18-447 Lecture 16: Memory Hierarchy

James C. Hoe
Dept of ECE, CMU
March 23, 2009

Announcements: Project 3 due this week
Midterm 2 next Monday

Handouts: Handout #12 HW 3 solutions (on Blackboard)

Format of the Quiz

- Coverage
  - lectures (L1~L14, emphasis on L8~L14), HWs, projects, assigned readings (textbooks and papers)
- Types of questions
  - freebies: can you remember the materials
  - probing: did you understand the materials
  - applied: can you apply the materials in original thoughts
- 100 minutes, 100 points
  - if a question is worth 5 points, don't spend 20 minutes
  - skip questions you can't do and come back to them later
  - closed-book, one 2-sided 8½x11 crib sheet
  - no calculators

*** Use pencil or black/blue ink only

*** Be on time, 2:30 sharp!!!

Wishful Memory

- Lecture 6: a program sees a contiguous 4GB memory
- Lecture 7: access anywhere in memory in 1 proc. cycle
- We are in good company

> 4.1. Ideally one would desire an indefinitely large memory c
> pacity such that any particular aggregate of 40 binary digits,
> *word* (cf. 2.3), would be immediately available—i.e. in a tin

---- Burks, Goldstein, von Neumann,1946

The Reality

- Can't afford and don't need as much memory as the size of the user address space (think about 64-bit ISAs)
- Most machines are multi-tasked between several programs
- You can't find memory technology that is affordable in GBytes and also cycle in GHz
- The "magic" memory abstraction are nevertheless very "useful" approximation of reality due to
  - memory hierarchy: large and fast
  - virtual memory: contiguous and private

The Law of Storage

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte

    Note these sample values scale with time

How to make memory Bigger, Faster, and Cheaper?

Principles behind the solution

Locality

- One's recent past is a very good predictor of his/her near future.
- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly
  - inverse is also true
- Spatial Locality: If you just did something, it is very likely you will do something similar/related
  - every time I find you in this room, you are probably sitting in the same seat
  - you are probably sitting near the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  *** typical programs are composed of "loops"
- Temporal: A program tends to reference the same memory location many times and all within a small window of time
- Spatial: A program tends to reference a cluster of memory locations at a time (most notable examples 1. instruction memory references and 2. array/data structure references)
- Corollary: a program may reference a large number of different memory locations over its live time but not all at the same time

Memoization

- If something is expensive to compute, you might want to remember the answer for a while, just in case you will need the same answer again
- Memoization needs locality to work effectively
- Without locality
  - storing a large number of different answers (many of which never reused)
  - storing a very large number of answers and later locating an answer on demand can be more expensive than recomputing it
- With locality
  - store only small number of the most frequently used answers avoids most recomputations
  - the same answer gets reused many, many times!

Cost Amortization

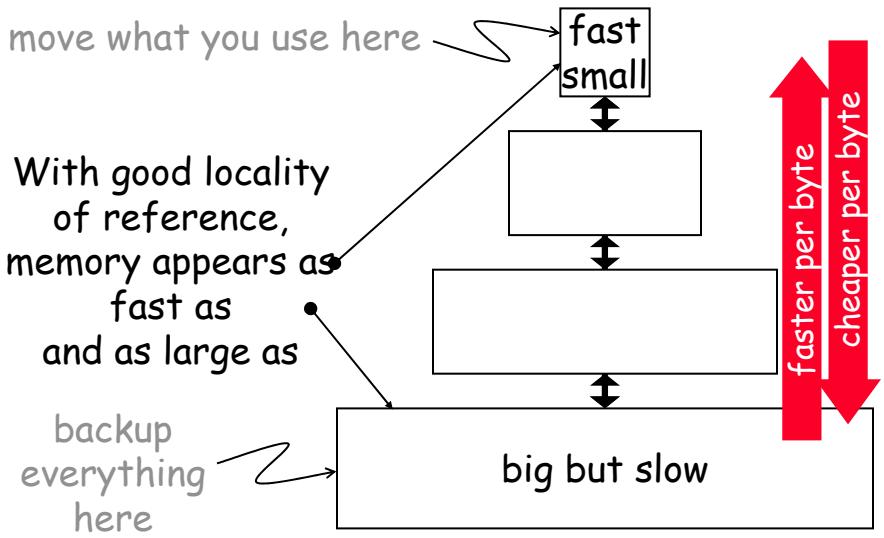
- overhead cost : one-time cost to set something up
- per-unit cost : cost for per unit of operation

