



CACHE DESIGN

汪东升(Prof. Dongsheng Wang)

wds@tsinghua.edu.cn

清华大学 计算机系科学与技术系



阿拉伯数字



阿拉伯数字最初由古印度人发明，后由阿拉伯人传向欧洲，之后再经欧洲人将其现代化。正因阿拉伯人的传播，成为该种数字最终被国际通用的关键节点，所以人们称其为“阿拉伯数字”。

1	2	3	4	5	6	7	8	9	10	11	12	13	0
—	=	≡	≡	⋈	∩	+)()(┐	┐	┐	┐
20	30	40	50	80	88								
⋈	⋈	⋈	⋈)()(
100	162	200	500	600	656								
⊖	⊖	∩	=	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
1000	2000	3000	4000										
⋈	⋈	⋈	⋈										

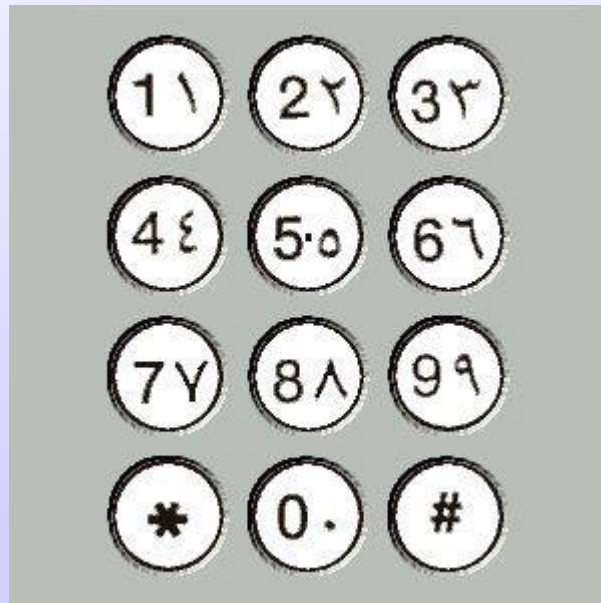
0和7



- 0 的出现是数学史上一大创造
- 0 乘以任何一个数，都使这个数变成0
- 大乘空宗由印度龙树及其弟子提婆所创立，强调“一切皆空”。0的这一特殊就反映了“一切皆空”这一命题所留下的痕迹。
- 0是正数和负数的分界点，也是解析几何中笛卡儿坐标轴上的原点。没有0也就没有原点，也就没有了坐标系，几何学大厦就会分崩离析。

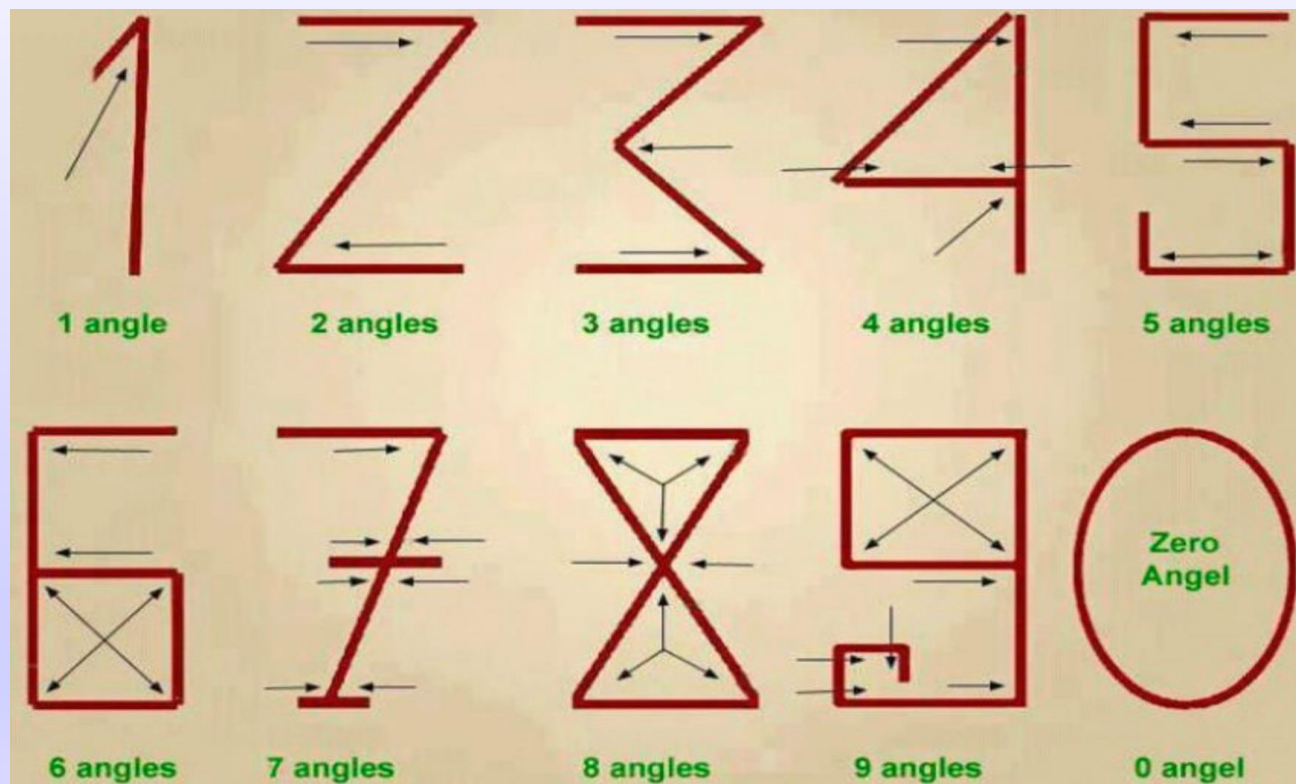


與某些地區使用的數字6區分





阿拉伯数字是古印度的数学家发明的，由阿拉伯商人传入欧洲





問題:

- 以色列希伯來文字為何從右向左書寫?





存储器的层次结构&Cache

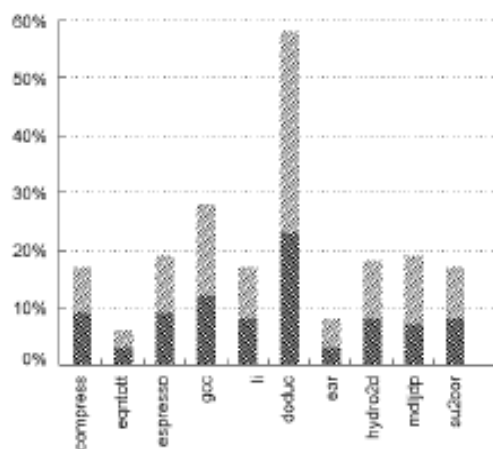
- Memory Hierarchy
 - Principle of locality
- Four Questions for Memory Hierarchy
 - Block placement: $y=f(x)$
 - Block identification: $y=f(a)$
 - Block replacement
 - Write strategy
- 4Cs of Cache Misses
 - Compulsory Misses: sad facts of life. Example: cold start misses.
 - Capacity Misses: increase cache size
 - Conflict Misses: increase cache size and/or associativity
 - Coherence: Misses caused by cache coherence
- Cache Performance
 - Reduce the miss rate,
 - Reduce the miss penalty, or
 - Reduce the time to hit in the cache.

Principle of Locality

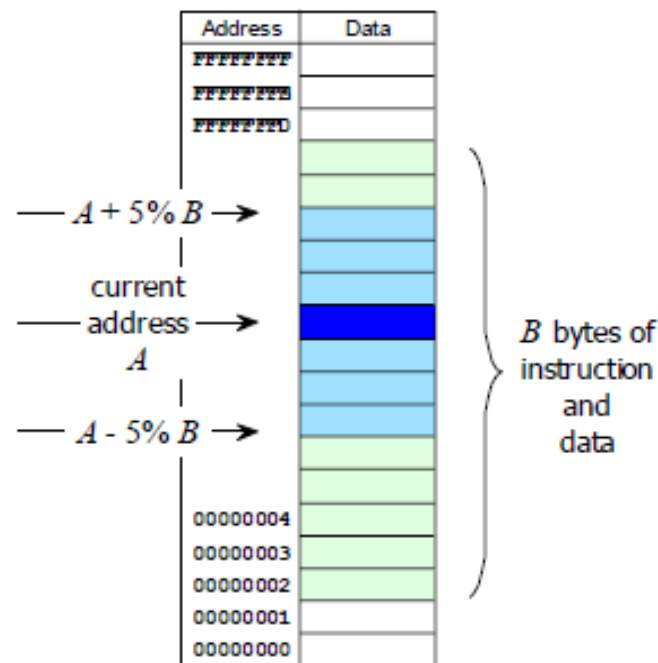
Locality — small proportion of memory accounts for most run time

Rule of thumb — For 90% of run time next instruction/data will come from 10% of program/data closest to current instruction

