

IMPROVING CACHE PERFORMANCE



Dr Noor Mahammad Sk

Improving Cache Performance

- *Average-memory-access-time = Hit-time + Miss-rate * Miss-penalty*
- Strategies for improving cache performance
 - ▣ Reducing the miss penalty
 - ▣ Reducing the miss rate
 - ▣ Reducing the miss penalty or miss rate via parallelism
 - ▣ Reducing the time to hit in the cache



Reducing Cache Miss Penalty

Techniques for Reducing Miss Penalty



- ❑ Multilevel Caches (the most important)
- ❑ Critical Word First and Early Restart
- ❑ Giving Priority to Read Misses over Writes
- ❑ Merging Write Buffer
- ❑ Victim Caches

Multi-Level Caches

- Probably the best miss-penalty reduction
- Performance measurement for 2-level caches
 - ▣ $AMAT = \text{Hit-time-L1} + \text{Miss-rate-L1} * \text{Miss-penalty-L1}$
 - ▣ $\text{Miss-penalty-L1} = \text{Hit-time-L2} + \text{Miss-rate-L2} * \text{Miss-penalty-L2}$
 - ▣ $AMAT = \text{Hit-time-L1} + \text{Miss-rate-L1} * (\text{Hit-time-L2} + \text{Miss-rate-L2} * \text{Miss-penalty-L2})$

Multi-Level Caches (Cont.)

□ 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 CPU (Miss-rate-L1 x Miss-rate-L2)
- ▣ **Global Miss Rate is what matters**

□ Advantages:

- ▣ Capacity misses in L1 end up with a significant penalty reduction since they likely will get supplied from L2
 - No need to go to main memory
- ▣ Conflict misses in L1 similarly will get supplied by L2

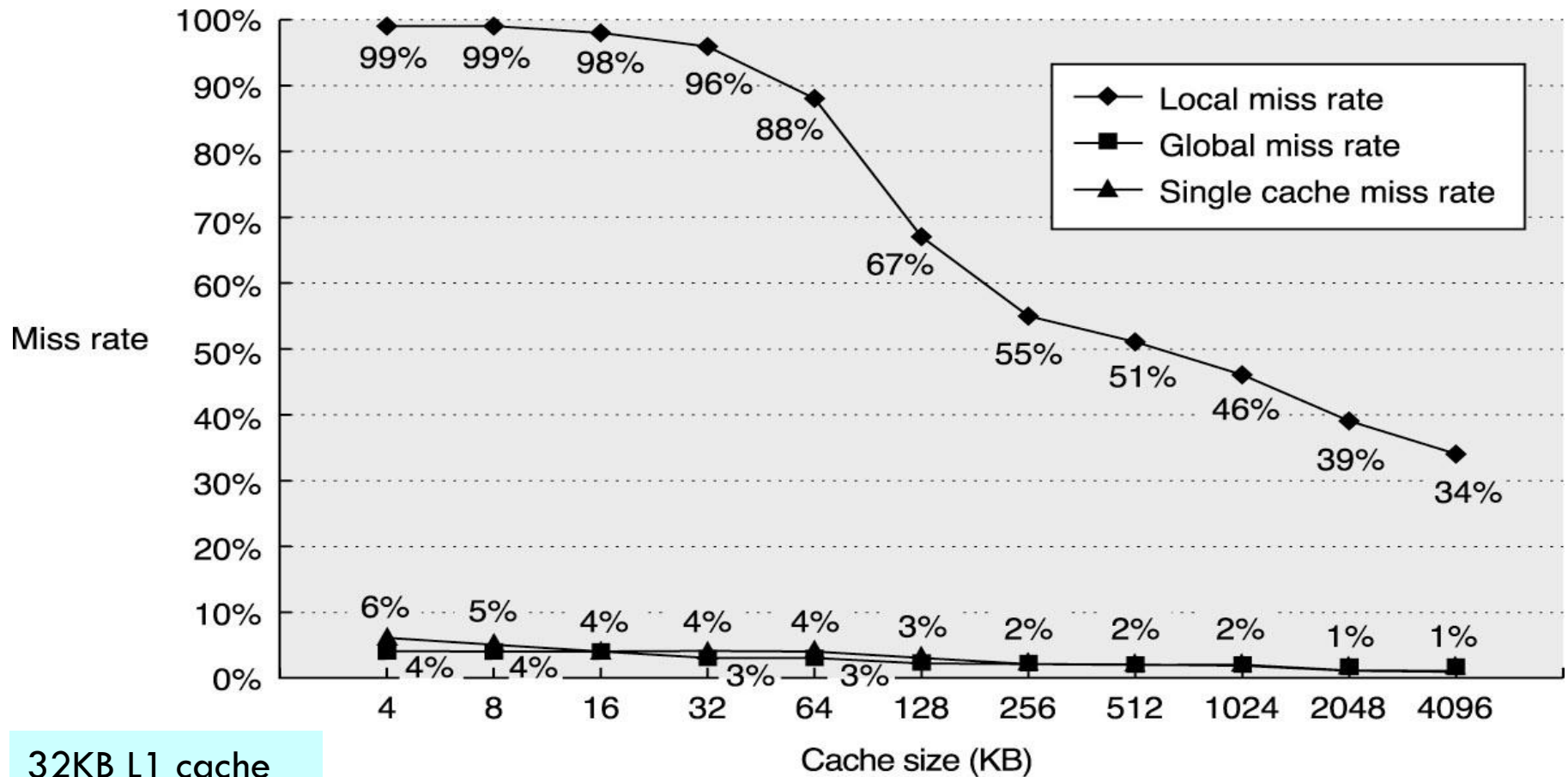
Miss Rate Example

- Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache
 - ▣ Miss rate for the first-level cache = $40/1000$ (4%)
 - ▣ Local miss rate for the second-level cache = $20/40$ (50%)
 - ▣ Global miss rate for the second-level cache = $20/1000$ (2%)

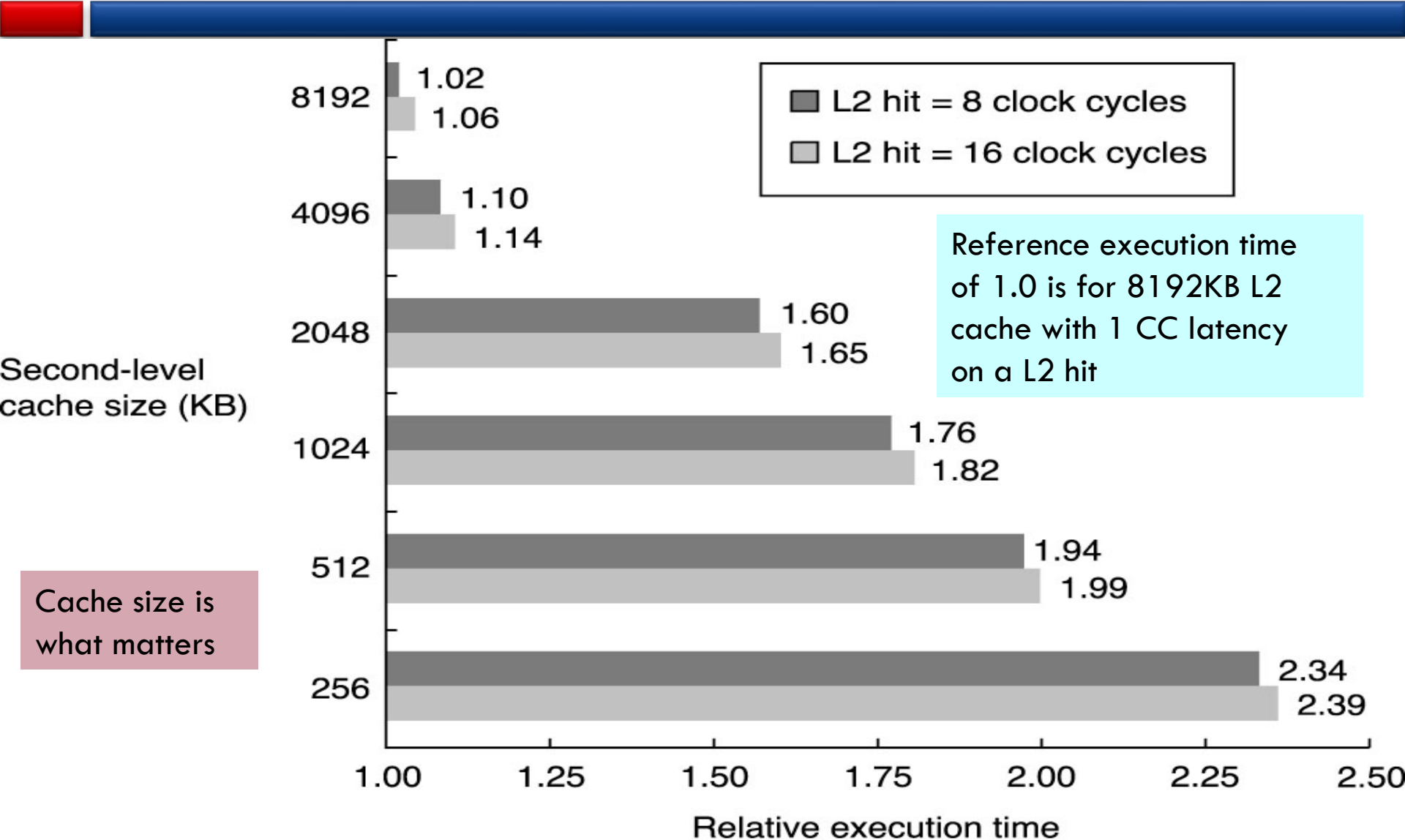
Miss Rate Example (Cont.)

- Assume miss-penalty-L2 is 100 CC, hit-time-L2 is 10 CC, hit-time-L1 is 1 CC, and 1.5 memory reference per instruction. What is average memory access time and average stall cycles per instructions? Ignore writes impact.
 - ▣ $AMAT = \text{Hit-time-L1} + \text{Miss-rate-L1} * (\text{Hit-time-L2} + \text{Miss-rate-L2} * \text{Miss-penalty-L2}) = 1 + 4\% * (10 + 50\% * 100) = 3.4 \text{ CC}$
 - ▣ $\text{Average memory stalls per instruction} = \text{Misses-per-instruction-L1} * \text{Hit-time-L2} + \text{Misses-per-instructions-L2} * \text{Miss-penalty-L2}$
 $= (40 * 1.5 / 1000) * 10 + (20 * 1.5 / 1000) * 100 = 3.6 \text{ CC}$
 - Or $(3.4 - 1.0) * 1.5 = 3.6 \text{ CC}$

Comparing Local and Global Miss Rates



Relative Execution Time by L2-Cache Size



Comparing Local and Global Miss Rates

- Huge 2nd level caches
- Global miss rate close to single level cache rate provided $L2 \gg L1$
- Global cache miss rate should be used when evaluating second-level caches (or 3rd, 4th, ... levels of hierarchy)
- Many fewer hits than L1, target **reduce misses**

Impact of L2 Cache Associativity

- Hit-time-L2
 - ▣ Direct mapped = 10 CC; 2-way set associativity = 10.1 CC (usually round up to integral number of CC, 10 or 11 CC)
- Local-miss-rate-L2
 - ▣ Direct mapped = 25%; 2-way set associativity = 20%
- Miss-penalty-L2 = 100CC
- Miss-penalty-L2
 - ▣ Direct mapped = $10 + 25\% * 100 = 35$ CC
 - ▣ 2-way (10 CC) = $10 + 20\% * 100 = 30$ CC
 - ▣ 2-way (11 CC) = $11 + 20\% * 100 = 31$ CC

Critical Word First and Early Restart

- Do not 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
- Benefits of critical word first and early restart depend on
 - ▣ **Block size**: generally useful only in large blocks
 - ▣ Likelihood of another access to the portion of the block that has not yet been fetched
 - **Spatial locality problem**: tend to want next sequential word, so not clear if benefit



