ESE 345 Computer Architecture

Memory Hierarchy and Cache Design

“a memory cache” image created by SDXL text-to-image AI generative model 2023
Levels of the Memory Hierarchy

**CPU Registers**
- 100s Bytes
- <10s ns

**Cache**
- K Bytes
- 10-100 ns
- 1-0.1 cents/bit

**Main Memory**
- M Bytes
- 200ns- 500ns
- $.0001-.0001 cents/bit

**Disk**
- G Bytes, 10 ms (10,000,000 ns)
- 10^-5 - 10^-6 cents/bit

**Tape**
- infinite sec-min
- 10^-8

---

**Upper Level**
- **Staging**
- prog./compiler
- 1-8 bytes

**Xfer Unit**
- cache cntl
- 8-128 bytes

---

**Lower Level**
- **Cache design**
- OS
- 512-4K bytes

- user/operator
- Mbytes

- faster
- Larger
Management of Memory Hierarchy

- Small/fast storage, e.g., registers
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    - but hardware might do things behind software’s back, e.g., stack management, register renaming

- Larger/slower storage, e.g., main memory
  - Address usually computed from values in register
  - Generally implemented as a hardware-managed cache hierarchy
    - hardware decides what is kept in fast memory
    - but software may provide “hints”, e.g., don’t cache or prefetch
Memory Hierarchy: Terminology

- **Hit**: data appears in some block in the upper level (example: Block X)
  - **Hit Rate**: the fraction of memory access found in the upper level
  - **Hit Time**: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieve from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty

![Diagram of Memory Hierarchy](image)

Cache design
Real Memory Reference Patterns

Principle of Locality

- Programs access a small proportion of their address space at any time
  - Temporal locality
    - Items accessed recently are likely to be accessed again soon
    - E.g., instructions in a loop, induction variables
  - Spatial locality
    - Items near those accessed recently are likely to be accessed soon
    - E.g., sequential instruction access, array data
Memory Reference Patterns

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU
Caches

Caches exploit both types of predictability:

- Exploit temporal locality by remembering the contents of recently accessed locations.
- Exploit spatial locality by fetching blocks of data around recently accessed locations.
Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given accesses $X_1, \ldots, X_{n-1}, X_n$

<table>
<thead>
<tr>
<th>$X_4$</th>
<th>$X_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>$X_1$</td>
</tr>
<tr>
<td>$X_{n-2}$</td>
<td>$X_{n-2}$</td>
</tr>
<tr>
<td>$X_{n-1}$</td>
<td>$X_{n-1}$</td>
</tr>
<tr>
<td>$X_2$</td>
<td>$X_2$</td>
</tr>
<tr>
<td>$X_3$</td>
<td>$X_3$</td>
</tr>
</tbody>
</table>

- How do we know if the data is present?
- Where do we look?

  a. Before the reference to $X_n$
  b. After the reference to $X_n$
Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits of the block address
  - Called the **tag**

- What if there is no data in a location?
  - **Valid bit**: 1 = present, 0 = not present
  - Initially 0
Block Size and Spatial Locality

Block is unit of transfer between the cache and memory

- **Tag**
- **Word0** **Word1** **Word2** **Word3**

4 word (16 Byte) block

**Split CPU address**

- **block address**
- **Block offset**

- 32-b bits
- \( b \) bits \( = \log_2 \) (Block size in bytes)

- \( 2^b = \) block size *a.k.a.* line size (in bytes)

Larger block size has distinct hardware advantages
- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

**What are the disadvantages of increasing block size?**

Fewer blocks => more conflicts. Can waste bandwidth.
Inside a Cache (first look)

- Processor
  - Word Address
  - Data word
- CACHE
- Block Address
- Main Memory
  - Data block

**Address Tag of the block**
- Here a full block address (BA) is used for tag but we will need only high-order BA bits for the tag (later)