Amdahl's Law — make access to most local memory as fast as possible



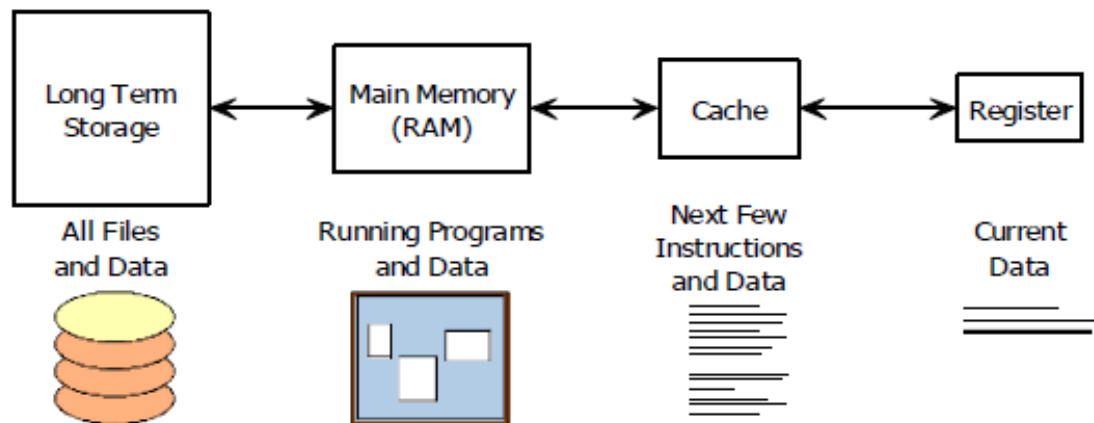
Percentage of memory accounting for 90% of run time for SPEC 92





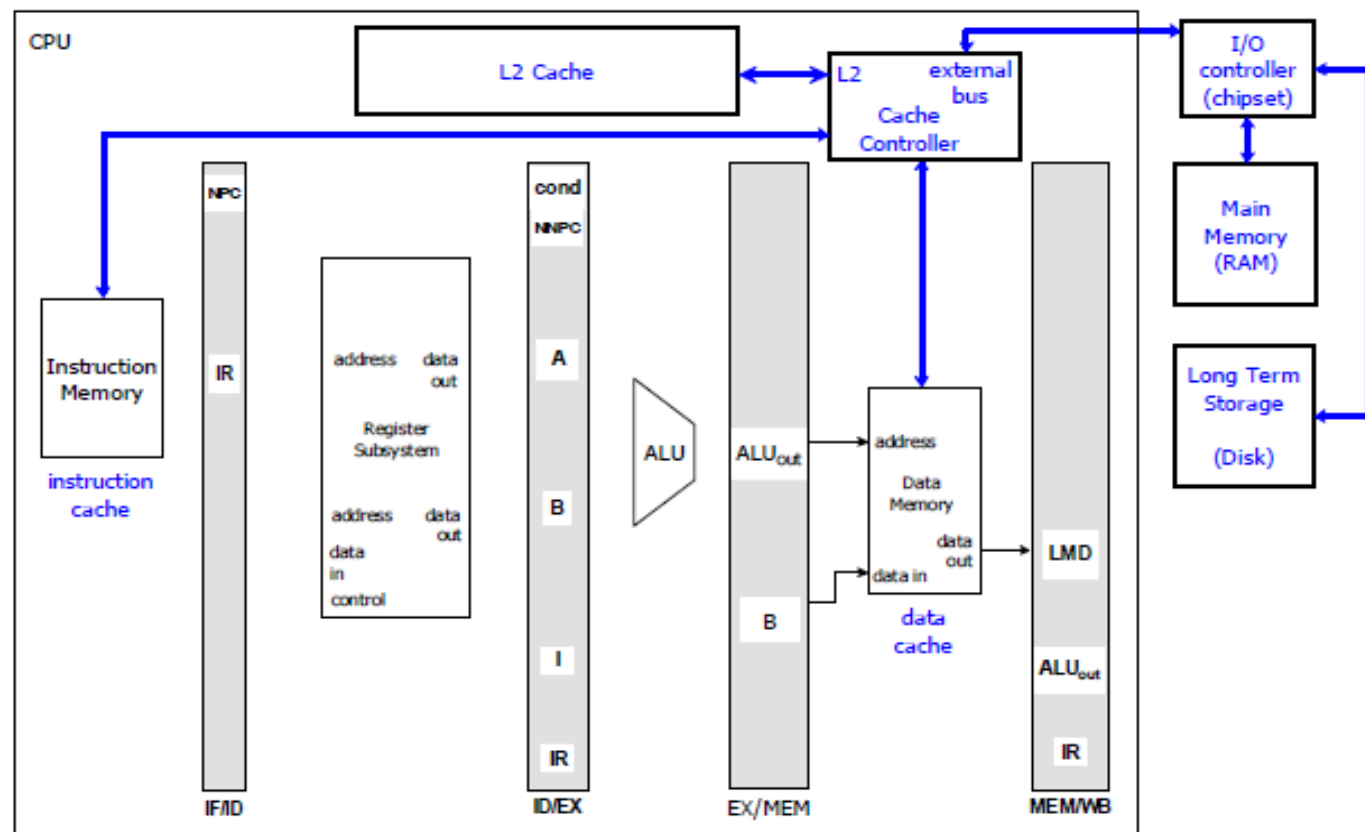
Memory Hierarchy

Memory locations outside CPU and RAM	Memory location outside CPU	Memory location in or near CPU	Memory location inside CPU
Stores data and instructions of "all" programs	Stores "all" data and instructions of running programs	Fast access to important data and instructions from RAM	Fast access to small amount of information
Organized by OS	Organized by addresses	Copy of RAM section	Organized by CPU





CPU and Memory Hierarchy





CPU and Memory Hierarchy — 1

L1 (level 1 cache) holds copy of small section of main memory

Most recently accessed addresses (memory locations)

L1 split into physically separate Instruction Cache and Data Cache

CPU accesses L1 cache directly

If (address in L1 cache) {access performed in 1 clock cycle}

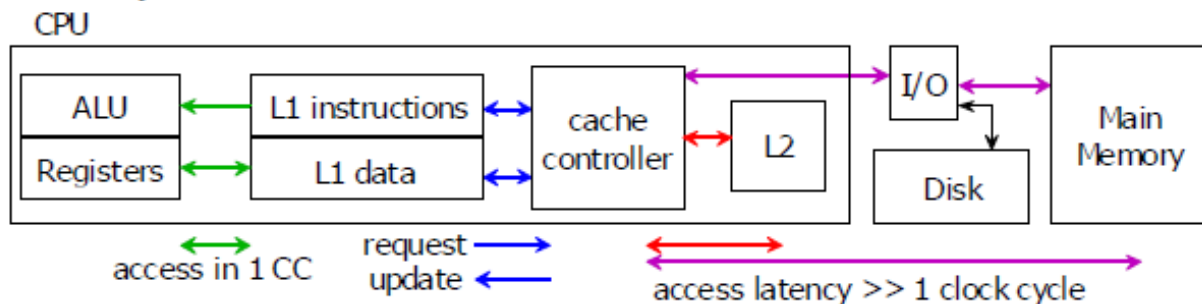
Else {

L1 cache accesses cache controller

If (address in L2 cache) {controller copies contents to L1 from L2}

Else {controller copies location to L1 from main memory}

}





CPU and Memory Hierarchy — 2

CPU accesses disk and I/O devices by memory addressing

Part of total address space reserved for I/O and storage devices

Disk write of k bytes — CPU performs k stores to same I/O address

Disk read of k bytes — CPU performs k loads from same I/O address

I/O addresses are not copied to cache (marked **NON-CACHEABLE**)

Virtual Memory

Total memory space larger than physical memory

Divided into pages of specific size

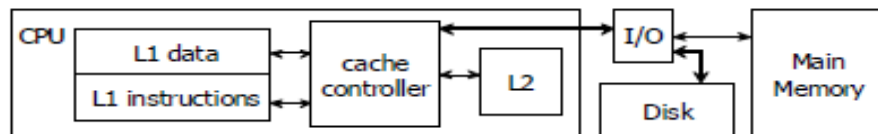
Infrequently used pages moved to "page file" on disk

Page properties and location specified in page tables

Virtual address divided into

Page address — points to page table entry

Offset — points to byte address in page





Memory Organization — 1

N-bit address space

Physical Address = $A_{N-1} A_{N-2} \dots A_1 A_0$

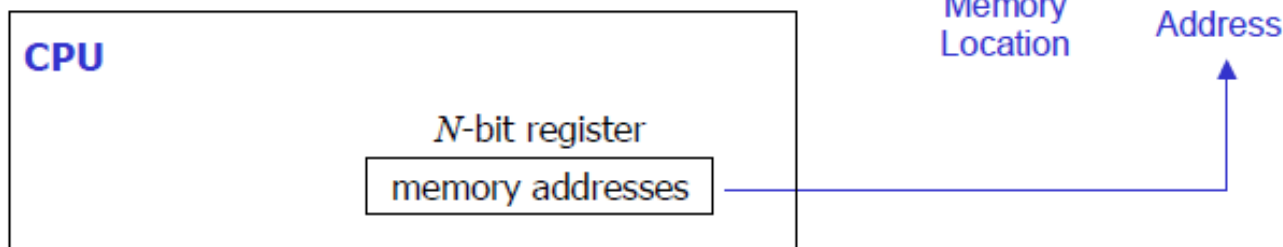
Can form 2^N addresses, from 0 to $(2^N - 1)$

