ECE 251: Computer Architecture

Week 14: Memory Hierarchies (Part 2)

Prof Rob Marano

Spring 2026

Week 13 Retrospective: The Basics of Caches

Before concluding the memory system, let's review the foundations:

- **Principle of Locality:** Caches work due to *Temporal Locality* (re-accessing data) and *Spatial Locality* (accessing nearby data).
- **Memory Technologies:** Fast, expensive SRAM (caches) vs slower, denser DRAM (main memory).
- **Cache Mapping Topologies:**
  - **Direct-Mapped:** Fast, simple, prone to conflict misses.
  - **Fully Associative:** Eliminates conflicts, requires massive hardware comparators.
  - **Set-Associative:** The N-way compromise to balance conflicts and hardware costs.
- **AMAT:** Average Memory Access Time = `Hit Time + (Miss Rate` $\times$ `Miss Penalty)`.

Agenda

1. **Cache Performance:** The Memory Wall & Multi-Level Caches
2. **Hardware Realization:** SystemVerilog (SV) Cache Controllers
3. **Mathematical Proof:** Quantifying the Upgrade
4. **Dependable Memory:** Soft Errors & ECC (Parity/Hamming)
5. **Virtual Memory:** Pages, Faults, and the TLB
6. **Interactive Practice Problems**

5.4 The Memory Hierarchy Pyramid

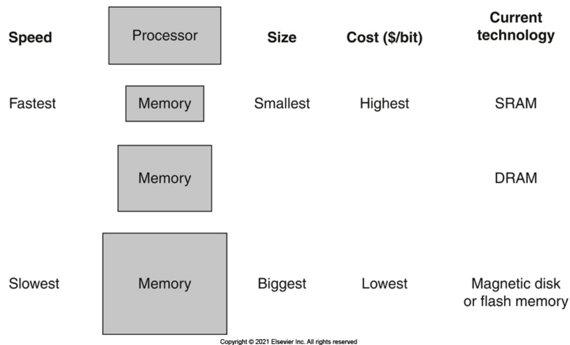


Figure 5.1: The basic structure of a memory hierarchy.

5.4 The Memory Wall & Hiding Penalties

- **The Memory Wall (Harris):** Processor execution speeds have scaled exponentially (Moore's Law), but DRAM access times have scaled linearly. A modern CPU accessing DRAM loses *hundreds* of cycles. Multi-core systems exacerbate this bus bandwidth bottleneck.
- **Hiding Miss Penalties (COaD):** Modern superscalar processors "hide" penalties.
  - *Out-of-order execution:* Looks ahead and executes independent instructions while fetching from DRAM.
  - *Non-blocking caches:* Continue to supply hits to the CPU even while processing a miss for a different address.

Misses are generally categorized into three types:

- **1. Compulsory Misses:** The first time a block is accessed. Unavoidable, but larger block sizes help pull in more data at once (Spatial Locality).
- **2. Capacity Misses:** The cache isn't big enough to hold the active working set of the program. Solved by increasing cache size (which increases Hit Time and cost).
- **3. Conflict Misses:** Multiple blocks compete for the same cache index. Solved by increasing associativity (moving from Direct-Mapped to N-Way Set Associative).

5.4 Reducing Miss Penalty: Multi-Level Caches

If Main Memory is 100 cycles away, a cache miss is catastrophic. We add more levels!

- **L1 Cache:** Attached directly to the processor core. Extremely fast (1-2 cycles), but very small (e.g., 32KB). Optimized for the lowest possible **Hit Time**.
- **L2 Cache:** Larger and slightly slower (e.g., 10-20 cycles, 256KB-1MB). Captures the misses from L1 before they hit DRAM. Optimized for the lowest possible **Miss Rate**.
- *Modern processors often have L3 caches shared across multiple cores.*

5.4 The Multi-Level AMAT Formula

With two levels of cache, the AMAT formula expands drastically:

$$\text{AMAT} = \text{L1 Hit Time} + (\text{L1 Miss Rate} \times \text{L1 Miss Penalty})$$

where

$$\text{L1 Miss Penalty} = \text{L2 Hit Time} + (\text{L2 Miss Rate} \times \text{L2 Miss Penalty})$$

SystemVerilog (SV) Review: Direct-Mapped Hit Logic

Review from Week 13: Slicing the physical address for a direct-mapped cache.

```
module cache_controller_direct (
    input  logic [31:0] paddr,
    input  logic [15:0] cache_tags [0:4095],
    input  logic        valid_bits [0:4095],
    output logic        cache_hit
);
    // 1. Physically slice the 32-bit address
    logic [11:0] index = paddr[15:4];
    logic [15:0] tag   = paddr[31:16];

    // 2. Lookup the stored tag and valid bit
    logic [15:0] stored_tag = cache_tags[index];
    logic        is_valid   = valid_bits[index];

    // 3. Combinational Hit Logic
    assign cache_hit = is_valid & (tag == stored_tag);
endmodule
```

