

(Review) Cache Organization

- Cache is the name given to the first level of the memory hierarchy, encountered once the address leaves the CPU
 - It serves as a **temporary place** where **frequently-used values** can be stored
 - Retains the **same name** as in memory (different from registers)
 - To **avoid** having to go to memory every time this value is needed
 - Caches are faster (hence more expensive, limited in size) than DRAM
- Caches store values at the granularity of **cache blocks** (lines)
 - Larger than a single word: efficiency and **spatial locality** concerns
 - Cache **hit** if value in cache, else cache **miss**
- Effect of caches on CPU execution time

$$\begin{aligned}\text{CPU time} &= (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time} \\ \text{Memory stall clock cycles} &= (\text{Reads} \times \text{Read miss rate} \times \text{Read miss penalty} + \\ &\quad \text{Writes} \times \text{Write miss rate} \times \text{Write miss penalty}) \\ &= \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

Four Questions for Memory Hierarchy Designers

Q1: Where can a block be placed in the upper level?

(Block placement)

- Fully Associative, Set Associative, Direct Mapped

Q2: How is a block found if it is in the upper level?

(Block identification)

- Tag per block

Q3: Which block should be replaced on a miss?

(Block replacement)

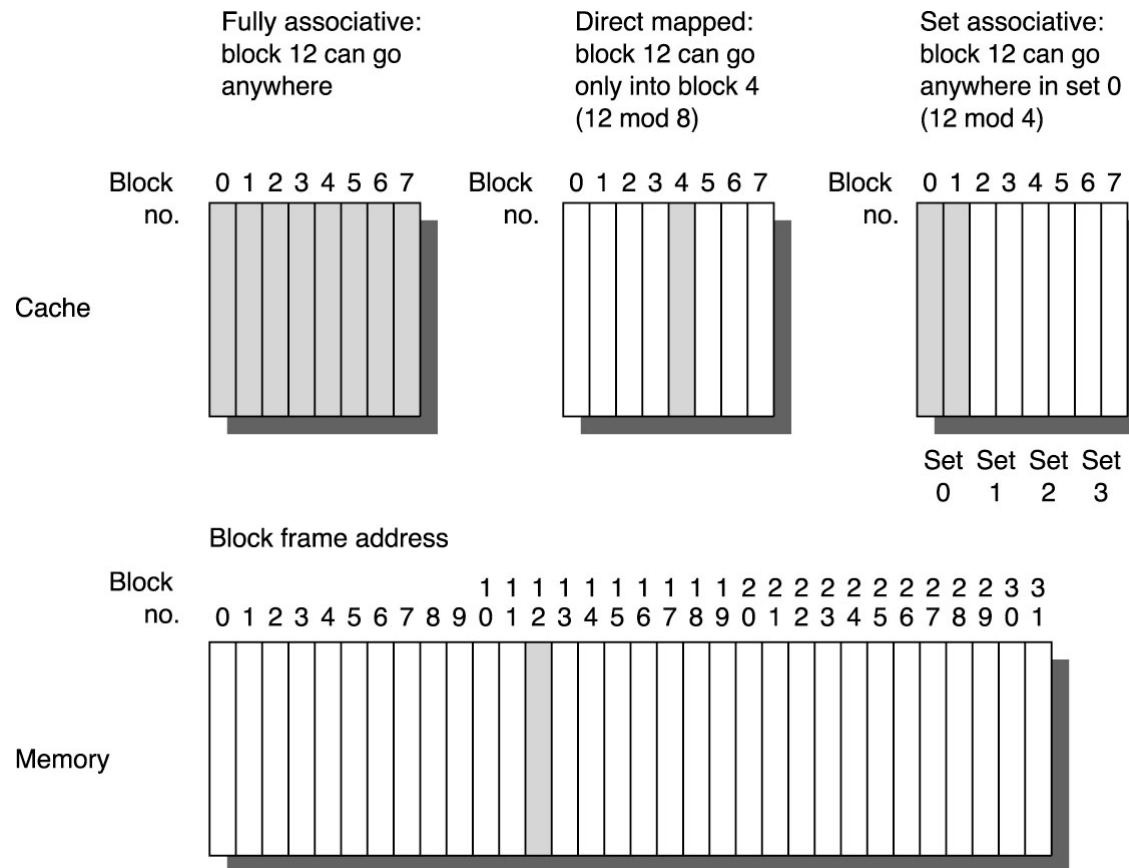
- Random, LRU

Q4: What happens on a write?

(Write strategy)

- Write Back or Write Through (with Write Buffer)

Question 1: Block Placement



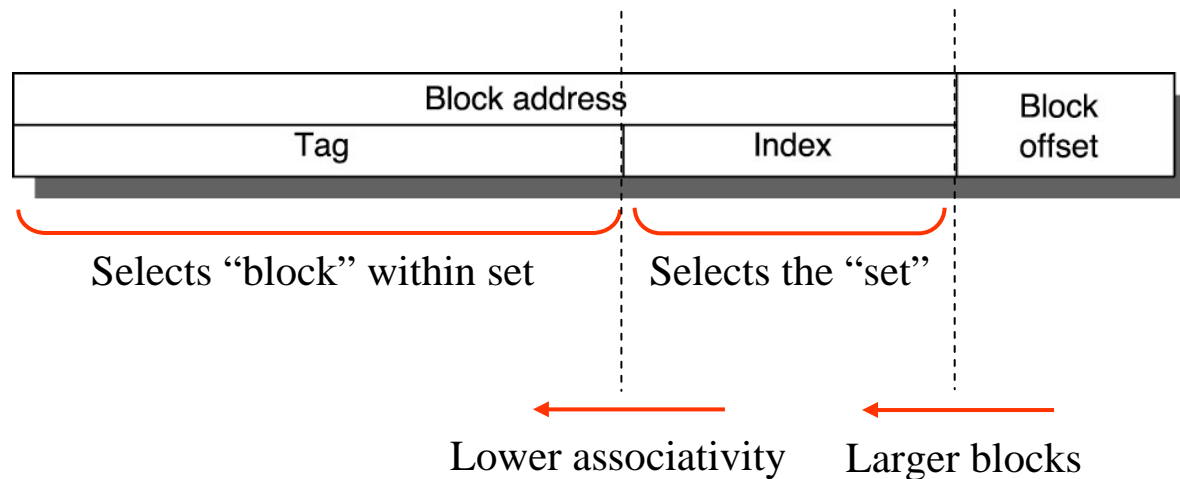
- Fully associative: block can be placed anywhere
- Direct map: each block has one place
- Set associative: block can be placed anywhere in a set

Range of caches is really a continuum of levels of **set associativity**

Most caches today are direct-mapped (1-way), 2-way or 4-way associative

Question 2: Block Identification

- Caches have a **tag** on each block frame that gives the block address
 - All possible tags, where the block may be present, are checked in **parallel**
- Quick check of whether a block contains data: **Valid bit**
- Organization determines which (subset of) blocks need to be checked
 - View memory address as below



