

*Course on: "Advanced Computer Architectures"*

# Memory Hierarchy: Advanced Concepts



Prof. Cristina Silvano  
Dipartimento di Elettronica, Informazione e Bioingegneria  
Politecnico di Milano

**email: [cristina.silvano@polimi.it](mailto:cristina.silvano@polimi.it)**

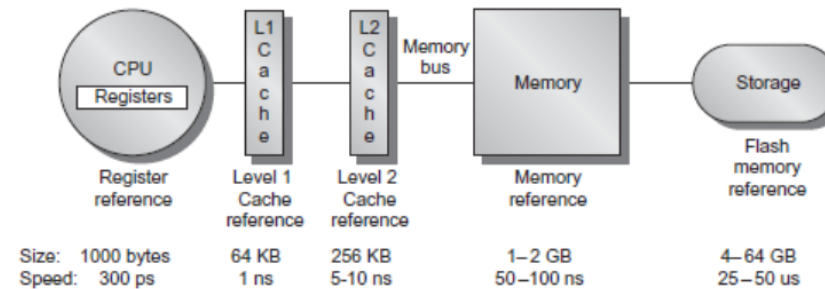


# Introduction

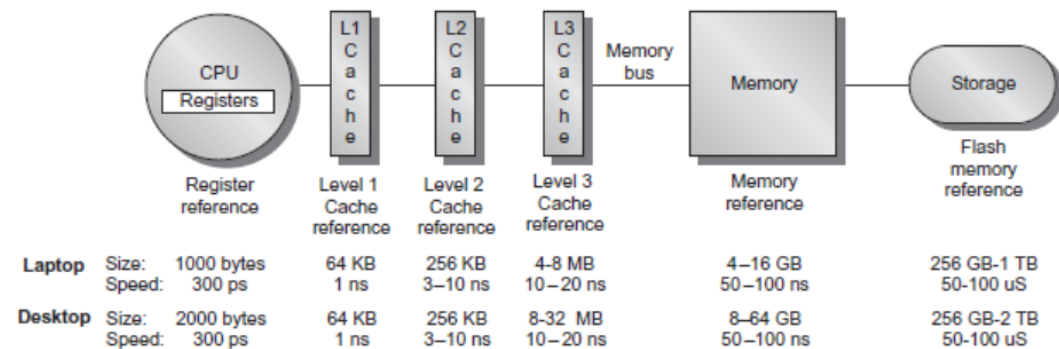
---

- Programmers want unlimited amounts of memory with low latency
- Fast memory technology (SRAM) is more expensive per bit than slower memory (DRAM).
- **Solution: organize memory system into a hierarchy**
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a copy of a subset of the memory below it
- **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

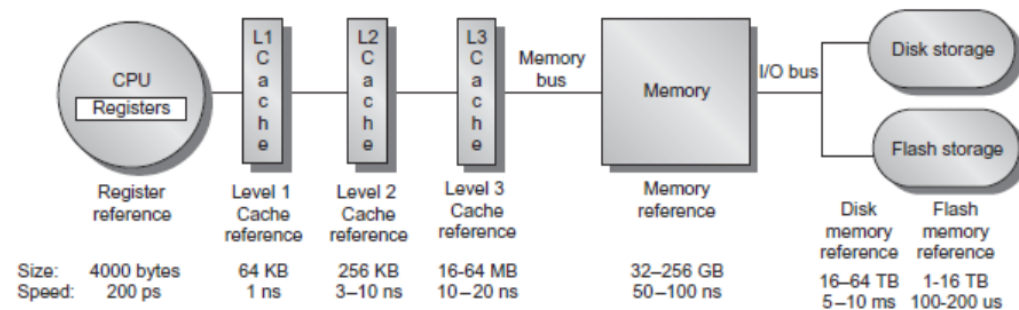
# Levels of Memory Hierarchy



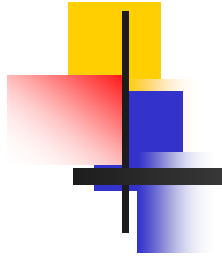
(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server



# **Classifying Cache Misses: 3 Cs**



# Classifying Cache Misses: 3 Cs

---

- **Three** major categories of cache misses:
  1. **Compulsory Misses:** cold start misses or first reference misses
  2. **Capacity Misses:** can be reduced by increasing cache size
  3. **Conflict Misses:** can be reduced by increasing cache size and/or associativity



# Classifying Cache Misses: 3 Cs

---

## 1. **Compulsory Misses:**

- ❑ The first access to a block is not in the cache, so the block must be loaded in the cache from the MM.
- ❑ Also called ***cold start misses*** or ***first reference misses***.
- ❑ *There are compulsory misses even in an infinite cache: they are independent of the cache size*



# Classifying Cache Misses: 3 Cs

---

## 2. Capacity Misses:

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



# Classifying Cache Misses: 3 Cs

---

## 3. Conflict Misses:

- ❑ If the block-placement strategy is set associative or direct mapped, conflict misses will occur because a block can be replaced and later retrieved when other blocks map to the same location in the cache.
  - ❑ Also called ***collision misses*** or ***interference misses***.
  - ❑ *Conflict misses decrease as associativity increases.*
  - ❑ *By definition, fully associative caches avoid all conflict misses, but they are consuming area.*
- 
- ❑ *More recent, 4th "C": **Coherence Misses** caused by the cache coherence problem in multi-processor architectures*





# Improving Cache Performance

---

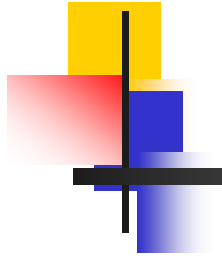
- Average Memory Access Time:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

- **How to improve cache performance:**

1. Reduce the miss rate
2. Reduce the miss penalty
3. Reduce the hit time

***Overall goal: Balancing fast hits and few misses***



# **1) How to reduce the miss rate**



## 0. Reducing Misses via Larger Cache Size

---

- *Obvious way to reduce capacity misses: to increase the cache capacity*
- Drawback: Increases hit time, area, power consumption and cost

