved temporal 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];
   }
```

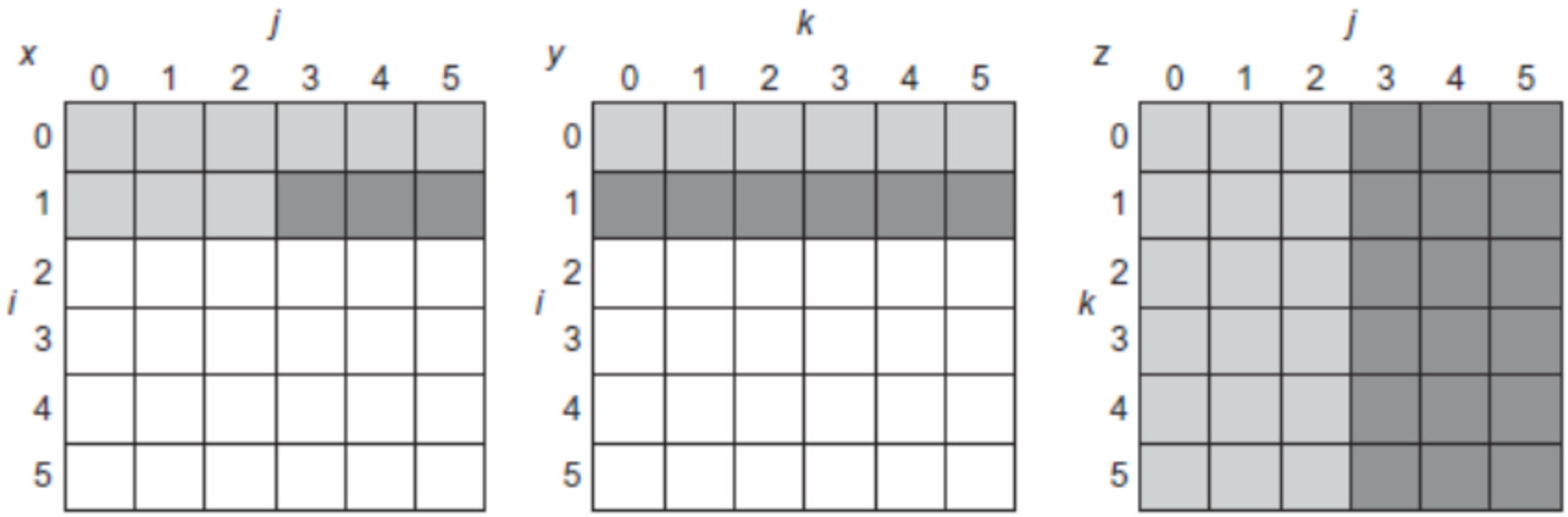
Blocking

Motivation: multiple (LARGE) arrays, some accessed by rows and some by columns

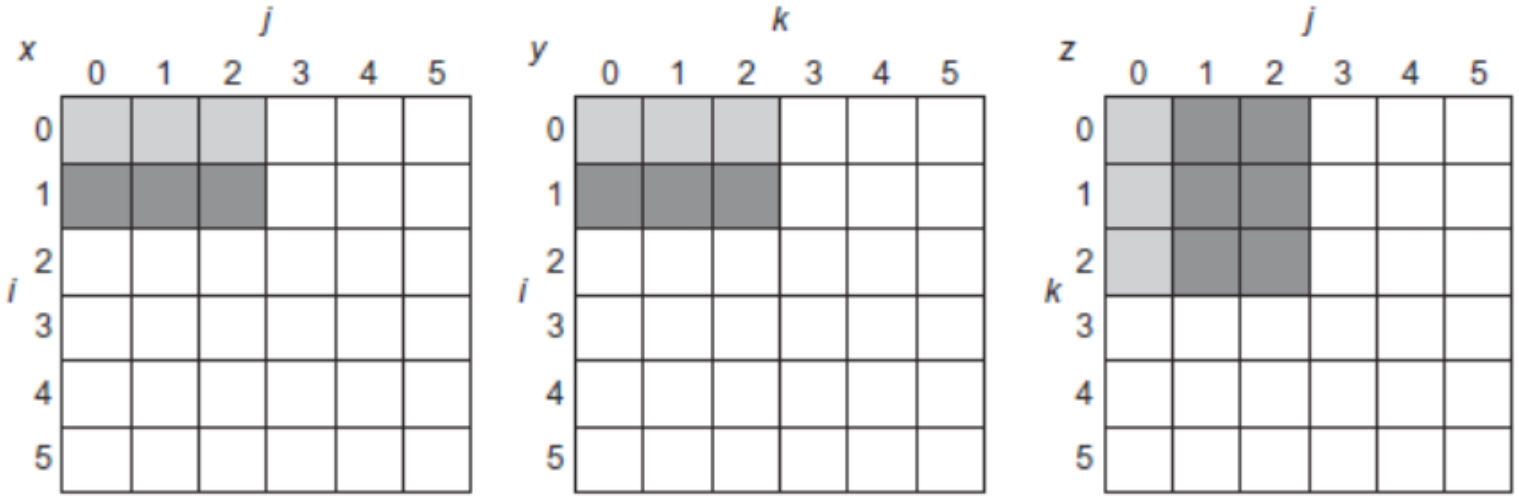
Storing the arrays row by row (row major order) or column by column (column major order) does not help: both rows and columns are used in every iteration of the loop (Loop Interchange cannot help)

Solution: instead of operating on entire rows and columns of an array, blocked algorithms operate on **submatrices or blocks** => maximize accesses to the data loaded into the cache before the data are replaced

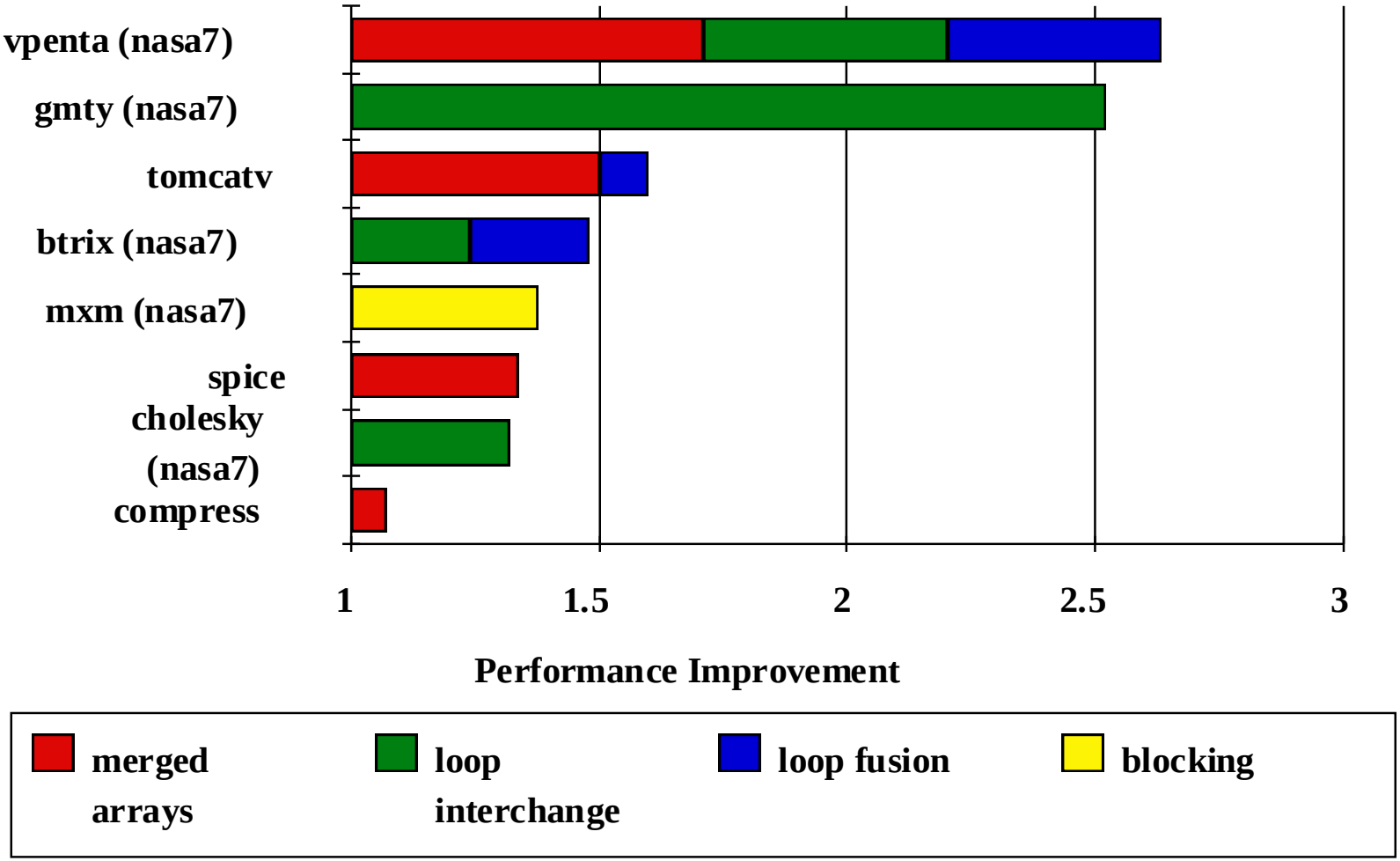
Blocking



Blocking



Summary of Compiler Optimizations to Reduce Miss Rate (by hand)



Summary: Miss Rate Reduction

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

3 Cs: Compulsory, Capacity, Conflict

- Larger Cache => Reduce Capacity
- Larger Block Size => Reduce Compulsory
- Higher Associativity => Reduce Conflicts
- Way Prediction & Pseudo-Associativity
- Compiler Optimizations

Reducing Miss Penalty

Motivation

AMAT = HitTime + MissRate x *MissPenalty*

Technology trend =>
relative cost of miss penalties increases over time

Techniques that address miss penalties

- Multilevel Caches
- Critical Word First and Early Restart
- Giving Priority to Read Misses over Writes
- Merging Write Buffer
- Victim Caches

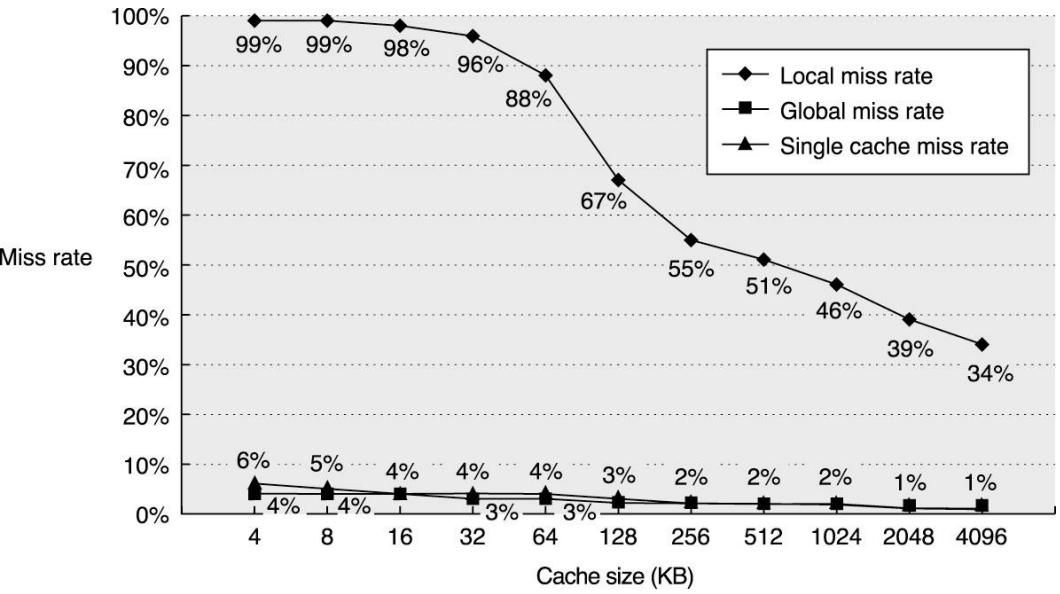
1st Miss Penalty Reduction Technique: Multilevel Caches

- Architect's dilemma
  - Make cache faster to keep pace with speed of CPU?
  - Make cache larger to overcome widening gap between CPU and main memory?
  - Both! L1 cache small and simple enough to match cycle time, L2 cache large enough to capture most misses.
- L2 Equations
  - $AMAT = HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1}$
  - $MissPenalty_{L1} = HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}$
  - $AMAT = HitTime_{L1} + MissRate_{L1} \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2})$
- Definitions:
  - Local miss rate — misses in this cache divided by the total number of memory accesses to this cache ($MissRate_{L1}$,$MissRate_{L2}$)
  - Global miss rate — misses in this cache divided by the total number of memory accesses generated by the CPU (L2: $MissRate_{L1} \times MissRate_{L2}$)
  - Global Miss Rate is what matters most: what fraction of the memory accesses go all the way to memory.

1st Miss Penalty Reduction Technique: Multilevel Caches

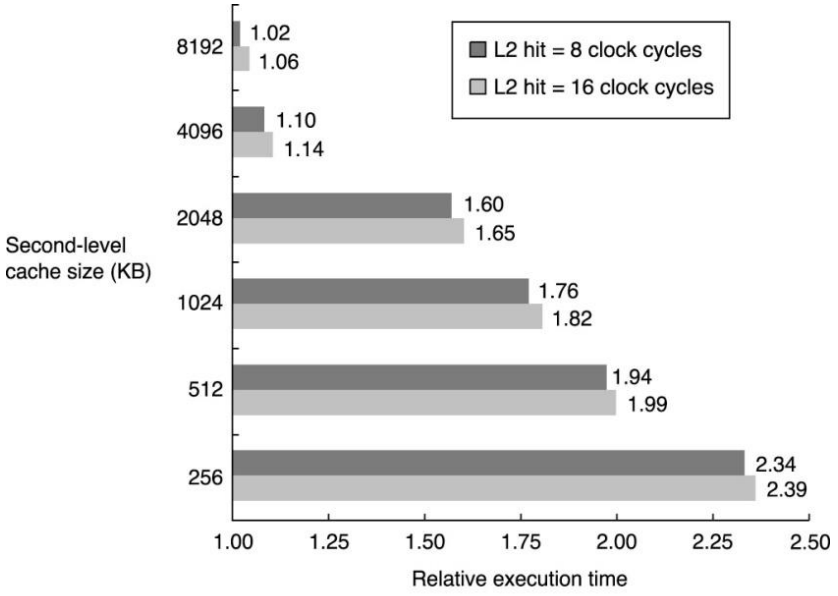
Global vs. Local Miss Rate L2

(L1 64KB)



Relative Execution Time

1.0 is 8MB L2, 1cc hit

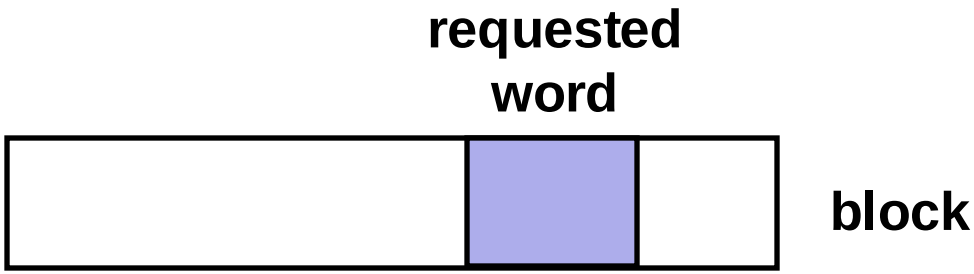


2nd Miss Penalty Reduction Technique: Early Restart and Critical Word First

Don't wait for full block to be loaded before restarting CPU

- Early restart — As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- Critical Word First — Request 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.
