Memory Hierarchy

CS 282 – KAUST – Spring 2010

Slides by:
Mikko Lipasti
Muhamed Mudawar

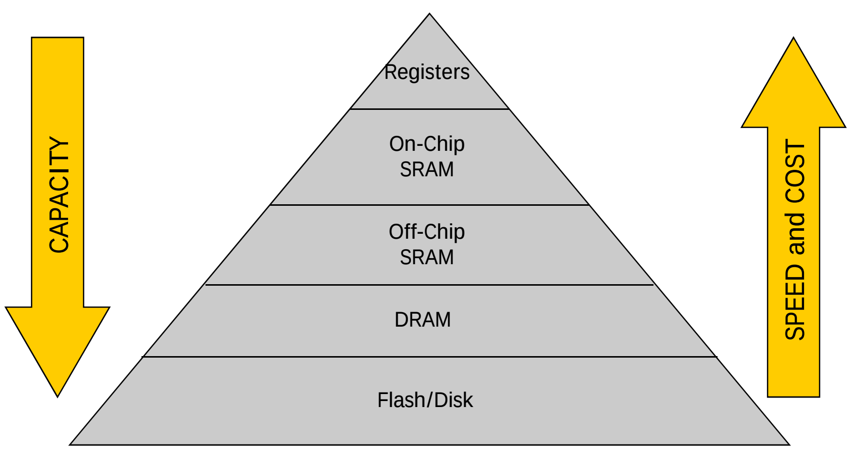
Memory Hierarchy

- Memory
  - Just an "ocean of bits"
  - Many technologies are available
- Key issues
  - Technology (how bits are stored)
  - Placement (where bits are stored)
  - Identification (finding the right bits)
  - Replacement (finding space for new bits)
  - Write policy (propagating changes to bits)
- Must answer these regardless of memory type

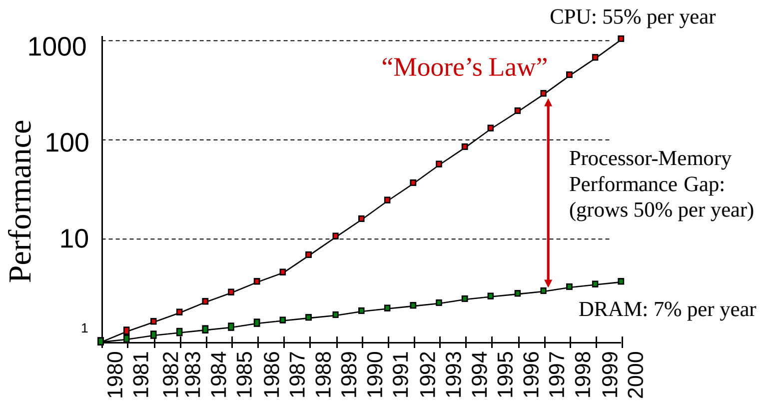
Types of Memory

Type	Size	Speed	Cost/bit
Register	< 1KB	< 1ns	$$$$
On-chip SRAM	8KB-16MB	< 10ns	$$$
DRAM	64MB – 1TB	< 100ns	$
Flash	64MB – 32GB	< 100us	c
Disk	40GB – 1PB	< 20ms	~0

Memory Hierarchy



Processor-Memory Performance Gap



- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

5

Why Memory Hierarchy?

- Bandwidth:

$$BW = \frac{1.0 inst}{cycle} \times \left[\frac{1 Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4 Dref}{inst} \times \frac{8B}{Dref} \right] \times \frac{3 Gcycles}{sec}$$
$$= \frac{21.6 GB}{sec}$$

- Capacity:
  - 1+GB for Windows PC to multiple TB
- Cost:
  - (TB x anything) adds up quickly
- These requirements appear incompatible

6

Why Memory Hierarchy?

- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state
  - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

Why Does a Hierarchy Work?

