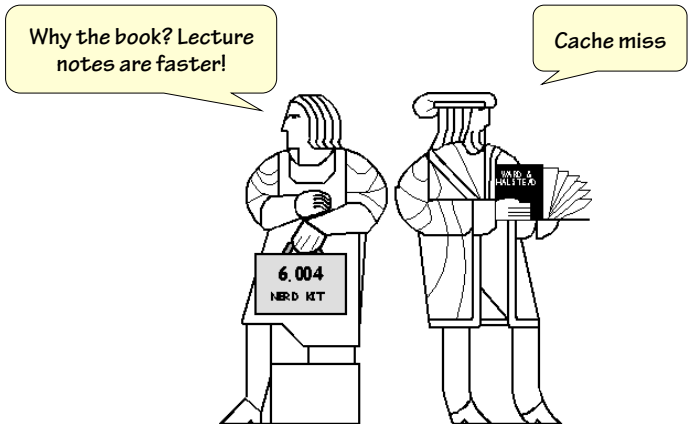
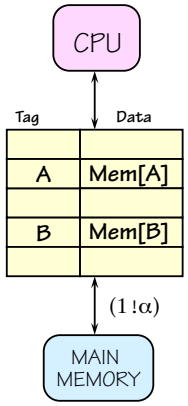
6.004 Computation Structures
Spring 2009

Cache Issues



Quiz #3 Friday!

Basic Cache Algorithm



ON REFERENCE TO Mem[X]:
Look for X among cache tags...

HIT: *X = TAG(i) , for some cache line i*
- READ: return DATA(i)
- WRITE: change DATA(i); Start Write to Mem(X)

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X] (Allocation)
- READ: Read Mem[X]
  Set TAG(k)=X, DATA(K)=Mem[X]
- WRITE: Start Write to Mem(X)
  Set TAG(k)=X, DATA(K)= new Mem[X]

Cache Design Issues

Associativity – a basic tradeoff between
- Parallel Searching (expensive) vs
- Constraints on which addresses can be stored where

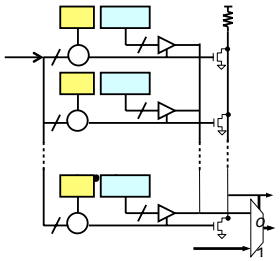
Replacement Strategy:
- OK, we've missed. Gotta add this new address/value pair to the cache. What do we kick out?
  - *Least Recently Used:* discard the one we haven't used the longest.
  - Plausible alternatives, (e.g. random replacement.

Block Size:
- Amortizing cost of tag over multiple words of data

Write Strategy:
- When do we write cache contents to main memory?

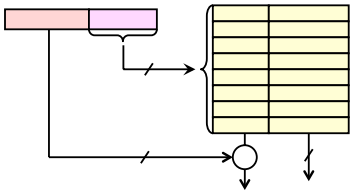
Associativity

Lots...



Fully Associative
- expensive!
- flexible: any address can be cached in any line

... or NONE!



Direct Mapped
- cheap (ordinary SRAM)
- contention: addresses compete for cache lines

Direct-Mapped Cache Contention

	Memory Address	Cache Line	Hit/ Miss
Loop A: Pgm at 1024, data at 37:	1024	0	HIT
	37	37	HIT
	1025	1	HIT
	38	38	HIT
	1026	2	HIT
	39	39	HIT
	1024	0	HIT
	...		
Loop B: Pgm at 1024, data at 2048:	1024	0	MISS
	2048	0	MISS
	1025	1	MISS
	2049	1	MISS
	1026	2	MISS
	2050	2	MISS
	1024	0	MISS
	...		

Works GREAT here…

Assume 1024-line direct-mapped cache, 1 word/line. Consider tight loop, at steady state: (assume WORD, not BYTE, addressing)

… but not here!

