Computer Architecture
----A Quantitative Approach

陈文智

浙江大学计算机学院

chenwz@zju.edu.cn

5.4 Reducing Cache miss penalty

1.Reduce the miss penalty

2. Reduce the miss rate

3. Reduce the miss penalty and miss rate via parallelism

4. Reduce the time to hit in the cache.

First Miss Penalty Reduction Technique:

- Multilevel Caches
  - This method focuses on the interface between the cache and main memory.
  - We can add an second-level cache between main memory and a small, fast first-level cache.
    - This helps satisfy the desire to make the cache fast and large.
  - The first –level cache allows:
    - 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 allows:
    - The larger second-level 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

$$AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} \times Miss\ Penalty_{L1}$$

$$Miss\ Penalty_{L1} = Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}$$

$$AMAT = Hit\ Time_{L1} + Miss\ Rate_{L1} \times (Hit\ Time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2})$$

- 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*
  - Global Miss Rate is what matters

Parameter about Multilevel cache

- *The performa… …r way to the performan…*
- 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.

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

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

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

$$Miss\ rate_{L1} = \frac{Misses_{L1}}{M}$$

$$Miss\ rate_{L2} = \frac{Misses_{L2}}{M_{L2}} = \frac{Misses_{L2}}{M \times Miss\ rate_{L1}}$$

- 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*
  - Global Miss Rate is what matters

Parameter about Multilevel cache

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 :

$$Miss\ rate_{Global-L2} = \frac{Misses_{L2}}{M} = \frac{M \times Miss\ rate_{L1}}{M} \times \frac{Misses_{L2}}{M \times Miss\ rate_{L1}}$$

$$= \frac{Misses_{L1}}{M} \times \frac{Misses_{L2}}{M \times Miss\ rate_{L1}} = Miss\ rate_{L1} \times Miss\ rate_{L2}$$

Memory stall per instruction

- This local miss rate is large for second-level caches before the first-level cache skims the cream of the memory accesses. This is why the global miss rate is more useful measure: It indicates what fraction of the memory accesses that leave the CPU go all the way to memory.
- Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second-level cache:

Average memory stalls per instruction

$$= \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}$$

Example7: Multilevel cache

Assume(p414):

L1 cache: Misses: 40 misses/1000 memory
Hit time: 1 clock cycles
Memory references per instruction: 1.5

L2 cache: 20 misses
Miss penalty: 100 clock cycles
Hit time: 10 clock cycles

What is the average memory access time and average stall cycles per instruction (Ignore the impact write) ?

Answer: Calculating Miss rate for local and global.

$$Miss\ rate_{L1} = \frac{Misses_{L1}}{M} = \frac{40}{1000} = 4\% \qquad Miss\ rate_{L2} = \frac{Misses_{L2}}{M_{L2}} = \frac{20}{40} = 50\%$$

$$Miss\ rate_{G2} = \frac{Misses_{L2}}{M} = \frac{20}{1000} = 2\%$$

AMAT = Hit time$_{L1}$ + Miss rate$_{L1}$ × (Hit time$_{L2}$ + Miss rate$_{L2}$ × Miss penalty$_{L2}$)
= 1 + 4% × (10 + 50% × 100) = 1 + 4% × 60 = 3.4 clock cycle

Example7: Multilevel cache

Misses number Per 1000 instructions:

L1(global): 1.5 × 4% ×1000＝60

L2(global): 1.5 × 2% ×1000＝30

Form average memory stalls per instruction for distributed uniformly:

Form average memory stalls per instruction

＝Misses per instruction$_{L1(global)}$ ×Hit time$_{L2}$+Misses per instruction$_{L2(global)}$ ×Miss penalty$_{L2}$＝(60/1000) ×10+(30/1000) ×100

＝0.060 ×10+0.030 ×100＝3.6 clock cycles

if we subtract the L1 hit time from AMAT and then multiply by the average number of memory references per instruction,we get the same average memory stalls per instruction:

(3.4-1.0) ×1.5＝2.4 ×1.5＝3.6 clock cycles

Example8: Multilevel cache

Assume(p714): Hit time$_{L2}$ for direct mapped = 10 clock cycles.

Hit time$_{L2}$ for two-way set = 10.1 clock cycles.

Local miss rate$_{L2}$ for direct mapped = 25%.

Local miss rate$_{L2}$ for two-way set associative =20%.

Miss penalty$_{L2}$ = 100 clock cycles.

What is the impact of second-level cache associativity on its miss penalty?

Answer: For a direct-mapped L2 cache, L1 cache miss penalty is :

$$\text{Miss penalty}_{1\text{-way}L1} = 10 + 25\% \times 100 = 35.0 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycles, making the new L1 cache miss penalty

$$\text{Miss penalty}_{2\text{-way}L1} = 10.1 + 20\% \times 100 = 30.1 \text{ clock cycles}$$

Example8: Multilevel cache

In reality, L2 cache are almost always synchronized with the L1 cache and CPU. Accordingly, the L2 hit time must be an integral number of clock cycles. If we are lucky,we shave the second-level hit time to 10 cycles;if not,we round up to 11 cycles. Either choice is an improvement over the direct-mapped L2 cache:

$$\text{Miss penalty}_{\text{2-way L1}} = 10 + 20\% \times 100 = 30.0 \text{ clock cycles}$$

$$\text{Miss penalty}_{\text{2-way L1}} = 11 + 20\% \times 100 = 31.0 \text{ clock cycles}$$

Now we can reduce the miss penalty by reducing the miss rate of the second-level cache

Second 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

Example9: Critical Word First

Assume(p419): 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.

Answer:

The average miss penalty is 11 clock cycles for critical word first.

The Athlon can issue 2 loads per clock, which is faster the L2 cache can supply data. Thus, for the CPU to sequentially read a full cache block it would take:

$$11+(8-1)\times 2=25 \text{ clock cycle}$$

Without for critical word first, it would take 25 clock cycles to load the block, and then 8/2 or 4 clocks to issue the loadd, giving 29 clock cycles total

Third 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.*

Third Miss Penalty Reduction Technique:

- 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 w/ 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

Example10: Giving Priority to Read Misses over Writes

Assume: Code sequence following:

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

Direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer

Will the value in R2 always be equal to the value in R3?

Answer: There is a read-after-write data hazard in memory.

Let's follow a cache access to see the danger.

The data in R3 are placed into the write buffer after the store.

The following load uses the same cache index and is therefore a miss.

The second load instruction tries to put the value in location 512 into register R2; this also results in a miss.

If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2. Without proper precautions,

R3 would not be equal to R2!

Fourth Miss Penalty Reduction Technique:

- Merging write Buffer
  - *One word writes replaces with multiword writes, and it improves buffers's efficiency.*
  - *In write-through , 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.*
  - *The optimization also reduces stalls due to the write buffer being full.*

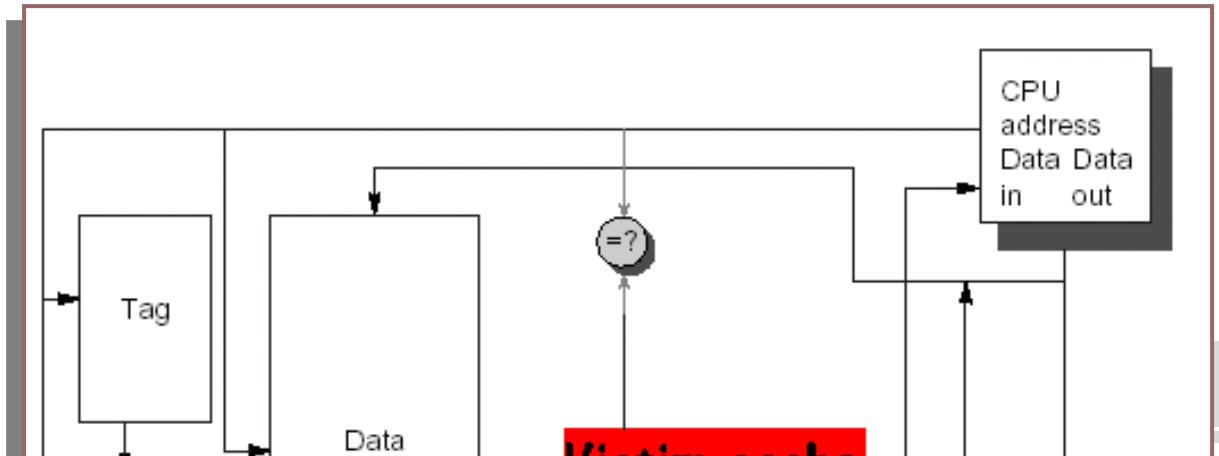
Fourth Miss Penalty Reduction Technique:

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	

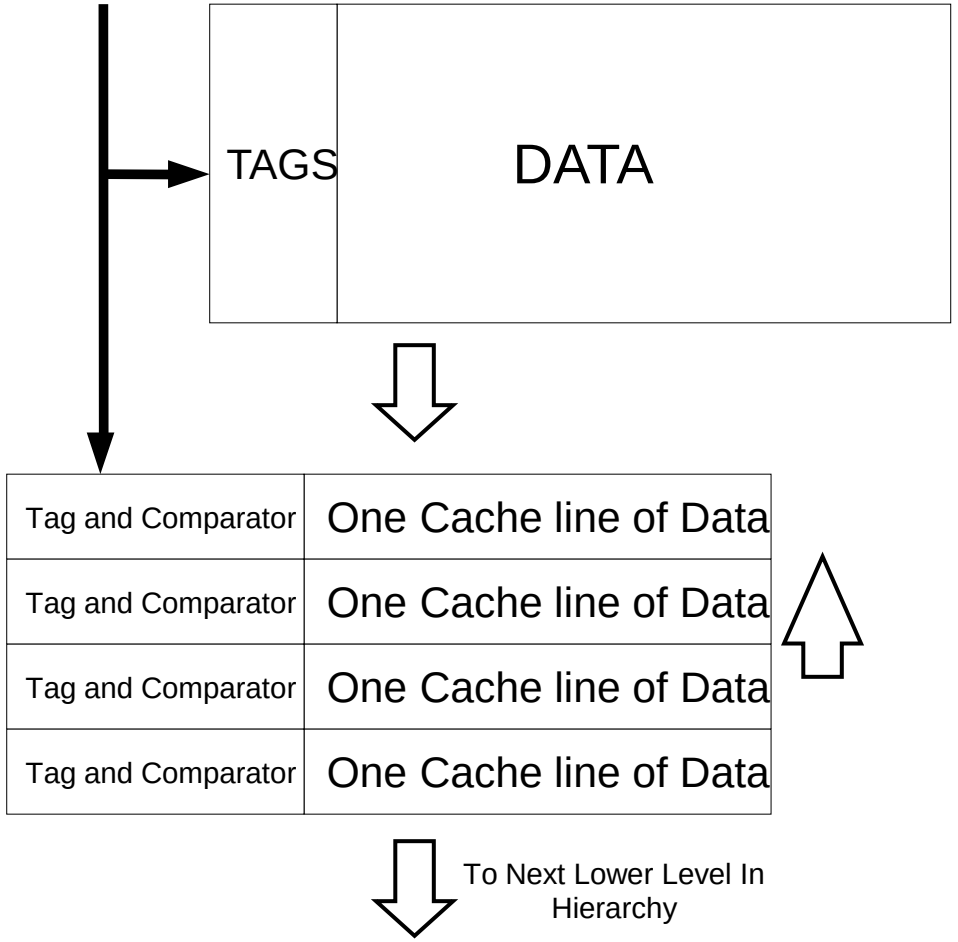
Fifth Miss Penalty Reduction Technique:

- 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
    - If found, the victim block and the cache block are swapped.
    - The AMD Athlon has a victim caches with 8 entries.



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

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

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
5. Reducing penalty via Victim Caches

5.5 Reducing miss rate

1.Reduce the miss penalty

2. Reduce the miss rate

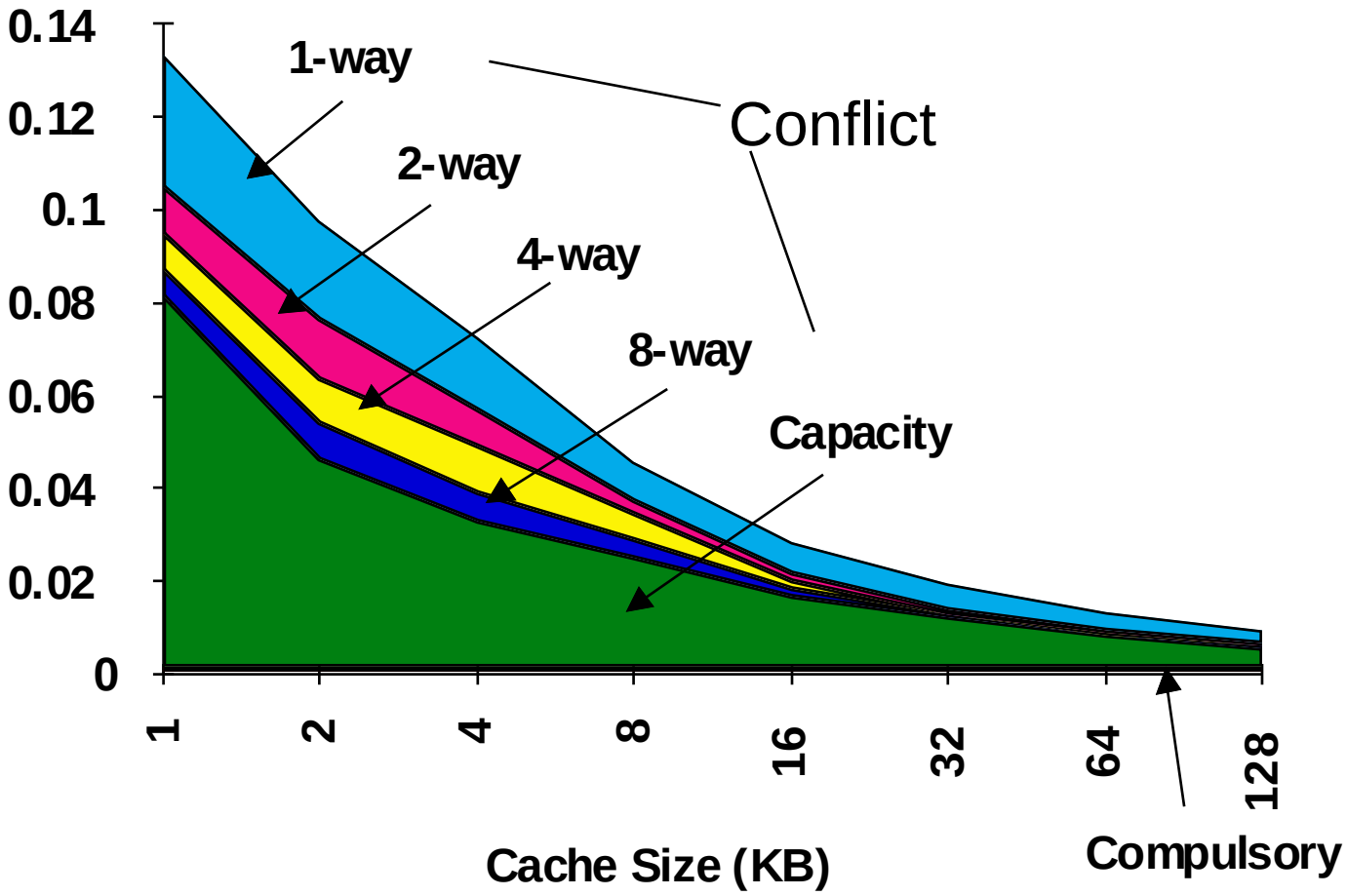
3. Reduce the miss penalty and miss rate via parallelism

4. Reduce the time to hit in the cache.

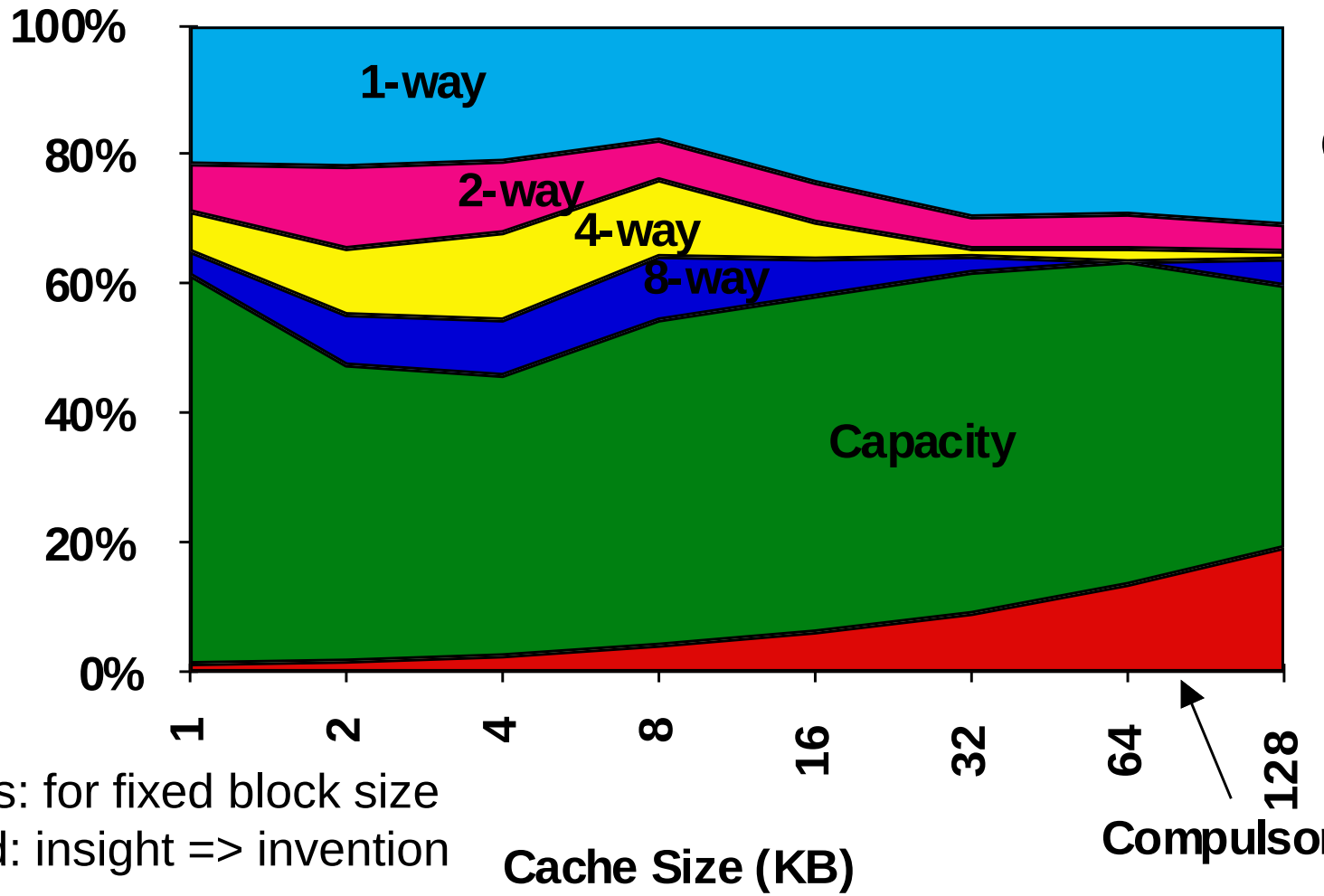
Where do misses come from?