- Fully-associative caches: Only tag

Question 3: Block Replacement

- When a new block needs to be brought in (on demand), an existing cache block may need to be freed up
- Three commonly-used schemes
(we only select a block within the appropriate “set”)
 - **Random**: Easiest to implement
 - Least-recently used (**LRU**)
 - First-in, first-out (**FIFO**): used as an approximation to LRU
- LRU outperforms Random and FIFO on smaller caches
 - FIFO outperforms Random
- Differences not as big for larger caches
 - Bigger benefit from avoiding misses in the first place

Question 4: Write Strategy

- When is memory updated with the contents of a store?
- *Issue*: Reads dominate cache traffic (writes typically 10% of accesses)
 - Optimization for read: Do tag checking and data transfer in parallel
 - Cannot do this for writes (also, only sub-portion of block needs update)
- Two write policies
 - **Write through**
 - Information written to both cache and memory
 - Simplifies replacement procedure (block is clean)
 - Also, simplifies **data coherency** (later in the course)
 - **Write back**
 - Information only written to the cache
 - **Dirty** bit keeps track of which blocks have data that needs to be sync-ed
 - Multiple writes lead to less number of writes to memory
 - Reduces memory bandwidth requirement (hence power)
 - Variants: With or without **write-allocate** (usually used with write back)
- Write stalls in write-through caches reduced using **write buffers**

Improving Cache Performance

CPU time = (CPU execution clock cycles + **Memory stall clock cycles**) x clock cycle time

Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty +
Writes x Write miss rate x Write miss penalty)
= Memory accesses x **Miss rate** x **Miss penalty**

- Above assumes 1-cycle to hit in cache
 - Hard to achieve in current-day processors (faster clocks, larger caches)
 - More reasonable to also include hit time in the performance equation

Average memory access time = **Hit Time** + **Miss rate** x **Miss Penalty**

Small/simple caches
Avoiding address translation
Pipelined cache access
Trace caches

(D)

Larger block size
Larger cache size
Higher associativity
Way prediction
Compiler optimizations

(B)

Multilevel caches
Critical word first
Read miss before write miss
Merging write buffers
Victim caches

(A)

Nonblocking caches
Hardware prefetching
Compiler prefetching

(C)

A.1. Reducing Miss Penalty via Multilevel Caches

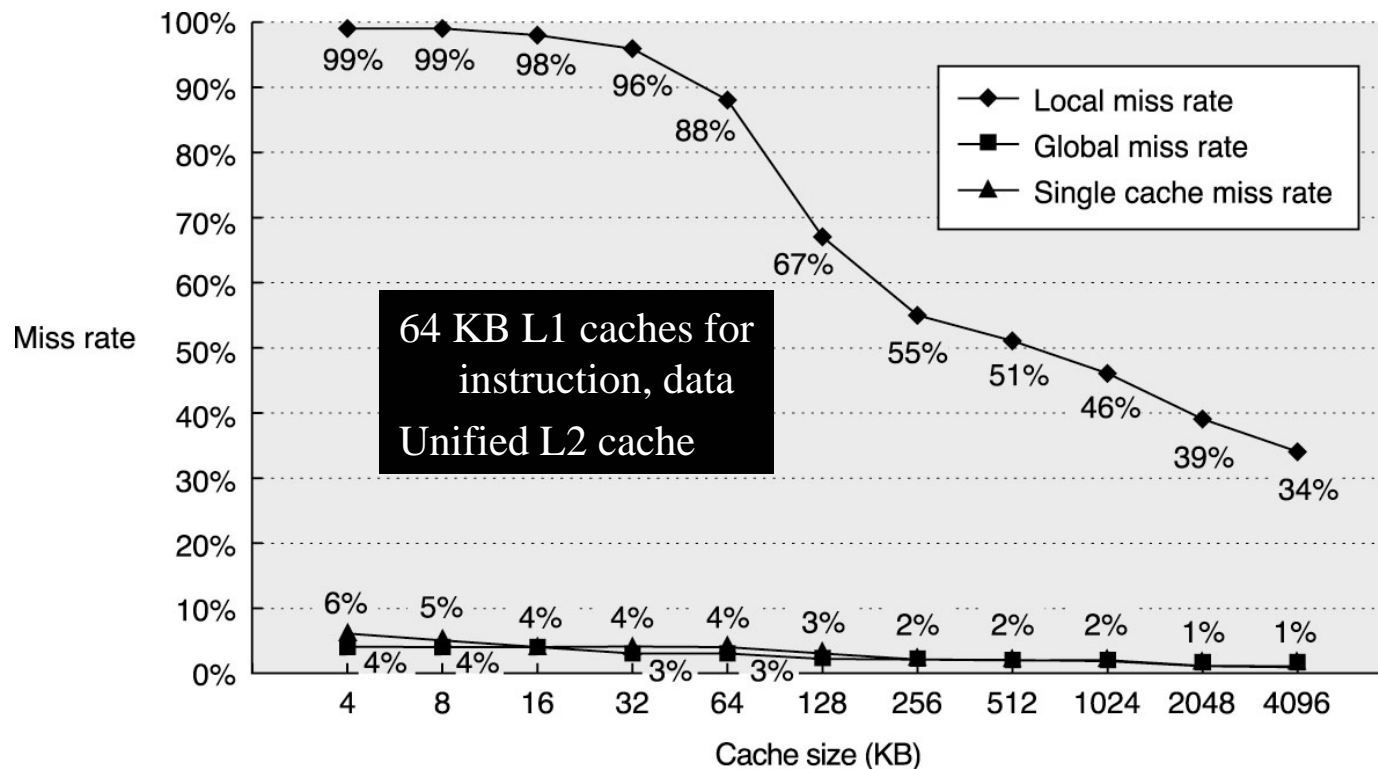
- *Idea*: Have multiple levels of caches
 - Tradeoff between size (cache effectiveness) and cost (access time)

- For a 2-level cache

Average memory access time = Hit time (L1) + Miss rate (L1) x Miss penalty (L1)
Miss penalty (L1) = Hit time (L2) + Miss rate (L2) x Miss penalty (L2)

- Distinguish between two kinds of miss rates
 - **Local** miss rate = Miss rate (L1) or Miss rate (L2)
 - **Global** miss rate = Number of misses/total number of memory accesses
= Miss rate (L1), but Miss rate (L1) x Miss rate(L2)
- Example: 1000 references, 40 misses in L1 cache and 20 in L2
 - **Local miss rates: 4% (L1), 50% (L2) = 20/40**
 - Global miss rates: 4% (L1), 2% (L2)
 - Avg. memory access time = $1 + 4\% \times (10 + 50\% \times 100) = 3.4$ cycles