We need *some* associativity, But not *full* associativity…

Fully-assoc. vs. Direct-mapped

Fully-associative N-line cache:

- N tag comparators, registers used for tag/data storage ($$$)
- Location A might be cached in any one of the N cache lines; no restrictions!
- Replacement strategy (e.g., LRU) used to pick which line to use when loading new word(s) into cache
- PROBLEM: Cost!

Direct-mapped N-line cache:

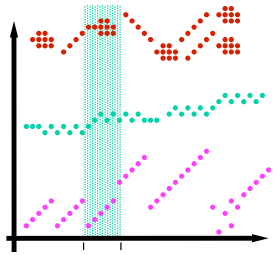
- 1 tag comparator, SRAM used for tag/data storage ($)
- Location A is cached in a specific line of the cache determined by its address; address "collisions" possible
- Replacement strategy not needed: each word can only be cached in one specific cache line
- PROBLEM: Contention!

Cost vs Contention

two observations...

1. Probability of collision diminishes with cache size...
   ... so lets build HUGE direct-mapped caches, using cheap SRAM!
2. Contention mostly occurs between independent "hot spots" --
   - Instruction fetches vs stack frame vs data structures, etc
   - Ability to simultaneously cache a few (2? 4? 8?) hot spots eliminates most collisions

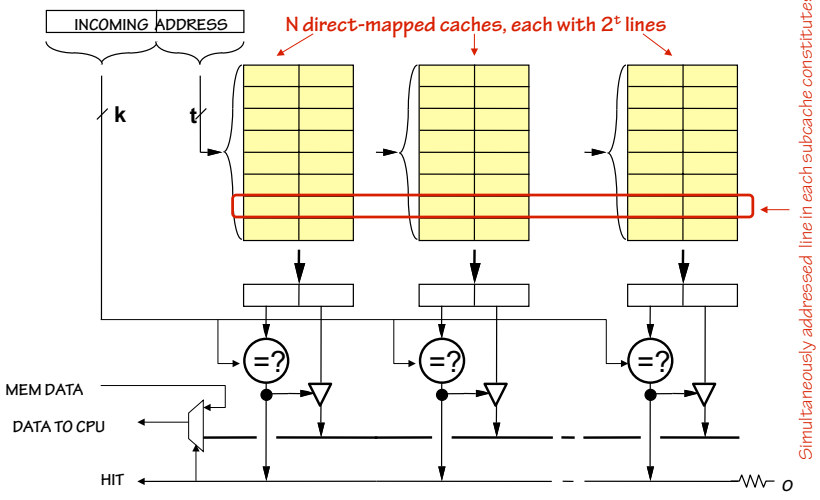
   ... so lets build caches that allow each location to be stored in some restricted set of cache lines, rather than in exactly one (direct mapped) or every line (fully associative).



Insight: an N-way *set-associative* cache affords modest parallelism

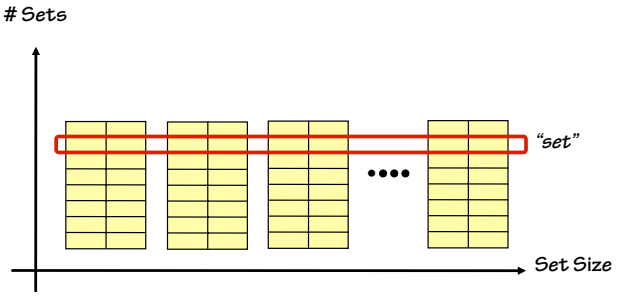
- parallel lookup (associativity): restricted to small set of N lines
- modest parallelism deals with most contention at modest cost
- can implement using N direct-mapped caches, running in parallel