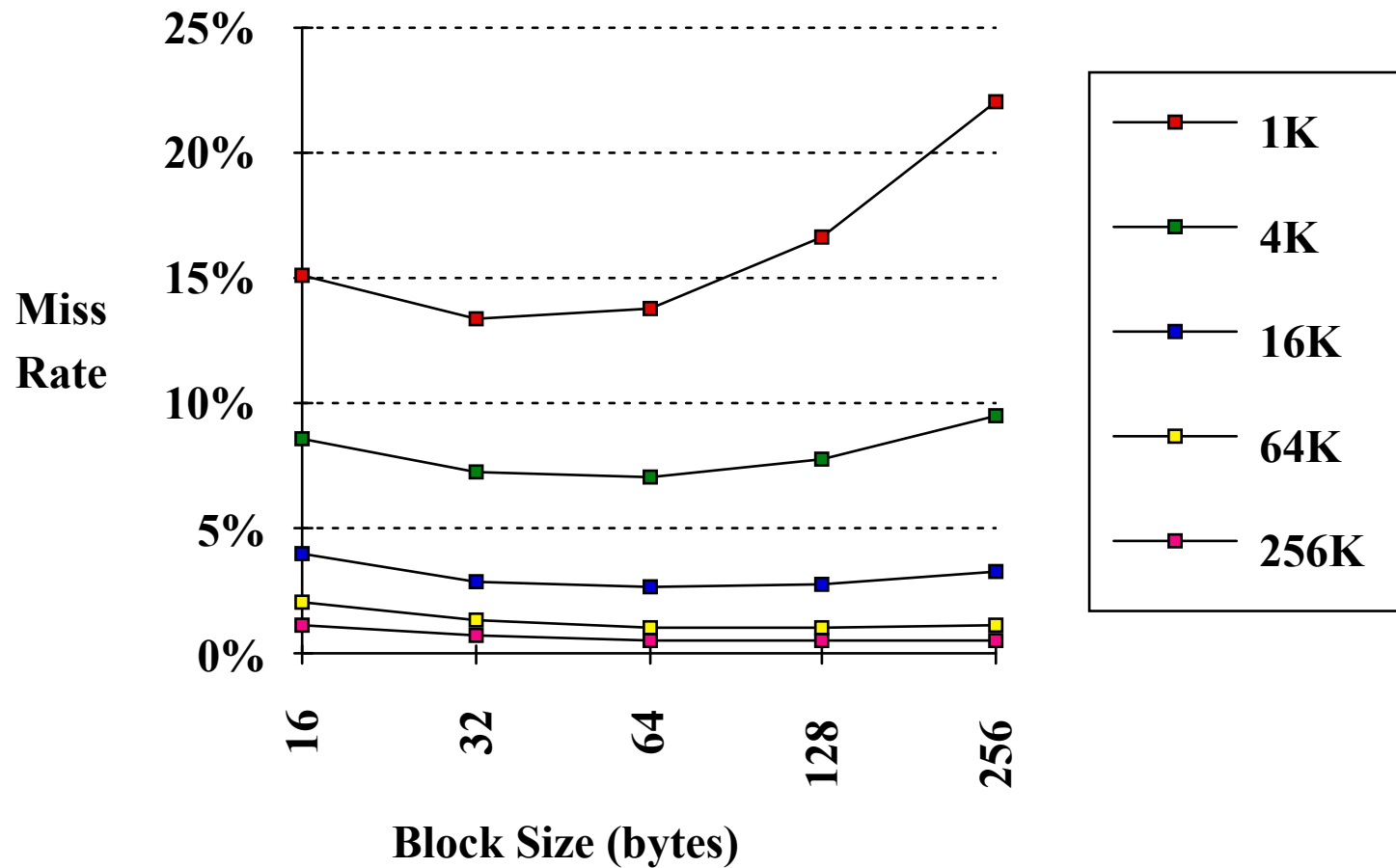


# 1. Reducing Misses via Larger Block Size

---

- Increasing the block size reduces the miss rate up to a point when the block size is too large with respect to the cache size
- Larger block size will reduce compulsory misses taking advantage of spatial locality
- **Main drawbacks:**
  - Larger blocks increase miss penalty
  - Larger blocks reduce the number of blocks so increase conflict misses (and even capacity misses) if the cache is small.

# 1. Reducing Misses via Larger Block Size





## 2. Reducing Misses via Higher Associativity

---

- Higher associativity decreases the conflict misses

- **Main drawbacks:**

- The complexity increases hit time, area, power consumption and cost.

- The 2:1 Cache Rule:


Miss Rate Cache Size **N**  $\cong$  Miss Rate **2-way** Cache Size **N/2**

# Multibanked Caches

- Multibanked caches introduce a sort of associativity
- Organize cache as **independent banks** to support simultaneous access to increase the cache bandwidth
  - 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 to access each bank in turn (sequential interleaving):

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

**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.



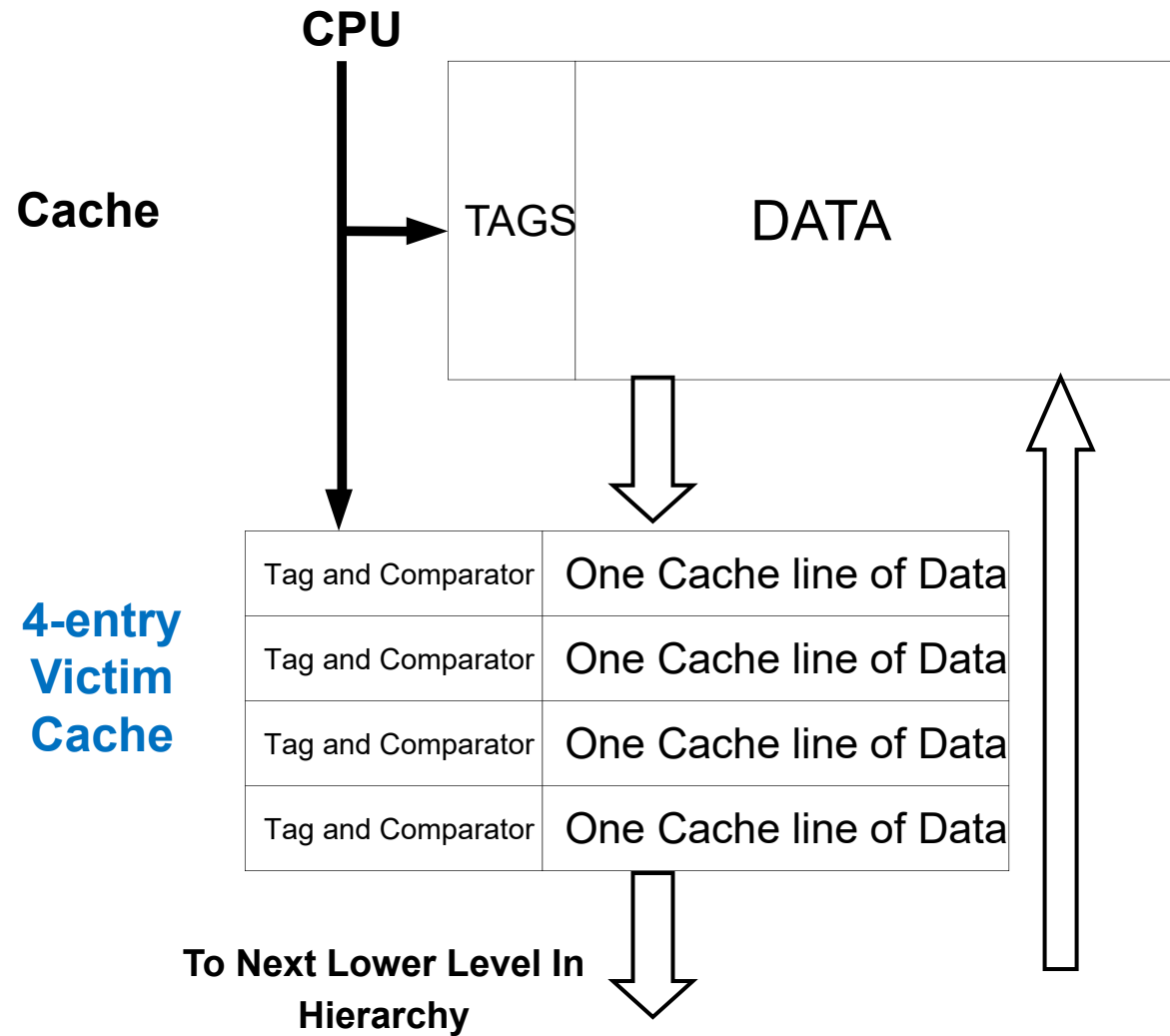
### 3. Reduce Misses (and Miss Penalty) via a “Victim Cache”