**Cache Line**
- Copy of main memory location 0x100 (block offset=0)
- Copy of main memory location 0x101 (block offset =1)

**Cache design**
4 Questions for Memory Hierarchy

■ Q1: Where can a block be placed in the upper level? *(Block placement)*

■ Q2: How is a block found if it is in the upper level? *(Block identification)*

■ Q3: Which block should be replaced on a miss? *(Block replacement)*

■ Q4: What happens on a write? *(Write strategy)*
Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

- #Blocks is a power of 2
- Use low-order address bits
Example: Address Subdivision

- 1KB Direct Mapped (DM) Cache
- 16 bytes/block (4 bits per block offset)
- $1,024/16=64$ blocks (6 bits per cache index)
  - To what block number does byte address 1204 map?
- Block offset $= 1204 \mod 16 = 4$
- Block address number $= \left\lfloor \frac{1204}{16} \right\rfloor = 75$
- Cache Index $= 75 \mod 64 = 11$

```
0000000.....00001  001011  0100
```

Cache design
Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access
Direct-Mapped Cache

- Tag
- Index
- Block Offset

- $t$ 
- $k$ 
- $b$

- $V$, Tag, Data Block

- $2^k$ lines

- HIT

- 32 bits

Cache design
Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%
Example: Intrinsity FastMATH

DM Cache size = 16 KBytes = \(2^{14}\) bytes

Number of blocks/cache = \(2^{14} / 2^6 = 2^8\)

Block size = 16 words = 64 = \(2^6\) bytes

Block offset \((b) = \log_2(64) = 6\) bits \([5:0]\) field

Address (showing bit positions)

for lw instructions the last 2 bits

(byte offset in a word) are 0s

Tag size = Address size – cache index – block offset =

\[32 - 8 - 6 = 18\] bits

DM Cache size = 16 KBytes = \(2^{14}\) bytes

# of blocks/cache = \(2^{14} / 2^6 = 2^8\)

Tag size = Address size – cache index – block offset =

\[32 - 8 - 6 = 18\] bits

for lw instructions the last 2 bits

(byte offset in a word) are 0s

Block size = 16 words = 64 = \(2^6\) bytes

Block offset \((b) = \log_2(64) = 6\) bits \([5:0]\) field

Cache design
2-Way Set-Associative (SA) Cache

N = 2^n - way SA way cache
(n=1) = 2^1 : 2-way SA cache
(n=2) = 2^2 : 4-way SA cache

Cache size= 2^c bytes
# of blocks/cache= 2^c / 2^b
= 2^{c-b}

k = Set index size =
log_2 (2^c \cdot b / 2^n) = c - b - n

t = Tag size =
Address size – k – b

Stony Brook University
Fully Associative Cache
Replacement Policy:
Which Block to Replace on Miss

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity
- Pseudo-LRU
Binary Tree Pseudo-LRU (Tree-PLRU):

- Rather than true LRU, use a binary tree
- Each node records which half is older/newer
- To find a pseudo-LRU element, traverse the tree according to the values of the node states.
- To update the tree with an access to an item N, traverse the tree to find N and, during the traversal, set the node states to denote the direction that is opposite to the direction taken.
  - For the nodes that are already in the right state, there will be no change
- Used in Intel 486 and Apple PowerPC-based computers
Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim (can be pseudo-random, or rotating, or fixed left-to-right)

- Simple, similar to virtual memory clock algorithm
- Relies on “randomizing” evictions rather than strict LRU order
- Used by Intel Itanium, Sparc T2
Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence (those are block (not byte!) addresses): 0, 8, 0, 6 (0b110), 8

- Direct mapped (DM)
  - Cache index size is \( \log_2 4 = 2 \) (i.e., use 2 least significant bits of the block address for cache index)

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[8]</td>
</tr>
</tbody>
</table>
Associativity Example

- **2-way set associative (2WSA)**
  - Set index size is \( \log_2 4 \)/2=1 (i.e., use a least significant bit of the block address for set index)

