CS250: Memory Hierarchy [Adapted from A. Roth]

CS250
Computer Organization and Design

Memory Hierarchy

• Basic concepts
• Technology background
• Organizing a single memory component
  • ABC
  • Write issues
  • Miss classification and optimization
• Organizing an entire memory hierarchy
  • Virtual memory
    • Highly integrated into real hierarchies, but...
    • ...won’t talk about until later

CPU
Mem
I/O
System software
App
App
App

Admin
• Homework 3: Due Friday
  • Questions? Concerns?
• Next homework:
  • We’ll do C instead of LogiSim
• Midterm: Nov 9
• Reading: Ch 5

CS250: Memory Hierarchy [Adapted from A. Roth]

How Do We Build Insn/Data Memory?

• Pretended memory elements were many DFFs
  • How many?
    • $2^{32} = 4\text{GB}$!!!
      • It would be huge, expensive, and pointlessly slow
      • And we can’t build something that big on-chip anyway
  • Good news: most ISAs now 64-bit → memory is $2^{64} = 16\text{EB}$

So What Do We Do? Actually...

• “Primary” in/data memory are single-ported SRAMs...
  • “primary” = “in the datapath”
  • Key 1: they contain only a dynamic subset of “memory”
    • Subset is small enough to fit in a reasonable SRAM
  • Key 2: missing chunks fetched on demand (transparent to program)
    • From somewhere else... (next slide)
  • Program has illusion that all 4GB (16EB) of memory is physically there
    • Just like it has the illusion that all insns execute atomically

But...

• If requested in/data not found in primary memory
  • Doesn’t the place it comes from have to be a 4GB (16EB) SRAM?
  • And won’t it be huge, expensive, and slow? And can we build it?
CS250: Memory Hierarchy [Adapted from A. Roth]

Memory Overview

- **Functionality**
  - "Like a big array…"
  - N-bit *address* bus (on N-bit machine)
  - Data bus: typically read/write on same bus
  - Can have multiple *ports*: address/data bus pairs

- **Access time:**
  - Access latency ~ \#bits \* \#ports

Memory Performance Equation

- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another component
  - **Fill**: action of placing data in M
    - \%miss (miss-rate): \#misses \/#accesses
    - \( t_{hit} \): time to read data from (write data to) M
    - \( t_{miss} \): time to read data into M
  - **Performance metric**: average access time
    - \( t_{avg} = t_{hit} + \%miss \* t_{miss} \)

Memory Hierarchy

- \( t_{miss} = t_{hit} + \%miss \* t_{miss} \)
- **Problem**: hard to get low \( t_{hit} \) and \%miss in one structure
  - Large structures have low \%miss but high \( t_{hit} \)
  - Small structures have low \( t_{hit} \) but high \%miss
- **Solution**: use a hierarchy of memory structures
  - Known from the very beginning

"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, Von Neumann

"Preliminary discussion of the logical design of an electronic computing instrument"
IAS memo 1946

Why Memory Hierarchy Works

- **10/90 rule (of thumb)**
  - 10% of static insns/data account for 90% of accessed insns/data
  - **Insns**: inner loops
  - **Data**: frequently used globals, inner loop stack variables

- **Temporal locality**
  - Recently accessed insns/data likely to be accessed again soon
  - **Insns**: inner loops (next iteration)
  - **Data**: inner loop local variables, globals
  - **Hierarchy can be "reactive":** move things up when accessed

- **Spatial locality**
  - Insns/data near recently accessed insns/data likely accessed soon
  - **Insns**: sequential execution
  - **Data**: elements in array, fields in struct, variables in stack frame
  - **Hierarchy is "proactive":** move things up speculatively

Exploiting Heterogeneous Technologies

- **Apparent problem**
  - Lower level components must be huge
    - Huge SRAMs are difficult to build and expensive

- **Solution**: don’t use SRAM for lower levels
  - **Cheaper, denser storage technologies**
    - Will be slower than SRAM, but that’s OK
    - Won’t be accessed very frequently
    - We have no choice anyway
  - **Upper levels**: SRAM \( \rightarrow \) expensive (/B), fast
  - **Going down**: DRAM, Disk \( \rightarrow \) cheaper (/B), fast
Memory Technology Overview

- **Latency**
  - SRAM: <1 to 5ns (on chip)
  - DRAM: ~100ns — 100x or more slower than SRAM
  - Disk: 10,000,000ns or 10ms — 100,000x slower than DRAM
  - Flash: ~200ns — 2x slower than SRAM (read, much slower writes)

- **Bandwidth**
  - SRAM: 10-100GB/sec
  - DRAM: ~1GB/sec — 10x less than SRAM
  - Disk: 100MB/sec (0.1 GB/sec) — sequential access only
  - Flash: about same as DRAM for read (much less for writes)

- **Cost**:
  - SRAM: 4MB
  - DRAM: 1,000MB (1GB) — 250x cheaper than SRAM
  - Disk: 400,000MB (400GB) — 400x cheaper than DRAM
  - Flash: 4,000 MB (4GB) — 4x cheaper than DRAM

(Traditional) Concrete Memory Hierarchy

- 1st level: I$D$ (insn/data caches)
- 2nd level: L2 (cache)
- 3rd level: L3 (cache)
- N-1 level: main memory
- N level: disk (swap space)

Virtual Memory Teaser

- For 32-bit ISA
  - 4GB disk is easy
  - Even 4GB main memory is common

- For 64-bit ISA
  - 16GB main memory is right out
  - Even 16GB disk is extremely difficult

- Virtual memory
  - Never referenced addresses don’t have to physically exist anywhere!
  - Next week...