---

- **Victim cache** is a small fully associative cache used as a buffer to place data discarded from cache to better exploit temporal locality
- **Victim cache** placed between cache and its refilling path towards the next lower-level in the hierarchy
- **Victim cache** is checked on a miss to see if it has the required data before going to lower-level memory
- If the block is found in **Victim cache** the victim block and the cache block are swapped

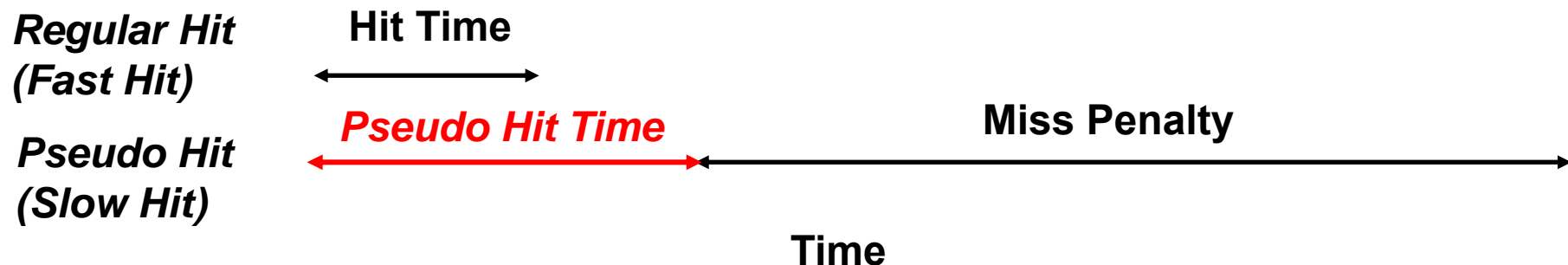


### 3. Reducing Misses (and Miss Penalty) via a "Victim Cache"



## 4. Reducing Misses via Pseudo-Associativity and Way Prediction

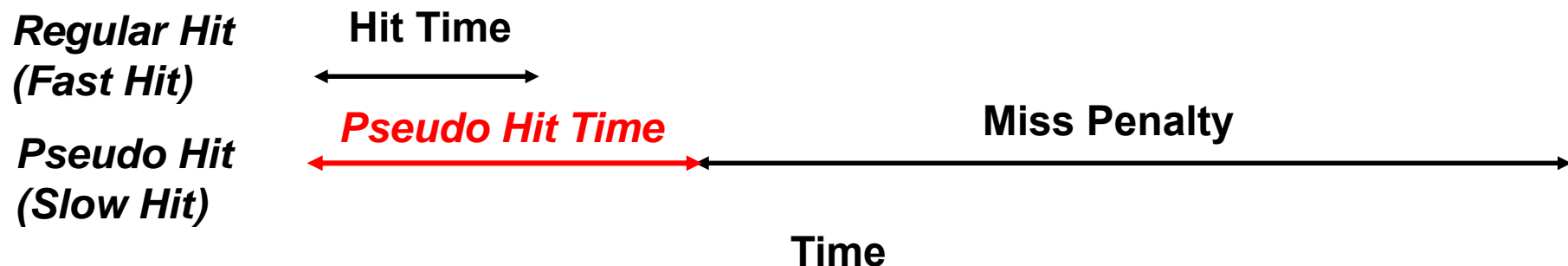
- **Way prediction** in 2-way set associative caches:  
Use extra bits to predict for each set which of the two ways to try on the next cache access (predict the way to pre-set the mux):
  - If the way prediction is correct  $\Rightarrow$  Hit time
  - If way misprediction  $\Rightarrow$  **Pseudo hit time** in the other way and change the way predictor
  - Otherwise go to the lower level of hierarchy (Miss penalty)





## 4. Reducing Misses via Pseudo-Associativity and Way Prediction

- **Pseudo-associativity** in direct mapped caches: Divide the cache in two banks in a sort of associativity.
  - If the bank prediction is correct  $\Rightarrow$  Hit time
  - If misprediction on the first bank, check the other bank to see if there, if so have a ***pseudo hit (slow hit) and change the bank predictor***
  - Otherwise go to the lower level of hierarchy (Miss penalty)





