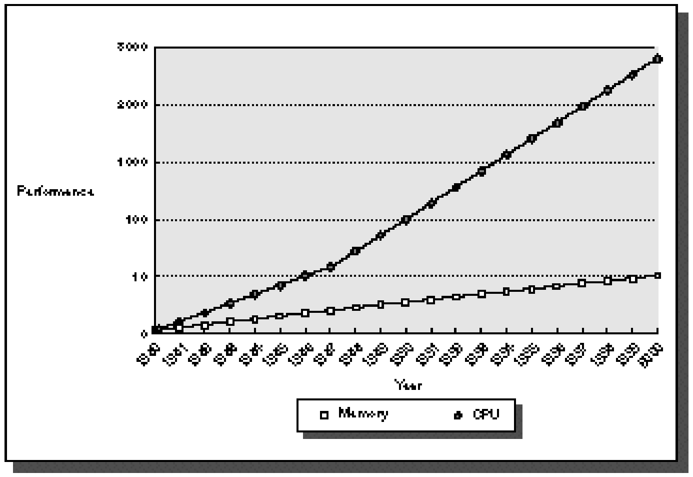
Lecture 11: Memory Hierarchy Design

Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu

http://www-leland.stanford.edu/class/ee282h/

CPU-Memory Performance Gap



The Memory Bottleneck

- Typical CPU clock rate
  - » 300MHz (3.3ns cycle time)
- Typical DRAM access time
  - » 60ns (about 20 cycles)
- Typical main memory access
  - » 200ns (60 cycles)
    - – DRAM (60), precharge (40), chip crossings (50), overhead (50).
- Our pipeline designs assume 1 cycle access (3.3ns)
- Average instruction references
  - » 1 instruction word
  - » 0.3 data words
- This problem gets worse
  - » CPUs get *faster*
  - » Memories get *bigger*
- Memory delay is mostly communication time
  - » reading/writing a bit is *fast*
  - » it takes time to
    - – select the right bit
    - – route the data to/from the bit
- Big memories are *slow*
- Small memories can be made *fast*

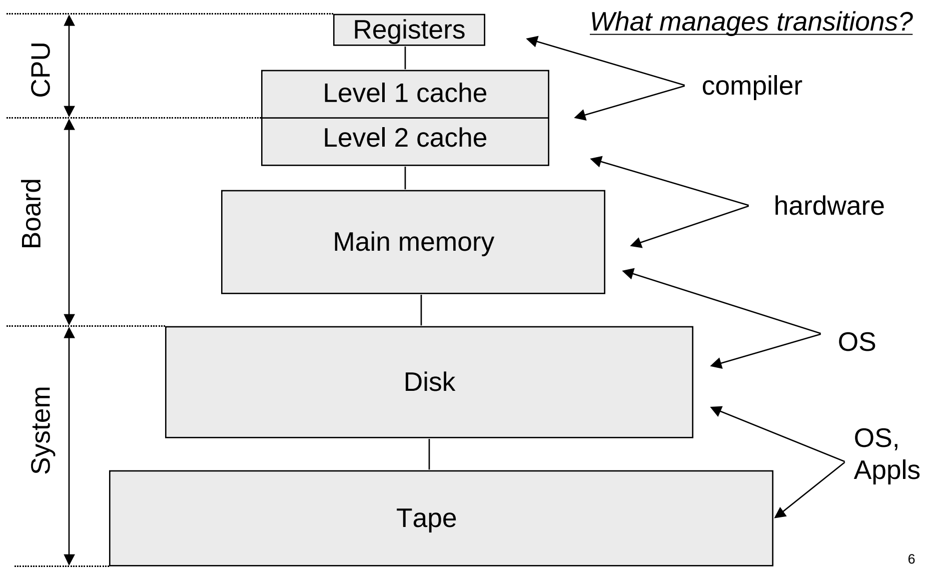
Memory design

- Outline
  - » Cache memory designs
  - » Simple cache memory performance analysis
  - » Improving performance
    - – Reducing misses
    - – Reducing miss penalty
    - – Reducing hit time
  - » Main memory techniques
    - – Interleaving, etc.
  - » Virtual memory
    - – by itself
    - – interactions with cache

Memory Hierarchies

- Our pipelines have assumed memory access takes one cycle!
- We can’t use large amounts of fast memory
  - » expensive in dollars, watts, and space
  - » even fast chips make slow big memory systems
- Tradeoff cost-speed and size-speed using a hierarchy of memories:
  - » small, fast, expensive caches at the top
  - » large, slow, and cheap mass memory at the bottom
- Ideal: the composite should be almost as fast as the top level, and almost as big and cheap as the bottom level
- Why it works: exploit program locality (“working set”)
  - » spatial: nearby addresses are likely to be referenced soon
    - – Ifetch, array
  - » temporal: the same address is likely to be re-referenced
    - – variables, stack
- Pipelining and and caches are the fundamental design ideas used in uniprocessors for the last two decades.

