

Lec 11

How to improve cache performance (cont.)



Cache performance metrics

- Miss rate

- Independent of the speed of hardware.

- Average memory access time(AMAT)

- Better than miss rate , but
- Indirect measure of performance

- CPUtime

Ex1: Impact on Performance

- Suppose a processor executes at
 - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
 - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty
- Calculate the AMAT and real CPI.

• **Answer:** $CPI = \text{ideal CPI} + \text{average stalls per instruction}$

$$\begin{aligned} &= 1.1(\text{cycles/ins}) + [0.30 (\text{DataMops/ins}) \\ &\quad \times 0.10 (\text{miss/DataMop}) \times 50 (\text{cycle/miss})] + \\ &\quad 1 (\text{InstMop/ins}) \times 0.01 (\text{miss/InstMop}) \times 50 (\text{cycle/miss})] \\ &= (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1 \end{aligned}$$

• 58% of the time the proc is stalled waiting for memory!

• $AMAT = (1/1.3) \times [1.1 + 0.01 \times 50] + (0.3/1.3) \times [1.1 + 0.1 \times 50]$

• $= 2.54$

Ex2: Impact on Performance

Assume : Ideal CPI=1 (no misses)

- L/S's structure . 50% of instructions are data accesses
- Miss penalty is 25 clock cycles
- Miss rate is 2%
- How faster would the computer be if all instructions were cache hits?

- **Answer:** first compute the performance for always hits:

$$\text{CPU}_{\text{execution time}} = (\text{CPU clock cycles} + \text{memory stall cycles}) \times \text{clock cycle}$$

$$= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle}$$

$$= \text{IC} \times 1.0 \times \text{clock cycle}$$

Answer for example 2 (cont.)

Now for the computer with the real cache, first compute memory

$$\begin{aligned} \text{Memory stall cycles} &= IC \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Missrate} \times \text{Miss penalty} \\ &= IC \times (1 + 0.5) \times 0.02 \times 25 = IC \times 0.75 \end{aligned}$$

The total performance is thus:

$$\begin{aligned} \text{CPU execution time cache} &= (IC \times 1.0 + IC \times 0.75) \times \text{Clock cycle} \\ &= 1.75 \times IC \times \text{Clock cycle} \end{aligned}$$

The performance ratio is the inverse of the execution times

$$\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} = \frac{1.75 \times IC \times \text{Clock cycle}}{1.0 \times IC \times \text{clock cycle}}$$

The computer with no cache misses is 1.75 time faster.

Ex3: Impact on Performance

Assume : unified caches: 32K unified cache

- Split cache: 16K D-cache and 16K I-cache
- 36% of the instructions are data transfer instructions
- A hit takes 1 colck cycle
- The miss penalty is 100 clock cycles
- A load/store **take 1 extra clock cycle on a unified cache**
- Write-through with a write-buffer
and ignore stalls due to the write buffer
- **What is the average memory access time in each case?**

MR for Uni.cache & split cache

Miss per 1000 instructions for 2-way associate cache.

size	Instruction Cache	Data Cache	Unified Cache
8KB	8.16	44.0	63.0
16KB	3.82	40.9	51.0
32KB	1.36	38.4	43.3
64KB	0.61	36.9	39.4
128KB	0.30	35.3	36.2
256KB	0.02	32.6	32.9



Answer for example 3

Answer : first let's convert misses per 1000 instructions into miss rate.

$$\text{Miss rate} = \frac{\text{Misses / 1000 instruction}}{\text{Memory accesses/instruction}}$$

Since every instruction access has exactly one memory access to fetch the instruction, according as Figure 5.8 the instruction miss rate is

$$\text{Miss rate}_{16\text{KB instruction}} = \frac{3.82/1000}{1.0} = 0.0038$$

Since 36% of the instructions are data transfers, according as Figure 5.8 the data miss rate is

$$\text{Miss rate}_{16\text{KB data}} = \frac{40.9/1000}{0.36} = 0.1136$$

Answer for example 3 (cont.)

Form Figure 5.8 The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{32\text{KB unified}} = \frac{43.3/1000}{1.00+0.36} = 0.0318$$

Basing on Figure 2.32 on page 138 there is 74% instruction references in split cache. The average miss rate for the split cache is:

$$(74\% \times 0.0038) + (26\% \times 0.1136) = 0.0323$$

Thus , a 32KB unified cache has a slightly lower effective miss rate than two 16KB caches.

Answer for Example3 (cont.)

- The average memory access time can be divided into instruction and data accesses:

Average memory access time

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

- Therefore, the time for each organization is

Average memory access time_{split}

$$= 74\% \times (1 + 0.0038 \times 100) + 26\% \times (1 + 0.1136 \times 100) \\ = (74\% \times 1.38) + (26\% \times 12.36) = 1.021 + 3.214 = 4.25$$

Average memory access time_{unified}

$$= 74\% \times (1 + 0.0318 \times 100) + 26\% \times (1 + 1 + 0.0318 \times 100) \\ = (74\% \times 4.18) + (26\% \times 5.18) = 3.093 + 1.347 = 4.40$$

Ex4: Impact on Performance

Assume: in-order execution computer, such as the Ultra SPARC III.

Miss penalty: 100 clock cycles

Miss rate : 2%

Memory references Per instruction: 1.5

Average cache misses per 1000 instructions: 30

$CPI = 1.0$ (ignoring memory stalls)

What is the impact on performance when behavior of the cache is included (Calculate the impact using both misses per instruction and miss rate.)?

Answer for example 4

Answer : The performance, including cache misses, is

$$CPU_{time} = IC \times \left(CPI_{execution} + \frac{Memstallclockcycles}{Instruction} \right) \times Clockcycletime$$

$$\begin{aligned} CPU_{time} \text{ with cache} &= \\ &= IC \times (1.0 + (30/1000 \times 100)) \times \text{Clock cycle time} \\ &= IC \times 4.00 \times \text{Clock cycle time} \end{aligned}$$

Now caculating performance using miss rate:

$$CPU_{time} = IC \times \left(CPI_{execution} + Missrate \times \frac{Mem\ accesses}{Instruction} \times Misspenalty \right) \times Clockcycletime$$

$$\begin{aligned} CPU_{time} \text{ with cache} &= \\ &= IC \times (1.0 + (1.5 \times 2\% \times 100)) \times \text{Clock cycle time} \\ &= IC \times 4.00 \times \text{Clock cycle time} \end{aligned}$$

Answer for example 4 (cont.)

- The clock cycles time and instruction count are the same, with or without a cache. Thus, CPU time increases fourfold, with CPI from 1.00 a “perfect cache” to 4.00 with a cache that can miss.
- Without any memory hierarchy at all the CPI would increase again to $1.0 + 100 \times 1.5$ or 151—factor of almost 40 time longer than a system with a cache.

Cache misses have a double-barreled impact on a CPU

- The lower the $CPI_{\text{execution}}$, the higher the relative impact of a fix number of cache miss clock cycle.
- When calculating CPI, the cache miss penalty is measured in CPU clock cycles for a miss. Therefore, even if memory hierarchies for two computers are identical, the CPU with the higher clock rate has a larger number of clock cycles per miss and hence a higher memory portion of CPI.

Ex5: Impact on Performance

Assume: CPI=2 (perfect cache) clock cycle time=1.0 ns

- MPI(memory reference per instruction)=1.5
- Size of both caches is 64K and size of bath block is 64 bytes
- One cache is direct mapped and other is two-way set associative. the former has miss rate of 1.4%, the latter has miss rate 1.0%
- The selection multiplexor forces CPU clock cycle time to be stretched 1.25 times
- Miss penalty is 75ns, and hit time is 1 clock cycle

■ What is the impact of two diffect cache organizations on performance of CPU (first, calculate the average memory access time and then CPU performance.)?

Answer for example 5

Answer : Average memory access time is

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

Thus, the time for each organization is

Average memory access time_{1-way} = $1.0 + (0.014 \times 75) = 2.05$ ns

Average memory access time_{2-way} = $1.0 \times 1.25 + (0.01 \times 75)$
= 2.00 ns

The average memory access time is better for the 2-way set-associative cache.

Answer for example 5 (cont.)

CPU performance is

$$\begin{aligned} CPUtime &= IC \times \left(CPI_{execution} + \frac{Misses}{Instruction} \times Misspenalty \right) \times Clockcycletime \\ &= IC \times \left[\left(CPI_{execution} \times Clockcycletime \right) \right. \\ &\quad \left. + \left(Missrate \times \frac{Memoryaccesses}{Instruction} \times Misspenalty \times Clockcycletime \right) \right] \end{aligned}$$

Substituting 75 ns for (miss penalty \times Clock cycle time), the performance of each cache organization is

$$CPU\ time_{1-way} = IC \times (2 \times 1.0 + (1.5 \times 0.014 \times 75)) = 3.58 \times IC$$

$$CPU\ time_{2-way} = IC \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.010 \times 75)) = 3.63 \times IC$$

Answer for example 5 (cont.)

Relative performance is

$$\frac{CPU\ time_{2-way}}{CPU\ time_{1-way}} = \frac{3.63 \times Instruction\ count}{3.58 \times Instruction\ count} = \frac{3.63}{3.58} = 1.01$$

In contrast to the results of average memory access time, the direct-mapped leads to slightly better average performance. **Since CPU time is our bottom-line evaluation.**

Miss penalty and Out-of-order Execution Processors

How do you define “miss penalty”?

- Is it the full latency of the miss to memory, or is it just the “exposed” or nonoverlapped latency when the processor must stall?
- To In-order processor, there is out of question, but here is out of the question.
- Refine memory stalls to lead to a new definition of miss penalty as nonoverlapped latency :

$$\frac{\text{Memory stall cycles}}{\text{instruction}} = \frac{\text{Misses}}{\text{instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

Two definition

- **Length of memory latency** -What to consider as the start and the end of a memory operation in an out-of-order processor.
- **Length of latency overlapped**—What is the start of overlap with the processor(or equivalently, when do we say a memory operation is stalling the processor)

Ex6: Performance on out-of-order processor

Assume: 30% of the 75 ns (52.5) miss penalty can be overlapped; Another parameters are same with example 5 (above example)

What is the impact of performance for out-of-order (OOO) CPU in direct-mapped cache?

Answer: Average memory access time for the OOO computer is:

$$\begin{aligned}\text{Average memory access time}_{1\text{-way},\text{OOO}} &= 1.0 * 1.25 + (0.014 \times 52.5) \\ &= 1.99 \text{ ns}\end{aligned}$$

The performance of the OOO cache is:

$$\begin{aligned}\text{CPU time}_{1\text{-way},\text{OOO}} &= IC \times (2 \times 1.0 * 1.25 + (1.5 \times 0.014 \times 52.5)) \\ &= 3.60 \times IC\end{aligned}$$

Hence, despite a much slower clock cycle time and the higher miss rate of a direct-mapped cache, the OOO computer can be slightly faster if it can hide 30% of the miss penalty.

How to Improve Cache Performance?

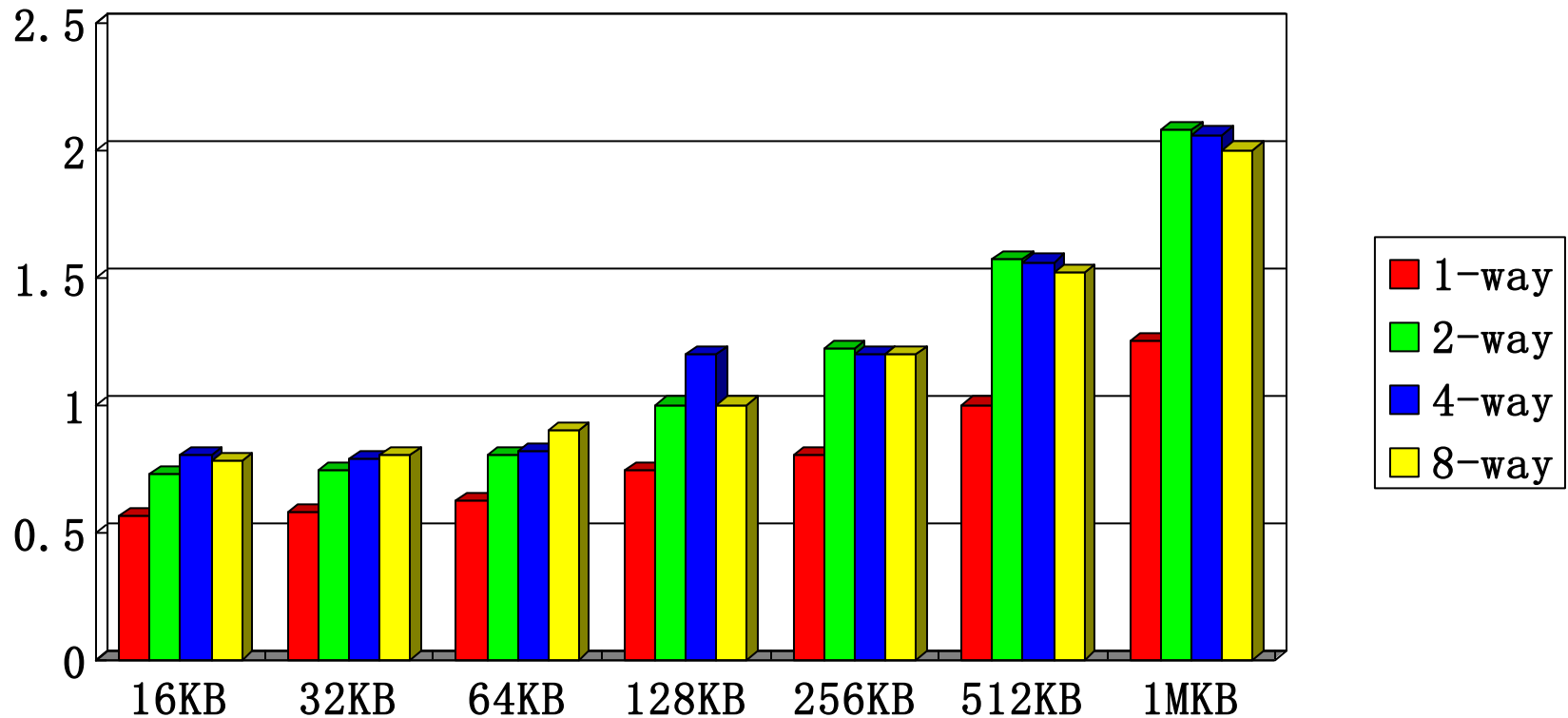
$$AMAT = HitTime + MissRate \times MissPenalty$$

1. Reduce the time to hit in the cache.--4
 - small and simple caches, avoiding address translation, way prediction, and trace caches
2. Increase cache bandwidth.--3
 - pipelined cache access, multibanked caches, non-blocking caches,
3. Reduce the miss penalty--4
 - multilevel caches, critical word first, read miss prior to writes, merging write buffers, and victim caches
4. Reduce the miss rate--4
 - larger block size, large cache size, higher associativity, and compiler optimizations
5. Reduce the miss penalty and miss rate via parallelism--2
 - hardware prefetching, and compiler prefetching

1st Hit Time Reduction Technique: Small and Simple Caches

- Using small and Direct-mapped cache
- The less hardware that is necessary to implement a cache, the shorter the critical path through the hardware.
 - **Direct-mapped** is faster than set associative for both reads and writes.
 - Fitting the cache **on the chip with the CPU** is also very important for fast access times.

