Chapter 7

Memory Hierarchy
Outline

• Memory hierarchy
• The basics of caches
• Measuring and improving cache performance
• Virtual memory
• A common framework for memory hierarchy
Technology Trends

<table>
<thead>
<tr>
<th>Year</th>
<th>Capacity</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64 Kb 2:1!</td>
<td>250 ns</td>
</tr>
<tr>
<td>1983</td>
<td>256 Kb</td>
<td>220 ns</td>
</tr>
<tr>
<td>1986</td>
<td>1 Mb</td>
<td>190 ns</td>
</tr>
<tr>
<td>1989</td>
<td>4 Mb</td>
<td>165 ns</td>
</tr>
<tr>
<td>1992</td>
<td>16 Mb</td>
<td>145 ns</td>
</tr>
<tr>
<td>1995</td>
<td>64 Mb</td>
<td>120 ns</td>
</tr>
</tbody>
</table>

Logic: 4x in 1.5 years  4x in 3 years
DRAM: 4x in 3 years  2x in 10 years
Disk: 4x in 3 years  2x in 10 years
Processor Memory Latency Gap

- Moore’s Law: Processor-memory performance gap (grows 50% / year)
- Proc 60%/yr. (2X/1.5 yr)
- DRAM 9%/yr. (2X/10 yrs)

Year

Performance
Solution: Memory Hierarchy

- An Illusion of a large, fast, cheap memory
  - Fact: Large memories slow, fast memories small
  - How to achieve: hierarchy, parallelism

- An expanded view of memory system:
Memory Hierarchy: Principle

- At any given time, data is copied between only two adjacent levels:
  - Upper level: the one closer to the processor
    - Smaller, faster, uses more expensive technology
  - Lower level: the one away from the processor
    - Bigger, slower, uses less expensive technology

- **Block**: basic unit of information transfer
  - Minimum unit of information that can either be present or not present in a level of the hierarchy

![Diagram of Memory Hierarchy]

- To Processor
- From Processor
Why Hierarchy Works?

- **Principle of Locality:**
  - Program access a relatively small portion of the address space at any instant of time
  - 90/10 rule: 10% of code executed 90% of time

- Two types of locality:
  - Temporal locality: if an item is referenced, it will tend to be referenced again soon
  - Spatial locality: if an item is referenced, items whose addresses are close by tend to be referenced soon

![Probability of reference graph]

\[ P(n) = \frac{1}{2^{n-1}} \]
How Does It Work?

- Temporal locality: keep most recently accessed data items closer to the processor
- Spatial locality: move blocks consists of contiguous words to the upper levels
Levels of Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>100s Bytes</td>
<td>&lt;10s ns</td>
</tr>
<tr>
<td>Cache</td>
<td>K Bytes</td>
<td>10-100 ns</td>
</tr>
<tr>
<td>Main Memory</td>
<td>M Bytes</td>
<td>100ns-1us</td>
</tr>
<tr>
<td>Disk</td>
<td>G Bytes</td>
<td>ms</td>
</tr>
<tr>
<td>Tape</td>
<td>infinite sec-min</td>
<td>10^-6</td>
</tr>
</tbody>
</table>

Upper Level
- Staging Transfer Unit
  - prog./compiler
    - 1-8 bytes
  - cache controller
    - 8-128 bytes
OS
- 512-4K bytes
user/operator
- Mbytes
Lower Level
- faster
- Larger
How Is the Hierarchy Managed?

- Registers <-> Memory
  - by compiler (programmer?)

- cache <-> memory
  - by the hardware

- memory <-> disks
  - by the hardware and operating system (virtual memory)
  - by the programmer (files)
Memory Hierarchy Technology

- **Random access:**
  - Access time same for all locations
  - **DRAM:** *Dynamic Random Access Memory*
    - High density, low power, cheap, slow
    - Dynamic: need to be refreshed regularly
    - Addresses in 2 halves (memory as a 2D matrix):
      - RAS/CAS (Row/Column Access Strobe)
    - Use for main memory
  - **SRAM:** *Static Random Access Memory*
    - Low density, high power, expensive, fast
    - Static: content will last (forever until lose power)
    - Address not divided
    - Use for caches
## Comparisons of Various Technologies

