ECE/CS 552: Memory Hierarchy
Instructor: Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on notes by Mark Hill
Updated by Mikko Lipasti

Memory Hierarchy

CPU
I & D L1 Cache

Temporal Locality
• Keep recently referenced items at higher levels
• Future references satisfied quickly

Spatial Locality
• Bring neighbors of recently referenced to higher levels
• Future references satisfied quickly

Four Burning Questions
● These are:
  – Placement
    • Where can a block of memory go?
  – Identification
    • How do I find a block of memory?
  – Replacement
    • How do I make space for new blocks?
  – Write Policy
    • How do I propagate changes?
● Consider these for registers and main memory
  – Main memory usually DRAM

Placement

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Placement</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Anywhere; Int, FP, SPR</td>
<td>Compiler/programmer manages</td>
</tr>
<tr>
<td>Cache (SRAM)</td>
<td>Fixed in H/W</td>
<td>Direct-mapped, set-associative, fully-associative</td>
</tr>
<tr>
<td>DRAM</td>
<td>Anywhere</td>
<td>O/S manages</td>
</tr>
<tr>
<td>Disk</td>
<td>Anywhere</td>
<td>O/S manages</td>
</tr>
</tbody>
</table>

Register File

● Registers managed by programmer/compiler
  – Assign variables, temporaries to registers
  – Limited name space matches available storage
  – Learn more in CS536, CS701

<table>
<thead>
<tr>
<th>Placement</th>
<th>Flexible (subject to data type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification</td>
<td>Implicit (name == location)</td>
</tr>
<tr>
<td>Replacement</td>
<td>Spill code (store to stack frame)</td>
</tr>
<tr>
<td>Write policy</td>
<td>Write-back (store on replacement)</td>
</tr>
</tbody>
</table>
Main Memory and Virtual Memory

- Use of virtual memory
  - Main memory becomes another level in the memory hierarchy
  - Enables programs with address space or working set that exceed physically available memory
    - No need for programmer to manage overlays, etc.
    - Sparse use of large address space is OK
  - Allows multiple users or programs to timeshare limited amount of physical memory space and address space
- Bottom line: efficient use of expensive resource, and ease of programming

Virtual Memory

- Enables
  - Use more memory than system has
  - Program can think it is the only one running
    - Don’t have to manage address space usage across programs
    - E.g. think it always starts at address 0x0
  - Memory protection
    - Each program has private VA space: no-one else can clobber
    - Better performance
      - Start running a large program before all of it has been loaded from disk

Virtual Memory – Placement

- Main memory managed in larger blocks
  - Page size typically 4K – 16K
- Fully flexible placement; fully associative
  - Operating system manages placement
  - Indirection through page table
  - Maintain mapping between:
    - Virtual address (seen by programmer)
    - Physical address (seen by main memory)

Virtual Memory – Identification

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20004000</td>
<td>0x2000</td>
<td>Y/N</td>
</tr>
</tbody>
</table>

- Similar to cache tag array
  - Page table entry contains VA, PA, dirty bit
- Virtual address:
  - Matches programmer view; based on register values
  - Can be the same for multiple programs sharing same system, without conflicts
- Physical address:
  - Invisible to programmer, managed by O/S
  - Created/deleted on demand basis, can change

Virtual Memory – Replacement

- Similar to caches:
  - FIFO
  - LRU; overhead too high
    - Approximated with reference bit checks
    - Clock algorithm
  - Random
- O/S decides, manages
  - CS537
Virtual Memory – Write Policy

- Write back
  - Disks are too slow to write through
- Page table maintains dirty bit
  - Hardware must set dirty bit on first write
  - O/S checks dirty bit on eviction
  - Dirty pages written to backing store
  - Disk write, 10+ ms

Virtual Memory Implementation

- Caches have fixed policies, hardware FSM for control, pipeline stall
- VM has very different miss penalties
  - Remember disks are 10+ ms!
  - Hence engineered differently

Page Faults

- A virtual memory miss is a page fault
  - Physical memory location does not exist
  - Exception is raised, save PC
  - Invoke OS page fault handler
  - Find a physical page (possibly evict)
  - Initiate fetch from disk
  - Switch to other task that is ready to run
  - Interrupt when disk access complete
  - Restart original instruction
- Why use O/S and not hardware FSM?