Start With “Caches”

- "Cache": hardware managed
- SRAM technology
- Cache organization
- Some example calculations

Why Are There 2-3 Levels of Cache?

- "Memory Wall": memory 100X slower than primary caches
  - Multiple levels of cache needed to bridge the difference

- "Disk Wall": disk is 100,000X slower than memory
  - Why aren’t there 56 levels of main memory to bridge that difference?
  - Doesn’t matter: program can’t keep itself busy for 10M cycles
  - So slow, may as well swap out and run another program

Evolution of Cache Hierarchies

- Chips today are 30–70% cache by area
RAM and SRAM

- Reality: large storage arrays implemented in "analog" way
  - Not as flip-flops (FFs) + giant muxes
- RAM (random access memory)
  - Ports implemented as shared buses called wordlines/bitlines
- SRAM: static RAM
  - Static = bit maintains its value indefinitely, as long as power is on
  - Bits implemented as cross-coupled inverters (CCIs)
  + 2 gates, 4 transistors per bit
  - All processor storage arrays: regfile, caches, branch predictor, etc.
- Other forms of RAM: Dynamic RAM (DRAM), Flash

Basic RAM

- Storage array
  - M words of N bits each (e.g., 4w, 2b each)
- RAM storage array
  - M by N array of CCIs (e.g., 4 by 2)
- RAM port
  - Grid of wires that overlays bit array
  - M wordlines: carry 1H decoded address
  - N bitlines: carry data
- RAM port operation
  - Send address → 1 wordline goes high
  - "bits" on this line read/write bitline data
  - Operation depends on bit/W/B connection
    + "Magic" analog stuff

ROMS:

- ROMs = Read Only memory
- ROMs = Similar layout (wordlines, bitlines) to RAMs
  - Except not writeable: fixed connections to Power/Gnd instead of CCI
  - Also EPROMs
  - Programmable once electronically
  - And EEPROMs
  - Eraseable and re-programable (very slow)

Two SRAM Styles

- Regfile style
  - Modest size: <4KB
  - Many ports: some read-only, some write-only
  - Write and read both take half a cycle (write first, read second)

- Cache style
  - Larger size: >8KB
  - 1 or 2 ports (mostly 1): read/write in a single port
  - Write and read can both take full cycle (or more)

SRAM Read/Write Port

- Cache: read/write on same port
  - Not at the same time
  - Trick: write port with additional bitline
  - "Double-ended" or "differential" bitlines
  - Smaller → faster than separate ports
- Two phase "edge" read
  - Phase I: CLK = 0
  - Equalize voltage on bitline pairs (0.5)
  - Phase II: CLK = 1
  - SRAM cells "swing" bitline voltage up on 1 side (0.6), down on 0 side (0.4)
  - Sense-amplifiers (giants CCIs) "detect" swing, quickly set outputs to full 0/1
**RAM Latency Model**

- **RAM access latency**: \( \sim \text{wire delay}^2 \)  
  \( \sim (\text{Wordline length})^2 + (\text{Bitline length})^2 \)  
  \( \sim (\#ports \times N)^2 + (\#ports \times M)^2 \)  
  \( \sim 2 \times \#ports \times \#bits \times \text{bandwidth} + \#ports \times \text{capacity} \)

- To minimize \( M + N \), RAMs "squarified" so \( N=M=\sqrt{NM} \)

- Want large capacity, high bandwidth, low latency
- Can’t have all three in one SRAM → choose one, at most two
- Regfile: ++ bw (3-4 ports), + latency (1 cycle), – size (1-4 KB)
- D$/I$: + size (8-64 KB), + latency (1-3 cycles), – bw (1 port)
- L2: +++ size (128 KB-2 MB), – latency (10-20 cycles), – bw (1 port)

---

**Logical Cache Organization**

- The setup
  - 32-bit ISA → 4B words/addresses, 232 B address space

- Logical cache organization
  - 4KB, organized as 1K 4B blocks
  - Each block can hold one word

- Physical cache implementation
  - 1K (1024) by 4B (32) SRAM
  - Called data array

- 10-bit address input  
  - 32-bit data input/output

---

**Looking Up A Block**

- Q: which 10 of the 32 address bits to use?  
  - A: bits [11:2]
    - 2 LS bits [1:0] are the offset bits  
      - Locate byte within word
    - Next 10 LS bits [11:2] are the index bits  
      - These locate the word  
      - Nothing says index must be these bits  
      - But these work best in practice  
      - Why? (think about it)

---

**Knowing that You Found It**

- Each cache row corresponds to \( 2^{10} \) blocks  
  - How to know which if any is currently there?  
  - Build separate and parallel tag array  
    - 1K by 21-bit SRAM  
    - 20-bit (next slide)

- Lookup algorithm
  - Read tag indicated by index bits  
  - (Tag matches & valid bit set)

- Hit → data is good  
  - Miss → data is garbage, wait...

