Memory Hierarchy

Dave Eckhardt
de0u@andrew.cmu.edu

Outline

- Lecture versus book
  - Some of Chapter 2
  - Some of Chapter 10
- Memory hierarchy
  - A principle (not just a collection of hacks)

Am I in the wrong class?

- “Memory hierarchy”: OS or Architecture?
  - Yes
- Why cover it here?
  - OS manages several layers
    - RAM cache(s)
    - Virtual memory
    - File system buffer cache
  - Learn core concept, apply as needed

Memory Desiderata

- Capacious
- Fast
- Cheap
- Compact
- Cold
  - Pentium-4 2 Ghz: ***75 watts!?***
- Non-volatile (can remember w/o electricity)

You can't have it all

- Pick one
  - ok, maybe two
- Bigger ⇒ slower (speed of light)
- Bigger ⇒ more defects
  - If constant per unit area
- Faster, denser ⇒ hotter
  - At least for FETs

Users want it all

- The ideal
  - Infinitely large, fast, cheap memory
  - Users want it (those pesky users!)
- They can't have it
  - Ok, so cheat!

Locality of reference

- Users don't really access 4 gigabytes uniformly
- 80/20 “rule”
  - 80% of the time is spent in 20% of the code
  - Great, only 20% of the memory needs to be fast!
- Deception strategy
  - Harness 2 (or more) kinds of memory together
  - Secretly move information among memory types

Cache

- Small, fast memory...
- Backed by a large, slow memory
- Indexed via the ***large memory's*** address space
- Containing the most popular parts
  - (at present)