## 5. Reducing Misses by Hardware Pre-fetching of Instructions & Data

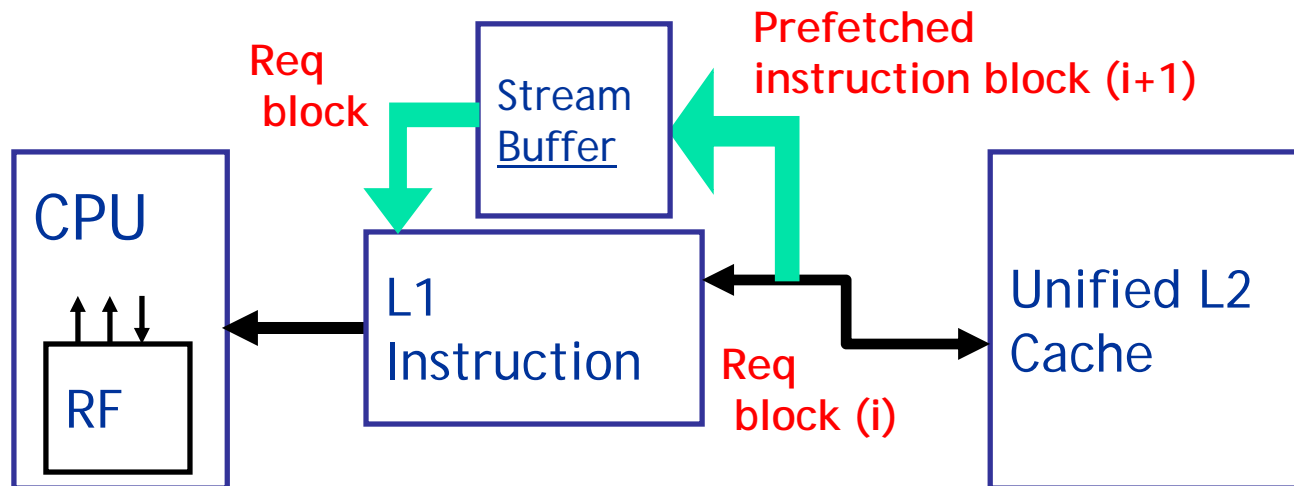
---

- **Basic idea:** To exploit locality, pre-fetch next instructions (or data) before they are requested by the processor
  - Pre-fetching can be done in cache or in an *external stream buffer*
- **Instruction Pre-fetching:**
  - Alpha AXP 21064 fetches 2 blocks on a miss: requested block fetched in cache, while the next block is placed in **"I-stream buffer"**
  - On miss, stream buffer to be checked

## 5. Reducing Misses by Hardware Pre-fetching of Instructions & Data

### Instruction pre-fetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, while next block in **instruction stream buffer**
- If miss in cache, but hit in stream buffer, move stream buffer block into cache and pre-fetch next block (i+2)





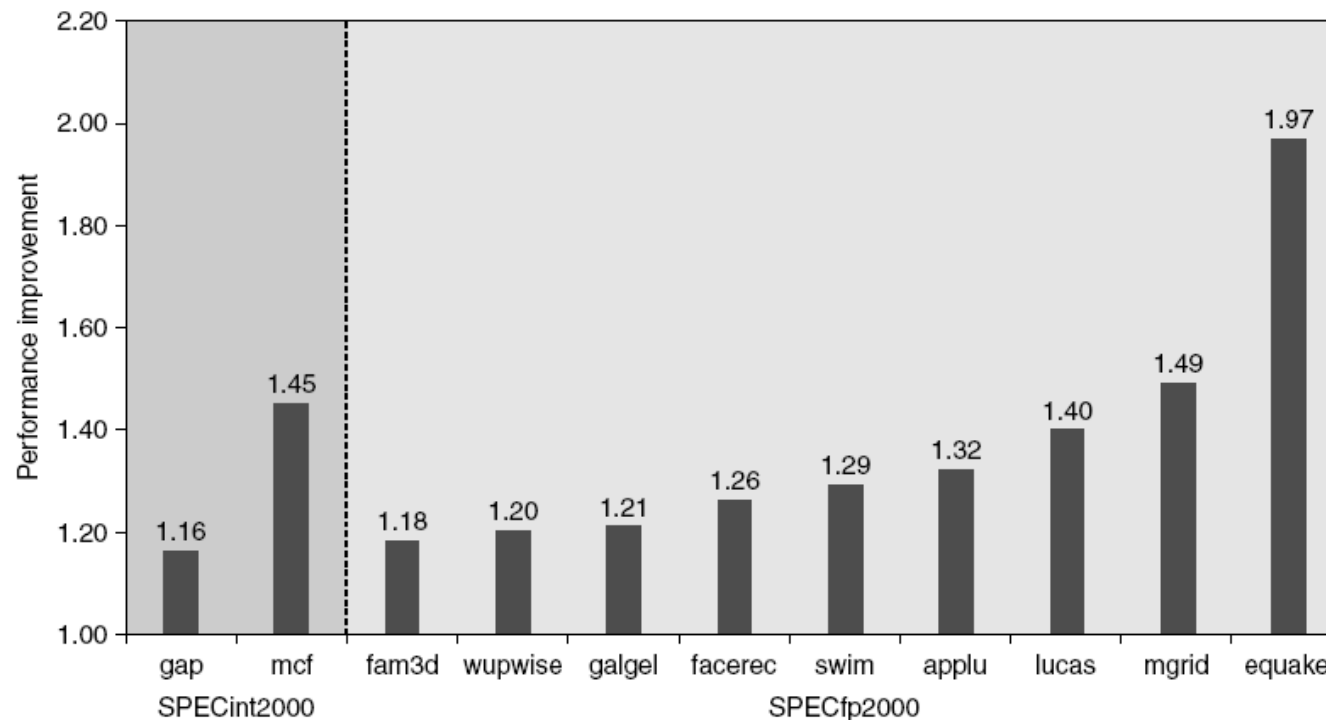
## 5. Reducing Misses by Hardware Pre-fetching of Instructions & Data

---

- **Data cache** block pre-fetching by “**D-stream buffer**”
  - 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
- Pre-fetching relies on **extra memory bandwidth** that can be used without penalty
- **Drawback:** If pre-fetching interferes with demand misses, it can lower performance

## 5. Hardware Pre-fetching Example

Fetching 2 blocks on a miss (include next sequential block) in Intel Pentium 4 increases performance up to 1.97



Intel Pentium 4 Pre-fetching



## 6. Reducing Misses by Software Pre-fetching Data

---

- **Compiler-controlled pre-fetching** (the compiler can help in reducing useless pre-fetching): Compiler inserts pre-fetch LOAD instructions to load data in registers/cache before they are needed
- Data Pre-fetch
  - Register Pre-fetch: Load data into register (HP PA-RISC loads)
  - Cache Pre-fetch: Load data into cache (MIPS IV, PowerPC, SPARC v. 9)
- Issuing pre-fetch instructions takes time (instr. overhead)
  - Is cost of pre-fetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth





## 7. Reducing Misses by Compiler Optimizations

---

- **Basic idea:** Apply profiling on SW applications, then use profiling info to apply code transformations
  - McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks **in software** by using profiling information
- **Managing instructions:**
  - Reorder instructions in memory so as to reduce conflict misses
  - Profiling to look at instruction conflicts



