CS184b: Computer Architecture [Single Threaded Architecture: abstractions, quantification, and optimizations]

Day12: February 15, 2000

Cache Introduction

Today

- Issue
- Structure
- Idea
- Cache Basics

Memory and Processors

- Memory used to compactly store
  - state of computation
  - description of computation (instructions)
- Memory access latency impacts performance
  - timing on load, store
  - timing on instruction fetch

Issues

- Need big memories:
  - hold large programs (many instructions)
  - hold large amounts of state
- Big memories are slow
- Memory takes up areas
  - want dense memories
  - densest memories not fast
    - fast memories not dense
- Memory capacity needed not fit on die
  - inter-die communication is slow

Problem

- Desire to contain problem
  - implies large memory
- Large memory
  - implies slow memory access
- Programs need frequent memory access
  - e.g. 20% load operations
  - fetch required for every instruction
- Memory is the performance bottleneck?
  - Programs run slow?

Opportunity

- Architecture mantra:
  - exploit structure in typical problems
- What structure exists?

Memory Locality

- What percentage of accesses to unique addresses
  - addresses distinct from the last N unique addresses

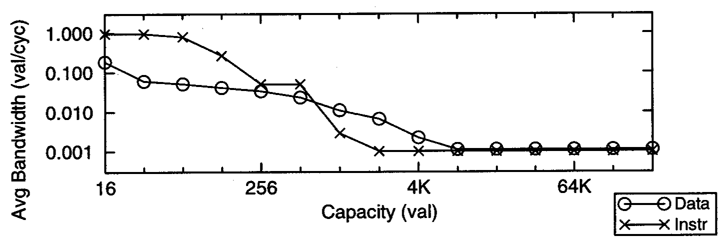


Figure 3-1 Bandwidth spectrums for *cjpeg*.

[Huang+Shen, Intrinsic BW, ASPLOS 7]

[from CS184a]

Hierarchy/Structure Summary

- "Memory Hierarchy" arises from area/bandwidth tradeoffs
  - Smaller/cheaper to store words/blocks
    - (saves routing and control)
  - Smaller/cheaper to handle long retiming in larger arrays (reduce interconnect)
  - High bandwidth out of registers/shallow memories

	DRAM	SRAM	RF bit	FF/RF	RF×1	XC	In FF	net FF	FF/LUT
λ^2	100	1200	2K	5K	40K	40K	75K	200K	800K
bw/cap.	$1/10^7$	$1/10^5$–10^3		1/100	1/100	1/16	1/4	1/1	1/1

Opportunity

- Small memories are fast
- Access to memory is not random
  - temporal locality
  - short and long retiming distances
- Put commonly/frequently used data (instructions) in small memory

Memory System Idea

- Don’t build single, flat memory
- Build a hierarchy of speeds/sizes/densities
  - commonly accessed data in fast/small memory
  - infrequently used data in large/dense/cheap memory
- Goal
  - achieve speed of small memory
  - with density of large memory

Hierarchy Management

- Two approaches:
  - explicit data movement
    - register file
    - overlays
  - transparent/automatic movement
    - invisible to model

Opportunity: Model

- Model is simple:
  - read data and operate upon
  - timing not visible
- Can vary timing
  - common case fast (in small memory)
  - all cases correct
    - can answered from larger/slower memory

Cache Basics

- Small memory (cache) holds commonly used data
- Read goes to cache first
- If cache holds data
  - return value
- Else
  - get value from bulk (slow) memory
- Stall execution to hide latency
  - full pipeline, scoreboarding

Cache Questions

- How manage contents?
  - decide what goes (is kept) in cache?
- How know what we have in cache?
- How make sure consistent ?
  - between cache and bulk memory

Cache contents

- **Ideal:** cache should hold the N items that maximize the fraction of memory references which are satisfied in the cache
- **Problem:**
  - don't know future
  - don't know what values will be needed in the future
    - partially limitation of model
    - partially data dependent
    - halting problem
      - (can't say if will execute piece of code)

Cache Contents

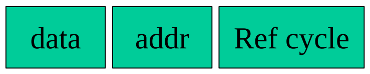
- Look for heuristics which keep most likely set of data in cache
- **Structure:** temporal locality
  - high probability that recent data will be accessed again
- **Heuristic goal:**
  - keep the last N references in cache

