18-447 Lecture 14: Memory Hierarchy

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – understand memory system and memory hierarchy design in big pictures

• Notices
  – HW 4, due 3/21
  – Lab 3, due week 10

• Readings
  – P&H Ch5 for the next many lectures
Wishful Memory

• So far we imagined
  – a program owns contiguous 4GB private memory
  – a program can access anywhere in 1 proc. cycle

• We are in good company

4.1. Ideally one would desire an indefinitely large memory capacity such that any particular aggregate of 40 binary digits, \textit{word} (cf. 2.3), would be immediately available—i.e. in a tin

---- Burks, Goldstein, von Neumann, 1946
The Reality

• Can’t afford/don’t need as much memory as size of address space

  RV32I said 4GB addr “space” not 4GB memory

• Can’t find memory technology that is affordable in GByte and also cycle in GHz

• Most systems multi-task several programs

• But, “magic” memory is nevertheless a useful approximation of reality due to
  – memory hierarchy: appear large and fast
  – virtual memory: appear contiguous and private

cover this part first

cover this part later
Memory Hierarchy: The Principles at Work
The Law of Storage

• Bigger is slower
  – SRAM 512 Bytes @ sub-nsec
  – SRAM KByte~MByte @ nsec
  – DRAM GByte @ ~50 nsec
  – SSD TByte @ msec
  – Hard Disk TByte @ ~10 msec

• Faster is more expensive (dollars and chip area)
  – SRAM ~$10K per GByte
  – DRAM ~$10 per GByte
  – “Drives” ~$0.1 per GByte

How to make memory bigger, faster and cheaper?
Memory Locality

• “Typical” programs have strong locality in memory references—instruction and data we put them there ... loops, arrays, and structs ...

• Temporal: after accessing A, how many other distinct addresses before accessing A again

• Spatial: after accessing A, how many other distinct addresses before accessing a “near-by” B

• Corollary: a program with strong temporal and spatial locality must be accessing only a compact “working set” at a time
Memoization

- If something is costly to compute, save the result to be reused

- With strong reuse
  - storing just a small number of frequently used results can avoid most recomputations

- With poor reuse
  - storing a large number of different results that are rarely or never reused
  - locating the needed result from a large number of stored ones can itself become as expensive as computing
Cost Amortization

- **overhead**: one-time cost to set up
- **unit-cost**: cost for each unit of work

- total cost = overhead + unit-cost x N
- average cost = total cost / N

\[
\text{average cost} = \frac{\text{overhead}}{N} + \text{unit-cost}
\]

In memoization, high up-front cost to compute once is no problem if results reused many times
Putting the principles to work
Memory Hierarchy

- Keep what you use actively here with strong locality:
  - Effectively as fast as
  - And as large as

- Hold what isn’t being used

- Faster and smaller cheaper per byte
Managing Memory Hierarchy

• Copy data between levels explicitly and manually
  – vacuum tubes vs Selectron (von Neumann paper)
  – “core” vs “drum” memory in the 50’s
  – “scratchpad” SRAM used on modern embedded and DSP

  Register file is a level of storage hierarchy

• Single address space, automatic management
  – as early as ATLAS, 1962
  – common in today’s fast processor with slow DRAM
  – programmers don’t need to know about it for typical programs to be both fast and correct

What about atypical programs?
Modern Storage Hierarchy

- **Memory Abstraction**
  - **regfile** (10~100 words, sub-nsec)
  - **L1 cache** ~32KB, ~nsec
  - **L2 cache** ~512KB~1MB, many nsec
  - **L3 cache** ....
  - **Main memory (DRAM)** GB, ~100nsec
  - **swap disk** 100GB~TB, ~10msec

- **user SW**
  - manual
  - register
  - spilling

- **automatic (HW)**
  - cache management

- **automatic (HW+OS)**
  - demand
  - paging
Average Memory Access Time

- Memory hierarchy level $L_1$ has raw access time of $t_1$
- Average access time $T_1$ is longer than $t_1$
  - a chance (hit-rate $h_1$) you find what you want $\Rightarrow t_1$
  - a chance (miss-rate $m_1$) you don’t find it $\Rightarrow t_1 + T_2$
  - $T_1 = h_1 \cdot t_1 + m_1 \cdot (t_1 + T_2)$ and $h_1 + m_1 = 1.0$
- In general

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

Think of this as “miss penalty”

Note: $h_i$ and $m_i$ are of references missed at $L_{i-1}$ $h_{bottom-most} = 1.0$
$$T_i = t_i + m_i \cdot T_{i+1}$$

- Goal: achieve desired $T_1$ within allowed cost
  \[ T_i \approx t_i \text{ is not a goal:} \]
- Keep $m_i$ low
  - increase capacity $C_i$ lowers $m_i$, but increases $t_i$
  - lower $m_i$ by smarter management, e.g.,
    - replacement: anticipate what you don’t need
    - prefetching: anticipate what you will need
- Keep $T_{i+1}$ low
  - reduce $t_{i+1}$ with faster next level memory leads to increased cost and/or reduced capacity
  - better solved by adding intermediate levels
Memory Hierarchy Design

- **DRAM**
  - optimized for capacity-per-dollar (cost)
  - $T_{\text{DRAM}}$ is essentially same regardless of capacity
- **SRAM**
  - optimized for latency at given capacity
  - tunable tradeoff between capacity and latency possible, $t = O(\sqrt{\text{capacity}})$
- **Memory hierarchy** bridges the difference between CPU speed and DRAM speed
  - $T_{\text{pclk}} \approx T_{\text{DRAM}} \Rightarrow$ no hierarchy needed
  - $T_{\text{pclk}} \ll T_{\text{DRAM}} \Rightarrow$ one or more levels of increasingly larger but slower SRAMs to minimize $T_1$
Aside: Why is DRAM slow?

