inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #24 Cache II

2007-8-6

Scott Beamer, Instructor

New Flow Based Routers

ANAGRAN
Go with the flow.™

www.anagran.com

Caching Terminology

- When we try to read memory, 3 things can happen:

1. cache hit: cache block is valid and contains proper address, so read desired word
2. cache miss: nothing in cache in appropriate block, so fetch from memory
3. cache miss, block replacement: wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

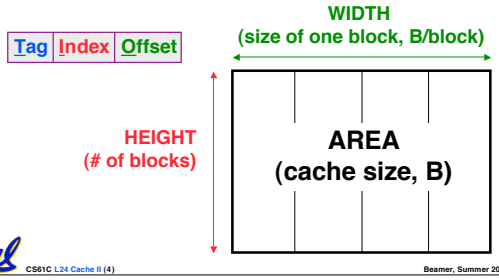
Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- Index: specifies the cache index (which "row" of the cache we should look in)
- Offset: once we've found correct block, specifies which byte within the block we want -- I.e., which "column"
- Tag: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

TIO Dan's great cache mnemonic

AREA (cache size, B)
= HEIGHT (# of blocks)
* WIDTH (size of one block, B/block)

$2^{(H+W)} = 2^H * 2^W$



16 KB Direct Mapped Cache, 16B blocks

- Valid bit: determines whether anything is stored in that row (when computer initially turned on, all entries invalid)

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

1. Read 0x00000014

- 00000000000000000 0000000001 0100

Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

So load that data into cache, setting tag, valid

• 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read from cache at offset, return word b

• 00000000000000000000 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

2. Read 0x0000001C = 0…00 0..001 1100

• 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index is Valid

• 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index valid, Tag Matches

• 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index Valid, Tag Matches, return d

• 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Types of Cache Misses

- Compulsory Misses
  - When program starts, nothing is loaded
- Conflict Misses
  - Two (or more) needed blocks map to the same cache location
  - Fixed by Fully Associative Cache
- Capacity Misses
  - Not enough room to hold it all
  - Can be fixed by bigger cache

Fully Associative Cache

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
- What does this mean?
  - no "rows": any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

What to do on a write hit?

- Write-through
  - update the word in cache block and corresponding word in memory
- Write-back
  - update word in cache block
  - allow memory word to be "stale"

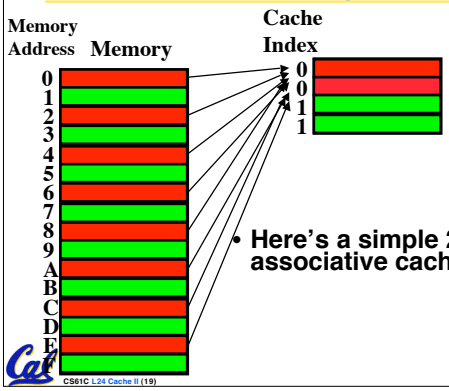
  ⇒ add 'dirty' bit to each block indicating that memory needs to be updated when block is replaced

  ⇒ OS flushes cache before I/O…
- Performance trade-offs?

N-Way Set Associative Cache (1/3)

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: points us to the correct "row" (called a set in this case)
- So what's the difference?
  - each set contains multiple blocks
  - once we've found correct set, must compare with all tags in that set to find our data

Associative Cache Example



- Here's a simple 2 way set associative cache.

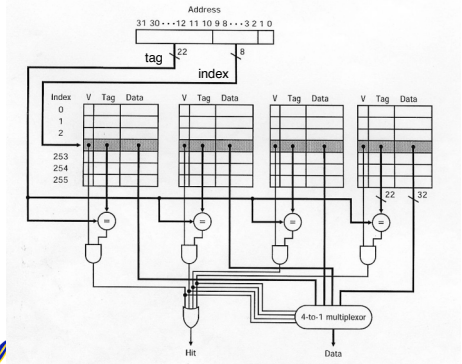
N-Way Set Associative Cache (2/3)

- Basic Idea
  - cache is direct-mapped w/respect to sets
  - each set is fully associative
  - basically N direct-mapped caches working in parallel: each has its own valid bit and data
- Given memory address:
  - Find correct set using Index value.
  - Compare Tag with all Tag values in the determined set.
  - If a match occurs, hit!, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the block.