---

**Handling a Cache Miss**

- What if requested word isn’t in the cache?
  - How does it get in there?
  - More on this later

- Cache controller: FSM  
  - Remembers miss address  
  - Pings next level of memory  
  - Waits for response  
  - Writes data/tag into proper locations  
  - All of this happens on the fill path  
  - Sometimes called backside

- Cache misses hurt CPI  
  - More on this later

---

**Tag Overhead**

- “4KB cache” means cache holds 4KB of data (capacity)  
  - Valid bit usually not counted  
  - Tag overhead = tag size / data size

- 4KB cache with 4B blocks?  
  - 4B blocks → 2-bit offset  
  - 4KB cache / 4B blocks → 1024 blocks → 10-bit index  
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag  
  - 20-bit tag / 32-bit block = 63% overhead
Cache Misses and Pipeline Stalls

- I$ and D$ misses stall pipeline just like data hazards
  - Stall logic driven by miss signal
  - Cache "logically" re-evaluates hit/miss every cycle
  - Block is filled → miss signal de-asserts → pipeline restarts

CPI Calculation with Cache Misses

- In a pipelined processor, I$/D$ miss is "built in" (effectively 0)
  - High miss will simply require multiple F or M stages

Parameters

- 5 stage pipeline: FDXMW, base CPI = 1
-Insn mix: 30% loads/stores
- I$: %miss = 2%, tmiss = 10 cycles
- D$: %miss = 10%, tmiss = 10 cycles

What is new CPI?

- CPI_I$ = \%missI$ * tmiss = 0.02 * 10 cycles = 0.2 cycle
- CPI_D$ = \%load/store * \%missD$ * tmissD$ = 0.3 * 0.1 * 10 cycles = 0.3 cycle
- CPINew = CPI_I$ + CPI_D$ = 1 + 0.2 + 0.3 = 1.5

Measuring Cache Performance

- Ultimate metric is tavg
  - Cache capacity roughly determines thit
  - Lower-level memory structures determine tmiss
  - Measure \%miss
  - Hardware performance counters (Pentium)
  - Performance Simulator
  - Paper simulation (next slide)

Cache Miss Paper Simulation

- 8B cache, 2B blocks
  - Figure out number of sets: 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag bits
  - Offset: least-significant log₂(block-size) = log₂(2) = 1 → 0000
  - Index: next log₂(number-of-sets) = log₂(4) = 2 → 0000
  - Tag: rest = 4 – 1 – 2 = 1 → 0000

Cache diagram

<table>
<thead>
<tr>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
</tbody>
</table>

Cache Miss Paper Simulation

- 88 cache, 2B blocks
  - Access address 1100: what happens?

- Set is 10. Tag does not match, so...
Cache Miss Paper Simulation

- 8B cache, 2B blocks

Cache contents (prior to access) Address Outcome

<table>
<thead>
<tr>
<th>Set 00</th>
<th>Set 01</th>
<th>Set 10</th>
<th>Set 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0001</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0100</td>
<td>0101</td>
</tr>
<tr>
<td>1100</td>
<td>1000</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1100</td>
<td>1000</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1100</td>
<td>1000</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1100</td>
<td>1000</td>
<td>1100</td>
<td>1100</td>
</tr>
</tbody>
</table>

- Line is a miss. Fill the cache so its there next time.

- You all try the rest...

How to reduce %miss? And hopefully tavg?

Capacity and Performance

- Simplest way to reduce %miss: increase capacity
  - Miss rate decreases monotonically
  - "Working set": insns/data program is actively using
    - t<sub>hit</sub> increases
    - t<sub>avg</sub>?
  - Given capacity, manipulate %miss by changing organization

Block Size

- One possible re-organization: increase block size
  - Exploit spatial locality
  - Caveat: increase conflicts too
  - Increases t<sub>hit</sub> need word select mux
    - By a little, not too bad
  - Reduce tag overhead

Block Size and Tag Overhead

- 4KB cache with 1024 4B blocks?
  - 48 blocks → 2-bit offset, 1024 frames → 10-bit index
  - 32-bit address = 3-bit offset + 10-bit index = 20-bit tag
  - 20-bit tag / 32-bit block = 63% overhead

- 4KB cache with 512 8B blocks
  - 8B blocks → 3-bit offset, 512 frames → 9-bit index
  - 32-bit address = 3-bit offset + 9-bit index = 20-bit tag
  - 20-bit tag / 64-bit block = 32% overhead
  - Notice: tag size is same, but data size is twice as big

- A realistic example: 64KB cache with 64B blocks
  - 16-bit tag / 512-bit block = ~ 2% overhead
Block Size Cache Miss Paper Simulation

- 8B cache, 4B blocks

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0: 11000010011011111111 1010</td>
<td>Hit (spatial locality)</td>
<td></td>
</tr>
<tr>
<td>10000010011011111111 0011</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>10011010011011111111 1000</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>10001010011011111111 1000</td>
<td>Miss (conflict)</td>
<td></td>
</tr>
<tr>
<td>00001010011011111111 0000</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>00011010011011111111 0011</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>00101010011011111111 1000</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>00111010011011111111 1000</td>
<td>Miss (conflict)</td>
<td></td>
</tr>
<tr>
<td>10001010011011111111 1000</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>10011010011011111111 0011</td>
<td>Miss</td>
<td></td>
</tr>
</tbody>
</table>

- Spatial "prefetching": miss on 1100 brought in 1110
- Conflicts: miss on 1000 kicked out 0011

Block Size and Miss Penalty

- Does increasing block size increase $t_{miss}$?
  - Don’t larger blocks take longer to read, transfer, and fill?
  - They do, but...