SV Upgrade: 2-Way Set Associative

Upgrading the Architecture to eliminate Conflict Misses.

```
module cache_controller_2way (
    input  logic [31:0] paddr,
    input  logic [16:0] way0_tags [0:2047], input logic way0_valid [0:2047],
    input  logic [16:0] way1_tags [0:2047], input logic way1_valid [0:2047],
    output logic        cache_hit, hit_way
);
    // Note: 1 fewer index bit, 1 larger tag bit!
    logic [10:0] index = paddr[14:4];
    logic [16:0] tag   = paddr[31:15];

    // Parallel Hardware Comparators
    logic hit_way0 = way0_valid[index] & (tag == way0_tags[index]);
    logic hit_way1 = way1_valid[index] & (tag == way1_tags[index]);

    // Global Hit Logic (OR gate)
    assign cache_hit = hit_way0 | hit_way1;
    assign hit_way = hit_way1; // Controls a downstream Data Mux
endmodule
```

SV Upgrade: Multi-Level L1/L2 Handshake

The L1 Cache acts like the "CPU" to the L2 Cache.

```
module multi_level_cache_system (
    input  logic [31:0] paddr,
    input  logic        cpu_read_req,
    output logic [31:0] cpu_read_data,
    output logic        cpu_stall    // Freezes the pipeline!
);
    // ... cache instantiations omitted ...

    // The Critical CPU Stall Logic
    // The CPU is forced to stall if L1 misses AND L2 hasn't resolved yet.
    // L2 resolves data either by hitting instantly, or waiting 100 cycles for DRAM.
    assign cpu_stall = cpu_read_req & l1_miss & ~(l2_hit | dram_data_ready);

    // Data Routing Multiplexer
    assign cpu_read_data = (l1_hit) ? l1_data : l2_data;

endmodule
```

The Iron Law & Effective CPI

Why go through the immense hardware complexity of building Multi-Level Caches? Let's prove its worth using the **Iron Law of Performance**.

The Baseline CPU:

- Base CPI $= 1.0$
- Memory instructions (`lw`/`sw`) $= 30\%$ of all instructions.
- L1 Miss Rate $= 5\%$
- DRAM Miss Penalty $= 100$ cycles

Scenario A: L1 Cache Only

When L1 misses, we immediately stall for 100 cycles waiting for DRAM.

$$\text{Stalls} = \text{Mem Insts} \times \text{L1 Miss Rate} \times \text{DRAM Penalty}$$

$$\text{Stalls} = 0.30 \times 0.05 \times 100 = \mathbf{1.5} \text{ cycles}$$

$$\text{Effective CPI} = \text{Base CPI} + \text{Stalls}$$

$$\text{Effective CPI} = 1.0 + 1.5 = \mathbf{2.5}$$

The CPU is running 2.5x slower than its theoretical maximum!

Scenario B: Adding the L2 Cache

We add the L2 Cache from our RTL. L2 Hit Time = 10 cycles (new L1 penalty). Local L2 Miss Rate = 20%. We only hit DRAM if L2 misses.

$$\text{L1 Miss Penalty (Global)} = \text{L2 Hit Time} + (\text{L2 Miss Rate} \times \text{DRAM Penalty})$$

$$\text{L1 Miss Penalty} = 10 + (0.20 \times 100) = \mathbf{30}\text{ cycles}$$

$$\text{New Stalls} = 0.30 \times 0.05 \times 30 = \mathbf{0.45}\text{ cycles}$$

$$\text{Effective CPI} = 1.0 + 0.45 = \mathbf{1.45}$$

The Conclusion: Performance Speedup

- **Scenario A (L1 Only):** Effective CPI $= 2.5$
- **Scenario B (L1 + L2):** Effective CPI $= 1.45$

By adding the multi-level cache hardware, the execution time of the program **drops by 42%**.

This results in a processor that is **72% faster (1.72x speedup)** without changing the clock speed or the ISA!

5.5 The Vulnerability of DRAM

- As processors scale to smaller nanometer fabrication nodes, cosmic rays and alpha particles can physically flip bits in RAM, causing **Soft Errors**.
- **DRAM 1T1C Organization (Hamacher):** To maximize density, a single transistor and a microscopic capacitor represent an entire bit.
- These capacitors hold so few electrons that a stray cosmic ray striking the silicon can instantly discharge it, flipping a 1 to a 0.
- A truly dependable system must detect and recover from these microscopic physical phenomena.