<table>
<thead>
<tr>
<th>Memory technology</th>
<th>Typical access time</th>
<th>$ per GB in 2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>0.5 – 5 ns</td>
<td>$4000 – $10,000</td>
</tr>
<tr>
<td>DRAM</td>
<td>50 – 70 ns</td>
<td>$100 – $200</td>
</tr>
<tr>
<td>Magnetic disk</td>
<td>5,000,000 – 20,000,000 ns</td>
<td>$0.05 – $2</td>
</tr>
</tbody>
</table>
Memory Hierarchy Technology

● Performance of main memory:
  – Latency: related directly to Cache Miss Penalty
  ● Access Time: time between request and word arrives
  ● Cycle Time: time between requests
  – Bandwidth: Large Block Miss Penalty (interleaved memory, L2)

● Non-so-random access technology:
  – Access time varies from location to location and from time to time, e.g., disk, CDROM

● Sequential access technology: access time linear in location (e.g., tape)
Memory Hierarchy: Terminology

- **Hit**: data appears in upper level (Block X)
  - Hit rate: fraction of memory access found in the upper level
  - Hit time: time to access the upper level
    - RAM access time + Time to determine hit/miss
- **Miss**: data needs to be retrieved 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 to the processor (latency + transmit time)
- **Hit Time \ll Miss Penalty**
4 Questions for Hierarchy Design

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
Memory System Design

Workload or Benchmark programs

Processor

reference stream
<op,addr>, <op,addr>,<op,addr>,<op,addr>, . . .

op: i-fetch, read, write

Memory

$ Mem

Optimize the memory system organization to minimize the average memory access time for typical workloads
Summary of Memory Hierarchy

- Two different types of locality:
  - Temporal Locality (Locality in Time)
  - Spatial Locality (Locality in Space)

- Using the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

- DRAM is slow but cheap and dense:
  - Good for presenting users with a BIG memory system

- SRAM is fast but expensive, not very dense:
  - Good choice for providing users FAST accesses
Outline

• Memory hierarchy
• The basics of caches (7.2)
• Measuring and improving cache performance
• Virtual memory
• A common framework for memory hierarchy
Levels of Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU Registers</strong></td>
<td>100s Bytes</td>
<td>&lt;10s ns</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>K Bytes</td>
<td>10-100 ns</td>
</tr>
<tr>
<td></td>
<td>$.01-.001/bit</td>
<td></td>
</tr>
<tr>
<td><strong>Main Memory</strong></td>
<td>M Bytes</td>
<td>100ns-1us</td>
</tr>
<tr>
<td></td>
<td>$.01-.001</td>
<td></td>
</tr>
<tr>
<td><strong>Disk</strong></td>
<td>G Bytes</td>
<td>ms</td>
</tr>
<tr>
<td></td>
<td>10^{-3} - 10^{-4} cents</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Tape</strong></td>
<td>infinite sec-min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10^{-6}</td>
</tr>
</tbody>
</table>

Upper Level
- Staging
- Transfer Unit
  - prog./compiler
  - 1-8 bytes
- cache controller
  - 8-128 bytes
- OS
  - 512-4K bytes
- user/operator
  - Mbytes

Lower Level
- faster
- Larger
Basics of Cache

- Our first example: *direct-mapped cache*

- **Block Placement:**
  - For each item of data at the lower level, there is exactly one location in cache where it might be
  - Address mapping: modulo number of blocks

- **Block identification:**
  - How to know if an item is in cache?
  - If it is, how do we find it?

![Diagram of tag and valid bit](image)
Accessing a Cache

- 1K words, 1-word block:
  - Cache index: lower 10 bits
  - Cache tag: upper 20 bits
  - Valid bit (When start up, valid is 0)

Fig. 7.7
Hits and Misses

• Read hits: this is what we want!

• Read misses
  – Stall CPU, freeze register contents, fetch block from memory, deliver to cache, restart
  – Block replacement?