- Generally useful only in large blocks

Problem of spatial locality: tend to want next sequential word, so not clear if benefit by early restart and CWF



3rd Miss Penalty Reduction Technique: Read Priority over Write on Miss

- Serves reads before writes have been completed
- **Write-through with write buffers** have RAW conflicts with main memory reads on cache misses (they might hold the updated value of a location needed on a read miss)

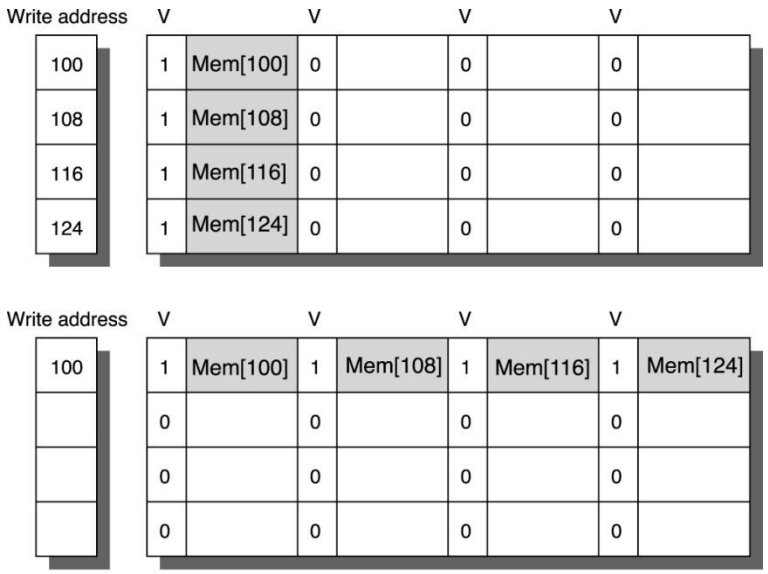
Example: DM, WT, 512 & 1024 map to the same block:

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

  - If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
  - Check write buffer contents before read; if no conflicts, let read miss continue
- **Write-back** also wants write buffer (not present in original idea) to hold evicted blocks
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and 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 does the read

4th Miss Penalty Reduction Technique: Merging Write Buffer

- Write Through caches rely on write-buffers (WBs)
  - on write, data and full address are written into the buffer; write is finished from the CPU's perspective
  - Problem: WB full stalls
- Write merging
  - multiword writes faster than single word writes
  - Write buffer less often full



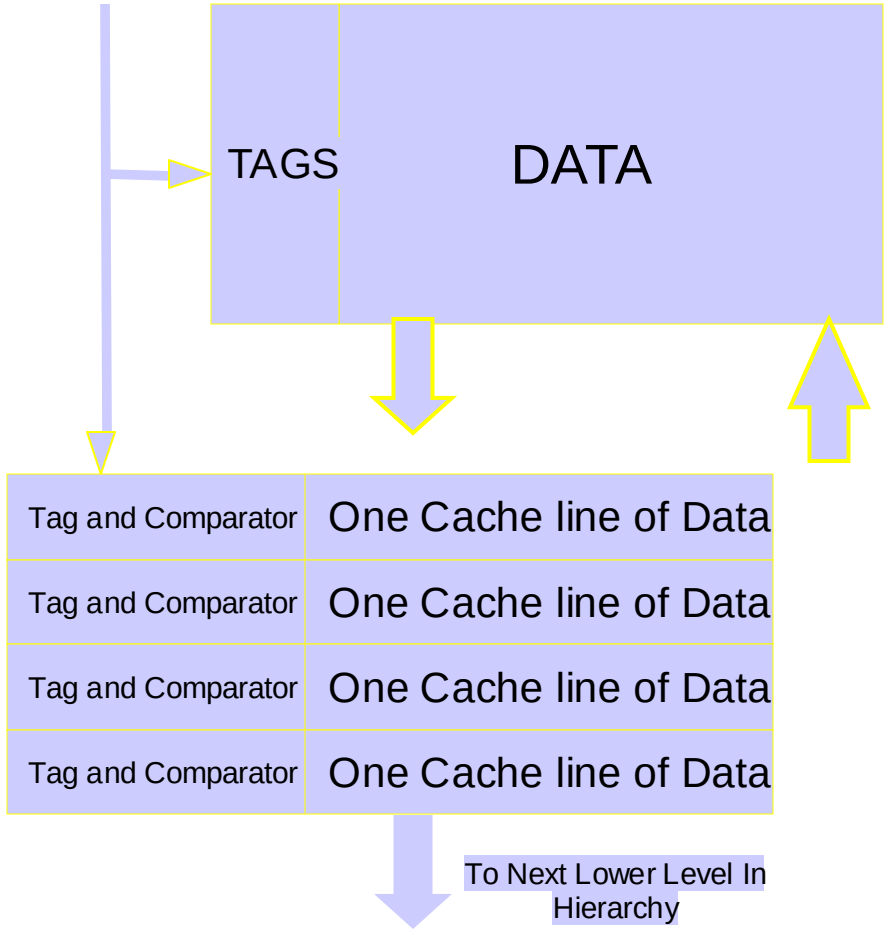
5th Miss Penalty Reduction Technique: Victim Caches

How to combine fast hit time of direct mapped yet still avoid conflict misses?

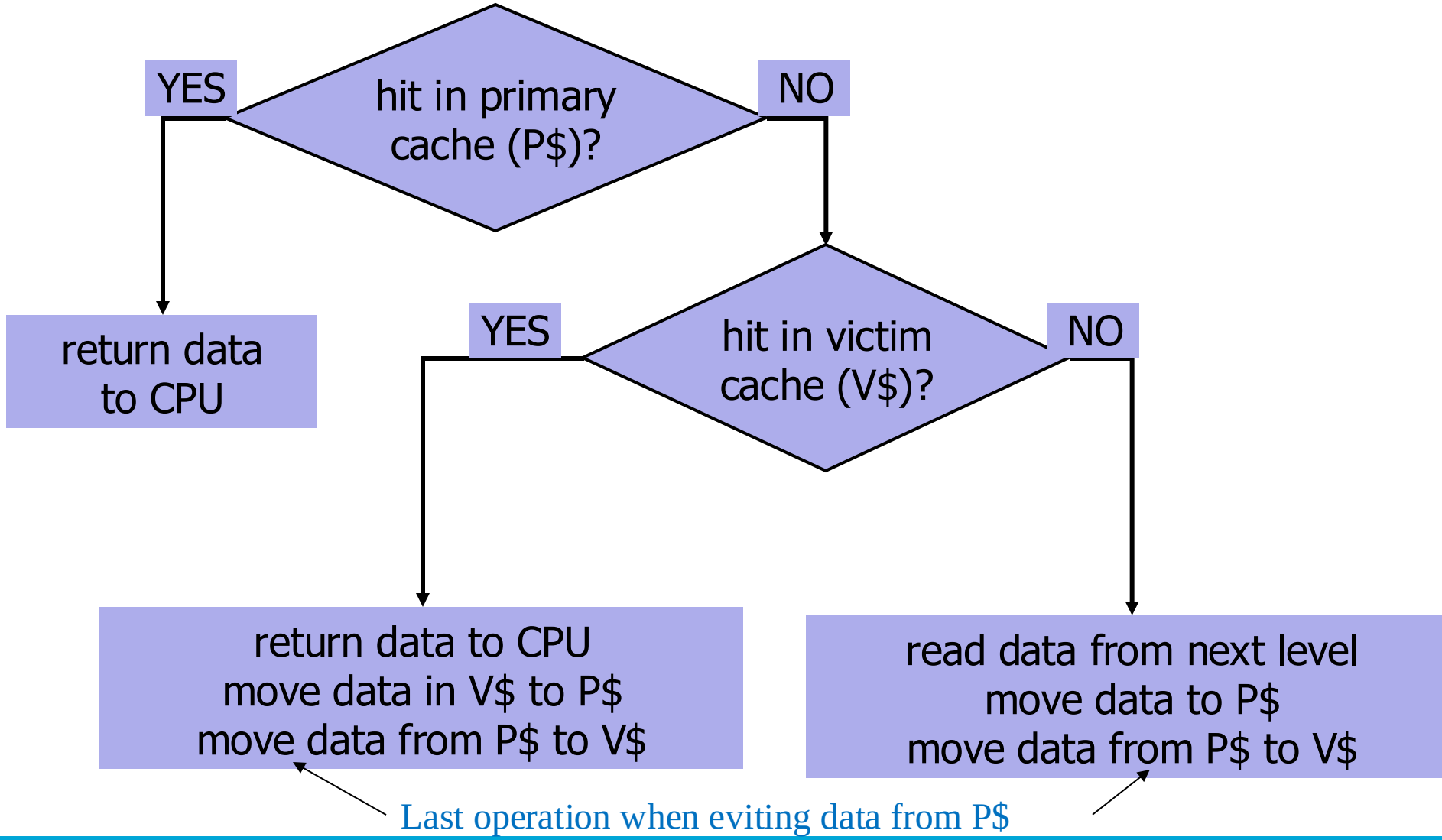
Idea: Add buffer to place data evicted from cache in case it is needed again

Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4KB direct mapped data cache

Used in Alpha, HP machines, AMD Athlon (8 entries)



Victim Cache



Summary of Miss Penalty Reduction Techniques

Multilevel Caches

Critical Word First and Early Restart

Giving Priority to Read Misses over Writes

Merging Write Buffer

Victim Caches

Reducing Cache Miss Penalty or Miss Rate via Parallelism

Idea: overlap the execution of instructions with activity in memory hierarchy

Miss Rate/Penalty reduction techniques

- Nonblocking caches
  - reduce stalls on cache misses in CPUs with out-of-order completion
- Hardware prefetching of instructions and data
  - reduce miss penalty
- Compiler controlled prefetching

Reduce Misses/Penalty: 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

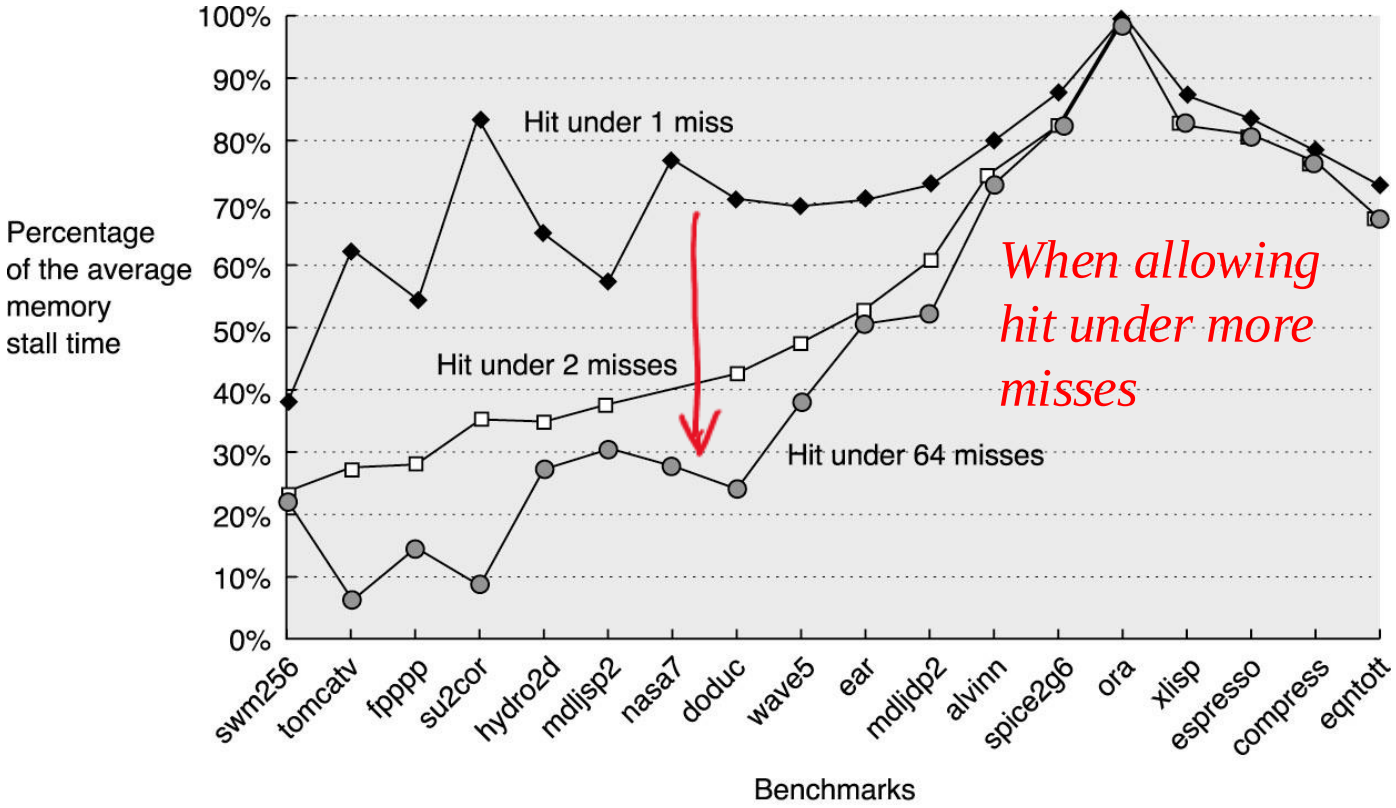
- requires F/E bits on registers or out-of-order execution
- requires multi-bank memories

"hit under miss" reduces the effective miss penalty by working during miss vs. ignoring CPU requests

"hit under multiple miss" or "miss under miss" may further lower the effective miss penalty by overlapping multiple misses

- Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
- Requires multiple memory banks (otherwise cannot support parallel access)
- e.g., Pentium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC



Reducing Misses/Penalty by Hardware Prefetching of Instructions & Data

Prefetching: fetch instructions or data before they are needed

Instruction Prefetching:

- Alpha 21064 fetches 2 blocks on a miss
- Extra block placed in "**stream buffer**"
- On miss check stream buffer

Data prefetching:

- Jouppi [1990] 1 **data stream buffer** got 25% misses from 4KB cache; 4 streams got 43%
- Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches

Prefetching relies on having extra memory bandwidth that can be used without penalty

If prefetched data are placed in cache, can cause cache pollution

Reducing Misses/Penalty by Software Prefetching

Extend ISA with prefetch instructions

Compiler must insert them to fetch data before they are needed

Data prefetching comes in two flavors:

- Register prefetch: Load data into register (HP PA-RISC loads)
- Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- Prefetching instructions cannot cause page faults; a form of speculative execution

Issuing Prefetch Instructions takes time and may pollute the cache

- Is cost of prefetch issues < savings in reduced misses?

Review: Improving Cache Performance

Reduce the miss rate,

Reduce the miss penalty, or

Reduce the time to hit in the cache.