- DRAM fabrication at forefront of VLSI, but scaled with Moore’s law in capacity and cost not speed
- Between 1980 ~ 2004
  - 64K bit → 1024M bit (exponential ~55% annual)
  - 250ns → 50ns (linear)
- A deliberate engineering choice
  - memory capacity needs to grow linearly with processing speed in a balanced system – Amdahl’s Other Law
  - DRAM/processor speed difference reconcilable by SRAM cache hierarchies (L1, L2, L3, ....)

Pareto-optimal faster/smaller/more-costly DRAM do exist
Intel P4 Example
(very fast, very deep pipeline)

- 90nm, 3.6 GHz
- 16KB L1 D-cache
  - $t_1 = 4 \text{ cyc int (9 cycle fp)}$
- 1024KB L2 D-cache
  - $t_2 = 18 \text{ cyc int (18 cyc fp)}$
- Main memory
  - $t_3 = \sim 50\text{ns or 180 cyc}$

Notice:
- best case latency is not 1 cycle
- worst case access latency is 300+ cycles depending on exactly what happens
What is $m_1$ and $m_2$? (check definition)
Don’t Forget Bandwidth and Energy

• Assume RISC pipeline 1GHz and IPC=1
  – 4GB/sec of instruction fetch bandwidth
  – 1GB/sec load and 0.6GB/sec store (if 25% LW and 15% SW, Agerwala&Cocke)
  – multiply by number of cores if multicore
• DDR4 ~20GB/sec/channel (under best-case access pattern) and ~10 Watt at full blast
• With memory hierarchy

\[ \text{BW}_{i+1} = \text{BW}_1 \cdot \prod_{j=1}^{i} m_j \]

Critical for multicore and GPU
Now we can talk about caches . . .

Generically in computing, any structure that “memoizes” frequently repeated computation results to save on the cost of reproducing the results from scratch, e.g. a web cache.
Cache in Computer Architecture

- An invisible, automatically-managed memory hierarchy
- Program expects reading M[A] to return most-recently written value, with or without cache
- Cache keeps “copies” of frequently accessed DRAM memory locations in a small fast memory
  - service load/store using fast memory copies if found
  - transparent to program if memory idempotent \( (L13) \)
  - funny things happen if mmap’ed or if memory can change (e.g., by other cores or DMA)
Cache Interface for Dummies

- Like the magic memory
  - present address, R/W command, etc
  - result or update valid after a short/fixed latency
- Except occasionally, cache needs more time
  - will become valid/ready eventually
  - what to do with pipeline until then? Stall!!

[Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
The Basic Problem

• Potentially $M=2^m$ bytes of memory, how to keep “copies” of most frequently used locations in $C$ bytes of fast storage where $C << M$

• Basic issues (intertwined)
  (1) when to cache a “copy” of a memory location
  (2) where in fast storage to keep the “copy”
  (3) how to find the “copy” later on (*LW and SW only give indices into $M$*)
Basic Operation
(demand-driven version)

1. **hit?**
   - yes: return data
   - no: fetch new from \( L_{i+1} \)

2. choose location

3. cache lookup

M address

data
Basic Cache Parameters

- **M = 2^m**: size of address space in bytes
  - sample values: $2^{32}$, $2^{64}$
- **G = 2^g**: cache access granularity in bytes
  - sample values: 4, 8
- **C**: “capacity” of cache in bytes
  - sample values: 16 KByte (L1), 1 MByte (L2)

![Graph showing hit rate vs cache size (C)]
Direct-Mapped Placement (first try)

\[ \text{let } t = \log_2 M - \log_2 C \]

What about writes?

\[ \text{hit?} \]

\[ \text{G bytes} \]

\[ \text{data} \]
Storage Overhead and Block Size

• For each cache block of $G$ bytes, also storing “$t+1$” bits of tag (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% overhead; 16KB cache actually 25.5KB SRAM

• Solution: “amortize” tag over larger $B$-byte block
  – manage $B/G$ consecutive words as indivisible unit
  – if $M=2^{32}$, $B=16$, $G=4$, $C=16K$
  $\Rightarrow t=18$ bits for each 16-byte block
  15% overhead; 16KB cache actually 18.4KB SRAM

  – spatial locality also says this is good *(Q1: when)*

• Larger caches wants even bigger blocks
Direct-Mapped Placement (final)

\[ t = \log_2 M - \log_2 C \]

\[ \text{Tag Array} \]

\[ \text{Data Array} \]

\[ \text{valid} \]

\[ \text{C/B-by-B bytes} \]

\[ \text{G bytes} \]

\[ \text{data} \]

\[ \text{hit?} \]

\[ \text{lg}_2 M \text{-bit address} \]

\[ \text{tag} \quad \text{idx} \quad \text{bo} \quad \text{g} \]

\[ \text{lg}_2 (C/B) \text{ bits} \]

\[ t \text{ bits} \]

\[ \text{lg}_2 (B/G) \text{ bits} \]

\[ \text{C/B-by-t bits} \]
Basic Cache Parameters

- **M = 2^m**: size of address space in bytes
  - sample values: $2^{32}$, $2^{64}$
- **G = 2^g**: cache access granularity in bytes
  - sample values: 4, 8
- **C**: “capacity” of cache in bytes
  - sample values: 16 KByte (L1), 1 MByte (L2)
- **B = 2^b**: “block size” in bytes
  - sample values: 16 (L1), >64 (L2)
- **a**: “associativity” of the cache
  - sample values: 1, 2, 4, 5(?),... “C/B”

C/a should be a 2-power

to be continued