Address Translation

<table>
<thead>
<tr>
<th>VA</th>
<th>PA</th>
<th>Dirty</th>
<th>Ref</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2000</td>
<td>0x2000</td>
<td>Y/N</td>
<td>Y/N</td>
<td>Read/Write/Execute</td>
</tr>
</tbody>
</table>

- O/S and hardware communicate via PTE
- How do we find a PTE?
  - &PTE = PTBR + page number * sizeof(PTE)
  - PTBR is private for each program
  - Context switch replaces PTBR contents

Page Table Size

- How big is page table?
  - $2^{32} / 4K \times 4B = 4M$ per program (!)
  - Much worse for 64-bit machines
- To make it smaller
  - Use limit register(s)
    - If VA exceeds limit, invoke O/S to grow region
  - Use a multi-level page table
  - Make the page table pageable (use VM)
Multilevel Page Table

- PT contains an entry for each real address
  - Instead of entry for every virtual address
  - Entry is found by hashing VA
  - Oversize PT to reduce collisions: \#PTE = 4 \times \#\text{phys. pages}

Hashed Page Table

- Use a hash table or inverted page table
  - PT contains an entry for each real address
  - Entry is found by hashing VA
  - Oversize PT to reduce collisions: \#PTE = 4 \times \#\text{phys. pages}

Hashed Page Table

- VA translation
  - Additional memory reference to PTE
  - Each instruction fetch/load/store now 2 memory references
  - Or more, with multilevel table or hash collisions
  - Even if PTE are cached, still slow
  - Hence, use special-purpose cache for PTEs
  - Called TLB (translation lookaside buffer)
  - Caches PTE entries
  - Exploits temporal and spatial locality (just a cache)

High-Performance VM

- Sharing is possible, necessary, desirable
  - Grant specific permissions
    - Read
    - Write
    - Execute
    - Any combination
  - Store permissions in PTE and TLB

Virtual Memory Protection
VM Sharing

- Share memory locations by:
  - Map shared physical location into both address spaces:
    - E.g. PA 0xC00DA becomes:
      - VA 0x2D000DA for process 0
      - VA 0x4D000DA for process 1
    - Either process can read/write shared location
  - However, causes synonym problem

VA Synonyms

- Virtually-addressed caches are desirable
  - No need to translate VA to PA before cache lookup
  - Faster hit time, translate only on misses
- However, VA synonyms cause problems
  - Can end up with two copies of same physical line
- Solutions:
  - Flush caches/TLBs on context switch
  - Extend cache tags to include PID & prevent duplicates
    - Effectively a shared VA space (PID becomes part of address)

Main Memory Design

- Storage in commodity DRAM
- How do we map these to logical cache organization?
  - Block size
  - Bus width
  - Etc.

Main Memory Access

- Each memory access
  - 1 cycle address
  - 5 cycle DRAM (really >> 10)
  - 1 cycle data
  - 4 word cache block
- One word wide: (a=addr, d=delay, b=bus)
  - adddddbdddddddbdddddddb
  - $1 + 4 \times (5+1) = 25$ cycles

Four word wide:
  - adddddb
  - $1 + 5 + 1 = 7$ cycles
- Interleaved (pipelined)
  - adddddb
  - ddddd b
  - ddddd b
  - ddddd b
  - $1 + 5 + 4 = 10$ cycles
Error Detection and Correction

- Main memory stores a huge number of bits
  - Probability of bit flip becomes nontrivial
  - Bit flips (called soft errors) caused by:
    - Slight manufacturing defects
    - Gamma rays and alpha particles
    - Interference
    - Etc.
  - Getting worse with smaller feature sizes
- Reliable systems must be protected from soft errors via ECC (error correction codes)
  - Even PCs support ECC these days

Error Correcting Codes

- Probabilities:
  \[ P(\text{1 word no errors}) > P(\text{single error}) > P(\text{two errors}) > P(>2 \text{ errors}) \]
- Detection - signal a problem
- Correction - restore data to correct value
- Most common:
  - Parity - single error detection
  - SECDED - single error correction; double bit detection
- Supplemental reading on course web page!