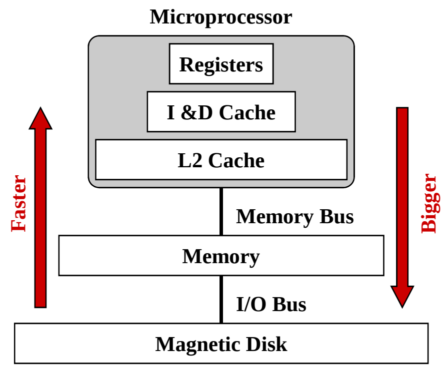
- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Typical Memory Hierarchy

- Registers are at the top of the hierarchy
  - Typical size < 1 KB
  - Access time < 0.5 ns
- Level 1 Cache (8 – 64 KB)
  - Access time: 0.5 – 1 ns
- L2 Cache (512KB – 8MB)
  - Access time: 2 – 10 ns
- Main Memory (1 – 2 GB)
  - Access time: 50 – 70 ns
- Disk Storage (> 200 GB)
  - Access time: milliseconds

<u>Temporal Locality</u>
- Keep recently referenced items at higher levels

<u>Spatial Locality</u>
- Bring neighbors of recently referenced to higher levels



Four Key Issues

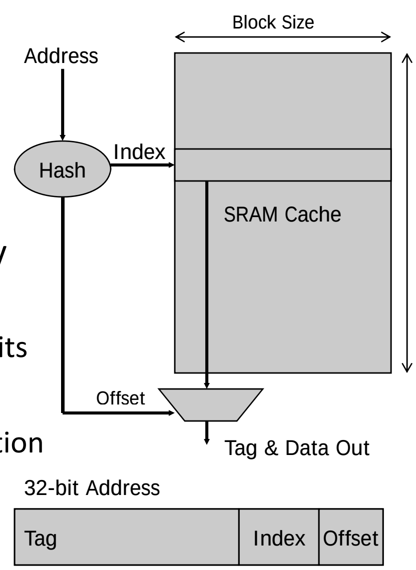
- Placement
  - Where can a block of memory go?
- Identification
  - How do I find a block of memory?
- Replacement
  - How do I make space for new blocks?
- Write Policy
  - How do I propagate changes?

Placement

Memory Type	Placement	Comments
Registers	Anywhere; Int, FP, SPR	Compiler/programmer manages
Cache (SRAM)	Fixed in H/W	*Direct-mapped, set-associative, fully-associative*
DRAM	Anywhere	O/S manages
Disk	Anywhere	O/S manages

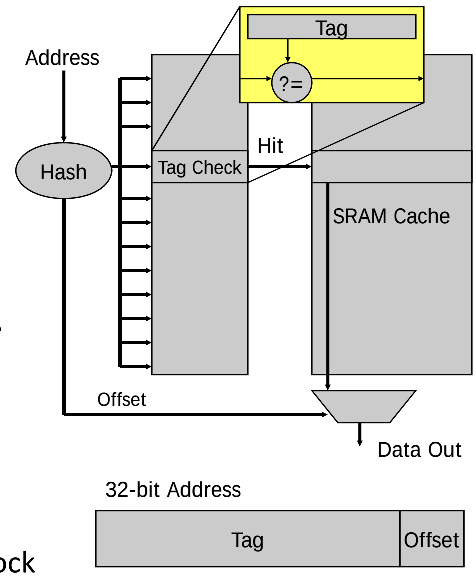
Placement

- Address Range
  - Exceeds cache capacity
- Map address to finite capacity
  - Called a *hash*
  - Usually just masks high-order bits
- *Direct-mapped*
  - Block can only exist in one location
  - Hash collisions cause problems
  - Must check tag (*identification*)



Identification

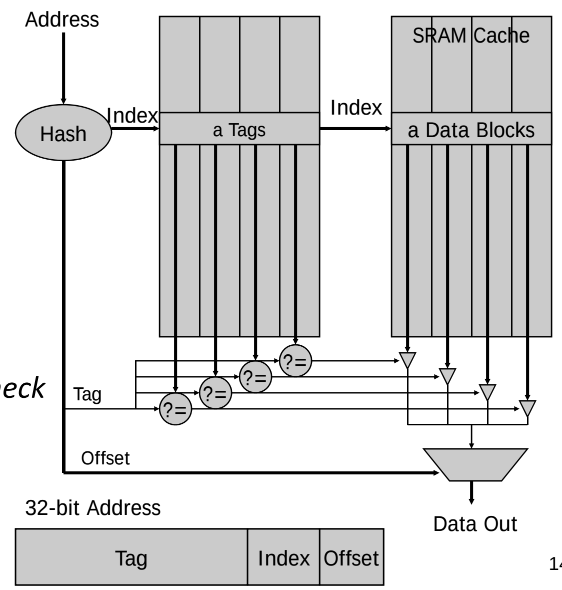
- *Fully-associative*
  - Block can exist anywhere
  - No more hash collisions
- *Identification*
  - How do I know I have the right block?
  - Called a *tag check*
    - Must store address tags
    - Compare against address
- Expensive!
  - Tag & comparator per block