• Write hits: keep cache/memory consistent?
  – Write-through: write to cache and memory at same time => but memory is very slow!
  – Write-back: write to cache only (write to memory when that block is being replaced)
    • Need a dirty bit for each block
Hits and Misses

- **Write misses:**
  - **Write-allocated:** read block into cache, write the word
    - low miss rate, complex control, match with write-back
  - **Write-non-allocate:** write directly into memory
    - high miss rate, easy control, match with write-through

- **DECStation 3100 uses write-through, but no need to consider hit or miss on a write (one block has only one word)**
  - index the cache using bits 15-2 of the address
  - write bits 31-16 into tag, write data, set valid
  - write data into main memory
Miss Rate

- Miss rate of Instrinsity FastMATH for SPEC2000 Benchmark:

<table>
<thead>
<tr>
<th>Instrinsity FastMATH</th>
<th>Instruction miss rate</th>
<th>Data miss rate</th>
<th>Effective combined miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.4%</td>
<td>11.4%</td>
<td>3.2%</td>
</tr>
</tbody>
</table>

Fig. 7.10
- Use a write buffer (WB):
  - Processor: writes data into cache and WB
  - Memory controller: write WB data to memory

- Write buffer is just a FIFO:
  - Typical number of entries: 4

- Memory system designer’s nightmare:
  - Store frequency > 1 / DRAM write cycle
  - Write buffer saturation => CPU stalled
Exploiting Spatial Locality (I)

- Increase block size for spatial locality

Total no. of tags and valid bits reduced

Fig. 7.9
Exploiting Spatial Locality (II)

- Increase block size for spatial locality
  - Read miss: bring back the whole block
  - Write:
    - Write through: Tag-check and write to the cache in one cycle
      - Miss: fetch-on-write, or no-fetch-on-write (just allocate)
    - Write back:
      - **If cache is dirty, the old block overwritten**
        (a) tag-check and then write (two cycles)
        Why?
        (b) need one extra cache buffer (one cycle)
        - Miss: write to memory buffer
Increase block size tends to decrease miss rate

Fig. 7.8

Block Size on Performance
Block Size Tradeoff

- Larger block size take advantage of spatial locality and improve miss ratio, BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size too big, miss rate goes up
    - Too few blocks in cache => high competition

- Average access time:
  \[ \text{Ave. Access Time} = \text{hit time} \times (1 - \text{miss rate}) + \text{miss penalty} \times \text{miss rate} \]
Memory Design to Support Cache

• How to increase memory bandwidth to reduce miss penalty?

a. One-word-wide memory organization

b. Wide memory organization

c. Interleaved memory organization

Fig. 7.11
Interleaving for Bandwidth

- Access pattern without interleaving:
  - Cycle time
  - Access time
  - D1 available
  - Start access for D1
  - Start access for D2

- Access pattern with interleaving
  - Data ready
  - Access Bank 0, 1, 2, 3
  - Transfer time
  - Access Bank 0 again
Miss Penalty for Different Memory Organizations

Assume
- 1 memory bus clock to send the address
- 15 memory bus clocks for each DRAM access initiated
- 1 memory bus clock to send a word of data
- A cache block = 4 words

Three memory organizations:
- A one-word-wide bank of DRAMs
  - Miss penalty = 1 + 4 x 15 + 4 x 1 = 65

- A two-word-wide bank of DRAMs
  - Miss penalty = 1 + 2 x 15 + 2 x 1 = 33
Access of DRAM

Row decoder 11-to-2048

2048 3 2048 array

Column latches

Mux

Dout

Address[10:0]
DDR SDRAM

Double Data Rate Synchronous DRAMs

- Burst access from a sequential locations
- Starting address, burst length
- Data transferred under control of clock
  
  (300 MHz, 2004)

- Clock is used to eliminate the need of synchronization and the need of supplying successive address
- Data transfer on both leading an falling edge of clock
Cache Performance

• Simplified model: (instruction misses)
  CPU time = (CPU execution cycles +
  memory stall cycles) x cycle time
  Memory stall cycles = instruction count x
  miss ratio x miss penalty

• Impact on performance: (data misses)
  – Suppose CPU executes at clock rate = 200MHz, CPI=1.1,
    50% arith/logic, 30% ld/st, 20% control
  – 10% memory op. get 50-cycle miss penalty
  – CPI = ideal CPI + average stalls per instruction
    = 1.1+(0.30 mops/ins x 0.10 miss/mop x 50 cycle/miss) = 1.1
    cycle + 1.5 cycle = 2.6
  – 58% of the time CPU stalled waiting for memory!
  – 1% inst. miss rate adds extra 0.5 cycles to CPI!
Improving Cache Performance

- Decreasing the miss ratio
- Reduce the time to hit in the cache
- Decreasing the miss penalty
Basics of Cache