- $t_{miss}$ of an isolated miss is not affected
  - Critical Word First / Early Restart (CRF/ER)
    - Requested word fetched first, pipeline restarts immediately
    - Remaining words in block transferred/filled in the background

- $t_{miss}$ of a cluster of misses will suffer
  - Reads/transfer/fills of two misses cannot be overlapped
  - Latencies start to pile up
  - This is technically a bandwidth problem (more later)

Conflicts

- 88 cache, 2B blocks

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0: 00001010011011111111 1110</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>00001010011011111111 0111</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>00001010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>00011010011011111111 0111</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>00101010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>00111010011011111111 0111</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10001010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10011010011011111111 0111</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10101010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10111010011011111111 0111</td>
<td>Hit</td>
<td></td>
</tr>
</tbody>
</table>

- Pairs like 0000/1000 will always conflict
  - Regardless of block-size and capacity (assuming capacity < 16)
  - Q: can we allow pairs like these to simultaneously reside?
    - A: yes, reorganize cache to do so

Associativity and Miss Paper Simulation

- 88 cache, 2B blocks, 2-way set-associative

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0: 00001010011011111111 1110</td>
<td>Miss</td>
<td></td>
</tr>
<tr>
<td>10001010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10011010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10101010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
<tr>
<td>10111010011011111111 1010</td>
<td>Hit</td>
<td></td>
</tr>
</tbody>
</table>

- Avoid conflicts: 0000 and 1000 can both be in set 0
  - Introduce some new conflicts: notice address re-arrangement
    - Happens, but conflict avoidance usually dominates
Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Belady’s (oracle): block that will be used furthest in future
- Random
- FIFO (first-in first-out)
- LRU (least recently used)
  - Fits with temporal locality, LRU = least likely to be used in future
- NMRU (not most recently used)
  - An easier to implement approximation of LRU
  - Equal to LRU for 2-way SA caches

NMRU Implementation

- Add MRU field to each set
  - MRU data is encoded “way”
  - Hit? update MRU
  - Fill? write enable ~MRU

Associativity And Performance

- The associativity game
  - Higher associative caches have lower %miss
    - \( t_m \) increases
  - But not much for low associativities (2,3,4,5)
  - \( t_m \)?

- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
  - 5-way set-associativity? No problem

Full Associativity

- How to implement full (or at least high) associativity?
  - This way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?

Full-Associativity with CAMs

- CAM: content associative memory
  - Array of words with built-in comparators
  - Input is data (tag)
  - Output is 1H encoding of matching slot
- FA cache
  - Tags as CAM, data as RAM
  - Effective but expensive (EE reasons)
  - Upshot: used for 16-/32-way associativity
  - No good way to build 1024-way associativity
  - No real need for it, either

CAM Circuit (for Kicks)

- Two phase match
  - Phase I: CLK = 0
    - Pre-charge wordlines to 1
  - Phase II: CLK = 1
    - Enable matchlines
    - Non-matching bit discharges wordline

- Like RAM read in reverse
  - RAM read
    - Input: wordline (address)
    - Output: bitline (data)
  - CAM match
    - Input: matchline (data)
    - Output: wordline (address)
Analyzing Misses: 3C Model (Hill)

- Divide cache misses into categories based on cause
  - Compulsory: block size is too small (i.e., address not seen before)
  - Capacity: capacity is too small
  - Conflict: associativity is too low

- How to classify in hand simulation?
  - Compulsory: easy
  - Capacity: consecutive accesses to block separated by access to at least $N$ other distinct blocks where $N$ is number of frames in cache
  - Conflict: all other misses

---

Cache Miss (3C Model) Paper Simulation

- 8B cache, 2B blocks, direct-mapped

### Cache contents (prior to access) | Address | Outcome
--- | --- | ---
0000|0001 | Set 00 | Miss (compulsory)
0000|0001 | Set 00 | Miss (compulsory)
0000|0001 | Set 00 | Miss (compulsory)
0000|0001 | Set 00 | Miss (compulsory)
0000|0001 | Set 00 | Miss (compulsory)
0000|0001 | Set 00 | Miss (compulsory)
0000|0001 | Set 00 | Miss (compulsory)

- Capacity miss: 1100, 1110, 1000, 0001 accessed since 0000
- Conflict miss: only 0000 accessed since 1000

---

### ABC

- **Capacity**
  - Decreases capacity misses
  - Increases $t_{hit}$

- **Associativity**
  - Decreases conflict misses
  - Increases $t_{hit}$

- **Block size**
  - Increases conflict misses
  - Increases or decreases capacity misses
  - Little effect on $t_{hit}$ may exacerbate $t_{miss}$

- Why do we care about 3C miss model?
  - So that we know what to do to eliminate misses
  - If you don’t have conflict misses, increasing associativity won’t help