N-Way Set Associative Cache (3/3)

- What's so great about this?
  - even a 2-way set assoc cache avoids a lot of conflict misses
  - hardware cost isn't that bad: only need N comparators
- In fact, for a cache with M blocks,
  - it's Direct-Mapped if it's 1-way set assoc
  - it's Fully Assoc if it's M-way set assoc
  - so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit



Peer Instructions

1. In the last 10 years, the gap between the access time of DRAMs & the cycle time of processors has decreased. (I.e., is closing)
2. A 2-way set-associative cache can be outperformed by a direct-mapped cache.
3. Larger block size ⇒ lower miss rate

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

Administrivia

- Proj4 due Sunday (will be posted tonight)
- Proj3 Face-to-Face grading starts today
  - Make an appointment (and show up)
- Final Review Session in works
- Course Survey during last lecture
  - 2 points extra added for taking survey (still anonymous)
  - I don't see results until long after grades due

Block Replacement Policy

- Direct-Mapped Cache: index completely specifies position which position a block can go in on a miss
- N-Way Set Assoc: index specifies a set, but block can occupy any position within the set on a miss
- Fully Associative: block can be written into any position
- Question: if we have the choice, where should we write an incoming block?
  - If there are any locations with valid bit off (empty), then usually write the new block into the first one.
  - If all possible locations already have a valid block, we must pick a replacement policy: rule by which we determine which block gets "cached out" on a miss.

Block Replacement Policy: LRU

- LRU (Least Recently Used)
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ⇒ recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Block Replacement Example

- We have a 2-way set associative cache with a four word *total* capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):

  0, 2, 0, 1, 4, 0, 2, 3, 5, 4

  How many hits and how many misses will there be for the LRU block replacement policy?

Block Replacement Example: LRU

- Addresses 0, 2, 0, 1, 4, 0, ...

0: miss, bring into set 0 (loc 0)

2: miss, bring into set 0 (loc 1)

0: hit

1: miss, bring into set 1 (loc 0)

4: miss, bring into set 0 (loc 1, replace 2)

0: hit

	loc 0	loc 1
set 0	0 lru	
set 1		
set 0	lru 0	2
set 1		
set 0	0	lru 2
set 1		
set 0	0	lru 2
set 1	1 lru	
set 0	lru 0	4
set 1	1	lru
set 0	0	lru 4
set 1	1 lru	

Big Idea

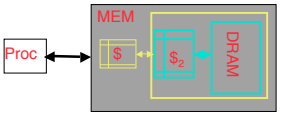
- How to choose between associativity, block size, replacement & write policy?
- Design against a performance model
  - Minimize: *Average Memory Access Time*

    = Hit Time
    + Miss Penalty x Miss Rate
  - influenced by technology & program behavior
- Create the illusion of a memory that is large, cheap, and fast - on average

How can we improve miss penalty?

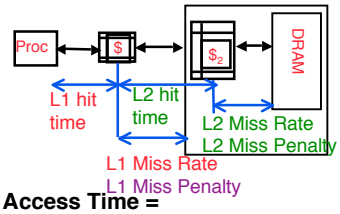
Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM ⇒ 200 processor clock cycles!



Solution: another cache between memory and the processor cache: Second Level (L2) Cache

Analyzing Multi-level cache hierarchy



Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty =
L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

Example

- Assume
  - Hit Time = 1 cycle
  - Miss rate = 5%
  - Miss penalty = 20 cycles
  - Calculate AMAT…
- Avg mem access time

  = 1 + 0.05 x 20
  = 1 + 1 cycles
  = 2 cycles

Ways to reduce miss rate

- Larger cache
  - limited by cost and technology
  - hit time of first level cache < cycle time (bigger caches are slower)
- More places in the cache to put each block of memory – associativity
  - fully-associative
    - any block any line
  - N-way set associated
    - N places for each block
    - direct map: N=1

Typical Scale

- L1
  - size: tens of KB
  - hit time: complete in one clock cycle
  - miss rates: 1-5%
- L2:
  - size: hundreds of KB
  - hit time: few clock cycles
  - miss rates: 10-20%
- L2 miss rate is fraction of L1 misses that also miss in L2
  - why so high?

Example: with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 200 cycles
- L1 miss penalty = 5 + 0.15 * 200 = 35
- Avg mem access time = 1 + 0.05 x 35
  = 2.75 cycles

Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
- Avg mem access time = 1 + 0.05 x 200
  = 11 cycles
- 4x faster with L2 cache! (2.75 vs. 11)

Peer Instructions

1. All caches take advantage of spatial locality.
2. All caches take advantage of temporal locality.
3. On a read, the return value will depend on what is in the cache.

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

And in Conclusion…

- We've discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - Filesystem cache
  - Web page cache
  - Game databases / tablebases
  - Software memoization
  - Others?
- Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.
- Cache design choices:
  - Write through v. write back
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - 3rd level cache?
- Use performance model to pick between choices, depending on programs, technology, budget, ...