Giving Priority to Read Misses Over Writes

- In write through, write buffers complicate memory access in that they might hold the updated value of location needed on a read miss
 - RAW conflicts with main memory reads on cache misses
- **Read miss waits until the write buffer empty** → increase read miss penalty (old MIPS 1000 with 4-word buffer by 50%)
- **Check write buffer contents before read**, and if no conflicts, let the memory access continue
- Write Back?
 - 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

```
SW R3, 512(R0)
LW R1, 1024(R0)
LW R2, 512(R0)
```

Merging Write Buffer

- An entry of write buffer often contain multi-words. However, a write often involves single word
 - ▣ A single-word write occupies the whole entry if no write-merging
- Write merging: check to see if the address of a new data matches the address of a valid write buffer entry. If so, the new data are combined with that entry
- Advantage
 - ▣ Multi-word writes are usually faster than single-word writes
 - ▣ Reduce the stalls due to the write buffer being full

Write-Merging Illustration

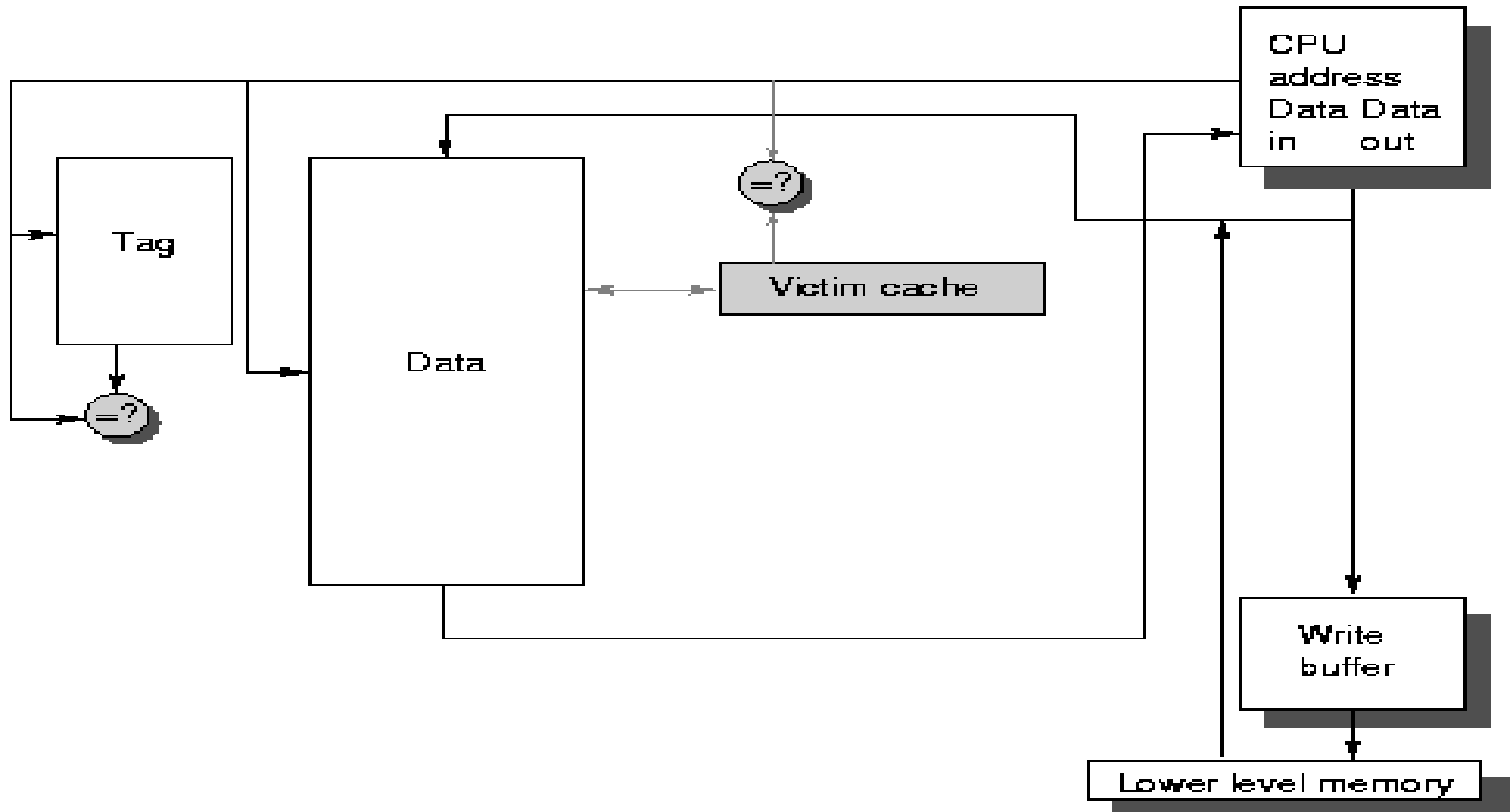
Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

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	

Victim Caches

- Remember what was just discarded in case it is need again
- Add small fully associative cache (called *victim cache*) between the cache and the refill path
 - ▣ Contain only blocks discarded from a cache because of a miss
 - ▣ Are checked on a miss to see if they have the desired data before going to the next lower-level of memory
 - If yes, swap the victim block and cache block
 - ▣ Addressing both victim and regular cache at the same time
 - The penalty will not increase
- Jouppi (DEC SRC) shows miss reduction of 20 - 95%
 - ▣ For a 4KB direct mapped cache with 1-5 victim blocks