- 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:
  - Change Block Size
  - Change Associativity
  - Change Compiler
- Which of 3Cs is obviously affected?

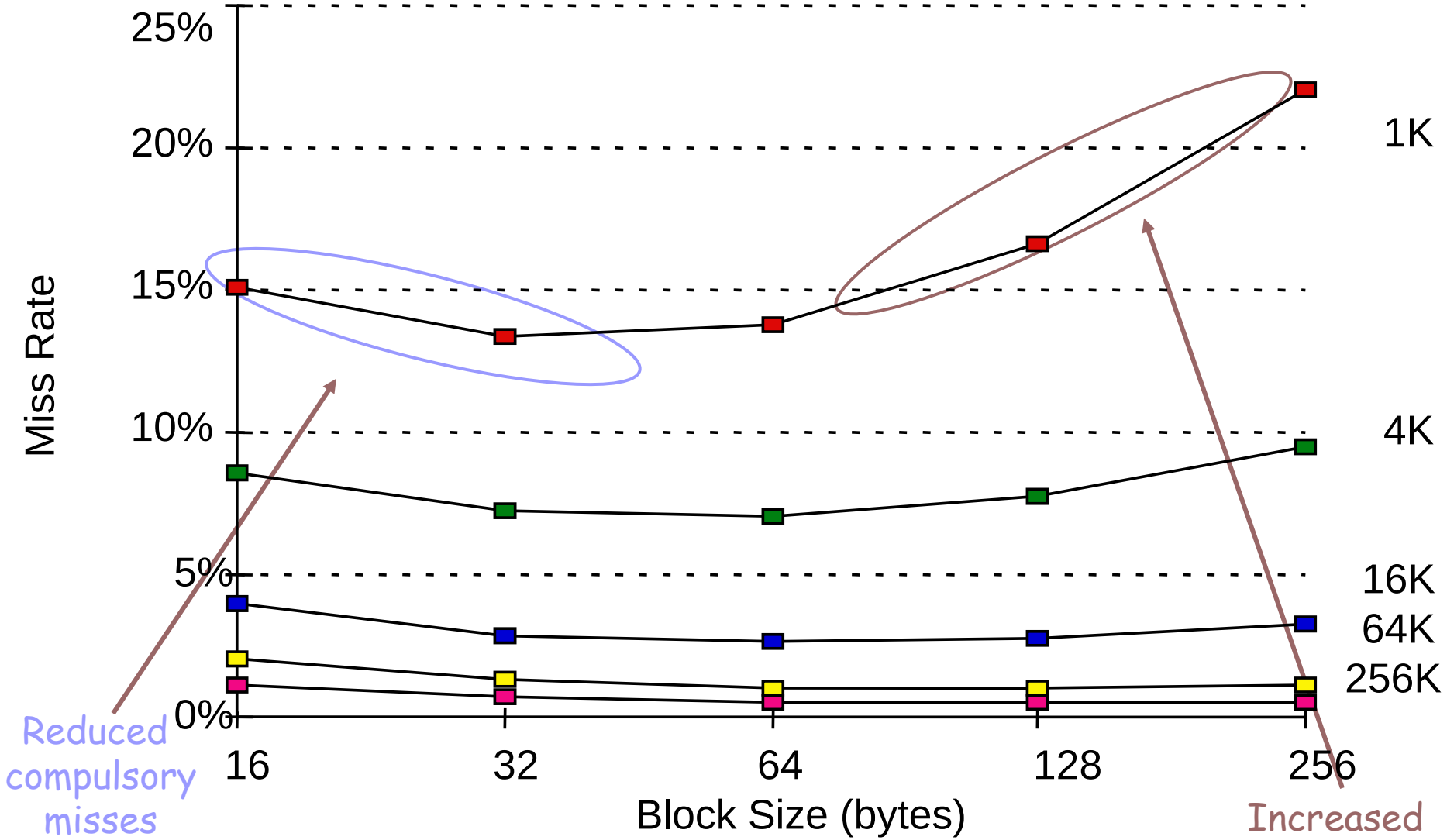
First Miss Rate Reduction Technique

- 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

First Miss Rate Reduction Technique

Block size	Cache size				
	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

The performance curve is U-shaped



Example11: Larger Block Size

Assume: actual miss rates shows in the table

Block size	Cache size				
	1K	4K	16K	64K	256K
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Hit time: 1CLK independent of block size

Memory system overhead: 80 CLK
Delivering time: 16 bytes/2 CLK

This equals that it can supply 16 bytes in 82 CLK, 32 bytes in 84 CLK and so on.

Which block size has the smallest average memory access time for each cache size in above table?

Example11: Larger Block Size-2

Answer: Average memory access time is:

Average memory access time＝Hit time + Miss rate×Miss penalty

16-byte block in 1KB cache is

Average memory access time＝1 + (15.05% ×82)＝13.314 CLK

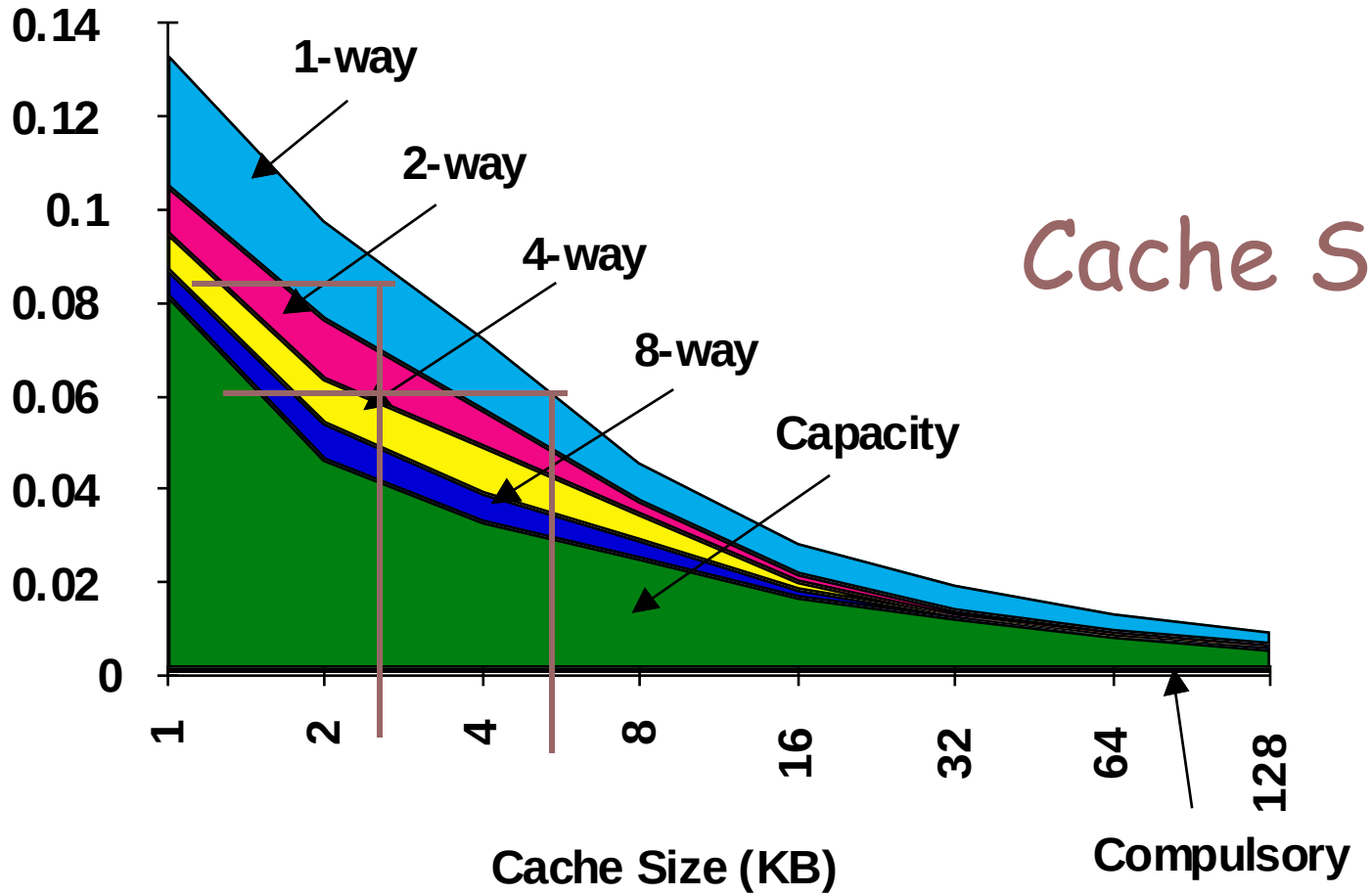
256-byte block in 256KB cache is

Average memory access time＝1 + (0.49% ×112)＝1.549 CLK

All average memory access times are calculated showing in following table

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

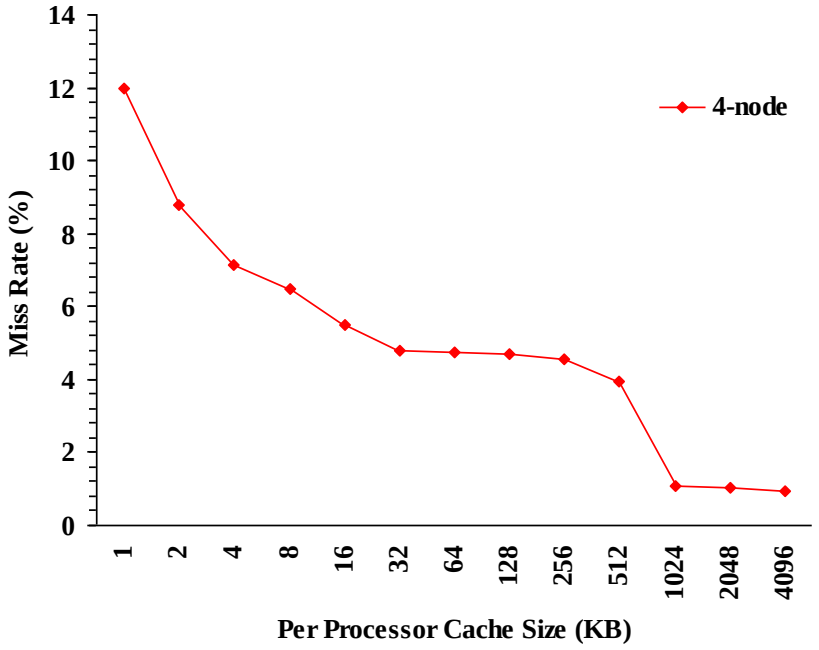
Second Miss Rate Reduction Technique

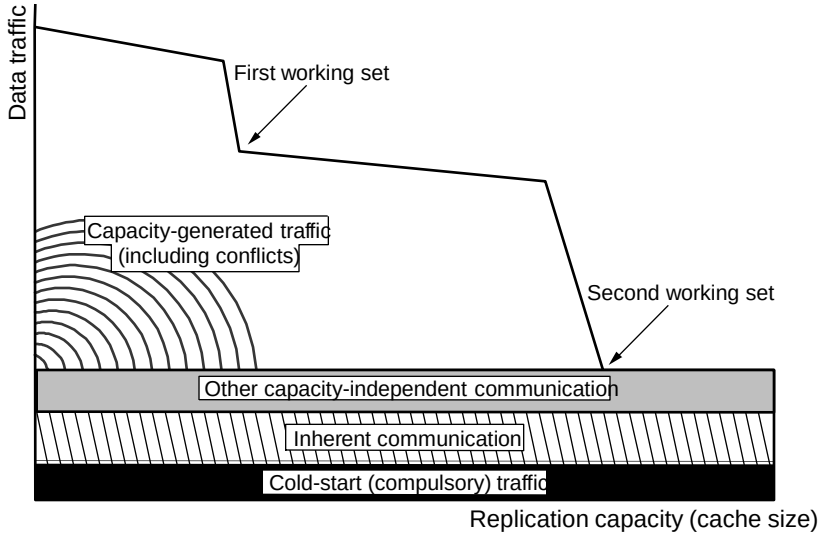


- Larger Caches
  - Old rule of thumb: 2 x size => 25% cut in miss rate
  - What does it reduce?

- Drawback
  - Longer hit time.
  - Higher cost.

Huge Caches => Working Sets

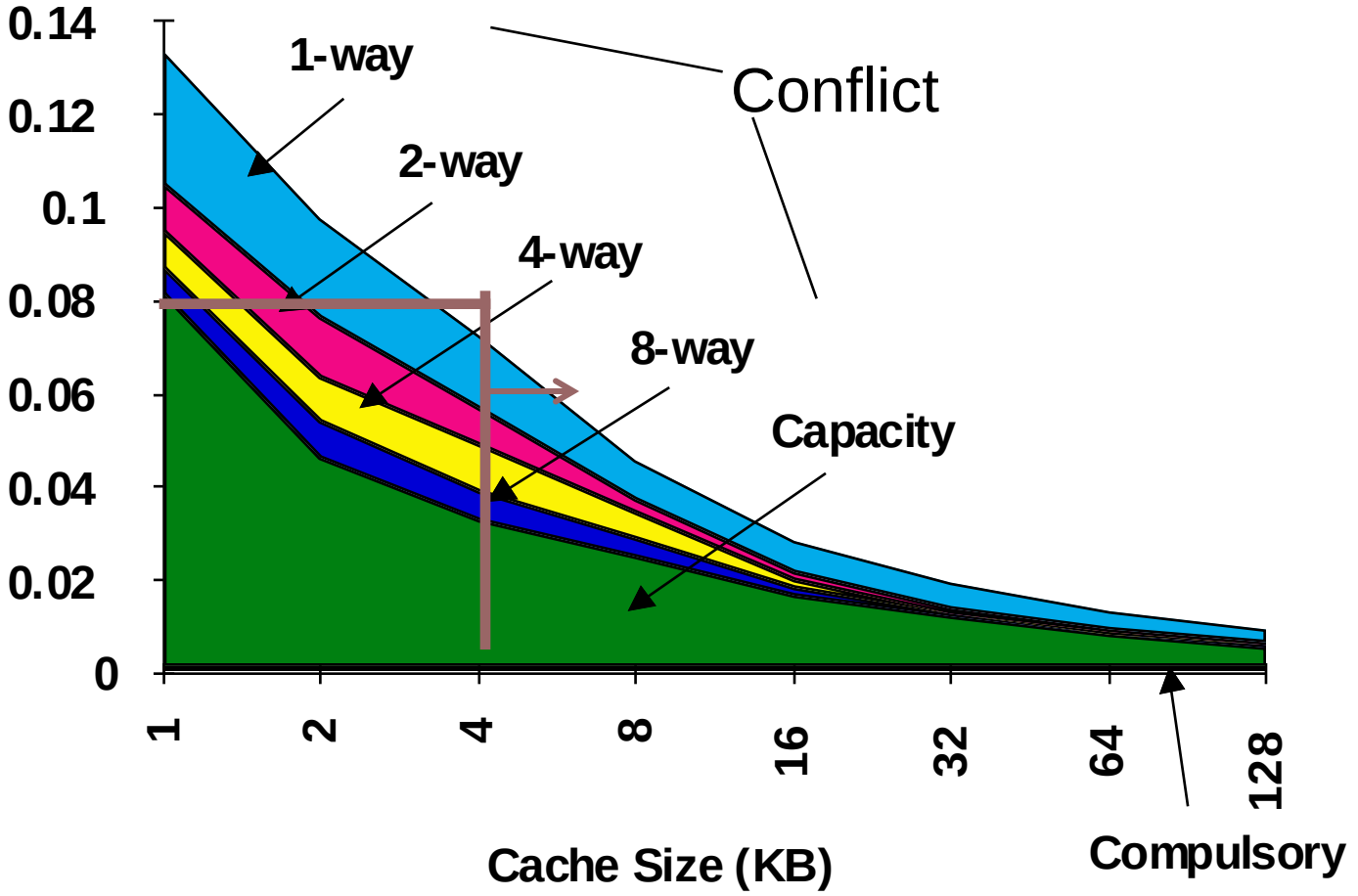




Example LU Decomposition
from NAS Parallel Benchmarks

Third 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.


0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
1-way
2-way
4-way
8-way
Conflict
Capacity
Compulsory
1
2
4
8
16
32
64
128
Cache Size (KB)

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

- Effective cycle time of assoc
  - pzrbski ISCA

Example12: Higher Associativity

Assume:

$$\text{Clock Cycle time}_{2\text{-way}} = 1.36 \times \text{Clock Cycle time}_{1\text{-way}}$$
$$\text{Clock Cycle time}_{4\text{-way}} = 1.44 \times \text{Clock Cycle time}_{1\text{-way}}$$
$$\text{Clock Cycle time}_{8\text{-way}} = 1.52 \times \text{Clock Cycle time}_{1\text{-way}}$$

Hit time: 1 CLK

Miss penalty$_{\text{direct-mapped}}$: 25 CLK

L2 cache that never misses, and that the penalty need not rounded to integral number of clock cycle

Using Figure 5.14(p424) for miss rates, which cache sizes each of these three statements true?

$$\text{Average memory access time}_{8\text{-way}} < \text{Average memory access time}_{4\text{-way}}$$
$$\text{Average memory access time}_{4\text{-way}} < \text{Average memory access time}_{2\text{-way}}$$
$$\text{Average memory access time}_{2\text{-way}} < \text{Average memory access time}_{1\text{-way}}$$

Example12: Higher Associativity-2

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)

Fourth 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
    - Extra bits are kept in the cache to predict the way, or block within the set of the next cache access.
    - The Alpha 21264 user uses way prediction in its two-way set-associative instruction cache.
      - 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

