These slides are provided by:
David Patterson
Electrical Engineering and Computer Sciences, University of California, Berkeley

Ann Gordon-Ross
Electrical and Computer Engineering
University of Florida
http://www.ann.ece.ufl.edu/

Since 1980, CPU has outpaced DRAM ...

Q. How do architects address this gap?
A. Put smaller, faster "cache" memories between CPU and DRAM. Create a "memory hierarchy".

Since 1980, CPU has outpaced DRAM ...

These slides are provided by:
David Patterson
Electrical Engineering and Computer Sciences, University of California, Berkeley

Ann Gordon-Ross
Electrical and Computer Engineering
University of Florida
http://www.ann.ece.ufl.edu/

1977: DRAM faster than microprocessors

Apple II (1977)

CPU: 1000 ns
DRAM: 400 ns

Levels of the Memory Hierarchy

Upper Level

Registers
Instr. Operands

Cache

Main Memory

Cache

Memory

Disk

Files

Tape

Lower Level

faster

Larger
### Memory Hierarchy: Apple iMac G5

<table>
<thead>
<tr>
<th></th>
<th>Managed by compiler</th>
<th>Managed by hardware</th>
<th>Managed by OS, hardware, application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>1K</td>
<td>64K, 32K</td>
<td>512K, 256M, 80G</td>
</tr>
<tr>
<td>Latency/Cycles</td>
<td>1, 0.6 ns</td>
<td>3, 1.9 ns</td>
<td>3, 1.9 ns, 11, 6.9 ns, 88, 10^7, 55 ns, 12 ms</td>
</tr>
</tbody>
</table>

**Goal:** Illusion of large, fast, cheap memory

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access.

### The Principle of Locality

- **The Principle of Locality:**
  - Program access a relatively small portion of the address space at any instant of time.
- **Two Different Types of Locality:**
  - *Temporal Locality* (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - *Spatial Locality* (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- **Last 15 years, HW relied on locality for speed**

It is a property of programs which is exploited in machine design.
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** (500 instructions)

  - May be better to recalculate results instead of refetching

<table>
<thead>
<tr>
<th>To Processor</th>
<th>Upper Level Memory</th>
<th>Lower Level Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLK X</td>
<td>BLK Y</td>
</tr>
</tbody>
</table>

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** = Hit time + Miss rate x Miss penalty (ns or clocks)
- **Miss penalty**: time to replace a block from lower level, including time to replace in CPU
  - access time: time to lower level
    = f(latency to lower level)
  - transfer time: time to transfer block
    = f(BW between upper & lower levels)

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)
Q2: How is a block found if it is in the upper level?
• Tag on each block
  – No need to check index or block offset
• Increasing associativity shrinks index, expands tag

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

Q3: Which block should be replaced on a miss?
• Easy for Direct Mapped
• Set Associative or Fully Associative:
  – Random
  – LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Assoc:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Ran</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>

Q4: What happens on a write?

<table>
<thead>
<tr>
<th>Policy</th>
<th>Write-Through</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data written to cache block also written to lower-level memory</td>
<td>Write data only to the cache</td>
<td>Update lower level when a block falls out of the cache</td>
</tr>
<tr>
<td>Debug</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td>Do read misses produce writes?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Do repeated writes make it to lower level?</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Write Buffers for Write-Through Caches

Hold data awaiting write-through to lower level memory

Q. Why a write buffer? A. So CPU doesn’t stall
Q. Why a buffer, why not just one register? A. Bursts of writes are common.
Q. Are Read After Write (RAW) hazards an issue for write buffer? A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Additional option — let writes to an un-cached address allocate a new cache line ("write-allocate").
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. Giving Reads Priority over Writes
      • E.g., Read complete before earlier writes in write buffer

Outline

• Memory hierarchy
• Locality
• Cache design
• Virtual address spaces
• Page table layout
• TLB design options

The Limits of Physical Addressing

"Physical addresses" of memory locations

All programs share one address space: The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource

Solution: Add a Layer of Indirection

"Virtual Addresses" -> "Physical Addresses"

User programs run in an standardized virtual address space

Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory

Hardware supports “modern” OS features: Protection, Translation, Sharing
Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program ("Working Set") must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.

- Protection:
  - Different processes protected from each other.
  - Different pages can be given special behavior
    - (Read Only, invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs

- Sharing:
  - Can map same physical page to multiple users ("Shared memory")

Page tables encode virtual address spaces

A virtual address space is divided into blocks of memory called pages

A machine usually supports pages of a few sizes (MIPS R4000):

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Kbytes</td>
<td>Frame</td>
</tr>
<tr>
<td>16 Kbytes</td>
<td>Frame</td>
</tr>
<tr>
<td>64 Kbytes</td>
<td>Frame</td>
</tr>
<tr>
<td>256 Kbytes</td>
<td>Frame</td>
</tr>
<tr>
<td>1 Mbyte</td>
<td>Frame</td>
</tr>
<tr>
<td>4 Mbytes</td>
<td>Frame</td>
</tr>
<tr>
<td>16 Mbytes</td>
<td>Frame</td>
</tr>
</tbody>
</table>

A valid page table entry codes physical memory “frame” address for the page

Details of Page Table

- Page table maps virtual page numbers to physical frames ("PTE" = Page Table Entry)
- Virtual memory ≠ treat memory = cache for disk
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries.
Each process needs its own address space!

Two-level Page Tables

32 bit virtual address

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 index</td>
<td>P2 index</td>
<td>Page Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top-level table wired in main memory
Subset of 1024 second-level tables in main memory; rest are on disk or unallocated

VM and Disk: Page replacement policy

Set of all pages in Memory

Tail pointer:
Clear the used bit in the page table

Head pointer:
Place pages on free list if used bit is still clear.
Schedule pages with dirty bit set to be written to disk.

Architect’s role:
support setting dirty and used bits

MIPS Address Translation: How does it work?

Translation Look-Aside Buffer (TLB)
A small fully-associative cache of mappings from virtual to physical addresses
TLB also contains protection bits for virtual address
Fast common case: Virtual address is in TLB, process has permission to read/write it.