Hit time varies with size and associativity

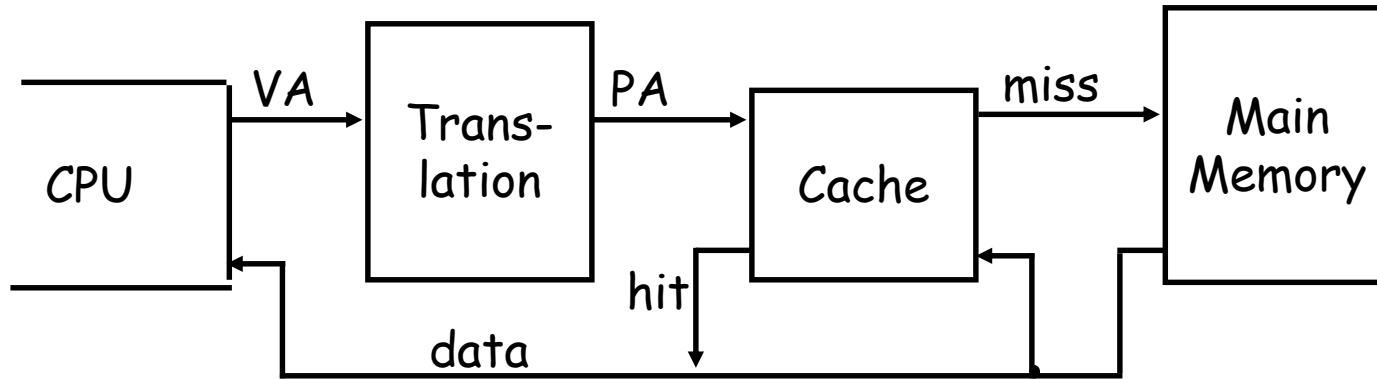


2nd Hit Time Reduction Technique: Way Prediction

- Way Prediction (Pentium 4)
 - Extra bits are kept in the cache to predict the way, or block within set of the *next* cache access.
 - If the predictor is correct, the instruction cache latency is 1 clock cycle.
 - If not, it tries the other block, changes the way predictor, and has a latency of 1 extra clock cycle.
 - Simulation using SPEC95 suggested set prediction accuracy is excess of 85%, so way prediction saves pipeline stage in more than 85% of the instruction fetches.

3rd Hit Time Reduction Technique:

Avoiding Address Translation during Indexing of the Cache



- Page table is a large data structure in memory
- TWO memory accesses for every load, store, or instruction fetch!!!
- Virtually addressed cache?
 - synonym problem

Cache the address translations?

TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

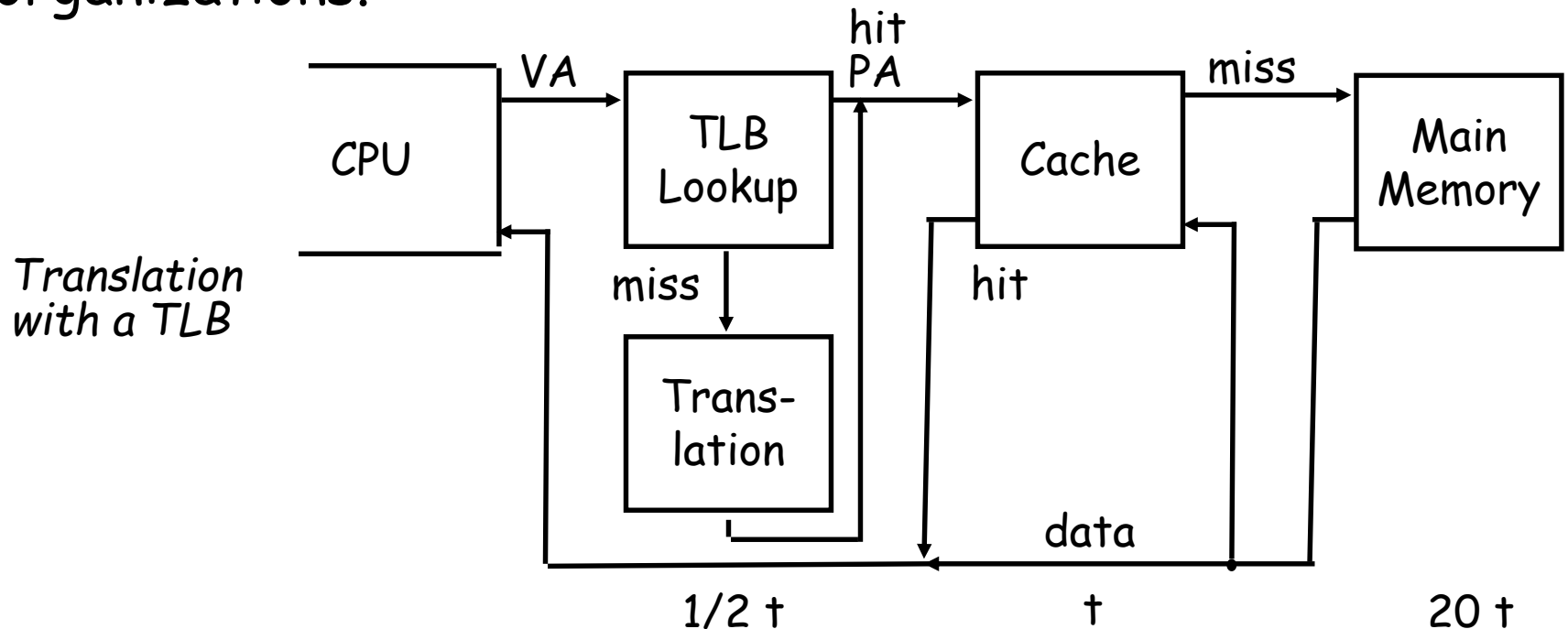
Really just a cache on the page table mappings

TLB access time comparable to cache access time
(much less than main memory access time)

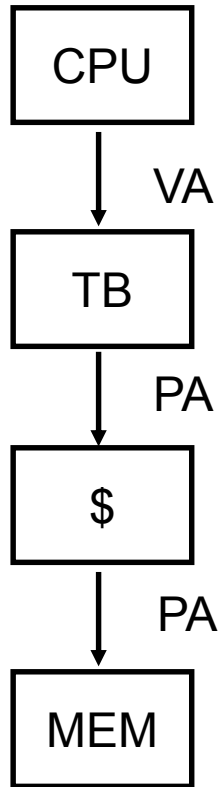
Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

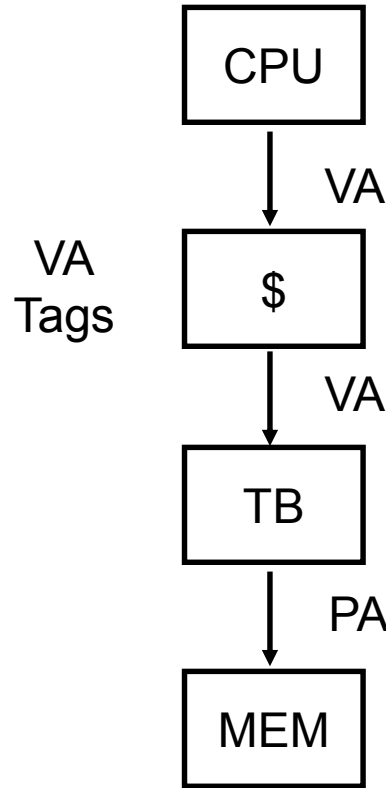
TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



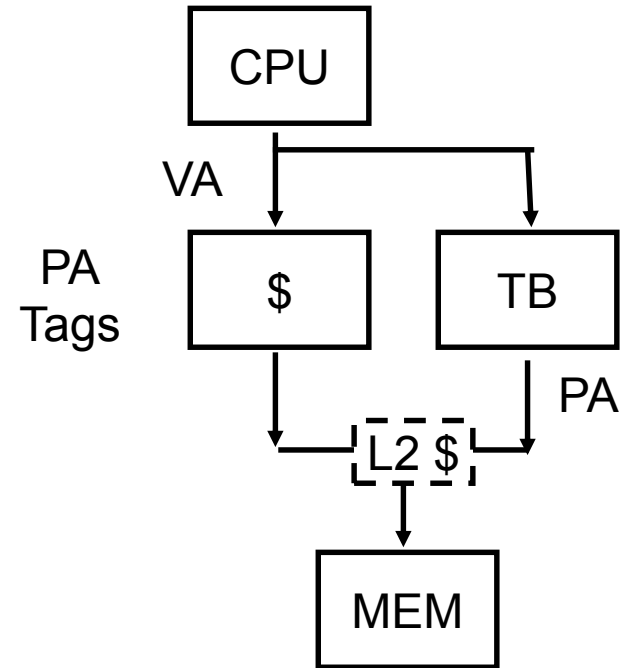
Fast hits by Avoiding Address Translation



Conventional Organization



Virtually Addressed Cache
Translate only on miss
Synonym Problem

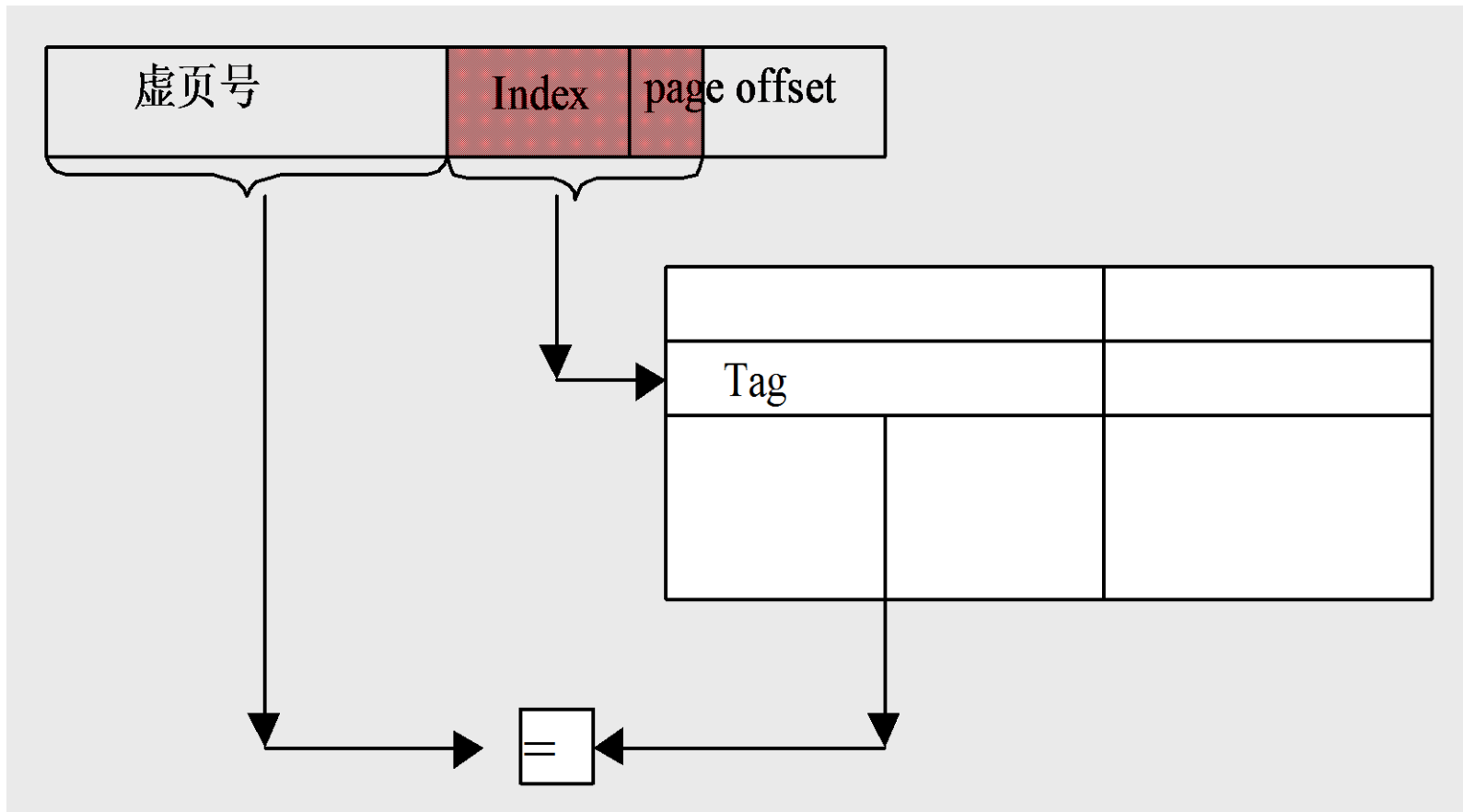


Virtual indexed, Physically tagged
Overlap \$ access
with VA translation:
requires \$ index to
remain invariant
across translation

Virtual Addressed Cache

- Send virtual address to cache? Any Problems ?
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
 - Add process identifier tag that identifies process as well as address within process: can't get a hit if wrong process