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

1st Hit Time Reduction Technique: Small and Simple Caches

Smaller and simpler hardware is faster => small, direct-mapped or XOR-mapped cache helps hit time

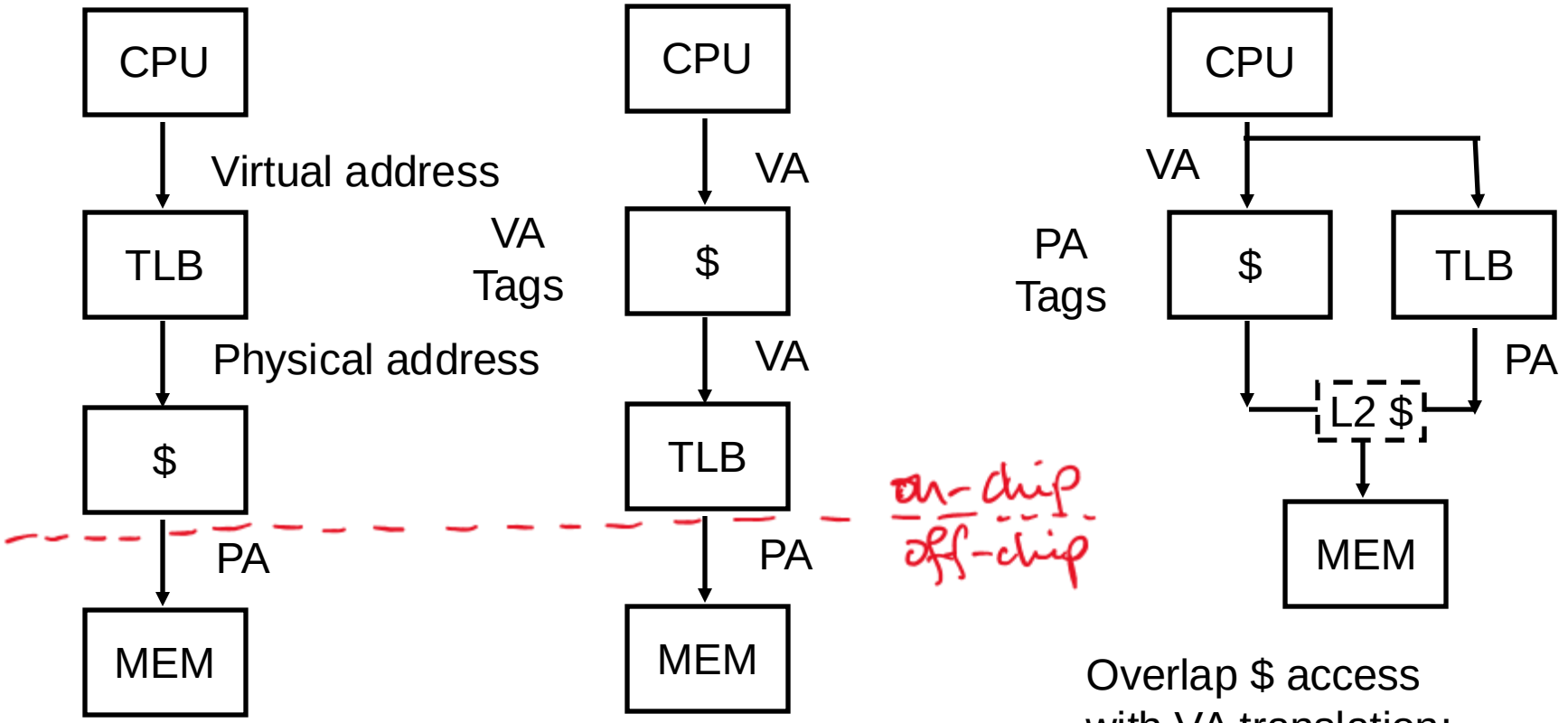
Keep cache small enough to fit on same chip as processor (avoid time penalty of going off-chip)

Keep cache simple enough to have hit time of 1 cc

- Use Direct Mapped cache: it overlaps the tag check with the transmission of data
- Set associative cache cannot transfer data before tag check completed

2nd Hit Time Reduction Technique: Avoiding Address Translation

TLB = time consuming



Conventional Organization

Virtually Addressed Cache
Translate only on miss
Synonym Problem

Overlap $ access with VA translation: requires $ index to remain invariant across V->P translation

2nd Hit Time Reduction Technique: Avoiding Address Translation

Send virtual address to cache? Called Virtually Addressed Cache or just Virtual Cache vs. Physical Cache

- Every time process is switched logically **must flush the cache**; otherwise get false hits
  - Cost is time to flush + "compulsory" misses from empty cache
- Must deal with aliases (sometimes called synonyms);
  Two different virtual addresses map to same physical address => multiple copies of the same data in a virtual cache

Solution to cache flush

- Add process identifier tag that identifies process as well as address within process: can't get a hit if wrong process

Split caches

In classic pipeline, in one cc must read instruction and load or store data

IF ID EX MEM WB

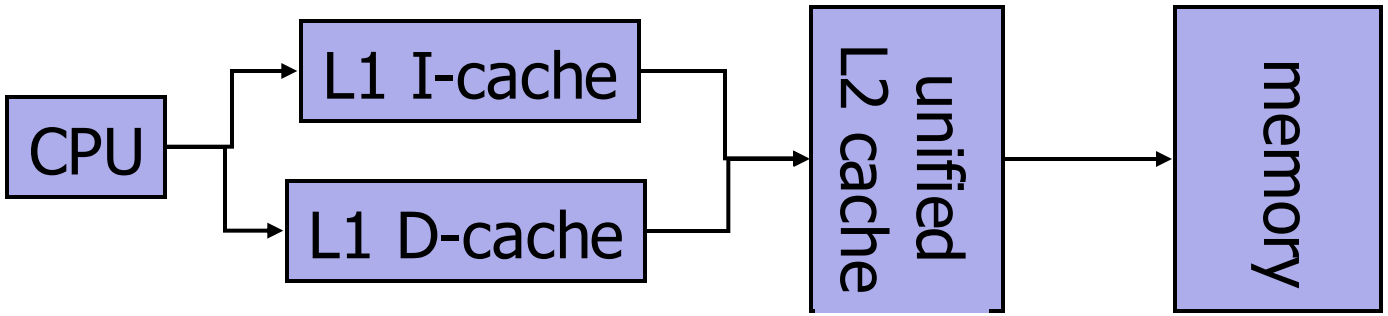
Potential structural hazard

Solutions:

Serialize accesses => wastes cycles

Use multiport cache => slower, consume more power, more sensitive to hw faults

Use a split I-cache and D-cache



Embedded Caches

- Caches increase average case performance
- Not sufficient in real-time environments
  - Instruction caches widely used in embedded processors
  - Data caches real issue
  - Solution 1: employ scratchpad memory
    - fast, addressable memory
    - compiler or application developer must explicitly place frequently used data in scratchpad
  - Solution 2: lock portions of the cache
    - locked data cannot be replaced
- Cache access consumes less power than memory access
  - To further improve power efficiency, some techniques used for power
  - MIPS 4300 embedded processor employs way prediction
- Scratchpad consumes less power than cache

Cache Optimization Summary

Technique	MR	MP	HT	Complexity
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
Priority to Read Misses		+		1
Early Restart & Critical Word 1st		+		2
Non-Blocking Caches		+		3
Second Level Caches		+		2
Better memory system		+		3
Small & Simple Caches	-		+	0
Avoiding Address Translation			+	2
Pipelining Caches			+	2

Consider

Intermezzo

- Until now → followed structure of earlier versions of book
- From now → follow 6th version and skip overlap

Memory Hierarchy Basics

$$\frac{Misses}{Instruction} = \frac{Miss\,rate \times Memory\,accesses}{Instruction\,count} = Miss\,rate \times \frac{Memory\,accesses}{Instruction}$$

$$Average\,memory\,access\,time = Hit\,time + Miss\,rate \times Miss\,penalty$$

Speculative and multithreaded processors may execute other instructions during a miss

- Reduces performance impact of misses

Memory Hierarchy Basics

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
- Higher number of cache levels
  - Reduces overall memory access time
- Giving priority to read misses over writes
  - Reduces miss penalty
- Avoiding address translation in cache indexing
  - Reduces hit time

Memory Technology and Optimizations