Every byte in RAM has an *N*-bit address

Processor refers to memory locations by
physical RAM addresses

Processor stores memory addresses in
N-bit address registers

Data Byte	11111...111
Data Byte	11111...110
Data Byte	11111...101
Data Byte	11111...100
...	...
Data Byte	00000...111
Data Byte	00000...110
Data Byte	00000...101
Data Byte	00000...100
Data Byte	00000...011
Data Byte	00000...010
Data Byte	00000...001
Data Byte	00000...000

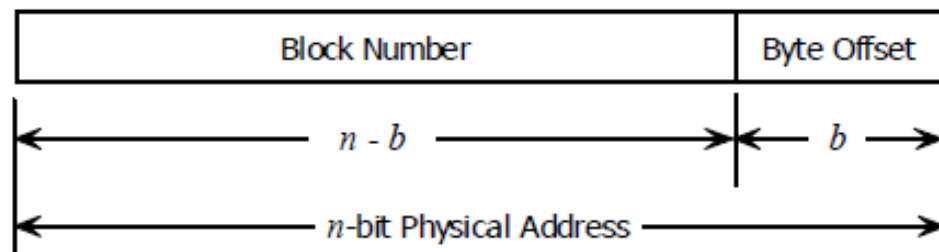




Memory Organization — 2

Address space has n -bit physical address

$N = 2^n$ byte address space



Address space divided (logically) into address **BLOCKS** (lines)

Block size $B = 2^b$ bytes / block

Blocks in address space $= N / B = 2^{n-b}$

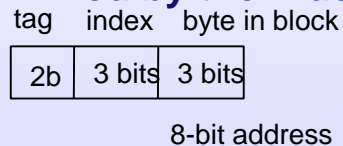
$(n-b)$ -bit Block Number $= \text{Int}(\text{Address} / B)$

b -bit byte offset $= \text{Address} \% B = 0, 1, \dots, B - 1 = 2^b - 1$



Blocks and Addressing the Cache

- Memory is logically divided into fixed-size **blocks**
- Each block maps to a location in the cache, determined by the **index bits** in the address
 - used to index into the tag and data stores
- Cache access:
 - 1) index into the tag and data stores with index bits in address
 - 2) check valid bit in tag store
 - 3) compare tag bits in address with the stored tag in tag store
- If a block is in the cache (cache hit), **the stored tag should be valid and match the tag of the block**





Where Misses Come From?

■ Classifying Misses: 3 Cs

- **Compulsory** — The first access to a block is not in the cache,
so the block must be brought into the cache.
Also called **cold start misses** or **first reference misses**.
(Misses in even an Infinite Cache)
- **Capacity** — If the cache cannot contain all the blocks
needed during execution of a program, capacity misses will
occur due to blocks being discarded and later retrieved.
- **Conflict** — If block-placement strategy is set associative or
direct mapped, conflict misses (in addition to compulsory &
capacity misses) will occur because a block can be
discarded and later retrieved if too many blocks map to its
set. Also called collision misses or **interference misses**.

■ More recent, 4th “C”:

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

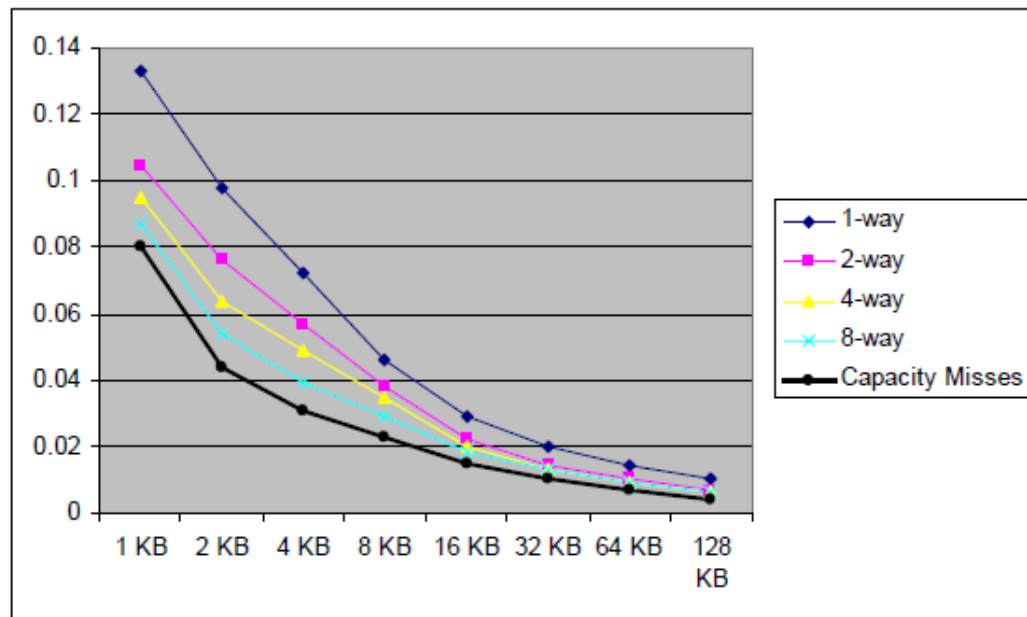


Effect of Cache Parameters on Performance

- Larger cache size
 - reduces Capacity misses
 - hit time will increase.
- Higher associativity
 - reduces conflict misses
 - may increase hit time
- Larger block size
 - reduces compulsory misses
 - exploit burst transfers in memory and on buses
 - increases miss penalty and conflict misses.



Total Miss Rate



Total miss rate drops as capacity or associativity increases



Cache Organization?

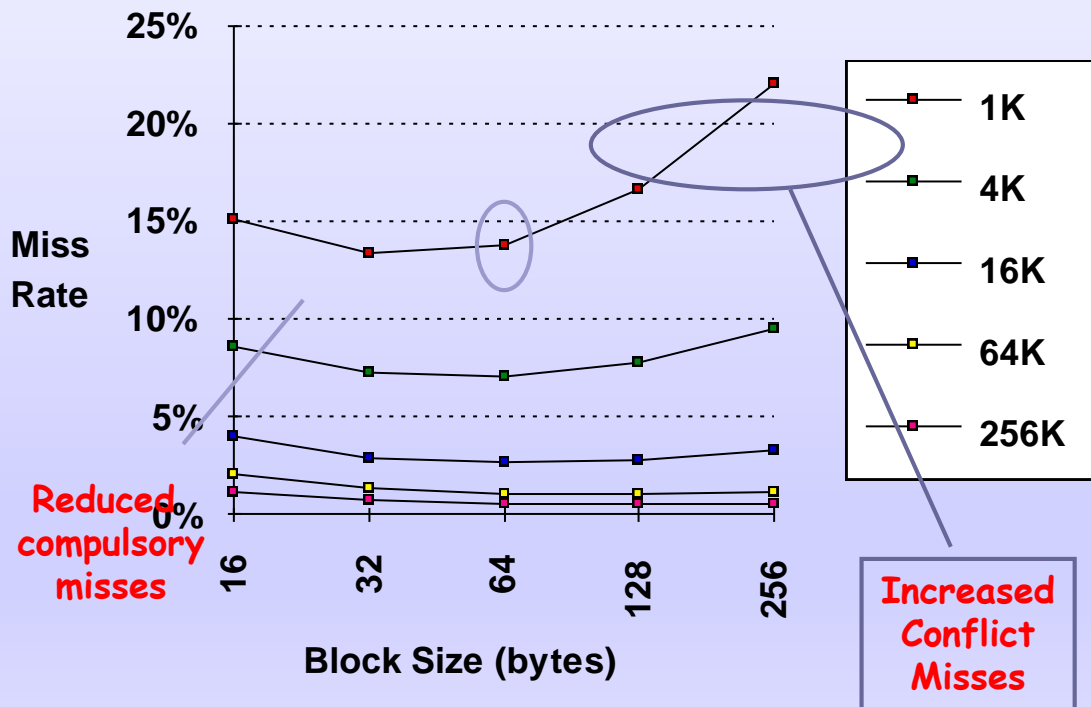
Assume **total cache size** not changed

What happens if:

- Change Block Size
- Change Cache Internal Organization
- Change Associativity
- Change Compiler
- Which of 3Cs is obviously affected?