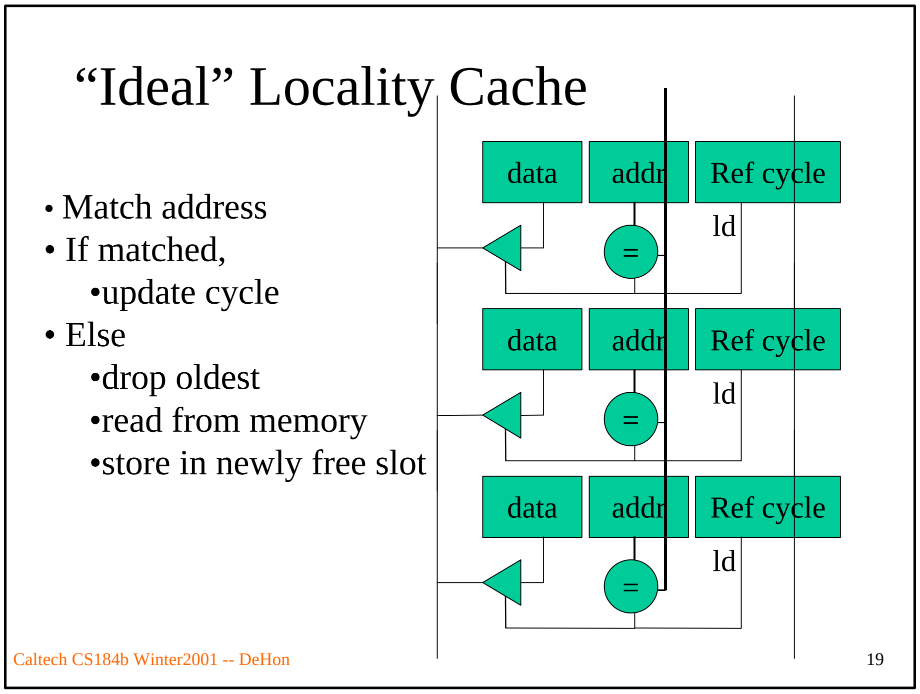
Temporal Locality Heuristic

- Move data into cache on access (load, store)
- Remove "old" data from cache to make space

"Ideal" Locality Cache

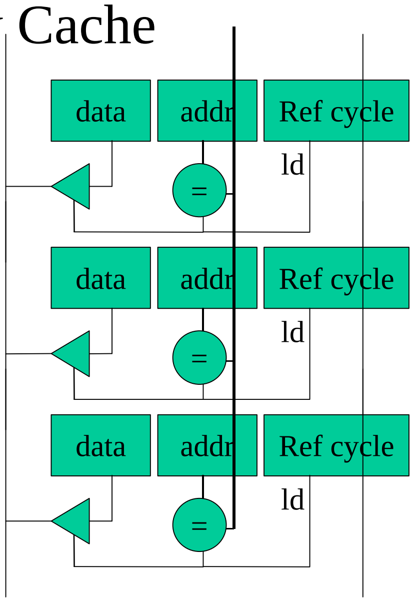
- Stores N most recent things
  - store any N things
  - know which N things accessed
  - know when last used