Fifth 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

①Loop Interchange

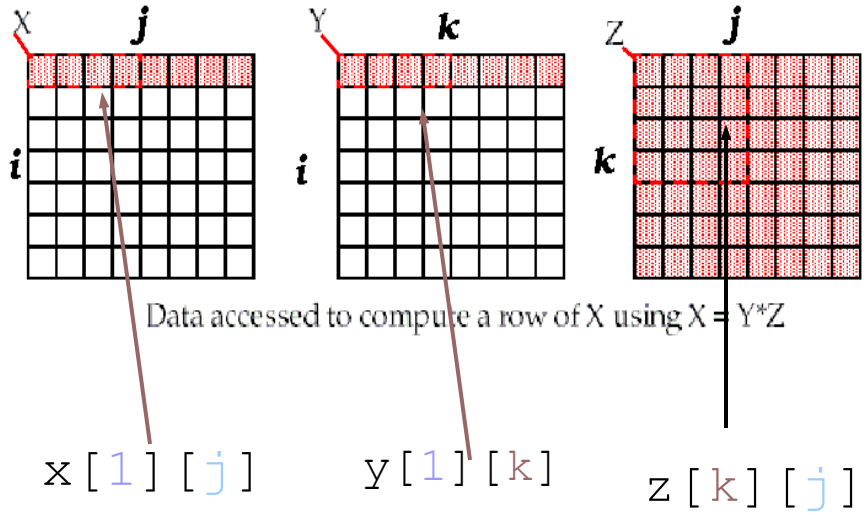
By switching the order in which loops execute, misses can be reduced due to improvements in spatial locality.

```
/* 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;

② 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;
      };
```

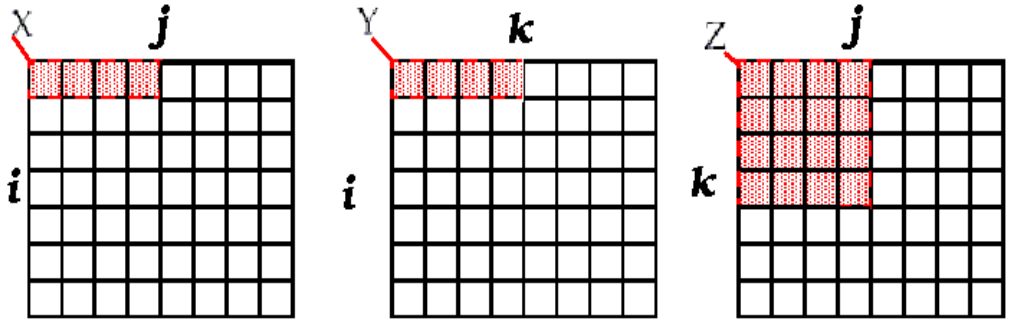


- Two Inner Loops:
  - Write N elements of 1 row of X[ ]
  - Read N elements of 1 row of Y[ ] repeatedly
  - Read all NxN elements of Z[ ]
- Capacity Misses a function of N & Cache Size:
  - $2N^3 + N^2$ => (assuming no conflict; otherwise …)
- Idea: compute on BxB submatrix that fits

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

Blocking optimized Matrix Multiplication

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



BN BN B×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

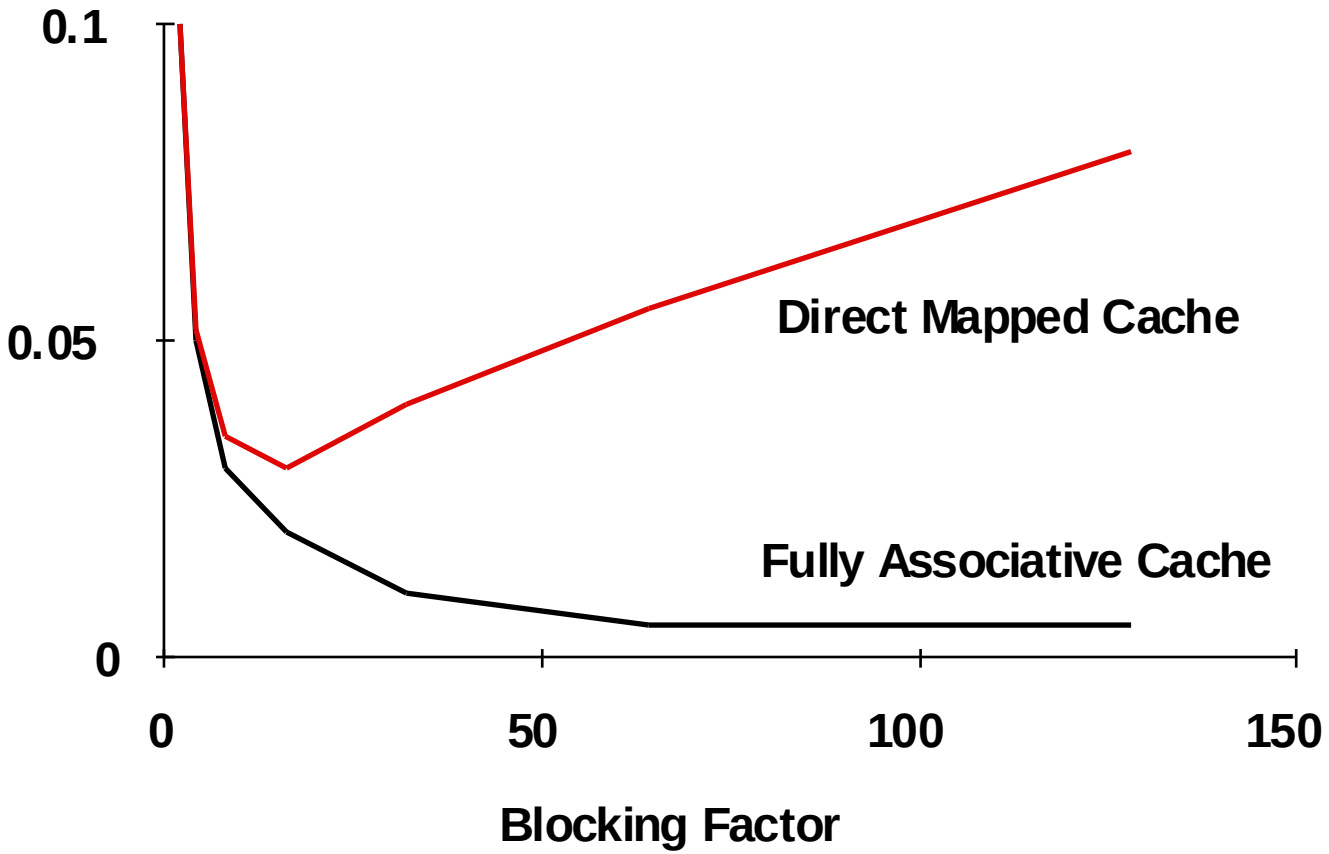
Y benefits from spatial locality
Z benefits from temporal locality
Capacity Misses from $2N^3 + N^2$ to $2N^3/B+N^2$

- I→N;J→B;K→B;jj→N/B;KK→N/B; SO

- For x[I,J]
  - N/B * N/B * (N*B)
- For y[I,K]
  - N/B * N/B * (N*B)
- For z[K,J]
  - N/B * N/B * (B*B)

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



③Loop Fusion

```
/* 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

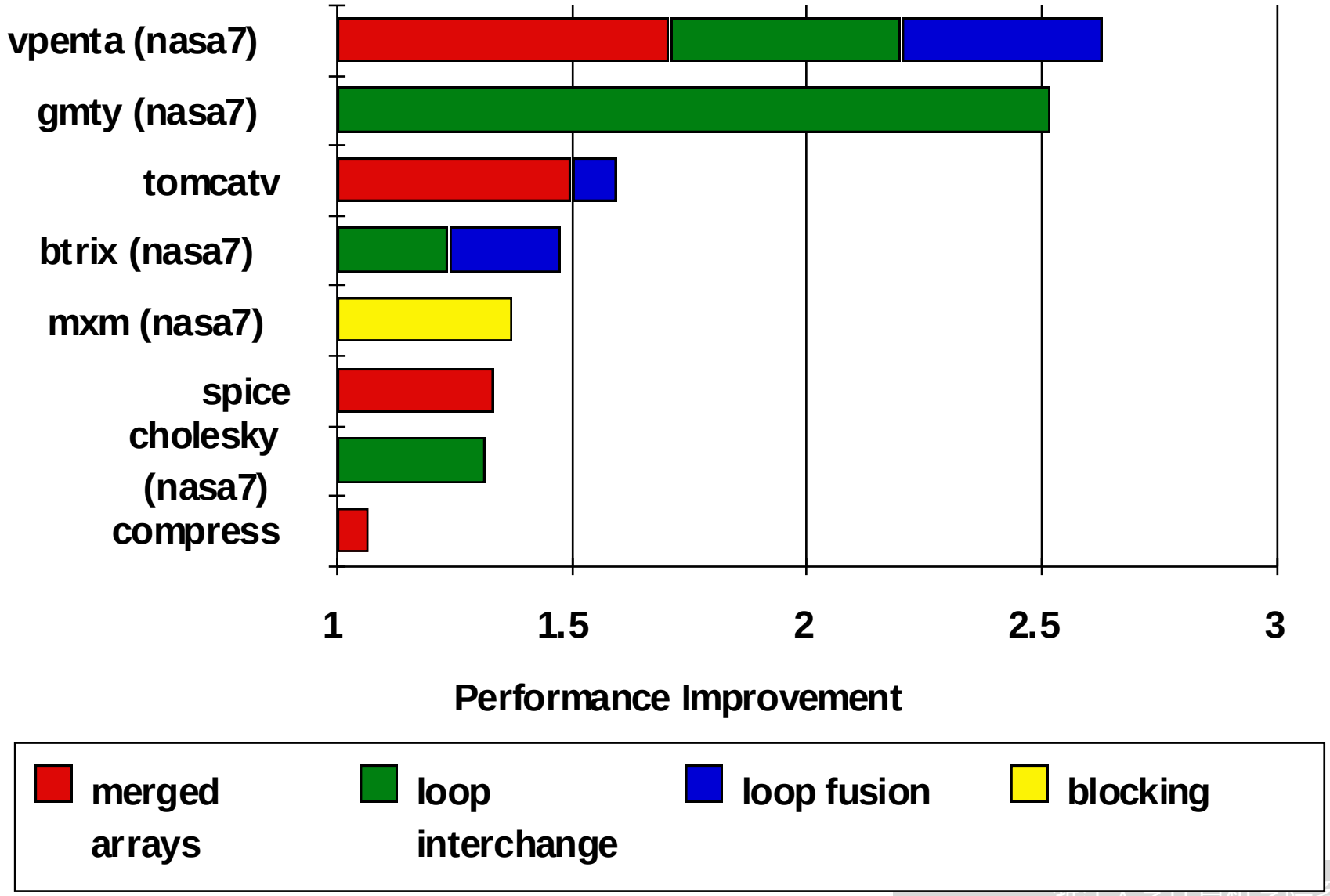
④Merging Arrays