---

### Two Optimizations

- **Victim buffer**: for conflict misses
  - Technical: reduces $t_{miss}$ for these misses, doesn’t eliminate them
  - Depends how you do your accounting

- **Prefetching**: for capacity/compulsory misses

---

### Victim Buffer

- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed
  - 3 blocks mapping to same 2-way set and accessed (XYZ)+

- **Victim buffer (VB)**: small FA cache (e.g., 4 entries)
  - Small so very fast
  - Blocks kicked out of cache placed in VB
  - On miss, check VB: hit ? Place block back in cache
  - 4 extra ways, shared among all sets
  - Only a few sets will need it at any given time
  - On cache fill path: reduces $t_{miss}$ no impact on $t_{hit}$
  - Very effective in practice

---

### Prefetching

- **Prefetching**: put blocks in cache proactively/speculatively
  - In software: insert prefetch (non-binding load) insns into code
  - In hardware: cache controller generates prefetch addresses

- Keys: anticipating upcoming miss addresses accurately
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Accuracy**: don’t evict useful data
  - Prioritize misses over prefetches

- Simple algorithm: **next block prefetching**
  - Miss address X → prefetch address X+block-size
  - Works for insns: sequential execution
  - What about non-sequential execution?
  - Works for data: arrays
  - What about other data-structures?
  - Address prediction is actively researched area

---
"Streaming"
- Suppose you are doing some media computation
- Data you bring in will be used once and discarded
- Why cache it? Just to kick out something else potentially useful?

Streaming/cache-bypassing
- Read (miss): "cache block" goes straight to some big register
- Write: same (in reverse)
- Intel SSE2 supports this (and has the big registers)

Write Issues
- So far we have looked at reading from cache
  - insn fetches, loads
- What about writing into cache
  - Stores, not an issue for insn caches (why they are simpler)
- Several new issues
  - Tag/data access
  - Dirty misses and write-backs
  - Write misses

Tag/Data Access
- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK, stall until good data arrives)
- Writes: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - Tag match in SA cache → which way was written into?
- In pipelined processors: writes are a pipelined 2 cycle process
  - Cycle 1: match tag
  - Cycle 2: write to matching way
- Some more issues we’ll talk about when we get to pipelining.

Stores and Loads
- How do loads and stores work together in D$?
  - Loads: parallel tag/data access, 1 cycle
  - Stores: serial tag/data access, 2 cycles
- Four cases
  - Store-after-load: no problem
  - Store-after-store: no problem
  - Load-after-load: no problem
  - Load-after-store: two problems
- Structural hazard in data array
  - Stall? May not have to if read/write to different offset in a block
  - Or buffer store and do later...
  - Data hazard: store/load to same address
  - Stall? No, bypass

Write Propagation
- When to propagate new value to (lower level) memory?
- Write-thru: immediately
  - Requires additional bus bandwidth
  - Not common
- Write-back: when block is replaced
  - Requires additional "dirty" bit per block
  - Clean miss: one transaction with lower level (fill)
  - Dirty miss: two (writeback + fill)
- Write-back buffer: allows you to do fill first
  - Write dirty block to buffer (fast)
  - Read (fill) new block from next-level to cache
  - Write buffer contents to next-level
  - Minimal bus bandwidth (only writeback dirty blocks)
  - Combine with victim buffer

Write Misses and Write Buffers
- Read miss?
  - Pipeline has to stall, load can’t go on without the data
- Write miss?
  - Technically, no insn is waiting for data, why stall?
- Write buffer: a small FIFO
  - Stores write address/value to write buffer, pipeline keeps
  - Write buffer writes stores to D$ in the background
  - Loads must read write buffer in addition to D$
  - Eliminates stalls on write misses (mostly)
    - Creates some problems (later)
- Write buffer vs. writeback/victim-buffer
  - Write: "in front" of D$, for store misses
  - Writeback: "in back" of D$, for dirty writebacks/conflict misses
**Write Miss Handling**

- **How is a write miss actually handled?**

  - **Write-allocate**: read block from lower level, write into it
    - Decreases read misses
    - Requires additional bandwidth
    - Commonly used
  
  - **Write-non-allocate**: just write to next level
    - Potentially more read misses
    - Uses less bandwidth
    - Use with write-thru

**Performance Calculation**

- **Memory system parameters**
  - $D$: $t_{hit} = 1\text{ns}$, $\%miss = 10\%$, 50% dirty, write-back-buffer, write-buffer
  - Main memory: $t_{hit} = 50\text{ns}$

- **Reference stream**: 20% stores, 80% loads

- **What is $t_{avgD}$?**
  - Write-buffer $\rightarrow$ no store misses
  - Write-back-buffer $\rightarrow$ no dirty misses

- $t_{missD} = t_{hitM}$

- $t_{avgD} = t_{hitD} + \%loads \times t_{missD} = 1\text{ns} + (0.8 \times 0.10 \times 50\text{ns}) = 5\text{ns}$

**Performance Calculation Cont’d**

- **Memory system parameters**
  - $D$: $t_{hit} = 10\text{ns}$, $\%miss = 20\%$, 50% dirty, no write-back-buffer

- **What is new $t_{avgD}$?**
  - No write-back buffer $\rightarrow$ dirty misses cost double