- Our first example: *direct-mapped cache*

- Block Placement:
  - For each item of data at the lower level, there is exactly one location in cache where it might be
  - Address mapping: modulo number of blocks

- Block identification:
  - How to know if an item is in cache? **Tag and valid bit**
  - If it is, how do we find it?
Exploiting Spatial Locality (I)

- Increase block size for spatial locality

Total no. of tags and valid bits reduced

Fig. 7.9
Reduce Miss Ratio with Associativity

- A fully associative cache:
  - Compare cache tags of all cache entries in parallel
  - Ex.: Block Size = 8 words, N 27-bit comparators

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
\text{Cache Tag (27 bits long)} & \text{Valid Bit} & \text{Cache Data} \\
\hline
\text{Byte 0} & \text{...} & \text{Byte 1} & \text{Byte 0} \\
\text{Byte 1} & \text{...} & \text{Byte 33} & \text{Byte 32} \\
\text{Byte 27} & \text{...} & \text{...} & \text{...} \\
\hline
\end{array}
\]
Set-Associative Cache

- N-way: N entries for each cache index
  - N direct mapped caches operates in parallel
- Example: two-way set associative cache
  - Cache Index selects a set from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result
Possible Associativity Structures

An 8-block cache

Fig. 7.14
A 4-Way Set-Associative Cache

- Increasing associativity shrinks index, expands tag

Fig. 7.17
Block Placement

- Placement of a block whose address is 12:

Fig. 7.13
Data Placement Policy

● Direct mapped cache:
  – Each memory block mapped to one location
  – No need to make any decision
  – Current item replaces previous one in location

● N-way set associative cache:
  – Each memory block has choice of \( N \) locations

● Fully associative cache:
  – Each memory block can be placed in ANY cache location

● Misses in N-way set-associative or fully associative cache:
  – Bring in new block from memory
  – Throw out a block to make room for new block
  – Need to decide on which block to throw out
Cache Block Replacement

- Easy for direct mapped
- Set associative or fully associative:
  - Random
  - LRU (Least Recently Used):
    - Hardware keeps track of the access history and replace the block that has not been used for the longest time
  - An example of a pseudo LRU (for a two-way set associative):
    - use a pointer pointing at each block in turn
    - whenever an access to the block the pointer is pointing at, move the pointer to the next block
    - when need to replace, replace the block currently pointed at
Comparing the Structures

- **N-way set-associative cache**
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection

- **Direct mapped cache**
  - Cache block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue, recover later if miss
## Cache Performance

### Table: Cache Miss Rate

<table>
<thead>
<tr>
<th>Asso. Size</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Rdm</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>
Reduce Miss Penalty with Multilevel Caches

- Add a second level cache:
  - Often primary cache is on same chip as CPU
  - L1 focuses on minimizing hit time to reduce effective CPU cycle => faster (smaller), higher miss rate
  - L2 focuses on miss rate to reduce miss penalty
    => larger cache and larger block
    => miss penalty goes down if data is in L2 cache
  - Average access time
    = L1 hit time + L1 miss rate × L1 miss penalty
  - L1 miss penalty
    = L2 hit time + L2 miss rate × L2 miss penalty
Performance Improvement Using L2

- Example:
  - CPI of 1.0 on a 5GHz machine with a 2% miss rate,
  - 100ns DRAM access
  - Adding a L2 cache with 5ns access time and decrease of overall main memory miss rate to 0.5%, what miss penalty reduced?

100 ns / 0.2 (ns/clock cycle) = 500 clock cycles
Without L2:
1.0 + 2% x 500 = 11
With L2:
5ns / 0.2 (ns/clock cycle) = 25 clock cycles
1.0 + 2% x 25 + 0.5% x 500 = 2.8
Sources of Cache Misses

- **Compulsory (cold start, process migration):**
  - First access to a block, not much we can do
  - Note: If you are going to run billions of instruction, compulsory misses are insignificant

- **Conflict (collision):**
  - >1 memory blocks mapped to same location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- **Capacity:**
  - Cache cannot contain all blocks by program
  - Solution: increase cache size

- **Invalidation:**
  - Block invalidated by other process (e.g., I/O) that updates the memory
Cache Design Space

● Several interacting dimensions
  – cache size
  – block size
  – associativity
  – replacement policy
  – write-through vs write-back
  – write allocation