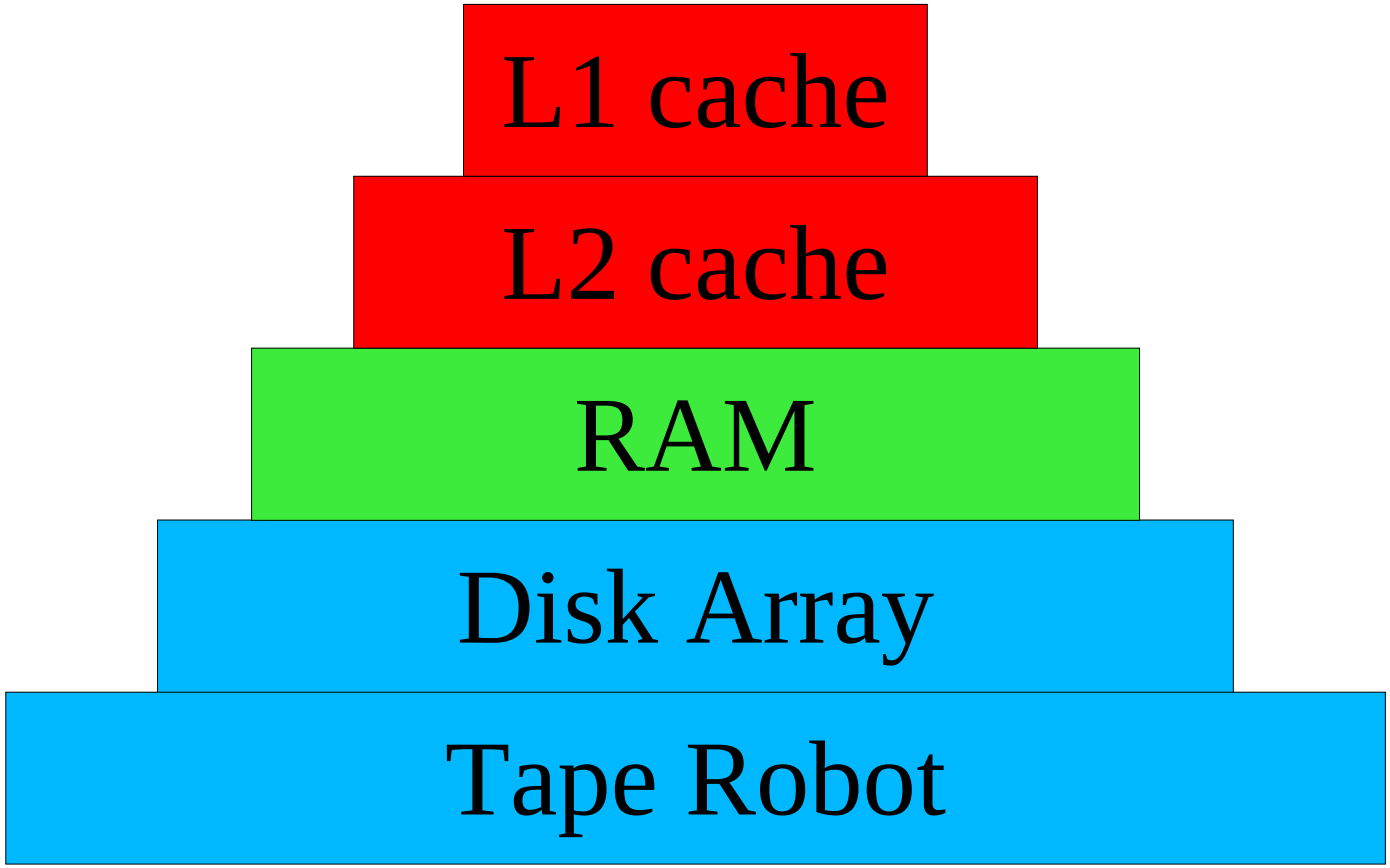
Cache Example – Satellite Images

- SRAM cache holds popular pixels
- DRAM holds popular image areas
- Disk holds popular satellite images
- Tape holds one orbit's worth of images

Great Idea...

- Clean general-purpose implementation?
  - #include <cache.h>
- No: tradeoffs different at each level
  - Size ratio: data address / data size
  - Speed ratio
  - Access time = f(address)
- But ***the idea*** is general-purpose

Pyramid of Deception



Key Questions

- Line size
- Placement/search
- Miss policy
- Eviction
- Write policy

Today's Examples

- L1 CPU cache
  - Smallest, fastest
  - Maybe on the same die as the CPU
  - Maybe 2nd chip of multi-chip module
  - Probably SRAM
  - 2003: “around a megabyte”
    - ~ 0.1% of RAM
  - As far as CPU is concerned, this is ***the*** memory
    - Indexed via RAM addresses (0..4 GB)

Today's Examples

- Disk block cache
  - Holds disk sectors in RAM
  - Entirely defined by software
  - ~ 0.1% to maybe 1% of disk (varies widely)
  - Indexed via (device, block number)

“Line size” = item size

- Many caches handle fixed-size objects
  - Simpler search
  - Predictable operation times
- L1 cache line size
  - 4 32-bit words (486, IIRC)
- Disk cache line size
  - Maybe disk sector (512 bytes)
  - Maybe “file system block” (~16 sectors)

Picking a Line Size

- What should it be?
  - Theory: see “locality of reference”
    - (“typical” reference pattern)

Picking a Line Size

- Too big
  - Waste throughput
    - Fetch a megabyte, use 1 byte
  - Waste cache space ⇒ reduce “hit rate”
    - String move: *q++ = *p++
    - Better have at least two cache lines!
- Too small
  - Waste latency
    - Frequently must fetch another line