Victim Cache Organization



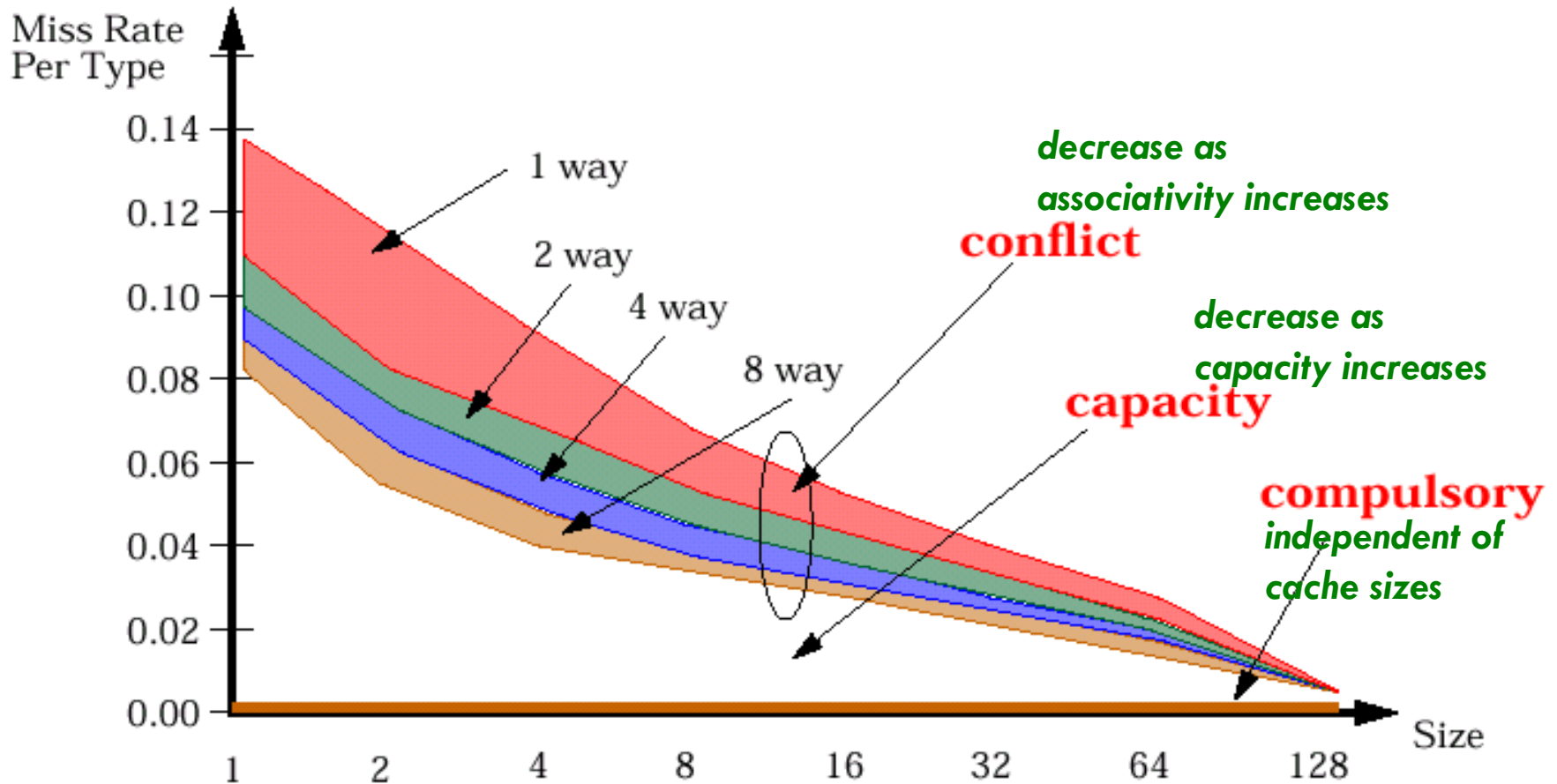


Reducing Miss Rate

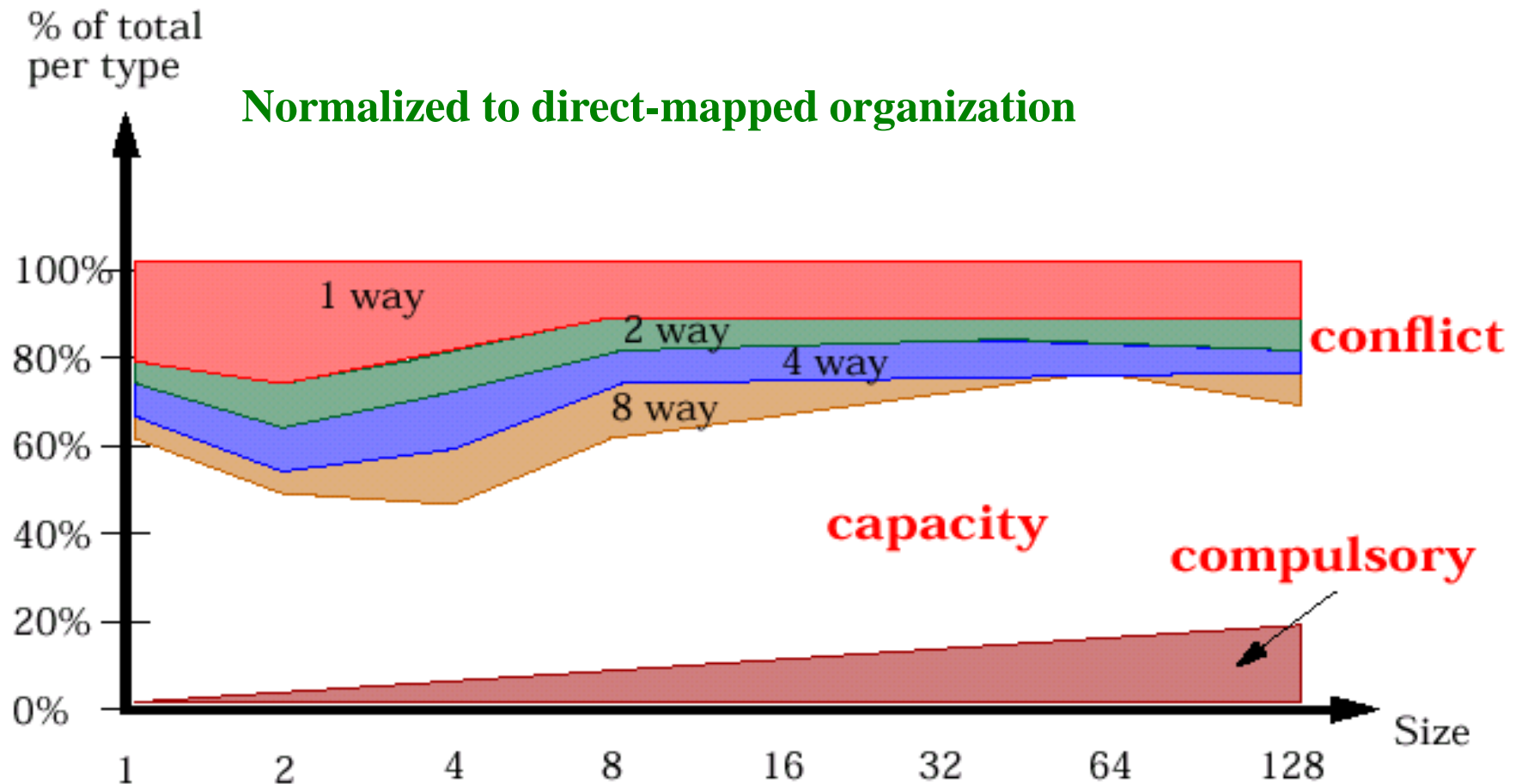
Classify Cache Misses - 3 C's

- **Compulsory** → independent of cache size
 - ▣ First access to a block → no choice but to load it
 - ▣ Also called *cold-start* or *first-reference* misses
 - ▣ Measured by a infinite cache (ideal)
- **Capacity** → decrease as cache size increases
 - ▣ Cache cannot contain all the blocks needed during execution, then blocks being discarded will be later retrieved
 - ▣ Measured by a fully associative cache
- **Conflict (Collision)** → decrease as associativity increases
 - ▣ Side effect of set associative or direct mapping
 - ▣ A block may be discarded and later retrieved if too many blocks map to the same cache block

Miss Distributions vs. the 3 C's (Total Miss Rate)



Miss Distributions



Techniques for Reducing Miss Rate



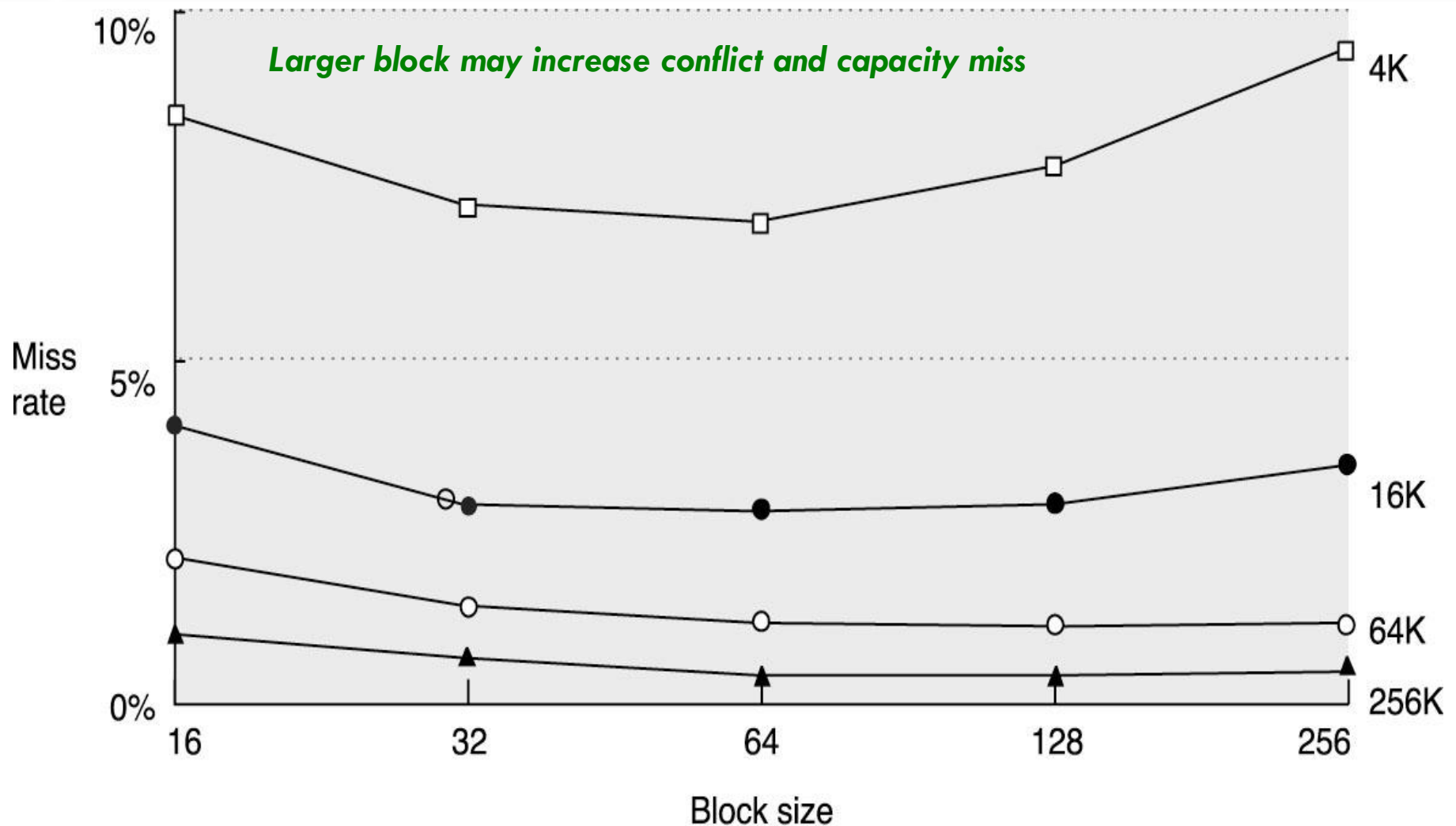
- ❑ Larger Block Size
- ❑ Larger Caches
- ❑ Higher Associativity
- ❑ Way Prediction and Pseudo-associative Caches
- ❑ Compiler optimizations

Larger Block Sizes

- Obvious advantages: reduce compulsory misses
 - ▣ Reason is due to spatial locality
- Obvious disadvantage
 - ▣ **Higher miss penalty:** larger block takes longer to move
 - ▣ May increase conflict misses and capacity miss if cache is small

Don't let increase in miss penalty outweigh the decrease in miss rate

Miss Rate Vs Block Size



Actual Miss Rate Vs. 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%

- Actual miss rate versus block size for the four different-sized caches
- Note that for a 4 KiB cache, 256-byte blocks have a higher miss rate than 32byte blocks.
- In this example, the cache would have to be 256 KiB in order for a 256-byte block to decrease misses.

Miss Rate VS. Miss Penalty

- Assume memory system takes 80 CC of overhead and then deliver 16 bytes every 2 CC. Hit time = 1 CC
- Miss penalty
 - ▣ Block size 16 = $80 + 2 = 82$
 - ▣ Block size 32 = $80 + 2 * 2 = 84$
 - ▣ Block size 256 = $80 + 16 * 2 = 112$
- $AMAT = hit_time + miss_rate * miss_penalty$
 - ▣ 256-byte in a 256 KB cache = $1 + 0.49\% * 112 = 1.549$ CC

AMAT VS. Block Size for Different-Size 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

- Average memory access time versus block size for four different-sized cache.
- Block sizes of 32 and 64 bytes dominate.
- The smallest average time per cache size is boldfaced.

Large Caches



- Help with both conflict and capacity misses
- May need longer hit time and/or higher HW cost
- Popular in off-chip caches

Higher Associativity

- 8-way set associative is for practical purposes as effective in reducing misses as fully associative
- 2: 1 Rule of thumb
 - ▣ 2 way set associative of size $N/2$ is about the same as a direct mapped cache of size N (held for cache size < 128 KB)
- Greater associativity comes at the cost of increased hit time
 - ▣ Lengthen the clock cycle
 - ▣ Hill [1988] suggested hit time for 2-way vs. 1-way: external cache +10%, internal + 2%

Effect of Higher Associativity for AMAT

Clock-cycle-time (2-way) = 1.10 * Clock-cycle-time (1-way)

Clock-cycle-time (4-way) = 1.12 * Clock-cycle-time (1-way)

Clock-cycle-time (8-way) = 1.14 * Clock-cycle-time (1-way)

Cache size (KiB)	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	2.23	2.40	2.46	2.53
32	2.06	2.30	2.37	2.45
64	1.92	2.14	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

Average memory access time using miss rates

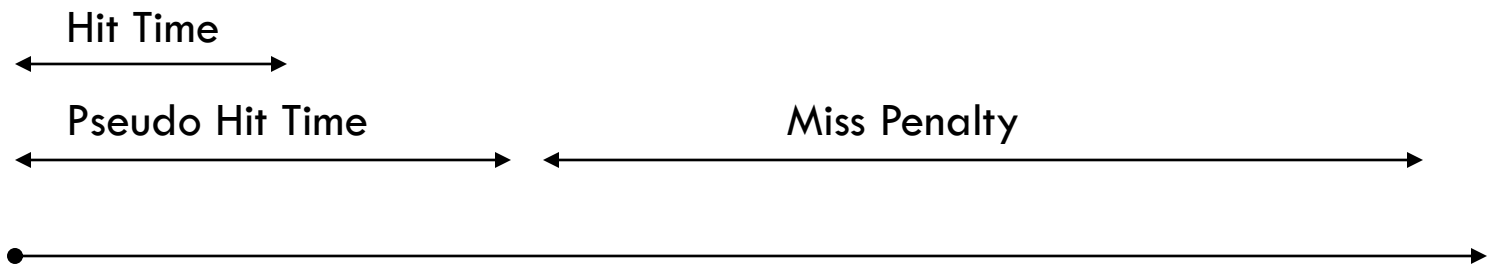
Way Prediction

- ❑ Extra bits are kept in cache to predict the way, or block within the set of the next cache access
- ❑ Multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle
- ❑ A miss results in checking the other blocks for matches in subsequent clock cycles
- ❑ Alpha 21264 uses way prediction in its 2-way set-associative instruction cache. Simulation using SPEC95 suggested way prediction accuracy is in excess of 85%

Pseudo-Associative Caches

- Attempt to get the miss rate of set-associative caches and the hit speed of direct-mapped cache
- Idea
 - ▣ Start with a direct mapped cache
 - ▣ On a miss check another entry
 - ▣ Usual method is to invert the high order index bit to get the next try
 - 010111 → 110111
- Problem - fast hit and slow hit
 - ▣ May have the problem that you mostly need the slow hit
 - ▣ In this case it is better to swap the blocks
- 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

Relationship Between a Regular Hit Time, Pseudo Hit Time and Miss Penalty



Effect of Pseudo-Associative Caches

- Assume that it takes two extra cycles to find the entry in the alternative location (1 to check and 1 to swap)
- $AMAT = \text{Hit-time} + \text{Miss-rate} * \text{Miss-penalty}$
 - ▣ Miss-penalty is 1 cycle more than a normal miss penalty (why??)
 - ▣ $\text{Miss-rate} * \text{Miss-penalty} = \text{Miss-rate}(2\text{-way}) * \text{Miss-penalty}(1\text{-way})$
 - ▣ $\text{Hit-time} = \text{Hit-time}(1\text{-way}) + \text{Alternate_hit_rate} * 2$
 - ▣ $\text{Alternate-hit-rate} = \text{Hit-rate}(2\text{-way}) - \text{Hit-rate}(1\text{-way}) = \text{Miss-rate}(1\text{-way}) - \text{Miss-rate}(2\text{-way})$
- $AMAT(\text{pseudo}) = 4.950 \text{ (2K)}, 1.371 \text{ (128K)}$
- $AMAT(1\text{-way}) = 5.90 \text{ (2K)}, 1.50 \text{ (128K)}$
- $AMAT(2\text{-way}) = 4.90 \text{ (2K)}, 1.45 \text{ (128K)}$