5.5 Measures of Dependability

- **Reliability:** The measure of continuous service accomplishment.
  - Measured as **MTTF** (Mean Time To Failure).
- **Availability:** The measure of the fraction of time a service is functioning.
- $$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$
- *(MTTR = Mean Time To Repair)*

5.5 Error Detection and Correction (EDC)

- **Parity Bits:** Adding an extra bit to a word to ensure the total number of 1s is always even (or odd). Can detect single-bit errors, but cannot correct them.
- **Hamming Codes (SEC/DED) (COaD):** Strategically adding multiple parity bits (e.g., 8 check bits for 64 bits of data).
- Arranges data into intersecting logical circles (parity equations).
- If a bit flips, it breaks a unique combination of equations called the "syndrome", allowing hardware to pinpoint and correct the flipped bit (**Single Error Correction**).

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Copyright © 2021 Elsevier Inc. All rights reserved

Figure 5.23: Generating parity bits to pinpoint errors mathematically.

SV Realization: Even Parity Generator

Generating these protective codes in hardware is incredibly fast because it relies on simple parallel XOR gates. SystemVerilog provides a built-in unary reduction operator (ˆ).

```
module parity_generator (
    input  logic [7:0] data_in,
    output logic       even_parity_bit
);
    // The unary XOR reduction operator XORs all bits together.
    // If there is an odd number of 1s, it outputs 1.
    // If there is an even number of 1s, it outputs 0.

    assign even_parity_bit = ^data_in;

endmodule
```

SV Realization: Hamming Syndrome Decoder

To pinpoint a flipped bit, the memory controller calculates the syndrome using three overlapping parity equations executing perfectly in parallel.

```
module hamming_syndrome_decoder (
    input  logic [7:1] read_word, // 7-bit Hamming encoded word
    output logic [2:0] syndrome   // 3-bit pointer to the error
);
    // Synthesizes to three separate XOR trees.
    logic p1_check, p2_check, p4_check;

    assign p1_check = read_word[1]^read_word[3]^read_word[5]^read_word[7];
    assign p2_check = read_word[2]^read_word[3]^read_word[6]^read_word[7];
    assign p4_check = read_word[4]^read_word[5]^read_word[6]^read_word[7];

    // If syndrome == 101 (5), then bit 5 is corrupted.
    assign syndrome = {p4_check, p2_check, p1_check};
endmodule
```

5.6 Virtual Memory Concepts

Caches provide the illusion of *fast* memory. **Virtual Memory** provides the illusion of *infinite, isolated* memory. Together they create the **Memory Hierarchy**.

- **Pages:** Instead of "Cache Blocks", Virtual Memory divides data into "Pages" (typically 4KB).
- **Virtual vs. Physical Addresses:** Programs run using Virtual Addresses. The hardware translates these to Physical Addresses in DRAM.
- **Page Table:** A data structure in DRAM that maps Virtual Pages to Physical Pages. Provides strict isolation/security between running applications.

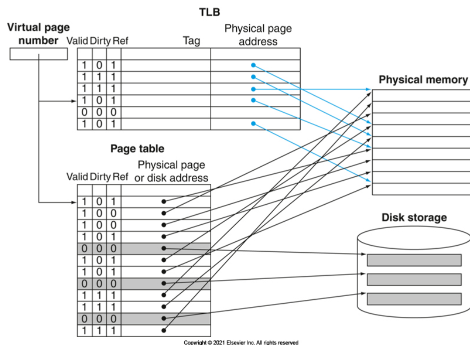
5.6 Page Faults

- Main Memory (DRAM) acts as a “cache” for the Magnetic Disk / SSD.
- **Page Faults:** A “Cache Miss” in Virtual Memory. The requested Virtual Page is not in DRAM.
- Because the miss penalty to Disk is measured in millions of cycles, the hardware cannot handle misses by stalling the pipeline.
- The CPU throws a hardware exception, context switches to the OS, and the OS fetches the page from Disk into DRAM.

5.6 The Translation Lookaside Buffer (TLB)

- Since the Page Table is stored in memory, every `lw`/`sw` would theoretically require *two* memory accesses (halving performance!).
- **Solution: The TLB**
- A tiny, incredibly fast Fully Associative cache dedicated entirely to storing recent Virtual-to-Physical address translations.
- **Hardware Integration:** The TLB and L1 Cache are queried *concurrently*. The TLB translates the upper Tag bits in parallel with the L1 SRAM retrieving the data via the Index.

5.6 Parallel Cache & TLB Access



Copyright © 2021 Elsevier Inc. All rights reserved

Figure 5.28: Concurrently accessing the TLB on every memory reference.