13

Placement

- *Set-associative*
  - Block can be in *a* locations
  - Hash collisions:
    - Up to *a* still OK
- *Identification*
  - Still perform *tag check*
  - However, only *a* in parallel



14

Placement and Identification

32-bit Address

Tag	Index	Offset

Portion	Length	Purpose
Offset	$o=\log_2$(block size)	Select word within block
Index	$i=\log_2$(number of sets)	Select set of blocks
Tag	t=32 - o - i	ID block within set

- Consider: <BS=block size, S=sets, B=blocks>
  - <64,128,128>: o=6, i=7, t=19: direct-mapped (S=B)
  - <64,32,128>: o=6, i=5, t=21: 4-way S-A (S = B / 4)
  - <64,1,128>: o=6, i=0, t=26: fully associative (S=1)
- Total size = BS x B = BS x S x (B/S)

15

Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
- Same idea:
  - Move blocks to next level of cache

16

Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where *a* = *associativity*
  - If *a* <= 2, LRU is cheap and easy (1 bit)
  - If *a* > 2, it gets harder
  - Pseudo-random works pretty well for caches

17

Write Policy

- Replication in memory hierarchy
  - 2 or more copies of same block
    - Main memory and/or disk
    - Caches
- What to do on a write?
  - Eventually, all copies must be changed
  - Write must *propagate* to all levels

18

Write Policy

- Easiest policy: *write-through*
- Every write propagates directly through hierarchy
  - Write in L1, L2, memory?
- Why is this a bad idea?
  - Very high bandwidth requirement
  - Remember, large memories are slow
- Popular in real systems only to the L2
  - Every write updates L1 and L2
  - Beyond L2, use *write-back* policy

Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a *writeback* or *castout*

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

21

Write Miss Policy

- What happens on a write miss?
- Write Allocate:
  - Allocate new block in cache
  - Write miss acts like a read miss, block is fetched and updated
- No Write Allocate:
  - Send data to lower-level memory
  - Cache is not modified
- Typically, write back caches use write allocate
  - Hoping subsequent writes will be captured in the cache
- Write-through caches often use no-write allocate
  - Reasoning: writes must still go to lower level memory

22

Caches and Performance

- Caches
  - Enable design for common case: cache hit
    - Pipeline tailored to handle cache hits efficiently
    - Cache organization determines access latency, cycle time
  - Uncommon case: cache miss
    - Stall pipeline
    - Fetch from next level
      - Apply recursively if multiple levels
- What is performance impact?

Cache Misses and Performance

- Miss penalty
  - Detect miss: 1 or more cycles
  - Find victim (replace line): 1 or more cycles
    - Write back if dirty
  - Request line from next level: several cycles
  - Transfer line from next level: several cycles
    - (block size) / (bus width)
  - Fill line into data array, update tag array: 1+ cycles
  - Resume execution
- In practice: 6 cycles to 100s of cycles

Cache Miss Rate

- Determined by:
  - Program characteristics
    - Temporal locality
    - Spatial locality
  - Cache organization
    - Block size, associativity, number of sets
- Measured:
  - In hardware
  - Using simulation
  - Analytically

25

Cache Misses and Performance

- How does this affect performance?
- Performance = Time / Program

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) **(CPI)** **(cycle time)**

- Cache organization affects cycle time
  - Hit latency
- Cache misses affect **CPI**

26

Memory Stall Cycles

- The processor stalls on a Cache miss
  - When fetching instructions from the Instruction Cache (I-cache)
  - When loading or storing data into the Data Cache (D-cache)

  Memory stall cycles = Combined Misses × Miss Penalty
- Miss Penalty: clock cycles to process a cache miss

  Combined Misses = I-Cache Misses + D-Cache Misses

  I-Cache Misses = I-Count × I-Cache Miss Rate

  D-Cache Misses = LS-Count × D-Cache Miss Rate

  LS-Count (Load & Store) = I-Count × LS Frequency
- Cache misses are often reported per thousand instructions

27

Memory Stall Cycles Per Instruction

- Memory Stall Cycles Per Instruction =

  Combined Misses Per Instruction × Miss Penalty
- Miss Penalty is assumed equal for I-cache & D-cache
- Miss Penalty is assumed equal for Load and Store
- Combined Misses Per Instruction =