● The optimal choice is a compromise
  – depends on access characteristics
    ● workload
    ● use (I-cache, D-cache, TLB)
  – depends on technology / cost

● Simplicity often wins
Cache Summary

● Principle of Locality:
  – Program likely to access a relatively small portion of address space at any instant of time
    ● Temporal locality: locality in time
    ● Spatial locality: locality in space

● Three major categories of cache misses:
  – Compulsory: e.g., cold start misses.
  – Conflict: increase cache size or associativity
  – Capacity: increase cache size

● Cache design space
  – total size, block size, associativity
  – replacement policy
  – write-hit policy (write-through, write-back)
  – write-miss policy
Outline

• Memory hierarchy
• The basics of caches
• Measuring and improving cache performance
• Virtual memory
• A common framework for memory hierarchy
Levels of Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100s Bytes</td>
<td>&lt;10s ns</td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K Bytes</td>
<td>10-100 ns</td>
<td>$.01-.001/bit</td>
</tr>
<tr>
<td>Main Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M Bytes</td>
<td>100ns-1us</td>
<td>$.01-.001</td>
</tr>
<tr>
<td>Disk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G Bytes</td>
<td>ms</td>
<td>10⁻³ - 10⁻⁴ cents</td>
</tr>
<tr>
<td>Tape</td>
<td></td>
<td></td>
</tr>
<tr>
<td>infinite sec-min</td>
<td>10⁻⁶</td>
<td></td>
</tr>
</tbody>
</table>

- Registers
- Cache
- Memory
- Disk
- Files
- Blocks
- Instr. Operands

Upper Level:
- Staging Transfer Unit
  - prog./compiler
    - 1-8 bytes
  - cache controller
    - 8-128 bytes

Lower Level:
- OS
  - 512-4K bytes
- user/operator
  - Mbytes

Levels of Memory Hierarchy:
- Upper Level (faster)
- Lower Level (Larger)
Virtual Memory

• Provide illusion of a large single-level store
  – Every program has its own address space, starting at address 0, only accessible to itself
    • yet, any can run anywhere in physical memory
    • executed in a name space (virtual address space) different from memory space (physical address space)
    • virtual memory implements the translation from virtual space to physical space
  – Every program has lots of memory (> physical memory)

• Many programs run at once with protection and sharing
• OS runs all the time and allocates physical resources
Virtual Memory

- View main memory as a cache for disk

Fig. 7.19
Why Virtual Memory?

- Sharing: efficient and safe sharing of main memory among multiple programs
  - Map multiple virtual addresses to same physical addr.
- Generality: run programs larger than size of physical memory (Remove prog. burden of a small physical memory)
- Protection: regions of address space can be read-only, exclusive, ...
- Flexibility: portions of a program can be placed anywhere, without relocation
- Storage efficiency: retain only most important portions of program in memory
- Concurrent programming and I/O: execute other processes while loading/dumping page
Basic Issues in Virtual Memory

- **Size of data blocks** that are transferred from disk to main memory
- Which region of memory to hold new block
  => **placement policy**
- When memory is full, then some region of memory must be released to make room for the new block => **replacement policy**
- When to fetch missing items from disk?
  - Fetch only on a fault => **demand load policy**
Paging

- Virtual and physical address space
  - Pages partitioned into blocks of equal size

- Key operation: address mapping
  MAP: \( V \rightarrow M \cup \{\emptyset\} \) address mapping function
  \[
  \text{MAP}(a) = a' \text{ if data at virtual address } a \text{ is present in physical address } a' \text{ and } a' \text{ in } M
  \]
  \[
  = \emptyset \text{ if data at virtual address } a \text{ is not present in } M
  \]

Diagram:
- Processor
- Name Space V
- Addr Trans Mechanism
- Main Memory
- Secondary Memory
- Fault handler
- Missing item fault
- OS does this transfer
Key Decisions in Paging

- Huge miss penalty: a page fault may take millions of cycles to process
  - Pages should be fairly large (e.g., 4KB) to amortize the high access time
  - Reducing page faults is important
    - LRU replacement is worth the price
    - fully associative placement
      => use page table (in memory) to locate pages
  - Can handle the faults in software instead of hardware, because handling time is small compared to disk access
    - the software can be very smart or complex
    - the faulting process can be context-switched
  - Using write-through is too expensive, so we use write back \(\leq\) write policy (dirty bit)
Choosing the Page Size