Multilevel Caches (cont'd)



- Doesn't make much sense to have L2 caches smaller than L1 caches
- L2 needs to be significantly bigger to have reasonable miss rates
 - Cost of big L2 is smaller than big L1
- Exclusive and cooperative caches

A.2. Reduce Miss Penalty via Critical Word First and Early Restart

- *Idea:* Don't **wait for full block** to be loaded before restarting CPU
 - **Early restart:** request the words in a block in order. As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
 - **Critical Word First:** Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
 - Also called **wrapped fetch** and **requested word** first
- Drawbacks
 - Generally useful only in large blocks
 - Programs exhibiting spatial locality a problem; tend to want next sequential word, so limited benefit by early restart

A.3. Reducing Miss Penalty by giving Reads Priority over Writes on Misses

- Write buffers ensure that writes to memory do not stall the processor
- On the other hand, processor is blocked till read returns
- *Solution:* Give read misses **priority**

Challenges

- Write-through with write buffers may result in RAW conflicts
 - Solution 1: Wait for write buffer to empty (not great)
 - Solution 2: Check write buffer contents before read; if no conflicts, let the memory access continue
- Write-back caches: Read miss may require replacing a dirty block
 - Normal: Write dirty block to memory, and then do the read
 - Better alternative: 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 read is done

A.4. Reducing Miss Penalty using Merging Write Buffers

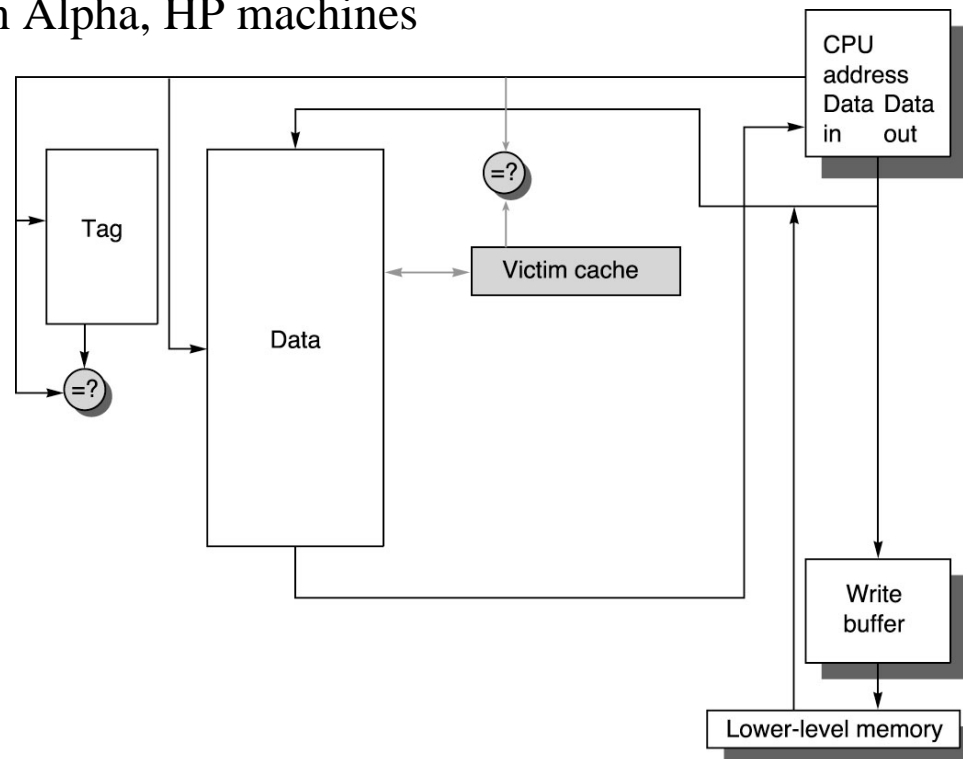
- Normal mode of operation of a write buffer
 - Absorb write from CPU, commit it to memory in the background
- Problem (particularly in write-through caches)
 - Small write-buffers may end up stalling processor if they fill up
 - Processor needs to wait till write committed to memory
- *Solution:* **Merge cache-block entries** in the write buffer
 - Multiword writes are usually faster than writes performed one at a time
 - Writes usually modify one word in a block; If a write buffer already contains some words from the given data block we will merge current modified word with the block parts already in the buffer

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

Write address	V	V	V	V	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		

A.5. Reducing Miss Penalty via a “Victim Cache”

- How to combine the fast hit time of direct-mapped caches, yet still avoid conflict misses?
- Remember what was recently discarded, just in case it is needed again
 - Jouppi [1990]: 4-entry **victim cache** reduced conflict misses by **20% - 95%** for a 4 KB direct mapped data cache
 - Used in Alpha, HP machines