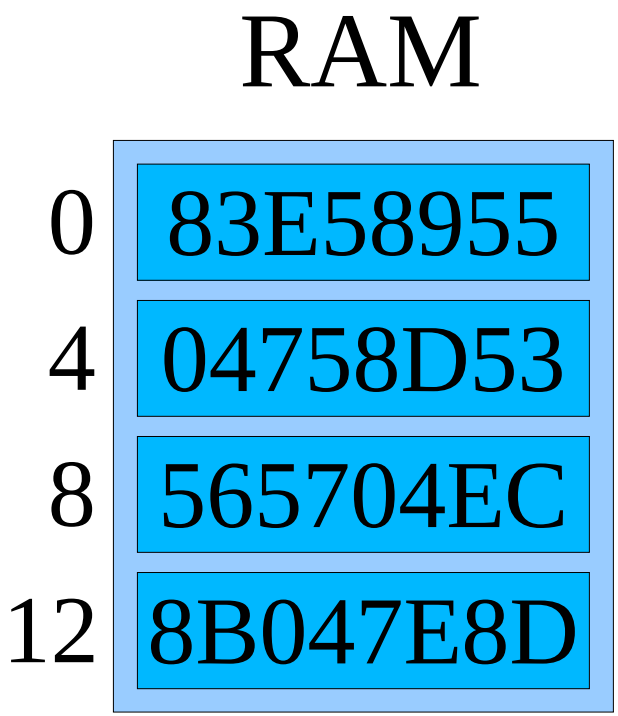
Content-Addressable Memory

- RAM
  - store(address, value)
  - fetch(address) ⇒ value
- CAM
  - store(address, value)
  - fetch(value) ⇒ address
- “It's always the last place you look”
  - Not with a CAM!