SV Realization: Fully Associative TLB Array

A TLB must compare the requested Virtual Page Number against *every single entry simultaneously*. A SystemVerilog `for` loop unrolls into a massive bank of parallel comparators!

```
module tlb_array #(parameter TLB_ENTRIES = 64) (
    input  logic [19:0] vpn_in,
    input  logic [19:0] tlb_vpn_tags [0:TLB_ENTRIES-1],
    input  logic [19:0] tlb_ppn_data [0:TLB_ENTRIES-1],
    input  logic        tlb_valid    [0:TLB_ENTRIES-1],
    output logic [19:0] ppn_out,
    output logic        tlb_hit
);
    always_comb begin
        tlb_hit = 1'b0; ppn_out = 20'h00000;
        for (int i = 0; i < TLB_ENTRIES; i++) begin
            if (tlb_valid[i] && (tlb_vpn_tags[i] == vpn_in)) begin
                tlb_hit = 1'b1; ppn_out = tlb_ppn_data[i];
            end
        end
    end
endmodule
```

Practice: [EASY] Basic AMAT Calculation

Context: Before optimizing a system, an engineer must quantify its baseline performance.

Problem: A processor has a single L1 data cache.

- Hit Time = 1 clock cycle.
- Miss Rate = 5%.
- Miss Penalty to fetch data from DRAM = 100 clock cycles.

Calculate the AMAT.

Solution: [EASY] Basic AMAT Calculation

Step 1: Identify the AMAT formula.

$$\text{AMAT} = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

Step 2: Plug in the values.

$$\text{AMAT} = 1 + (0.05 \times 100)$$

Step 3: Solve.

$$\text{AMAT} = 1 + 5 = \mathbf{6} \text{ cycles}$$

Conclusion: Even with a 95% hit rate, the system takes an average of 6 cycles per access, making it 6x slower than a "perfect" 1-cycle memory!

Practice: [MEDIUM] Effective CPI Impact

Context: Hardware engineers care about how memory stalls slow down the *entire* CPU pipeline.

Problem: A pipelined processor has a base CPI of 1.0 (assuming perfect memory).

- 25% of all instructions executed are memory accesses (`lw`/`sw`).
- The L1 cache has a Miss Rate of 4%.
- The DRAM Miss Penalty is 80 cycles.

Calculate the Effective (Actual) CPI of the processor.

Solution: [MEDIUM] Effective CPI Impact

Step 1: Calculate the Memory Stall Cycles per Instruction.

$$\text{Stalls} = \text{Memory Insts} \times \text{Miss Rate} \times \text{Miss Penalty}$$

$$\text{Stalls} = 0.25 \times 0.04 \times 80$$

$$\text{Stalls} = \mathbf{0.8} \text{ cycles}$$

Step 2: Add the stalls to the Base CPI.

$$\text{Effective CPI} = \text{Base CPI} + \text{Stalls}$$

$$\text{Effective CPI} = 1.0 + 0.8 = \mathbf{1.8}$$

Conclusion: Cache misses almost doubled the execution time of the entire program!

Practice: [HARD] Multi-Level Cache Architecture

Context: To fix the terrible performance degradation, we add an L2 cache.

Problem: You upgrade the previous processor with an L2 cache.

- L1 Hit Time: 1 cycle
- L1 Miss Rate: 4%
- L2 Hit Time: 10 cycles *(This becomes the L1 Miss Penalty!)*
- L2 Miss Rate: 20% *(20% of requests that reach L2 miss and go to DRAM)*
- DRAM Miss Penalty: 80 cycles

Calculate the new global AMAT.

Solution: [HARD] Multi-Level Cache Architecture

Step 1: Calculate the L2 AMAT (which acts as the L1 Miss Penalty).

$$\text{L1 Miss Pen.} = \text{L2 Hit Time} + (\text{L2 Miss Rate} \times \text{DRAM Penalty})$$

$$\text{L1 Miss Pen.} = 10 + (0.20 \times 80)$$

$$\text{L1 Miss Pen.} = 10 + 16 = \mathbf{26} \text{ cycles}$$

Step 2: Calculate the Global System AMAT.

$$\text{AMAT} = \text{L1 Hit Time} + (\text{L1 Miss Rate} \times \text{L1 Miss Penalty})$$

$$\text{AMAT} = 1 + (0.04 \times 26)$$

$$\text{AMAT} = 1 + 1.04 = \mathbf{2.04} \text{ cycles}$$

Conclusion: By adding L2, we dropped AMAT from 4.2 cycles down to 2.04 cycles!

Practice: [EASY] System Availability

Context: Datacenters guarantee service-level agreements (SLAs) based on statistical availability metrics.

Problem: A database server has a Mean Time To Failure (MTTF) of 3 years (approx 1,095 days). When memory corruption occurs, the OS crashes and the automated Mean Time To Repair (MTTR) reboot sequence takes exactly 1 day.