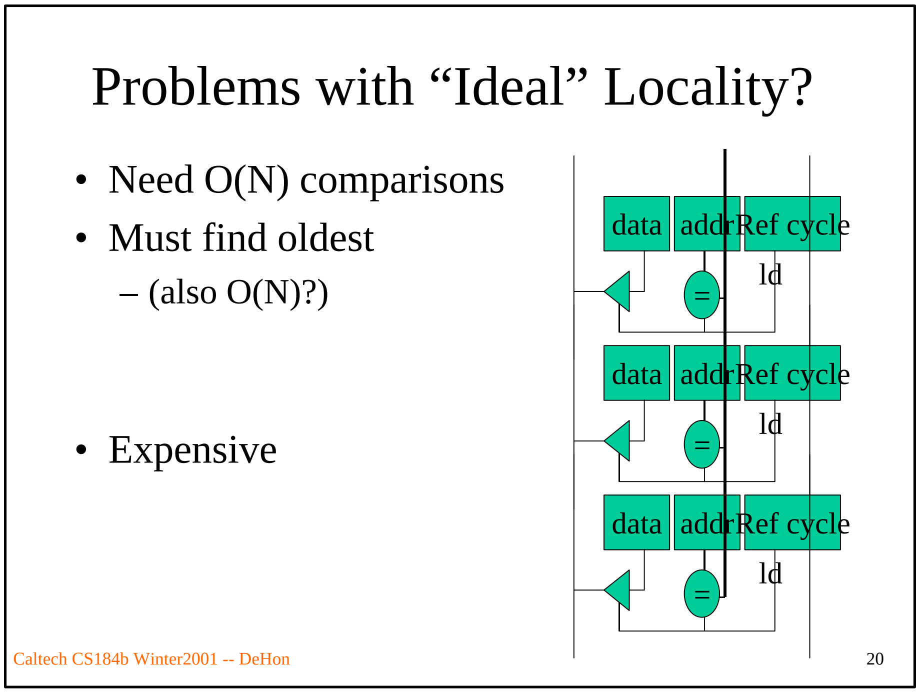




“Ideal” Locality Cache

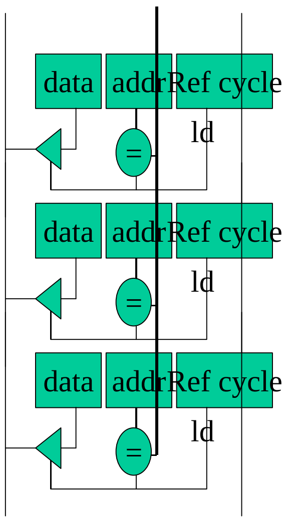
- Match address
- If matched,
    - update cycle
- Else
    - drop oldest
    - read from memory
    - store in newly free slot





Problems with “Ideal” Locality?

- Need O(N) comparisons
- Must find oldest
    - (also O(N)?)
- Expensive

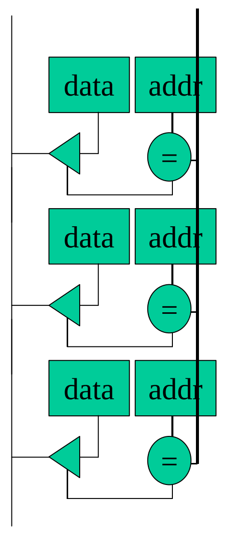


Relaxing “Ideal”

- Keeping usage (and comparing) expensive
- Relax:
  - Keep only a few bits on age
  - Don’t bother
    - pick victim randomly
    - things have expected lifetime in cache
    - old things more likely than new things
    - if evict wrong thing, will replace
    - very simple/cheap to implement

Fully Associative Memory

- Store both
  - address
  - data
- Can store any N addresses
- approaches ideal of “best” N things

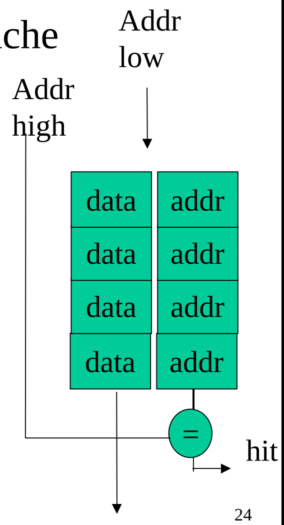


Relaxing “Ideal”

- Comparison for every address is expensive
- Reduce comparisons
  - deterministically map address to a small portion of memory
  - Only compare addresses against that portion

Direct Mapped

- Extreme is a “direct mapped” cache
- Memory slot is f(addr)
  - usually a few low bits of address
- Go directly to address
  - check if data want is there