- $t_{missD} = t_{hitL2}$

- $t_{avgD} = t_{hitD} + (1 + \%dirty) \times \%missL2 \times t_{hitM} = 10\text{ns} + (1.5 \times 0.2 \times 50\text{ns}) = 25\text{ns}$

- $t_{avgL2} = t_{hitL2} + (1 + \%dirty) \times t_{hitM} = 10\text{ns} + (1.5 \times 0.05 \times 25\text{ns}) = 2\text{ns}$

**Designing a Cache Hierarchy**

- **For any memory component**: $t_{hit}$ vs. $\%miss$ tradeoff

- **Upper components ($I$, $D$)** emphasize low $t_{hit}$
  - Frequent access $\rightarrow$ $t_{hit}$ important
  - $t_{miss}$ is not bad $\rightarrow$ $\%miss$ less important

- **Lower components ($L2$, $L3$)** emphasis turns to $\%miss$
  - Infrequent access $\rightarrow$ $t_{hit}$ less important
  - $t_{miss}$ is bad $\rightarrow$ $\%miss$ important

- **High capacity/associativity/block size** (to reduce $\%miss$)

**Memory Hierarchy Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>$I$/D$</th>
<th>L2</th>
<th>L3</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{hit}$</td>
<td>3ns</td>
<td>10ns</td>
<td>50ns</td>
<td>100ns</td>
</tr>
<tr>
<td>$t_{miss}$</td>
<td>10ns</td>
<td>30ns</td>
<td>100ns</td>
<td>10ms (10M ns)</td>
</tr>
<tr>
<td>Capacity</td>
<td>8-64KB</td>
<td>128KB-2MB</td>
<td>1-4MB</td>
<td>7</td>
</tr>
<tr>
<td>Block size</td>
<td>16-32B</td>
<td>32-256B</td>
<td>256B</td>
<td>7</td>
</tr>
<tr>
<td>Associativity</td>
<td>1-4</td>
<td>4-16</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>Replacement</td>
<td>NRU</td>
<td>NRU</td>
<td>NRU</td>
<td>7</td>
</tr>
<tr>
<td>Prefetching</td>
<td>Maybe</td>
<td>Probably</td>
<td>Probably</td>
<td>7</td>
</tr>
</tbody>
</table>

- **Some other design parameters**
  - Split vs. unified insns/data
  - Inclusion vs. exclusion vs. nothing
  - On-chip, off-chip, or partially on-chip?
  - SRAM or embedded DRAM?

**Split vs. Unified Caches**

- **Split $I$/D$: insns and data in different caches
  - To minimize structural hazards and $t_{hit}$
  - Larger unified $I$/D$ would be slow, 2nd port even slower
  - Optimize $I$ for wide output (superscalar), no writes
  - Why is 486 I/D$ unified?

- **Unified L2, L3**: insns and data together
  - To minimize $\%miss$
  - Fewer capacity misses: unused insn capacity can be used for data
  - More conflict misses: insn/data conflicts
  - A much smaller effect in large caches
  - Insn/data structural hazards are rare: simultaneous $I$/D$ miss
  - Go even further: unify L2, L3 of multiple cores in a multi-core
Inclusion vs. Exclusion

- **Inclusion**: \( M_n \) must contain superset of \( M_{n-1} \) (transitively)
  - Makes multiprocessing and I/O more efficient (later)
  - Kind of a pain to enforce

- **Exclusion**: \( M_n \) must be disjoint from \( M_{n-1} \) (transitively)
  - Reduces waste in small L2 caches
  - Also a pain enforce

- **No special relationship**: \( M_n \) not necessarily related to \( M_{n-1} \)

Off-Chip (L2, L3...) Caches

- Off-chip L2's, L3's quite common
- Separate die, same package or in chipset
  - Somewhat longer \( t_{hit-L2} \)
  - Higher packaging cost
  - Lower manufacturing cost
  - Product differentiation!

- What are *-Xeon and *-Celeron?
  - Big L2 and small L2 versions of *
  - PentiumIII: 256kB L2, Celeron: 128kB, Xeon 512kB
  - Xeon for people willing to pay top dollar for 15% more speed
  - Celeron for people who want "value"
  - Off-chip L2's make these easier (Celerons have on-chip L2s)

On-Chip Tags, Off-Chip Data

- Another common design for "last-level" (L2 or L3) caches
  - Tags on-chip, data off-chip
    - E.g., IBM Power5 L3

  - Implies access tags first, then data
    - Increases \( t_{hit} \) by ~3 cycles
    - That's OK, L2 \( t_{hit} \) initially 10+ cycles
    - Would not want to use this for I$\rightarrow$D$
    - Allows you to determine whether block in L2 quickly
    - Allows you to access only one L2 way to read data (low power)
    - Maintain most of manufacturing/product differentiation advantages
    - Think of it as "moving tags back on-chip"

Cache Hierarchy Examples

- Intel 486: unified 8KB I/D$, no L2
- IBM Power 5: private 64KB I/D$, shared 1.5M L2, L3 tags
  - Dual core: L2, L3 tags shared between both cores

- Desktop Intel Pentium III
  - 16KB 16KB 256KB none
  - Intel Pentium 4 "128KB" 9KB 256–512KB none
  - AMD Athlon 64 64KB 64KB off-chip none

- Server Intel Itanium II
  - 16KB 16KB 256KB 3MB
  - IBM Power4 64KB 64KB 1.5MB tags

- Design caches for target workload
  - Desktops: relatively small datasets, < 1MB cache probably enough
  - Server: large datasets, > 2MB cache
  - Design cache hierarchy all at once
  - Intel: small I$/D$ → large L2/L3, preferably on-chip
  - AMD/IBM: large I$/D$ → small (or off-chip or no) L2/L3

Designing a Complete Memory Hierarchy

- SRAM vs. embedded DRAM...