Memory hierarchies: the levels

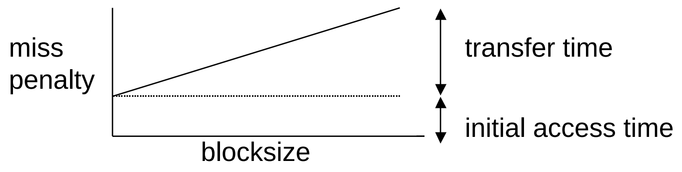


Memory hierarchies: Typical level parameters

	Quantity	Speed	Cost
Registers	512 bytes	1	?
L1 cache	32 KB	2	?
L2 cache	512 KB	10	$200/MB
Main memory	32 MB	100	$50/MB

Terminology

- Lower levels are closer to the CPU
- At each level a block is the minimum amount checked for and transferred
  - » block size is a power of 2
  - » block sizes at higher levels are usually fixed multiples of block sizes at lower levels
  - » memory address: | block address | offset in block |
- A reference at level N is a hit if it is found at that level
  - » hit rate = # hits at level N / # references at level N
  - » miss rate = 1 - hit rate
- The access time of a hit = hit time, usually = miss determination time
- Miss penalty = initial access time + transfer time



Memory hierarchy performance

- Miss rate is only one component of performance
- Average memory access time for one-level cache
  - » AMAT = hit time + miss rate x miss penalty
  - » hit time = miss determination time
- Average memory access time for two-level cache
  - » AMAT = L1 hit time +
    (L1 miss rate x L2 hit time) + (L2 miss rate x L2 miss penalty)
- Example:
  - » L1 cache: 90% hits, 10 ns
  - » L2 cache: 95% hits, 100 ns
  - » L2 miss: 200 ns
  - » AMAT = 10 ns + (0.1 x 100 ns) + (.05 x 200 ns)
    = 10 ns + 10 ns + 10 ns
    = 30 ns
- Need to factor in CPU performance to understand the effect. (later)

Cache size tradeoff

- The largest effect on miss rate is the total cache size
- Compare L1 (on-chip) to L2 (on-board) cache:

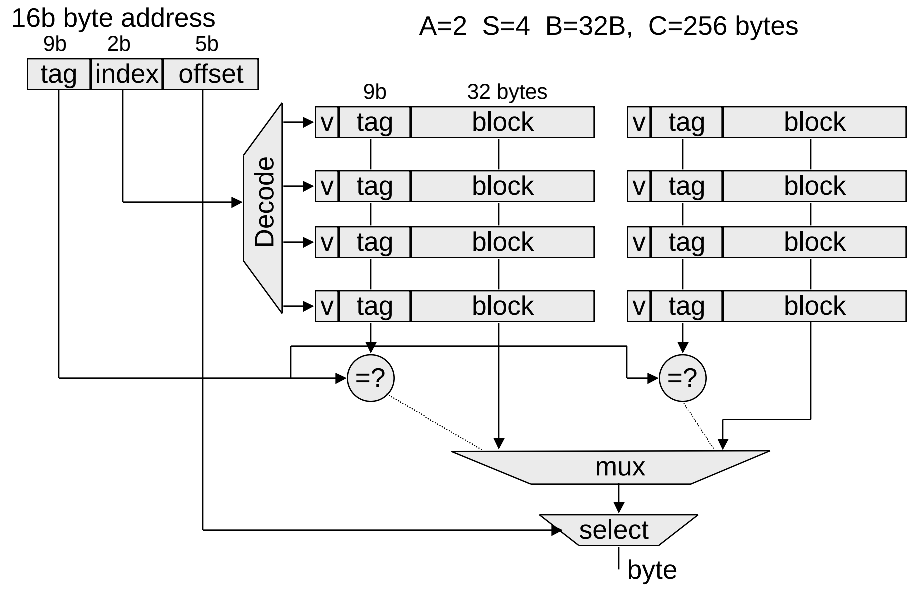
	L1 only	L2 only	Both
Hit rate	90%	95%	
Hit time	10 ns	20 ns	
Miss penalty	180 ns	180 ns	
AMAT	10+.1x180 =28 ns	20+.05x180 =29 ns	10 + 0.1x20 + 0.5x180 = 21 ns