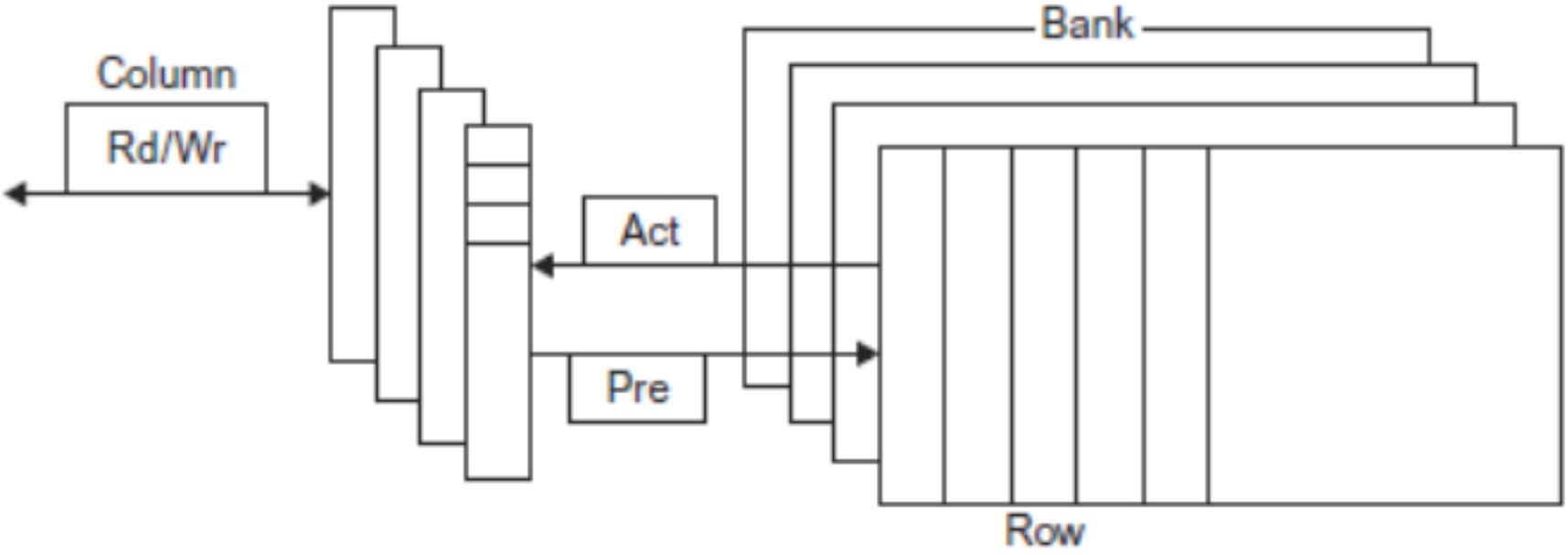
- Performance metrics
  - Latency is concern of cache
  - Bandwidth is concern of multiprocessors and I/O
  - Access time
    - Time between read request and when desired word arrives
  - Cycle time
    - Minimum time between unrelated requests to memory
- SRAM memory has low latency, use for cache
- Organize DRAM chips into many banks for high bandwidth, use for main memory

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 refeshed
  - Every ~ 8 ms (roughly 5% of time)
  - Each row can be refreshed simultaneously
- One transistor/bit
- Address lines are multiplexed:
  - Upper half of address: row access strobe (RAS)
  - Lower half of address: column access strobe (CAS)

Internal Organization of DRAM



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

			Best case access time (no precharge)			Precharge needed
Production year	Chip size	DRAM type	RAS time (ns)	CAS time (ns)	Total (ns)	Total (ns)
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39

Memory Optimizations

Standard	I/O clock rate	M transfers/s	DRAM name	MiB/s/DIMM	DIMM name
DDR1	133	266	DDR266	2128	PC2100
DDR1	150	300	DDR300	2400	PC2400
DDR1	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	1333	2666	DDR4-2666	21,300	PC21300

Memory Optimizations

DDR:

- DDR2
  - Lower power (2.5 V -> 1.8 V)
  - Higher clock rates (266 MHz, 333 MHz, 400 MHz)
- DDR3
  - 1.5 V
  - 800 MHz
- DDR4
  - 1-1.2 V
  - 1333 MHz

GDDR5 is graphics memory based on DDR3

Memory Optimizations

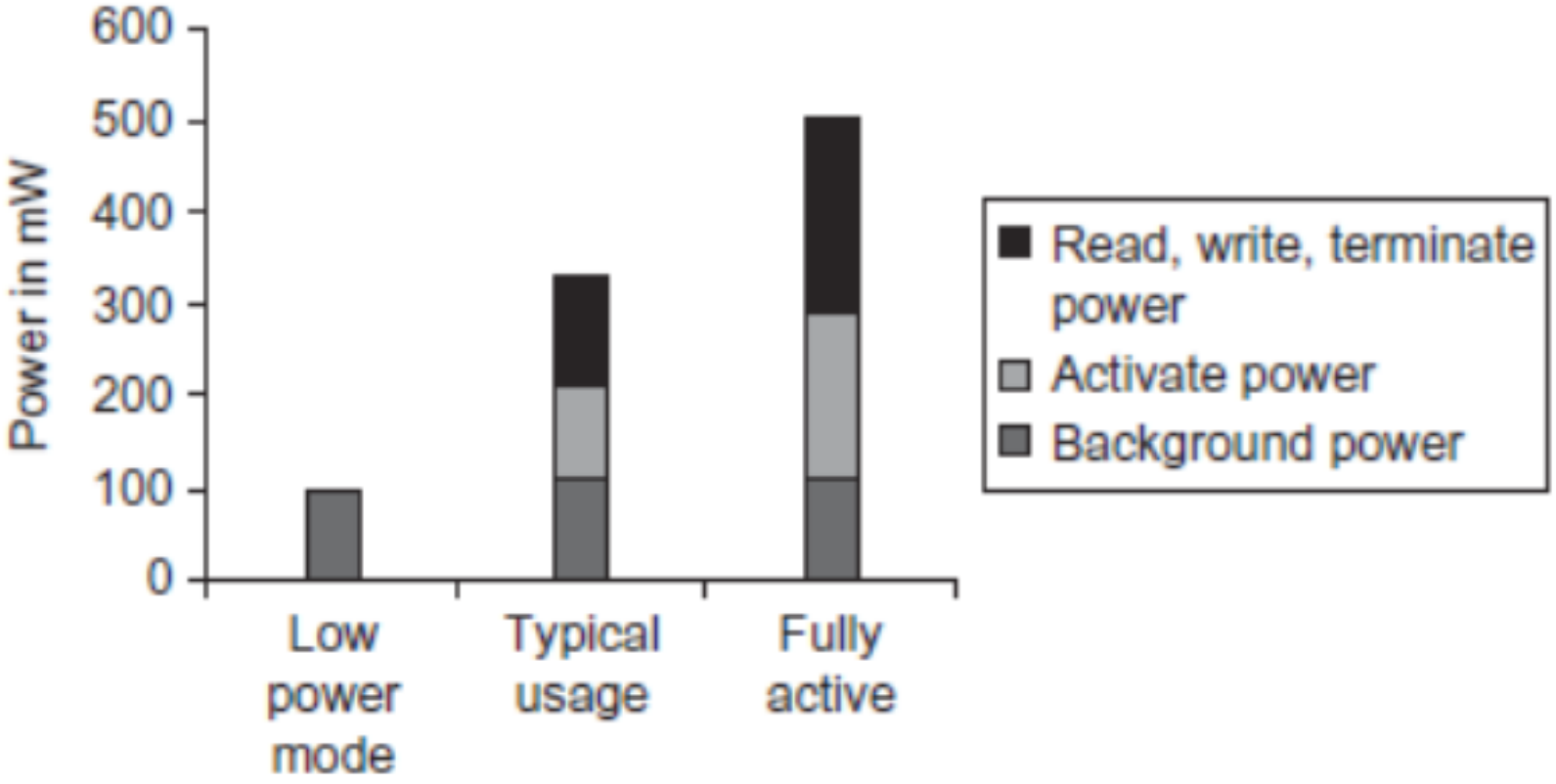
Reducing power in SDRAMs:

- Lower voltage
- Low power mode (ignores clock, continues to refresh)

Graphics memory:

- Achieve 2-5 X bandwidth per DRAM vs. DDR3
  - Wider interfaces (32 vs. 16 bit)
  - Higher clock rate
    - Possible because they are attached via soldering instead of socketted DIMM modules

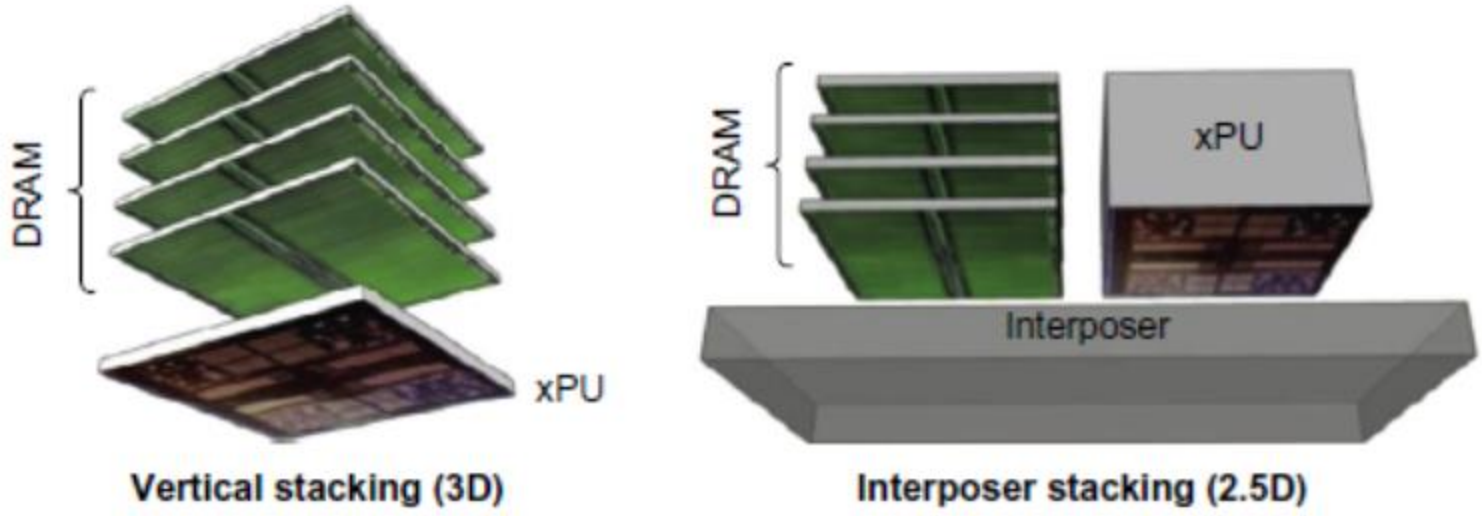
Memory Power Consumption



Stacked/Embedded DRAMs

Stacked DRAMs in same package as processor

- High Bandwidth Memory (HBM)



Flash Memory

Type of EEPROM

Types: NAND (denser) and NOR (faster)

NAND Flash:

- Reads are sequential, reads entire page (.5 to 4 KiB)
- 25 us for first byte, 40 MiB/s for subsequent bytes
- SDRAM: 40 ns for first byte, 4.8 GB/s for subsequent bytes
- 2 KiB transfer: 75 uS vs 500 ns for SDRAM, 150X slower
- 300 to 500X faster than magnetic disk

NAND Flash Memory

Must be erased (in blocks) before being overwritten

Nonvolatile, can use as little as zero power

Limited number of write cycles (~100,000)

$2/GiB, compared to $20-40/GiB for SDRAM and $0.09 GiB for magnetic disk

Phase-Change/Memrister Memory

- Possibly 10X improvement in write performance and 2X improvement in read performance

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 spare rows to replace defective rows

Chipkill: a RAID-like error recovery technique

Advanced Optimizations

Reduce hit time
- Small and simple first-level caches
- Way prediction

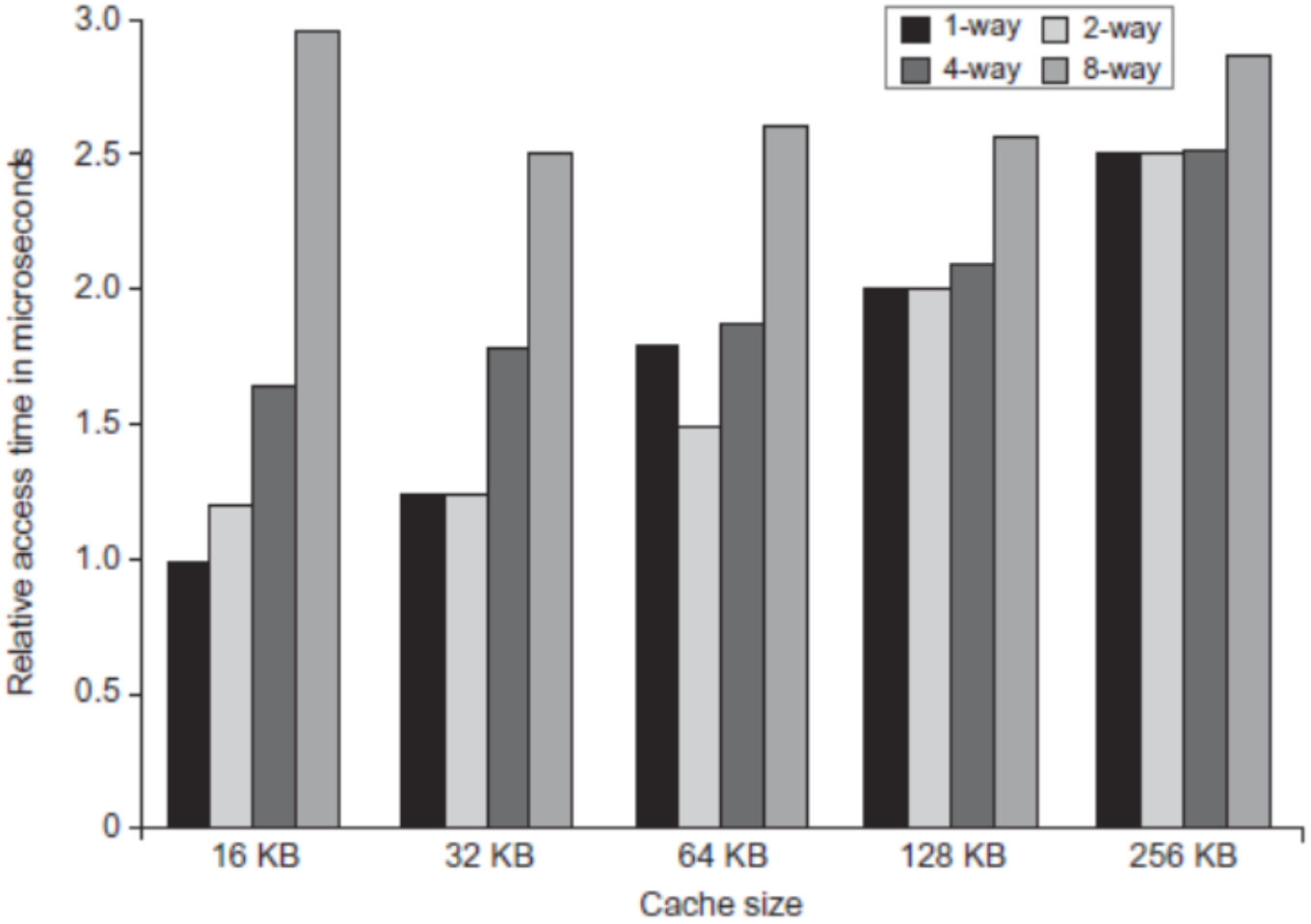
Increase bandwidth
- Pipelined caches, multibanked caches, non-blocking caches

Reduce miss penalty
- Critical word first, merging write buffers

Reduce miss rate
- Compiler optimizations

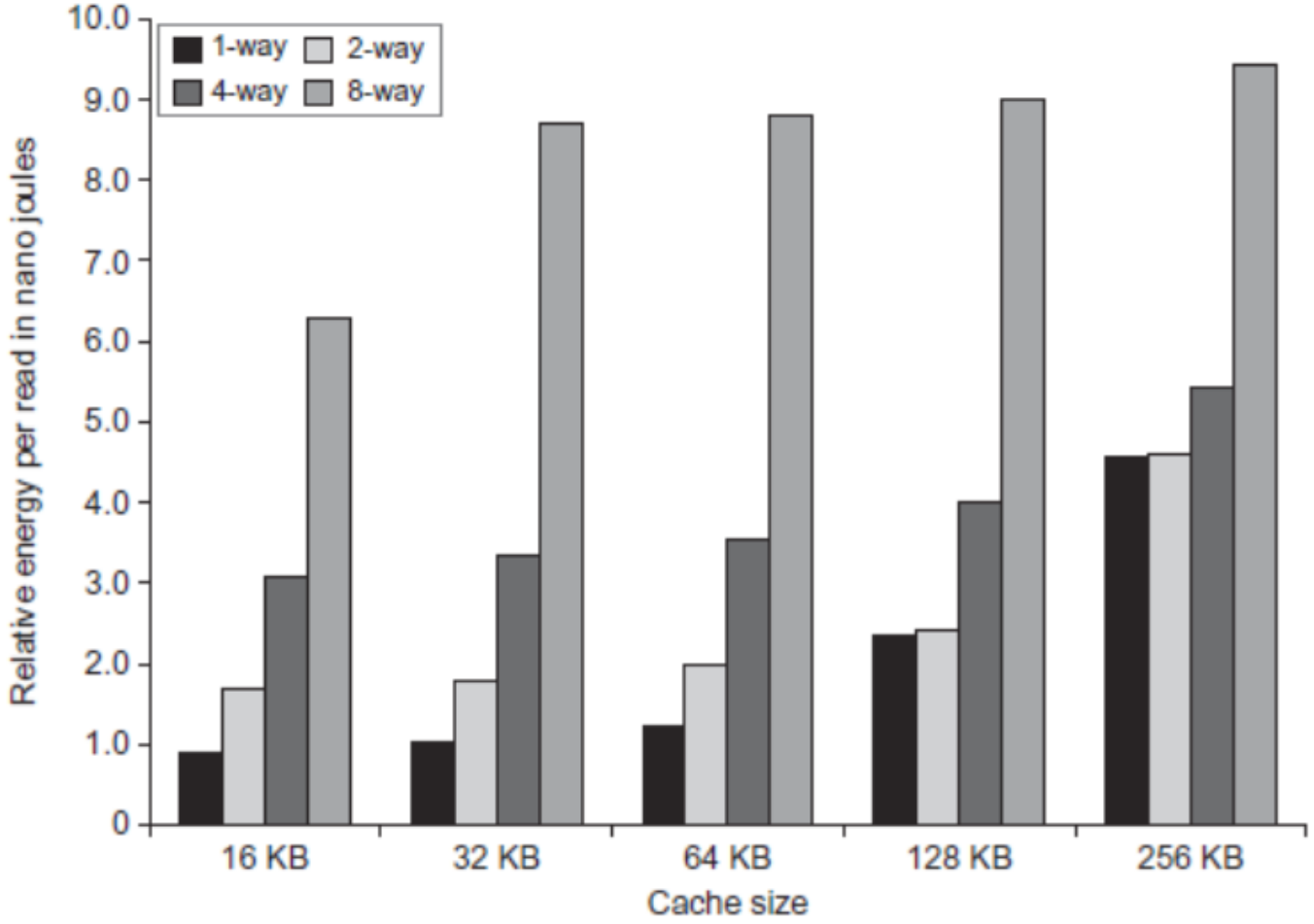
Reduce miss penalty or miss rate via parallelization
- Hardware or compiler prefetching

L1 Size and Associativity



Access time vs. size and associativity

L1 Size and Associativity



Energy per read vs. size and associativity

Way Prediction

To improve hit time, predict the way to pre-set mux

- Mis-prediction gives longer hit time
- 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
- Used on ARM Cortex-A8