  I-Cache Miss Rate + LS-Frequency × D-Cache Miss Rate
- Therefore, Memory Stall Cycles Per Instruction =

  I-Cache Miss Rate × Miss Penalty +

  LS-Frequency × D-Cache Miss Rate × Miss Penalty

28

Example on Memory Stall Cycles

- Consider a program with the given characteristics
  - Instruction count (I-Count) = 10^6 instructions
  - 30% of instructions are loads and stores
  - D-cache miss rate is 5% and I-cache miss rate is 1%
  - Miss penalty is 100 clock cycles for instruction and data caches
  - Compute combined misses per instruction and memory stall cycles
- Combined misses per instruction in I-Cache and D-Cache
  - 1% + 30% × 5% = 0.025 combined misses per instruction
  - Equal to 25 misses per 1000 instructions
- Memory stall cycles
  - 0.025 × 100 (miss penalty) = 2.5 stall cycles per instruction
  - Total memory stall cycles = $10^6 \times 2.5$ = 2,500,000

29

CPU Time with Memory Stall Cycles

$$\text{CPU Time} = \text{I-Count} \times CPI_{MemoryStalls} \times \text{Clock Cycle}$$

$$CPI_{MemoryStalls} = CPI_{PerfectCache} + \text{Mem Stalls per Instruction}$$

- $CPI_{PerfectCache}$ = CPI for ideal cache (no cache misses)
- $CPI_{MemoryStalls}$ = CPI in the presence of memory stalls
- Memory stall cycles increase the CPI

30

Example on CPI with Memory Stalls

- A processor has CPI of 1.5 without any memory stalls
  - Cache miss rate is 2% for instruction and 5% for data
  - 20% of instructions are loads and stores
  - Cache miss penalty is 100 clock cycles
- What is the impact on the CPI?
- Answer:

Mem Stalls per Instruction = $\underbrace{0.02\times100}_{\text{Instruction}} + \underbrace{0.2\times0.05\times100}_{\text{data}} = 3$

$CPI_{MemoryStalls}$ = 1.5 + 3 = 4.5 cycles per instruction

$CPI_{MemoryStalls} / CPI_{PerfectCache}$ = 4.5 / 1.5 = 3

Processor is 3 times slower due to memory stall cycles

31

Improving Cache Performance

- Average Memory Access Time (AMAT)

  AMAT = Hit time + Miss rate * Miss penalty
- Used as a framework for optimizations
- Reduce the Hit time
  - Small & simple caches, avoid address translation for indexing
- Reduce the Miss Rate
  - Larger cache size, higher associativity, and larger block size
- Reduce the Miss Penalty
  - Multilevel caches, give priority to read misses over writes

32

Small and Simple Caches

- Hit time is critical: affects the processor clock cycle
  - Fast clock rate demands small and simple L1 cache designs
- Small cache reduces the indexing time and hit time
  - Indexing a cache represents a time consuming portion
  - Tag comparison also adds to this hit time
- Direct-mapped overlaps tag check with data transfer
  - Associative cache uses additional mux and increases hit time
- Size of L1 caches has not increased much
  - L1 caches are the same size on Alpha 21264 and 21364
  - Same also on UltraSparc II and III, AMD K6 and Athlon
  - Reduced from 16 KB in Pentium III to 8 KB in Pentium 4

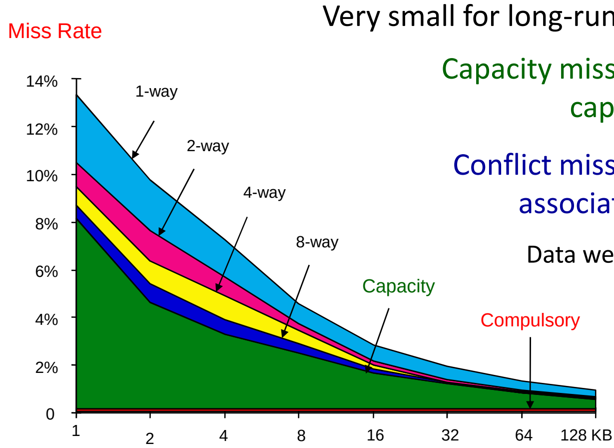
33

Classifying Misses: 3 C's [Hill]