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[6]</td>
</tr>
</tbody>
</table>

- **Fully associative**

<table>
<thead>
<tr>
<th>Block address</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>miss</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[8]</td>
</tr>
</tbody>
</table>
How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
CPU-Cache Interaction

(5-stage pipeline)

Cache design

Stall entire CPU on data cache miss
Cache Measures

- **Hit rate**: fraction found in that level
  - So high that usually talk about *Miss rate*
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory

- **Average memory access time**
  - \( \text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty} \)
  - (ns or clocks)

- **Miss penalty**: time to replace a block from lower level, including time to replace in CPU
  - Does not include hit time!

- **access time**: time to lower level
  - \( \text{access time} = f(\text{latency to lower level}) \)

- **transfer time**: time to transfer block
  - \( \text{transfer time} = f(\text{BW between upper \& lower levels}) \)
Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

\[
\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}
\]

Contribution to average CPI due to cache misses
Cache Performance Example

- **Given**
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
    - includes hit times for both caches!
  - Load & stores are 36% of instructions

- **Miss cycles per instruction**
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$

- **Actual CPI = 2 + 2 + 1.44 = 5.44**
  - Ideal CPU is $5.44/2 = 2.72$ times faster
Average Memory Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, l-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2 cycles (2 ns)
    - 2 cycles per instruction
- AMAT should never be used instead of miss penalty when calculating processor CPI!
Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance
Improving Cache Performance

Average memory access time =
Hit time + Miss rate x Miss penalty

To improve performance:
• reduce the hit time
• reduce the miss rate
• reduce the miss penalty

What is best cache design for 5-stage pipeline?

Biggest cache that doesn’t increase hit time past 1 cycle
(approx 8-32KB in modern technology)
[design issues more complex with deeper pipelines and/or
out-of-order superscalar processors]
Causes for Cache Misses

- **Compulsory**: first-reference to a block *a.k.a.* cold start misses
  - misses that would occur even with infinite cache

- **Capacity**: cache is too small to hold all data needed by the program
  - misses that would occur even under perfect replacement policy

- **Conflict**: misses that occur because of collisions due to block-placement strategy
  - misses that would not occur with full associativity
5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
  5. Smaller size & less associativity for L1 cache
Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks $\Rightarrow$ fewer of them
    - More competition $\Rightarrow$ increased miss rate
  - Larger blocks $\Rightarrow$ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Critical-word-first can help
Effect of Cache Parameters on Performance

• Larger cache size
  + reduces capacity and conflict misses
  - hit time will increase

• Higher associativity
  + reduces conflict misses
  - may increase hit time

• Larger block size
  + reduces compulsory and capacity misses
  - increases conflict misses and miss penalty
Write Policy Choices

- **Cache hit:**
  - *write through:* write both cache & memory
    - Generally higher traffic but simpler pipeline & cache design
  - *write back:* write cache only, memory is written only when the entry is evicted
    - A dirty bit per block further reduces write-back traffic
    - Must handle 0, 1, or 2 accesses to memory for each load/store

- **Cache miss:**
  - *no write allocate:* only write to main memory
  - *write allocate (aka fetch on write):* fetch into cache

- **Common combinations:**
  - write through and no write allocate
  - write back with write allocate
Write-Through

- On data-write hit, update the block in cache and also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1\times100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full
Write-Back

- Alternative: On data-write hit, just update the block in cache
  - But then cache and memory would be inconsistent
  - Keep track of whether each block is dirty with a dirty bit
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first
Write Buffer to Reduce Read Miss Penalty

Processor is not stalled on writes, and read misses can go ahead of write to main memory

**Problem:** Are Read After Write (RAW) hazards an issue for a write buffer? Yes!
The write buffer may hold updated value of location needed by a read miss.