Extend to predict block as well

- "Way selection"
- Increases mis-prediction penalty

Pipelined Caches

Pipeline cache access to improve bandwidth

- Examples:
  - Pentium: 1 cycle
  - Pentium Pro – Pentium III: 2 cycles
  - Pentium 4 – Core i7: 4 cycles

Increases branch mis-prediction penalty

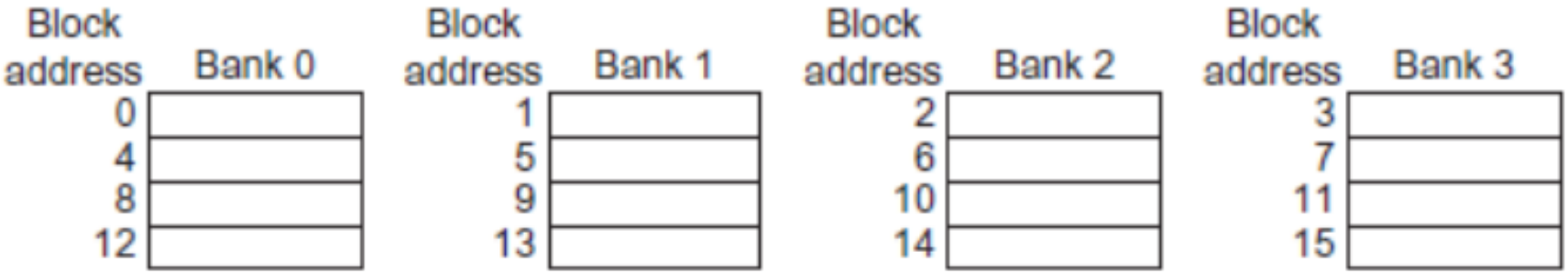
Makes it easier to increase associativity

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



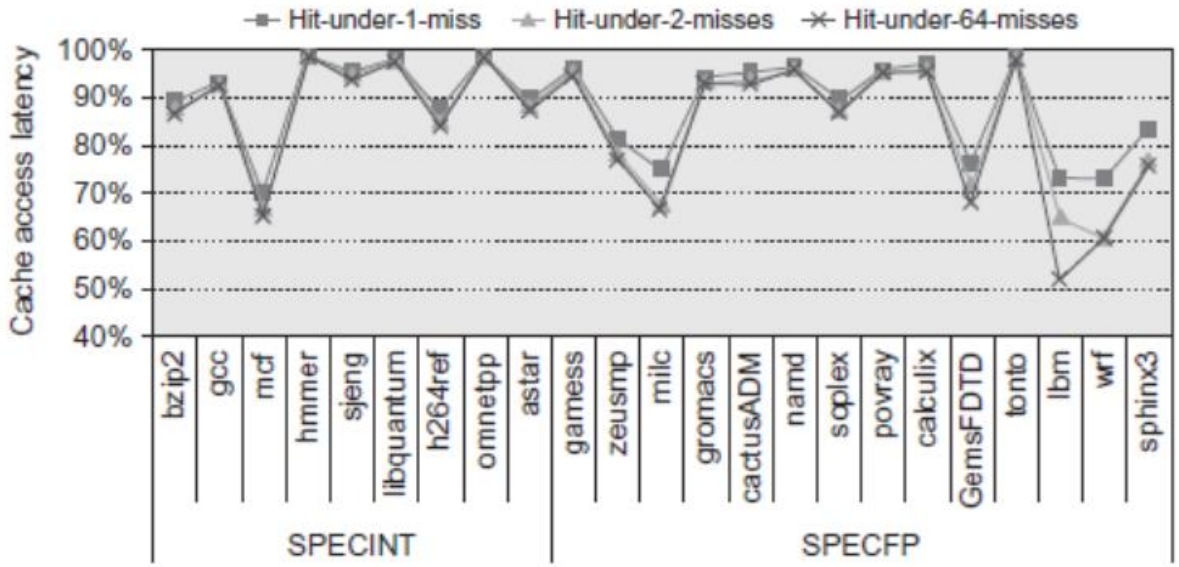
Nonblocking Caches

Allow hits before previous misses complete

- "Hit under miss"
- "Hit under multiple miss"

L2 must support this

In general, processors can hide L1 miss penalty but not L2 miss penalty



Critical Word First, Early Restart

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

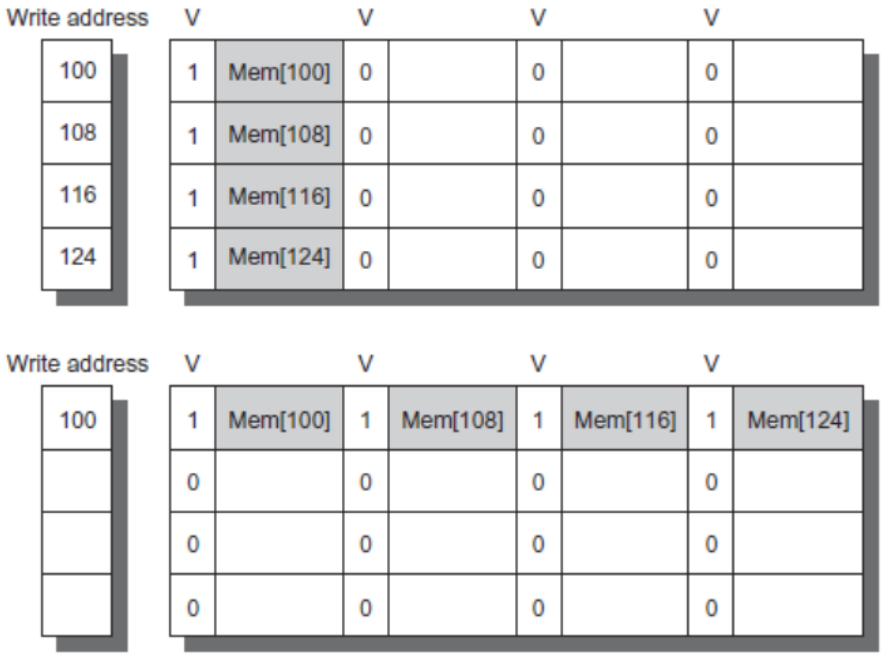
Effectiveness of these strategies depends on block size and likelihood of another access to the portion of the block that has not yet been fetched

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



No write buffering

Write buffering

Compiler Optimizations

Loop Interchange

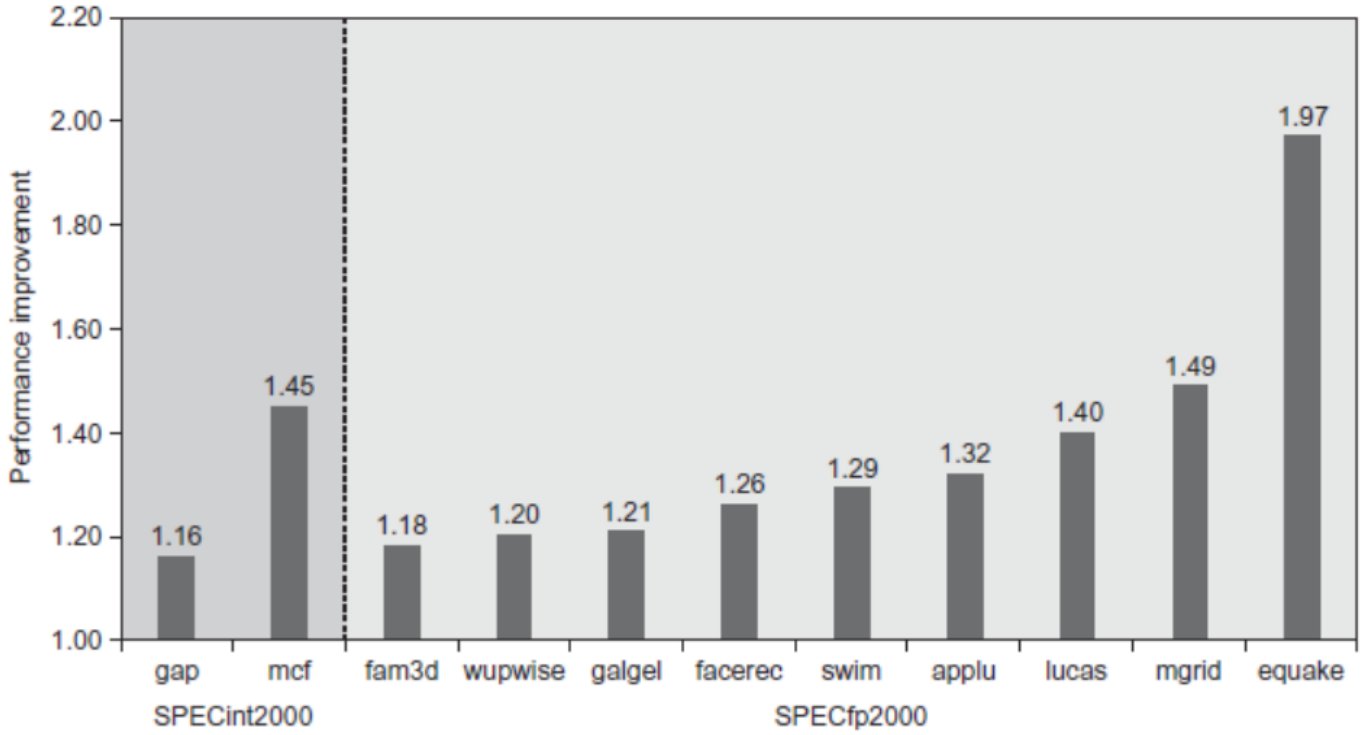
- Swap nested loops to access memory in sequential order

Blocking

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

Hardware Prefetching

Fetch two blocks on miss (include next sequential block)



Pentium 4 Pre-fetching

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

Use HBM to Extend Hierarchy

128 MiB to 1 GiB
Smaller blocks require substantial tag storage
Larger blocks are potentially inefficient

One approach (L-H):

- Each SDRAM row is a block index
- Each row contains set of tags and 29 data segments
- 29-set associative
- Hit requires a CAS

Use HBM to Extend Hierarchy

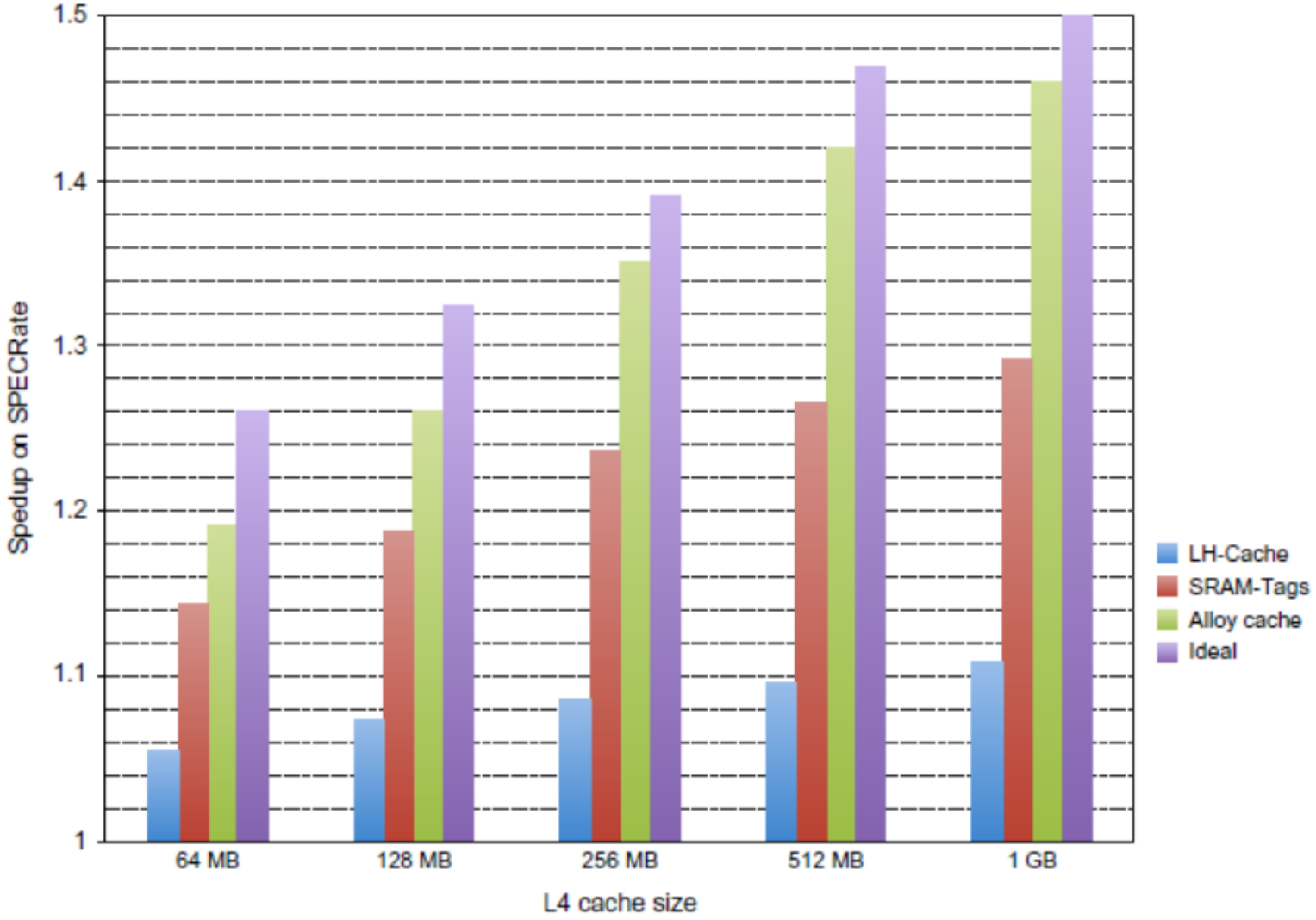
Another approach (Alloy cache):

- Mold tag and data together
- Use direct mapped

Both schemes require two DRAM accesses for misses

- Two solutions:
  - Use map to keep track of blocks
  - Predict likely misses

Use HBM to Extend Hierarchy



Summary

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 & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
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
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Virtual Memory and Virtual Machines

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
- Provide mechanisms for switching between user mode and supervisor mode
- Provide mechanisms to limit memory accesses
- Provide TLB to translate addresses

Virtual Memory: Topics

Why virtual memory?

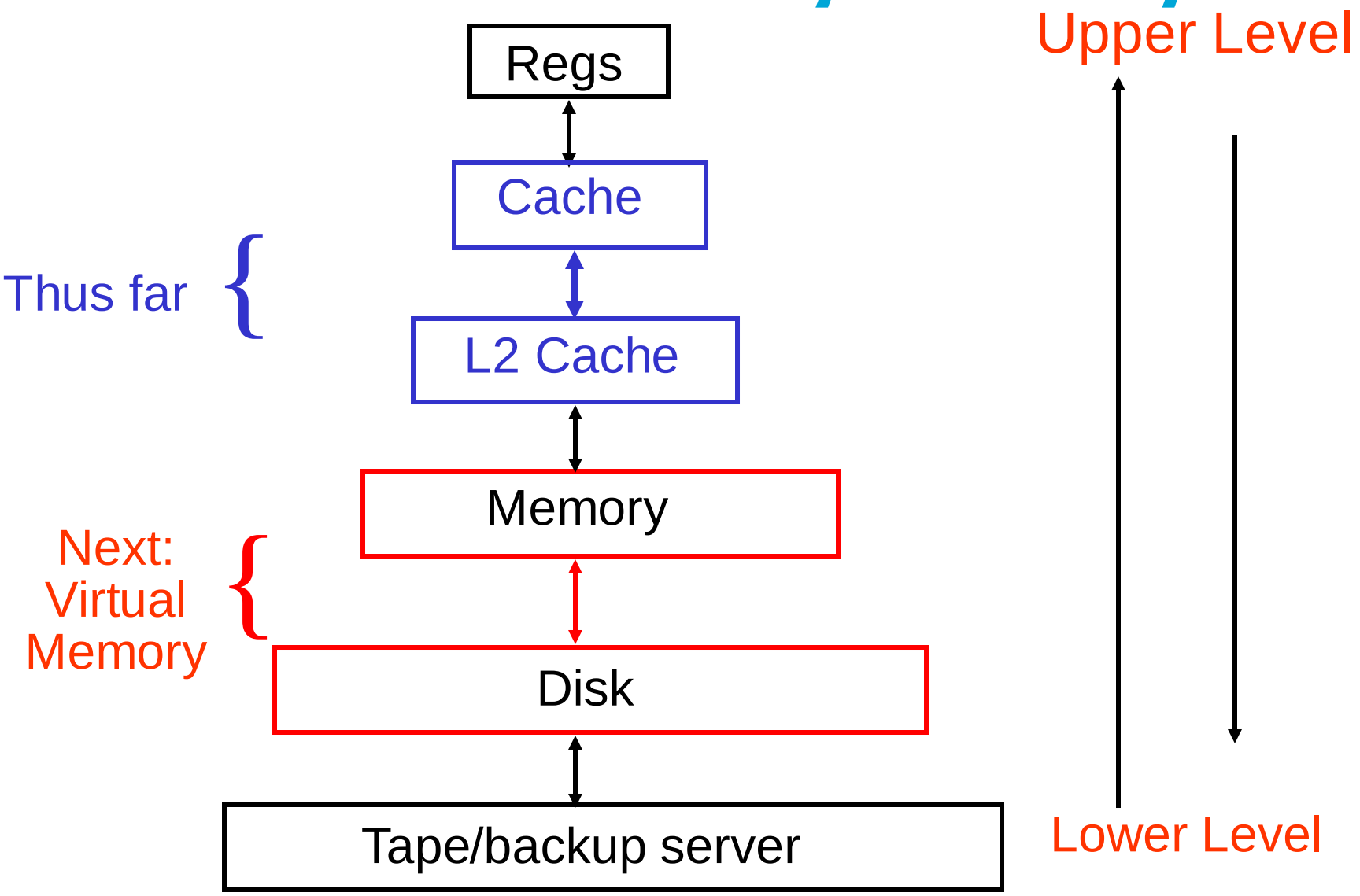
Virtual to physical address translation

Page Table

- Keeping things simple here!
- Check OS (books, articles) for more complex structures, e.g., multi-level hierarchical tables

Translation Lookaside Buffer (TLB)

Another View of Memory Hierarchy

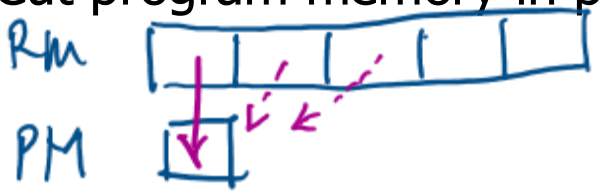


VM: From (recent) past to modern times

Past

Required memory (RM) > Physical memory (PM)

Cut program memory in pages and use memory as "cache"

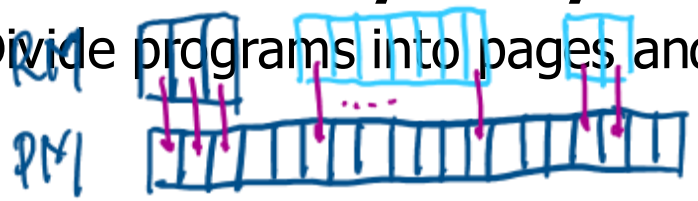


Thinking out loud: each "block" (or page) in the RM can be from same program or from different programS!!

Recent Past

Required memory < Physical memory

Divide programs into pages and map to physical memory



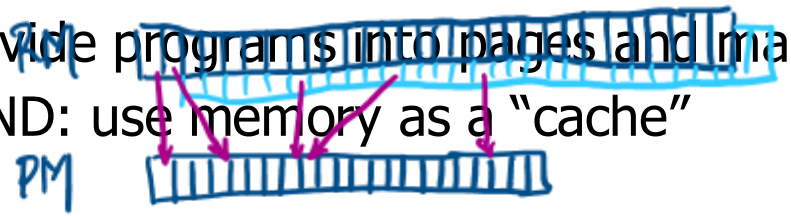
Thinking out loud: support for multiple programs —> start address of each program?
What is preferred? **Think more!!**

Now

Required (program) memory >> Physical memory

Divide programs into pages and map to physical memory

AND: use memory as a "cache"



Analogy:

- Page vs. (cache) block
- Page fault vs. (cache) miss

Why Virtual Memory?