Compiler Optimization for Code

- Code can easily be arranged without affecting correctness
- Reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses
- McFarling's observation using profiling information [1988]
 - Reduce miss by 50% for a 2KB direct-mapped instruction cache with 4-byte blocks, and by 75% in an 8KB cache
 - Optimized programs on a direct-mapped cache missed less than unoptimized ones on an 8-way set-associative cache of same size

Compiler Optimization for Data

- Idea – improve the spatial and temporal locality of the data
- Lots of options
 - ▣ **Array merging** – Allocate arrays so that paired operands show up in same cache block
 - ▣ **Loop interchange** – Exchange inner and outer loop order to improve cache performance
 - ▣ **Loop fusion** – For independent loops accessing the same data, fuse these loops into a single aggregate loop
 - ▣ **Blocking** – Do as much as possible on a sub- block before moving on

Merging Arrays Example

```
/* Before: 2 sequential arrays */
```

```
int val[SIZE];
```

```
int key[SIZE];
```

val	key
-----	-----

```
/* After: 1 array of stuctures */
```

```
struct merge {
```

```
    int val;
```

```
    int key;
```

```
};
```

```
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key; improve spatial locality

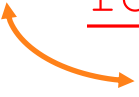
Loop Interchange Example

```
/* Before */
```

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

```
/* After */
```

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



Sequential accesses instead of striding through memory every 100 words;
improve spatial locality

Loop Fusion Example

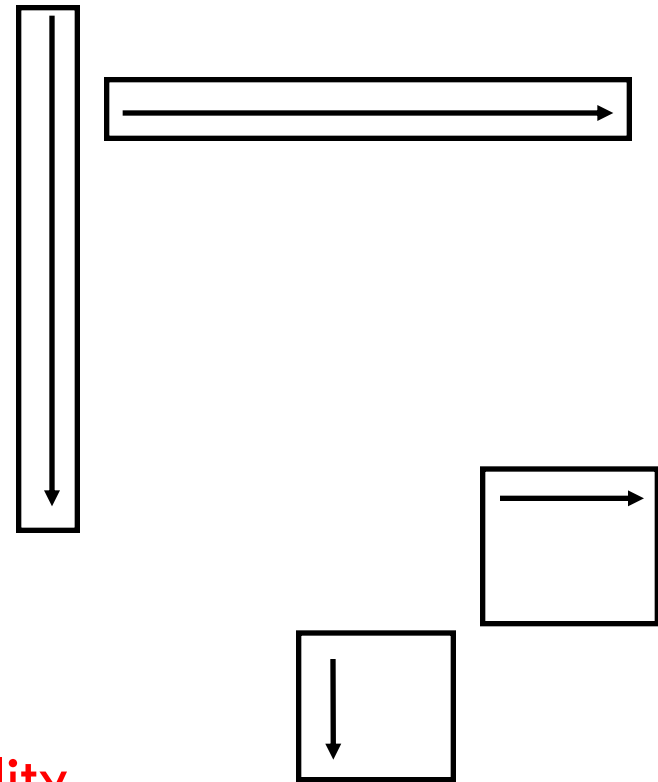
```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        a[i][j]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

Perform different computations on the common data in two loops → fuse the two loops

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

Blocking Example

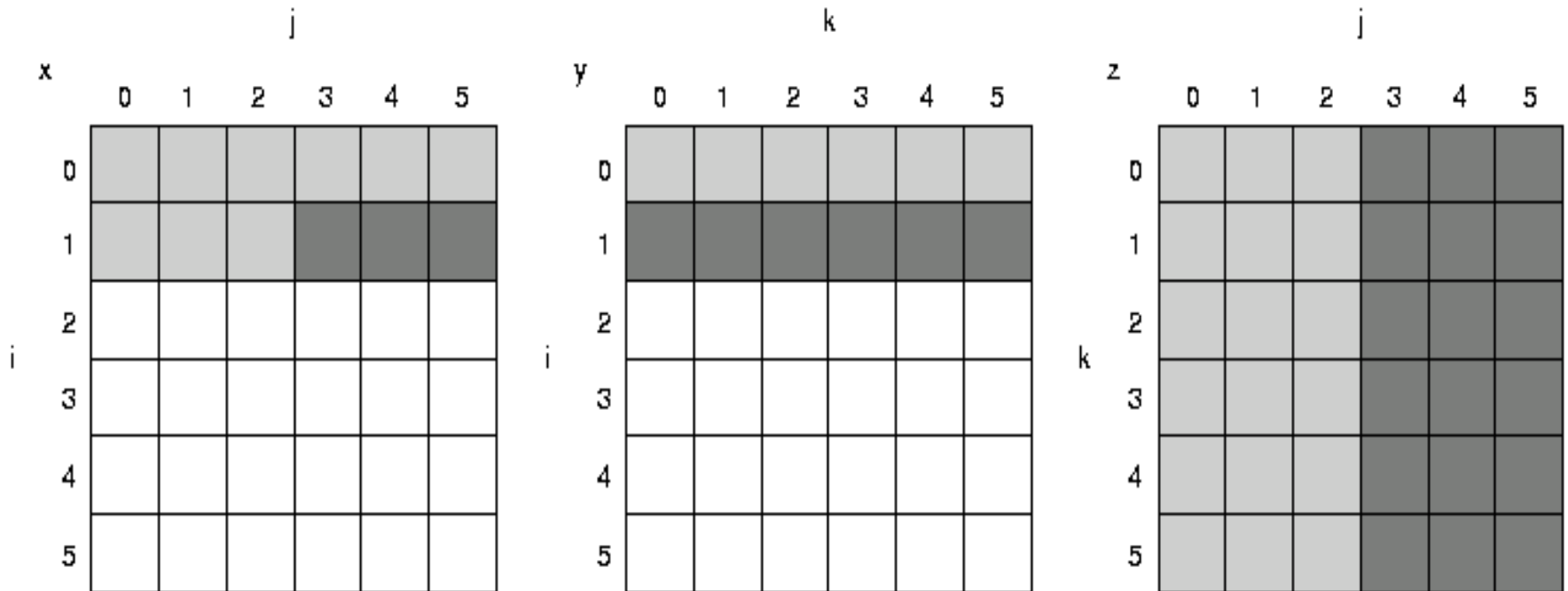
```
/* Before */  
for (i = 0; i < N; i = i+1)  
  for (j = 0; j < N; j = j+1)  
    {r = 0;  
      for (k = 0; k < N; k = k+1) {  
        r = r + y[i][k]*z[k][j];}  
      x[i][j] = r;  
    };
```



Improve temporal locality + spatial locality

Snapshot of x , y , z when $i=1$

(Figure 5.21)



White: not yet touched

Light: older access

Dark: newer access

Blocking Example (Cont.)

- Dealing with multiple arrays, with some arrays accessed by rows and some by columns
 - ▣ Row-major or column-major-order no help → loop interchange no help
- Idea: compute on BxB **submatrix** that fits
- 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 $x[]$
- Capacity Misses a function of N & Cache Size:
 - ▣ $3 N \times N \times 4 \rightarrow$ no capacity misses; otherwise ...