- total cost = overhead + per-unit cost x N
- average cost = total cost / N
  = (overhead / N) + per-unit cost

- It is often okay to have a high overhead cost if the cost can be distributed over a large number of units
  $\Rightarrow$ lower the average cost

Putting the principles to work

Memory Hierarchy

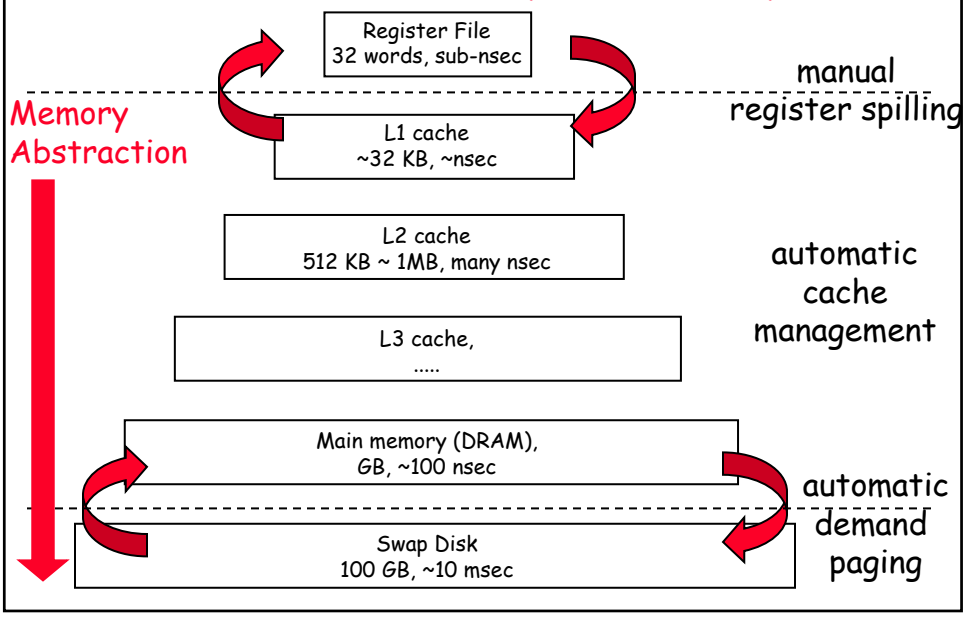


Managing Memory Hierarchy

- You could manage data movement across hierarchies manually
  - already discussed in von Neumann paper (vacuum tubes vs Selectron)
  - "core" vs "drum" memory in the 50's
  - too painful for programmers on substantial programs
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)
- Automatic management
  - simple heuristic: keep most recently used items in fast mem
  - dates back to ATLAS,1962
  - today in every modern desktop and server system
  - the average programmer doesn't need to know about it

    You don't need to know how big the cache is to write a "correct" program! (You may if you want a "fast" program.)

Modern Memory Hierarchy



Hierarchical Performance Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i
- The perceived access time T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate h_i) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + \underbrace{m_i \cdot T_{i+1}}$$ ← think of this as the "miss penalty"

keep in mind, h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Compromises

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable but not necessary
- Keep m_i low
  - increase capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)
- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Hierarchy Design Considerations

- DRAM
  - optimized for capacity/dollar
  - T_{DRAM} is essentially fixed for a given technology generation
- SRAM
  - optimized first for capacity/latency (second for capacity/dollar)
  - different compromise between capacity and latency possible

$$t_i = O(\sqrt{C_i})$$

- Hierarchies bridge the difference between CPU speed and DRAM speed
  - $T_{pclk} \approx T_{DRAM} \Rightarrow$ no hierarchy needed
  - $T_{pclk} << T_{DRAM} \Rightarrow$ one or more levels of SRAM hierarchies to minimize T_1 while staying within cost

Intel P4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- Main memory
  - t_3 = ~ 50ns or 180 cyc
- Notice
  - best case latency is not 1 anymore
  - worst case access latency are into 300+ cyc, depending exactly what happens

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

Why not?

