Introduction

• Computer performance depends on:
  – Processor performance
  – Memory system performance

Topics:
1. Cache memory
2. DRAM
3. Virtual memory
**Introduction**

- Up until now, assumed memory could be accessed in 1 clock cycle

![Graph showing the performance of processors and memory over years.](image)
Memory hierarchy

Ideal memory
Access time of SRAM
Capacity and cost/GB of disk
SRAM and DRAM cells

Classical 6T SRAM cell

Classical 1T+1C DRAM cell

• SRAM cell takes more silicon area than 1T DRAM cell → more memory capacity per unit area for DRAM → cheaper memory per bit.

• SRAM holds its charge, more stable and much faster than DRAM.

• DRAM requires constant refreshing due to leakage.
Principle of locality

• Exploit locality to make memory accesses fast

• **Temporal Locality:**
  – Locality in time (e.g., if looked at a Web page recently, likely to look at it again soon)
  – If data used recently, likely to use it again soon
  – **How to exploit:** keep recently accessed data in higher levels of memory hierarchy

• **Spatial Locality:**
  – Locality in space (e.g., if read one page of book recently, likely to read nearby pages soon)
  – If data used recently, likely to use nearby data soon
  – **How to exploit:** when access data, bring nearby data into higher levels of memory hierarchy too
Memory hierarchy levels

- **Block (aka line):** unit of copying
  - May be multiple words
- **If accessed data is present in upper level**
  - **Hit:** access satisfied by upper level
    - Hit ratio: hits/accesses
- **If accessed data is absent**
  - **Miss:** block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      - Miss ratio = 1 – Hit ratio
  - Then accessed data supplied from upper level
Example

• A program has 2,000 load and store instructions
• 1,250 of these data values found in cache; the rest are supplied by the main memory
• What are the hit and miss rates for the cache?
  Hit Rate $= \frac{1250}{2000} = 0.625$
  Miss Rate $= \frac{750}{2000} = 0.375 = 1 - \text{Hit Rate}$
• $t_{\text{cache}} = 1$ cycle, $t_{\text{MM}} = 100$ cycles
• What is the average memory access time (AMAT) of the program from Example 1?
  AMAT $= t_{\text{cache}} + MR_{\text{cache}}(t_{\text{MM}})$
  $= [1 + 0.375(100)]$ cycles
  $= 38.5$ cycles
Cache memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
  - Fast (typically ~ 1 cycle access time)
  - Made out of SRAM cells
- Given accesses $X_1, \ldots, X_{n-1}, X_n$

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

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

- What data is held in the cache?
- How is data found?
- What data is replaced?

a. Before the reference to $X_n$
b. After the reference to $X_r$
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Set Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>11...11111100</td>
<td>7 (111)</td>
</tr>
<tr>
<td>11...11111000</td>
<td>6 (110)</td>
</tr>
<tr>
<td>11...11110100</td>
<td>5 (101)</td>
</tr>
<tr>
<td>11...11101100</td>
<td>4 (100)</td>
</tr>
<tr>
<td>11...11101000</td>
<td>3 (011)</td>
</tr>
<tr>
<td>11...11100100</td>
<td>2 (010)</td>
</tr>
<tr>
<td>11...11100000</td>
<td>1 (001)</td>
</tr>
<tr>
<td>00...00100100</td>
<td>0 (000)</td>
</tr>
</tbody>
</table>

2³ Word Cache

2⁰ Word Main Memory
Tag 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
  - Called the tag

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0
Cache hardware design

Memory Address

8-entry x (1+27+32)-bit SRAM

Hit
Example

• How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?
Direct cache performance example

Miss Rate = 3/15
= 20%

Temporal Locality
Compulsory Misses

```
MOV    R0, #0
MOV    R1, #5

loop:  CMP    R1, R0
       BEQ    DONE
       LDR    R2, [R0, 0x4]
       LDR    R3, [R0, 0xC]
       LDR    R4, [R0, 0x8]
       ADD    R1, R1, #-1
       B      LOOP

done:
```
Direct cache map: conflict

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...01</td>
<td>001</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
MOV R0, #0
MOV R1, #5
loop: CMP R1, R0
      BEQ DONE
      LDR R2, [R0, 0x4]
      LDR R3, [R0, 0x24]
      ADD R1, R1, #-1
      B LOOP
done:
```

Miss Rate = 10/10
            = 100%

Conflict Misses
Impact of larger block sizes

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
  - Block address = $\left\lfloor \frac{1200}{16} \right\rfloor = 75$
  - Block number = 75 modulo 64 = 11

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 bits</td>
<td>6 bits</td>
<td>4 bits</td>
</tr>
</tbody>
</table>
```
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
Two-way set associate cache