- Compulsory Misses or Cold start misses
  - First-ever reference to a given block of memory
  - Measure: number of misses in an infinite cache model
  - Can be reduced with pre-fetching
- Capacity Misses
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced
  - Good replacement policy is crucial!
  - Measure: additional misses in a fully-associative cache
- Conflict Misses
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as *capacity within set*
  - Good replacement policy is crucial!
  - Measure: additional misses in cache of interest

34

Classifying Misses – cont'd



Compulsory misses are independent of cache size

Very small for long-running programs

Capacity misses decrease as capacity increases

Conflict misses decrease as associativity increases

Data were collected using LRU replacement

35

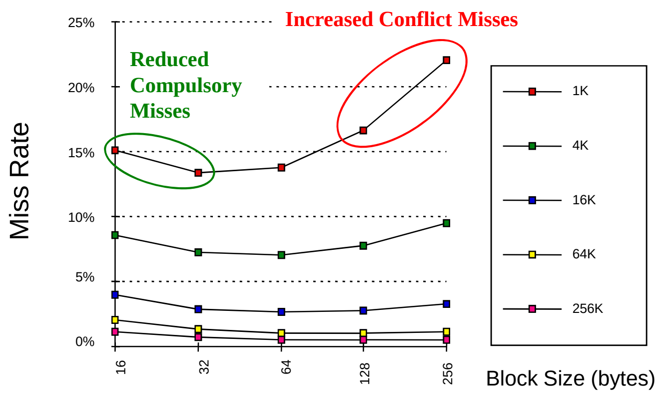
Six Basic Cache Optimizations

1. Larger block size to reduce miss rate
2. Larger caches to reduce miss rate
3. Higher associativity to reduce miss rate
4. Multilevel caches to reduce miss penalty
5. Give priority to read misses over writes to reduce miss penalty
6. Avoiding address translation for indexing the cache

36

Larger Block Size

- Simplest way to reduce miss rate is to increase block size
- However, it increases conflict misses if cache is small



64-byte blocks are common in L1 caches

128-byte block are common in L2 caches

37

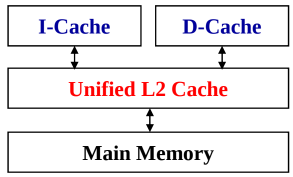
Larger Size and Higher Associativity

- Increasing cache size reduces capacity misses
- It also reduces conflict misses
  - Larger cache size spreads out references to more blocks
- Drawbacks: longer hit time and higher cost
- Larger caches are especially popular as 2^{nd} level caches
- Higher associativity also improves miss rates
  - Eight-way set associative is as effective as a fully associative

38

Multilevel Caches

- Top level cache should be kept small to
  - Keep pace with processor speed
- Adding another cache level
  - Can reduce the memory gap
  - Can reduce memory bus loading



- Local miss rate
  - Number of misses in a cache / Memory accesses to this cache
  - Miss $Rate_{L1}$ for L1 cache, and Miss $Rate_{L2}$ for L2 cache
- Global miss rate

  Number of misses in a cache/Memory accesses generated by CPU

  Miss $Rate_{L1}$ for L1 cache, and

  Miss $Rate_{L1}$ × Miss $Rate_{L2}$ for L2 cache

39

Multilevel Cache Policies

- Multilevel Inclusion
  - L1 cache data is always present in L2 cache
  - A miss in L1, but a hit in L2 copies block from L2 to L1
  - A miss in L1 and L2 brings a block into L1 and L2
  - A write in L1 causes data to be written in L1 and L2
  - Typically, write-through policy is used from L1 to L2
  - Typically, write-back policy is used from L2 to main memory
    - To reduce traffic on the memory bus
  - A replacement or invalidation in L2 must be propagated to L1

40

Multilevel Cache Policies – cont'd

- Multilevel exclusion
  - L1 data is never found in L2 cache – Prevents wasting space
  - Cache miss in L1, but a hit in L2 results in a swap of blocks
  - Cache miss in both L1 and L2 brings the block into L1 only
  - Block replaced in L1 is moved into L2
  - Example: AMD Opteron
- Same or different block size in L1 and L2 caches
  - Choosing a larger block size in L2 can improve performance
  - However different block sizes complicates implementation
  - Pentium 4 has 64-byte blocks in L1 and 128-byte blocks in L2