1st Miss Rate Reduction Technique: Larger Block Size





1st Miss Rate Reduction Technique: Larger Block Size (cont'd)

■ Example:

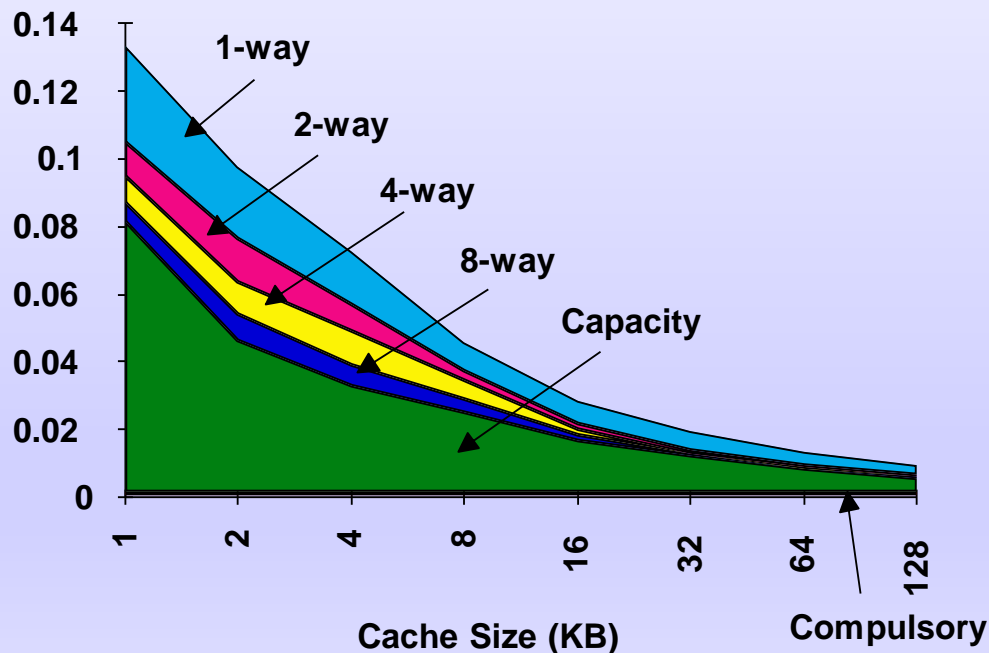
- Memory system takes 40 clock cycles of overhead, and then delivers 16 bytes every 2 clock cycles
- Miss rate vs. block size (see table); hit time is 1 cc
- AMAT? $AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$

	Cache Size							Cache Size				
BS	1K	4K	16K	64K	256K	BS	MP	1K	4K	16K	64K	256K
16	15.05	8.57	3.94	2.04	1.09	16	42	7.32	4.60	2.66	1.86	1.46
32	13.34	7.24	2.87	1.35	0.70	32	44	6.87	4.19	2.26	1.59	1.31
64	13.76	7.00	2.64	1.06	0.51	64	48	7.61	4.36	2.27	1.51	1.25
128	16.64	7.78	2.77	1.02	0.49	128	56	10.32	5.36	2.55	1.57	1.27
256	22.01	9.51	3.29	1.15	0.49	256	72	16.85	7.85	3.37	1.83	1.35

- **Block size** depends on both **latency and bandwidth** of **lower level memory**
- **low latency and bandwidth => decrease block size**
- **high latency and bandwidth => increase block size**

2nd Miss Rate Reduction Technique: Larger Caches

- Reduce Capacity misses
- Drawbacks: Higher cost, Longer hit time



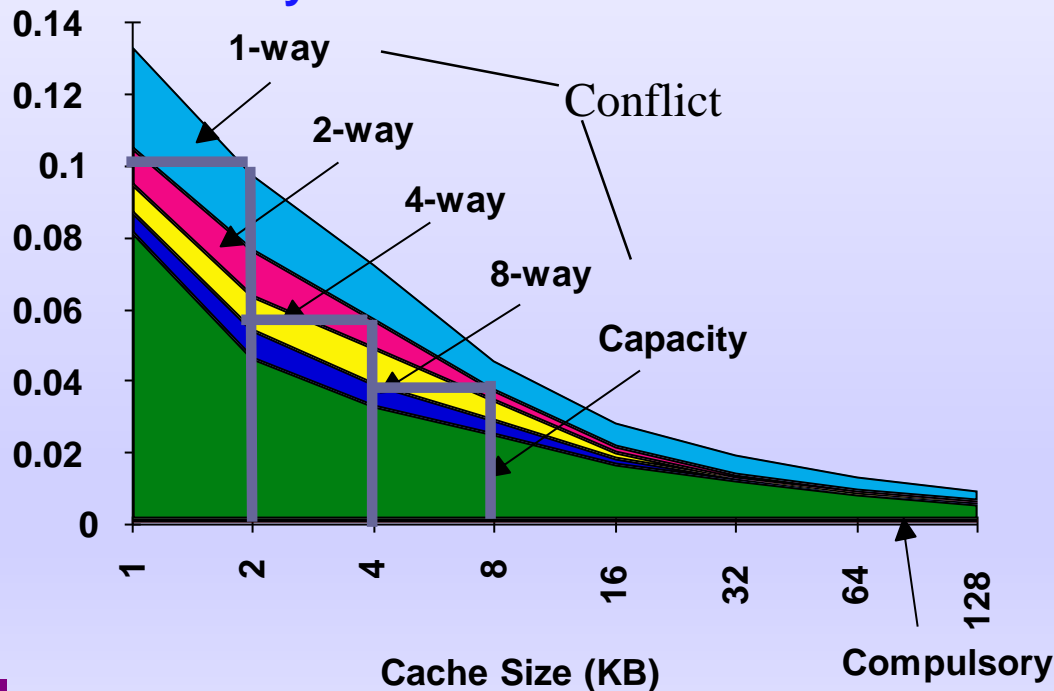


3rd Miss Rate Reduction Technique: Higher Associativity

- Miss rates improve with higher associativity
- Two rules of thumb
 - **8-way set-associative** is almost as effective in reducing misses as **fully-associative cache** of the same size
 - **2:1 Cache Rule**: Miss Rate DM cache size N = Miss Rate 2-way cache size $N/2$
- **Beware: Execution time is only final measure!**
 - Will Clock Cycle time increase?
 - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%

3rd Miss Rate Reduction Technique: Higher Associativity (2:1 Cache Rule)

Miss rate 1-way associative cache size X
= Miss rate 2-way associative cache size $X/2$





4th Miss Rate Reduction Technique: Way Prediction, “Pseudo-Associativity”

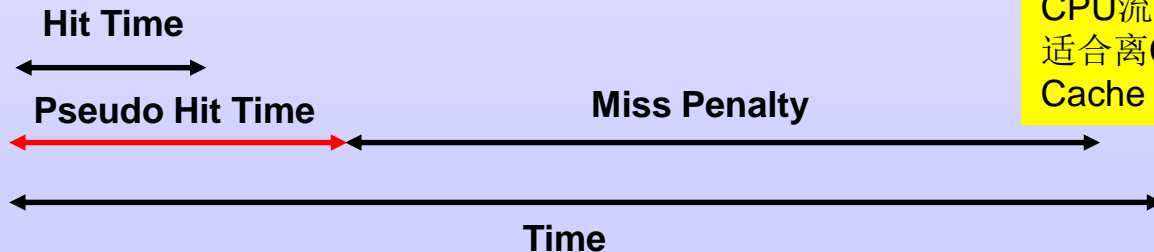
- How to combine **fast hit time of Direct Mapped** and have the **lower conflict misses of 2-way SA cache**?
- Way Prediction: **extra bits** are kept to predict the way or block within a set
 - **Mux** is set early to select the desired block
 - Only a single tag comparison is performed
 - What if 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 proc. to lower power



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

■ Pseudo-Associative Cache

- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)
- Accesses proceed just as in the DM cache for a hit
- On a miss, check the second entry
 - Simple way is to invert the MSB bit of the INDEX field to find the other block in the “pseudo set”



■ What if too many hits in the slow part?

- swap contents of the blocks



5th Miss Rate Reduction Technique: Compiler Optimizations

- Reduction comes from software
- McFarling [1989] reduced caches misses by 75% (8KB, DM, 4 byte blocks) in software
- Instructions
 - Reorder procedures in memory so as to reduce conflict misses
 - Profiling to look at conflicts(using tools they developed)
- Data
 - **Merging Arrays**: improve spatial locality by single array of compound elements vs. 2 arrays
 - **Loop Interchange**: change nesting of loops to access data in order stored in memory
 - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
 - **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows



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 => reduce misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded



Loop Interchange Example

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

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



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

Reduces misses if the arrays do not fit in the cache.