Today computers run multiple processes, each with its own address space

Too expensive to dedicate a full-address-space worth of memory for each process

Principle of Locality

allows caches to offer speed of cache memory with size of DRAM memory

DRAM can act as a "cache" for secondary storage (disk) ⇒ Virtual Memory

Virtual memory – divides physical memory into blocks and allocates them to different processes

Virtual Memory Motivation

Historically virtual memory was invented when programs became too large for physical memory

Allows OS to share memory and protect programs from each other (main reason today)

Provides illusion of very large memory

- sum of the memory of many jobs greater than physical memory
- allows each job to exceed the size of physical memory

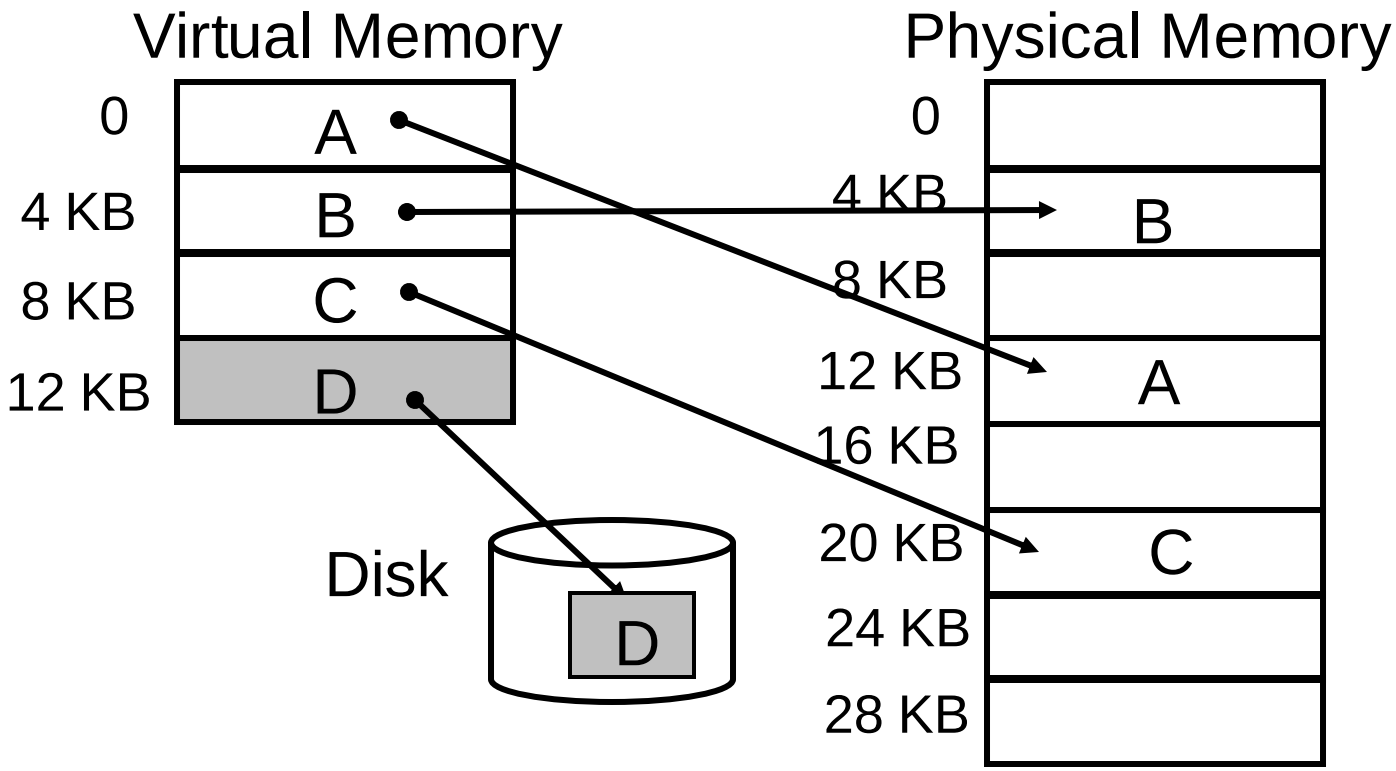
Allows available physical memory to be well utilized

Exploits memory hierarchy to keep average access time low

Mapping Virtual to Physical Memory

Program with 4 pages (A, B, C, D)

Any chunk of Virtual Memory assigned to any chuck of Physical Memory ("page")



Virtual Memory Terminology

Virtual Address
- address used by the programmer; CPU produces virtual addresses

Virtual Address Space
- set of all possible virtual addresses

Memory (Physical or Real) Address
- address of word in physical memory

Memory mapping or address translation
- process of virtual to physical address translation

More on terminology
- Page or Segment ⇔ Block
- Page Fault or Address Fault ⇔ Miss

In general & not discussed: when processes are involved, also mechanisms needed to track process id!

Comparing the 2 levels of hierarchy

Parameter	L1 Cache	Virtual Memory
Block/Page	16B – 128B	4KB – 64KB
Hit time	1 – 3 cc	50 – 150 cc
Miss Penalty (Access time) (Transfer time)	8 – 150 cc 6 – 130 cc 2 – 20 cc	1M – 10M cc (Page Fault) 800K – 8M cc 200K – 2M cc
Miss Rate	0.1 – 10%	0.00001 – 0.001%
Placement:	DM or N-way SA	Fully associative (OS allows pages to be placed anywhere in main memory)
Address Mapping	25-45 bit physical address to 14-20 bit cache address	32-64 bit virtual address to 25-45 bit physical address
Replacement:	LRU or Random (HW controlled)	LRU (SW controlled)
Write Policy	WB or WT	WB

Consider the hierarchy here!!

Page replacement (intermezzo)

Question: Who's task is this? User??

Answer:

- Operating system — same as user?
- THUS: user mode and supervisor mode (*more in servers*)

Question: How to keep track of LRU of the pages (of main memory)?

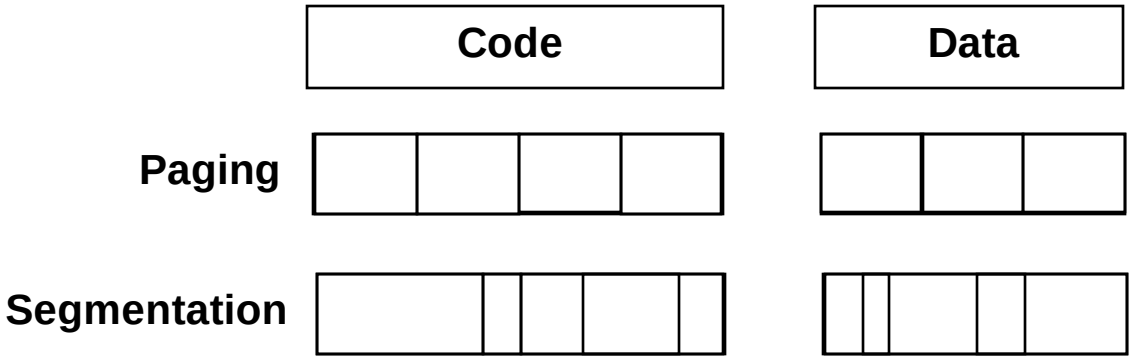
Answer:

- Maintain a "reference" bit for each page
- Regularly sample all "reference" bits and maintain table
- Based on table determine which page is LRU

Paging vs. Segmentation

Two classes of virtual memory

- Pages - fixed size blocks (4KB – 64KB)
- Segments - variable size blocks (1KB – 64KB/4GB)
- Hybrid approach: Paged segments – a segment is an integral number of pages (*see next slide*)

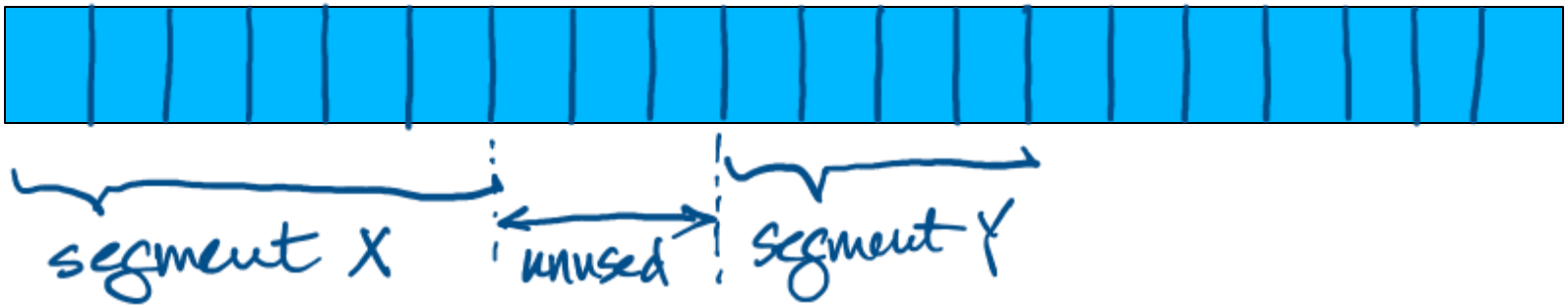


What is the main problem when only use paging?

- Maintain page table **per program** —> how large?
- See book —> typically several MBs (per program)

Paged Segments

Divide the memory into fixed pages
Each segment is contiguous series of pages



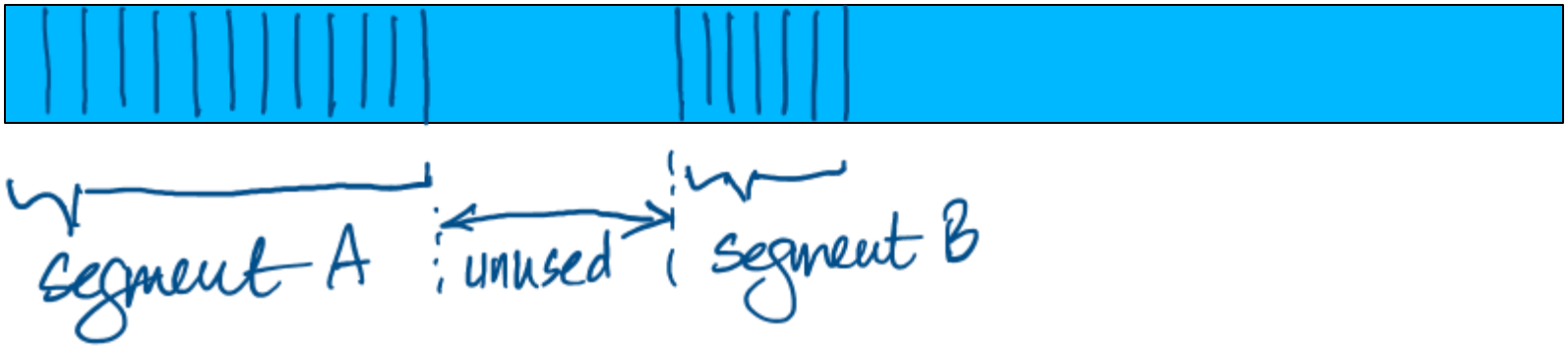
How to address this?

Segment # —> Page # —> Offset

Page tables are now much smaller, but need to maintain also segment table —> hierarchical tables (goes too far for this lecture)

Segmented Paging

Divide the memory into segments (any size)
Use pages within segments



How to address this?

Segment # —> Page # —> Offset

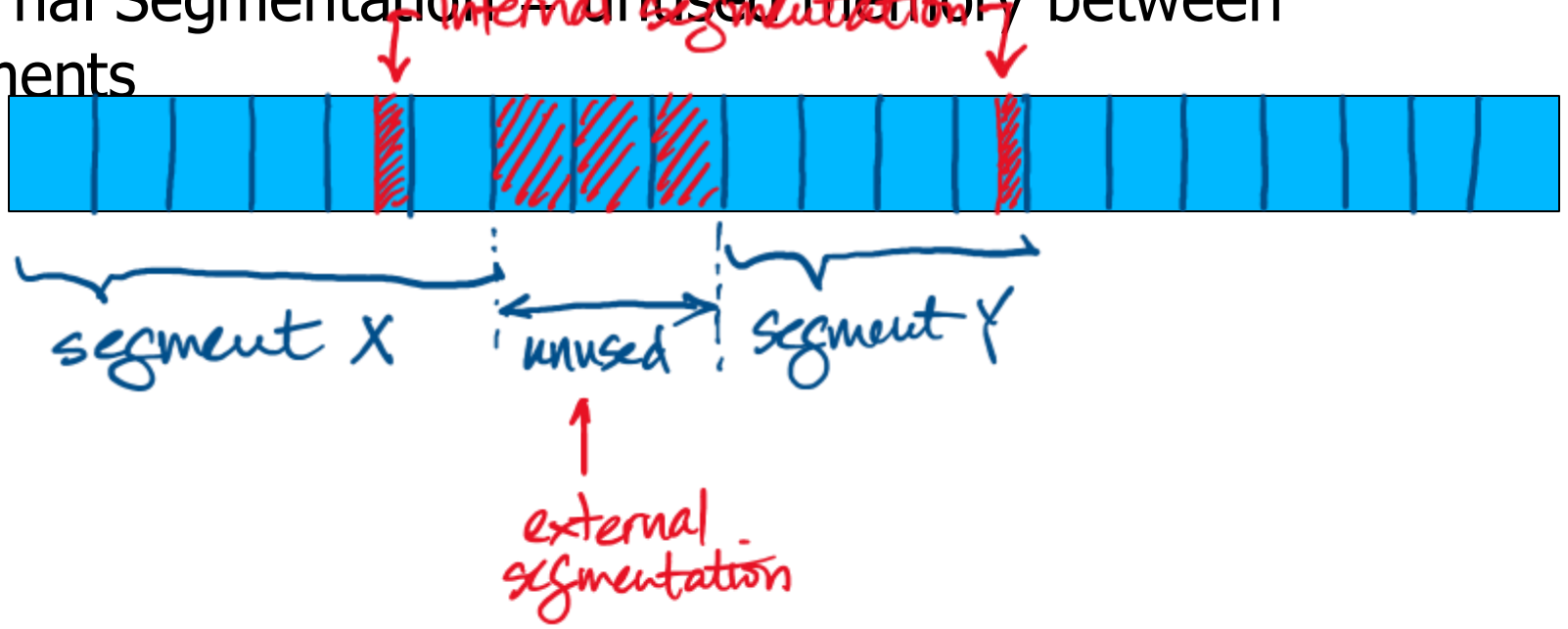
Page tables are now much smaller, but need to maintain also segment table —> hierarchical tables (goes too far for this lecture)

Internal vs. External Segmentation

Definitions!