41

Multilevel Caches and CPI

$$CPI = CPI_{PerfectCache} + \sum_{l=1}^{n} Penalty_l \times MPI_l$$

- $Penalty_l$ is miss penalty at each of n levels of cache
- MPI_l is miss rate per instruction at each cache level
- Miss rate specification:
  - Misses Per Instruction: easy to incorporate in CPI
  - Misses Per Reference: must convert to per instruction
    - Local: misses per local reference
    - Global: misses per ifetch or load or store

42

Cache Performance Example

- Assume following:
  - $CPI_{PerfectCache}$ = 1.15 (if no cache misses)
  - L1 instruction cache: hit rate = 98% per instruction
  - L1 data cache: hit rate = 96% per instruction
  - Shared L2 cache: local miss rate = 40% per reference
  - L1 miss penalty of 8 cycles
  - L2 miss penalty of:
    - 10 cycles latency to request word from memory
    - 2 cycles per 16B bus transfer, 4x16B = 64B block transferred
    - Hence 8 cycles transfer plus 1 cycle to fill L2
    - Total L2 miss penalty = 10+8+1 = 19 cycles

43

Cache Performance Example

$$CPI = CPI_{PerfectCache} + \sum_{l=1}^{n} Penalty_l \times MPI_l$$

$$CPI = 1.15 + \frac{8cycles}{miss} \times \left(\frac{0.02miss}{inst} + \frac{0.04miss}{inst} \right) + \frac{19cycles}{miss} \times \frac{0.40miss}{ref} \times \frac{0.06ref}{inst}$$

$$= 1.15 + 0.48 + \frac{19cycles}{miss} \times \frac{0.024miss}{inst}$$

$$= 1.15 + 0.48 + 0.456 = 2.086$$

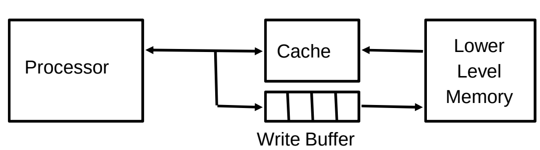
44

Cache Misses and Performance

- CPI equation
  - Only holds for misses that cannot be overlapped with other activity
  - Store misses often overlapped
    - Place store in store queue
    - Wait for miss to complete
    - Perform store
    - Allow subsequent instructions to continue in parallel
  - Modern out-of-order processors also do this for loads
    - Cache performance modeling requires detailed modeling of entire processor core

Give Priority to Read Misses over Writes

- Write buffer:
  - Decouples CPU write from the memory bus writing
- Write-through: all stores are sent to write buffer
  - Eliminates processor stalls on writes until buffer is full
- Write-back: modified blocks are written when replaced
  - Write buffer used for evicted blocks to be written back
- Write buffer content should be checked on a read miss
  - Let the read miss continue if there is no conflict, given read misses priority over writes



Avoid Address Translation for indexing

- Modern systems use virtual memory
- Virtual Addresses are generated by programs
- We can use the virtual address to index the cache
  - While translating the virtual address to a physical address
- Virtual Cache is addressed by a virtual address
  - Address translation and cache indexing are done in parallel
- Physical Cache is addressed by a physical address
- However, virtual caches cause problems
  - Page level protection should be checked
  - Cache flushing and Process identifier tag (PID)
  - Aliasing: 2 virtual addresses mapping to same physical address

More on Block Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where *a* = *associativity*
  - If *a* <= 2, LRU is cheap and easy (1 bit)
  - If *a* > 2, it gets harder
  - Pseudo-random works pretty well for caches

48

Optimal Replacement Policy?

[Belady, IBM Systems Journal, 1966]

- Evict block with longest reuse distance
  - i.e. Block to replace is referenced farthest in future
  - Requires knowledge of the future!
- Can't build it, but can model it with trace
  - Process trace in reverse
  - [Sugumar&Abraham] describe how to do this in one pass over the trace with some lookahead (Cheetah simulator)
- Useful, since it reveals ***opportunity***