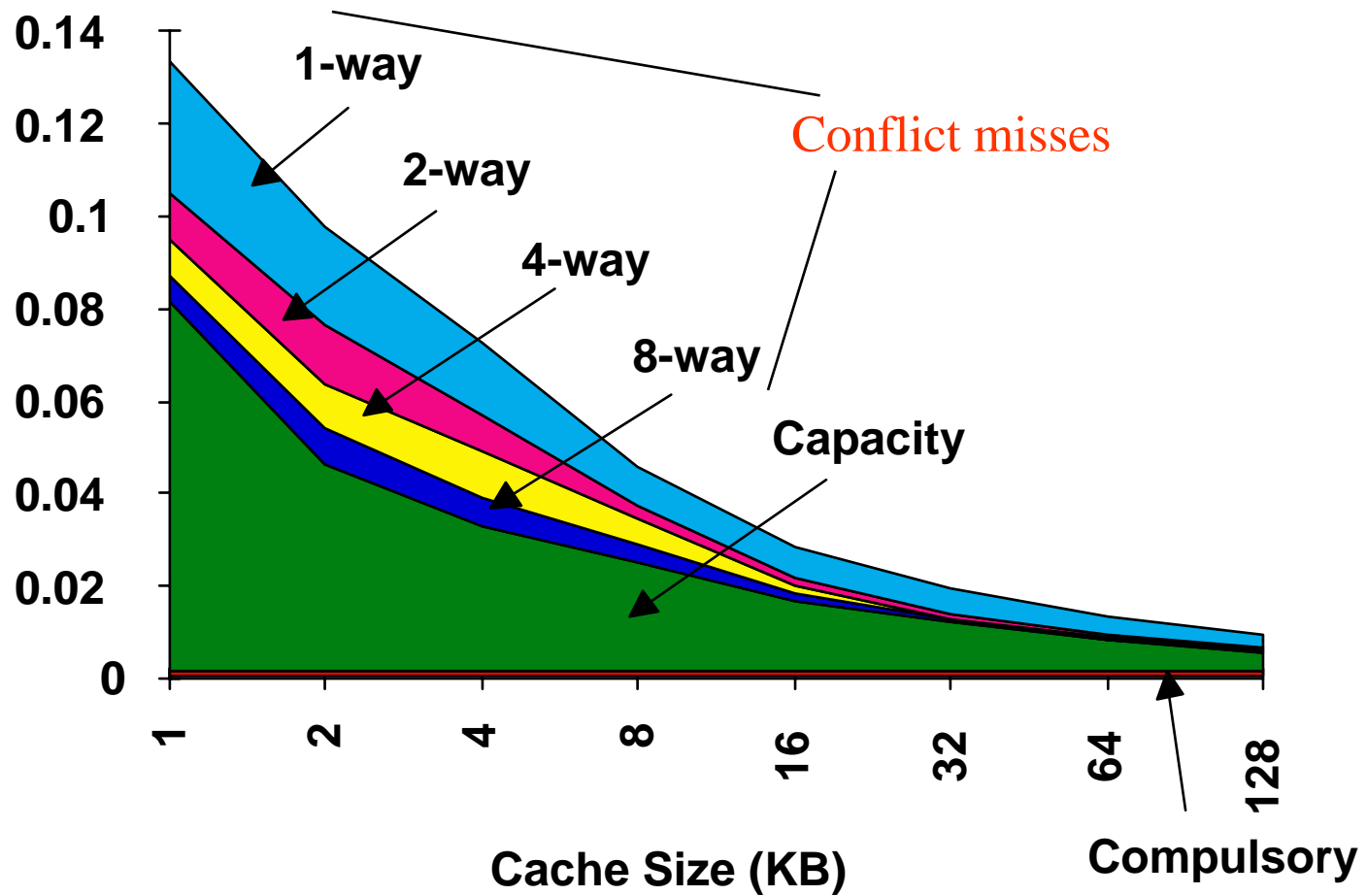


B. Reducing Cache Misses

Classifying Misses: 3 Cs

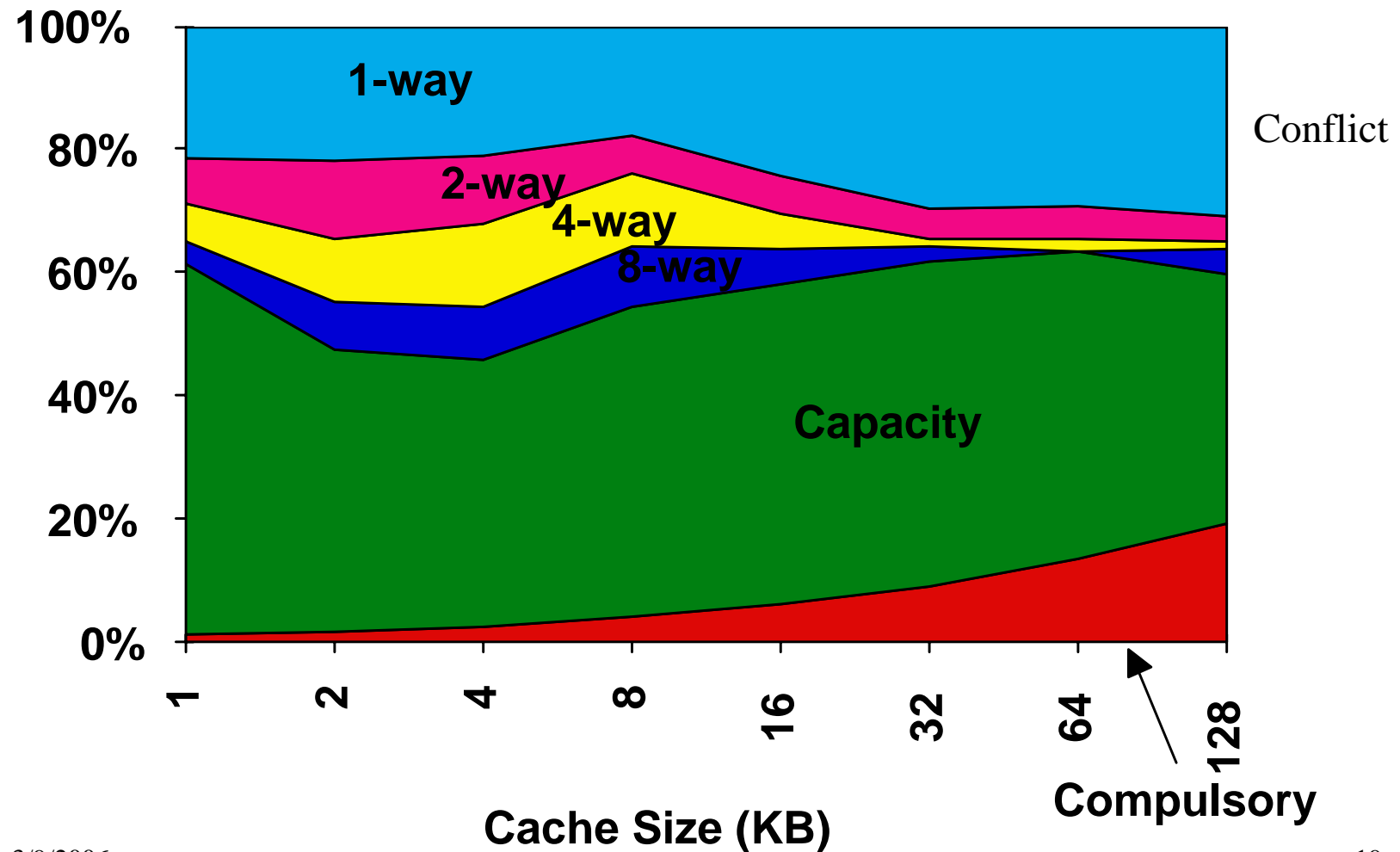
- **Compulsory** (Also called **cold start** or **first reference** misses)
 - The first access to a block is not in the cache, so the block must be brought into the cache.
(Misses in even an Infinite Cache)
- **Capacity**
 - The cache may not contain all blocks needed during program execution, so misses will occur due to blocks being discarded and later retrieved
(Misses in Fully Associative **Size X Cache**)
- **Conflict** (Also called **collision** or **interference** misses)
 - Additional misses that occur because another block is occupying cache (the rest of the cache might be unused)
(Misses in **N-way Associative**, Size X Cache)

3Cs Absolute Miss Rate (SPEC92)