Virtual cache

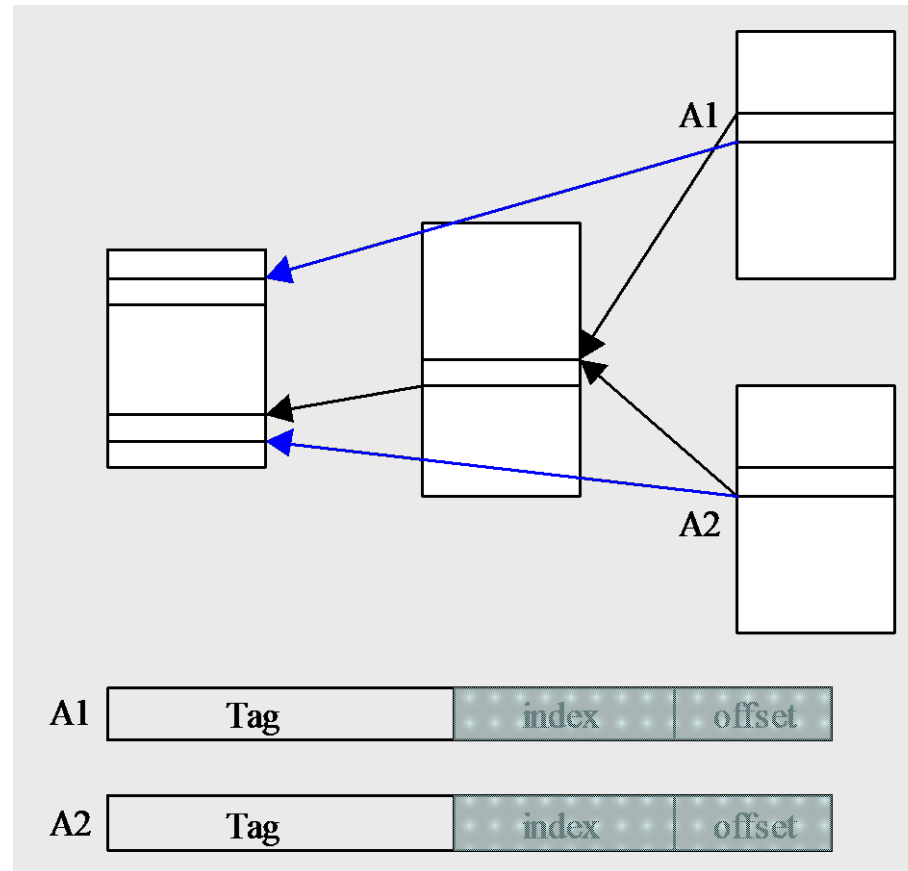


Dealing with aliases

- Dealing with aliases (synonyms); Two different virtual addresses map to same physical address
 - NO aliasing! What are the implications?
 - HW antialiasing: guarantees every cache block has unique address
 - verify on miss (rather than on every hit)
 - cache set size \leq page size ?
 - what if it gets larger?
 - How can SW simplify the problem? (called page coloring)
- I/O must interact with cache, so need virtual address

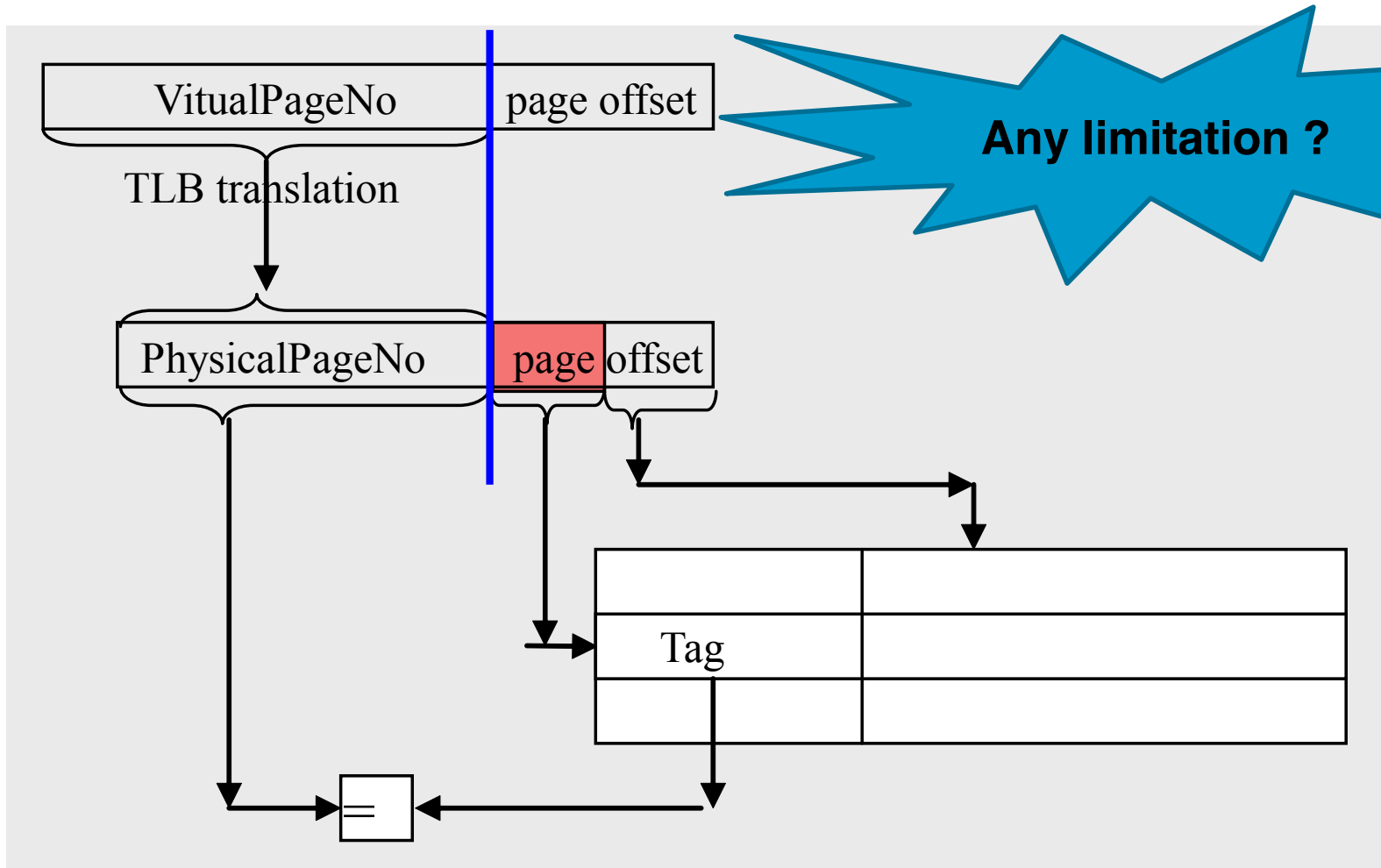


Aliases problem with Virtual cache

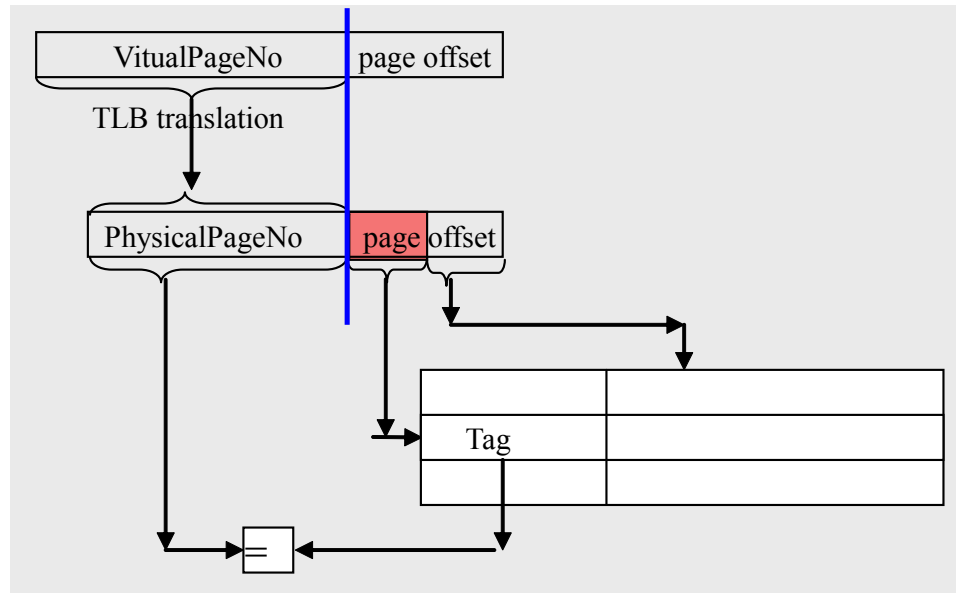


If the index and offset bits of two aliases are forced to be **the same**, then the aliases address will map to the same block in cache.

Overlap address translation and cache access (Virtual indexed, physically tagged)



What's the limitation?



IF it's **direct map cache**, then

$$\text{Cache size} = 2^{\text{index}} * 2^{\text{blockoffset}} \leq 2^{\text{pageoffset}}$$

How to solve this problem?

Use higher association. Say it's a 4 way, then cache size can reach 4 times $2^{\text{pageoffset}}$ with change nothing in index or tag or offset.

4th Hit Time Reduction Technique:

Trace caches

- Find a dynamic sequence of instructions including taken branches to load into a cache block.
- The block determined by CPU instead of by memory layout.
- Complicated address mapping mechanism

Cache ?

- Bring N instructions per cycle
 - No I-cache misses
 - No prediction miss
 - No packet breaks !

Because branch in each 5 instruction, so cache can only provide a packet in one cycle.

What's Trace ?

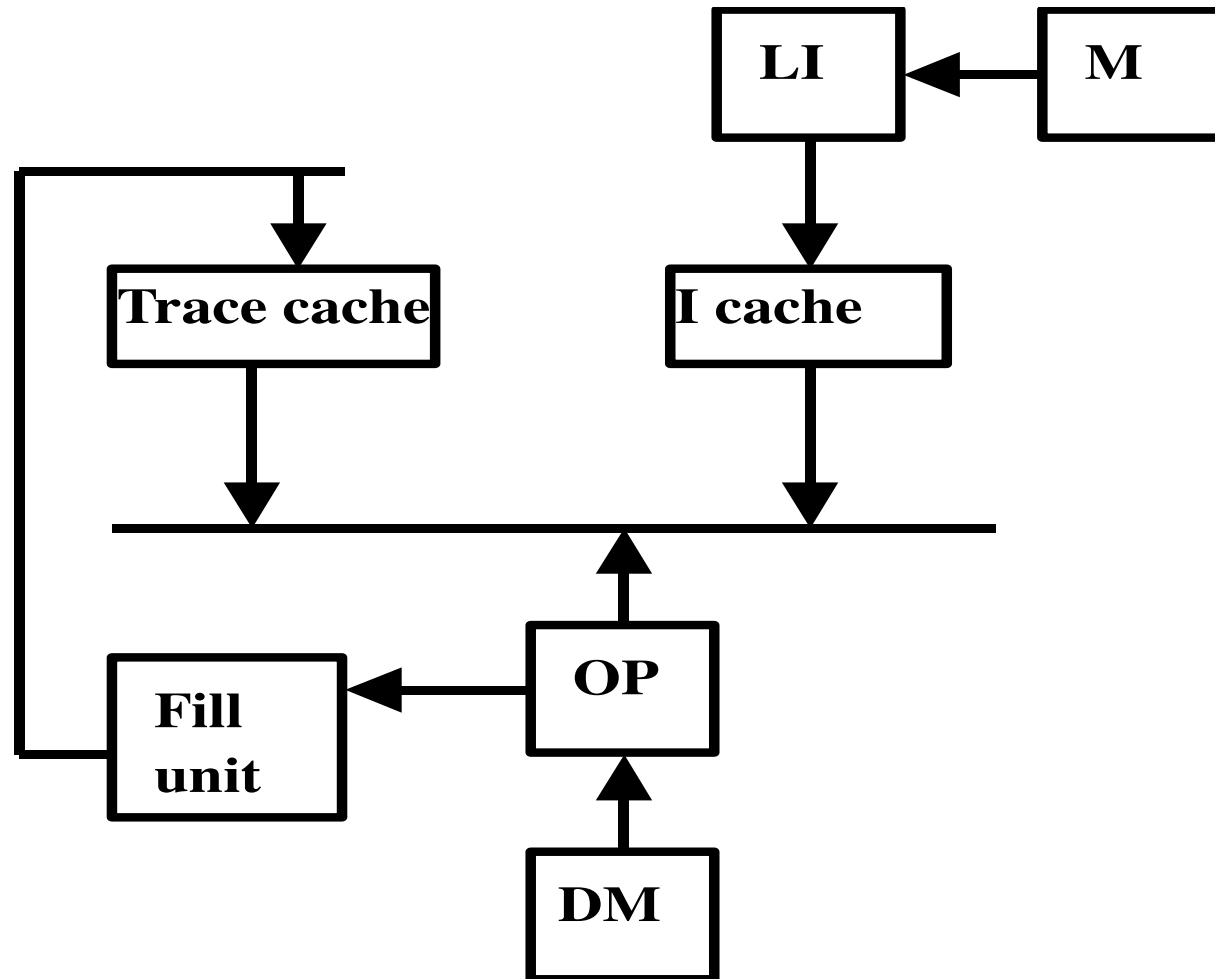
- Trace: dynamic instruction sequence
- When instructions (operations) retire from the pipeline, pack the instruction segments into **TRACE**, and store them in the TRACE cache, including the branch instructions.
- Though branch instruction may go a different target, but **most times** the next operation sequential will just be the same as the last sequential

Whose propose ?