49

Random and FIFO Replacement

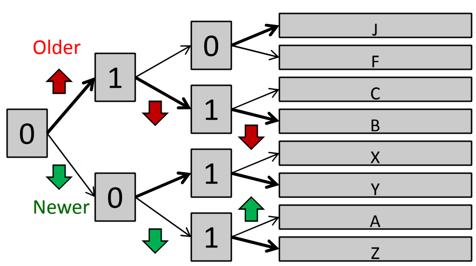
- Number of blocks to choose from a set = *a* blocks
- Random replacement
  - Candidate block is randomly selected
  - One counter for all sets: incremented on every cycle
  - $Log_2(a)$ bit Counter: counts from 0 to $a - 1$
  - On a cache miss replace block specified by counter
- First In First Out (FIFO) replacement
  - Replace oldest block in set
  - One counter per set: specifies oldest block to replace
  - $Log_2(a)$ bit counter per set
  - Counter is incremented on a cache miss

50

Least-Recently Used

- For a=2, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access
- For a>2, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
- List overhead: $a \times \log_2(a)$ bits per set
  - Shift register implementation

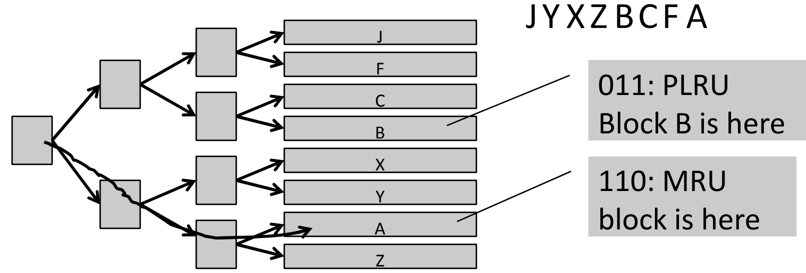
Practical Pseudo-LRU



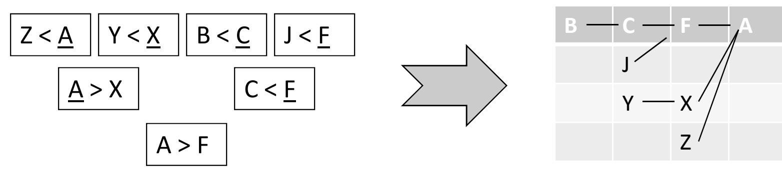
- Rather than true LRU, use binary tree
- Each node records which half is older/newer
- Update nodes on each reference
- Follow older pointers to find LRU victim

Practical Pseudo-LRU In Action

J Y X Z B C F A



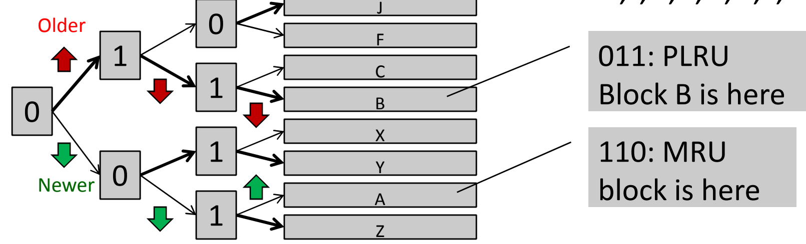
Partial Order Encoded in Tree:



53

Practical Pseudo-LRU

Refs: J,Y,X,Z,B,C,F,A



- Binary tree encodes PLRU *partial order*
  - At each level point to LRU half of subtree
- Each access: flip nodes along path to block
- Eviction: follow LRU path
- Overhead: (*a*-1) bits per set

54

LRU Shortcomings

- Streaming data/scans: $x_0, x_1, \ldots, x_n$
  - Effectively no temporal reuse
- Thrashing: *reuse distance > a*
  - Temporal reuse exists but LRU fails
- All blocks march from MRU to LRU
  - Other conflicting blocks are pushed out
- For *n>a* no blocks remain after scan/thrash
  - Incur many conflict misses after scan ends
- Pseudo-LRU sometimes helps a little bit

55

LRU Insertion Policy: LIP

- Memory-intensive: working set > cache size
  - Cache block goes from MRU to LRU without receiving any cache hit
- Insert new blocks into LRU, not MRU position
  - Qureshi et al. ISCA 2007
- Dynamic Insertion Policy: DIP (Adaptive)
  - Use *set dueling* to decide LIP vs. traditional LRU
  - 1 (or a few) set uses LIP vs. 1 that uses LRU
  - Compare hit rate for sets
  - Set policy for all other sets to match best set

56

Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim
    - Can be pseudo-random, or rotating, or fixed left-to-right
- Simple, similar to virtual memory clock algorithm
- Provides some scan and thrash resistance
  - Relies on “randomizing” evictions rather than strict LRU order
- Used by Intel Itanium, Sparc T2

57