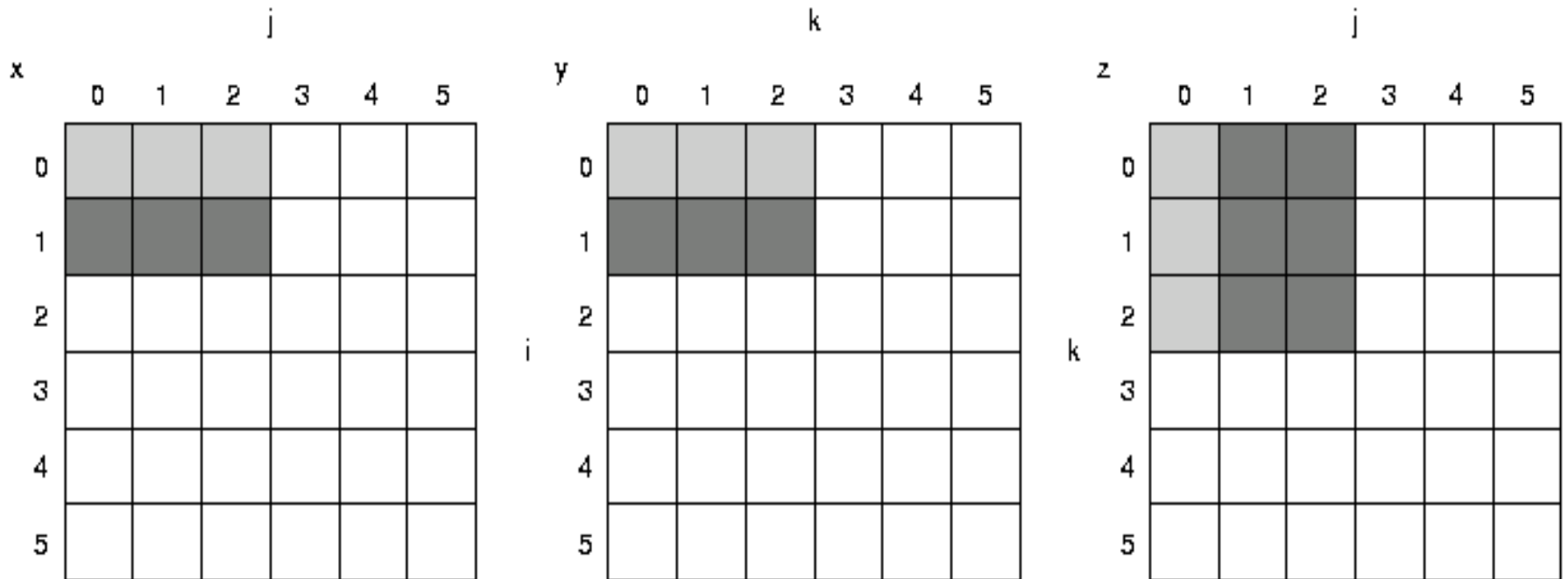
Blocking Example (Cont.)

```
/* 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*
- Worst-case capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$
- Help register allocation

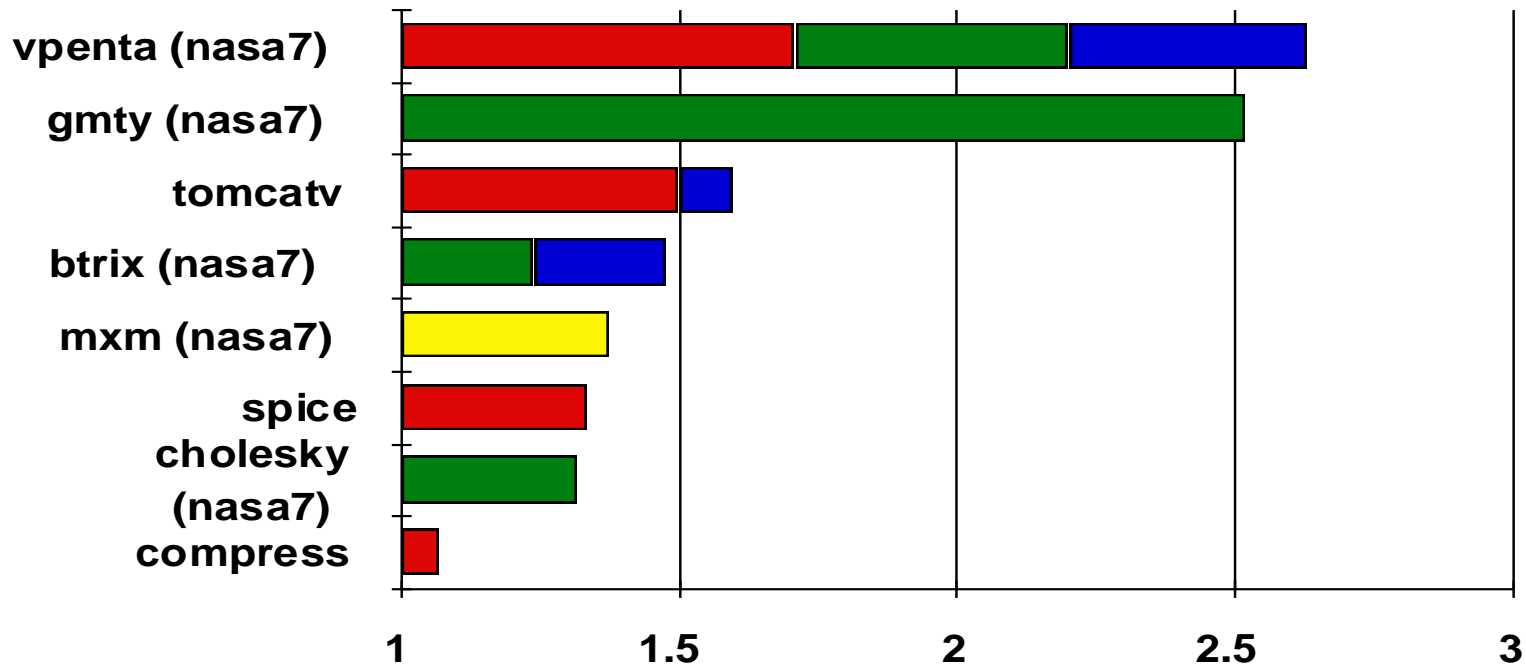
The Age of Accesses to x , y , z

(Figure 5.22)



Note in contrast to Figure 5.21, the smaller number of elements accessed

Summary of Compiler Optimizations to Reduce Cache Misses



Performance Improvement





Reducing Cache Miss Penalty or Miss Rate Via Parallelism

Overview



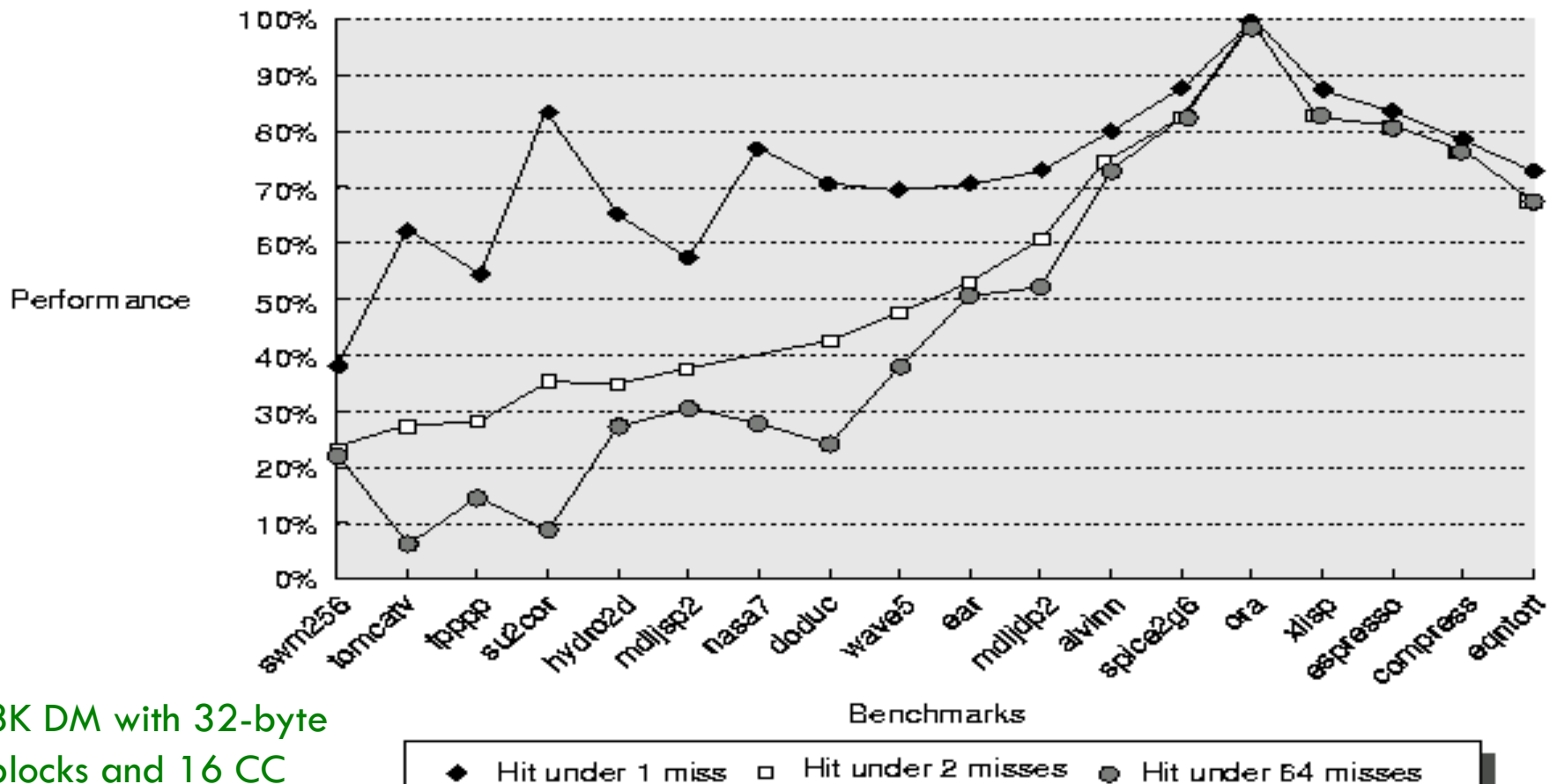
- Overlap the execution of instructions with activity in the memory hierarchy
- Techniques
 - ▣ Non-blocking caches to reduce stalls on cache misses – help with out-of-order processors
 - ▣ Hardware prefetch of instructions and data
 - ▣ Compiler-controlled prefetching

Non-blocking Caches

- **Non-blocking cache** or **lockup-free cache** allow data cache to continue to supply cache hits during a miss
 - ▣ Requires out-of-order execution CPU, like scoreboard or Tomasulo
- **Hit-under-miss**: reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- **Hit-under-multiple-miss** or **miss-under-miss**: may further lower the effective penalty by overlapping multiple misses
 - ▣ Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
 - ▣ Requires multiple memory banks (otherwise cannot support)
 - ▣ Pentium Pro allows 4 outstanding memory misses

Effect of Non-blocking Cache

Ratio of the average memory stall time (Compare with blocking cache)



8K DM with 32-byte
blocks and 16 CC
penalty

Example

- Compare 2-way set-associativity or hit-under-one-miss under 8KB data caches
 - ▣ FP miss rate: 11.4% (direct-mapped), 10.7% for (2-way)
 - ▣ INT miss rate: 7.4% (direct-mapped), 6.0% for (2-way)
- FP ($\text{Miss_rate} * \text{Miss_penalty}$)
 - ▣ Direct-mapped: $11.4\% * 16 = 1.84$
 - ▣ 2-way: $10.7\% * 16 = 1.71$
 - ▣ $1.71/1.84 = 93\% \rightarrow$ hit-under-one-miss is better
- Integer ($\text{Miss_rate} * \text{Miss_penalty}$)
 - ▣ Direct-mapped: $7.4\% * 16 = 1.18$
 - ▣ 2-way: $6.0\% * 16 = 0.96$
 - ▣ $0.96/1.18 = 81\% \rightarrow$ Almost the same

Hit-under-miss does not
increase hit time

Non-Blocking Cache (Cont.)

- Difficult to evaluate performance of non-blocking caches
 - ▣ A cache miss does not necessarily stall the CPU
 - ▣ Effective miss penalty is the nonoverlapped time that CPU is stalled
 - ▣ Difficult to judge the impact of any single miss
 - ▣ Difficult to calculate AMAT
- Out-of-order CPUs are capable of hiding the miss penalty of L1 data cache that hits in L2, but cannot hide a significant fraction of an L2 cache miss
- Possible to be more than one miss requests to same block
 - ▣ Must check on misses to be sure it is not to a block already being requested to avoid possible inconsistency and to save time

Hardware Prefetching of I&D

- Use hardware other than the cache to prefetch what you expect to need ahead of time
- AXP 21064 I-fetches 2 blocks on a miss
 - ▣ Target block goes to the I-cache
 - ▣ Next block goes to the instruction stream buffer (ISB)
 - ▣ If requested block is in the ISB then it moves to the I cache and next block only is promoted from the next lower level.
 - ▣ 1, 4, 16 block ISB catches 15-25%, 50%, 72% of the misses
- Works with data blocks too:
 - ▣ Jouppi: 1 DSB 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 (otherwise would be unused)

Effect of HW Prefetching

- $AMAT(\text{prefetch}) = \text{Hit-time} + \text{Miss-rate} * \text{Prefetch-hit-rate} * \text{prefetch-hit-time} + \text{Miss-rate} * (1 - \text{Prefetch-hit-rate}) * \text{Miss-penalty}$
- Parameters
 - ▣ Prefetch-hit-time: 1 clock cycle; prefetch hit rate: 25%
 - ▣ Miss rate: 1.10%(8KB cache); Hit time: 2 clock cycle; Miss penalty: 50 clock cycle
- $AMAT(\text{prefetch}) = 2.41525$
- The miss rate of a cache without prefetching has to be 0.83%(8(1.10%) \leftrightarrow 16(0.64%)) to achieve the equivalent AMAT

Compiler-Controlled Prefetching

□ Data Prefetch

- ▣ **Register Prefetch**: load data into register (HP PA-RISC loads)