N-way Set-Associative Cache



Associativity

the birds-eye view

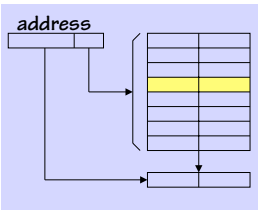
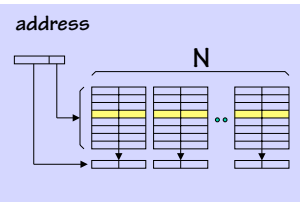
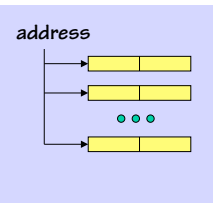


Can place caches in 2D space:

- Total lines = # Sets * Set Size
- # Sets = 1: Fully Associative
- Set Size = 1: Direct Mapped
- Set Size = N: N-way Set Associative

Associativity implies choices…

ISSUE: Replacement Strategy

Direct-mapped	N-way set associative	Fully associative
		
• compare addr with only one tag	• compare addr with N tags simultaneously	• compare addr with each tag simultaneously
• location A can be stored in exactly one cache line	• location A can be stored in exactly one set, but in any of the N cache lines belonging to that set	• location A can be stored in any cache line

Replacement Strategy

(0,1,2,3) Hit 2 -> (2,0,1,3)
(2,0,1,3) Hit 1 -> (1,2,0,3)
(1,2,0,3) Miss -> (3,1,2,0)
(3,1,2,0) Hit 3 -> (3,1,2,0)

LRU (Least-recently used)

- keeps most-recently used locations in cache
- need to keep ordered list of N items → N! orderings
  → $O(\log_2 N!) = O(N \log_2 N)$ "LRU bits" + complex logic

Overhead is $O(N \log_2 N)$ bits/set

FIFO/LRR (first-in, first-out/least-recently replaced)

- cheap alternative: replace oldest item (dated by access time)
- within each set: keep one counter that points to victim line

Overhead is $O(\log_2 N)$ bits/set

Random (select replacement line using random, uniform distribution)

- no "pathological" reference streams causing wost-case results
- use pseudo-random generator to get reproducible behavior;
- use *real* randomness to prevent reverse engineering!

Overhead is $O(\log_2 N)$ bits/cache!

Cache Benchmarking

Suppose this loop is entered with R3=4000:

```
ADR:   Instruction     I     D
400:   LD(R3,0,R0)     400   4000+...
404:   ADDC(R3,4,R3)   404
408:   BNE(R0,400)     408
```

GOAL: Given some cache design, simulate (by hand or machine) execution well enough to determine hit ratio.

1. Observe that the sequence of memory locations referenced is

   400, 4000, 404, 408, 400, 4004, ...

   We can use this simpler reference string, rather than the program, to simulate cache behavior.

2. We can make our life easier in many cases by converting to *word* addresses: 100, 1000, 101, 102, 100, 1001, ...

   (Word Addr = (Byte Addr)/4)

Cache Simulation

4-line Fully-associative/LRU

Addr	Line#	Miss?
100	0	M
1000	1	M
101	2	M
102	3	M
100	0	
1001	1	M
101	2	
102	3	
100	0	
1002	1	M
101	2	
102	3	
100	0	
1003	1	M
101	2	
102	3	

Compulsory Misses

1/4 miss

4-line Direct-mapped

Addr	Line#	Miss?
100	0	M
1000	0	M
101	1	M
102	2	M
100	0	M
1001	1	M
101	1	M
102	2	
100	0	
1002	2	M
101	1	
102	2	M
100	0	
1003	3	M
101	1	
102	2	

Collision

Collision miss

7/16 miss

Associativity: Full vs 2-way

8-line Fully-associative, LRU

Addr	Line#	Miss?
100	0	M
1000	1	M
101	2	M
102	3	M
100	0	
1001	4	M
101	2	
102	3	
100	0	
1002	5	M
101	2	
102	3	
100	0	
1003	6	M
101	2	
102	3	