- Minimize wasted storage (small page):
  - small page minimizes internal fragmentation
  - small page increase size of page table

- Minimize transfer time (large page):
  - large pages (multiple disk sectors) amortize access cost
  - sometimes transfer unnecessary info
  - sometimes prefetch useful data
  - sometimes discards useless data early

- A trend toward larger pages because
  - big cheap RAM
  - increasing memory/disk performance gap
  - larger address spaces
Page Tables

How many memory references for each address translation?

all addresses generated by the program are virtual addresses

Fig. 7.21
Page Fault: What Happens When You Miss?

- Page fault means that page is not resident in memory
- Hardware must detect situation (why? how?), but it cannot remedy the situation
- Therefore, hardware must trap to the operating system so that it can remedy the situation
  - Pick a page to discard (may write it to disk)
  - Load the page in from disk
  - Update the page table
  - Resume to program so HW will retry and succeed!

What can HW do to help the OS?
Handling Page Faults

- OS must know where to find the page
  - Create space on disk for all pages of process (swap space)
  - Use a data structure to record where each valid page is on disk (may be part of page table)
  - Use another data structure to track which process and virtual addresses use each physical page
    => for replacement purpose

How to determine which frame to replace?
=> LRU policy
How to keep track of LRU?
Handling Page Faults

Fig. 7.22
Page Replacement: 1-bit LRU

- Associated with each page is a reference flag:
  ref flag = 1 if page has been referenced in recent past
  = 0 otherwise

- If replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past

<table>
<thead>
<tr>
<th>dirty</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Or search for a page that is both not recently referenced AND not dirty

page fault handler:

- last replaced pointer (lrp)
- If replacement is to take place, advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE's have their reference bits set to zero.

Architecture part: support dirty and used bits in the page table (how?)

=> may need to update PTE on any instruction fetch, load, store
Impact of Paging (I)

- Page table occupies storage
  32-bit VA, 4KB page, 4bytes/entry
  => $2^{20}$ PTE, 4MB table

- Possible solutions:
  - Use bounds register to limit table size; add more if exceed
  - Let pages to grow in both directions
    => 2 tables, 2 limit registers, one for hash, one for stack
  - Use hashing => page table same size as physical pages
  - Multiple levels of page tables
  - Paged page table (page table resides in virtual space)
Hashing: Inverted Page Tables

- 28-bit virtual address
- 4 KB per page, and 4 bytes per page-table entry
  - Page table size: $64 \text{ K (pages)} \times 4 = 256 \text{ KB}$
  - Inverted page table:
    - Let the # of physical frames = $64 \text{ MB} = 16 \text{ K (frames)}$
    - $16 \text{ KB} \times 4 = 64 \text{ KB}$
Two-level Page Tables

32-bit address:

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 index</td>
<td>P2 index</td>
<td>page offset</td>
</tr>
</tbody>
</table>

- 4 GB virtual address space
- 4 KB of PTE1
  (Each entry indicate if any page in the segment is allocated)
- 4 MB of PTE2
  - paged, holes

What about a 48-64 bit address space?
Impact of Paging (II)

- Each memory operation (instruction fetch, load, store) requires a page-table access!
  - Basically double number of memory operations
Making Address Translation Practical

- In VM, memory acts like a cache for disk
  - Page table maps virtual page numbers to physical frames
  - Use a page table cache for recent translation
    => *Translation Lookaside Buffer (TLB)*
Translation Lookaside Buffer

Fig. 7.23
Translation Lookaside Buffer

• Typical RISC processors have memory management unit (MMU) which includes TLB and does page table lookup
  – TLB can be organized as fully associative, set associative, or direct mapped
  – TLBs are small, typically < 128 - 256 entries
    • Fully associative on high-end machines, small n-way set associative on mid-range machines

• TLB hit on write:
  – Toggle dirty bit (write back to page table on replacement)

• TLB miss:
  – If only TLB miss => load PTE into TLB (SW or HW?)
  – If page fault also => OS exception
TLB of MIPS R2000