- ▣ **Cache Prefetch**: load into cache (MIPS IV, PowerPC, SPARC v. 9)
- ▣ Prefetch instruction example: *prefetch(b[j+7][0])*
- ▣ Special prefetching instructions cannot cause faults; a form of speculative execution
- ▣ Best candidates are loops
- ▣ Issuing Prefetch Instructions takes time
 - Is cost of prefetch issues < savings in reduced misses?

□ Also works for instruction prefetch



Reducing Hit Time

Reducing Hit Time

- Hit time is critical because it affects the clock cycle time
 - ▣ On many machines, cache access time limits the clock cycle rate
- A fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything
 - ▣ *Average-Memory-Access-Time = Hit-Access-Time + Miss-Rate * Miss-Penalty*
 - Miss-penalty is clock-cycle dependent

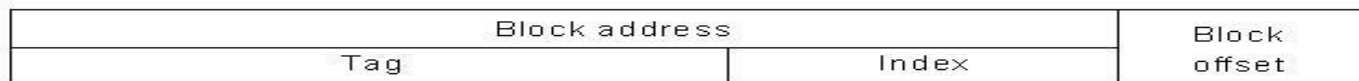
Techniques for Reducing Hit Time



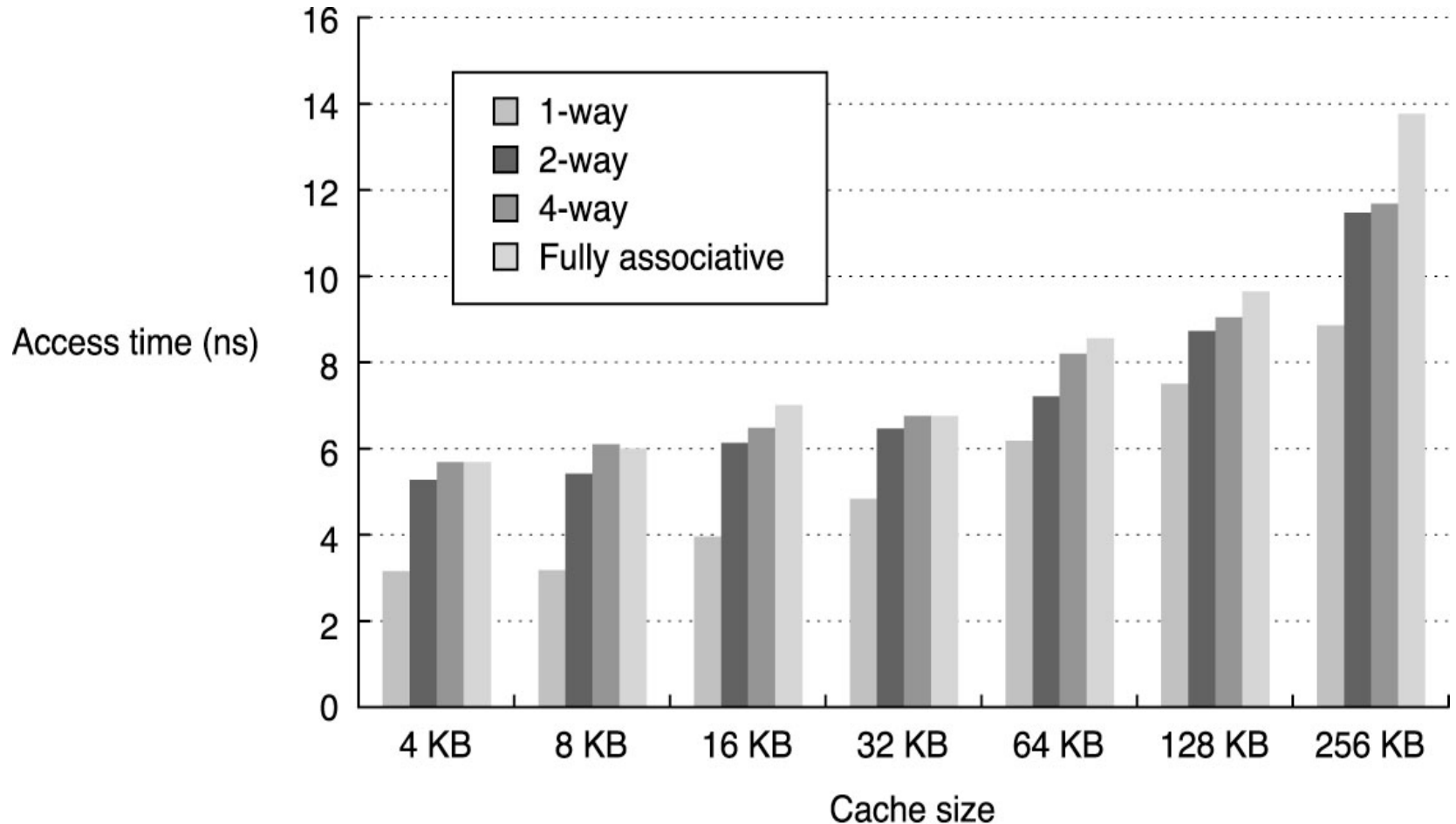
- ❑ Small and Simple Caches
- ❑ Avoid Address Translation during Indexing of the Cache
- ❑ Pipelined Cache Access
- ❑ Trace Caches

Small and Simple Caches

- A time-consuming portion of a cache hit: use the index portion to read the tag and then compare it to the address
- Small caches – smaller hardware is faster
 - ▣ Keep the L1 cache small enough to fit on the same chip as CPU
 - ▣ Keep the tags on-chip, and the data off-chip for L2 caches
- Simple caches – direct-Mapped cache
 - ▣ Trading hit time for increased miss-rate
 - Small direct mapped misses more often than small associative caches
 - But simpler structure makes the hit go faster



Access Time as Size and Associativity Vary in a CMOS Cache

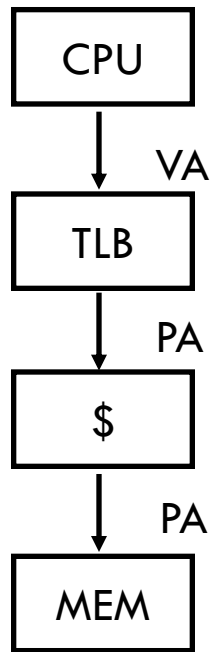


Virtual Addressed Caches

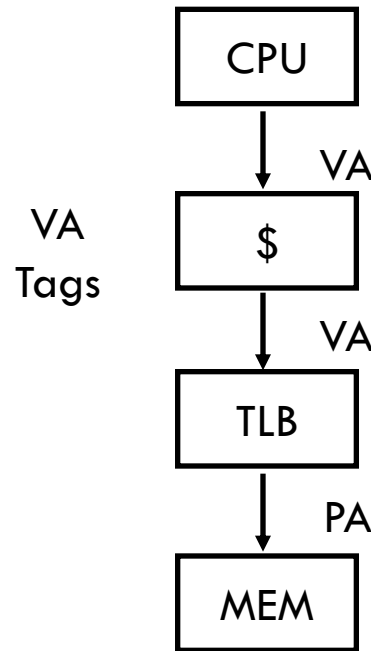
- Parallel rather than sequential access
 - ▣ **Physical addressed caches** access the TLB to generate the physical address, then do the cache access
- Avoid address translation during cache index
 - ▣ Implies **virtual addressed cache**
 - ▣ Address translation proceeds in parallel with cache index
 - If translation indicates that the page is not mapped - then the result of the index is not a hit
 - Or if a protection violation occurs - then an exception results
 - All is well when neither happen
- Too good to be true?

Virtually Addressed Caches

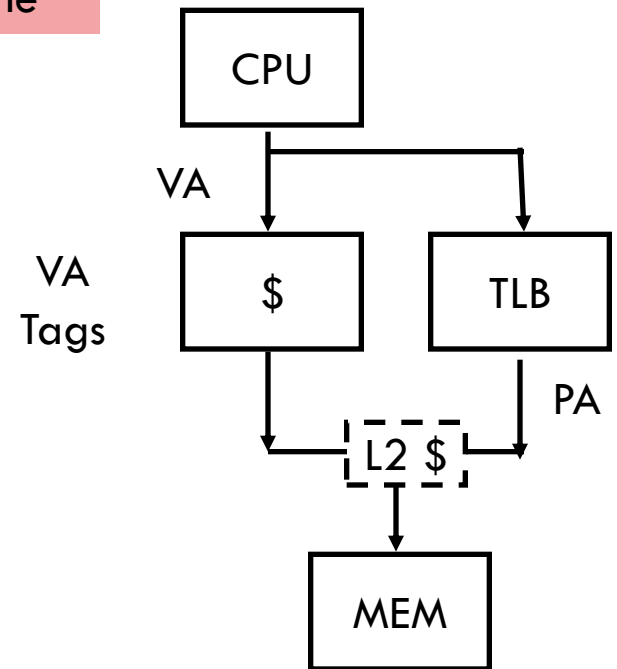
\$ means cache



Conventional Organization

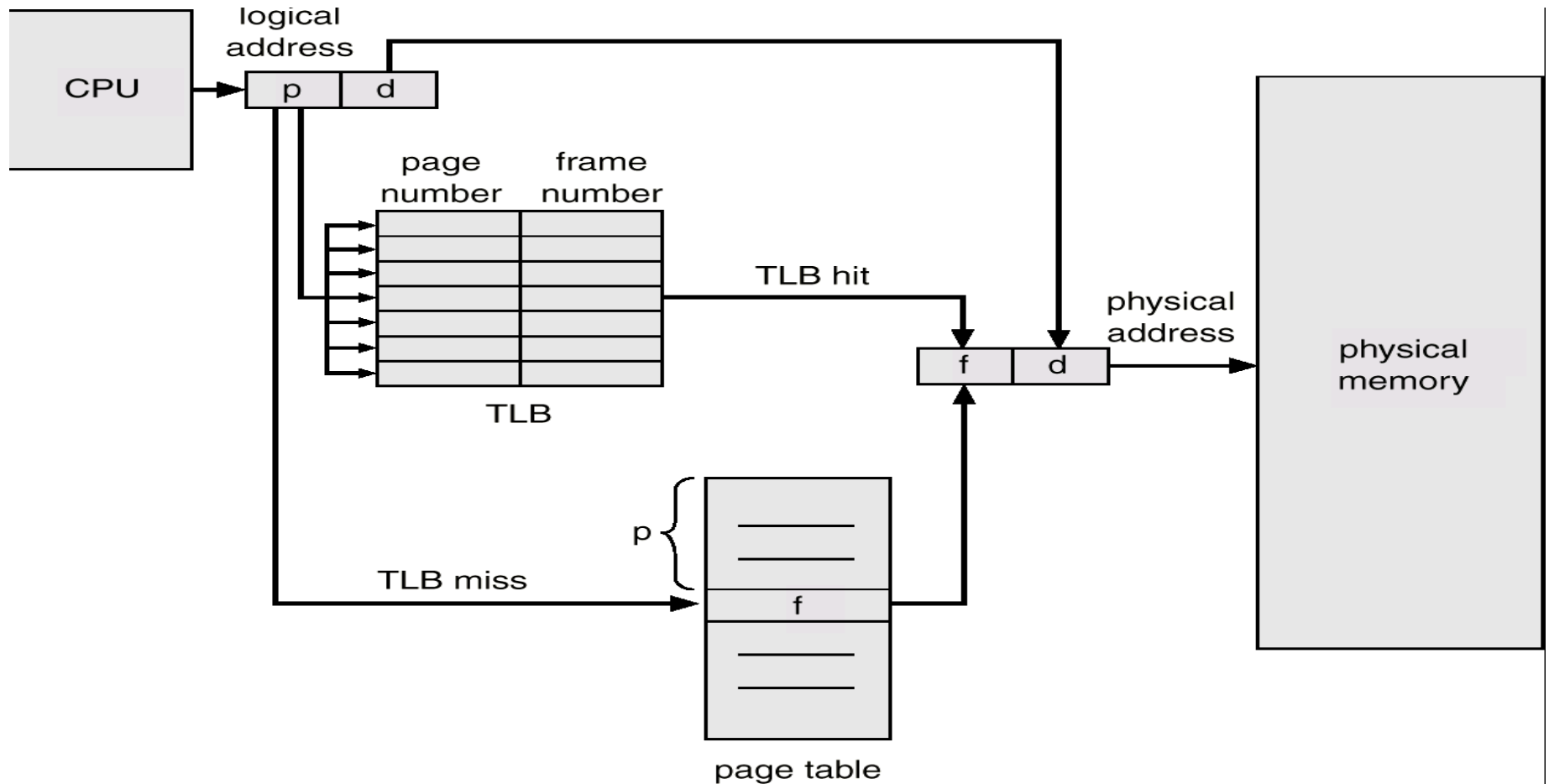


Virtually Addressed Cache
Translate only on miss
Synonym (Alias) Problem



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

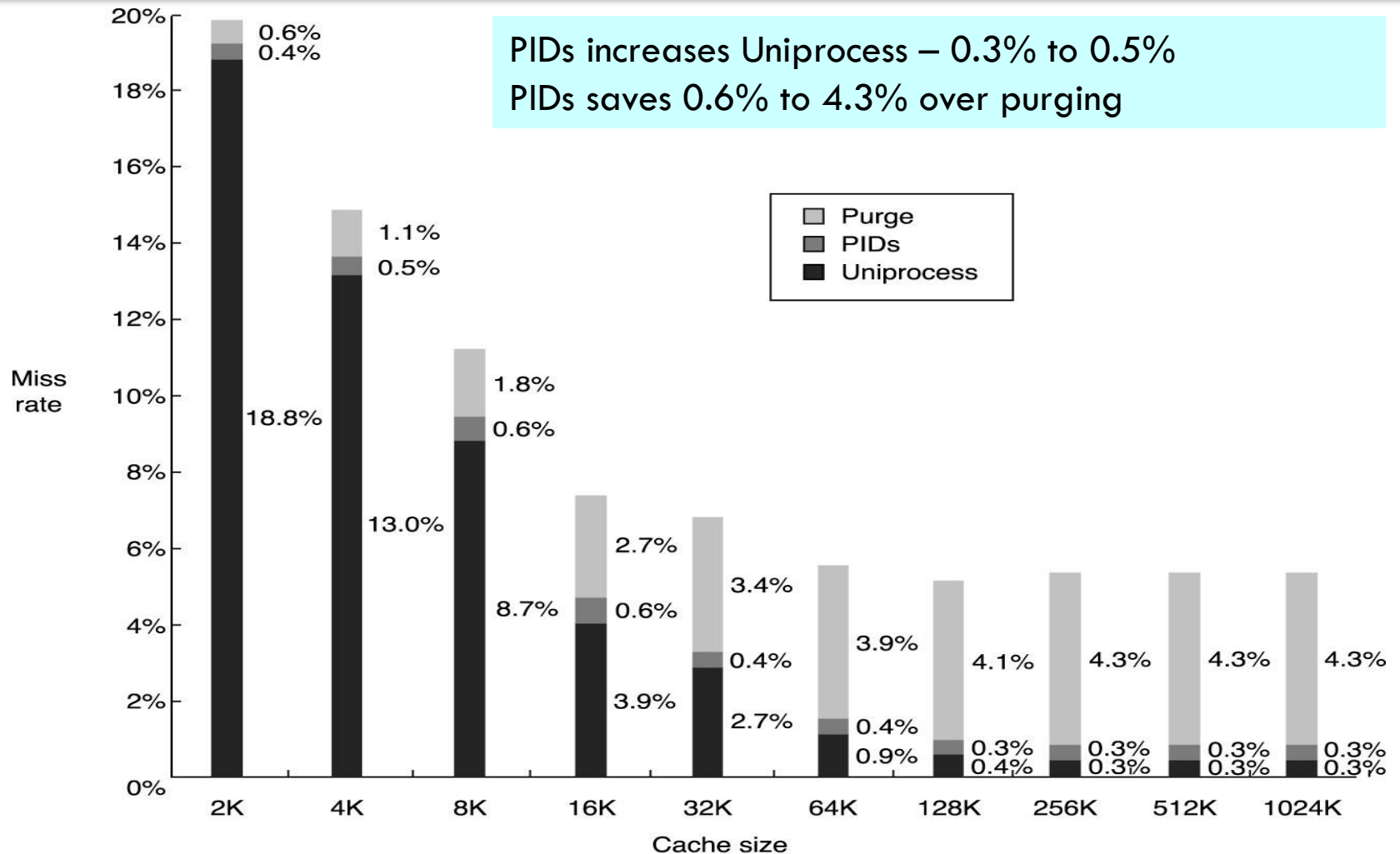
Paging Hardware with TLB



Problems with Virtual Caches

- Protection – necessary part of the virtual to physical address translation
 - ▣ Copy protection information on a miss, add a field to hold it, and check it on every access to virtually addressed cache.
- Task switch causes the same virtual address to refer to different physical address
 - ▣ Hence cache must be flushed
 - Creating huge task switch overhead
 - Also creates huge compulsory miss rates for new process
 - ▣ Use PID's as part of the tag to aid discrimination

Miss Rate of Virtual Caches



Problems with Virtual Caches (Cont.)

□ Synonyms or Alias

- ▣ OS and User code have different virtual addresses which map to the same physical address (facilitates copy-free sharing)
- ▣ Two copies of the same data in a virtual cache → consistency issue
- ▣ Anti-aliasing (HW) mechanisms guarantee single copy
 - On a miss, check to make sure none match PA of the data being fetched (must VA → PA); otherwise, invalidate
- ▣ SW can help - e.g. SUN's version of UNIX
 - Page coloring - aliases must have same low-order 18 bits

□ I/O – use PA

- ▣ Require mapping to VA to interact with a virtual cache

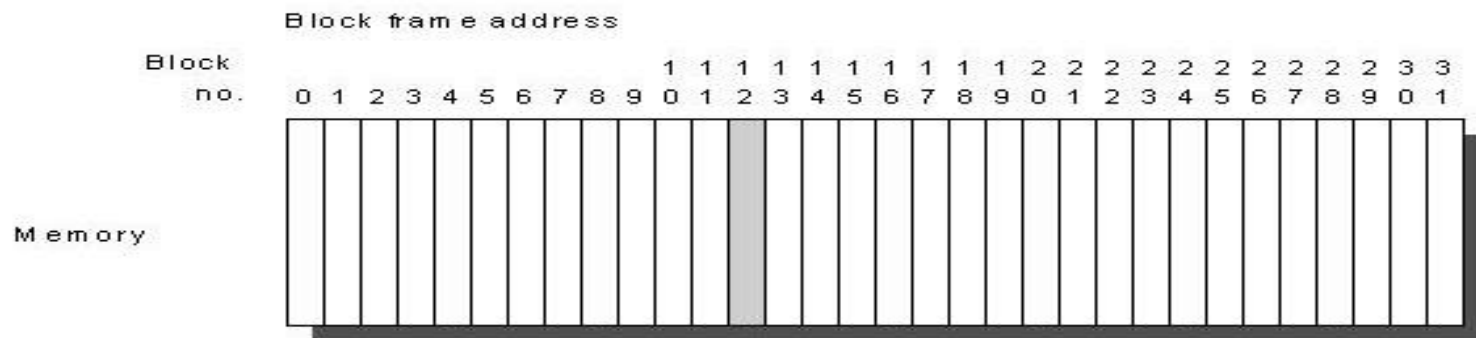
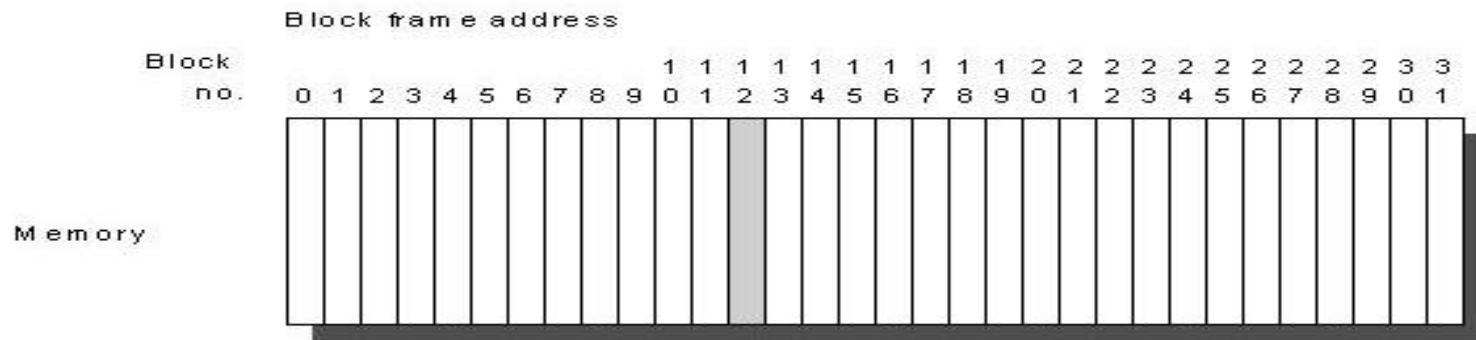
Pipelining Writes for Fast Write Hits – Pipelined Cache

- Write hits usually take longer than read hits
 - Tag must be checked before writing the data
- Pipelines the write
 - 2 stages – Tag Check and Update Cache (can be more in practice)