Aside: Why is DRAM slow?

- DRAM fabrication at the forefront of VLSI technology nodes, but scales with Moore's law in capacity and cost, not speed
- Between 1980 ~ 2004 DRAM
  - 64K bit → 1024M bit (exponential ~55% annual)
  - 250ns → 50ns (linear)

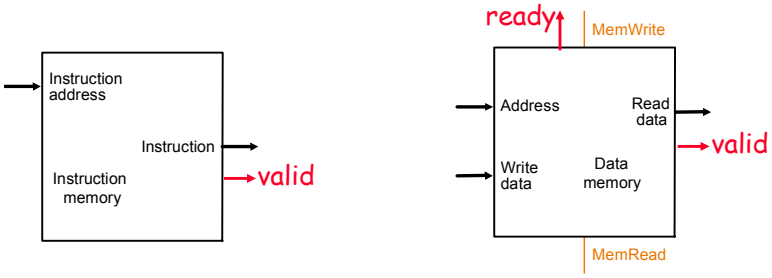
  But, remember, this is a very deliberate choice.
  We can "engineer" faster DRAM if we needed to
- Memory capacity needs to grow linearly with CPU speed to keep a balanced system – Amdahl
- DRAM/processor speed difference reconciled through memory hierarchies (L1, L2, L3, ...)
  - L2 became common place in the 90s
  - L3 becoming common place in the 00s

Cache Basics

Cache

- Generically, any structure that "memoizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache
- Most commonly, an automatically-managed memory hierarchy based on SRAM
  - memoize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Cache Interface

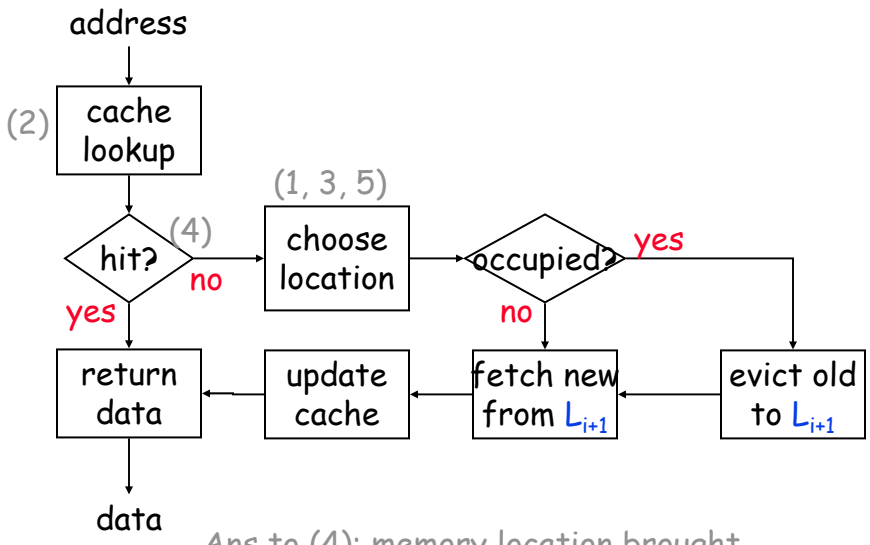


- Like the magic memory we assume in Lecture 8
  - present address, command, etc
  - most of the time result or side-effect valid after a short/fixed latency (1 cyc?)
- Except, cache may not be valid/ready on every cycle
  - the cache eventually must become valid/ready
  - what happens to the pipeline until then?