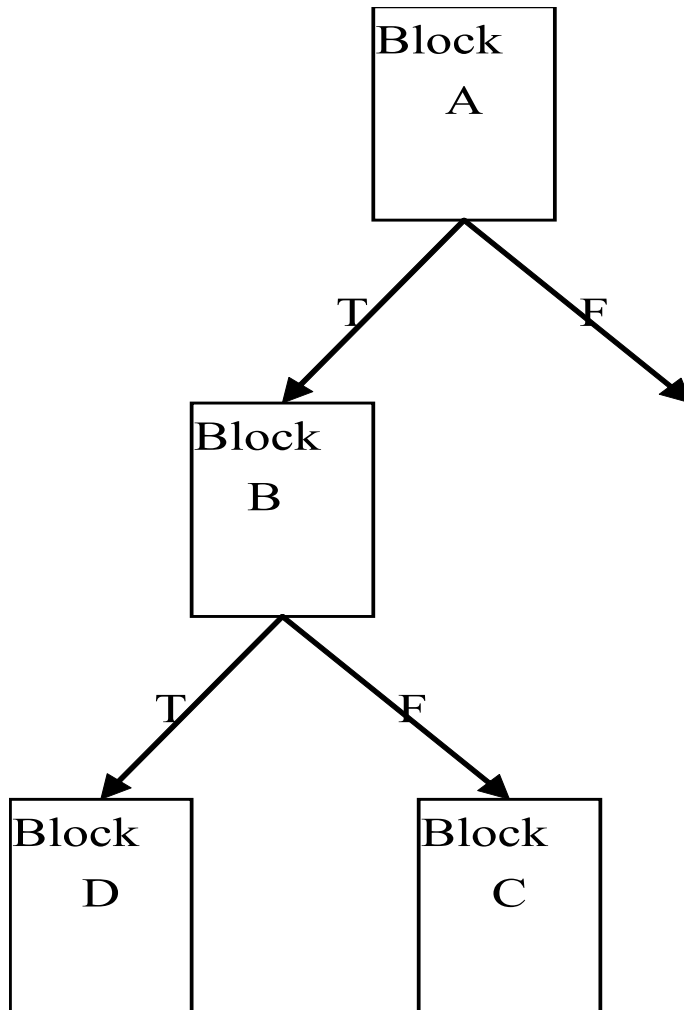
- Peleg Weiser (1994) in Intel corporation
- Patel / Patt (1996)
- Rotenberg / J. Smith (1996)

- Paper: ISCA'98

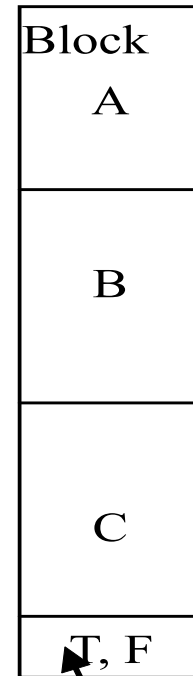
Trace in CPU



Instruction segment

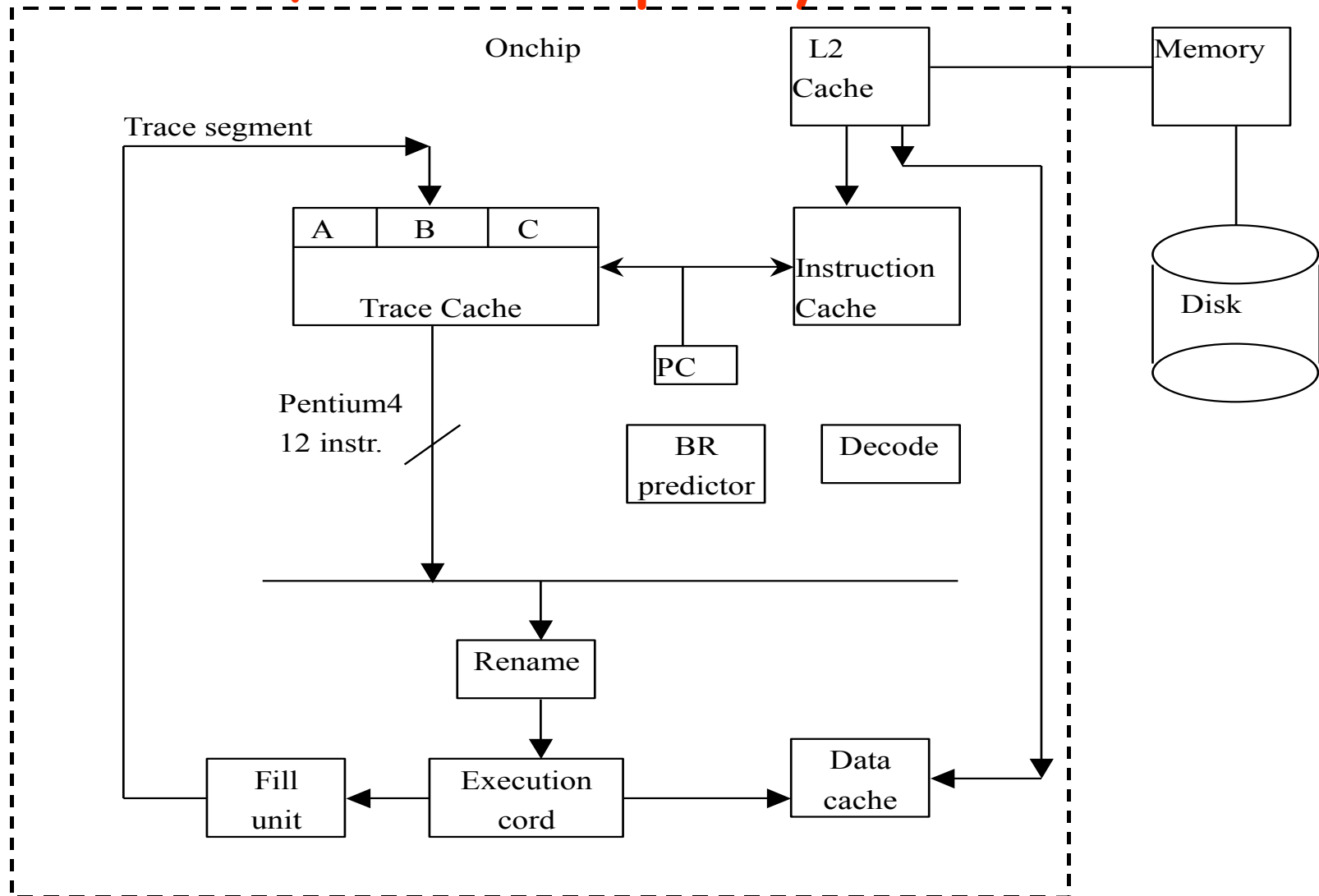


Fill unit pack
them into :



Predict info.

Pentium 4: trace cache, 12 instr./per cycle

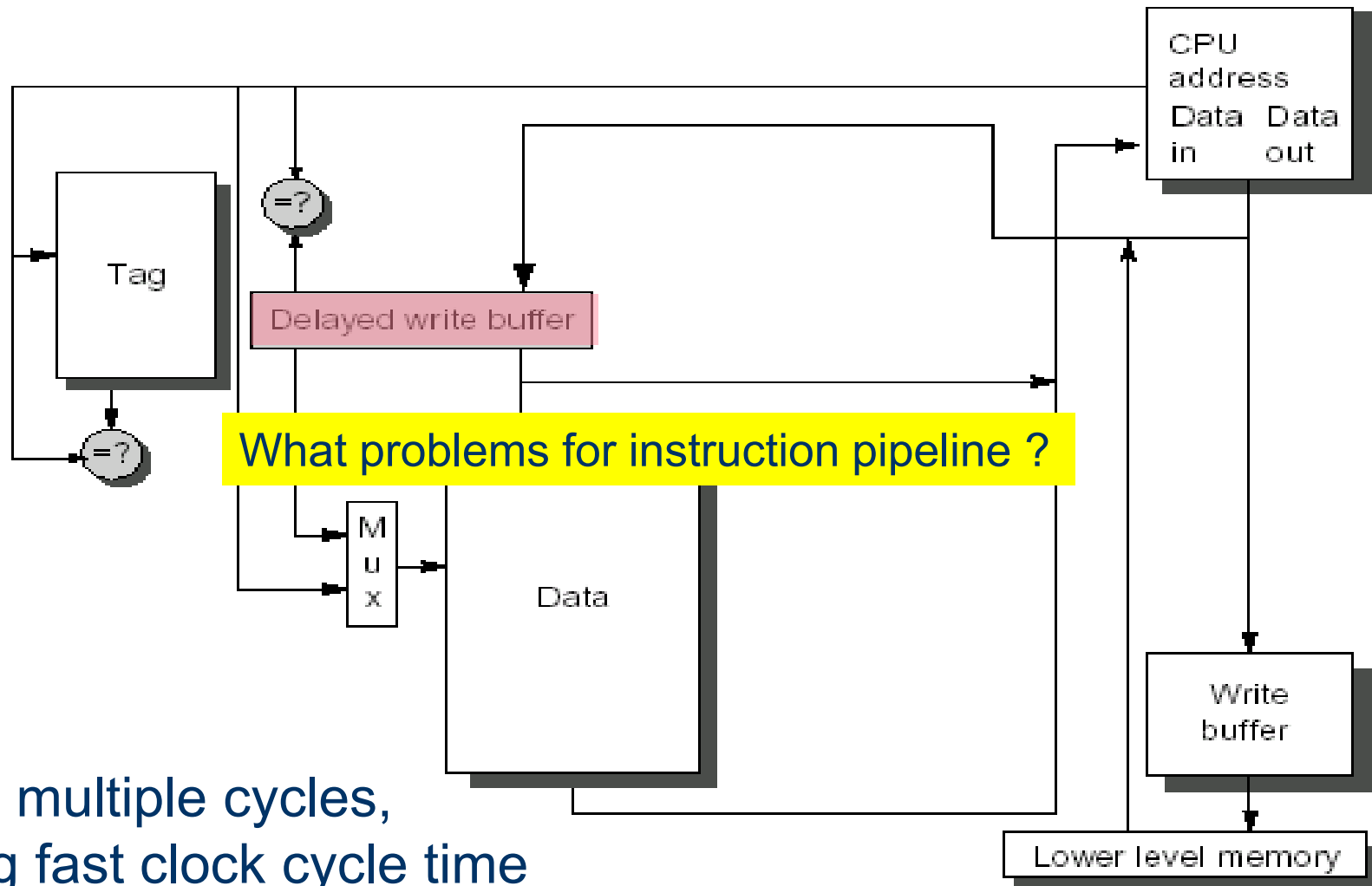


How to Improve Cache Performance?

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

1. Reduce the time to hit in the cache.--4
 - small and simple caches, avoiding address translation, way prediction, and trace caches
2. Increase cache bandwidth.--3
 - pipelined cache access, multibanked caches, non-blocking caches,
3. Reduce the miss penalty--4
 - multilevel caches, critical word first, read miss prior to writes, merging write buffers, and victim caches
4. Reduce the miss rate--4
 - larger block size, large cache size, higher associativity, and compiler optimizations
5. Reduce the miss penalty and miss rate via parallelism--2
 - hardware prefetching, and compiler prefetching

1st Increasing cache bandwidth: Pipelined Caches

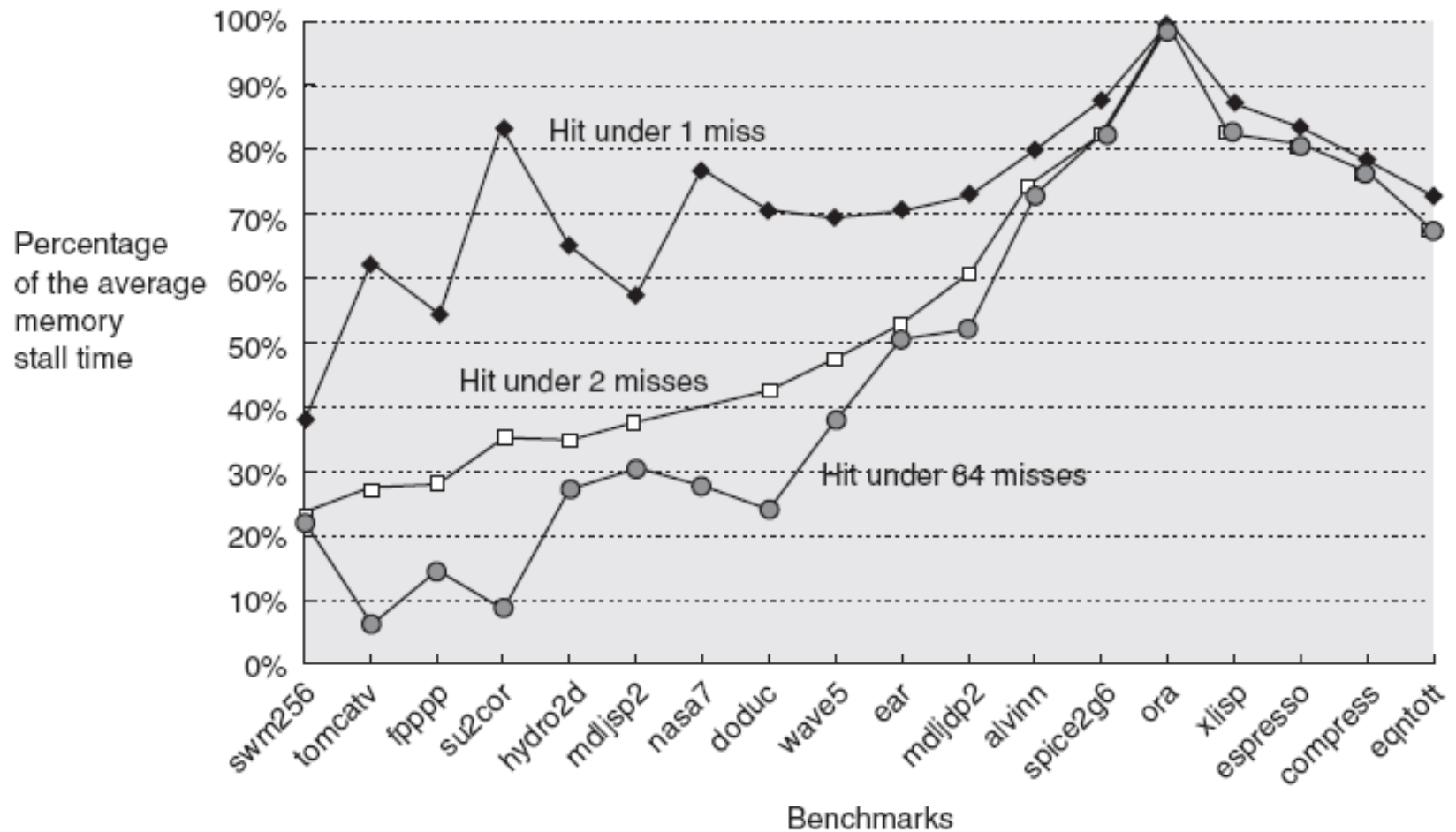


Hit in multiple cycles,
giving fast clock cycle time

2nd Increasing cache bandwidth: Nonblocking Caches

- A **nonblocking** (Lockup-free cache) cache, allows the cache to continue to supply hits while processing read misses (**hit under miss** , **hit under multiple miss**)
- **Complex caches** (e.g., **What's the precondition ?**) can handle multiple outstanding misses (**miss under miss**). It will further lower effective miss penalty
- **Nonblocking**, in conjunction with out-of-order execution, can allow the CPU to continue executing instructions after a data cache miss.