 - Current write tag check & previous write cache update
- Result
 - Looks like a write happens on every cycle
 - Cycle-time can stay short since real write is spread over
 - Mostly works if CPU is not dependent on data from a write
 - Spot any problems if read and write ordering is not preserved by the memory system?
- Reads play no part in this pipeline since they already operate in parallel with the tag check

Trace Caches

- Conventional caches limit the instructions in a static cache block to spatial locality
- Conventional caches may be entered from and exited by a taken branch → first and last portion of a block are unused
 - ▣ Taken branches or jumps are 1 in 5 to 10 instructions
 - A 64-byte block has 16 instructions → space utilization problem
- A trace cache stores instructions only from the branch entry point to the exit of the trace → avoid header and trailer overhead

Trace Cache



Trace Caches (Cont.)

- Complicated address mapping mechanism, as addresses are no longer aligned to power of 2 multiples of word size
- May store the same instructions multiple time in I-cache
 - ▣ Conditional branches making different choices result in the same instructions being part of separate traces, which each occupy space in the cache
- Intel NetBurst (foundation of Pentium 4)

Cache Optimization Summary

Technique	Miss penalty	Miss rate	Hit time	Hardware complexity	Comment
Multilevel caches	+			2	Costly hardware; harder if block size L1 \neq L2; widely used
Critical word first and early restart	+			2	Widely used
Giving priority to read misses over writes	+			1	Trivial for uniprocessor, and widely used
Merging write buffer	+			1	Used with write through; in 21164, UltraSPARC III; widely used
Victim caches	+	+		2	AMD Athlon has eight entries
Larger block size	-	+		0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size		+	-	1	Widely used, especially for L2 caches
Higher associativity		+	-	1	Widely used
Way-predicting caches		+		2	Used in I-cache of UltraSPARC III; D-cache of MIPS R4300 series
Pseudoassociative		+		2	Used in L2 of MIPS R10000
Compiler techniques to reduce cache misses		+		0	Software is a challenge; some computers have compiler option
Nonblocking caches	+			3	Used with all out-of-order CPUs
Hardware prefetching of instructions and data	+	+		2 instr., 3 data	Many prefetch instructions; UltraSPARC III prefetches data
Compiler-controlled prefetching	+	+		3	Needs nonblocking cache too; several processors support it
Small and simple caches		-	+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21164, UltraSPARC III
Pipelined cache access			+	1	Widely used
Trace cache			+	3	Used in Pentium 4



Main Memory

Main Memory Background

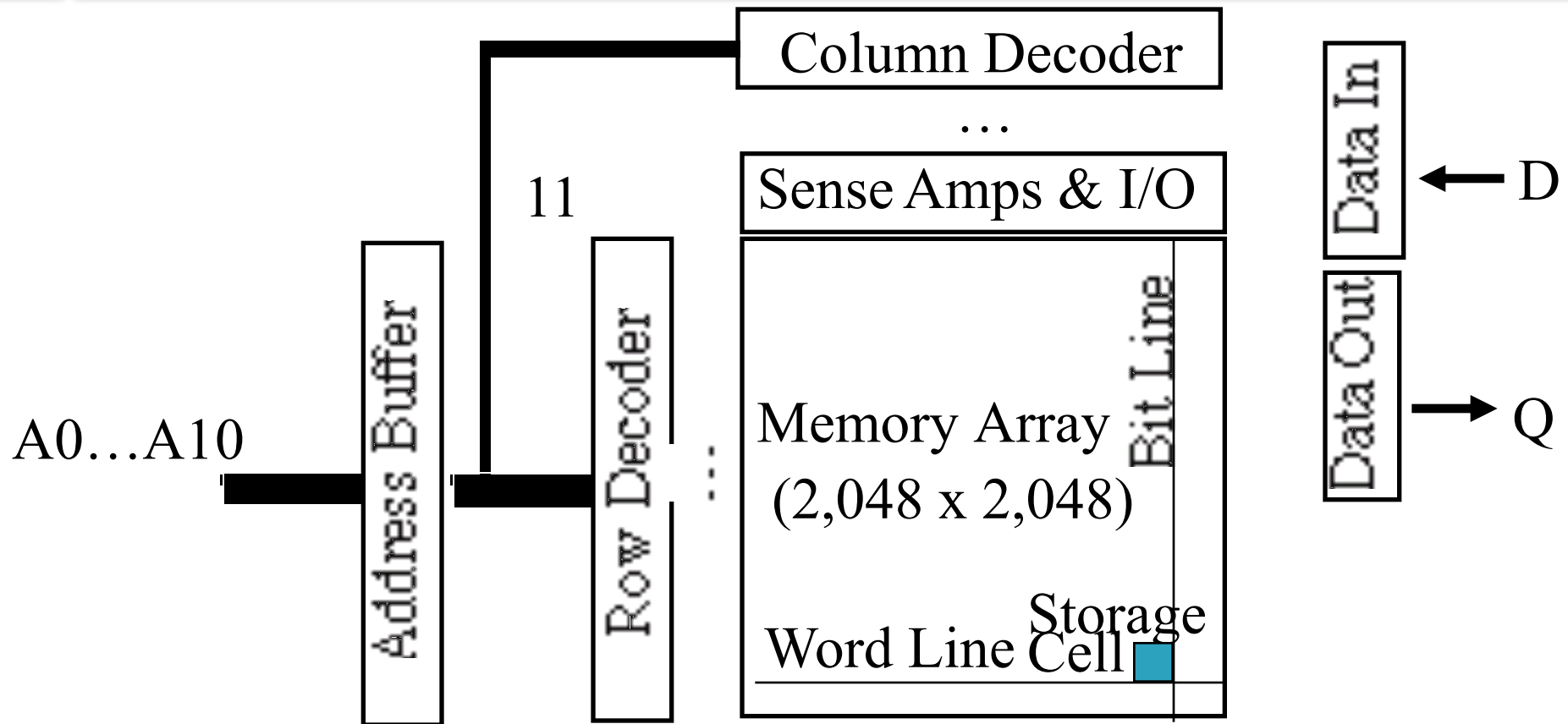
- Performance of Main Memory:
 - ▣ Latency: Cache Miss Penalty
 - *Access Time*: time between request and word arrives
 - *Cycle Time*: time between requests
 - ▣ Bandwidth: I/O & Large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
 - ▣ Dynamic since needs to be *refreshed* periodically (8 ms, 1% time)
 - ▣ Addresses divided into 2 halves (Memory as a 2D matrix):
 - *RAS* or *Row Access Strobe*
 - *CAS* or *Column Access Strobe*
- Cache uses *SRAM*: Static Random Access Memory
 - ▣ No refresh (6 transistors/bit vs. 1 transistor)
 - Size*: DRAM/SRAM 4-8,
 - Cost/Cycle time*: SRAM/DRAM 8-16

Main Memory Deep Background



- “Out-of-Core”, “In-Core,” “Core Dump”?
- “Core memory”?
- Non-volatile, magnetic
- Lost to 4 Kbit DRAM (today using 64Kbit DRAM)
- Access time 750 ns, cycle time 1500-3000 ns

DRAM logical organization (4 Mbit)



□ Square root of bits per RAS/CAS

Main Memory -- 3 important issues

- Capacity
- Latency
 - ▣ Access time: time between a read is requested and the word arrives
 - ▣ Cycle time: min time between requests to memory (> access time)
 - Memory needs the address lines to be stable between accesses
 - ▣ By addressing big chunks - like an entire cache block (amortize the latency)
 - ▣ Critical to cache performance when the miss is to main
- Bandwidth -- *# of bytes read or written per unit time*
 - ▣ Affects the time it takes to transfer the block