Memory Address

Tag  Set  Offset

00

Way 1

V  Tag  Data

Way 0

V  Tag  Data

Hit 1  Hit 0

Hit

Hit 1

Data

Hit

Data
Two-way set associative cache example

```
MOV R0, #0
MOV R1, #5

loop:  CMP R1, R0
       BEQ DONE
       LDR R2, [R0, 0x4]
       LDR R3, [R0, 0x24]
       ADD R1, R1, #-1
       B LOOP

done:
```

Miss Rate $= \frac{2}{10} = 20\%$

Associativity reduces conflict misses

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3
Set 2
Set 1
Set 0
4-way associative
Fully associative cache

No conflict misses
Expensive to build
Spectrum of associativity

- For a cache with 8 entries

One-way set associative (direct mapped)

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eight-way set associative (fully associative)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Replacement choices for associative memory

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- What is the optimal entry to remove?
- Random
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
LRU Replacement example

MOV R0, #0  
LDR R1, [R0, 0x04]  
LDR R2, [R0, 0x24]  
LDR R3, [R0, 0x54]

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...000</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 1 | Way 0

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

---

(a)

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...101</td>
<td>mem[0x00...54]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 1 | Way 0

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

(b)
Sources of misses

- **Compulsory misses** (aka cold start misses)
  - First access to a block

- **Capacity misses**
  - Due to finite cache size
  - A replaced block is later accessed again

- **Conflict misses** (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size
## Impact of cache design on miss rate

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effect on miss rate</th>
<th>Negative performance effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase cache size</td>
<td>Decrease capacity misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase associativity</td>
<td>Decrease conflict misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase block size</td>
<td>Decrease compulsory misses</td>
<td>Increases miss penalty.</td>
</tr>
</tbody>
</table>
Handling cache writes

• *How to handle cache writes?*
  – Write through
  – Write back

• *How to handle write misses?*
Write through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent

- Write through: 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 = $1 + 0.1 \times 100 = 11$

- 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
  - Keep track of whether each block is dirty

- 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 allocation on write misses

- 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
Multi-level caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache
Multi-level 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 = \(1 + 0.02 \times 400 = 9\)
Multi-level cache example

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = \(5\text{ns} / 0.25\text{ns} = 20\) cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- \(CPI = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4\)
- Performance ratio = \(9/3.4 = 2.6\)
Cache summary

- Cache is on-chip memory built using 6T SRAM cells
- Cache organization:
  - Direct
  - Fully associative
  - N-way set associative
- Cache write techniques
- Cache block replacement techniques (for associative)
- Multi-level cache
DRAM is usually a shared resource among multiple processors, GPU and I/O devices → a controller is needed to coordinate the access.
DRAM organization (1-bit array)

- Reads are destructive; content must be restored after reading
- Capacitors are leaky so they must be periodically refreshed
- DRAM controller must each refresh each row (by reading and re-writing it) every 64 ms or less
- Refreshing contributes to the slow access of DRAMs
Multiple bit DRAM organization

In a x4 DRAM part, four arrays each read 1 data bit in unison, and the part sends out 4 bits of data each time the memory controller makes a column read request.
DRAM operation: bus transmission
DRAM operation: row access
DRAM operation: column access
DRAM operation: data transfer
DRAM operation: bus transmission
DRAM memory bus organization

Interleaved memory organization
Simple asynchronous memory reading

Lower pin count (cost) by using same pins for row and column addresses
To increase throughput, SDRAM outputs the data of sequential columns of the same row into the data bus (BURST mode). Now needs a clock.

Matches well with cache blocks with multiple bytes.
Double Data Rate (DDR) SDRAM

Transfer data on positive and negative edges of the clock
DRAM connections

In PC

In smart phones/tablets

DRAM die
Processor die
A PoP implementation

Lower density I/O BGA ball count (0.85mm pitch and more) connects memory die to landing pads on the top of the bottom package. Memory die typically require less I/O than logic die.

Stacked memory die

Gold or copper wirebonds

Laminate substrate (essentially a mini PCB)

High density I/O BGA ball count (0.5mm pitch or less) connects ASIC logic die to motherboard

Single or multiple ASIC logic die

48
Virtual memory

• Each program uses *virtual addresses*
  – Entire virtual address space stored on a hard disk.
  – Subset of virtual address data in DRAM
  – CPU translates virtual addresses into *physical addresses*
  – Data not in DRAM is fetched from the hard disk

• Each program has its own virtual to physical mapping
  – Two programs can use the same virtual address for different data
  – Programs don’t need to be aware that others are running
  – One program (or virus) can’t corrupt the memory used by another
  – This is called *memory protection*
Virtual and physical addresses

Most accesses hit in physical memory
But programs have the large capacity of virtual memory
Virtual memory definitions

• **Page size**: amount of memory transferred from hard disk to DRAM at once

• **Address translation**: determining the physical address from the virtual address

• **Page table**: lookup table used to translate virtual addresses to physical addresses
Address translation
Virtual memory example