3Cs Relative Miss Rate

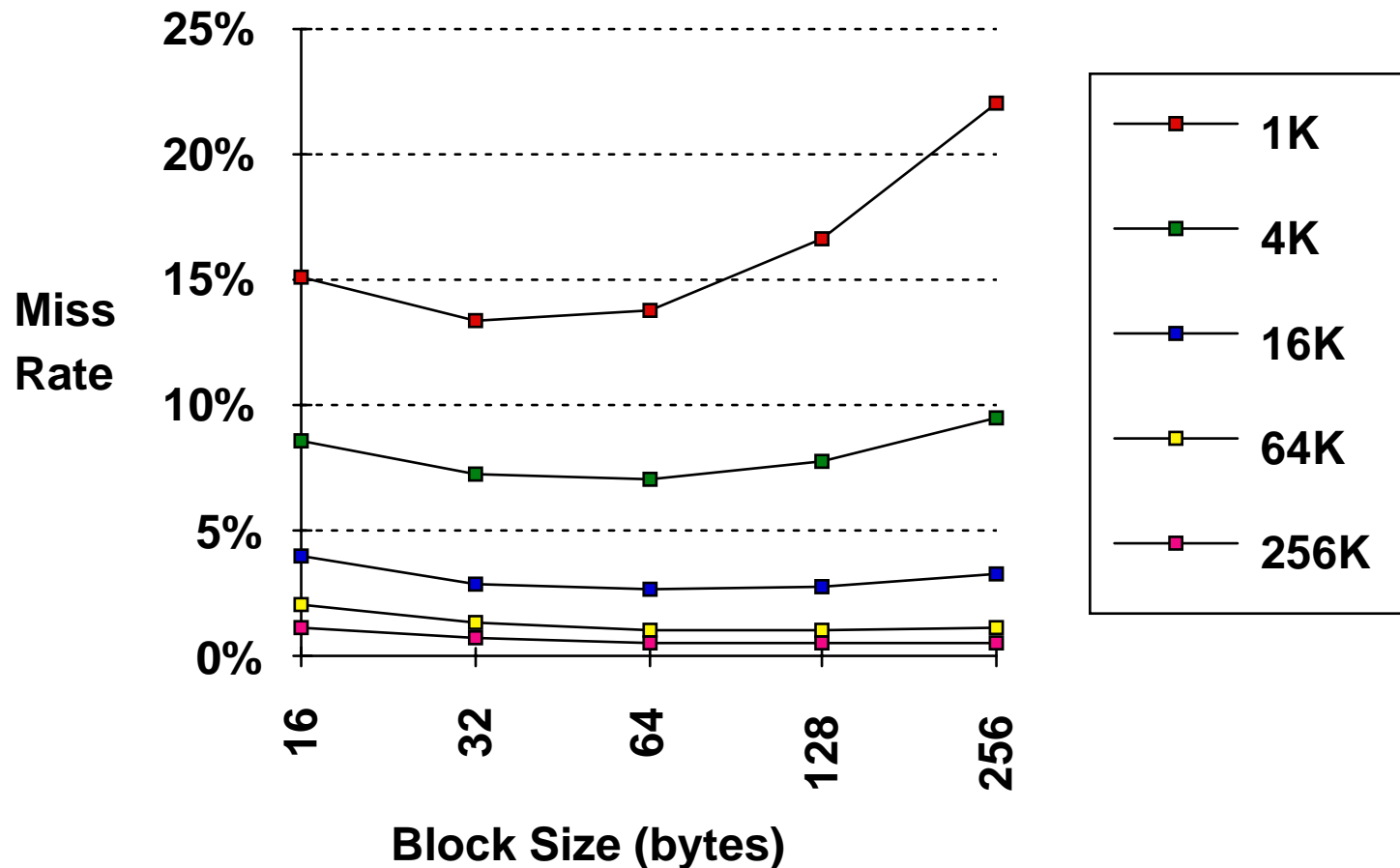
- Assumes fixed block size for each size cache



How Can We Reduce Misses?

- 3 Cs: Compulsory, Capacity, Conflict
- If we assume that total cache size is not changed, what happens if we
 1. Change **block size**
Which of 3Cs is obviously affected?
 2. Change **associativity**
Which of 3Cs is obviously affected?
 3. Change **compiler**
Which of 3Cs is obviously affected?

B.1. Reducing Miss Rate via Larger Block Sizes



- Small blocks: Data accesses spread over multiple blocks
- Large blocks: Not all the data is useful, but displaces useful data
- Also note larger blocks mean higher miss penalty

B.2. Reducing Miss Rate via Higher Associativity

- 2:1 Cache Rule
 - Miss Rate of a direct-mapped cache size of size $N \sim$
Miss Rate of a 2-way cache of size $N/2$
- Is this actually the case?
 - *Issue*: Increase in **clock cycle time (CCT)** may diminish benefits
- Higher associativity leads to higher hit time and can outweigh the benefit
- Average memory access time for SPEC92 vs. associativity
 - CCT = 1.0 for 1-way, 1.36 for 2-way, 1.44 for 4-way, 1.52 for 8-way

25 cycles to
access memory

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	2.23	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	1.86	1.92	2.00
256	1.32	1.66	1.74	1.82
512	1.20	1.55	1.59	1.66

B.3. Reducing Miss Rate via Way Prediction and Pseudoassociativity

- How to combine fast hit time of direct-mapped caches with the lower conflict misses of set-associative caches?
 - Previously looked at Victim Caches
- **Way prediction:** Predict which **block in a set** is likely to be accessed by the next memory access hitting this set
 - Tag comparison **only** with this block (cheaper as opposed to with all)
 - Higher cost to check non-predicted blocks
 - Simplest prediction: remember the last word accessed
 - Used in Alpha 21264 (1-cycle if correct prediction (**85%**), 3-cycles o.w.)
- **Pseudoassociative** or **Column associative**
 - Access proceeds as in direct-mapped cache
 - On a miss, check another location (“pseudoset”) before going to memory
 - Counts as a “slower hit”
 - If most hits become slow hits, **degrading** performance is possible
 - Used in MIPS R10000 L2 cache, similar in UltraSPARC

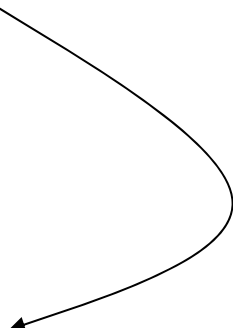
B.4. Reducing Miss Rate by Compiler Optimizations

- Compiler optimizations can help reduce both instruction and data cache misses (for a fixed cache organization)
- **Instruction misses**
 - **Reorder procedures** in memory so as to reduce conflict misses
 - Ensure that procedures used frequently do not map to same blocks/sets
 - Conflicts determined by profiling
 - Reduced I-cache misses by 75% in an 8KB cache (McFarling 1989)
 - **Cache-line alignment** of basic blocks
 - Decreases likelihood of cache miss on sequential code
- **Data misses**
 - Several optimizations that reorder data access patterns
 - Two examples
 - Loop interchange
 - Blocking