The problem

- Potentially $M=2^m$ bytes of memory, how to keep the most frequently used ones in C bytes of fast storage where $C << M$
- Basic issues (intertwined)
  - (1) where to "cache" a memory location?
  - (2) how to find a cached memory location?
  - (3) granularity of management: large, small, uniform?
  - (4) when to bring a memory location into cache?
  - (5) which cached memory location to evict to free-up space?
- Optimizations

Basic Operation

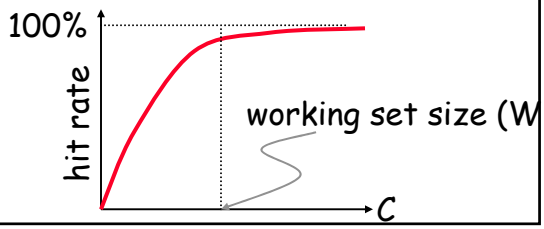


Ans to (4): memory location brought into cache "on-demand". What about prefetch?

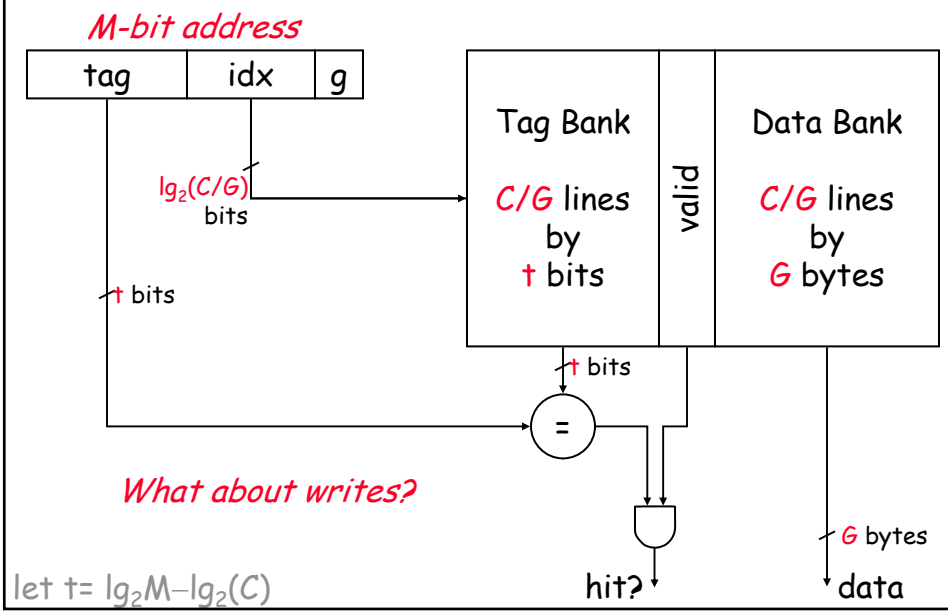
Basic Cache Parameters

- Let $M = 2^m$ be the size of the address space in bytes
  sample values: 2^{32}, 2^{64}
- Let $G=2^g$ be the cache access granularity in bytes
  sample values: 4, 8

- Let C be the "capacity" of the cache in bytes
  sample values: 16 KBytes (L1), 1 MByte (L2)



Direct-Mapped Cache (v1)



What about writes?