## 7. Reducing Misses by Compiler Optimizations

---

### ■ Managing Data

- a) **Merging Arrays:** improve **spatial locality** by single array of compound elements vs. 2 arrays (to operate on data in the same cache block)
- b) **Loop Interchange:** improve **spatial locality** by changing loops nesting to access data in the order stored in memory (re-ordering maximizes re-use of data in a cache block)
- c) **Loop Fusion:** improve **spatial locality** by combining 2 independent loops that have same looping and some variables overlap
- d) **Loop Blocking:** Improve **temporal locality** by accessing “sub-blocks” of data repeatedly vs. accessing by entire columns or rows



## a) 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];
```

- Reducing conflicts between val & key.
- Improve **spatial locality** by single array of compound elements vs. 2 arrays (to operate on data in the same cache block)



## b) Loop Interchange Example

Improve **spatial locality** by swapping nested loops to access data in the order as they are stored in memory

```
/* 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];
```

Advantage: Sequential accesses instead of striding through memory every 100 words.



## c) Loop Fusion Example

Improve **spatial locality** by combining 2 independent loops that have same looping indexes and some variables overlap

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

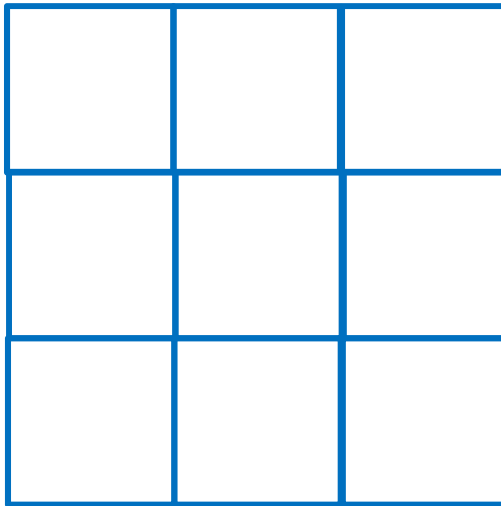
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {
    a[i][j] = 1/b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j];
  }
```

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



## d) Loop Blocking

- **Loop Blocking:** instead of accessing the matrix by entire rows or columns, subdivide the matrix into **BxB blocks** that fit the cache size to maximize accesses to data loaded in the cache before they are replaced
- **B** is called the **blocking factor**
- It improves the **locality** of accesses



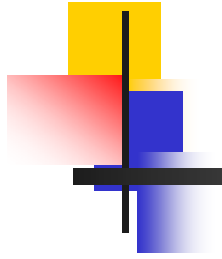


# Summary

---

## **How to reduce the miss rate:**

0. Reduce Misses by Larger Cache Sizes
1. Reduce Misses by Larger Block Size
2. Reduce Misses by Higher Associativity
3. Reducing Misses by Victim Cache
4. Reducing Misses by Pseudo-Associativity & Way Prediction
5. Reducing Misses by HW Prefetching Instructions / Data
6. Reducing Misses by SW Prefetching Data
7. Reducing Misses by Compiler Optimizations



## **2) How to reduce the miss penalty**



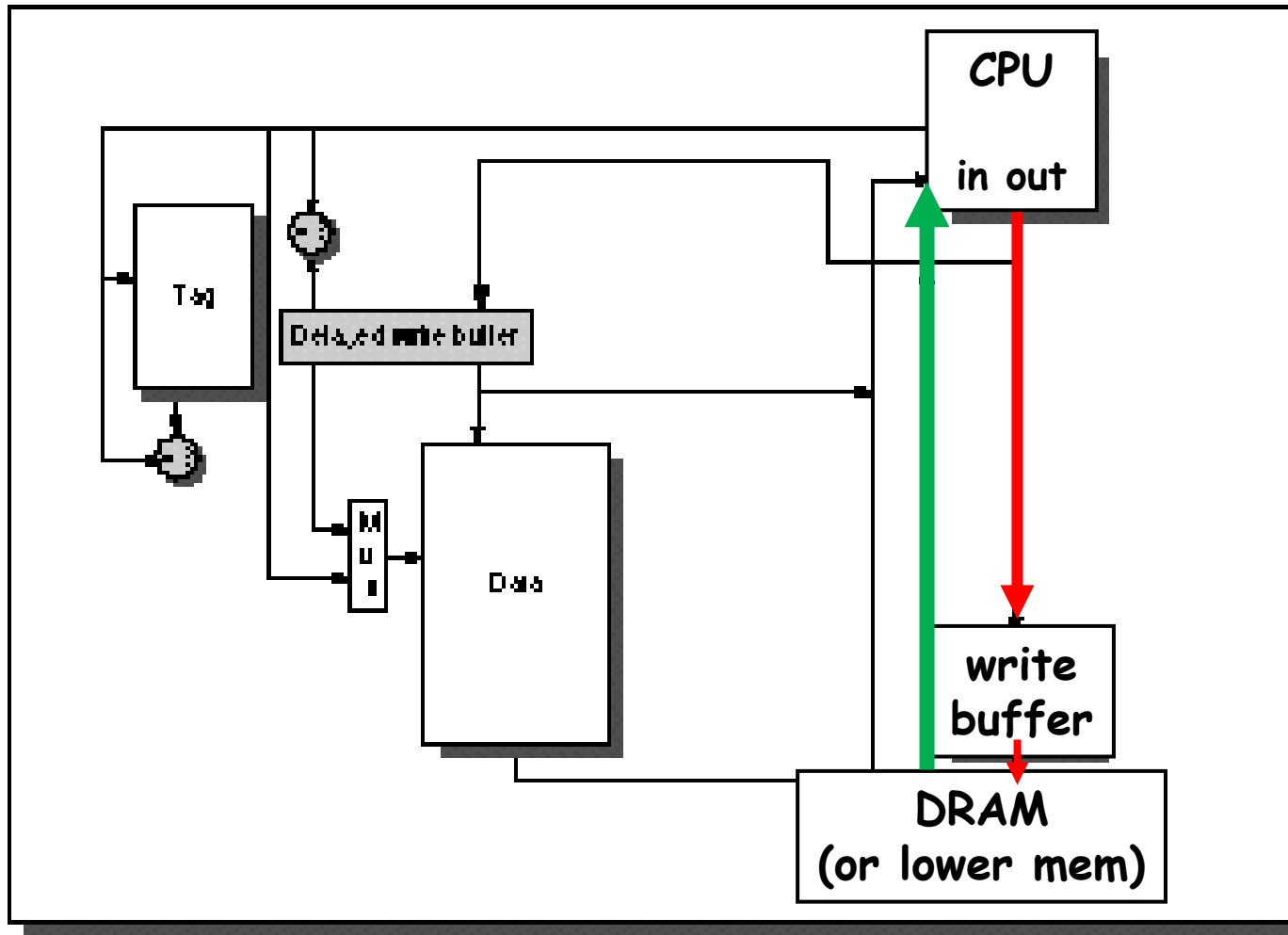


## 1. Reducing Miss Penalty: Read Priority over Write on Miss

---

- **Basic idea:** Giving higher priority to read misses over writes to reduce the miss penalty.
- The **write buffer** must be properly sized: Larger than usual to keep more writes in hold.
- **Drawback:** This approach can complicate the memory access because the **write buffer** might hold the updated value of a memory location needed on a read miss => RAW hazard through memory.