```
/* 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

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



Summary: Miss Rate Reduction

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

- 3 Cs: Compulsory, Capacity, Conflict
  - 1. Larger cache
  - 2. Reduce Misses via Larger Block Size
  - 3. Reduce Misses via Higher Associativity
  - 4. Reducing Misses via Pseudo-Associativity
  - 5. Reducing Misses by Compiler Optimizations

1.Reduce the miss penalty

2. Reduce the miss rate

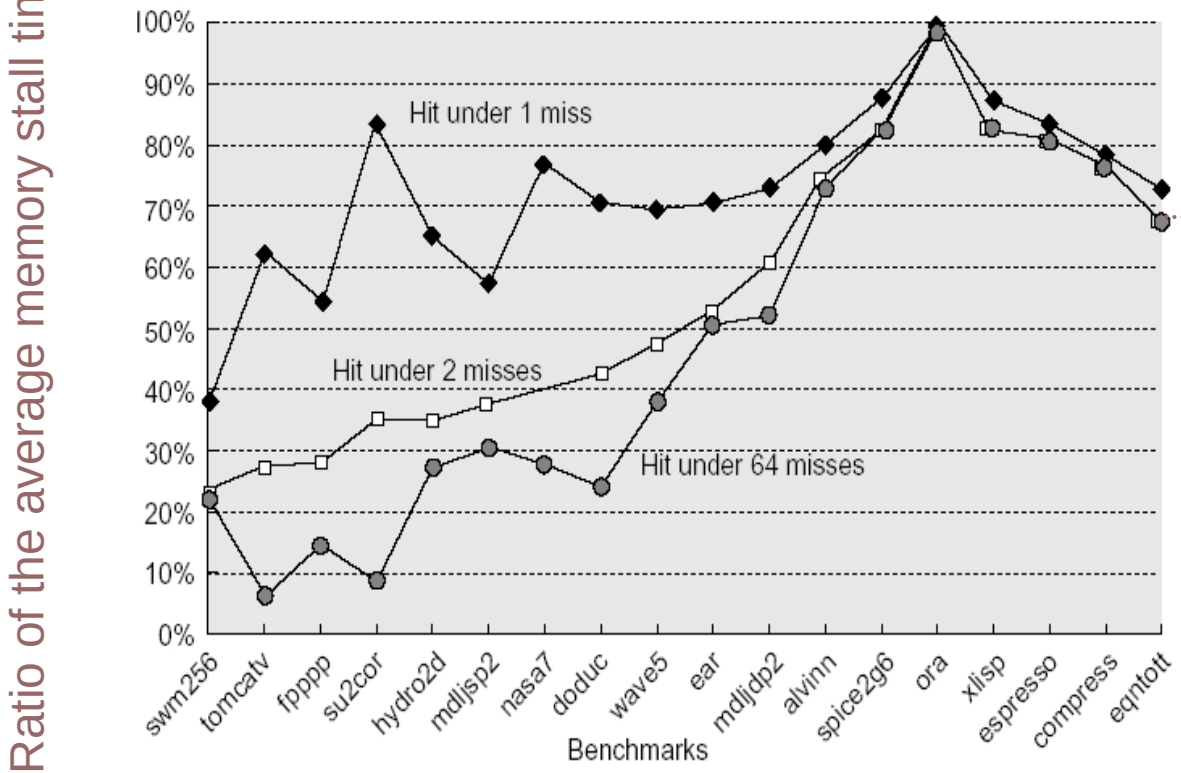
3. Reduce the miss penalty and miss rate via parallelism

4. Reduce the time to hit in the cache.

First Miss Penalty/Rate Reduction Technique:

- Nonblocking Caches to Reduce Stalls on Cache Misses
  - Reducing Miss Penalty
  - A nonblocking(Lockup-free cache) cache, allows The cache to continues to supply hits while processing read misses (hit under miss , hit under multiple miss).
  - Complex caches can even have 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.

Ratio of the average memory stall time for a blocking cache to hit-under-miss schemes as the number of outstanding misses is varied for 18 SPEC92 programs.



The hit-under-64-misses line allows one miss for every register in the machine.

The first 14 programs are floating-point programs: the average for hit under 1 miss is 76%, for 2 misses is 51%, and for 64 misses is 39%.

The final four are integer programs, and the three averages are 81%, 78%, and 78%,

These data were collected for an 8-KB direct-mapped data cache with 32-byte blocks and a 16-clock-cycle miss penalty, which today would imply a second-level cache. These data were generated using the VLIW Multiflow Compiler, which scheduled loads away from use [Farkas and Jouppi 1994].

Example13: Nonblocking

Assume: For the cache described in Figure 5.23；

Miss rates for 8KB: 11.4% for fp-DM cache

10.7% for 2-way cache

Miss rates for 8KB: 7.4% for int-DM cache

6.0% for 2-way cache

① which is more important for floating-point programs: 2-way or hit under one miss?

② What about integer programs?

Answer: The numbers for Figure 5.23 were based on a miss penalty of 16 clock cycles. Although this is low for a miss penalty, let's stick with it for consistency. For floating-point programs the average memory stall times are:

$$\text{Miss rate}_{\text{DM}} * \text{Miss penalty} = 11.4\% * 16 = 1.84$$

$$\text{Miss rate}_{\text{2-way}} * \text{Miss penalty} = 10.7\% * 16 = 1.71$$

The memory stalls of two-way are thus 1.71/1.84 or 93% of direct-mapped cache.

Example13: Nonblocking

The caption of Figure 5.23 says hit under one miss reduces the average memory stall time to 76%(<93%) of a blocking cache.

Hence, so for floating-point programs the direct-mapped data cache supporting hit under one miss gives better performance than a two-way set-associative cache that blocks on a miss.

For integer programs the calculation is:

$$\text{Miss rate}_{\text{DM}} * \text{Miss penalty} = 7.4\% * 16 = 1.18$$

$$\text{Miss rate}_{\text{2-way}} * \text{Miss penalty} = 6.0\% * 16 = 0.96$$

The memory stalls of two-way are thus 0.96/1.18 or 81% (=81%)of direct-mapped cache.

The caption of Figure 5.23 says hit under one miss reduces the average memory stall time to 81% of a blocking cache, so the two options give about the same performance for integer programs.

One advantage of hit under miss is that it cannot affect the hit time, as associativity can.

Second Miss Penalty/Rate Reduction Technique:

- Hardware Prefetching of Instructions and data
  - Reducing Misses
  - 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

Third Miss Penalty/Rate Reduction Technique:

- Compiler-controlled prefetch
  - Reducing Misses
  - 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

Example13: Compiler-controlled prefetch

Assume: cache sizes: 8-KB
blocks sizes: 16-byte
data cache strategy: direct-mapped 、 write-back cache 、 write allocate

The elements of `a` and `b` are 8 bytes long since they are double-precision floating-point arrays.
a array: 3 rows and 100 columns
b array: 101 rows and 3 columns
they are not in the cache at the start of the program.

For the code below:
First determine which accesses are likely to cause data cache misses.
Next, insert prefetch instructions to reduce misses.
Finally, calculate the number of prefetch instructions executed and the misses avoided due to prefetching.

```
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]
```

Example13: Compiler-controlled prefetch-2

Answer: First determine which accesses are likely to cause data cache misses.

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

Elements of a are written in the order that they are stored in memory, so a will benefit from spatial locality:

The even values of `j` will miss and the odd values will hit. Since :

A[I][J]: 3×100 accesses will lead to:

$$3 * \frac{100}{2} = 150 \quad \text{Misses}$$

```
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]
```

Since the accesses are not in the order it is stored. The array `b` does not benefit from spatial locality. The array `b` does benefit twice from temporal locality: the same elements are accessed for each iteration of `i`, and each iteration of `j` uses the same value of `b` as the last iteration.

j=0	B[0][0]、B[1][0]	2	accesses first
j=1	B[1][0]、B[2][0]	1	accesses next from j=1 to j=99

2+99=101 misses

Total misses: 150 + 101 =251

Example13: Compiler-controlled prefetch-3

Next, insert prefetch instructions to reduce misses.