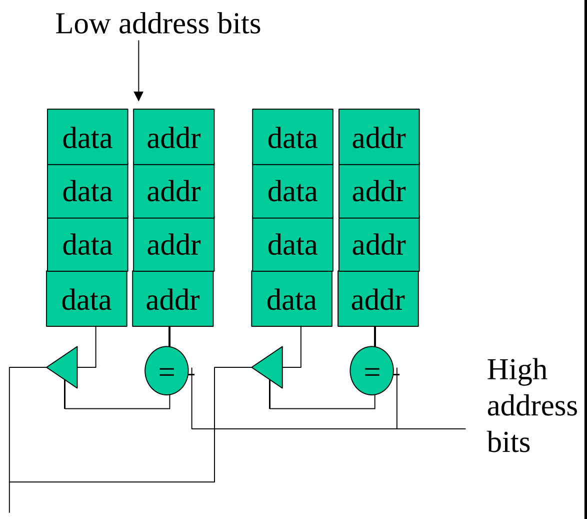
Direct Mapped Cache

- Benefit
  - simple
  - fast
- Cost
  - multiple addresses will need same slot
  - conflicts mean don’t really have most recent N things
  - can have conflict between commonly used items

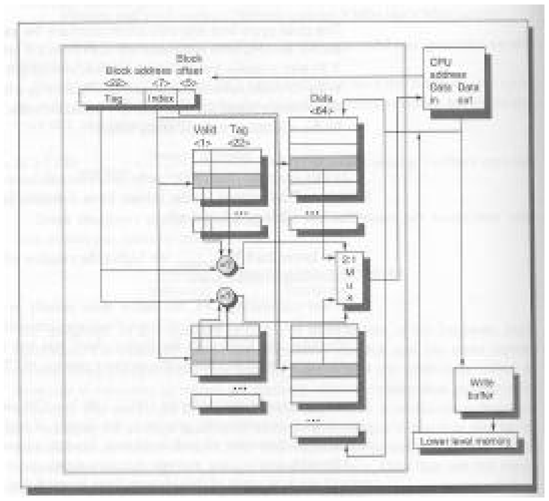
Set-Associative Cache

- Between extremes set-associative
- Think of M direct mapped caches
- One comparison for each cache
- Lookup in all M caches
- Compare and see if any have target data
- Can have M things which map to same address

Two-Way Set Associative



Two-way Set Associative



[Hennessy and Patterson 5.8]

Set Associative

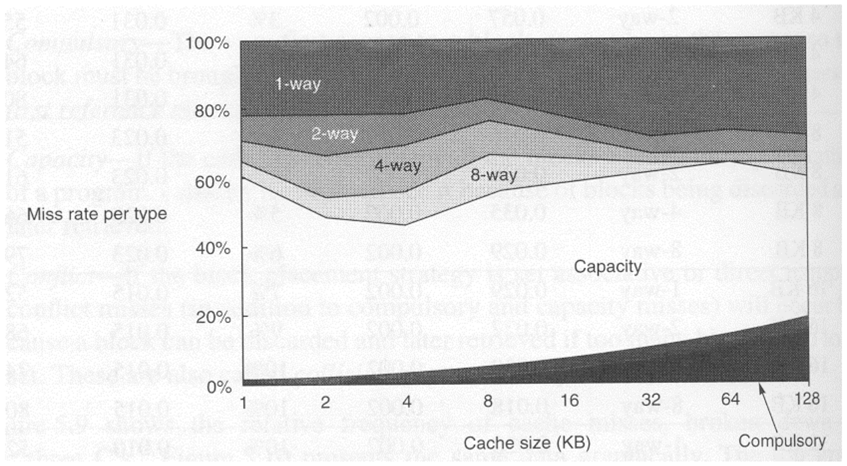
- More expensive that direct mapped
- Can decide expense
- Slower than direct mapped
  - – have to mux in correct answer

- Can better approximate holding N most recently/frequently used things

Classify Misses

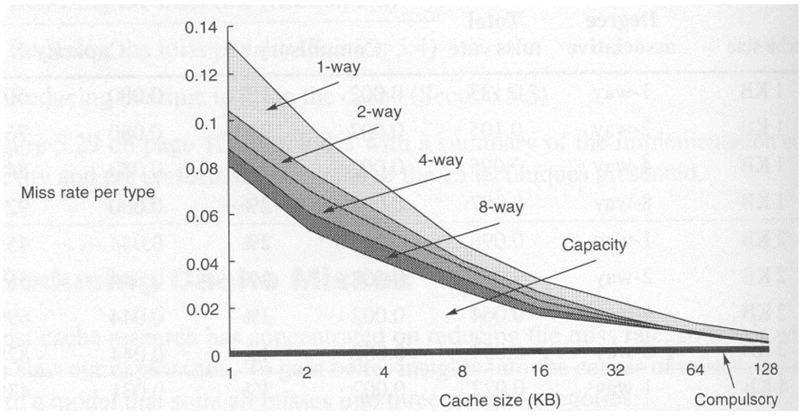
- Compulsory
  - – first refernce
  - – (any cache would have)
- Capacity
  - – misses due to size
  - – (fully associative would have)
- Conflict
  - – miss because of limit places to put