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





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

---

- **Write through** with write buffers might generate RAW conflicts with main memory reads on cache misses.
  - Check the contents of the write buffer on a read miss: ***if there are no conflicts***, let the memory access continue sending the read miss before the write
  - ***Otherwise***, the read miss has to wait until the write buffer is empty, but this might increase the read miss penalty.



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

---

## ■ Write Back

- Let's consider a read miss replacing a dirty block
- Instead of writing the dirty block to memory, then reading from memory, *we could copy the dirty block to a write buffer, then do the read miss, and then do the write to memory*
- CPU stalls are reduced since restarts as soon as do the read



## 2. Reducing Miss Penalty: Sub-block Placement

---

- Don't have to load full block on a miss: **move sub-blocks**
- We need **valid bits** per **sub-block** to indicate validity
- Drawback: no exploiting enough the spatial locality



### 3. Reducing Miss Penalty:

#### Early Restart and Critical Word First

---

- Usually the CPU needs just one word of the block on a miss.
- **Basic idea: Don't wait for full block** to be loaded before restarting CPU (by sending the requested missed word)
  - **Early restart:** Request the words in **normal order** from memory, but as soon as the requested word of the block arrives, send it to the CPU to let the CPU continue execution, while filling in the rest of the words in the cache block;
  - **Critical Word First:** Request the **missed word first** from memory and send it to the CPU as soon as it arrives to let the CPU continue execution, while filling the rest of the words in the cache block. This approach is also called *requested word first*.



### 3. Reducing Miss Penalty: Early Restart and Critical Word First

---

- Generally useful only for large blocks;
- Spatial locality tend to want next sequential words, so not clear if there is a real benefit by early restart;
- The benefits of this approach depend on the size of the cache block and the likelihood of another access to the portion of the block not yet been fetched.



## 4. Reducing Miss Penalty:

### Non-blocking Caches (*Hit under Miss*)

---

- ***Non-blocking cache*** (or ***lockup-free cache***) allows data cache **to continue to supply cache hits during a previous miss** (*Hit under Miss*);
- ***"Hit under Miss"*** reduces the effective miss penalty by working during a miss instead of stalling CPUs on misses
  - Requires **out-of-order** execution CPU: the CPU needs to do not stall on a cache miss: For example, the CPU can continue fetching instructions from I-cache while waiting for D-cache to return the missing data.
  - *This approach is a sort of "out-of-order" pipelined memory access*





## 4. Reducing Miss Penalty:

### Non-blocking Caches (*hit under miss*)

- ***"Hit under Multiple Miss" or "Miss under Miss"*** may further lower the effective miss penalty by overlapping multiple misses
  - Requires multiple memory banks (otherwise cannot support) to serve multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple simultaneous memory accesses
  - Pentium Pro allows 4 outstanding memory misses



## 5. Reducing Miss Penalty: Second Level Cache

---

### **Basic Idea: to introduce a second level cache (L2)**

- L1 cache small enough to match the fast CPU clock cycle
- **L2 cache** large enough to capture many accesses that would go to main memory reducing the effective miss penalty
- **L2 cache hit time** not tied to CPU clock cycle!
  - *Speed of L1 affects the CPU clock rate*
  - *Speed of L2 only affects the miss penalty of L1*

More in general, the second level cache concept can be applied recursively **to introduce multi-level caches** to reduce overall memory access time



## 5. Reducing Miss Penalty: Second Level Cache

---

- L2 Equations:

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

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

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



## 5. Reducing Miss Penalty: Second Level Cache

---

- Definitions:
  - **Local miss rate  $L_2$ :** misses in this cache divided by the total number of memory accesses *to this cache*  
**(Miss Rate<sub>L2</sub>)**
  - **Global miss rate:** misses in this cache divided by the total number of memory accesses *generated by the CPU*  
**(Miss Rate<sub>L1</sub> x Miss Rate<sub>L2</sub>)**
- **Global Miss Rate is what really matters:** it indicates what fraction of the memory accesses from CPU go all the way to main memory.

## 6. Reducing Miss Penalty: Merging Write Buffer

- Goal: To reduce stalls due to full write buffer
- Each entry of the Write Buffer can merge words from different memory addresses
- Write Buffer with 4 entries, where each entry holds 4 64-bit words:

Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

No write  
buffering

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Write buffering

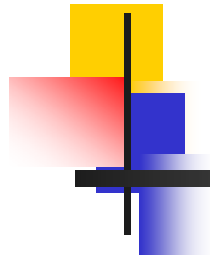


# Reducing Miss Penalty: Summary

---

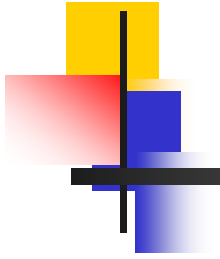
- **Six techniques:**

1. Read priority over write on miss
2. Sub-block placement
3. Early Restart and Critical Word First on miss
4. Non-blocking Caches (Hit under Miss, Miss under Miss)
5. Second Level Cache (and Multi-level Caches)
6. Merging Write Buffer  
*(Victim cache)*