Performance of Nonblocking cache



3rd Increasing cache bandwidth:

Multibanked Caches

- Cache is divided into independent banks that can support simultaneous accesses like interleaved memory banks.
 - E.g., T1 ("Niagara") L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks \Rightarrow mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is "sequential interleaving"

0
1
2
3
⋮
2^N-1

(a)

Single banked

0
1
⋮
$2^{N-1} - 1$

2^{N-1}
$2^{N-1} + 1$
\vdots
$2^N - 1$

(b)

two bank
consecutive

0
2
4
⋮
$2^N - 2$

1
3
5
⋮
$2^N - 1$

(c)

two bank
interleaving

0
1
4
⋮
$2^N - 4$
$2^N - 3$

2
3
6
⋮
$2^N - 2$
$2^N - 1$

(d)

two bank
group interleaving

Summary: Increase Cache Bandwidth

1. Increase bandwidth via **pipelined cache access**
2. Increase bandwidth via **multibanked caches**,
3. Increase bandwidth via **non-blocking caches**,

How to Improve Cache Performance?

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

1. Reduce the time to hit in the cache.--4
 - small and simple caches, avoiding address translation, way prediction, and trace caches
2. Increase cache bandwidth.--3
 - pipelined cache access, multibanked caches, non-blocking caches,
3. Reduce the miss penalty--4
 - multilevel caches, critical word first, read miss prior to writes, merging write buffers, and victim caches
4. Reduce the miss rate--4
 - larger block size, large cache size, higher associativity, and compiler optimizations
5. Reduce the miss penalty and miss rate via parallelism--2
 - hardware prefetching, compiler prefetching

1st Miss Penalty Reduction Technique: Multilevel Caches

- This method focuses on the interface between the cache and main memory.
- Add an second-level cache between main memory and a small, fast first-level cache, **to make the cache fast and large.**
- **The smaller first-level cache is fast** enough to match the clock cycle time of the fast CPU and to fit on the chip with the CPU, thereby lessening the **hits time.**
- **The second-level cache can be large** enough to capture many memory accesses that would go to main memory, thereby lessening the effective **miss penalty.**

Parameter about Multilevel cache

- The performance of a two-level cache is calculated in a similar way to the performance for a single level cache.
- L2 Equations

So the miss penalty for level 1 is calculated using the hit time, miss rate, and miss penalty for the level 2 cache.

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

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

$$\begin{aligned} \text{Miss rate}_{L1} &= \frac{\text{Misses}_{L1}}{M} \\ \text{Miss rate}_{L2} &= \frac{\text{Misses}_{L2}}{M_{L2}} = \frac{\text{Misses}_{L2}}{M \times \text{Miss rate}_{L1}} \end{aligned}$$

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

Two conceptions for two-level cache

■ Definitions:

- **Local miss rate**— misses in this cache divided by the total number of memory accesses **to this cache** (Miss rate_{L2})
- **Global miss rate**—misses in this cache divided by the total number of memory accesses **generated by the CPU**

Using the terms above, the global miss for the first-level cache is still just Miss rate_{L1} , but for the second-level cache it is :

$$\begin{aligned}\text{Miss rate}_{\text{Global-L2}} &= \frac{\text{Misses}_{L2}}{M} = \frac{M \times \text{Miss rate}_{L1}}{M} \times \frac{\text{Misses}_{L2}}{M \times \text{Miss rate}_{L1}} \\ &= \frac{\text{Misses}_{L1}}{M} \times \frac{\text{Misses}_{L2}}{M \times \text{Miss rate}_{L1}} = \text{Miss rate}_{L1} \times \text{Miss rate}_{L2}\end{aligned}$$



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

- Don't wait for full block to be loaded before restarting CPU
 - 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*
 - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- Generally *useful only in large blocks*,
- Spatial locality => tend to want next sequential word, so not clear if benefit by early restart

Example: Critical Word First

Assume:

cache block=64-byte

L2: take 11 CLK to get the critical 8 bytes,(AMD Athlon)
and then 2 CLK per 8 byte to fetch the rest of the
block

There will be no other accesses to rest of the block

Calculate the average miss penalty for critical word first.

Then assuming the following instructions read data
sequentially 8 bytes at a time from the rest of the block

Compare the times with and without critical word first.

3rd Miss Penalty Reduction Technique:

Giving Priority to Read Misses over Writes

- If a system has a write buffer, writes can be delayed to come after reads.
- The system must, however, be careful to check the write buffer to see if the value being read is about to be written.

Write buffer

- **Write-back** want buffer to hold displaced 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
- **Write-through** want write buffers => RAW conflicts with main memory reads on cache misses
 - 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 the memory access continue

4th Miss Penalty Reduction Technique: Merging write Buffer

- One word writes replaces with multiword writes, and it improves buffers's efficiency.
- **In write-through** ,When write misses if the buffer contains other modified blocks,the addresses can be checked to see if the address of this new data matches the address of a valid write buffer entry.If so,**the new data are combined with that entry.**

Write merging

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	

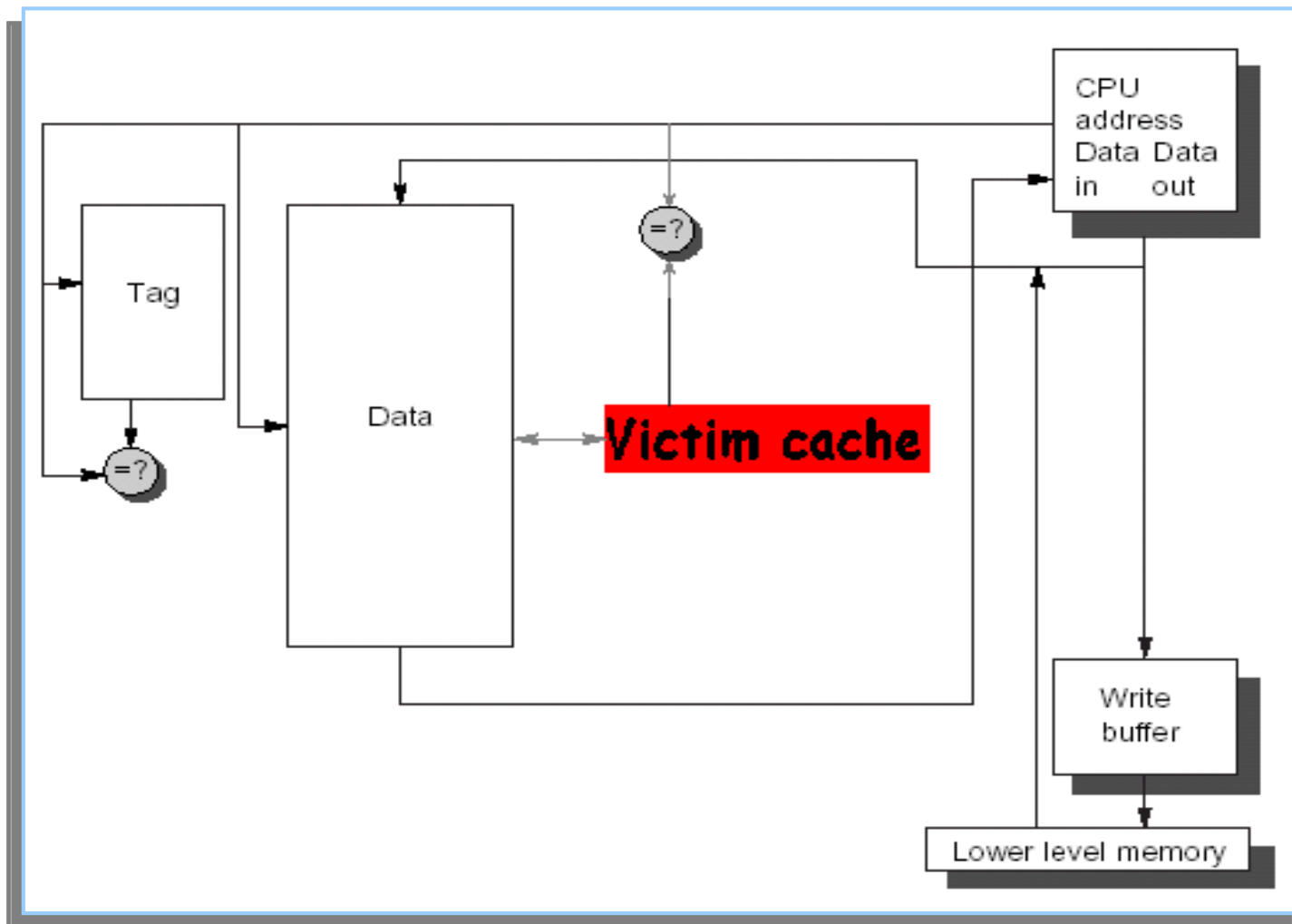
Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Miss Penalty Reduction Technique:

Victim Caches

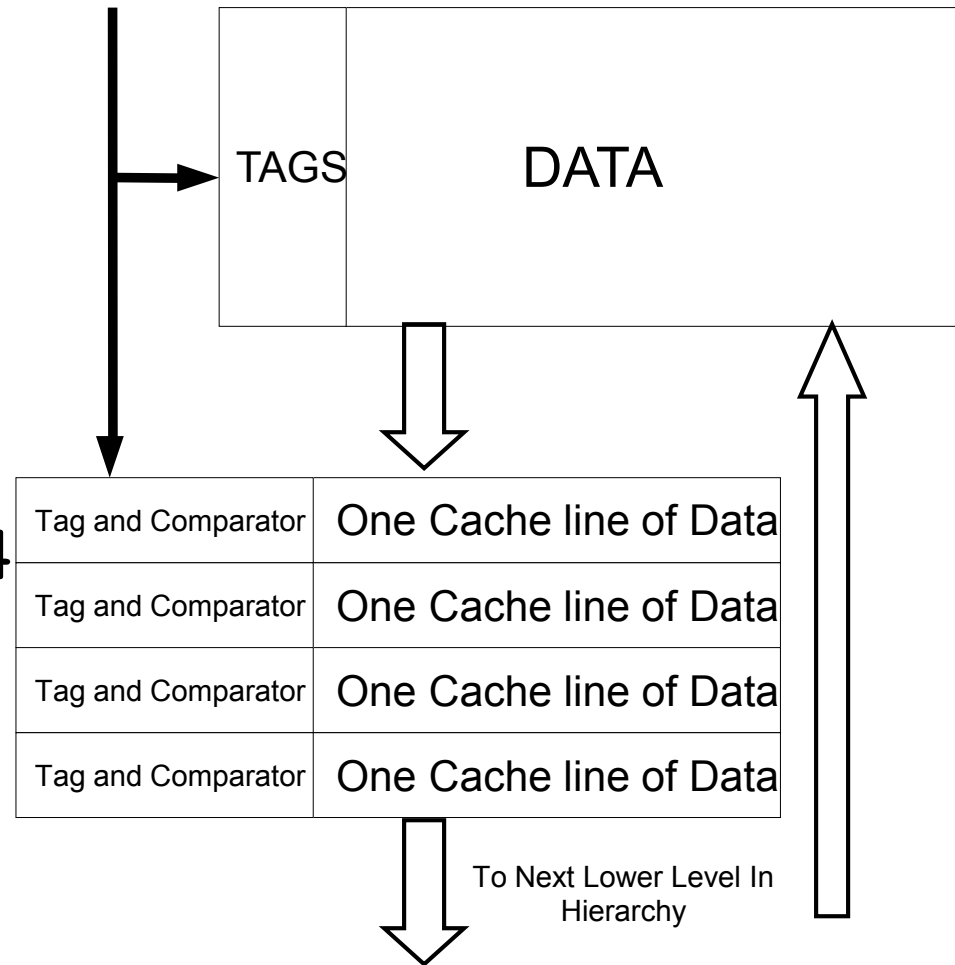
- A **victim cache** is a small (usually, but not necessarily) fully-associative cache that holds a few of the most recently replaced blocks or victims from the main cache.
- This cache is checked on a miss data before going to next lower-level memory(main memory).
 - to see if they have the desired item
 - If found, the victim block and the cache block are swapped.
 - The AMD Athlon has a victim caches (write buffer for write back blocks) with 8 entries.

The Victim Cache



How to combine victim Cache ?

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines



Summary: Miss Penalty Reduction

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

1. Reduce penalty via **Multilevel Caches**
2. Reduce penalty via **Critical Word First**
3. Reduce penalty via **Read Misses over Writes**
4. Reducing penalty via **Merging write Buffer**

How to Improve Cache Performance?

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

1. Reduce the time to hit in the cache.--4
 - small and simple caches, avoiding address translation, way prediction, and trace caches
2. Increase cache bandwidth.--3
 - pipelined cache access, multibanked caches, non-blocking caches,
3. Reduce the miss penalty--4
 - multilevel caches, critical word first, read miss prior to writes, merging write buffers, and victim caches
4. Reduce the miss rate--4
 - larger block size, large cache size, higher associativity, and compiler optimizations
5. Reduce the miss penalty and miss rate via parallelism--2
 - hardware prefetching, and compiler prefetching

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.