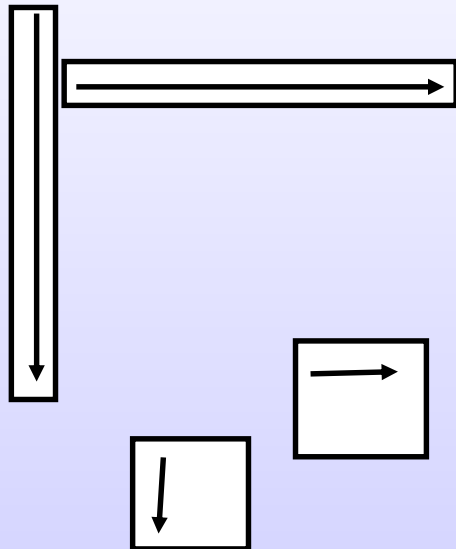
Blocking

- **Motivation:** multiple 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 is replaced

Blocking Example

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

- Two Inner Loops:
 - Read all $N \times N$ elements of $z[]$
 - Read N elements of 1 row of $y[]$ repeatedly
 - Write N elements of 1 row of $x[]$
- Idea: compute on $B \times B$ submatrix that fits





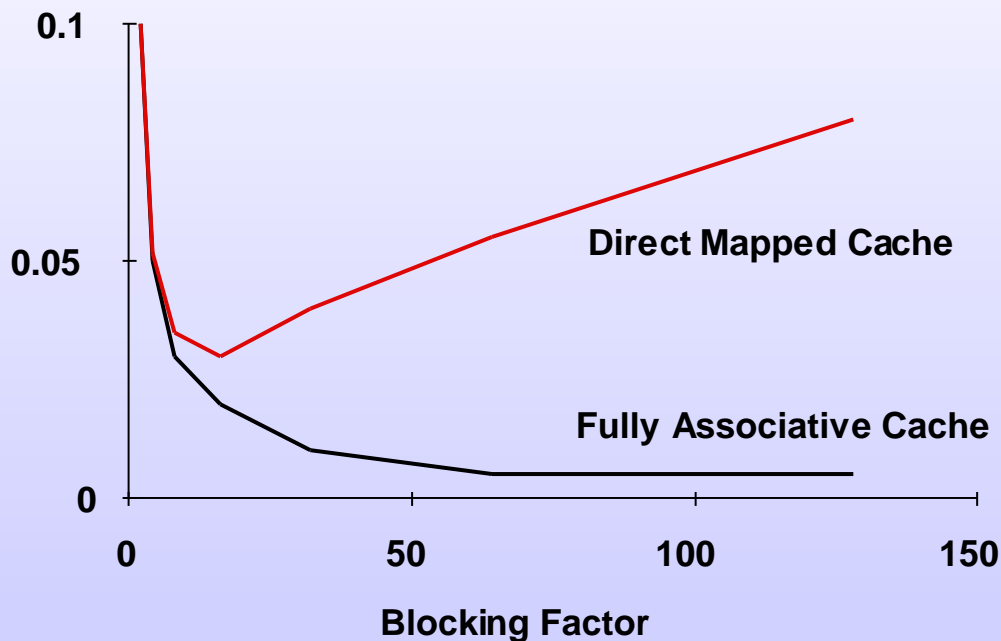
Blocking Example (cont'd)

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

- B called *Blocking Factor*
- Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$



Reducing Conflict Misses by Blocking



- Conflict misses in caches not FA vs. Blocking size

- Lam et al [1991] a blocking factor of 24 had 1/5 the misses vs. 48 despite both fit in cache



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**



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];
```



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

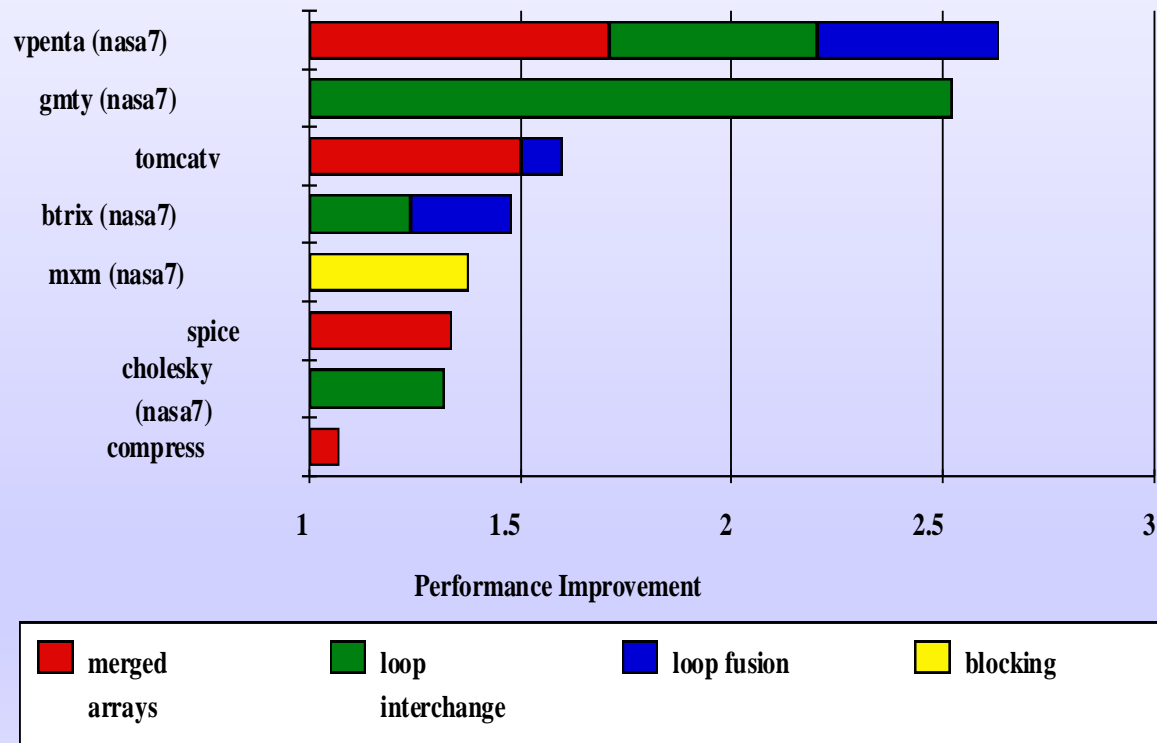
```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];