Loop Interchange Example

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

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

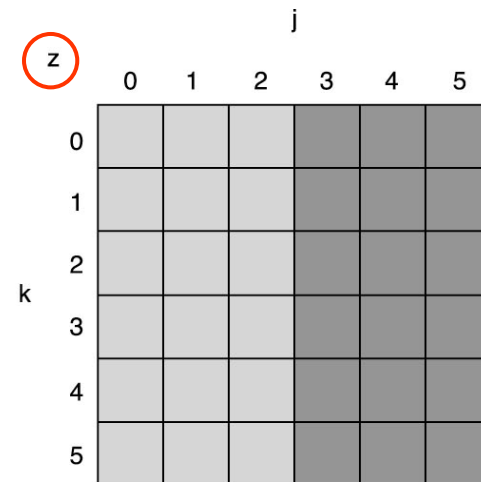
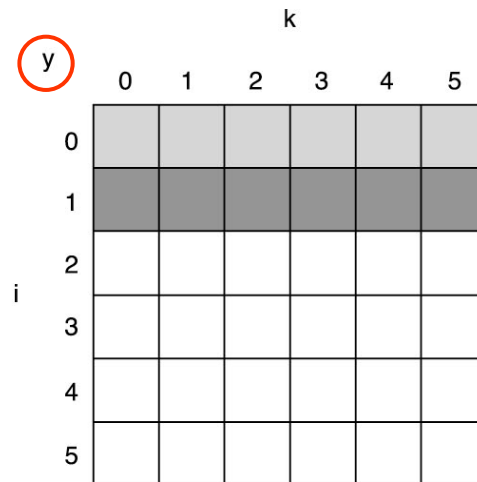
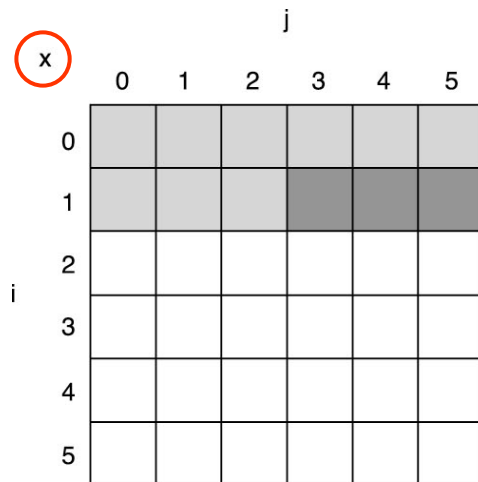


- “After” version accesses memory sequentially instead of in strides of 100 words
 - Improved **spatial locality**: use all of the words in fetched blocks

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;  
         };  
        };
```

Capacity misses depend on N, cache size
if all three matrices fit and there are
no conflict misses, best performance
if cache can hold one NxN matrix and
one row of N elements, then y and z can
be in the cache
else, misses for both y and z
worst case: $2N^3 + N^2$ misses



Blocking Example (cont'd)

/* After */

N^2/B^2