Calculate the Availability of the server as a percentage.

Solution: [EASY] System Availability

Step 1: Identify the Availability formula.

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Step 2: Plug in the values.

$$\text{Availability} = \frac{1095}{1095 + 1} = \frac{1095}{1096}$$

Step 3: Solve.

$$\text{Availability} \approx \mathbf{0.99908} \text{ (or } \mathbf{99.9\%} \text{ uptime)}$$

Practice: [MEDIUM] Even Parity Generation

Context: The simplest form of error detection is adding a single parity bit to a word before writing it to memory.

- **Even Parity:** The parity bit is set so the total number of 1s (data + parity) is an EVEN number.
- **Odd Parity:** The parity bit is set so the total number of 1s is an ODD number.

Problem: The CPU wants to store the 8-bit byte `1011 0101` in DRAM. The memory controller uses **Even Parity**.

1. What is the value of the parity bit appended to this byte?
2. If a cosmic ray flips the lowest bit (`b0`), what does the byte become, and how does the hardware detect the error during a read?

Solution: [MEDIUM] Even Parity Generation

Part 1: Generating the Parity Bit

- Count the number of 1s in 1011 0101. There are five 1s.
- For Even Parity, the total number of 1s must be even.
- The parity bit must be 1.
- The stored 9-bit word is 1_10110101.

Part 2: Detecting the Error

- The cosmic ray flips the lowest bit (0101 becomes 0100).
- The corrupted word read from memory is 1_10110100.
- The hardware counts the 1s. There are now five 1s.
- Because 5 is an *odd* number, the hardware immediately throws a Parity Exception, halting the CPU to prevent data corruption.

Practice: [HARD] Hamming Code Syndrome (SEC)

Context: Server-grade ECC memory uses overlapping parity equations to mathematically pinpoint flipped bits.

Problem: A 4-bit data word 1011 is encoded into a 7-bit Hamming Code: 0110011. While stored in DRAM, an alpha particle flips the sequence to 0110111. When the CPU reads this corrupted word, how does the hardware mathematically prove that **Bit 5** was the exact bit that flipped?

Solution: [HARD] Hamming Code Syndrome (SEC) - Part 1

Step 1: Understand the bit positions.

- Positions 1, 2, and 4 are the Parity Check Bits (p_1, p_2, p_4).
- Bit 1 (p_1): 0
- Bit 2 (p_2): 1
- Bit 3 (d_1): 1
- Bit 4 (p_4): 0
- **Bit 5 (d_2): 1 (Flipped)**
- Bit 6 (d_3): 1
- Bit 7 (d_4): 1

Solution: [HARD] Hamming Code Syndrome (SEC) - Part 2

Step 2: Recalculate Even Parity Equations.

- **Eq 1 (pos 1,3,5,7):** $0 + 1 + 1 + 1 = 3$ (Odd $\rightarrow$ FAIL $\rightarrow$ 1)
- **Eq 2 (pos 2,3,6,7):** $1 + 1 + 1 + 1 = 4$ (Even $\rightarrow$ PASS $\rightarrow$ 0)
- **Eq 3 (pos 4,5,6,7):** $0 + 1 + 1 + 1 = 3$ (Odd $\rightarrow$ FAIL $\rightarrow$ 1)

Step 3: Construct the Syndrome. Combine results from p_4 to p_1: `[Fail 4]` `[Pass 2]` `[Fail 1]` $=$ `101`

Step 4: Decode the Syndrome. Binary `101` to decimal equals **5**. The mathematics prove absolute certainty that **Bit 5** is the corrupted bit. The hardware automatically flips it back to `0` (Single Error Correction)!

Practice: [EASY] Virtual Address Partitioning

Context: Virtual memory divides an application's massive address space into manageable chunks called Pages.

Problem: A 32-bit architecture uses a standard Page Size of 4 KB (2^{12} bytes).
How many total Virtual Pages exist in the address space?

Solution: [EASY] Virtual Address Partitioning

Step 1: Determine total addressable bytes. A 32-bit address space can address 2^{32} total bytes (4 GB).

Step 2: Divide by the Page Size.

$$\text{Total Pages} = \frac{2^{32} \text{ bytes}}{2^{12} \text{ bytes/page}}$$

$$\text{Total Pages} = 2^{32-12} = 2^{20}$$

Step 3: Solve.

$$2^{20} = \mathbf{1,048,576} \text{ Virtual Pages}$$

Practice: [MEDIUM] Virtual to Physical Translation

Context: The OS Page Table acts as a dictionary translating fake Virtual Addresses to real Physical Addresses in DRAM.