Example of Memory Latency and Bandwidth

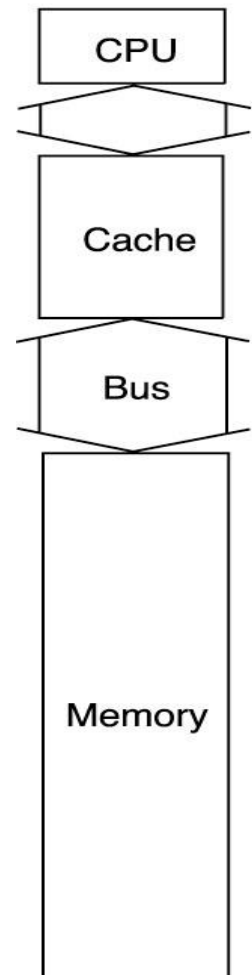
- Consider
 - ▣ 4 cycle to send the address
 - ▣ 56 cycles per word of access
 - ▣ 4 cycle to transmit the data
- Hence if main memory is organized by word
 - ▣ 64 cycles has to be spent for every word we want to access
- Given a cache line of 4 words (8 bytes per word)
 - ▣ 256 cycles is the miss penalty
 - ▣ Memory bandwidth = $1/8$ byte per clock cycle ($4 * 8 / 256$)

Improving Main Memory Performance

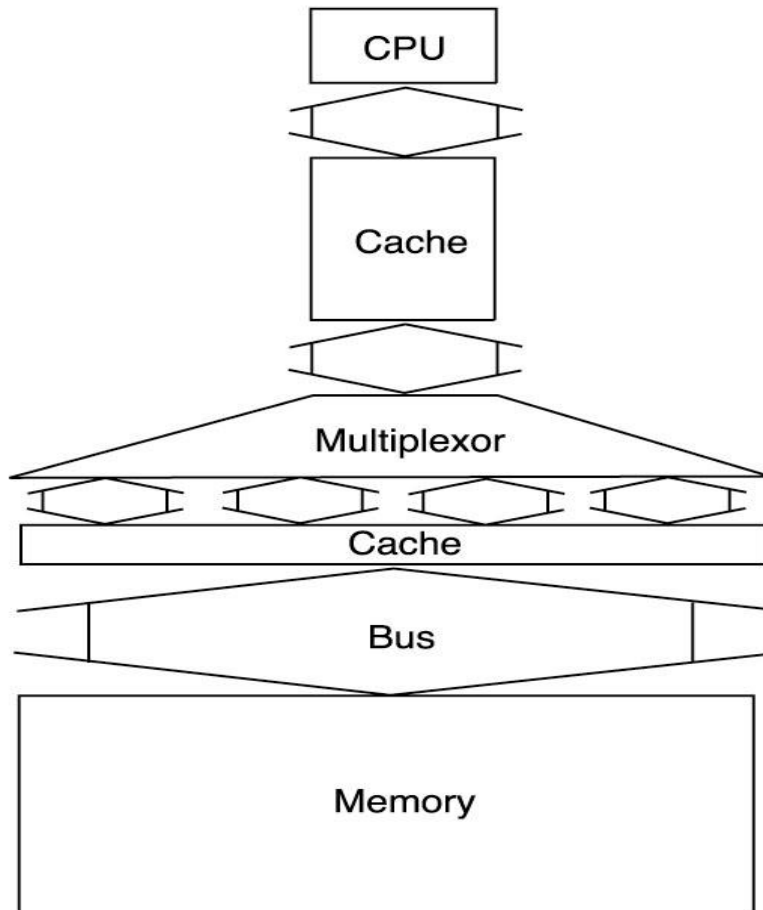
- Simple:
 - ▣ CPU, Cache, Bus, Memory same width (32 or 64 bits)
- Wide:
 - ▣ CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UltraSPARC 512)
- Interleaved:
 - ▣ CPU, Cache, Bus 1 word: Memory N Modules (4 Modules); example is word interleaved

3 Examples of Bus Width, Memory Width, and Memory Interleaving to Achieve Memory Bandwidth

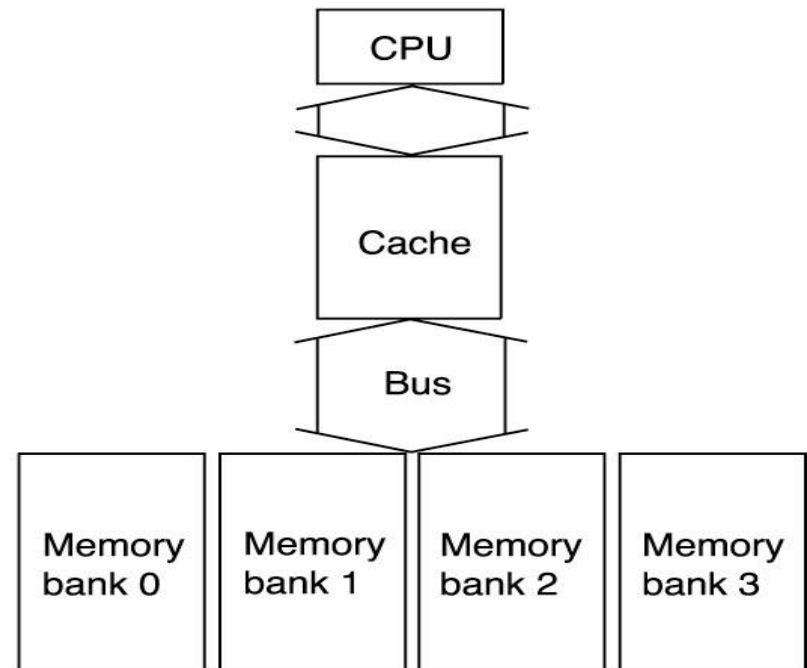
(a) One-word-wide memory organization



(b) Wide memory organization



(c) Interleaved memory organization



Wider Main Memory

- Doubling or quadrupling the width of the cache or memory will doubling or quadrupling the memory bandwidth
 - ▣ Miss penalty is reduced correspondingly
- Cost and Drawback
 - ▣ More cost on memory bus
 - ▣ Multiplexer between the cache and the CPU may be on the critical path (CPU is still access the cache one word at a time)
 - Multiplexors can be put between L1 and L2
 - ▣ The design of error correction become more complicated
 - If only a portion of the block is updated, all other portions must be read for calculating the new error correction code
 - ▣ Since main memory is traditionally expandable by the customer, the **minimum increment** is doubled or quadrupled

Simple Interleaved Memory

Bank_# = address MOD #_of_banks

Address_within_bank = Floor(Address / #_of_banks)

- Memory chips are organized into banks to read or write multiple words at a time, rather than a single word
 - ▣ Share address lines with a memory controller
 - ▣ Keep the memory bus the same but make it run faster
 - ▣ Take advantage of potential memory bandwidth of all DRAMs banks
 - ▣ The banks are often one word wide
 - ▣ Good for accessing consecutive memory location
- Miss penalty of $4 + 56 + 4 * 4$ or 76 CC (0.4 bytes per CC)

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

What Can Interleaving and a Wide Memory Buy?

- Block size = 1, 2, 4 words. Miss rate = 3%, 2% 1.2% correspondingly
 - Memory Bus width = 1 word, memory access per instruction = 1.2
 - Cache miss penalty = 64 cycles (as above)
 - Average cycles per instruction (ignore cache misses) = 2
 - $CPI = 2 + (1.2 * 3\% * 64) = 4.3$ (1-word block)
-
- Block size = 2 words
 - 64-bit bus and memory, no interleaving = $2 + (1.2 * 2\% * 2 * 64) = 5.07$
 - 64-bit bus and memory, interleaving = $2 + (1.2 * 2\% * (4+56+2*4)) = 3.63$
 - 128-bit bus and memory, no interleaving = $2 + (1.2 * 2\% * 1 * 64) = 3.54$
-
- Block size = 4 words
 - 64-bit bus and memory, no interleaving = $2 + (1.2 * 1.2\% * 4 * 64) = 5.69$
 - 64-bit bus and memory, interleaving = $2 + (1.2 * 1.2\% * (4+56+4*4)) = 3.09$
 - 128-bit bus and memory, no interleaving = $2 + (1.2 * 1.2\% * 2 * 64) = 3.84$

Simple Interleaved Memory (Cont.)

- Interleaved memory is logically a wide memory, except that accesses to bank are staged over time to share bus
- How many banks should be included?
 - ▣ More than # of CC to access word in bank
 - To achieve the goal that delivering information from a new bank each clock for sequential accesses → avoid waiting
- Disadvantages
 - ▣ Making multiple banks are expensive → larger chip, few chips
 - 512MB RAM
 - 256 chips of 4M*4 bits → 16 banks of 16 chips
 - 16 chips of 64M*4 bit → only 1 bank
 - ▣ More difficulty in main memory expansion (like wider memory)

Independent Memory Banks

- Memory banks for independent accesses vs. faster sequential accesses (like wider or interleaved memory)
 - ▣ Multiple memory controller
- Good for...
 - ▣ Multiprocessor I/O
 - ▣ CPU with Hit under n Misses, Non-blocking Cache