# Cache Optimization Summary

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



### **3) How to reduce the hit time**

***Very important because the speed of L1 (hit time)  
affects the CPU clock rate!***

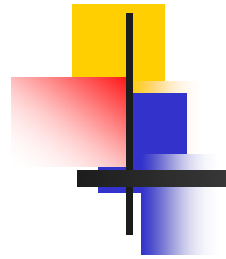




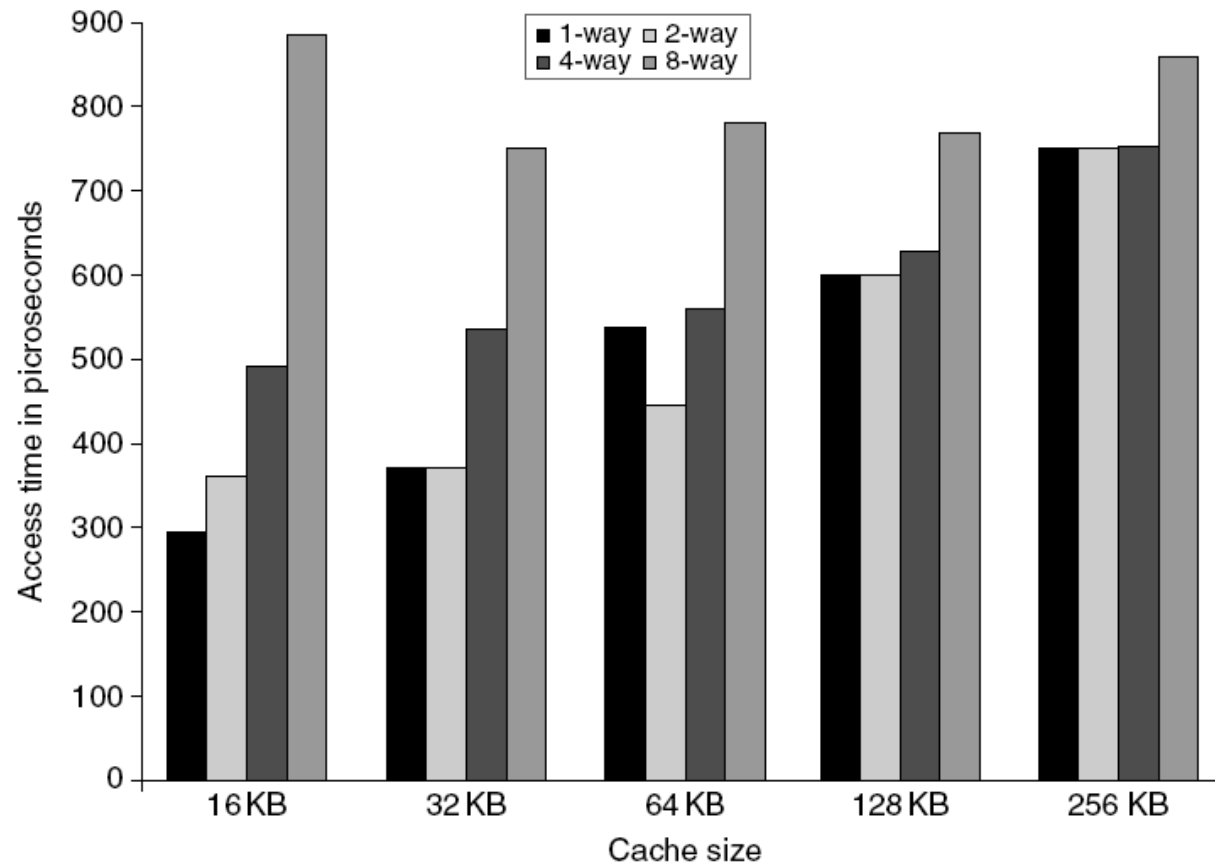
# 1. Fast Hit Times via Small and Simple L1 Caches

---

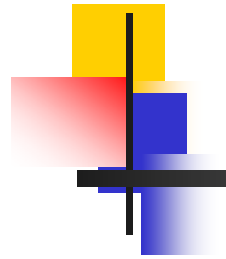
- Why Alpha 21164 has 8KB Instruction and 8KB data cache + 96KB second level cache?
  - Small data cache and fast clock rate
- **Direct Mapped, small and simple on-chip L1 cache**
  - Critical timing path:
    - addressing tag memory, then
    - comparing tags, then
    - selecting correct set
  - Direct-mapped caches can overlap tag compare and transmission of data
  - Lower associativity reduces power because fewer cache lines are accessed