  - Good segue to main memory
    - How is that component designed?
    - What is DRAM?
    - What makes some DRAM "embedded"?
Brief History of DRAM

- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
  - More closely resembles today’s disks than memory
  - And by mercury delay lines before that (ENIAC)
  - Re-circulating vibrations in mercury tubes

"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory... It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"  

Maurice Wilkes
Memoirs of a Computer Programmer, 1985

SRAM

- SRAM: "6T" cells
  - 6 transistors per bit
    - 4 for the CCI
    - 2 access transistors

- Static
  - CCI’s hold state

- To read
  - Equalize, swing, amplify

- To write
  - Overwhelm

DRAM

- DRAM: dynamic RAM
  - Bits as capacitors
  - Transistors as ports
  - "1T" cells: one access transistor per bit

"Dynamic" means
  - Capacitors not connected to pwr/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed

- Designed for density
  + ~6–8X denser than SRAM
  - But slower too

DRAM Operation I

- Read: similar to cache read
  - Phase I: pre-charge bitlines to 0.5V
  - Phase II: decode address, enable wordline
    - Capacitor swings bitline voltage up/down
    - Sense-amplifier interprets swing as 1(0)
    - Destructive read: word bits now discharged

- Write: similar to cache write
  - Phase I: decode address, enable wordline
  - Phase II: enable bitlines
  - High bitlines charge corresponding capacitors
  - What about leakage over time?

DRAM Operation II

- Solution: add set of D-latches (row buffer)
  - Read: two steps
    - Step I: read selected word into row buffer
    - Step II: read row buffer out to pins
  - Step III: write row buffer back to selected word
  + Solves “destructive read” problem

- Write: two steps
  - Step I: read selected word into row buffer
  - Step II: write data into row buffer
  - Step III: write row buffer back to selected word
  + Also solves leakage problem
**DRAM Refresh**

- DRAM periodically refreshes all contents
  - Loops through all words
  - Reads word into row buffer
  - Writes row buffer back into DRAM array
- 1–2% of DRAM time occupied by refresh

**Access Time and Cycle Time**

- DRAM access slower than SRAM
  - Not electrically optimized for speed, buffered access
  - SRAM access latency: 1–3ns
  - DRAM access latency: 30–50ns
- DRAM cycle time also longer than access time
  - **Access time**: latency, time of a single access
  - **Cycle time**: bandwidth, time between start of consecutive accesses
  - SRAM: cycle time < access time
  - DRAM: cycle time = 2 * access time
  - Why? Can't begin new access while DRAM is refreshing row

**DRAM Organization**

- Large capacity: e.g., 64–256Mb
  - Arranged as square
  - Minimizes wire length
  - Maximizes refresh efficiency
- Embedded (on-chip) DRAM
  - That's it
  - Huge data bandwidth

**Aside: Non-Volatile CMOS Storage**

- Before we leave the subject of CMOS storage technology...
- SRAM: storage as gate capacitor
  - Requires multiple transistors arranged in a physical loop
- DRAM: storage as trench capacitor
  - Requires a different kind of loop: refresh
- Another increasingly important kind: flash
  - Flash: storage as polysilicon/silicon capacitor with SiO\(_2\) insulator
  - Effectively no leakage (key feature)
  - Non-volatile: remembers state when power is off
  - Takes much longer to write
  - Used more of a programmable ROM than a RAM
- EEPROM: electrically-erasable programmable ROM

**CMOS Floating-Gate Storage**

- **Floating gate transistor**
  - Like a normal CMOS transistor with an additional (floating) gate
  - 1/0 interpreted by whether transistor conducts or not
  - How does this work?
    - Floating gate is completely encased in SiO\(_2\)
    - If floating-gate is uncharged, control-gate can open/close channel
    - Transistor cannot conduct, this is a 0
  - Read latency is \(\approx 100\)ns (\(\approx 3\times\) slower than DRAM)

- **How does floating gate charge and discharge?**
  - By quantum tunneling (aka "hot electron injection")
    - Charging (writing a 0) requires huge voltage \(\rightarrow\) slow
    - Write latency is \(\approx 30\) ns (300X slower than reads)
  - Discharging (re-writing a 1) requires huger voltage \(\rightarrow\) very slow
    - Erase latency is \(\approx 500\) ms (slower than disk)
  - Log-structured (infrequently erasing) file system
  - Most natural storage model for this combination of latencies
  - More about flash when we talk about I/O
Mixed CMOS Storage

- Multiple CMOS storage technologies on one chip?
  - DRAM and SRAM? SRAM and Flash? All three?
- Difficult because processes are different ...
  - DRAM: heavy doping, few metal layers (2), low voltage
  - SRAM: light doping, many metal layers (5–8), low voltage
  - Flash: light doping, few metal layers (2), high voltage
- ... but not impossible
  - Both will be sub-optimal, or one will be highly sub-optimal
  - Embedded DRAM (DRAM + SRAM) is already in use
    - Supercomputers (e.g., IBM BlueGene)
    - Some gaming systems (e.g., Sony Playstation 2)
    - Basically gives you a huge (but somewhat slower) cache
- Made easier with 3D die-stacking (wow!)