/* 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];
    }
```

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



Summary of Compiler Optimizations to Reduce Cache Misses (by hand)





Summary: Miss Rate Reduction

$$CPU\ time = \frac{IC \times \left(CPI_{Exec} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right)}{Clock\ rate}$$

■ 3 Cs: Compulsory, Capacity, Conflict

- ☐ 1. Larger Cache => Reduce Capacity
- ☐ 2. Larger Block Size => Reduce Compulsory
- ☐ 3. Higher Associativity => Reduce Confilcts
- ☐ 4. Way Prediction & Pseudo-Associativity
- ☐ 5. Compiler Optimizations



Types of Memory

Type	Size	Speed	Cost/bit
Register	< 1KB	< 1ns	\$\$\$\$
On-chip SRAM	8KB-6MB	< 10ns	\$\$\$
Off-chip SRAM	1Mb – 16Mb	< 20ns	\$\$
DRAM	64MB – 1TB	< 100ns	\$
Disk	40GB – 1PB	< 20ms	~0



Why Memory Hierarchy?

- Need lots of bandwidth

$$\begin{aligned} BW &= \frac{1.0 \text{ inst}}{\text{cycle}} \times \left[\frac{1 \text{ Ifetch}}{\text{inst}} \times \frac{4B}{\text{Ifetch}} + \frac{0.4 \text{ Dref}}{\text{inst}} \times \frac{4B}{\text{Dref}} \right] \times \frac{1 \text{ Gcycles}}{\text{sec}} \\ &= \frac{5.6 \text{ GB}}{\text{sec}} \end{aligned}$$

- Need lots of storage
 - 64MB (minimum) to multiple TB



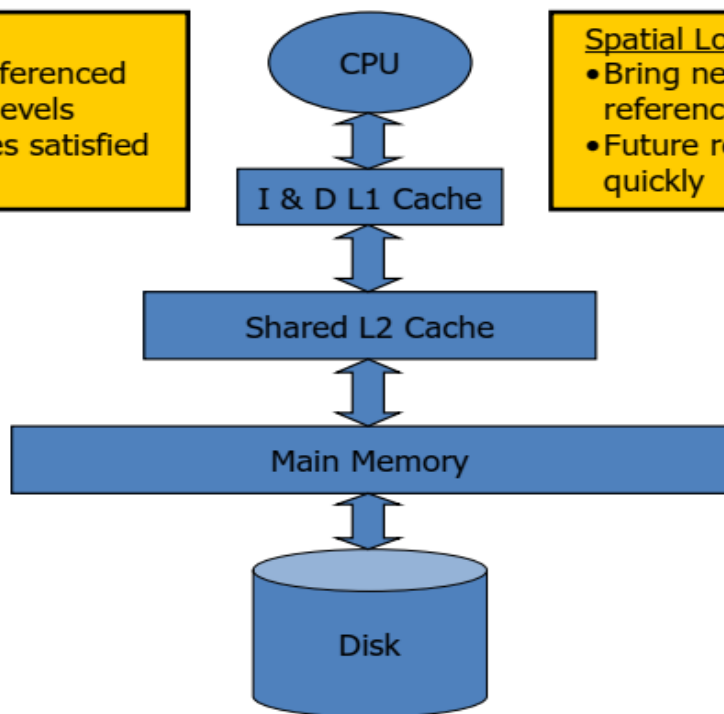
Memory Hierarchy

Temporal Locality

- Keep recently referenced items at higher levels
- Future references satisfied quickly

Spatial Locality

- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly





Reducing Miss Penalty

■ Motivation

- $AMAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Technology trends =>
relative cost of miss penalties increases over time

■ Techniques that address miss penalties

1. Multilevel Caches
2. Critical Word First and Early Restart
3. Giving Priority to Read Misses over Writes
4. Merging Write Buffer
5. Victim Caches



1st Miss Penalty Reduction Technique: Multilevel Caches

■ Architect's dilemma

- Should I make the cache faster to keep pace with the speed of CPUs
- Should I make the cache larger to overcome the widening gap between CPU and main memory

■ L2 Equations

- **AMAT** = Hit Time_{L1} + Miss Rate_{L1} x Miss Penalty_{L1}
- **Miss Penalty_{L1}** = Hit Time_{L2} + Miss Rate_{L2} x Miss Penalty_{L2}
- **AMAT** = Hit Time_{L1} + Miss Rate_{L1} x (Hit Time_{L2} + Miss Rate_{L2} + Miss Penalty_{L2})



Reducing Misses: Which apply to L2 Cache?

■ Reducing Miss Rate

- ☐ 1. Reduce Capacity Misses via **Larger Cache**
- ☐ 2. Reduce Compulsory Misses via **Larger Block Size**
- ☐ 3. Reduce Conflict Misses via Higher Associativity
- ☐ 4. Reduce Conflict Misses via Way Prediction & Pseudo-Associativity
- ☐ 5. Reduce Conflict/Capac. Misses via Compiler Optimizations

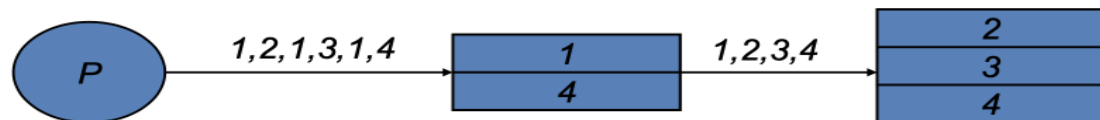


Multilevel Inclusion: Yes or No?

- **Inclusion property:**
L1 data are always present in L2
 - Good for I/O & caches consistency
(L1 is usually WT, so valid data are in L2)
- **Drawback: What if measurements suggest smaller cache blocks for smaller L1 caches and larger blocks for larger L2 caches?**
 - E.g., Pentium4: 64B L1 blocks, 128B L2 blocks
 - Add complexity: when replace a block in L2 should discard 2 blocks in the L1 cache => increase L1 miss rate
- **What if the budget for a L2 cache is slightly bigger than the L1 cache => L2 keeps redundant copy of L1?**
 - Multilevel Exclusion: L1 data is never found in a L2 cache
 - E.g., AMD Athlon uses this:
64KB L1I\$ + 64KB L1D\$ vs. 256KB L2U\$



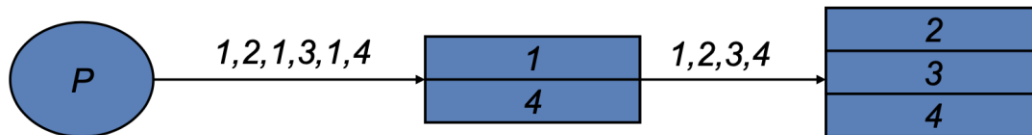
Multilevel Inclusion



- Example: local LRU not sufficient to guarantee inclusion
 - Assume L1 holds two and L2 holds three blocks
 - Both use local LRU
- Final state: L1 contains 1, L2 does not
 - Inclusion not maintained
- Different block sizes also complicate inclusion



Multilevel Inclusion



- Inclusion takes effort to maintain
 - Make L2 cache have bits or pointers giving L1 contents
 - Invalidate from L1 before replacing from L2
 - In example, removing 1 from L2 also removes it from L1
- Number of pointers per L2 block
 - L2 blocksize/L1 blocksize
- Reading list: [Wang, Baer, Levy ISCA 1989]



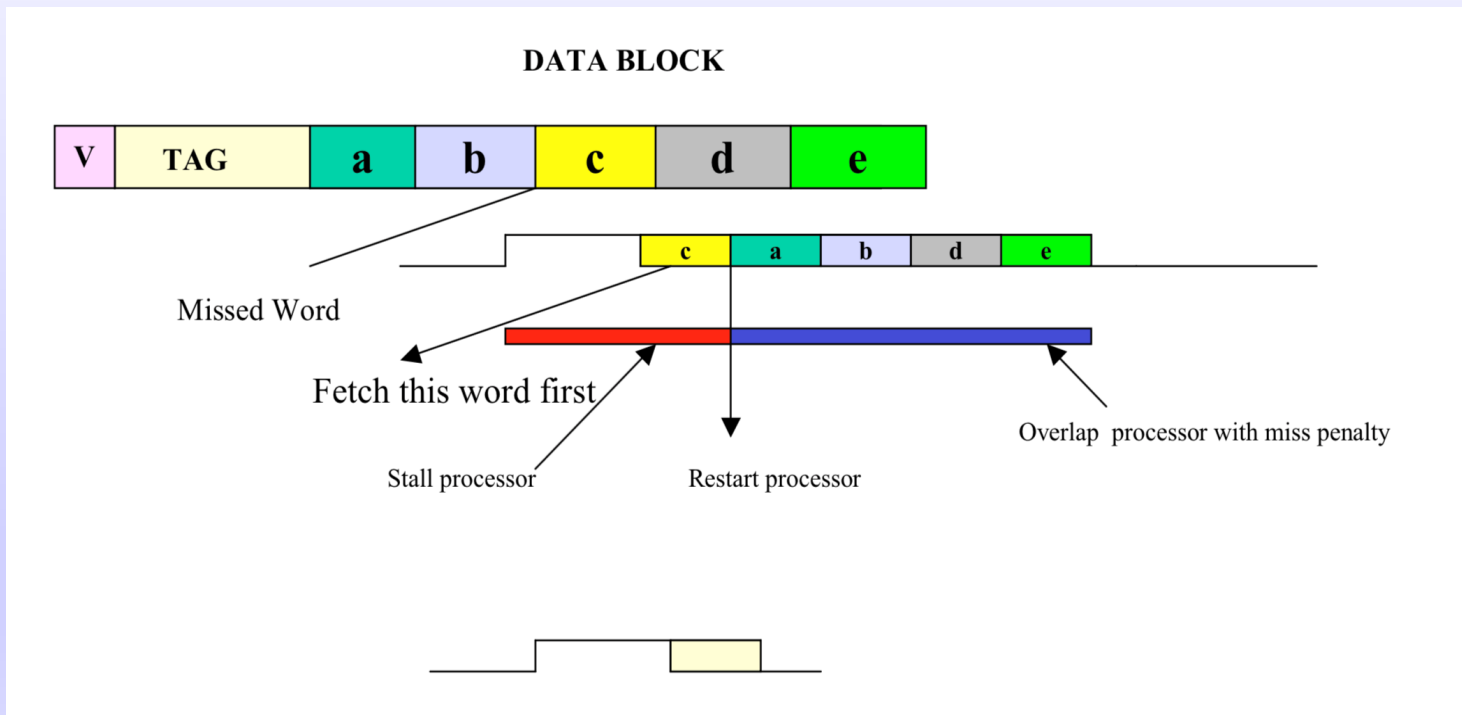
2nd Miss Penalty Reduction Technique: Early Restart and Critical Word First

- 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 the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Generally useful only in large blocks





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





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

- Write-through with write buffers offer RAW conflicts with main memory reads on cache misses

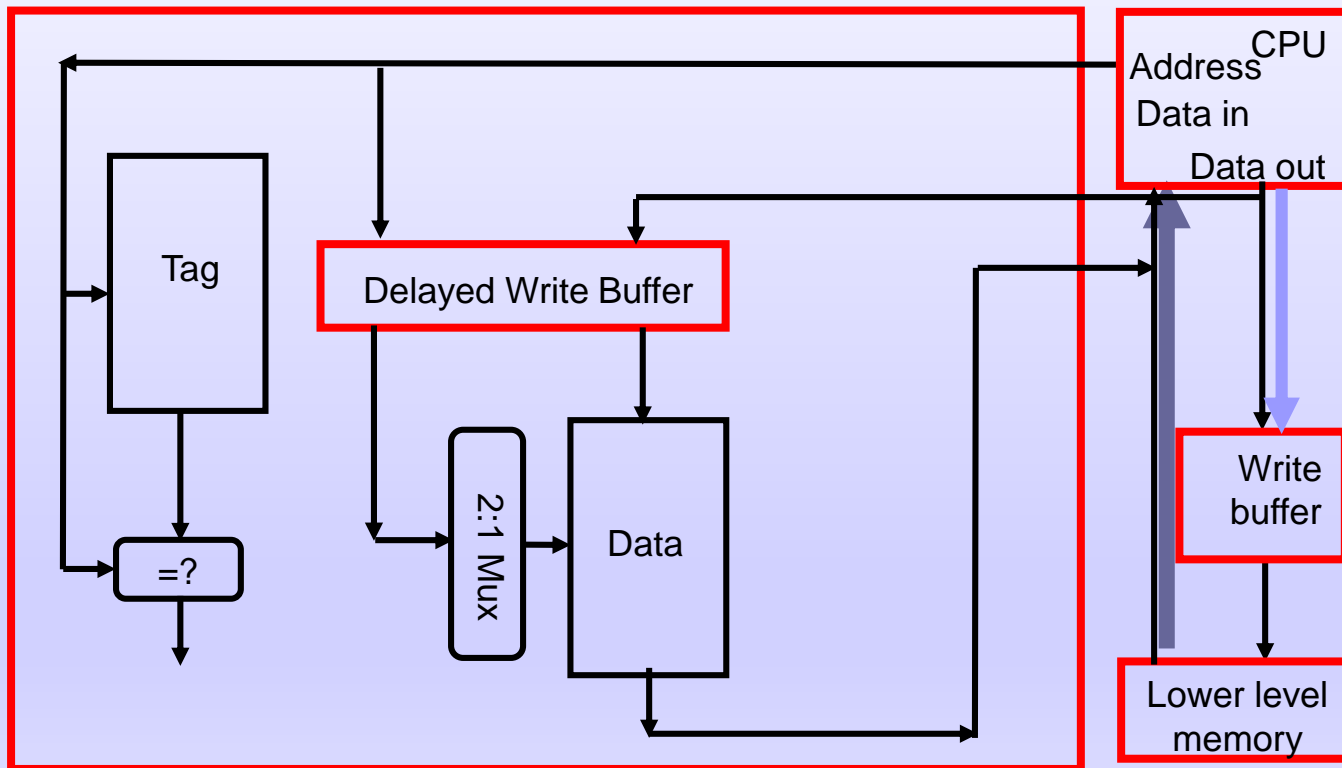
Example: DM, WT, 512 & 1024 map to the same block (Index 0)