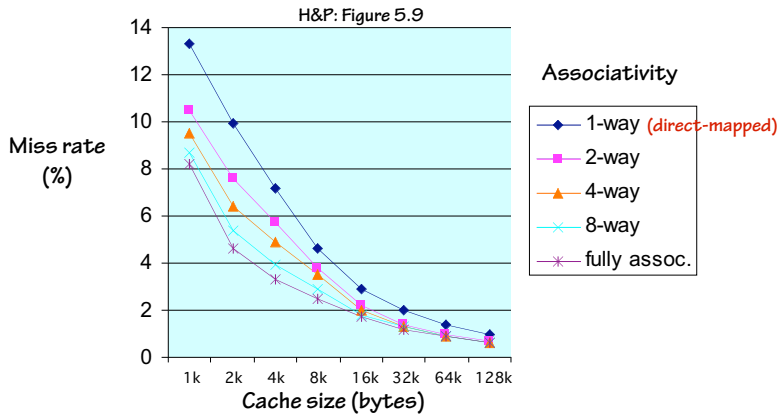
1/4 miss

2-way, 8-line total, LRU

Addr	Set#/N	Miss?
100	0,0	M
1000	0,1	M
101	1,0	M
102	2,0	M
100	0,0	
1001	1,1	M
101	1,0	
102	2,0	
100	0,0	
1002	2,1	M
101	1,0	
102	2,0	
100	0,0	
1003	3,1	M
101	1,0	
102	2,0	

1/4 miss

Associativity vs. miss rate



- 8-way is (almost) as effective as fully-associative
- rule of thumb: N-line direct-mapped == N/2-line 2-way set assoc.

Devil's Advocacy Games

Your company uses the cheaper FIFO cache, the competition uses LRU. Can you devise a benchmark to make your cache look better?

Assume 0x100 sets, 2-way…

Set 0 tags:	2-way, LRU #0: 1000, #1: 2000		2-way, FIFO #0: 2000, #1: 1000	
Adr	Set, #	H/M	Set, #	H/M
100	0, 0	M	0, 0	M
1000	0, 1	M	0, 1	M
100	0, 0	H	0, 0	H
2000	0, 1	M	0, 0	M
1000	0, 0	M	0, 0	H

BINGO!

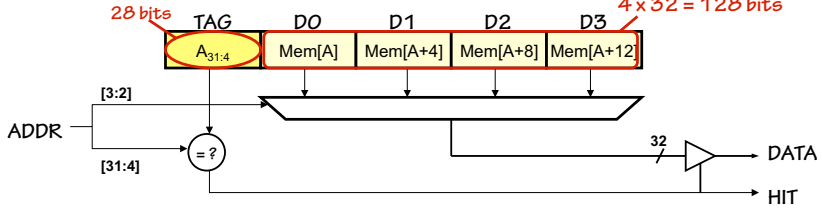
A carefully-designed benchmark can make *either* look better…
Pessimal case: next adr referenced is the one just replaced!
Random replacement makes this game harder…

Increasing Block Size

More Data/Tag

Overhead < ¼ bit of Tag per bit of data

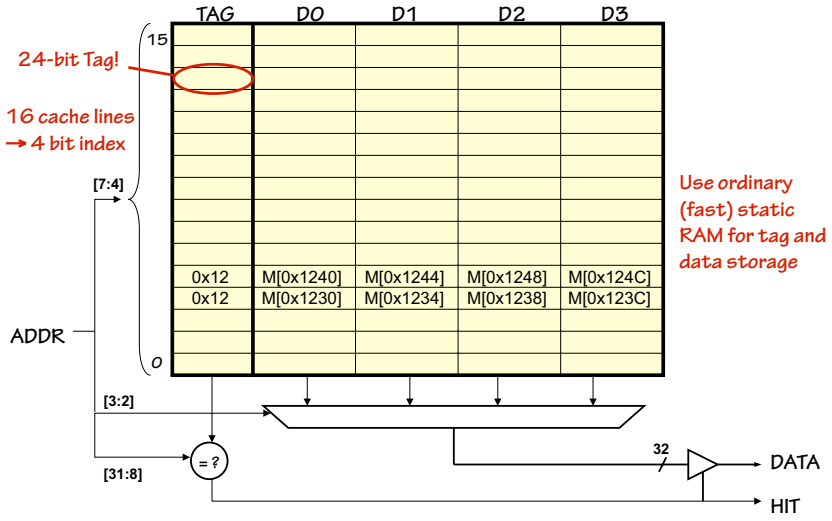
Enlarge each line in cache:



- blocks of 2^B words, on 2^B word boundaries
- always read/write 2^B word block from/to memory
- locality: access on word in block, others likely
- cost: some fetches of unaccessed words

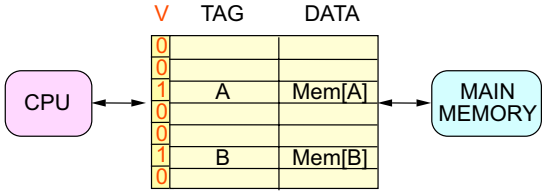
BIG WIN if there is a wide path to memory

4-word block, DM Cache



Only one comparator for entire cache!

Valid bits



Problem:

- Ignoring cache lines that don't contain anything of value... e.g., on
  - start-up
  - "Back door" changes to memory (eg loading program from disk)

Solution:

- Extend each TAG with VALID bit.
- Valid bit must be set for cache line to HIT.
- At power-up / reset : clear all valid bits
- Set valid bit when cache line is first *replaced.*
- Cache Control Feature: *Flush* cache by clearing all valid bits, Under program/external control.

Handling of WRITES

Observation: Most (90+%) of memory accesses are *READs*. How should we handle writes? Issues:

Write-through: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds "the truth".

Write-behind: CPU writes are cached; writes to main memory may be *buffered,* perhaps pipelined. CPU keeps executing while writes are completed (in order) in the background.

Write-back: CPU writes are cached, but not immediately written to main memory. Memory contents can be "stale".

Our cache thus far uses write-through.

Can we improve write performance?

Write-through

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X == TAG(i) , for some cache line i*

- READ: return DATA[I]
- WRITE: change DATA[I]; Start Write to Mem[X]

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X]
- READ: Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X]
- WRITE: Start Write to Mem[X]
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Write-back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

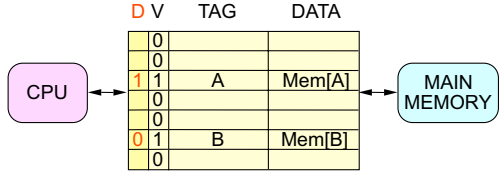
- READ: return DATA(i)
- WRITE: change DATA(i); ~~Start Write to Mem~~[X]

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X]
  - Write Back: Write Data(k) to Mem[Tag[k]]
- READ: Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X]
- WRITE: ~~Start Write to Mem[X]~~
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Is write-back worth the trouble? Depends on (1) cost of write; (2) consistency issues.

Write-back w/ "Dirty" bits



ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

- READ: return DATA(i)
- WRITE: change DATA(i); ~~Start Write to Mem~~[X] D[i]=1

MISS: *X not found in TAG of any cache line*

- REPLACEMENT SELECTION:
  - Select some line k to hold Mem[X]
  - If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]
- READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
- WRITE: ~~Start Write to Mem~~[X] D[k]=1
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Caches: Summary

Associativity:

- Less important as size increases
- 2-way or 4-way usually plenty for typical program clustering; BUT additional associativity
  - Smooths performance curve
  - Reduces number of select bits (we'll see shortly how this helps)
- TREND: Invest in RAM, not comparators.

Replacement Strategy:

- BIG caches: any sane approach works well
- REAL randomness assuages paranoia!

Performance analysis:

- Tedious hand synthesis may build intuition from simple examples, BUT
- Computer simulation of cache behavior on REAL programs (or using REAL trace data) is the basis for most real-world cache design decisions.