(Misses in Fully Associative Size X Cache)

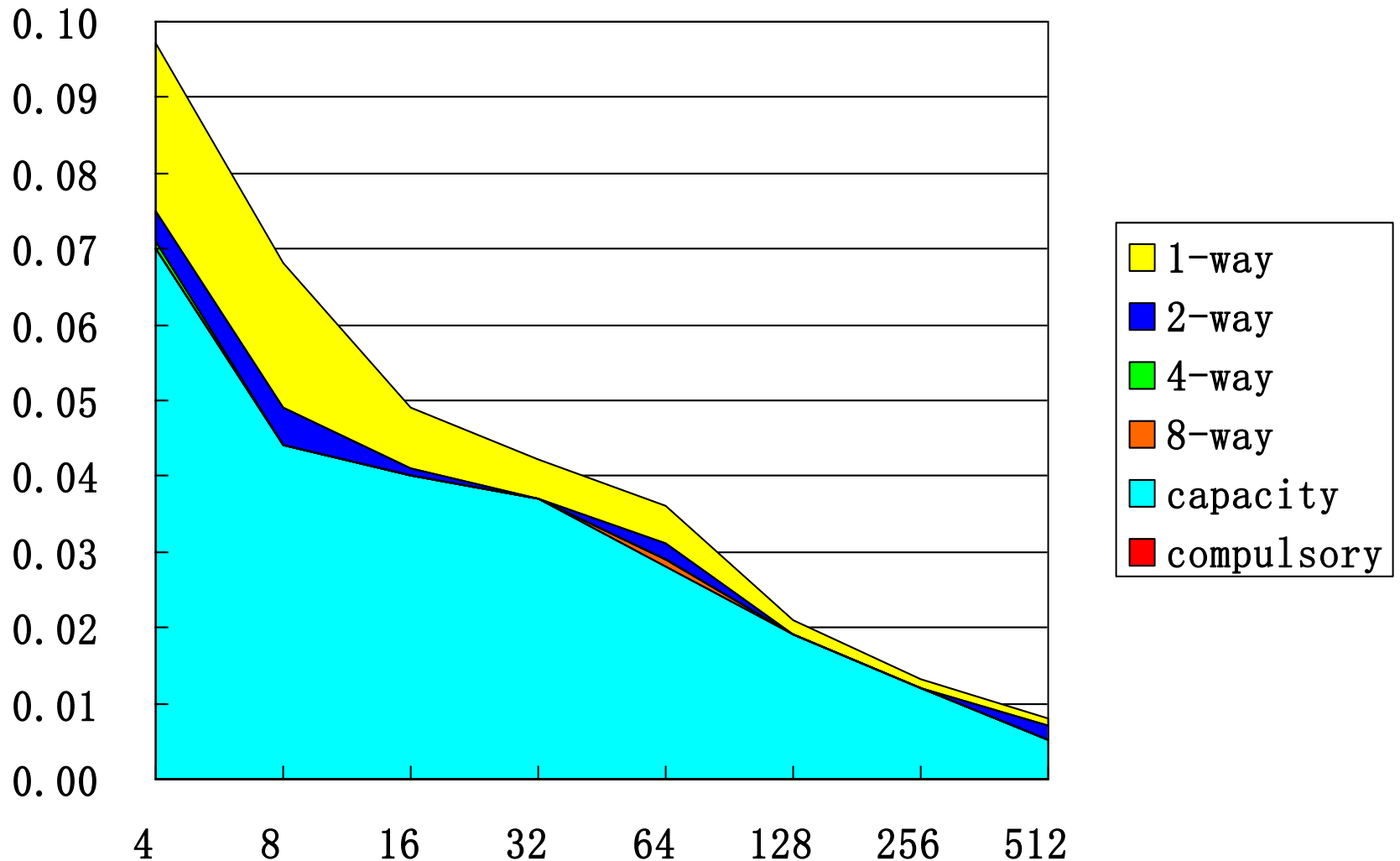
- **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.

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

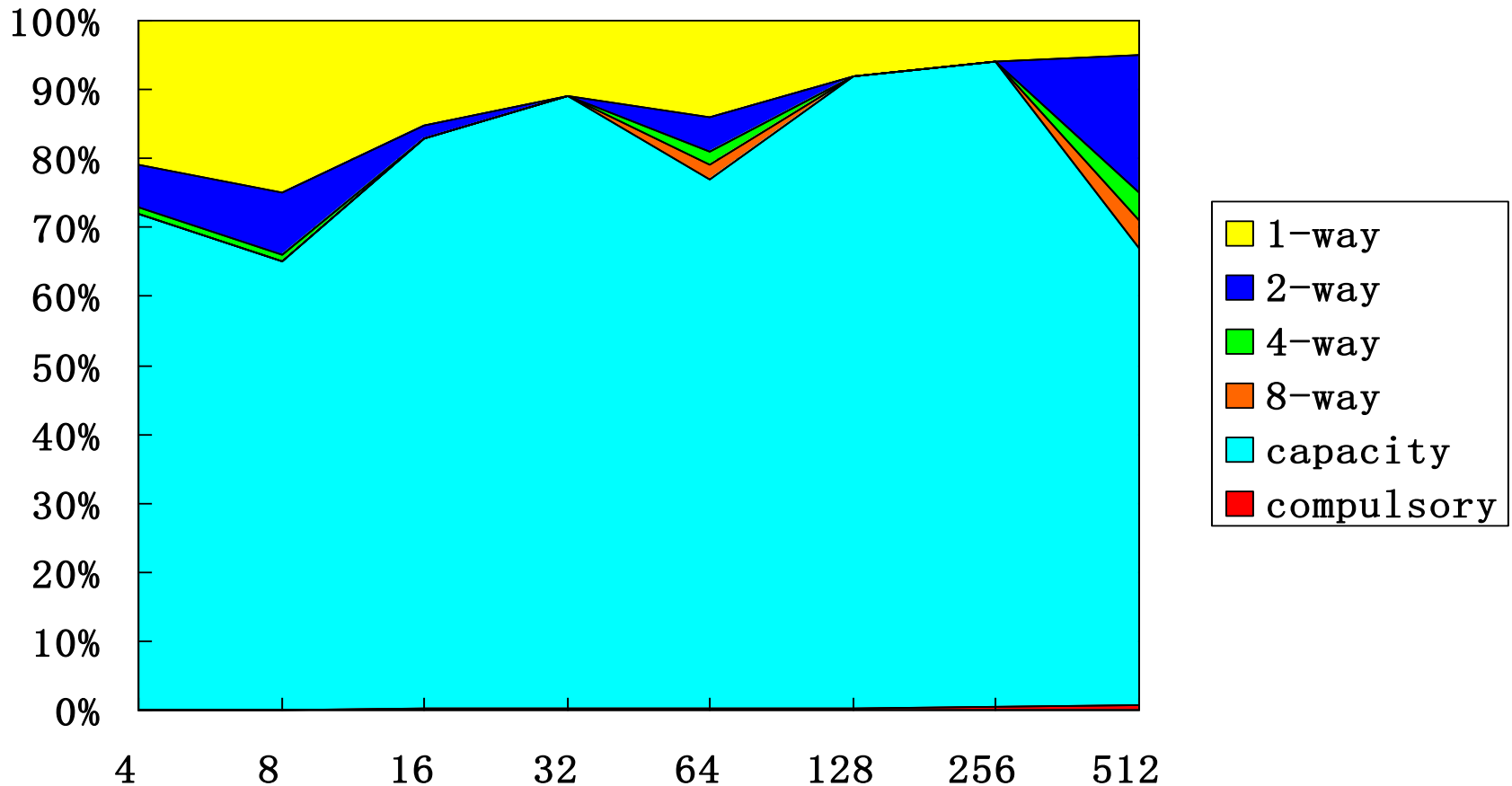
■ 4th "C":

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

3Cs Absolute Miss Rate (SPEC92)



3Cs Relative Miss Rate



Flaws: for fixed block size
Good: insight => invention

Reducing Cache Miss Rate

- To reduce cache miss rate, we have to eliminate some of the misses due to the three C's.
- We cannot reduce capacity misses much except by making the cache larger.
- We can, however, reduce the conflict misses and compulsory misses in several ways:

Cache Organization?

- Assume total cache size not changed:
- What happens if:

1) Change Block Size:

2) Change Associativity:

3) Change Compiler:

Which of 3Cs is obviously affected?

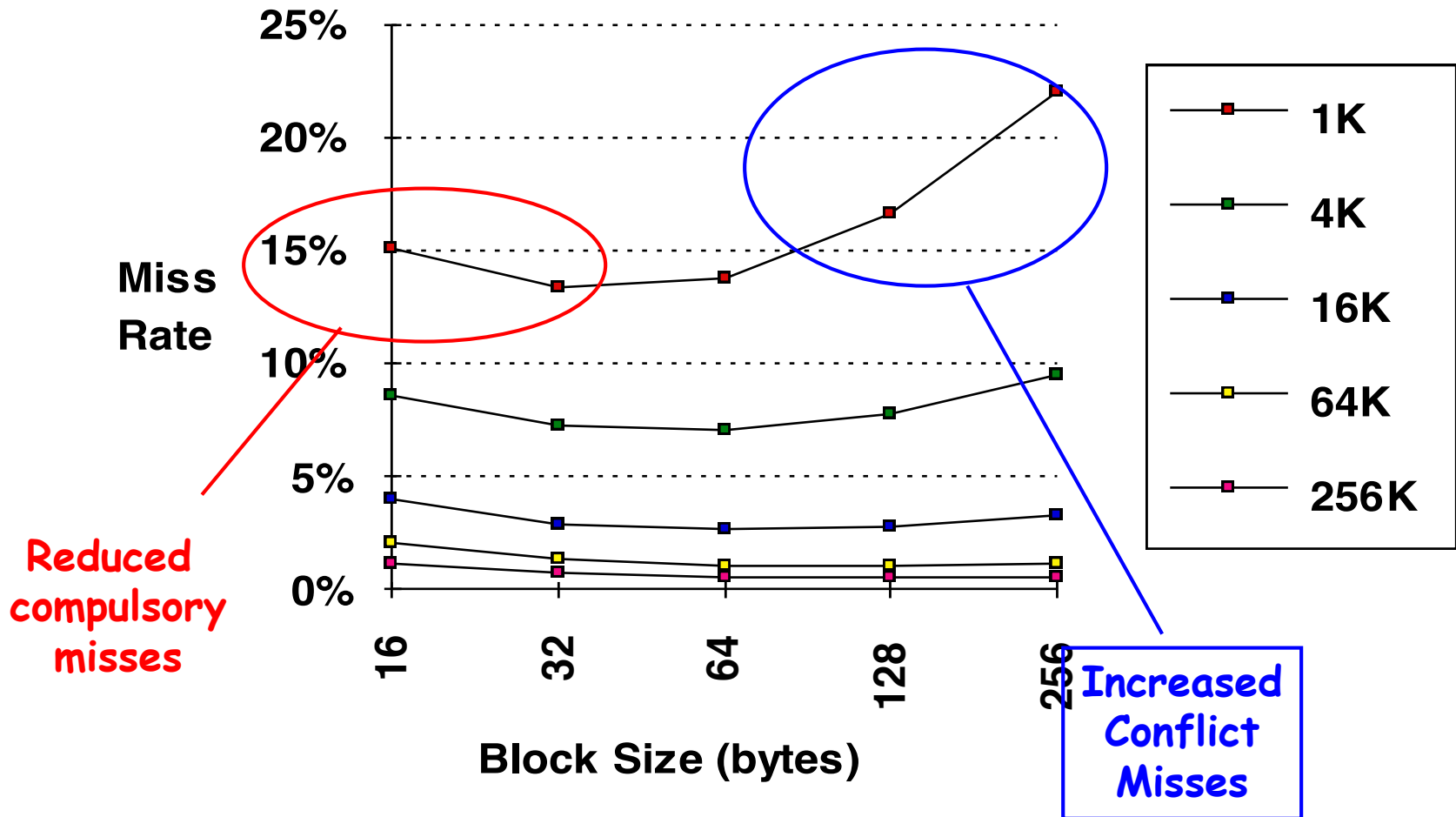
1st Miss Rate Reduction Technique: Larger Block Size (fixed size&assoc)

- Larger blocks decrease the compulsory miss rate by taking advantage of spatial locality.
- Drawback--curve is U-shaped
 - However, they may increase the miss penalty by requiring more data to be fetched per miss.
 - In addition, they will almost certainly increase conflict misses since fewer blocks can be stored in the cache.
 - And maybe even capacity misses in small caches
- Trade-off
 - Trying to minimize both the miss rate and the miss penalty.
 - The selection of block size depends on both the latency and bandwidth of lower-level memory

Miss Rate relates Block size

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

Performance curve is U-shaped



What else drives up block size?

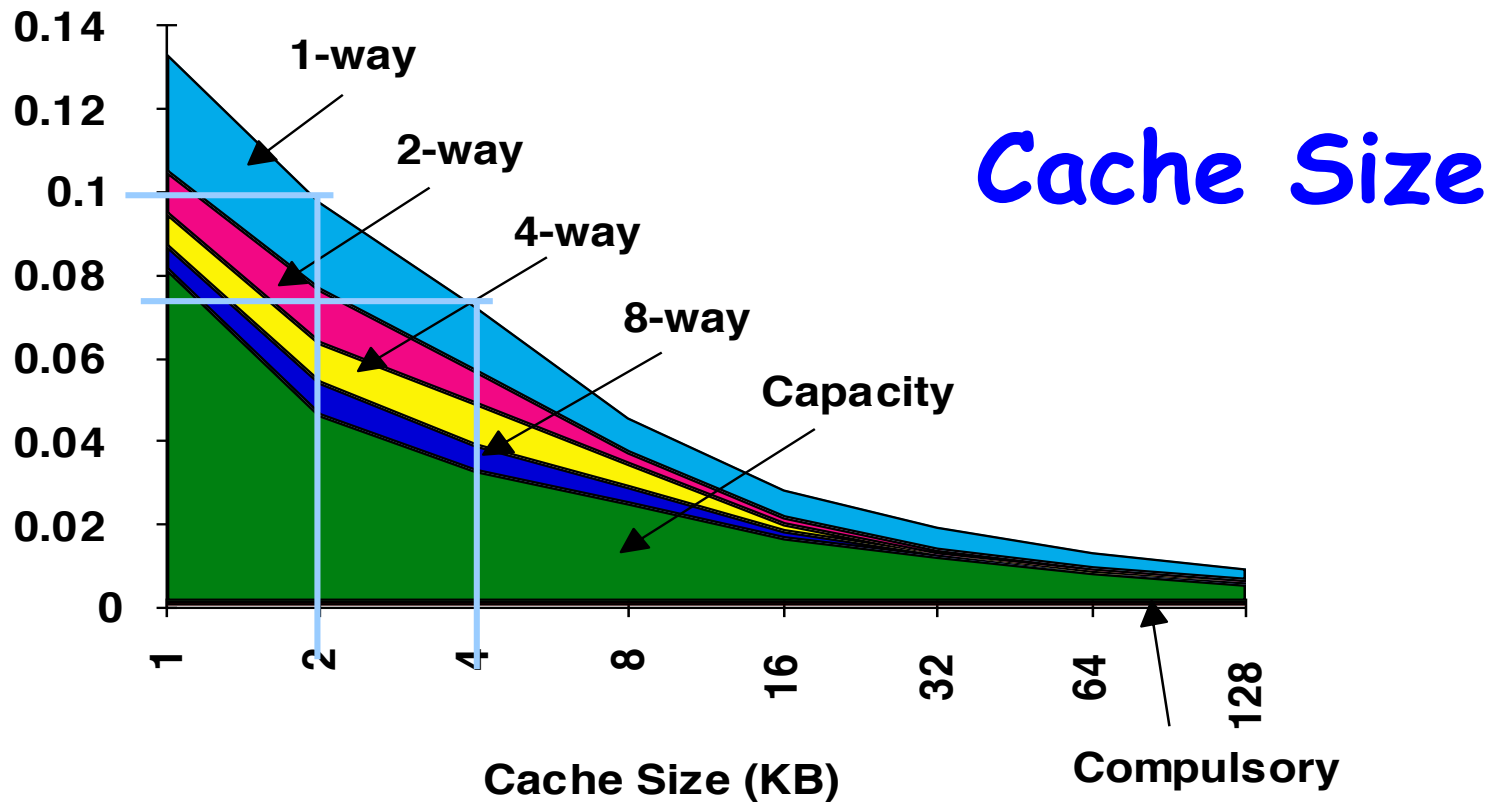
Example: Larger Block Size (C-26)

- Assume: memory takes 80 clock cycles of overhead and then delivers 16 bytes every 2 cycles.
- 1 clock cycle hit time independent of block size.
- Which block size has the smallest AMAT for each size in Fig.5.17 ?
- Answer:

$$AMAT_{16\text{-byte block, 4KB}} = 1 + (8.57\% * 82) = 8.027$$

$$AMAT_{256\text{-byte block, 256KB}} = 1 + (0.49\% * 112) = 1.549$$

2nd Miss Rate Reduction Technique: Larger Caches



- rule of thumb: $2 \times \text{size} \Rightarrow 25\% \text{ cut in miss rate}$
- What does it reduce ?