Problem: A system uses 16-bit addresses and a 4 KB page size.

- The CPU requests a memory read at Virtual Address `0x2F3A`.
- The Page Table holds the following mapping: `Virtual Page 0x2` maps to `Physical Page 0x8`.

What is the exact 16-bit Physical Address generated by the hardware?

Solution: [MEDIUM] Virtual to Physical Translation

Step 1: Calculate the Offset width. 4 KB (2^{12} bytes) means the lower 12 bits (3 hex digits) represent the byte offset.

Step 2: Split the Virtual Address. Virtual Address = `0x2F3A`

- Offset = `0xF3A`
- Virtual Page Number (VPN) = `0x2`

Step 3: Lookup the Physical Page Number (PPN). VPN `0x2` translates to PPN `0x8`.

Step 4: Reconstruct the Physical Address. Concatenate the new PPN with the original Offset.

$$\text{Physical Address} = \texttt{0x8} + \texttt{0xF3A} = \texttt{0x8F3A}$$

Practice: [HARD] TLB and Page Fault Synthesis

Context: Combine TLB caching, Main Memory Page Table lookups, and Disk Page Faults into one massive AMAT calculation.

Problem:

- TLB Hit Time: 1 cycle
- TLB Miss Rate: 2%
- Main Memory (DRAM) Access Time: 100 cycles
- Page Fault Rate: 0.1% *(Happens during a TLB miss)*
- Disk Access Time (Page Fault Penalty): 10,000,000 cycles

Calculate the AMAT just for the *translation* phase of the instruction.

Solution: [HARD] TLB and Page Fault Synthesis

Step 1: Calculate the Page Fault Penalty. If TLB misses, we check the Page Table in DRAM (100 cycles). 0.1% of the time, we Page Fault to Disk.

$$\text{DRAM AMAT} = \text{DRAM Hit Time} + (\text{Page Fault Rate} \times \text{Disk Penalty})$$

$$\text{DRAM AMAT} = 100 + (0.001 \times 10,000,000)$$

$$\text{DRAM AMAT} = 100 + 10,000 = \mathbf{10,100} \text{ cycles}$$

Step 2: Calculate the TLB AMAT.

$$\text{AMAT} = \text{TLB Hit Time} + (\text{TLB Miss Rate} \times \text{TLB Miss Penalty})$$

$$\text{AMAT} = 1 + (0.02 \times 10,100)$$

$$\text{AMAT} = 1 + 202 = \mathbf{203} \text{ cycles}$$

[EASY] Problem 5.17.1: Page Table Sizing

Context: Virtual memory provides the illusion of infinite memory, but the Page Tables themselves must be physically stored in DRAM, consuming actual memory.

Problem: Given a 32-bit Virtual Address size, an 8 KB page size, and a 4-byte Page Table Entry (PTE) size, calculate the total page table size for a system running 5 applications that utilize half of the memory available.

Solution: Page Table Sizing

1. **Calculate the number of index bits (Tag):**
   - Offset $= \log_2(8192) = 13$ bits.
   - The remaining bits form the Page Table Index (Tag) $= 32 - 13 = 19$ bits.
   - Total entries per application $= 2^{19}$ entries.
2. **Calculate the memory for one page table:**
   - Size $= 2^{19} \times 4$ bytes $= 2,097,152$ bytes $= 2$ MB.
3. **Calculate the total system memory overhead:**
   - Assuming the 5 applications utilize memory linearly, all 5 page tables must exist.
   - Total Size $= 5 \times 2$ MB $= \mathbf{10}$ MB.

[EASY] Problem 5.19.1: TLB Lifecycle

Context: The Translation Lookaside Buffer (TLB) caches active memory mappings. However, the OS must aggressively manage its validity.

Problem: Under what scenarios would a TLB entry's valid bit be set to zero by the hardware or the Operating System?

Solution: TLB Lifecycle

A TLB entry's valid bit is set to zero (invalidated) in the following scenarios:

1. **Context Switches:** When the OS switches execution from Process A to Process B, Process A's virtual addresses are no longer valid for Process B. The entire TLB is flushed (all valid bits set to zero).
2. **Page Replacement:** If the OS evicts a physical page from DRAM to the Disk to free up space, any TLB entry pointing to that physical page must be immediately invalidated.
3. **Process Termination:** When an application crashes or exits, its TLB entries are invalidated.

[EASY] Problem 5.13.1 & .2: Datacenter Reliability

Context: Cloud providers like AWS and GCP must statistically guarantee uptime using reliability metrics.

Problem: A storage array has a Mean Time To Failure (MTTF) of 3 years and a Mean Time To Repair (MTTR) of 1 day. Calculate the Mean Time Between Failures (MTBF) and the System Availability.