This revise code prefetched a[I][7] through a[I][99] and b[7][0] through b[100][0].

```
for (j = 0; j < 100; j = j+1) {
      prefe
  /* b(j,0)
      prefe
  /* a(0,j) for 7 iterations later */
      a[0][j] = b[j][0] * b[j+1][0];};
for (i = 1; i < 3; i = i+1)
      for (j = 0; j < 100; j = j+1) {
      prefetch(a[i][j+7]);
  /* a(i,j) for +7 iterations */
      a[i][j] = b[j][0] *b[j+1][0];}
```

Expense 400 prefetch instructions

Total misses 7+4×3＝19

7 misses for elements

4 misses ⌈7/2] for elements a[0][0], a[0][2], …, a[0][6]

4 misses ⌈7/2] for elements a[1][0], a[1][2], …, a[1][6]

4 misses ⌈7/2] for elements a[2][0], a[2][2], …, a[2][6]

Example13: Compiler-controlled prefetch-3

Expense 400 prefetch instructions
Total misses $7+4\times3=19$

Finally, calculate the number of prefetch instructions executed and the misses avoided due to prefetching.

The improving performance of decreased cache misses is:
$251-19=232$

Example13: Compiler-controlled prefetch-4

Assume: Here are the key loop times ignoring cache misses:

The original loop takes 7 clock cycles per iteration

The first prefetch loop takes 9 CLK per iteration

The second prefetch loop takes 8 CLK per iteration

A miss takes 100 CLK

How much time it taken that saved in the example?

Answer original doubly nested loop executing without cache miss:

$$3 \times 100 \times 7 = 2100$$

Plus cache miss time

$$2100 + 251 \times 100 = 27200 \text{ CLK}$$

After prefetch:

First iteration: $100 \times 9 + 11 \times 100 = 2000$

Second iteration: $200 \times 8 + 8 \times 100 = 2400$

Total time: $2000 + 2400 = 4400$ CLK

Example13: Compiler-controlled prefetch-4

If we assume that prefetches are completely Overlapped with the rest of the execution, then the prefetch code is faster:

$$\frac{27200}{4400} = 6.2 \text{ times}$$

Summary: Reduce Cache Miss Penalty or Miss Rate via Parallelism

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

1. Reduce Miss penalty via Nonblocking Caches

2. Reduce Misses via Hardware Prefetching

3. Reduce Misses via Compiler-controlled prefetch

5.7 Reducing Hit Time

1.Reduce the miss penalty

2. Reduce the miss rate

3. Reduce the miss penalty and miss rate via parallelism

4. Reduce the time to hit in the cache.

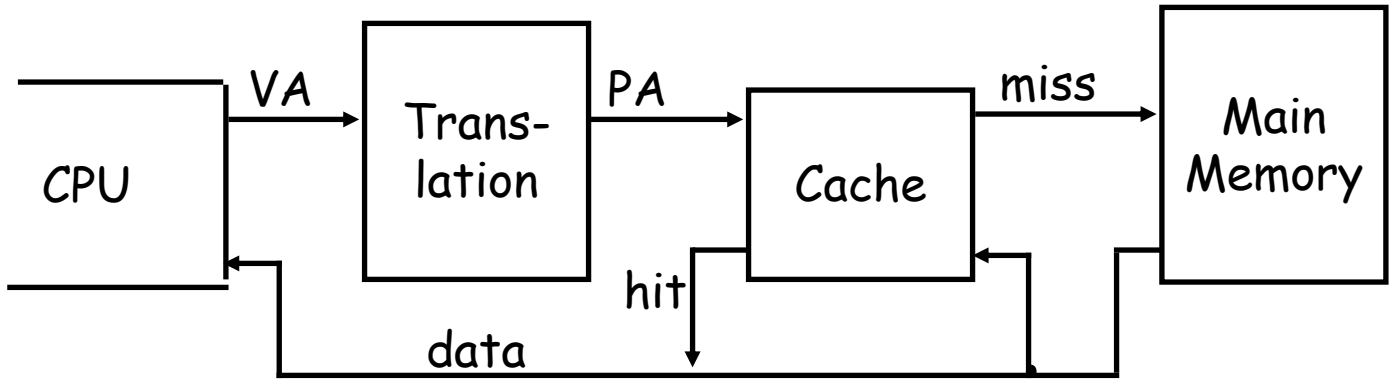
First Hit Time Reduction Technique:

- 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.

Second Hit Time Reduction Technique:

- Avoiding Address Translation during Indexing of the Cache
  - The CPU uses virtual addresses that must be mapped to a physical address
  - The cache may either use virtual or physical addresses.
    - A cache that indexes by virtual addresses is called a virtual cache , as opposed to a physical cache .
  - Address translation can be done in parallel with cache access, so penalties for misses are reduced as well.
    - Avoid address translation during indexing
  - A virtual cache reduces hit time since a translation from a virtual address to a physical address is not necessary on hits

a virtual address to a physical address



- 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

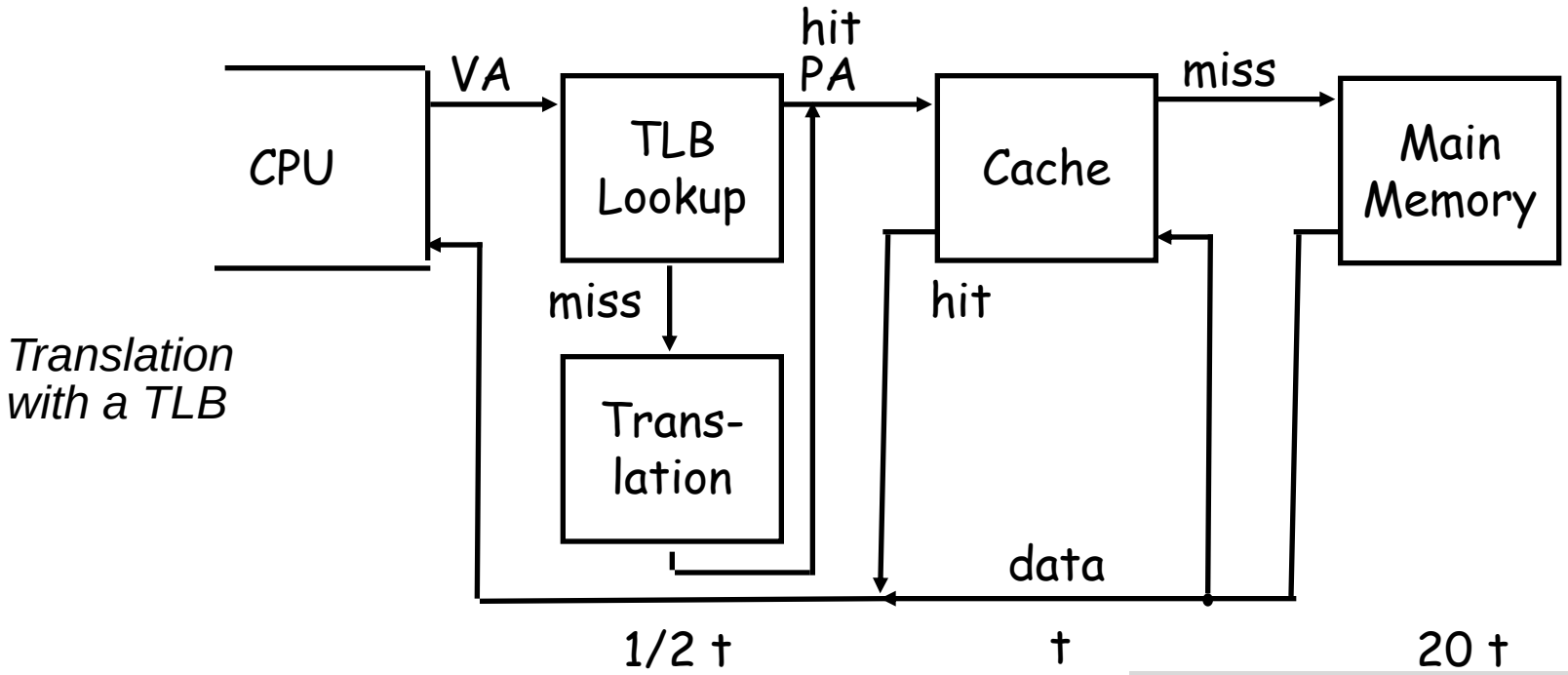
Really just a cache on the page table mappings

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

Translation Look-Aside Buffers

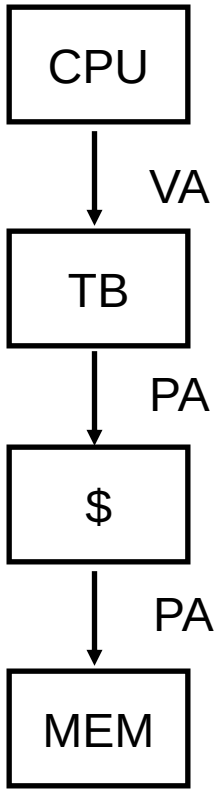
Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

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.