Set Associativity



[Hennessy and Patterson 5.10]

Absolute Miss Rates



[Hennessy and Patterson 5.10]

Policy on Writes

- Keep memory consistent at all times?
  - Or cache+memory holds values?
- Write through:
  - all writes go to memory and cache
- Write back:
  - writes go to cache
  - update memory only on eviction

Write Policy

- Write through
  - easy to implement
  - eviction trivial
    - (just overwrite)
  - every write is slow (main memory time)
- Write back
  - fast (writes to cache)
  - eviction slow/complicate

Cache Equation...

- Assume hits satisfied in 1 cycle
- CPI = Base CPI + Refs/Instr (Miss Rate)(Miss Latency)

Cache Numbers

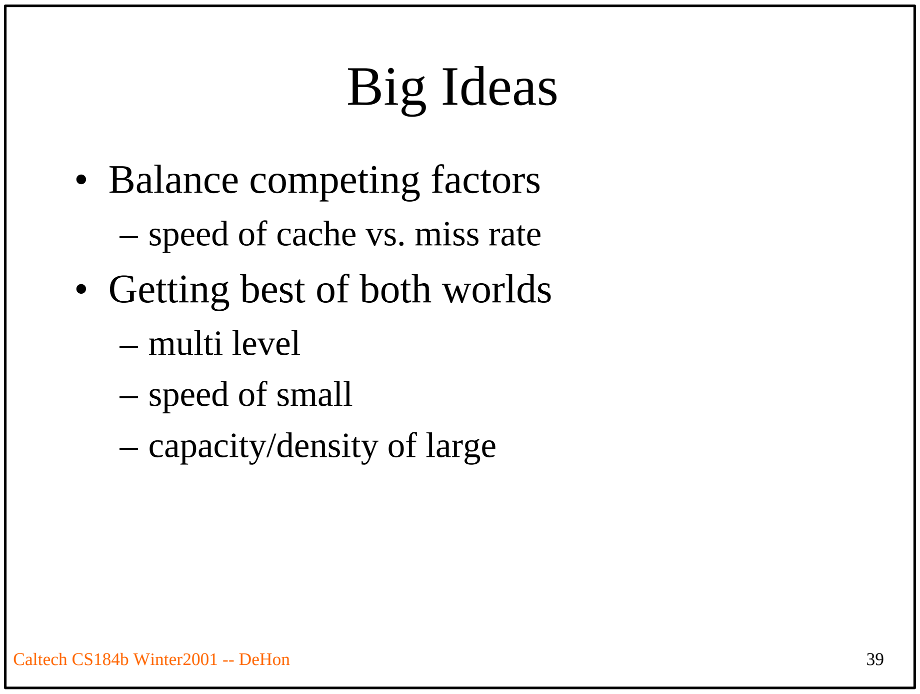
- CPI = Base CPI + Ref/Instr (Miss Rate)(Miss Latency)
- From ch2/experience
  - load-stores make up ~30% of operations
- Miss rates
  - …1-10%
- Main memory latencies
  - 50ns
- Cycle times
  - 1ns … shrinking

Cache Numbers

- No Cache
  - CPI=Base+0.3*50=Base+15
- Cache at CPU Cycle (10% miss)
  - CPI=Base+0.3*0.1*50=Base +1.5
- Cache at CPU Cycle (1% miss)
  - CPI=Base+0.3*0.01*50=Base +0.15

Big Ideas

- Structure
  - temporal locality
- Model
  - optimization preserving model
  - simple model
  - sophisticated implementation
  - details hidden



Big Ideas

- Balance competing factors
  - speed of cache vs. miss rate
- Getting best of both worlds
  - multi level
  - speed of small
  - capacity/density of large