Solution: Datacenter Reliability

1. **Convert to a common unit (days):**
   - $\text{MTTF} = 3 \times 365 = 1095$ days.
   - $\text{MTTR} = 1$ day.
2. **Calculate MTBF:**
   - $\text{MTBF} = \text{MTTF} + \text{MTTR}$
   - $\text{MTBF} = 1095 + 1 = \mathbf{1096}$ days
3. **Calculate Availability:**
   - $\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}}$
   - $\text{Availability} = \frac{1095}{1096} \approx \mathbf{0.99908}$ (99.908% uptime)

[MEDIUM] Problem 5.12.1: Multi-Level CPI

Context: Modern processors survive the "Memory Wall" by relying heavily on multi-level caches.

Problem: A 2 GHz processor has a base CPI of 1.5. Instruction/Data misses total 7% per instruction. Main memory takes 100 ns. Calculate the Effective CPI using:

1. Only an L1 cache.
2. An L1 + an 8-way Set Associative L2 cache (L2 hit time = 28 cycles, L2 miss rate = 1.5%).

Solution: Multi-Level CPI

1. **Convert 100 ns Memory Access to Cycles:**
   - Clock Cycle Time $= \frac{1}{2 \text{ GHz}} = 0.5$ ns.
   - DRAM Penalty $= \frac{100 \text{ ns}}{0.5 \text{ ns}} = 200$ cycles.
2. **Calculate CPI for L1 Only:**
   - $\text{CPI}_{\text{L1}} = 1.5 + (0.07 \times 200) = 1.5 + 14 = \mathbf{15.5}$
3. **Calculate CPI with L2 (8-Way):**
   - $\text{Stalls} = \text{L1 Misses} \times (\text{L2 Hit Time} + (\text{L2 Misses}_{\text{local}} \times \text{DRAM Penalty}))$
   - $\text{Stalls} = 0.07 \times (28 + (0.015 \times 200))$
   - $\text{Stalls} = 0.07 \times (28 + 3) = 0.07 \times 31 = 2.17$
   - $\text{CPI}_{\text{L2}} = 1.5 + 2.17 = \mathbf{3.67}$

Note: The textbook mathematically treats 1.5% as a Local Miss Rate out of the L1 misses. Adding the L2 drops CPI from 15.5 down to 3.67!

[MEDIUM] Problem 5.18.1: Single-Level Page Table

Context: Modern 64-bit operating systems use incredibly sparse Virtual Address spaces.

Problem: For a single-level page table on a system with a 43-bit Virtual Address, 16 GB Physical DRAM, 4 KB Pages, and 4-byte PTEs:

1. How many Page Table Entries (PTEs) are needed?
2. How much physical memory is required for the table?

Solution: Single-Level Page Table

1. **Calculate the Virtual Page Number (VPN) size:**
   - Virtual Address $= 43$ bits.
   - Page Size $= 4$ KB $= 2^{12}$ bytes (12 bits for the offset).
   - VPN $= 43 - 12 = 31$ bits.
2. **Calculate the number of PTEs:**
   - A single-level page table must contain an entry for *every* possible Virtual Page.
   - Number of PTEs $= 2^{31} = \mathbf{2,147,483,648}$ entries.
3. **Calculate the physical memory required:**
   - Memory $= 2^{31} \times 4$ bytes $= 2^{33}$ bytes $= \mathbf{8}$ GB.

Conclusion: A single-level table consumes 8 GB (50%) of the 16 GB DRAM just for metadata! This necessitates Multi-Level Page Tables.

[MEDIUM] Problem 5.14.2: ECC Cost vs Performance

Context: Enterprise servers must balance the cost of extra parity RAM chips against mathematical protection.

Problem: Modern Server DIMMs employ SEC/DED ECC to protect each 64 bits of data with 8 parity bits. A competitor suggests protecting 128 bits of data with 9 parity bits. Compute the cost/performance ratio of both codes (Cost = overhead %, Performance = protection rate %). Which is better?

Solution: ECC Cost vs Performance

1. **Analyze the standard (72, 64) code:**
   - Cost (Overhead) $= \frac{8}{64} = 12.5\%$.
   - Performance (Protection rate for 1 single error) $= \frac{1}{72} = 1.4\%$.
   - Ratio $= \frac{12.5}{1.4} \approx \mathbf{8.9}$.
2. **Analyze the competitor (137, 128) code:**
   - Cost (Overhead) $= \frac{9}{128} = 7.0\%$.
   - Performance (Protection rate) $= \frac{1}{137} = 0.73\%$.
   - Ratio $= \frac{7.0}{0.73} \approx \mathbf{9.6}$.