```
SW 512 (R0) , R3      ;  
LW R1 , 1024 (R0)    ; miss  
LW R2 , 512 (R0)     ; miss    R2=R3?
```

- ☐ 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, read memory, else read from Write Buffer
- **Write-back also want buffer to hold misplaced 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 do read



3rd Miss Penalty Reduction Technique: Giving Read Misses Priority over Writes



4th Miss Penalty Reduction Technique: Merging Write Buffer

- Write Through caches relay on write-buffers
 - 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 are faster than a single word writes => reduce write-buffer stalls
- Is this applicable to I/O addresses?

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	

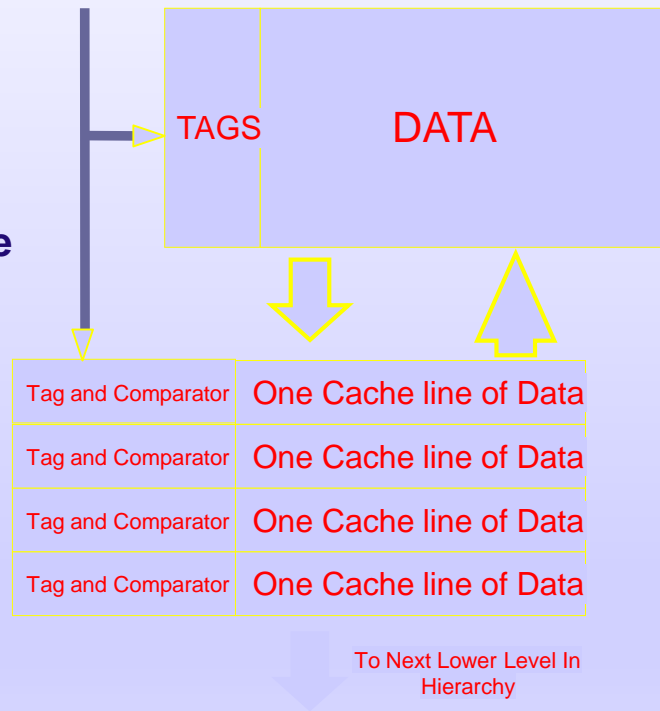
Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

© 2003 Elsevier Science (USA). All rights reserved.



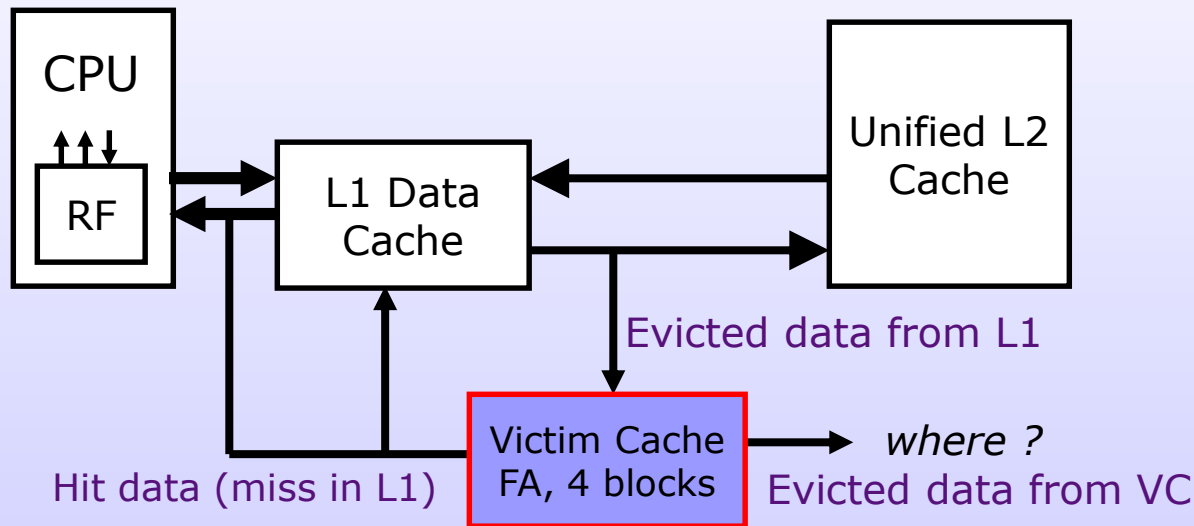
5th Miss Penalty Reduction Technique: Victim Caches

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- **Idea:** Add buffer to place data discarded from cache in the case it is needed again
- Jouppi [1990]:
4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines, AMD Athlon (8 entries)





Victim Caches (HP 7200)

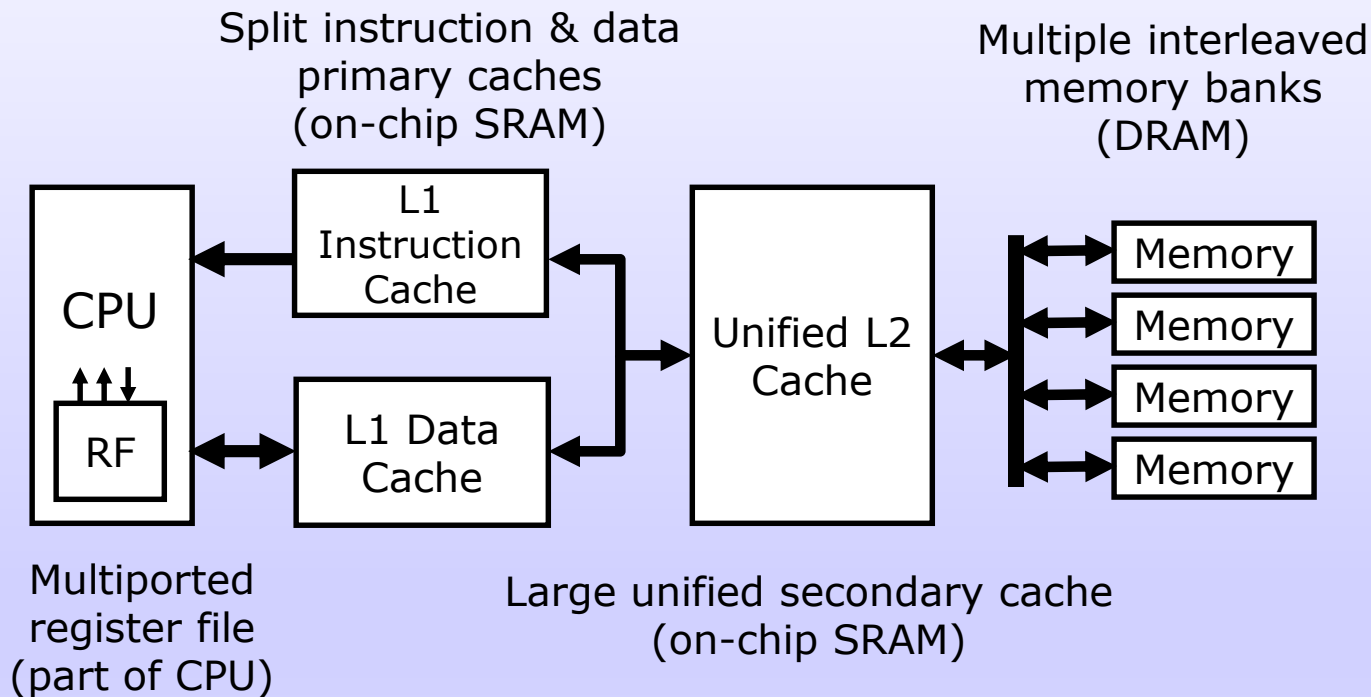


Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines

- First look up in direct mapped cache
- If miss, look in victim cache
- If hit in victim cache, **swap** hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim->?

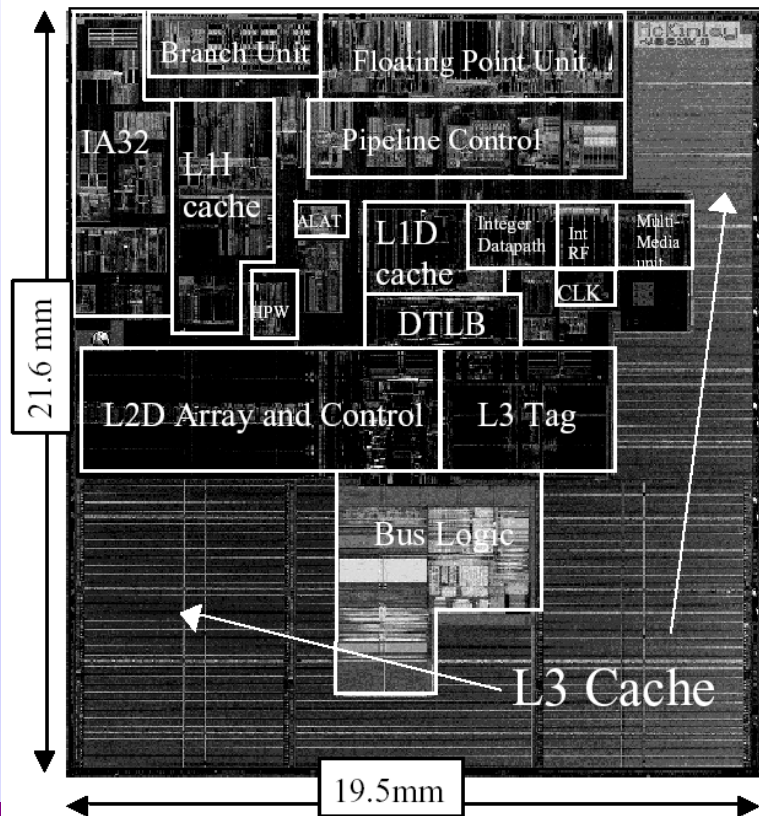
Fast hit time of direct mapped but with reduced conflict misses

A Typical Memory Hierarchy c.2006



Itanium-2 On-Chip Caches

(Intel/HP, 2002)



Level 1, 16KB, 4-way s.a., 64B line, **quad-port** (2 load+2 store), single cycle latency

Level 2, 256KB, 4-way s.a, 128B line, **quad-port** (4 load or 4 store), five cycle latency

Level 3, 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency



Summary of Miss Penalty Reducing Techniques

- 1. Multilevel Caches
- 2. Critical Word First and Early Restart
- 3. Giving Priority to Read Misses over Writes
- 4. Merging Write Buffer
- 5. Victim Caches



Reducing Misses/Penalty by Hardware Prefetching of Instructions & Data

■ E.g., **Instruction** Prefetching

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

■ Works with **data blocks** too:

- 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**



Reducing Misses/Penalty by Software Prefetching Data

■ Data Prefetch

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