Pro. Vs. cons for large caches

■ Pro.

- Reduce capacity misses

■ Con.

- Longer hit time, Higher cost, AMAT curve is U-shaped

■ Popular in off-chip caches

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

3rd Miss Rate Reduction Technique: Higher Associativity

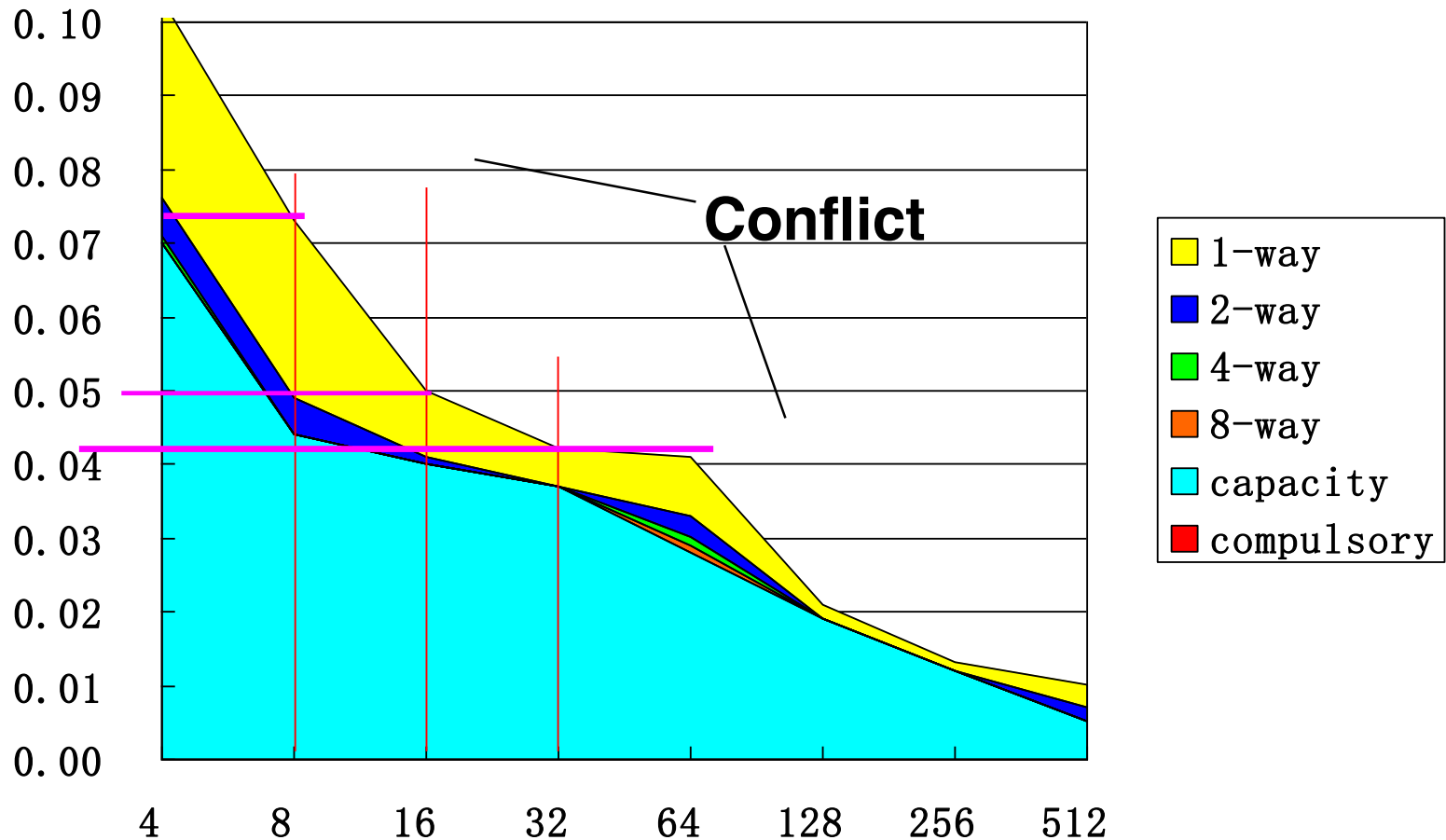
- Conflict misses can be a problem for caches with low associativity (especially direct-mapped).
- With higher associativity decreasing Conflict misses to improve miss rate

cache rule of thumb

- 2:1 rule of thumb a direct-mapped cache of size N has the same miss rate as a 2-way set-associative cache of size $N/2$.
- Eight-way set associative is for practical purposes as effective in reducing misses for these sized cache as fully associative.

Associativity

2:1 rule of thumb



Associativity vs Cycle Time

- Beware: Execution time is only final measure!
- Why is cycle time tied to hit time?
- Will Clock Cycle time increase?
 - Hill [1988] suggested hit time for 2-way vs. 1-way
external cache +10%,
internal + 2%
 - suggested big and dumb caches

Avg. Memory Access Time vs. Miss Rate (P430)

- Example: assume CCT = 1.36 for 2-way, 1.44 for 4-way, 1.52 for 8-way vs. CCT direct mapped

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	3.28
8	2.69	2.58	2.55	2.62
16	2.33	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.24	2.18	2.25
128	1.52	1.84	1.92	2.00
256	1.32	1.66	1.74	1.82
512	1.20	1.55	1.59	1.66

(Red means A.M.A.T. not improved by more associativity)

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

Using two Technique reduces conflict misses and yet maintains hit speed of direct-mapped cache

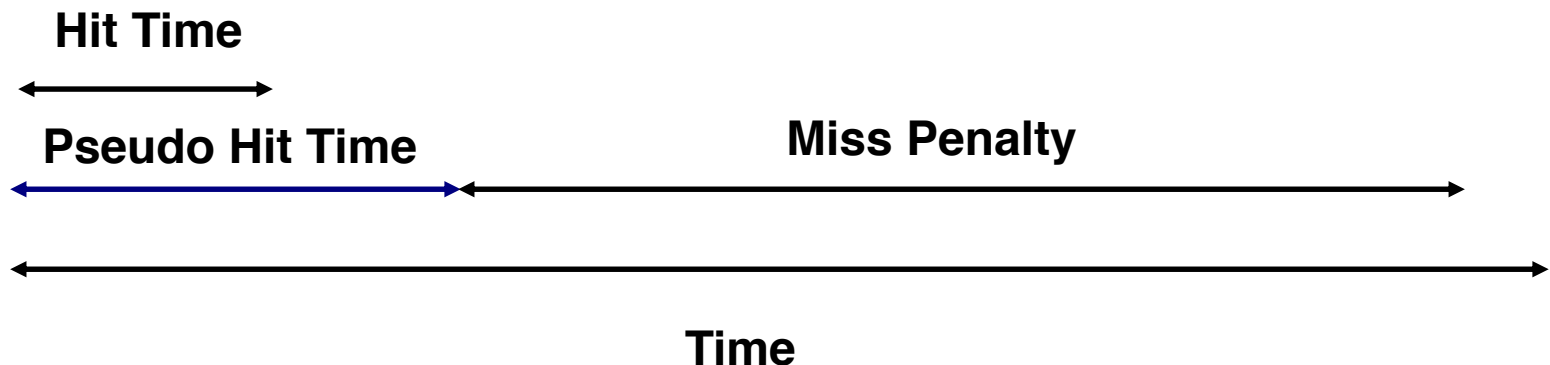
- Predictive bit
- Pseudo-Associative

■ Way Prediction (Alpha 21264)

- Extra bits are kept in the cache to predict the way, or block within set of the *next* cache access.
- If the predictor is correct, the instruction cache latency is 1 clock cycle.
- If not, it tries the other block, changes the way predictor, and has a latency of 3 clock cycles.
- Simulation using SPEC95 suggested set prediction accuracy is excess of 85%, so way prediction saves pipeline stage in more than 85% of the instruction fetches.

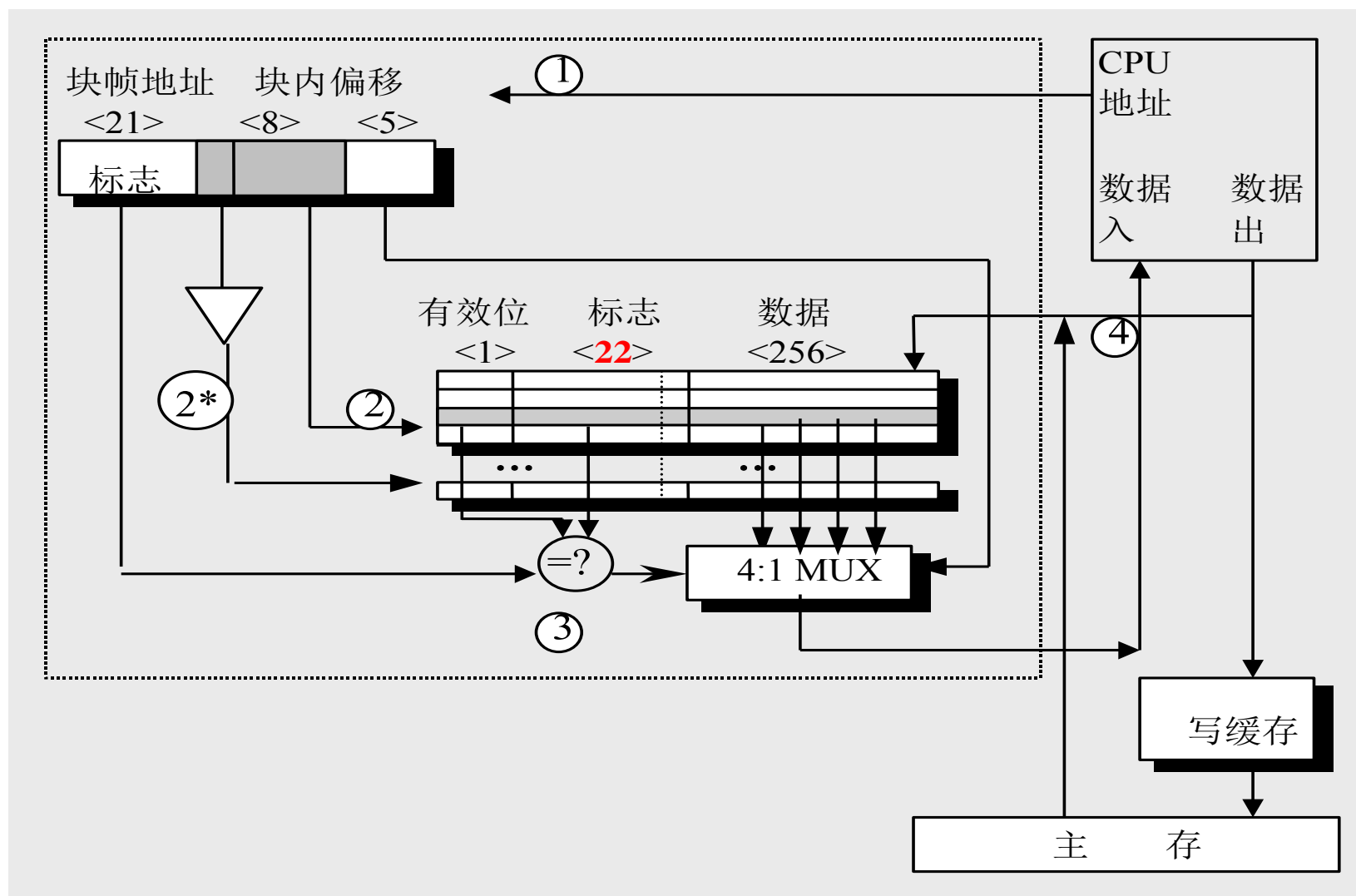
Pseudo-Associative Cache (column associative)

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



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

Pseudo-Associative Cache



4th Miss Rate Reduction Technique: Compiler Optimizations

- The techniques reduces miss rates *without* any hardware changes and reorders instruction sequence with compiler.
- **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

a. Merging Arrays

- Combining independent matrices into a single compound array.
- Improving spatial locality
- Example

```
/*before*/
```

```
Int val[SIZE];
```

```
Int key[SIZE];
```

```
/*after*/
```

```
Struct merge{
```

```
    int val;
```

```
    int key;
```

```
}
```

```
Struct merge merged_array[SIZE]
```

b. Loop Interchange


By switching the order in which loops execute, misses can be reduced due to improvements in spatial locality.