Cache design

- Ideal cache:
  - » Each word is stored separately
  - » Any word can be stored anywhere in the cache
  - » Requires too many comparators!
- Realistically
  - » Group words into blocks (“lines”) of consecutive words
  - » Limit the number of places a word (block) can be found
- Organizational parameters
  - » B (block size): the minimum unit of lookup or transfer
  - » A (associativity, set size): the number of places (“elements in set”) a particular block could go into
  - » S (number of sets): the number of groups of places
  - » C (cache size): the total cache size

    C = B x A x S
  - » F (fetch size)

General Case: The Classic Cache:



The cache organization extremes

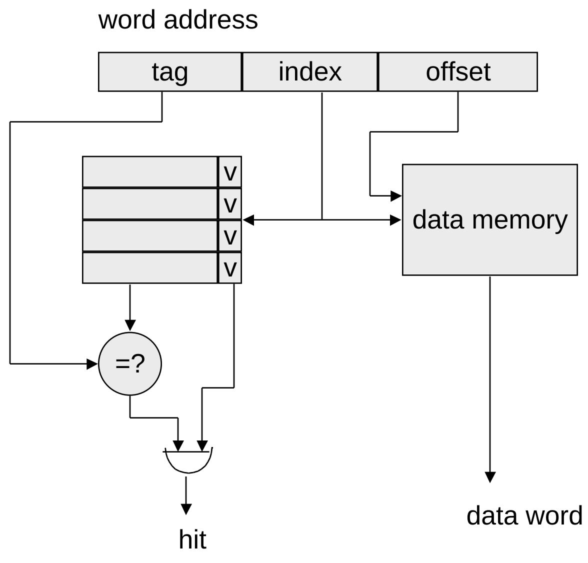
- For a given total cache size, we can trade off between hit rate and complexity
- If L = number of lines (blocks) in the cache, L = C/B

How many places (A)	Name	Number of sets (S)
1	direct-mapped	L
n	n-way set associative	L/n
L	fully associative	1

⇧

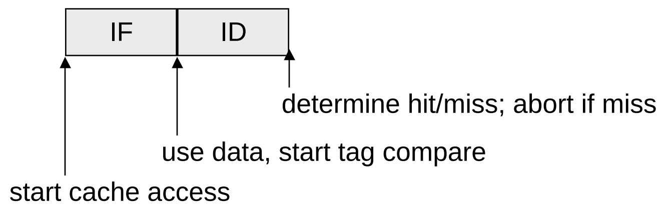
number of comparators

The Direct Mapped Cache



The Direct Mapped Cache

- Lowest access time:
  - » no multiplexer for the data
  - » the data can be speculatively used before the tag comparison is done



IF
ID
determine hit/miss; abort if miss
use data, start tag compare
start cache access

- Potentially bad hit rate:
  - » One location for each word in the cache
  - » pathlogical cases can easily create 0% hit rates

The fully associative cache

- Any word can go anywhere
- Best hit rate
  - » especially matters for small caches, where the odds of conflicts in a direct-mapped cache are high
- Requires:
  - » many comparators
  - » very wide memories and data paths
    - – appropriate within a chip, less so on a board

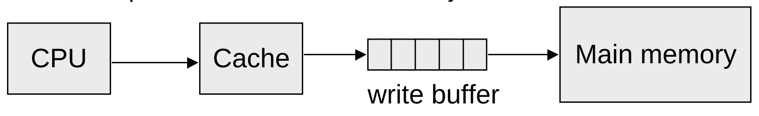
Read Miss
Cache Replacement Policy

- A cache miss requires a block to be replaced. Which one?
- For direct-mapped caches, there is no choice.
- For set-associative or fully associative,
  - » Ideal/impossible: the block that will be used farthest in the future, or never
  - » LRU: the block that was used least recently in the past
    - – close to ideal because of temporal locality
    - – easy to do for low associativity
    - – can be approximated for high associativity
  - » Random: pick a (psuedo-) random block.
    - – Shift register based on clock cycle?
    - – Surprise: almost as good as LRU!