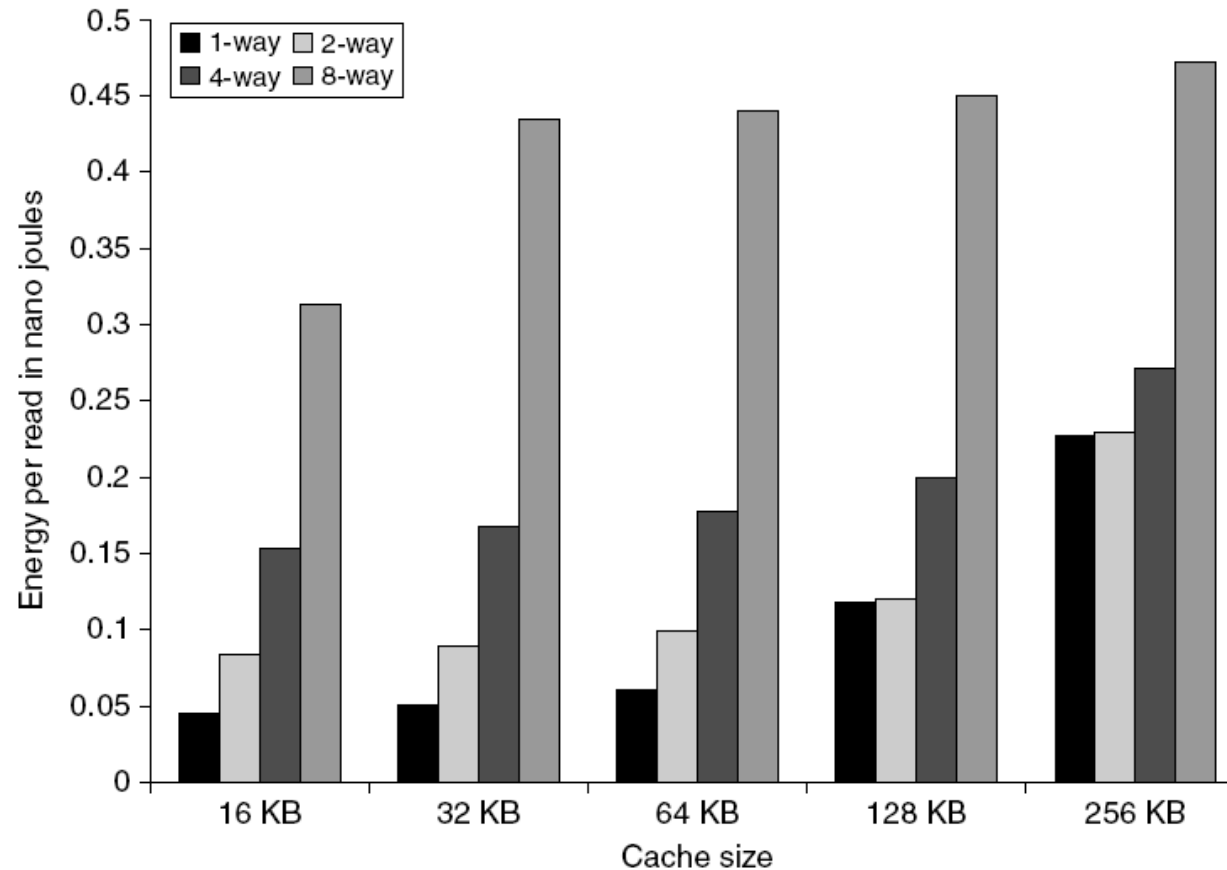
# L1 Size and Associativity



Access time vs. size and associativity



# L1 Size and Associativity



Energy per read vs. size and associativity



## 2. Fast Hit Times

### by 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
  - Dealing with ***aliases*** (sometimes called ***synonyms***); Two different virtual addresses map to same physical address
  - I/O must interact with cache, so need virtual address



## 2. Fast Hit Times

### by Avoiding Address Translation

---

- **Basic Idea:**

Avoiding virtual address translation during indexing of cache

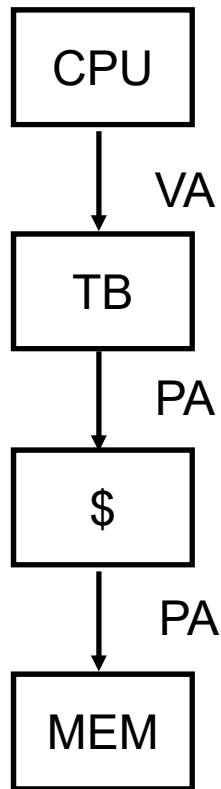
- **Index with Physical Portion of Virtual Address**

If index is physical part of address, can start tag access in parallel with address translation so that can compare to physical tag

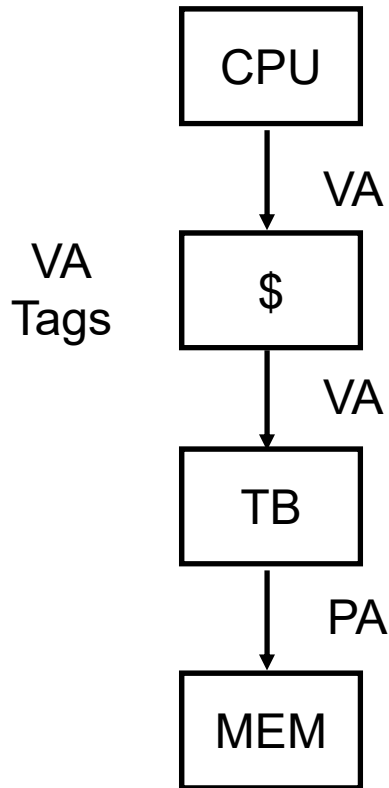
This approach limits cache size to page size: what if want bigger caches and uses same trick?

- Higher associativity moves barrier to right
- Page coloring

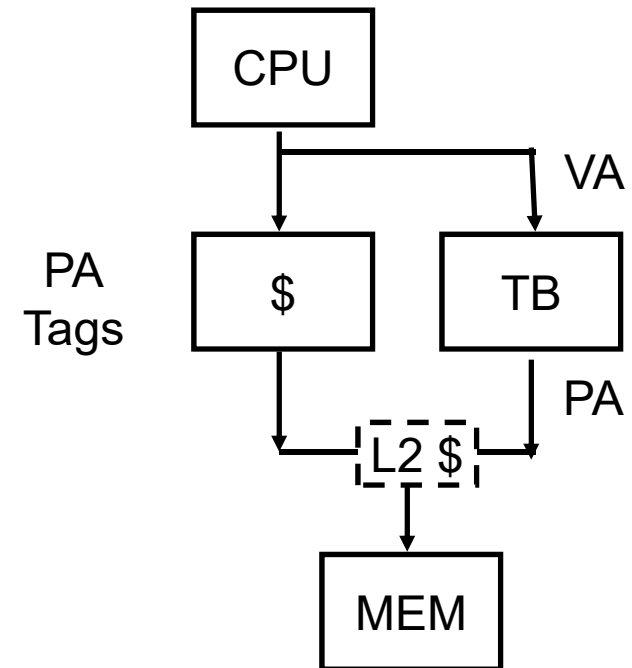
# Physically vs Virtually Addressed Caches



**Physically Addressed Cache**  
Conventional Organization



**Virtually Addressed Cache**  
Translate only on miss  
Synonym Problem



**Cache: Virtually Indexed & Physically Tagged**

Overlap cache access  
with VA translation:  
requires \$ index to  
remain invariant  
across translation



### 3. Fast Hit Times Via Pipelined Writes

---

- **Basic idea:** To pipeline Tag Check and Update Cache Data as separate stages: current write tag check & previous write cache update
- The “**Delayed Write Buffer**”; must be checked on reads; either complete write or read from write buffer



## 4. Fast Writes on Misses Via Small Sub-blocks for Write Through

- If most writes are 1 word, sub-block size is 1 word, & write through then always write sub-block & tag immediately
  - **Tag match and valid bit already set:** Writing the block was proper, & nothing lost by setting valid bit on again.
  - **Tag match and valid bit not set:** The tag match means that this is the proper block; writing the data into the sub-block makes it appropriate to turn the valid bit on.
  - **Tag mismatch:** This is a miss and will modify the data portion of the block. Since write-through cache, no harm was done; memory still has an up-to-date copy of the old value. Only the tag to the address of the write and the valid bits of the other sub-block need be changed because the valid bit for this sub-block has already been set
- Doesn't work with write back due to last case





# Cache Optimization Summary

	<i>Technique</i>	<i>MR</i>	<i>MP</i>	<i>HT</i>	<i>Complexity</i>
miss rate	Larger Block Size	+	—		0
	Higher Associativity	+		—	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Prefetching of Instr/Data	+			2
	Compiler Controlled Prefetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Subblock Placement		+	+	1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2
hit time	Small & Simple Caches	—		+	0
	Avoiding Address Translation			+	2
	Pipelining Writes			+	1

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.



# References

---

- Chapter 5 of: J. Hennessy and D. Patterson, "Computer Architecture, a Quantitative Approach", Morgan Kaufmann, Fourth Edition.
- Chapter 2 of: J. Hennessy and D. Patterson, "Computer Architecture, a Quantitative Approach", Morgan Kaufmann, Fifth Edition.

JOHN L. HENNESSY   DAVID A. PATTERSON