1-bit ECC

<table>
<thead>
<tr>
<th>Power</th>
<th>Correct</th>
<th>#bits</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing</td>
<td>0,1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SED</td>
<td>00,11</td>
<td>2</td>
<td>01,10 detect errors</td>
</tr>
<tr>
<td>SEC</td>
<td>000,111</td>
<td>3</td>
<td>001,010,100 =&gt; 0</td>
</tr>
<tr>
<td>SECDED</td>
<td>0000,111</td>
<td>4</td>
<td>One 1 =&gt; 0 Two 1's =&gt; error Three 1's =&gt; 1</td>
</tr>
</tbody>
</table>

ECC

- Reduce overhead by doing codes on word, not bit

<table>
<thead>
<tr>
<th># bits</th>
<th>SED overhead</th>
<th>SECDED overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 (100%)</td>
<td>3 (300%)</td>
</tr>
<tr>
<td>32</td>
<td>1 (3%)</td>
<td>7 (22%)</td>
</tr>
<tr>
<td>64</td>
<td>1 (1.6%)</td>
<td>8 (13%)</td>
</tr>
<tr>
<td>n</td>
<td>1 (1/n)</td>
<td>1 + log₂ n + a little</td>
</tr>
</tbody>
</table>

64-bit ECC

- 64 bits data with 8 check bits
  
- Use eight by 9 SIMMS = 72 bits
- Intuition:
  - One check bit is parity
  - Other check bits point to
    - Error in data, or
    - Error in all check bits, or
    - No error
**ECC**

- To store (write)
  - Use data₀ to compute check₀
  - Store data₀ and check₀
- To load
  - Read data₁ and check₁
  - Use data₁ to compute check₂
  - Syndrome = check₁ xor check₂
  - I.e. make sure check bits are equal

**ECC Syndrome**

<table>
<thead>
<tr>
<th>Syndrome</th>
<th>Parity</th>
<th>Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
<td>data₀ == data₀</td>
</tr>
<tr>
<td>n != 0</td>
<td>Not OK</td>
<td>Flip bit n of data₁ to get data₀</td>
</tr>
<tr>
<td>n != 0</td>
<td>OK</td>
<td>Signal uncorrectable error</td>
</tr>
</tbody>
</table>

**4-bit SECDED Code**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codeword</td>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
<td>C₃</td>
<td>b₀</td>
<td>b₁</td>
<td>b₂</td>
<td>b₃</td>
<td>P</td>
</tr>
<tr>
<td>C₀</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C₁</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C₂</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C₃</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₀</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₁</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₂</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₃</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>P</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- C₀ parity bits chosen specifically to:
  - Identify errors in bits where bit n of the index is 1
  - C₁ checks all odd bit positions (where LSB=1)
  - C₂ checks all positions where middle bit=1
  - C₃ checks all positions where MSB=1
- Hence, nonzero syndrome points to faulty bit

**4-bit SECDED Example**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codeword</td>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
<td>C₃</td>
<td>b₀</td>
<td>b₁</td>
<td>b₂</td>
<td>b₃</td>
</tr>
<tr>
<td>C₀</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C₁</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C₂</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C₃</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₀</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₁</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₂</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b₃</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>P</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- 4 data bits, 3 check bits, 1 parity bit
- Syndrome is xor of check bits C₁₃
  - If (syndrome==0) and (parity OK) => no error
  - If (syndrome != 0) and (parity OK) => flip bit position pointed to by syndrome
  - If syndrome != 0) and (parity OK) => double-bit error

**Summary**

- Memory hierarchy: Register file
  - Under compiler/programmer control
  - Complex register allocation algorithms to optimize utilization
- Memory hierarchy: Virtual Memory
  - Placement: fully flexible
  - Identification: through page table
  - Replacement: approximate LRU or LFU
  - Write policy: write-through

**Summary**

- Page tables
  - Forward page table
  - Multilevel page table
  - Inverted or hashed page table
  - Store PTE for each real page instead of each virtual page
  - HPT size scales up with physical memory
  - Also used for protection, sharing at page level
Summary

- **TLB**
  - Special-purpose cache for PTEs
  - Often accessed in parallel with L1 cache
- **Main memory design**
  - Commodity DRAM chips
  - Wide design space for
    - Minimizing cost, latency
    - Maximizing bandwidth, storage
  - Susceptible to soft errors
    - Protect with ECC (SECDED)
  - ECC also widely used in on-chip memories, busses