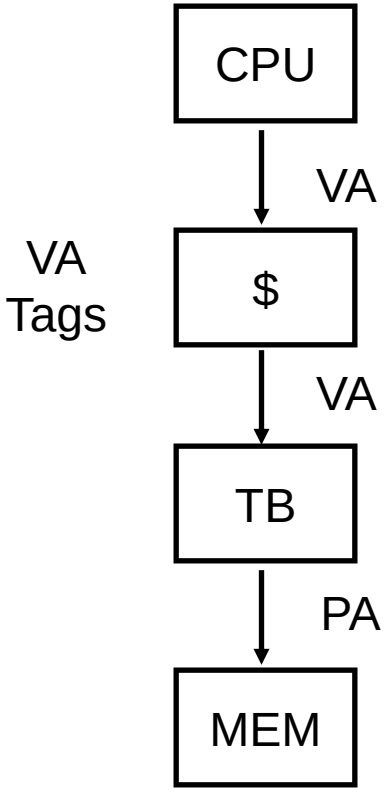


Translation with a TLB

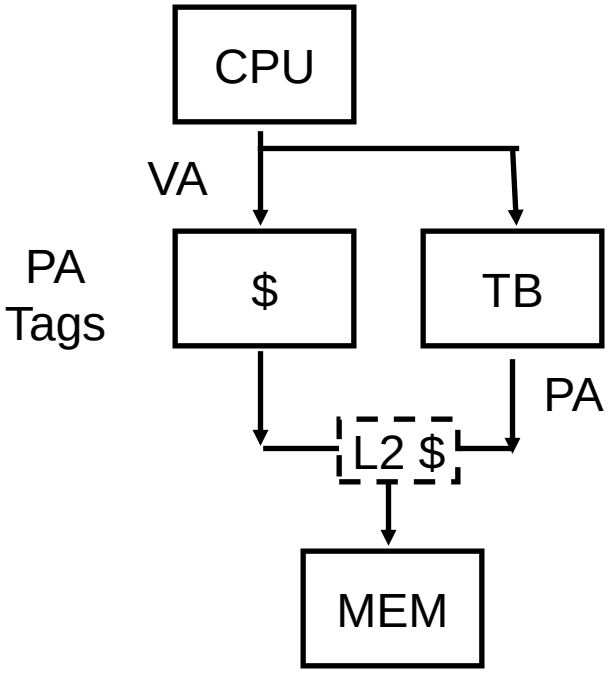
Fast hits by Avoiding Address Translation



Conventional Organization



Virtually Addressed Cache
Translate only on miss
Synonym Problem

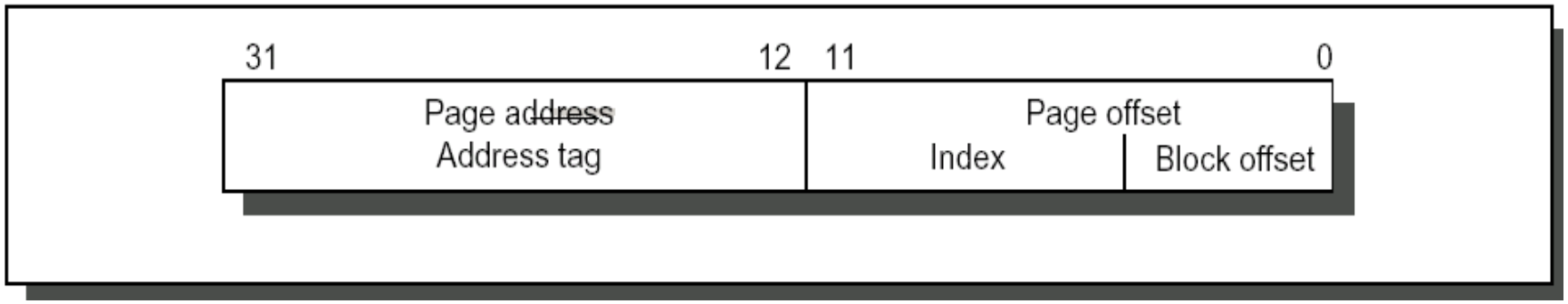


Overlap $ access with VA translation: requires $ index to remain invariant across translation

Fast Cache Hits by Avoiding Translation:

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

- Limits cache to page size: what if want bigger caches and uses same trick?
  - Higher associativity moves barrier to right
  - Page coloring



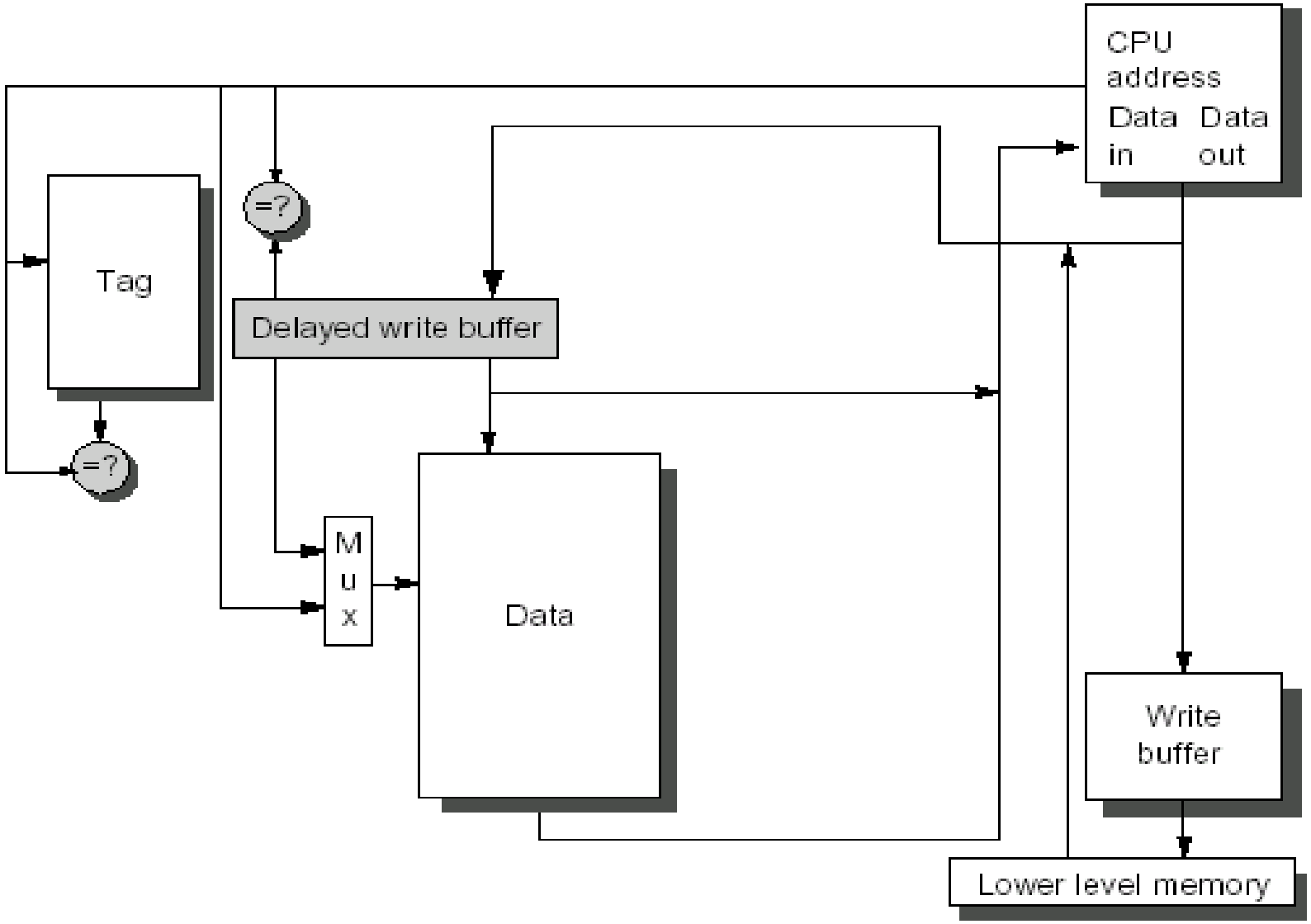
Fast hits by Avoiding Address Translation

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

- Dealing with *aliases* (sometimes called *synonyms*);
  Two different virtual addresses map to same physical address
  - solve by fiat: 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 <= 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

Third Hit Time Reduction Technique:

- Pipeli
  - The
    sta
  - Wri
    is v
  - Allo



the first

the data

Fourth Hit Time Reduction Technique:

- Trace caches
  - Instead of limiting the instructions in a static cache block to spatial locality, a trace cache finds a dynamic sequence of instructions including taken branches to load a cache block.
  - The cache blocks contains dynamic traces of the executed instructions as determined by CPU rather than containing static sequences of instructions as determined by memory.
    - The branch prediction is folded into the cache and must be validated alone with this addresses to have a valid fetch.

Summary: Reducing Hit Time

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

1.Reduce Hit Time via Small and Simple Caches

2.Reduce Hit Time via Avoiding Address Translation during Indexing of the Cache

3. Reduce Hit Time via pipelined Cache Access

4. Reduce Hit Time via Trace caches

Cache Optimization Summary

	Technique	MP	MR	HT	Complexity
miss penalty	1.Multilevel caches	+			2
	2.Early Restart & Critical Word 1st	+			2
	3.Priority to Read Misses	+			1
	4.Merging write buffer	+			1
	5.Victim caches	+	+		2
miss rate	6.Larger block size	-	+		0
	7.Larger cache size		+	-	1
	8.Higher Associativity		+	-	1
	9.Way-predicting cache and Pseudo-associative caches		+		2
	10.Compiler techniques reduce cache misses		+		0
parallelism	11.Non-Blocking Caches	+	+		3
	12.HW Prefetching of Instr/Data	+	+	2instr./3data	
	13.Compiler Controlled Prefetching	+	+		3
hit time	14.Small & Simple Caches		-	+	**0**
	15.Avoiding Address Translation			+	2
	16.Pipelined Cache Access			+	1
	17.Trace cache			+	3

- To be continued

THANK YOU