```
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;
```

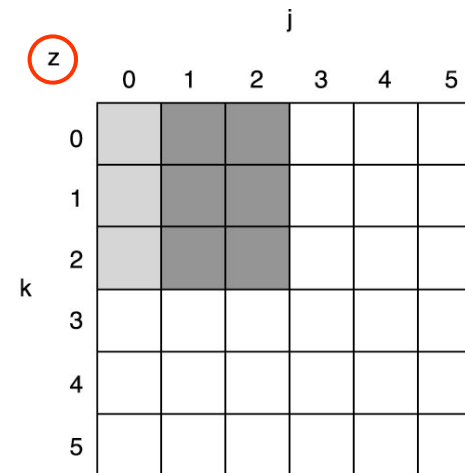
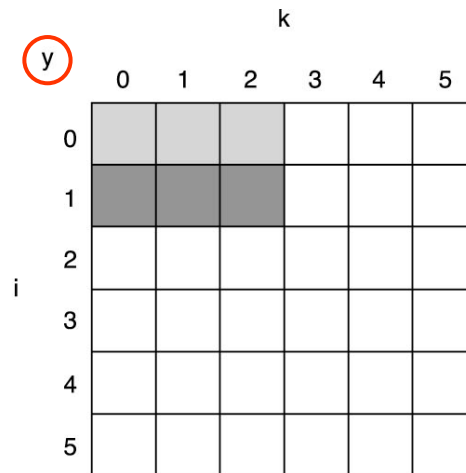
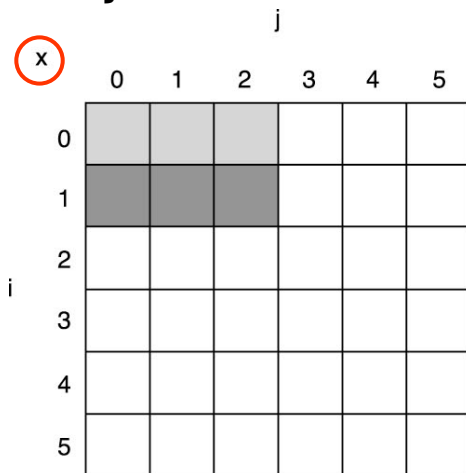
```
};
```

NB (x)
+
NB (y)
+
 B^2 (z)

Blocking factor: compute in blocks of $B \times B$

B chosen such that 1 row of B and 1 $B \times B$ matrix can fit in the cache. This ensures that y and z blocks are resident

Capacity misses:
 $2N^3/B + N^2$



C. Using Parallelism to Reduce Miss Penalty/Rate

- *Idea:* Permit multiple “outstanding” memory operations
 - Can overlap memory access latencies
 - Can benefit from activity done on behalf of other operations

Three commonly-employed schemes

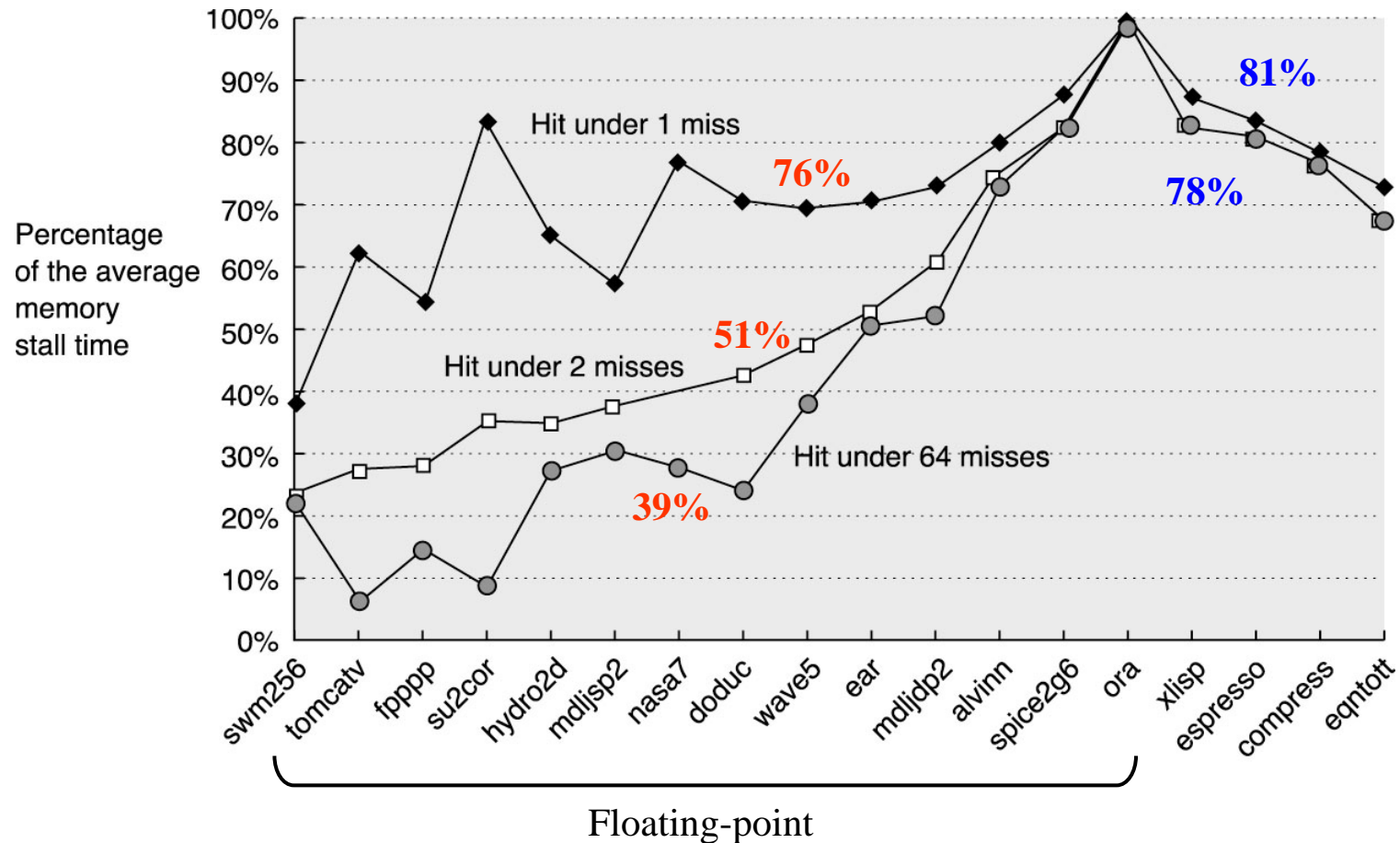
- Non-blocking caches
- Hardware prefetching
- Software prefetching

C.1. Non-blocking Caches to Reduce Stalls on Misses

- Decoupled instruction and data caches allow CPU to continue fetching instructions while waiting on a data cache miss
 - L1 cache misses can be tolerated by superscalar out-of-order machines
- **Non-blocking** or **lockup-free** caches allow data cache to continue to supply cache hits during a miss
 - requires out-of-order execution CPU
- “**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 miss penalty by overlapping multiple misses
 - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
 - Typically also requires multiple memory banks
 - Pentium Pro allows 4 outstanding memory misses

Value of Hit-Under-Miss for SPEC92

8KB direct-mapped cache, 32B blocks, 16-cycle penalty



C.2. Reducing Misses by Hardware Prefetching of Instructions & Data

- **Instruction Prefetching**

- Alpha 21064 fetches 2 blocks (requested and subsequent) on a miss
- Extra block in “**stream buffer**”
- On miss, check stream buffer

- **Works with data blocks too**

- Hardware identifies stream of accesses and then prefetches them
- Can compute stride by comparing current and previous access
- UltraSPARC III supports up to 8 simultaneous prefetches

- Prefetching relies on having extra memory bandwidth that **can be used without penalty**

How well does this work?

- Jouppi [1990]

- (for instructions w.r.t. a 4KB direct-mapped cache)
1-block stream buffer catches 15-25% of misses, 4-block stream buffer: 50%, 16-block stream buffer: 72%
- (for data w.r.t. a 4KB direct-mapped cache)
1-block buffer: 25%, 4 streams: 43% different streams prefetching at different addresses

- Palacharla & Kessler [1994]

- for scientific programs, 8 stream buffers got 50% to 70% of misses from a system with 2 64KB, 4-way set associative caches (one for instructions one for data)

C.3. Reducing Misses by Software Prefetching of Data

- Compiler can insert special instructions to request prefetching
- Two variants
 - Load data into **register** (HP PA-RISC loads)
 - Load data into **cache** (MIPS IV, PowerPC, SPARC v. 9)

Issues

- Special prefetching instructions typically cannot cause faults (a form of speculative execution: non-faulting vs. faulting)
- Processor must be able to proceed while prefetched data is being fetched to make this approach valuable
 - i.e., non-blocking data caches
- Issuing the prefetch instructions takes time
 - Is cost of prefetch issues < savings in reduced misses?
 - Higher superscalar reduces difficulty of issue bandwidth

D. Reducing Cache Hit Time

- Obvious approach: Smaller and simpler (low associativity) caches
 - Notable that L1 cache sizes have not increased
 - Alpha 21264/21364; UltraSPARC II/III; AMD K6/Athlon

Other techniques

- Avoiding address translation during cache lookup
 - Alternative 1: Index caches using “virtual addresses”
 - Needs to cope with several problems
 - Protection (performed during address translation)
 - Reuse of virtual addresses across processes (flushing cache after context switch)
 - Aliasing/synonyms: Two processes refer to the same physical address (results in having multiple copies of the same data)
 - I/O (typically uses physical addresses)
 - Alternative 2: Use part of the **page offset** to index the cache
does not change between virtual and physical addresses

D.1. Virtually Indexed, Physically Tagged Caches

- Overlap **indexing** of cache with translation of virtual addresses
 - **Tag comparison** done with physical addresses

Implications

Block address		Block offset
Tag	Index	

- Direct-mapped caches **can be no bigger than page size**
- Set-associative caches
 - Page offset can be viewed as (Index + block offset) above
 - Cache size = $2^{\text{page offset}}$ x Set associativity
 - So, **increased associativity allows larger cache sizes**
 - Pentium III (8KB pages): 2-way set-associative 16 KB cache
 - IBM 3033 (4KB pages): 16-way set-associative 64 KB cache

D.2. Trace Caches

- A challenge in multiple-issue processors is to supply enough instructions every cycle without dependencies
 - Challenge: fetching across branches
 - Cache impact is significant with large cache blocks
- Option 1: Combine branch prediction with instruction prefetching
 - Instructions stored according to memory addresses
- Option 2: A separate cache that stores and provides a **dynamic sequence of instructions** including taken branches (Trace Cache)
 - Pros
 - Effective use of cache block: no wasted words, no conflicts, ...
 - Cons
 - Complicated address mapping mechanisms
 - Same instruction may be stored multiple times
 - Used in the Intel NetBurst microarchitecture (Pentium 4)

Cache Optimization Summary

<i>Technique</i>	<i>MP</i>	<i>MR</i>	<i>HT</i>	<i>Complexity</i>
Multilevel caches	+			2
Early Restart & Critical Word 1st	+			2
Priority to Read Misses	+			1
Merging write buffer	+			1
Victim Caches	+	+		2
Larger Block Size	—	+		0
Higher Associativity		+	—	1
Pseudo-Associative Caches		+		2
Compiler Reduce Misses		+		0
Non-Blocking Caches	+			3
HW Prefetching of Instr/Data	+	+		2/3
Compiler Controlled Prefetching	+	+		3
Avoiding Address Translation			+	2
Trace Cache			+	3

Lab Assignment 2: Branch prediction

- To understand techniques for reducing pipeline stalls from control hazards*

1.a. 2 bit local predictor

1.b. (2,2) correlating predictor

<i>Prediction</i>	<i>Actual</i>	
	Taken	Not Taken
Taken	1 cycle	2 cycle
Not Taken	2 cycle	0 cycle

2. Branch Target Buffer (BTB) and

3. Return Address Stack (RAS)

<i>BTB result</i>	<i>Predicted Direction</i>	<i>Actual Outcome</i>	
		Taken	Not Taken
Hit (correct target)	Taken	0 cycle	2 cycle
Hit (mispredicted target)	Taken	1 cycle	2 cycle
Miss	Taken	1 cycle	2 cycle
Miss	Not Taken	2 cycle	0 cycle

IF: Instruction Fetch

// Check to see if you can proceed (stall)
// fetch instruction pointed to by fetch_pc

// Increment fetch_pc for next inst.

// For non-CTRL instructions

fetch_pc \leftarrow fetch_pc +
sizeof(md_inst_t)

// For CTRL instructions

// (MD_OP_FLAGS(op) & F_CTRL)

if (btb_enabled)

btb_fetch_pc = bpred_lookup(pred,
/* branch address */ if_id_s.PC,
/* target address */ 0,
/* opcode */ op,
/* call? */ MD_IS_CALL(op),
/* return? */ MD_IS_RETURN(op),
/* update */ &update_rec,
/* RSB index */ &stack_recover_idx
)

// BTB miss: branch is predicted not-taken

if_id_s.prediction \leftarrow 0

fetch_pc \leftarrow fetch_pc + sizeof(md_inst_t)

// BTB miss: branch is predicted taken

if_id_s.prediction \leftarrow 1

fetch_pc \leftarrow fetch_pc + sizeof(md_inst_t)

// BTB hit

if_id_s.prediction \leftarrow 2

fetch_pc \leftarrow btb_fetch_pc

// BTB not enabled

fetch_pc \leftarrow fetch_pc + sizeof(md_inst_t)

if_id_s.NPC \leftarrow fetch_pc

ID: Instruction Decode