	2-way		**4-way**		**8-way**	
Size	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Write hit strategies

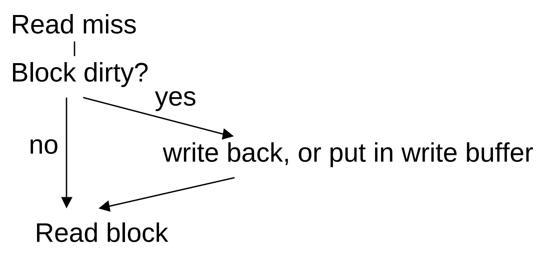
- Write hits are slower than read hits:
  - » Check tag
  - » THEN write data
- #1: Write-through
  - » Modify the cache, but then also immediately modify main memory
  - » Keeps main memory always up-to-date
    - – Good for multiprocessors
    - – Good for independent I/O (DMA)
  - » Easy to implement
  - » Speeds the common case: read misses do no writes
  - » Create a high volume of main memory bus (off-chip) traffic
    - – With 100 MIPS processor and 10% word stores, 40 MB/s
    - – Arrival rate is non-uniform, so we can add a write buffer to smooth the peaks and reduce write latency



(What about reads of data in the write buffer? What about 2 writes to the same block?)

Write hit strategies

- #2: Write back (copy back)
  - » Write only to the cache, and mark the cache block “dirty”
  - » Writes are faster
  - » but reads (if they replace a dirty block) may be slower
  - » Lowers traffic to main memory
    - – repetitive writes don’t cause traffic
    - – writes to adjacent words don’t cause traffic
  - » Main memory is not up-to-date (I/O and multiprocessor problems)



Write miss strategies

- If a word being written is *not* in the cache, what happens?
- #1: Write allocate, fetch-on-write
    - – fetch the block as if a read miss
    - – then treat it as a write hit
  - » Generally used by write-back caches, hoping that subsequent writes will hit in the cache
- #2: Write allocate, no fetch-on-write
    - – allocate the cache block, but keep individual VALID bits for each word
    - – complicates reads (multiple bits to check) and writebacks (may have to write discontiguous words in a block)
- #2: No write allocate (write around)
    - – write (or buffer) to memory without changing cache
  - » Generally used by write-through, since subsequent write have to go to main memory anyway

Write miss actions

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick replacment	pick replacment
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

Back-of-the-envelope Cache performance calculations

CPUtime = (CPU_exec_cycles + memory_stall_cycles) x CCT

Includes cache hits and hazard stalls

Includes miss penalty, write buffer stalls, virtual memory stalls

$$= IC \times (CPI_{execution} + CPI_{memory\_stall}) \times CCT$$

$$= IC \times \left(CPI_{execution} + \frac{Misses}{instr} \times miss\_penalty\right) \times CCT$$

$$= IC \times \left(CPI_{execution} + \frac{memrefs}{instr} \times miss\_rate \times miss\_penalty\right) \times CCT$$

Compare to:

AMAT = (1 + miss_rate x miss_penalty) x CCT

Cache Performance
Simple examples

- Assume:
  - » 1.4 memrefs/instr
  - » 10% miss rate
  - » 10 cycle miss penalty
- Then
  - » CPUtime = IC x (CPI$_{exec}$ + 1.4 x 0.1 x 10) x CCT
    = IC x (CPI$_{exec}$ + 1.4) x CCT
  - » A disaster for RISC; if CPI=1.4, 50% of time is spent in memory stall
  - » Not so bad for CISC; if CPI=8, 15% of time is spent in memory stall
- Decreasing CPI and CCT magnifies the memory system effect, since memory stall time is often independent of both.

Direct mapped vs. associative, and CPUtime vs AMAT

- Assume:
  - » 1.5 memrefs/instr
  - » Base CPI = 2.0
  - » Miss penalty = 500 ns
- Direct mapped cache:
  - » 5% miss rate, 50 ns CCT
  - » CPUtime = IC x (2.0 + 1.5 x 0.05 x ceil(500/50)) x 50
    = 137.5 x IC
  - » AMAT = 50 + .05 x 500 = 75 ns
- 2-way set associative:
  - » 4% miss rate, 60 ns CCT (mux, tag check)
  - » Assume miss penalty time doesn't change
  - » CPUtime = IC x (2.0 + 1.5 x 0.04 x ceil(500/60)) x 60
    = 152 x IC
  - » AMAT = 60 + .04 x 500 = 80 ns
- By CPUtime, direct mapped cache is 11% faster;
  By AMAT, direct mapped cache is only 7% faster

Understanding where misses come from: An intuitive model

- Compulsory: unavoidable first reference to a block
- Capacity: misses caused because the cache is too small
- Conflict: misses caused by mapping conflicts in the cache

Type	Measure by	Reduce by	Impact
Compulsory (C1)	C1 = MR of infinite cache	increase block size	C2 and C3 can increase
Capacity (C2)	C2 = MR of fully associative cache - C1	increase cache size	increased cycle time and cost
Conflict (C3)	C3 = MR of set associate cache - C2	increase associativity	increased cycle time and cost

- Just a way to think about misses; reality is more complicated
  - » Increasing cache size can eliminate some conflict misses.