Internal Segmentation = unused memory within pages

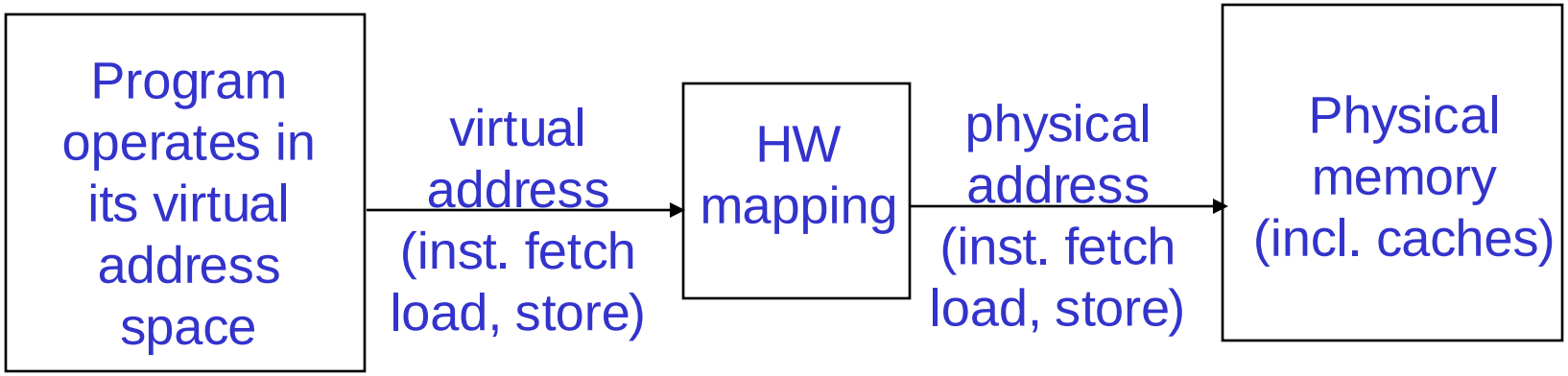
External Segmentation = unused memory between segments



Pros and Cons of Paging vs. Segmentation

	Page	Segment
Words per address	One	Two (segment + offset)
Programmer visible?	Invisible to AP	May be visible to AP
Replacing a block	Trivial (all blocks are same size)	Hard (must find contiguous, variable-size unused portion
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments transfer few bytes)

Virtual to Physical Address Translation



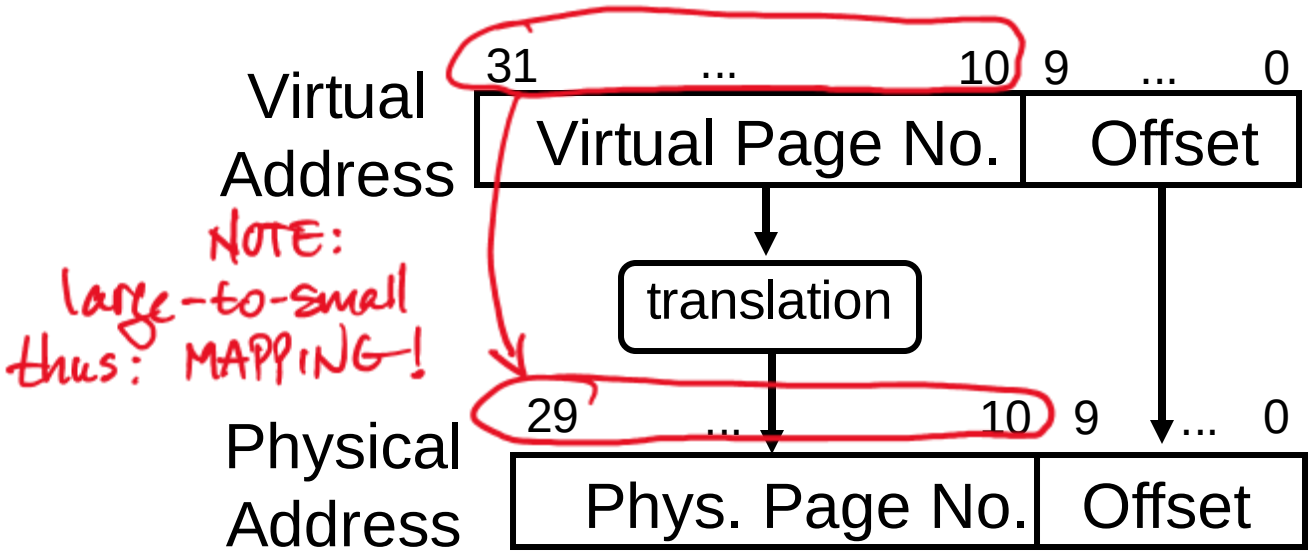
Each program operates in its own virtual address space

Each is protected from the other

OS can decide where each goes in (physical) memory

Combination of HW + SW provides virtual → physical mapping

Virtual Memory Mapping Function

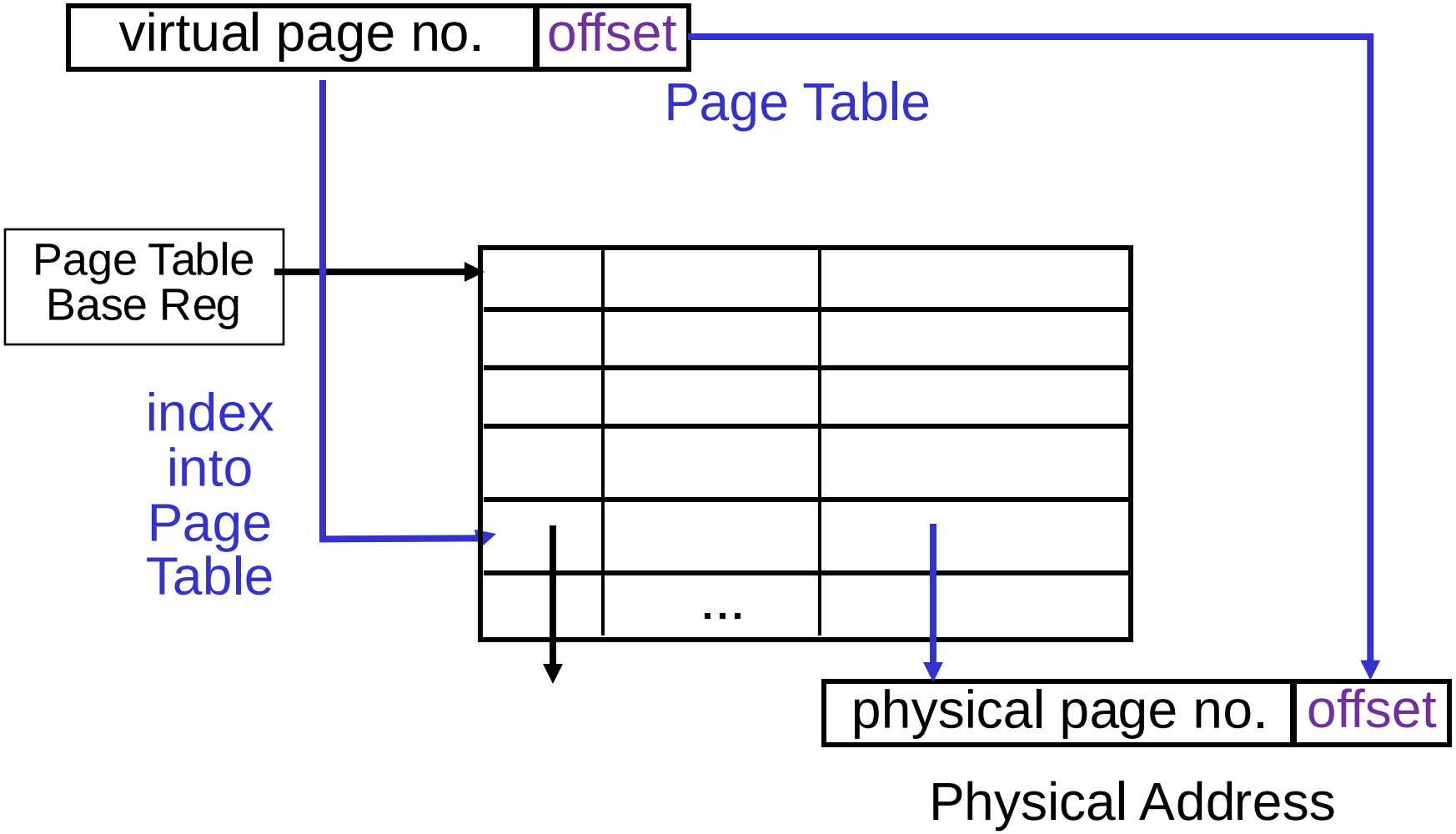


Use table lookup ("Page Table") for mappings: Virtual Page number is index

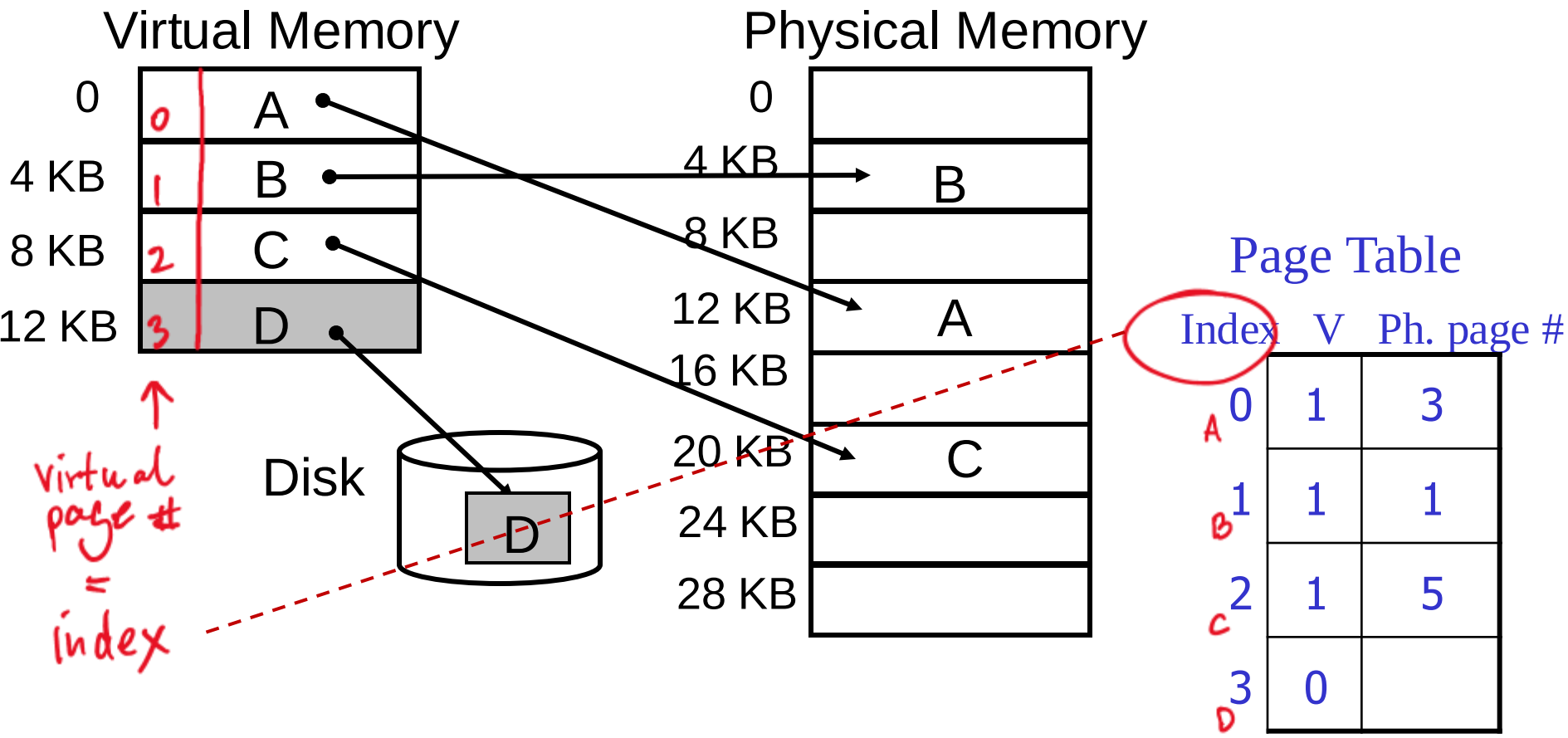
Virtual Memory Mapping Function

- Physical Offset = Virtual Offset
- Physical Page Number (P.P.N. or "Page frame") = PageTable[Virtual Page Number]

Address Mapping: Page Table



Example Page Table



Page Table

A page table is **an operating system structure** which contains the mapping of virtual addresses to physical locations

- There are several different ways, all up to the operating system, to keep these data around
- Remember: protection is needed —> user mode versus supervisor mode

Each process running in the operating system has its own page table

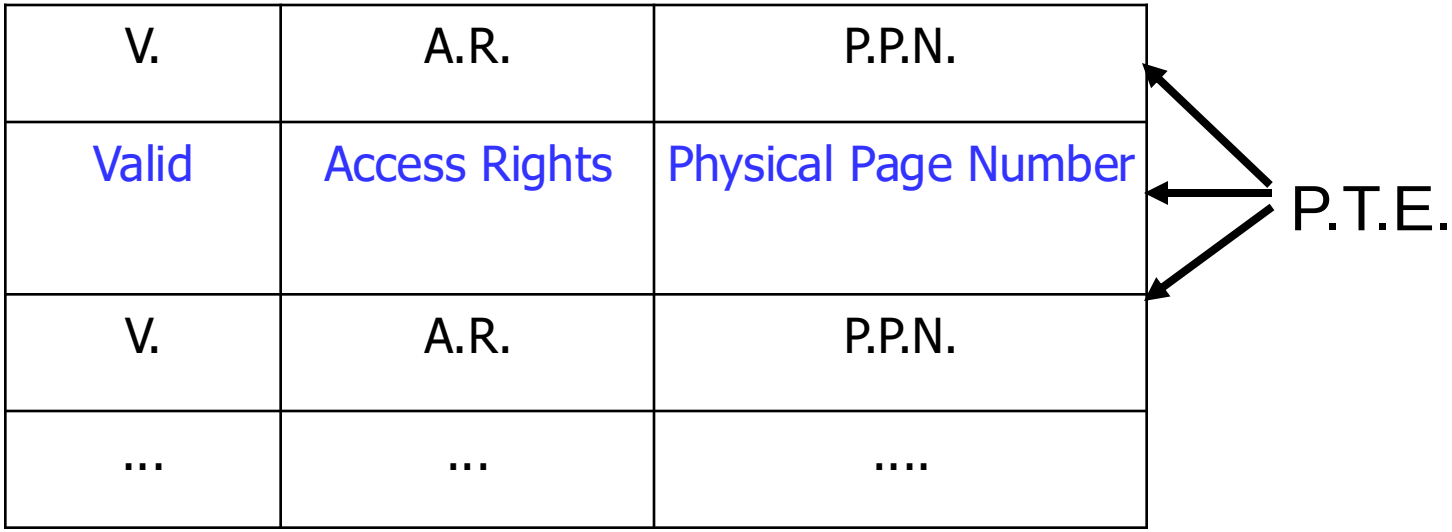
- "State" of process is PC, all registers, plus page table
- OS changes page tables (on a context switch) by changing contents of Page Table Base Register
- Remember: the cost of context switch to supervisor mode in order to be able to change the page table

Page Table Entry (PTE) Format

Valid bit indicates if page is in memory

Mapped to disk if Not Valid (V = 0)

Contains mappings for every possible virtual page



If valid, also check if have permission to use page:
Access Rights (A.R.) may be Read Only, Read/Write, Executable

Virtual Memory Problem #1

Not enough physical memory

- Only, say, 1GB of physical memory
- N processes, each 4GB of virtual memory!

Spatial Locality to the rescue

- Each page is 4 KB, lots of nearby references
- No matter how big the program is, at any time it is only accessing a few pages
- "Working Set": pages needed at a certain time (often corresponds to recently used pages)
- Recall:
  - spatial locality —> (use main memory as) "caches"

VM Problem #2: Fast Address Translation

PTs are stored in main memory ⇒ every memory access logically takes at least twice as long, one access to obtain physical address and second access to get the data.

Observation: locality in pages of data, must be locality in virtual addresses of those pages, i.e., also translations

⇒ Remember the last translation(s)

Address translation (a part of PT) is kept in a special cache called Translation Look-Aside Buffer or TLB

Question: Where to keep the TLB?

⇒ TLB must be on chip; its access time is comparable to cache

Typical TLB Entry Format

Tag	Ph. page #	Dirty	Ref	Valid	Access rights

Tag: Portion of virtual address (*recall: tags in caches*)

Physical Page number

Dirty: since **write back** technique is used, need to know whether or not to write page to disk when replaced

Reference: Used to help calculate LRU on replacement (*see future slide*)

Valid: Entry is valid

Access rights: R (read permission), W (write perm.)

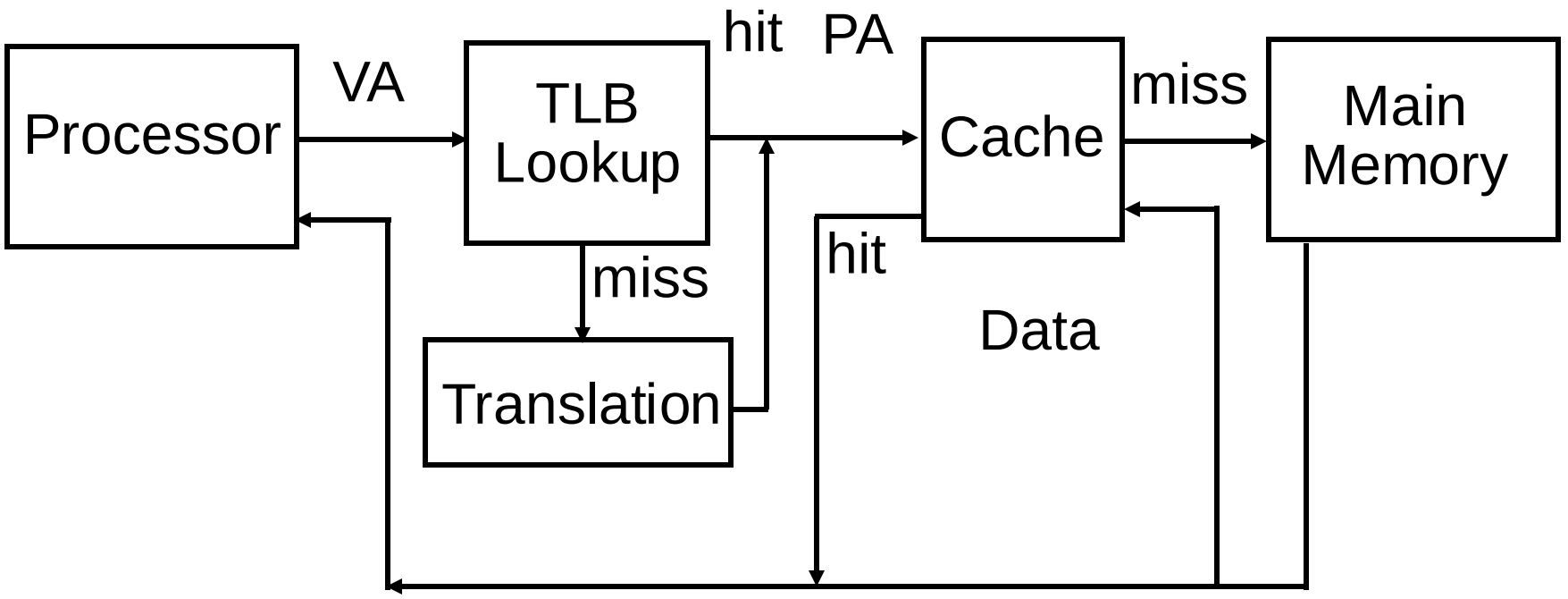
Intermezzo question:

- Should we apply write-through (WT) technique (concept from caches) to pages?