let $t = \lg_2 M - \lg_2(C)$

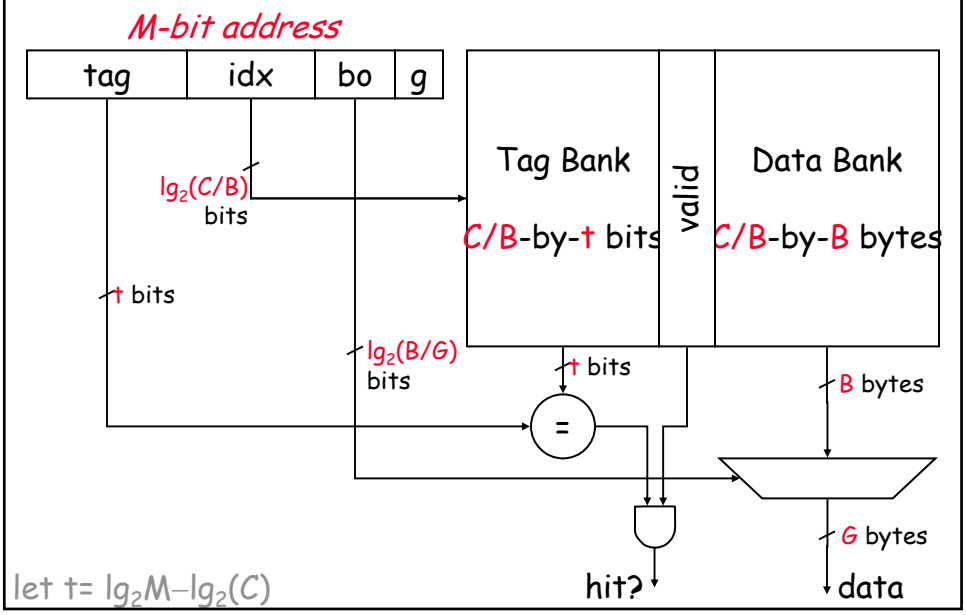
Storage Overhead

- For each cache block of G bytes, must also store additional "$t+1$" bits where $t=\lg_2 M-\lg_2(C)$
  - if $M=2^{32}$, $G=4$, $C=16K=2^{14}$
  - $\Rightarrow$ $t=18$ bits for each 4-byte block
    60% storage overhead
    16KB cache really needs 25.5KB of SRAM
- Solution: let multiple G-byte words share a common tag
  - each B-byte block holds B/G words
  - if $M=2^{32}$, $B=16$, $G=4$, $C=16K$
  - $\Rightarrow$ $t=18$ bits for each 16-byte block
    15% storage overhead
    16KB cache needs 18.4KB of SRAM

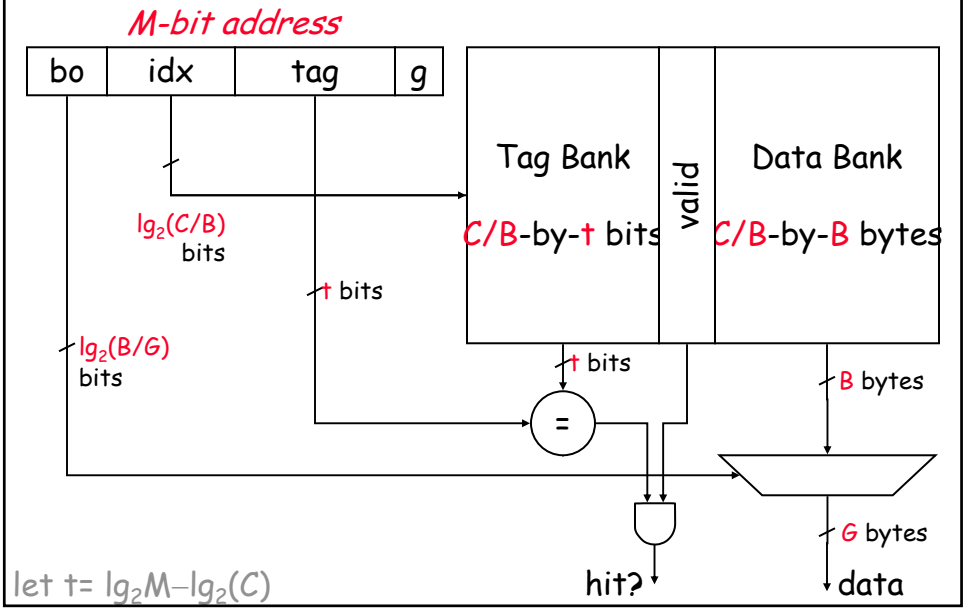
15% of 16KB is small, 15% of 1MB is 152KB

$\Rightarrow$ larger block size for lower/larger hierarchies

Direct-Mapped Cache (final)



Test yourself: What is wrong with this?



let t= $lg_2M - lg_2(C)$