Commodity DRAM

- Commodity (standalone) DRAM
  - Cheaper packages — few pins
  - Narrow data interface: e.g., 8 bits
  - Narrow address interface: N/2 bits
- Two-level addressing
  - Level 1: RAS high
    - Upper address bits on address bus
    - Read row into row buffer
  - Level 2: CAS high
    - Lower address bits on address bus
    - Mux row buffer onto data bus

Moore’s Law

<table>
<thead>
<tr>
<th>Year</th>
<th>Capacity</th>
<th>$/MB</th>
<th>Access time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64Kb</td>
<td>$1500</td>
<td>250ns</td>
</tr>
<tr>
<td>1988</td>
<td>4Mb</td>
<td>$50</td>
<td>120ns</td>
</tr>
<tr>
<td>1996</td>
<td>64Mb</td>
<td>$10</td>
<td>60ns</td>
</tr>
<tr>
<td>2004</td>
<td>1Gb</td>
<td>$0.5</td>
<td>35ns</td>
</tr>
</tbody>
</table>

- (Commodity) DRAM capacity
  - 16X every 8 years is 2X every 2 years
  - Not quite 2X every 18 months but still close

Memory Bus

- Memory bus: connects CPU package with main memory
  - Has its own clock
  - Typically slower than CPU internal clock: 100–500MHz vs. 3GHz
  - SDRAM operates on this clock
  - Is often itself internally pipelined
    - Clock implies bandwidth: 100MHz -> start new transfer every 10ns
    - Clock doesn’t imply latency: 100MHz !– transfer takes 10ns
  - Bandwidth is more important: determines peak performance

Effective Memory Latency Calculation

- CPU ↔ main memory interface: L2 miss blocks
  - What is latency L?
- Parameters
  - L2 with 128B blocks
  - DIMM with 20ns access, 40ns cycle SDRAMs
  - 200MHz (and 5ns latency) 64-bit data and address buses
  - 5ns (address) + 20ns (DRAM access) + 16 * 5ns (bus) = 105ns
  - Roughly 300 cycles on 3GHz processor
  - Where did we get 16 bus transfers? 128B / (8B / transfer)
  - Calculation assumes memory is striped across DRAMs in 1B chunks
Memory Latency and Bandwidth

- Nominal **clock frequency** applies to CPU and caches
- Careful when doing calculations
  - Clock frequency increases don’t reduce memory or bus latency
  - May make misses come out faster
  - At some point memory bandwidth may become a **bottleneck**
  - Further increases in clock speed won’t help at all

<table>
<thead>
<tr>
<th>Parameter</th>
<th>1GHz</th>
<th>L2</th>
<th>L3</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>2ns</td>
<td>10ns</td>
<td>30ns</td>
<td>100ns</td>
</tr>
<tr>
<td>Capacity</td>
<td>8–64KB</td>
<td>128KB–256KB</td>
<td>1–9MB</td>
<td>64MB–64GB</td>
</tr>
<tr>
<td>Block size</td>
<td>16–32B</td>
<td>128–256KB</td>
<td>256B</td>
<td>4KB+</td>
</tr>
<tr>
<td>Associativity</td>
<td>4–4</td>
<td>4–16</td>
<td>16</td>
<td>full</td>
</tr>
<tr>
<td>Replacement</td>
<td>NRU</td>
<td>NRU</td>
<td>NRU</td>
<td>working set</td>
</tr>
</tbody>
</table>

- Prefetching?
  - Maybe
  - Possibly
  - Other

Increasing Memory Bandwidth

- Memory bandwidth can already bottleneck a single CPU
- What about multi-core?
- Higher frequency memory bus, DDR
  - Processors are doing this
- Wider processor-memory interface
  - Multiple DIMMs
  - Can get expensive: only want bandwidth, pay for capacity too
  - Multiple on-chip memory “channels”
  - Processors are doing this too

Main Memory As A Cache

- How would you internally organize main memory
  - L1 is outrageously long, reduce “L1” at all costs
  - Full associativity: isn’t that difficult to implement?
  - Yes ... in hardware, main memory is “software-managed”
Summary

- \[ t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} \times t_{\text{miss}} \]

- Memory hierarchy:
  - Capacity: smaller \( t_{\text{hit}} \) \( \rightarrow \) bigger, low \( \%_{\text{miss}} \)
  - Technology: expensive \( \rightarrow \) cheaper

- 10/90 rule, temporal/spatial locality
- SRAM \( \rightarrow \) DRAM \( \rightarrow \) Disk: reasonable total cost

- Organizing a memory component
  - ABC, write policies
  - 3C miss model: how to eliminate misses?
  - What about bandwidth?