- Answer: (always) NO!

Translation Look-Aside Buffers

TLBs usually small, typically 128 - 256 entries

Like any other cache, the TLB can be fully associative, set associative, or direct mapped



TLB Translation Steps

Assume 32 entries, fully-associative TLB

(Alpha AXP 21064)

1. Processor sends the virtual address to all tags
2. If there is a hit (there is an entry in TLB with that Virtual Page number and valid bit is 1) and there is no access violation, then:
3. Matching tag sends the corresponding Physical Page Number
4. Combine Physical Page Number and Page Offset to get full physical address

What if not in TLB?

Option 1: Hardware checks page table and loads new Page Table Entry into TLB

Option 2: Hardware traps to OS, up to OS to decide what to do:

- When in the operating system, we don't do translation (turn off virtual memory)
- The operating system knows which program caused the TLB fault, page fault, and knows the virtual address desired
- So it looks the data up in the page table
- If the data are in memory, simply add the entry to the TLB, evicting an old entry from the TLB
- *If the data is not in memory, but on disk —> next slide!*

What if the data is on disk?

The page is loaded from the disk into a free block of memory, using a DMA transfer

- Meantime the OS switches to some other process waiting to be run

When the DMA is complete, the OS gets an interrupt and update the process' page table

- So when the OS can switch back to the process, desired data will be in memory

What if we don't have enough memory?

We chose some other page belonging to a program and transfer it onto the disk if it is dirty

- If clean (other copy is up-to-date), just overwrite that data in memory
- We chose the page to evict based on replacement policy (e.g., LRU)
- In Linux, there's an OS process that continuously checks if there is a dirty page in memory and writes it back if memory bandwidth available

And update that program's page table to reflect the fact that its page moved somewhere else

OR: replacement algorithm (*see next slide*)

Page Replacement Algorithms

First-In/First Out

- in response to page fault, replace the page that has been in memory for the longest period of time
- does not make good use of the principle of locality: an old but frequently used page could be replaced
- easy to implement (OS maintains history thread through page table entries)
- usually exhibits the worst behavior

Least Recently Used

- selects the least recently used page for replacement
- requires knowledge of past references
- more difficult to implement, good performance

Page Replacement Algorithms

Not Recently Used (an approximation of LRU)

- A reference bit flag is associated to each page table entry such that
  - Ref flag = 1 - if page has been referenced in recent past
  - Ref flag = 0 - otherwise
- If replacement is necessary, choose any page frame such that its reference bit is 0
- OS periodically clears the reference bits
- Reference bit is set whenever a page is accessed

Check again the intermezzo slide about page replacement

Selecting a Page Size

Balance forces in favor of larger pages versus those in favoring smaller pages

Larger page

- Reduce size PT (save space)
- Allow larger caches with fast hits (low associativity)
- More efficient transfer from the disk or possibly over the networks
- Less TLB entries or less TLB misses

Smaller page

- better conserve space, less wasted storage (*Internal Fragmentation*)
- shorten startup time, especially with plenty of small processes

Typical size 4KB – 64KB

VM Problem #3: Page Table too big

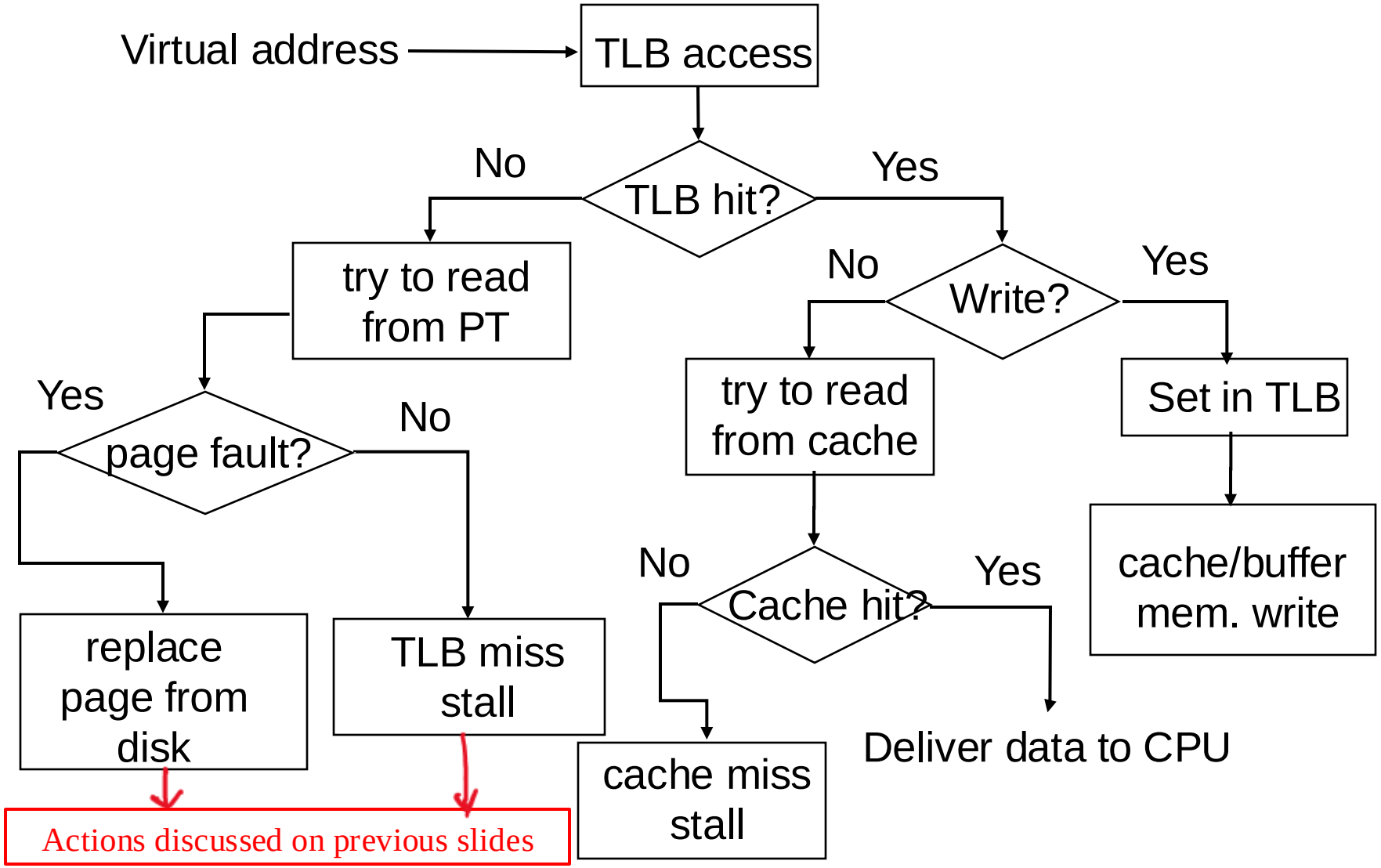
Example

32-bits virtual address => 4GB Virtual Memory ÷ 4 KB page
=> ~ 1 million Page Table Entries
=> 4 MB just for Page Table for 1 process,
25 processes => 100 MB for Page Tables!

Problem gets worse on modern 64-bits machines

Solution: Hierarchical Page Table

The Big Picture



Virtual Memory in Embedded Domain

Nothing fundamentally prevents virtual memory in embedded processors

- but need backing storage devices (flash, hard disk, …)
- can make real-time/quality-of-service goals harder to achieve (but can lock down pages just like blocks in cache)
- increases cost, power, complexity of system

Things to Remember

Apply Principle of Locality Recursively

Manage memory to disk? Treat as cache

- Included protection as bonus, now critical
- Use Page Table of mappings vs. tag/data in cache

Spatial locality means Working Set of pages is all that must be in memory for process to run

Virtual memory to Physical Memory Translation too slow?

- Add a cache of Virtual to Physical Address Translations, called a TLB

Need more compact representation to reduce memory size cost of simple 1-level page table (especially 32 ⇒ 64-bit address)

Main Memory Background

Next level down in the hierarchy after L2/L3 cache

satisfies the demands of caches

serves as the I/O interface

Performance of Main Memory:

Latency: Cache Miss Penalty

Access Time: time between when a read is requested and when the desired word arrives

Cycle Time: minimum time between requests to memory

Bandwidth (the number of bytes read or written per unit time):

impacts: multiprocessors, I/O, Miss Penalty for Large $ Block (L2)

Main Memory Background

Main Memory is DRAM: Dynamic Random Access Memory

- Dynamic since needs to be refreshed periodically (8 ms, <5% time)
- Addresses divided into 2 halves (Memory as a 2D matrix):
  - RAS or Row Access Strobe + CAS or Column Access Strobe

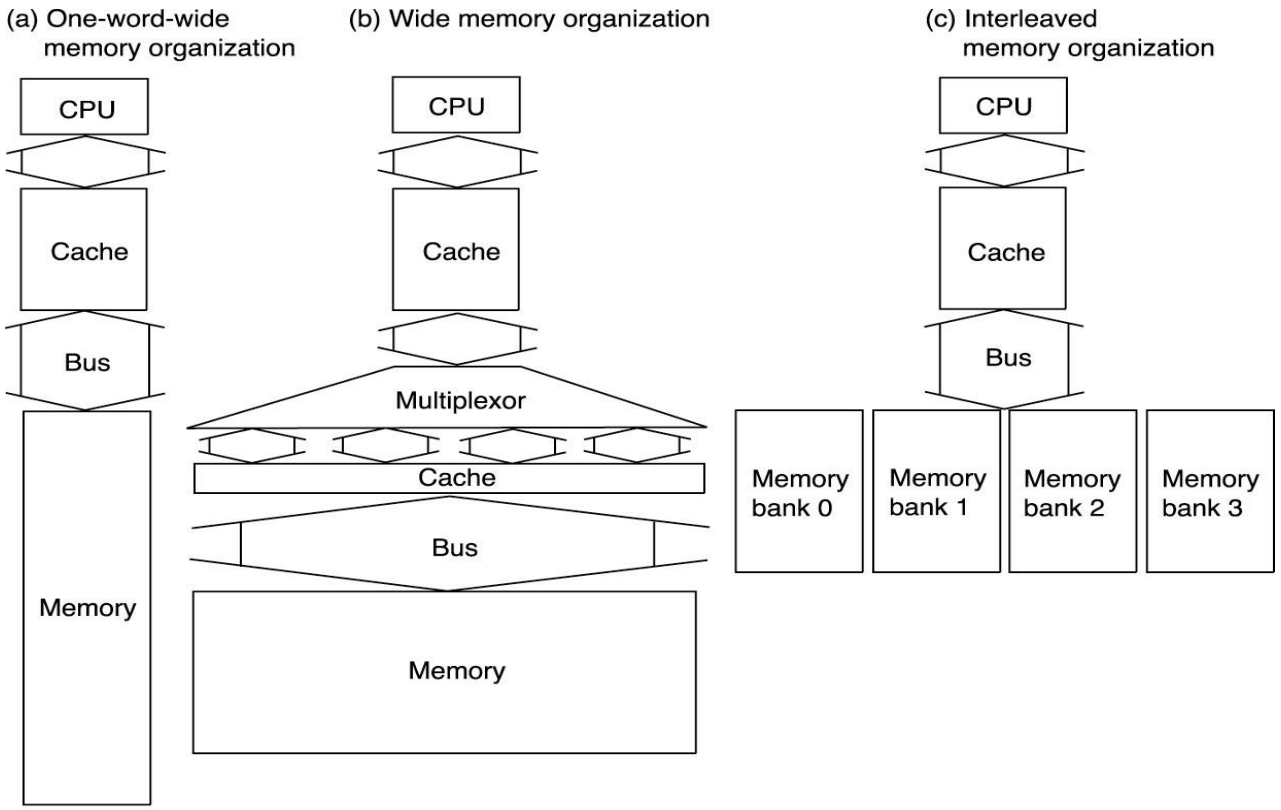
Cache uses SRAM: Static Random Access Memory

- No refresh (6 transistors/bit vs. 1 transistor/bit)

Techniques for Improving Main Memory Performance

1. Wider Main Memory
2. Simple Interleaved Memory
3. Independent Memory Banks

Memory Organizations



Simple: CPU, Cache, Bus, Memory same width (32 or 64 bits)

Wide: CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UtraSPARC 512)

Interleaved: CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is word interleaved

1st Technique for Higher Bandwidth: Wider Main Memory

Timing model (word size is 8 bytes = 64 bits)

- 4cc to send address, 56cc for access time per word, 4cc to send data
- Cache Block is 4 words

Simple M.O.: 4 x (4+56+4) = 256cc

Wide M.O.(2W): 2 x (4+56+4) = 128 cc

Wide M.O.(4W): 4+56+4 = 64 cc

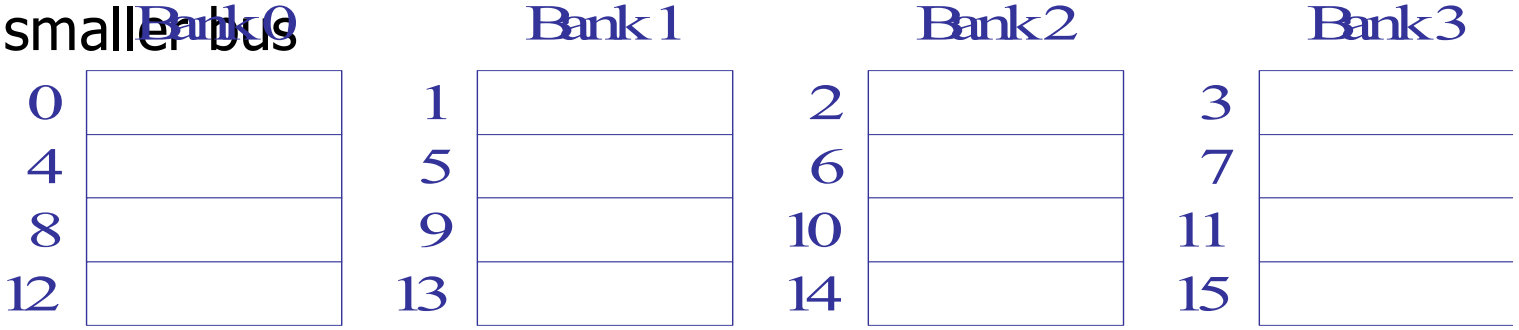
2nd Technique for Higher Bandwidth: Simple Interleaved Memory

Take advantage of potential parallelism of having many chips in a memory system

- Memory chips are organized in independent banks allowing multi-word read or writes at a time

Interleaved M.O.: 4 + 56 + 4x4 = 66 cc

- 4x4 – sequentially sending word from 4 banks through smaller bus



Summary

To provide illusion of large memory, even with many processes, use virtual memory

Page table contains mapping of virtual page numbers to physical page numbers (and more)

To make address translation fast, TLB contains recent translations from virtual to physical page numbers

To improve main memory bandwidth:

- use wider main memory
- use simple interleaved memory
- use independent memory banks

End – Virtual Memory

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"

Include??

Requirements of VMM

Guest software should:

- Behave on as if running on native hardware
- Not be able to change allocation of real system resources

VMM should be able to "context switch" guests

Hardware must allow

- System and use processor modes
- Privileged subset of instructions for allocating system resources

Include??

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

Include??

Extending the ISA for Virtualization

Objectives:

- Avoid flushing TLB
- Use nested page tables instead of shadow page tables
- Allow devices to use DMA to move data
- Allow guest OS's to handle device interrupts
- For security: allow programs to manage encrypted portions of code and data

Include??

Placeholder for VMM slides from 2023

Do not forget about story about TLBs!!

Page tables and paging

Include?? & Update

Fallacies and Pitfalls

Predicting cache performance of one program from another

Simulating enough instructions to get accurate performance measures of the memory hierarchy

Not deliverying high memory bandwidth in a cache-based system

END of NEW CH2