```
// Check to see if you can proceed  
// Check for hazards  
    id_stall = check_hazards( in1, in2,  
                             in3, out1, out2 )  
  
// Return if data hazard; otherwise  
// functionally simulate the instruction  
  
// Increment PC (registers)  
    regs.regs_PC = regs.regs_NPC  
    regs.regs_NPC += sizeof(md_inst_t)
```

```
// For CTRL instructions  
if ( !btb_enabled )  
    btb_fetch_pc  $\leftarrow$  bpred_lookup( pred,  
                                       /* branch address */ if_id_s.PC,  
                                       /* target address */ 0,  
                                       /* opcode */ op,  
                                       /* call? */ MD_IS_CALL(op),  
                                       /* return? */ MD_IS_RETURN(op),  
                                       /* update */ &update_rec,  
                                       /* RSB index */ &stack_recover_idx  
                                       )  
  
// prediction == Taken  
    id_ex_s.prediction  $\leftarrow$  1  
    if_stall  $\leftarrow$  1      // stall IF this cycle  
    fetch_pc  $\leftarrow$  target_PC  
  
// prediction == Not Taken  
    id_ex_s.prediction  $\leftarrow$  0
```

ID: Instruction Decode (Cont'd)

// BTB is enabled

// BTB miss: predicted Taken

id_ex_s.prediction \leftarrow 1

if_stall \leftarrow 1 **// stall IF this cycle**

fetch_pc \leftarrow target_PC

// BTB miss: predicted Not Taken

id_ex_s.prediction \leftarrow 0

// BTB hit

id_ex_s.prediction \leftarrow 1

// If BTB target mispredicted

if (if_id_s.NPC \neq target_PC)

if_stall \leftarrow 1 **// stall IF this cycle**

fetch_pc \leftarrow target_PC

// Update predictor

bpred_update(pred,

/* branch address */ if_id_s.PC,

/* target */ regs.regs_PC,

/* taken? */ (regs.regs_PC \neq (if_id_s.PC +
sizeof(md_inst_t))),

/* pred taken? */ (fetch_pc \neq (if_id_s.PC +
sizeof(md_inst_t))),

/* correct pred? */ (fetch_pc == regs.regs_PC),

/* opcode */ op,

/* predictor update pointer */ &update_rec

)

// Store branch target in id_ex_s.NPC

// such that fetch_pc can be updated if

// required in EX stage

id_ex_s.NPC \leftarrow regs.regs_PC

Hazard Detetction

// Forwarding enabled

// RAW: id_ex_s instruction conflicts with a MEM instruction in ex_mem_s

// All other results are available because of forwarding

```
hval ← ( ex_mem_s.busy &&
  (MD_OP_FLAGS(ex_mem_s.op) & F_MEM) &&
  ( ((in1 != NA) &&
    ((in1 == ex_mem_s.out1) ||
     (in1 == ex_mem_s.out2))) ||
    ((in2 != NA) &&
     ((in2 == ex_mem_s.out1) ||
      (in2 == ex_mem_s.out2))) ||
    ((in3 != NA) &&
     ((in3 == ex_mem_s.out1) ||
      (in3 == ex_mem_s.out2))) )
  )
```

// No need to check for WAW or WAR hazards

EX: Instruction Execute

// From ID stage:

// Store branch target so that it can be used if required in EX

id_ex_s.NPC \leftarrow regs.regs_PC

// Need this for mispredicted branches

branch_was_taken \leftarrow (id_ex_s.NPC \neq (id_ex_s.PC + sizeof(md_inst_t)))

if (branch_was_taken \neq id_ex_s.prediction)

fetch_pc \leftarrow id_ex_s.NPC

if_id_s.busy \leftarrow FALSE **// squash speculatively fetched instruction**

if_stall \leftarrow 1 **// stall IF this cycle**