/ 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;

c. Loop fusion

- By fusion the code into a single loop, the data that are fetched into the cache can be used repeatedly before being swapped out.
- Improving the temporal locality

- 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)
```

```
{
```

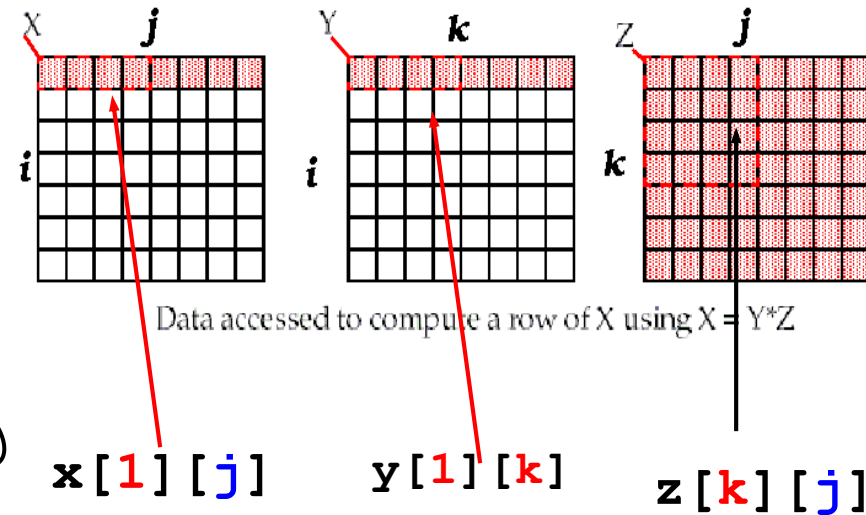
```
    a[i][j]=1/b[i][j]*c[i][j];
```

```
    d[i][j]=a[i][j]*c[i][j];
```

```
}
```

d. Unoptimized Matrix Multiplication

```
/* 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:

- Write N elements of 1 row of X[]
- Read N elements of 1 row of Y[] repeatedly
- Read all NxN elements of Z[]

$((N+N)N+N)N = 2N^3 + N^2$
Accessed For N^3 operations

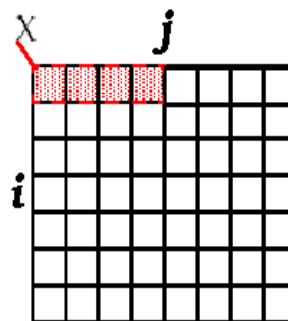
■ Capacity Misses a function of N & Cache Size:

- $2N^3 + N^2 \Rightarrow$ (assuming no conflict; otherwise ...)

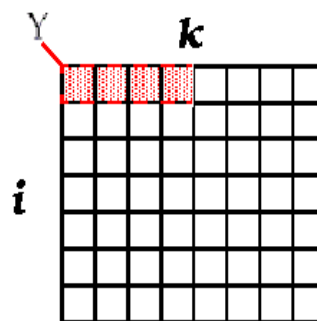
■ Idea: compute on BxB submatrix that fits

Blocking optimized Matrix Multiplication

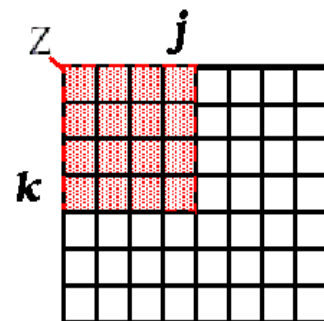
■ *Matrix multiplication is performed by multiplying the submatrices first.*



BN



BN



$B \times B$

B called **Blocking Factor**

/* After */

for (jj = 0; jj < N; jj = jj+B)

for (kk = 0; kk < N; kk = kk+B)

for (i = 0; i < N; i = i+1)

for (j = jj; j < min(jj+B-1, N); j = j+1)

{r = 0;

for (k = kk; k < min(kk+B-1, N); k = k+1)

r = r + y[i][k]*z[k][j];

x[i][j] = x[i][j] + r;

};

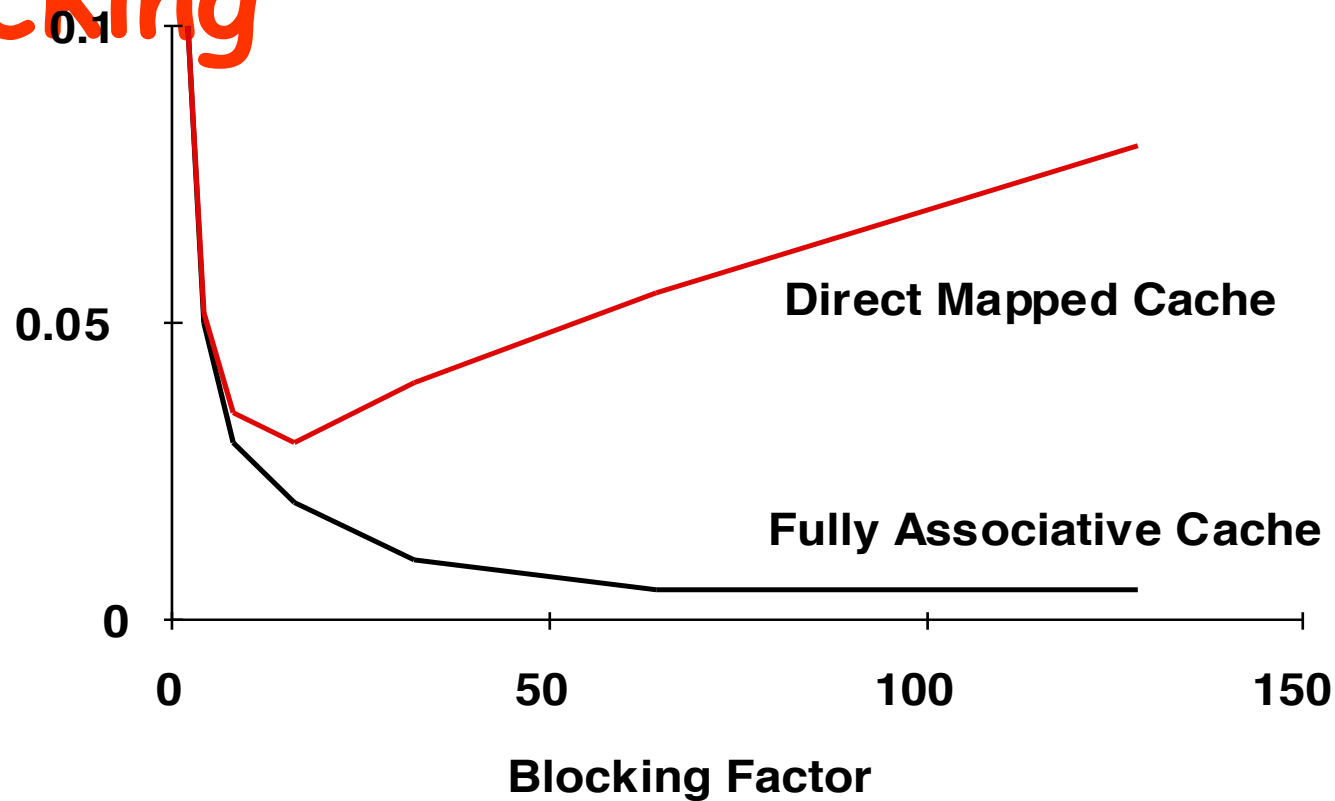
$(BN+BN)+B^2) \times (N/B)^2 = 2N^3/B + N^2$
Accessed For N^3 operations

Y benefits from **spatial** locality

Z benefits from **temporal** locality

Capacity Misses from $2N^3 + N^2$ to $N^3/B + 2N^2$

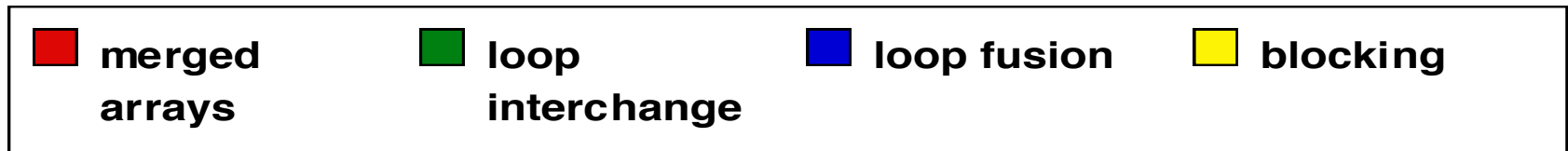
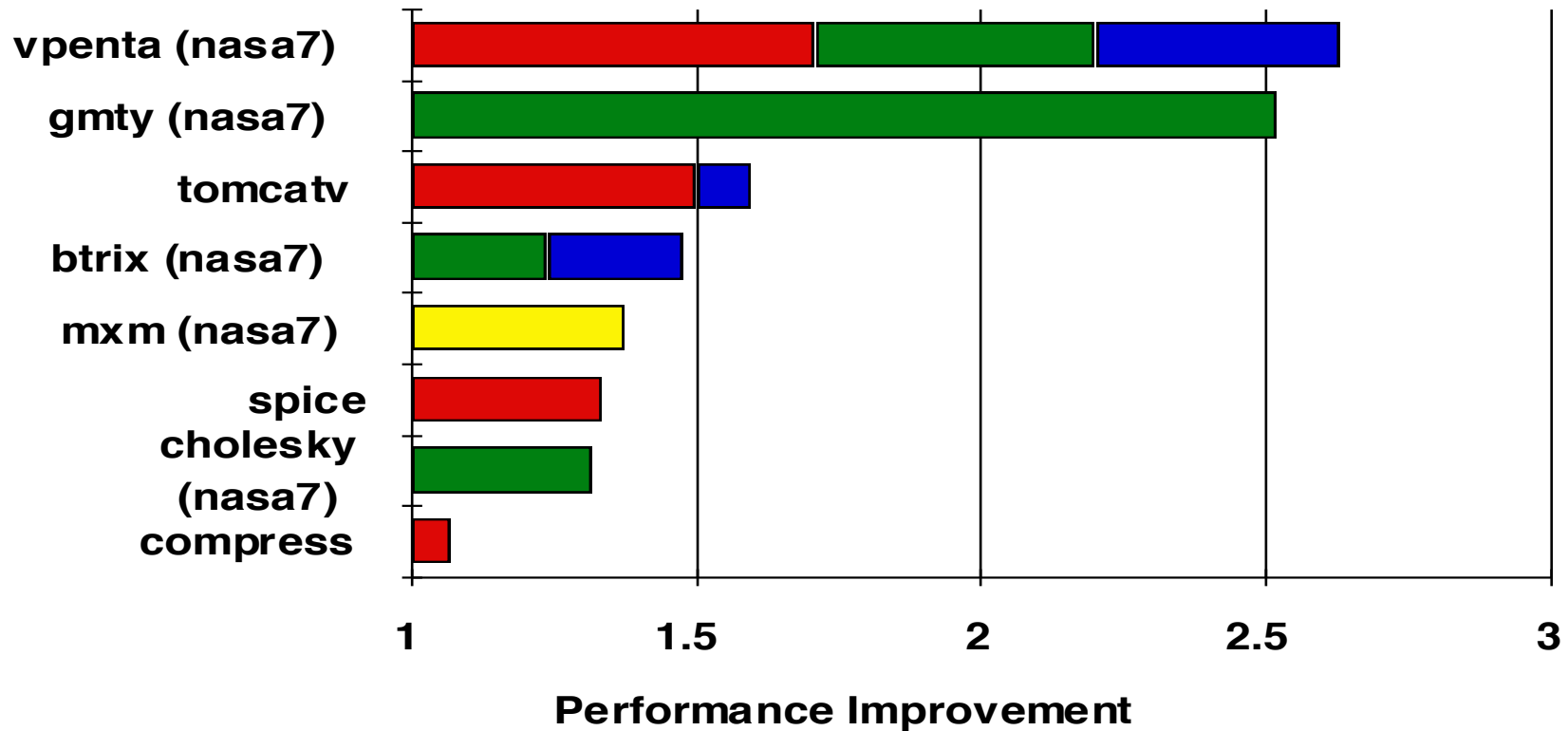
Reducing Conflict Misses by Blocking



■ Conflict misses in caches not FA vs. Blocking size

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

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



Summary: Miss Rate Reduction

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

■ 3 Cs: Compulsory, Capacity, Conflict

1. Larger cache
2. Reduce Misses via Larger Block Size
3. Reduce Misses via Higher Associativity
4. Reducing Misses by Compiler Optimizations
5. Pseudo associative cache

How to Improve Cache Performance?

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

1. Reduce the time to hit in the cache.--4
—small and simple caches, avoiding address translation, way prediction, and trace caches
2. Increase cache bandwidth.--3
—pipelined cache access, multibanked caches, non-blocking caches,
3. Reduce the miss penalty--4
—multilevel caches, critical word first, read miss prior to writes, merging write buffers, and victim caches
4. Reduce the miss rate--4
—larger block size, large cache size, higher associativity, and compiler optimizations
5. Reduce the miss penalty and miss rate via parallelism--2
—hardware prefetching, and compiler prefetching

1st Miss Penalty/Rate Reduction Technique: Hardware Prefetching of Inst. and data

- The act of *getting data from memory before it is actually needed by the CPU.*
- This *reduces compulsory misses* by retrieving the data before it is requested.
- Of course, this may increase other misses by removing useful blocks from the cache.
 - Thus, many caches hold prefetched blocks in a special buffer until they are actually needed.
- E.g., Instruction Prefetching
 - Alpha 21064 fetches 2 blocks on a miss
 - Extra block placed in "stream buffer"
 - On miss check stream buffer
- Prefetching relies on having extra memory bandwidth that can be used without penalty

2nd Miss Penalty/Rate Reduction Technique: Compiler-controlled prefetch

- The compiler inserts *prefetch instructions* to request the *data* before they are needed
- 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

Example (P307):

Compiler-controlled prefetch

```
for( i=0; i < 3; i = i + 1)
  for( j=0; j < 100; j = j + 1)
    a[i][j] = b[j][0] * b[j+1][0];
```

16B/block, 8B/element, 2elements/block

A[i][j]: 3*100 the even value of j will miss,
the odd values will hit, total **150** misses

B[i][j]: 101*3 the same elements are accessed for each
iteration of i

j=0 B[0][0], B[1][0] 2

j=1 B[1][0], B[2][0] 1

total 2+99=**101** misses

Example cont.:

Compiler-controlled prefetch

```
For (j=0; j<100; j=j+1){  
    Prefetch(b[j+7][0]);  
    prefetch(a[0][j+7]);  
    a[0][j]=b[j][0]*b [j+1][0];};
```

7 misses for b
4 misses for a

```
For (I=1; I<3; I=I+1)  
For (j=0; j<100; j=j+1){  
    prefetch(a[i][j+7]);  
    a[i][j]=b[j][0]*b [j+1][0];};
```

4 misses for a[1][j]
4 misses for a[2][j]

Total: 19 misses

save 232 cache misses at the price of 400 prefetch instructions.

End.