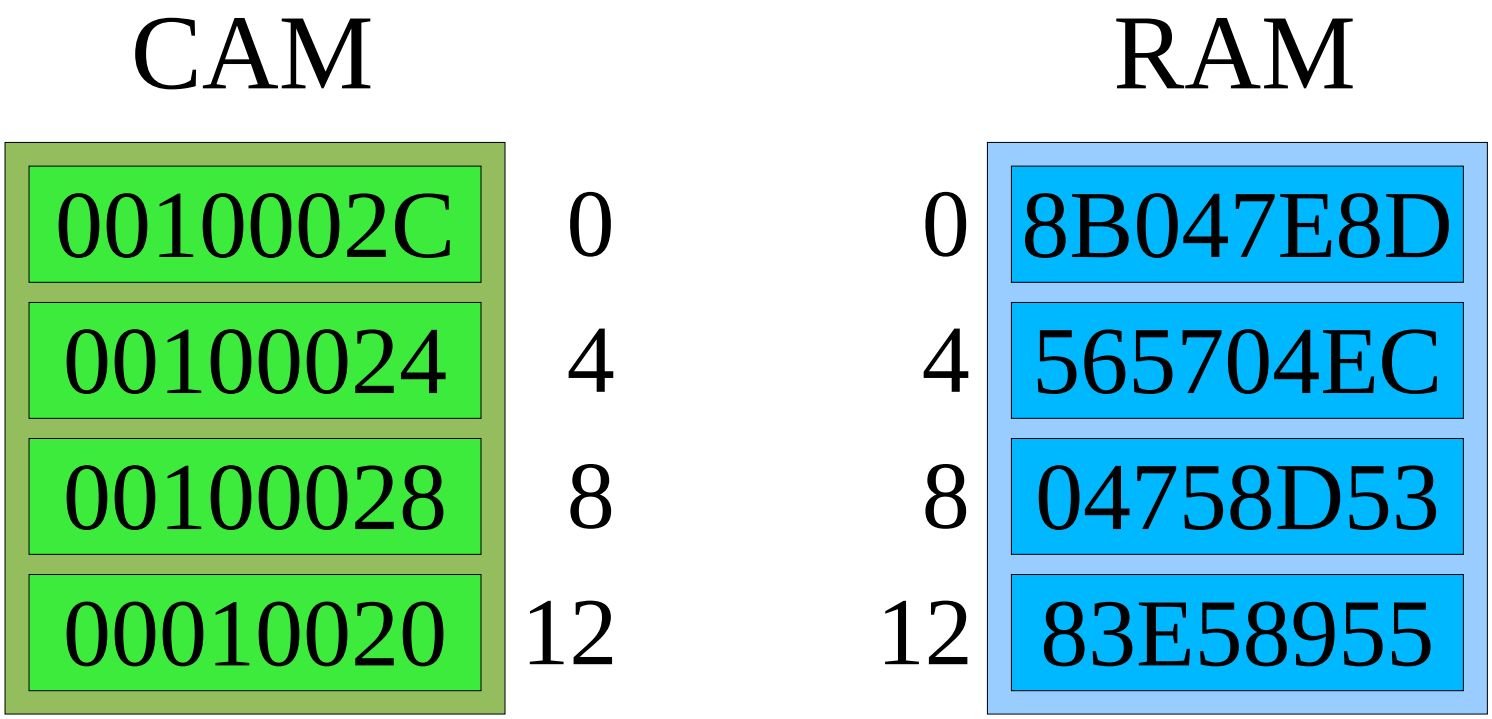
Memory Contents

Address	*Contents*
10020	**83E58955**
10024	**565704EC**
10028	**04758D53**
1002C	**8B047E8D**

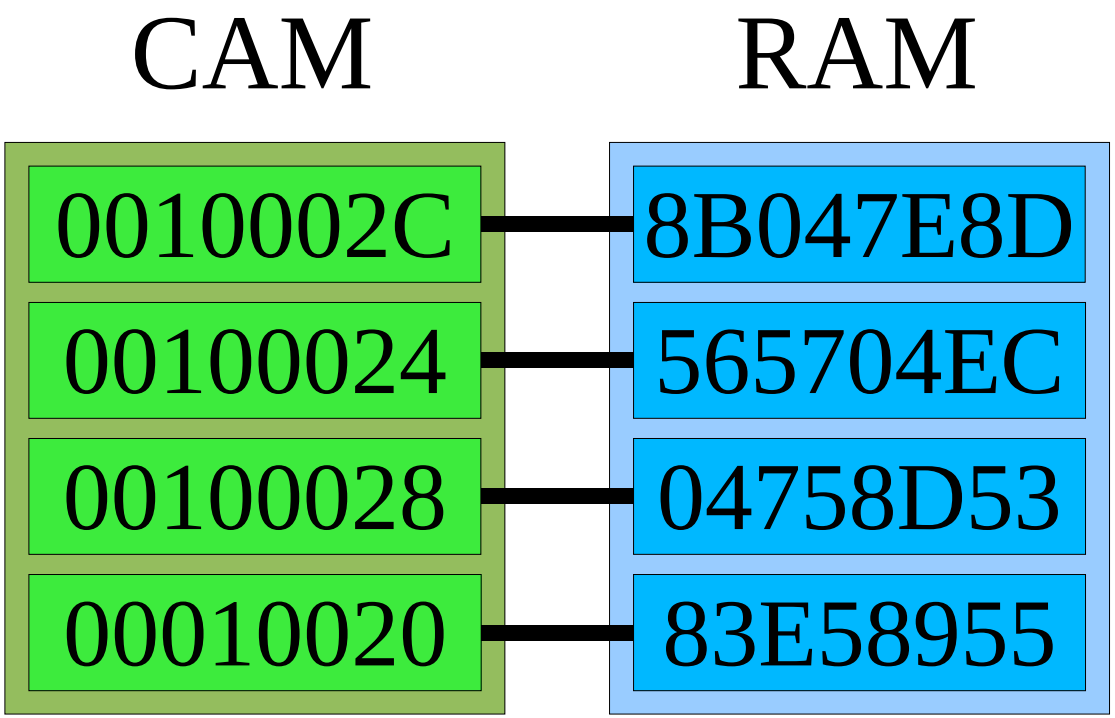
RAM + CAM = Cache



RAM + CAM = Cache



RAM + CAM = Cache



Content-Addressable Memory

- CAMS are cool!
- But fast CAMs are small (speed of light, etc.)
- If this were an architecture class...
  - We would have 5 slides on ***associativity***
  - Not today: only 2

Placement/search

- Placement = "Where can we put ____?"
  - “Direct mapped” - each item has ***one place***
    - Think: hash function
  - "Fully associative" - each item can be any place
    - Think: CAM
- Direct Mapped
  - Placement & search are trivial
  - False collisions are common
  - String move: *q++ = *p++;
  - Each iteration could be ***two*** cache misses!