**System:**

Virtual memory size: $2 \text{ GB} = 2^{31} \text{ bytes}$
Physical memory size: $128 \text{ MB} = 2^{27} \text{ bytes}$
Page size: $4 \text{ KB} = 2^{12} \text{ bytes}$

**Organization:**

Virtual address: 31 bits
Physical address: 27 bits
Page offset: 12 bits

# Virtual pages = $2^{31}/2^{12} = 2^{19}$ (VPN = 19 bits)
# Physical pages = $2^{27}/2^{12} = 2^{15}$ (PPN = 15 bits)
Virtual memory example

What is the physical address of virtual address **0x247C**?
- VPN = 0x2
- VPN 0x2 maps to PPN 0x7FFF
- The lower 12 bits (page offset) is the same for virtual and physical addresses (**0x47C**)
- Physical address = 0x7FFF47C

19-bit virtual page numbers - 15-bit physical page numbers
Using page tables to translate addresses

• Page Table stores placement information
  – Array of page table entries, indexed by virtual page number
  – Page table register in CPU points to page table in physical memory

• If page is present in memory
  – PTE stores the physical page number
  – Plus other status bits (referenced, dirty, …)

• If page is not present
  – PTE can refer to location in swap space on disk
Translation using page table
Mapping pages to storage
What is the physical address of virtual address 0x5F20?

- VPN = 5
- Entry 5 in page table indicates VPN 5 is in physical page 1
- Physical address is 0x1F20
Page table example 2

What is the physical address of virtual address 0x73E0?

- VPN = 7
- Entry 7 in page table is invalid, so the page is not in physical memory
- The virtual page must be *swapped* into physical memory from disk
Replacement and writes

- **Replacement**: To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- **Writes**: Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical; use write-back
  - Dirty bit in PTE set when page is written
  - Write a physical page and reloading a different virtual page is called **swapping**
Page table challenges

• Page table is large
  – usually located in physical memory

• Each load/store requires two main memory accesses:
  – one for translation (page table read)
  – one to access data (after translation)

• Cuts memory performance in half
  – Unless we get clever...
Fast translation using TLB

- Use a *translation lookaside buffer* (TLB)
  - Small cache of most recent translations
  - Reduces number of memory accesses required for most loads/stores from two to one

- TLB
  - Small: accessed in < 1 cycle
  - Typically 16 - 512 entries
  - Fully associative
  - > 99 % hit rates typical
  - Reduces # of memory accesses for most loads and stores from 2 to 1
Example: Two-entry TLB

Virtual Address

Virtual Page Number
0x00002

Page Offset
47C

Entry 1

Virtual Page Number
0x7FFFD

Physical Page Number
0x0000

Entry 0

Virtual Page Number
0x00002

Physical Page Number
0x7FFF

Physical Address
0x7FFF
47C

Hit
TLB misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler

- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction
Memory protection

- Multiple programs (*processes*) run at once
- Each process has its own page table – special register holds the starting address of the table
- Each process can use entire virtual address space without worrying about where other programs are
- A process can only access physical pages mapped in its page table – can’t overwrite memory from another process
- **Modifications to TLB:**
  - Invalidate the TLB every time there is a context switch
  - Or augment TLB with process id
1. TLB/Cache: physical tag, physical index

- **virtual address:** 20 bits
- **page offset:** 12 bits
- **4 KB page size**

Translation in TLB:

- 1 cycle

- **20 bits**

- **12 bits**

- **18 bits**

**16 KB cache size**

- 4 bytes / block
- Direct mapped

- **4096 blocks**

hit -> data

- 1 cycle
2. TLB/Cache: virtual tag, virtual index

- TLB is only accessed in case of miss
- Complications due to aliasing: Different virtual addresses for shared physical address
3. TLB/Cache: physical tag, virtual index

- Virtual address: 20 bits
- Page offset: 12 bits
- 4 KB page size
- Direct mapped cache
- 1024 blocks
- 4 KB cache size
- 4 bytes / block
- First design attempt
- 1 cycle
- Tag: 20 bits, 10 bits, 2 bits
- Hit: 1 cycle
- Data: 1 cycle
3. TLB/Cache: physical tag, virtual index

- TLB can now be accessed in parallel with cache
- Constrain cache design: *fundamental* cache size (# sets * block size = page size).
- Only way to increase cache size is by increasing associativity
Virtual memory summary

- Virtual memory increases **capacity and essential for multitasking OS**
- A subset of virtual pages are located in physical memory
- A **page table** maps virtual pages to physical pages – this is called address translation
- A **TLB** speeds up address translation
- Using different page tables for different programs provides **memory protection**
Memory subsystem summary

• Fast memories are small, large memories are slow
  – We really want fast, large memories
  – Caching gives this illusion

• Principle of locality
  – Programs use a small part of their memory space frequently

• Memory hierarchy
  – L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk

• Virtual memory