Conclusion: The standard (72, 64) code has a lower cost/performance ratio (8.9 vs 9.6), making it the superior mathematical choice for server memory.

[HARD] Problem 5.21.1: Virtual Machine Overhead

Context: Cloud platforms (AWS EC2) run hypervisors (VMMs) beneath the Guest OS, introducing massive performance penalties.

Problem: A native system has a Base CPI of 1.5. When running inside a VM, the Guest OS triggers a privileged instruction trap 120 times per 10,000 instructions. The trap penalty to the Guest OS is 15 cycles, but the trap to the VMM (Hypervisor) takes 175 cycles. Calculate the Effective CPI of this virtualized system.

Solution: Virtual Machine Overhead

1. **Identify the trap frequency per instruction:**
   - Trap Rate $= \frac{120}{10{,}000} = 0.012$ traps per instruction.
2. **Calculate the Total Trap Penalty:**
   - A virtualized OS trap must *first* hit the Hypervisor (VMM), and then the Hypervisor hands it to the Guest OS.
   - Total Penalty $= 175$ (VMM) $+ 15$ (Guest) $= 190$ cycles.
3. **Calculate the Effective CPI:**
   - $\text{CPI}_{\text{VM}} = \text{Base CPI} + (\text{Trap Rate} \times \text{Total Penalty})$
   - $\text{CPI}_{\text{VM}} = 1.5 + (0.012 \times 190)$
   - $\text{CPI}_{\text{VM}} = 1.5 + 2.28 = \mathbf{3.78}$

Conclusion: Running the OS inside a Virtual Machine slows the CPU down by over 2.5x!

[HARD] Problem 5.16.1: Full Virtual Memory Trace

Context: The Memory Management Unit (MMU) must orchestrate the TLB, the Page Table, and the Disk.

Problem: Assume a system with 4 KB pages. The TLB currently holds: `[Valid=1, Tag=11]`, `[Valid=1, Tag=7]`, `[Valid=1, Tag=3]`, `[Valid=0]`. The Page Table holds: `[Idx 0: Valid, PPN 5]`, `[Idx 1: Disk]`, `[Idx 3: Valid, PPN 6]`. The CPU requests Virtual Addresses (in decimal): 4669, 2227, 13916. Trace the hits and misses.

Solution: Full Virtual Memory Trace

1. **Access 1:** 4669
   - *Math:* $4669/4096 = 1$ (VPN 1).
   - VPN 1 is not in the TLB $\rightarrow$ **TLB Miss**.
   - Look up VPN 1 (Idx 1) in PT. It is marked as `Disk`! $\rightarrow$ **Page Fault**.
2. **Access 2:** 2227
   - *Math:* $2227/4096 = 0$ (VPN 0).
   - VPN 0 is not in the TLB $\rightarrow$ **TLB Miss**.
   - Look up VPN 0 (Idx 0) in PT. It is Valid (PPN 5) $\rightarrow$ **PT Hit**.
3. **Access 3:** 13916
   - *Math:* $13916/4096 = 3$ (VPN 3).
   - VPN 3 matches the third entry in our TLB (`Tag=3, Valid=1`) $\rightarrow$ **TLB Hit!**.

[HARD] Problem 5.15.3: Disk Performance and B-Trees

Context: Modern databases (like Postgres) align their B-Tree index node sizes with the optimal OS Page Size to minimize disk seeking.

Problem: If using a modern magnetic disk with a 3 ms seek latency and a 100 MB/s transfer rate, what is the optimal page size to minimize the cost of pulling an index off the disk? Explain why future servers are likely to use larger pages.

Solution: Disk Performance and B-Trees

1. **Understand the tradeoff:**
   - A larger page takes longer to transfer (100 MB/s), but pulling in more data amortizes the massive 3 ms seek latency.
2. **Compare Latency vs Transfer limits:**
   - At 100 MB/s, transferring 4 KB takes 0.04 ms. Total = 3.04 ms.
   - Transferring 64 KB takes 0.64 ms. Total = 3.64 ms.
   - Transferring 256 KB takes 2.56 ms. Total = 5.56 ms.
3. **Conclusion:**
   - The mathematical utility curve proves that **64 KB** is the optimal page size for this specific disk.
   - *Why larger pages?* Disk bandwidth is growing exponentially faster than disk seek latency. To hide the physical seek penalty, we must maximize the payload (e.g., 2MB or 1GB Huge Pages).

Conclusion

This concludes our journey through the computer architecture stack!

- **Logic Gates → ALU → Single-Cycle CPU → Pipelined Datapath**
- **SRAM Caches → DRAM Main Memory → Virtual Memory**

We have successfully bridged the gap from boolean logic up to the foundation of modern, high-performance, fault-tolerant microprocessors.

Questions?