■ Prefetching comes in two flavors:

- ☐ **Binding prefetch:** Requests load directly into register.
 - Must be correct address and register!
- ☐ **Non-Binding prefetch:** Load into cache.
 - Can be incorrect. Faults?

■ Issuing Prefetch Instructions takes time

- ☐ Is cost of prefetch issues < savings in reduced misses?
- ☐ Higher superscalar reduces difficulty of issue bandwidth



Review: Improving Cache Performance

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

$$AMAT = \text{HitTime} + \text{MissRate} \cdot \text{MissPenalty}$$

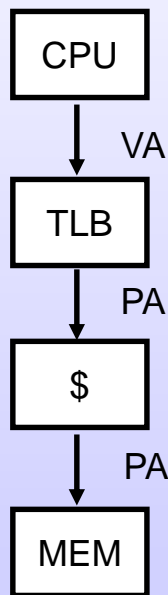


1st Hit Time Reduction Technique: Small and Simple Caches

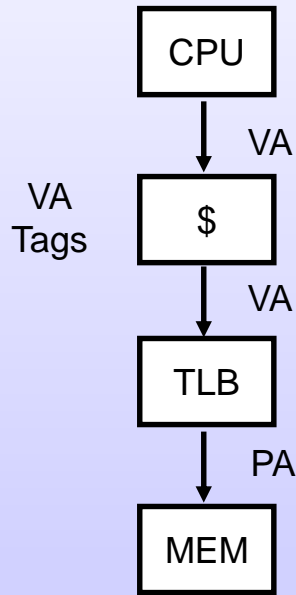
- Smaller hardware is faster =>
small cache helps the hit time
- Keep the cache small enough to fit on the same chip as the processor (avoid the time penalty of going off-chip)
- Keep the cache simple
 - Use Direct Mapped cache:
it overlaps the tag check
with the transmission of data

*[design issues more complex with out-of-chip cache that doesn't increase hit time past 1-2 cycles (approx 8-32KB in modern technology)
-order superscalar processors]*

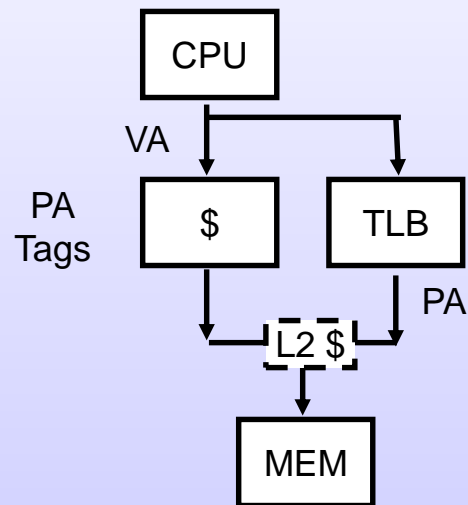
2nd Hit Time Reduction Technique: Avoiding Address Translation



Conventional
Organization



Virtually Addressed Cache
Translate only on miss
Synonym Problem



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



2nd Hit Time Reduction Technique: Avoiding Address Translation (cont'd)

- 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 =>
multiple copies of the same data in a virtual cache
 - **I/O** typically uses physical addresses; if I/O must interact with cache, mapping to virtual addresses is needed
- Solution to aliases
 - HW solutions guarantee every cache block a unique physical address
- 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



Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	-	Unified (instruction and data)
L3 cache size	-	8 MiB, shared
L3 cache associativity	-	16-way set associative
L3 replacement	-	Approximated LRU
L3 block size	-	64 bytes
L3 write policy	-	Write-back, Write-allocate
L3 hit time	-	35 clock cycles



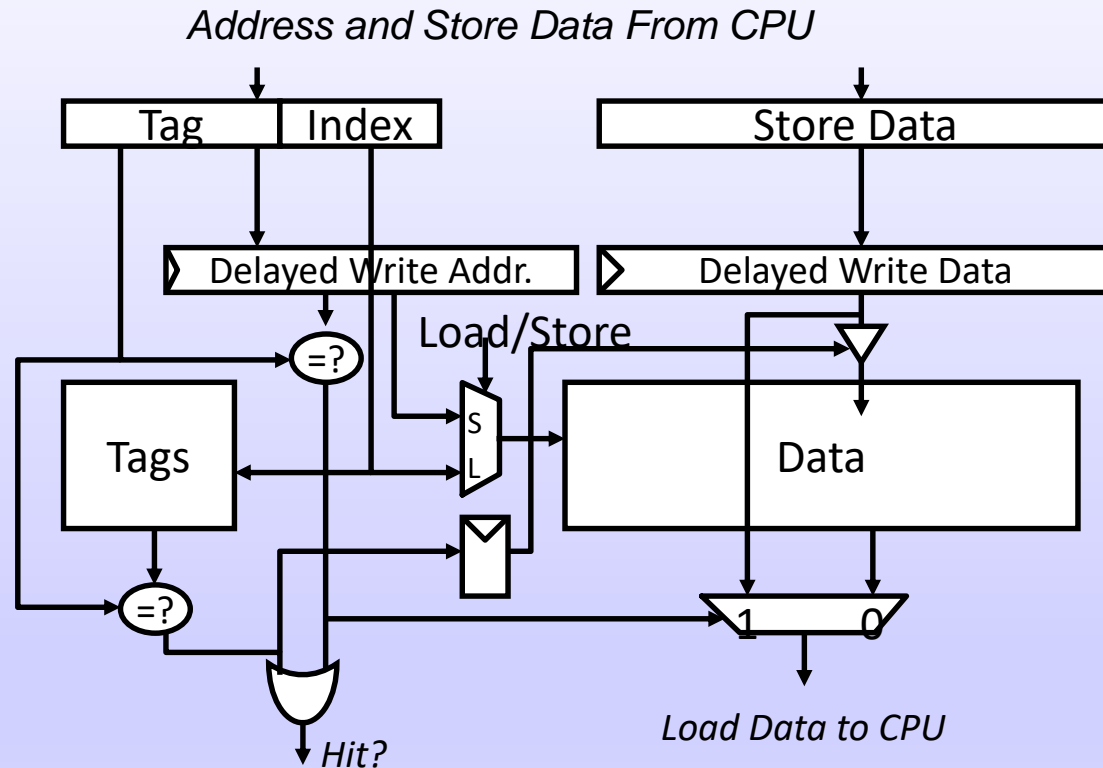
Reducing Write Hit Time

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

Solutions:

- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check
- Fully-associative (CAM Tag) caches: Word line only enabled if hit

Pipelining Cache Writes



Data from a store hit is written into data portion of cache during tag access of subsequent store