**Simple scheme:** on a read miss, wait for the write buffer to go empty

**Faster scheme:** Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer
Write Performance

Cache design

- Tag
- Index
- Block Offset

- Data

- HIT

Data Word or Byte

2^k lines

WE
Reducing Write Hit Time

**Problem:** Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

**Solutions:**
- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- Fully-associative (CAM Tag) caches: Word line only enabled if hit
- **Pipelined writes:** Hold write data for store in single buffer ahead of cache, write cache data during next store’s tag check
  - Forward delayed data to reads that have the same address
Pipelining Cache Writes w/ Forwarding
(always writing data during the second cycle)

Load/Store Address and Store Data From CPU

Tag | Index
---|---
sw $4, Addr
lw $5, Addr

Delayed Write Addr.

Delayed Write Data

Tags

Hit?

Data

Write Addr
Read Addr

Data from a store hit written into data portion of cache during tag access of subsequent store

Cache design

Dual-port cache (read & write in one cycle) is required

sw $4, Addr
lw $5, Addr

When writing, forward delayed write data, if the read occurring in the same cycle has the same address as write
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block
Multilevel Caches

Problem: A memory cannot be large and fast
Solution: Increasing sizes of cache at each level

Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions
Multilevel Caches

- **Primary cache**
  - Small, but fast
  - Focus on minimal hit time

- **Level-2 cache services misses from primary cache**
  - Larger, slower, but still faster than main memory
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact

- **Results**
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size
  - Main memory services L-2 cache misses
  - Some high-end systems include L-3 cache
Multilevel Cache Example

Given

- CPU base CPI = 1, clock rate = 4GHz
- Miss rate/instruction = 2%
- Main memory access time = 100ns

With just primary cache

- Miss penalty = 100ns/0.25ns = 400 cycles
- Effective CPI = 1 + 0.02 × 400 = 9
Example (cont.)

- Now add L2 cache
  - Access time = 5ns
  - Global miss rate/instruction to main memory = 0.5%
    - =>2% -0.5% = 1.5% out of 2% hit in L2 cache
- Primary miss with L2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L2 miss
  - Extra penalty = 400 cycles
- CPI = 1 + (0.02-0.005) × 20 + 0.005 × (20+400) = 1 + 0.3 + 2.1 = 3.4
- Performance ratio = 9/3.4 = 2.6
Inclusion Policy

- **Inclusive multilevel cache:**
  - Inner cache can only hold lines also present in outer cache
  - External coherence snoop access need only check outer cache

- **Exclusive multilevel caches:**
  - Inner cache may hold lines not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache
Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation
Power 7 On-Chip Caches [IBM 2009]

- 32KB L1 I$/core
- 32KB L1 D$/core
- 3-cycle latency

- 256KB Unified L2$/core
- 8-cycle latency

- 32MB Unified Shared L3$
- Embedded DRAM (eDRAM)
- 25-cycle latency to local slice

Cache design
IBM z196 Mainframe Caches 2010

- 96 cores (4 cores/chip, 24 chips/system)
  - Out-of-order, 3-way superscalar @ 5.2GHz
- L1: 64KB I-$/core + 128KB D-$/core
- L2: 1.5MB private/core (144MB total)
- L3: 24MB shared/chip (eDRAM) (576MB total)
- L4: 768MB shared/system (eDRAM)
Intel i7 (2012)

- All three caches use write-back
- A block size is 64 bytes for all caches
- L1 and L2 caches are separate for each core
- L3 cache is shared among the cores on a chip
AMD Ryzen Threadripper Pro 5995WX (2022)

- Clock rate (GHz):
  - Base: 2.7
  - Boost: 4.5
- Cores: 64
- L1 cache: 64 KB (32 KB data + 32 KB instruction) per core
- L2 cache: 512 KB per core
- L3 cache (total): 256 MB (shared by 64 cores)
Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- What types of misses does prefetching affect?
- Have a better idea why to use lw $0, 100($5)?
Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)