Placement/search

- Fully Associative
  - No false collisions
  - Cache size/speed limited by CAM size
- Choosing associativity
  - Trace-driven simulation
  - Hardware constraints

Thinking the CAM way

- Are we having P2P yet?
  - I want the latest ***freely available*** Janis Ian song...
    - www.janisian.com/article-internet_debacle.html
  - ...who on the Internet has a copy for me to download?
- I know what I want, but not where it is...
  - ...Internet as a CAM

Sample choices

- L1 cache
  - Often direct mapped
  - Sometimes 2-way associative
  - Depends on phase of transistor
- Disk block cache
  - Fully associative
  - Open hash table = large variable-time CAM
  - Fine since "CAM" lookup time << disk seek time

Miss Policy

- Miss policy: {Read,Write} X {Allocate,Around}
  - Allocate: miss ⇒ allocate a slot
  - Around: miss ⇒ don't change cache state
- Example: Read-allocate, write-around
  - Read miss
    - Allocate a slot in cache
    - Fetch data from memory
  - Write miss
    - Store straight to memory

Miss Policy – L1 cache

- Mostly read-allocate, write-allocate
- But not for "uncacheable" memory
  - ...such as Ethernet card ring buffers
- “Memory system” provides “cacheable” bit
- Some CPUs have "write block" instructions for gc

Miss Policy – Disk-block cache

- Mostly read-allocate, write-allocate
- What about reading (writing) a ***huge*** file?
  - Would toast cache for no reason
  - See (e.g.) madvise()

Eviction

- “The steady state of disks is 'full'”.
- Each placement requires an eviction
  - Easy for direct-mapped caches
  - Otherwise, policy is necessary
- Common policies
  - Optimal, LRU
  - LRU may be great, can be awful
  - 4-slot associative cache: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...

Eviction

- Random
  - Pick a random item to evict
  - Randomness protects against pathological cases
  - When could it be ***good***?
- L1 cache
  - LRU is easy for 2-way associative!
- Disk block cache
  - Frequently LRU, frequently modified
  - “Prefer metadata”, other hacks

Write policy

- Write-through
  - Store new value in cache
  - Also store it through to next level
  - Simple
- Write-back
  - Store new value in cache
  - Store it to next level ***only on eviction***
  - Requires “dirty bit”
  - May save substantial work

Write policy

- L1 cache
  - It depends
  - May be write-through if next level is L2 cache
- Disk block cache
  - Write-back
  - Popular mutations
    - Pre-emptive write-back if disk idle
    - Bound write-back delay (crashes happen)
    - Maybe don't write ***everything*** back (“softatime”)

Translation Caches

- Address mapping
  - CPU presents virtual address (%CS:%EIP)
  - Fetch segment descriptor from L1 cache (or not)
  - Fetch page directory from L1 cache (or not)
  - Fetch page table entry from L1 cache (or not)
  - Fetch the actual word from L1 cache (or not)

“Translation lookaside buffer” (TLB)

- Observe result of first 3 fetches
  - Segmentation, virtual ⇒ physical mapping
- Cache the ***mapping***
  - Key = virtual address
  - Value = physical address
- Q: Write policy?

Challenges – Write-back failure

- Power failure?
  - Battery-backed RAM!
- Crash?
  - Maybe the old disk cache is ok after reboot?

Challenges - Coherence

- Multiprocessor: 4 L1 caches share L2 cache
  - What if L1 does write-back?
- TLB: v $\Rightarrow$ p all wrong after context switch
- What about non-participants?
  - I/O device does DMA
- Solutions
  - Snooping
  - Invalidation messages (e.g., set_cr3())

Summary

- Memory hierarchy has many layers
  - Size: kilobytes through terabytes
  - Access time: nanoseconds through minutes
- Common questions, solutions
  - Each instance is a little different
  - But there are lots of cookbook solutions