- 4KB pages, 32-bit VA
  => virtual page number: 20 bits
- TLB organization:
  - 64 entries, fully assoc., serve instruction and data
  - 64-bit/entry (20-bit tag, 20-bit physical page number, valid, dirty)
- On TLB miss:
  - Hardware saves page number to a special register and generates an exception
  - TLB miss routine finds PTE, uses a special set of system instructions to load physical addr into TLB
- Write requests must check a write access bit in TLB to see if it has permit to write
  => if not, an exception occurs
**TLB in Pipeline**

- **MIPS R3000 Pipeline:**

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
</tr>
</tbody>
</table>

- TLB: 64 entry, on-chip, fully associative, software TLB fault handler
- Virtual address space:

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

  - 0xx User segment (caching based on PT/TLB entry)
  - 100 Kernel physical space, cached
  - 101 Kernel physical space, uncached
  - 11x Kernel virtual space

  Allows context switching among 64 user processes without TLB flush
Integrating TLB and Cache

Fig. 7.24
A reference may miss in all 3 components: TLB, VM, cache.
## Possible Combinations of Events

<table>
<thead>
<tr>
<th>Cache</th>
<th>TLB</th>
<th>Page table</th>
<th>Possible? Conditions?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
<td>Yes; but page table never checked if TLB hits</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
<td>TLB miss, but entry found in page table; after retry, data in cache</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
<td>TLB miss, but entry found in page table; after retry, data miss in cache</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
<td>TLB miss and is followed by a page fault; after retry, data miss in cache</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
<td>impossible; not in TLB if page not in memory</td>
</tr>
<tr>
<td>Hit</td>
<td>Hit</td>
<td>Miss</td>
<td>impossible; not in TLB if page not in memory</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Miss</td>
<td>impossible; not in cache if page not in memory</td>
</tr>
</tbody>
</table>
Virtual Address and Cache

- TLB access is serial with cache access
  - Cache is physically indexed and tagged

- Alternative: virtually addressed cache
  - Cache is virtually indexed and virtually tagged
Virtually Addressed Cache

- Require address translation only on miss!
- Problem:
  - Same virtual addresses (different processes) map to different physical addresses: tag + process id
  - *Synonym/alias problem:* two different virtual addresses map to same physical address
    - Two different cache entries holding data for the same physical address!
  - For update: must update all cache entries with same physical address or memory becomes inconsistent
  - Determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits;
  - Or software enforced alias boundary: same least-significant bits of VA & PA > cache size
An Alternative: Virtually Indexed but Physically Tagged (Overlapped Access)

IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag ! = PA)] and TLB hit THEN access memory with the PA from the TLB
ELSE do standard VA translation
Problem with Overlapped Access

- Address bits to index into cache must not change as a result of VA translation
  - Limits to small caches, large page sizes, or high n-way set associativity if want a large cache
  - Ex.: cache is 8K bytes instead of 4K:

  This bit is changed by VA translation, but is needed for cache lookup

  Solutions:
  - go to 8K byte page sizes;
  - go to 2 way set associative cache
Protection with Virtual Memory

- **Protection with VM:**
  - Must protect data of a process from being read or written by another process

- **Supports for protection:**
  - Put page tables in the addressing space of OS
    \[ \Rightarrow \] user process cannot modify its own PT and can only use the storage given by OS
  - Hardware supports: (2 modes: kernel, user)
    - Portion of CPU state can be read but not written by a user process, e.g., mode bit, PT pointer
      - These can be changed in kernel with special instr.
    - CPU from user to kernel: system calls
      From kernel to user: return from exception (RFE)

- **Sharing:** P2 asks OS to create a PTE for a virtual page in P1’s space, pointing to page to be shared
A Common Framework for Memory Hierarchies

- Policies and features that determine how hierarchy functions are similar qualitatively

- Four questions for memory hierarchy:
  - Where can a block be placed in upper level?
    - Block placement: one place (direct mapped), a few places (set associative), or any place (fully associative)
  - How is a block found if it is in the upper level?
    - Block identification: indexing, limited search, full search, lookup table
  - Which block should be replaced on a miss?
    - Block replacement: LRU, random
  - What happens on a write?
    - Write strategy: write through or write back
## Modern Systems

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Pentium Pro</th>
<th>PowerPC 604</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address</td>
<td>32 bits</td>
<td>52 bits</td>
</tr>
<tr>
<td>Physical address</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>4 KB, 4 MB</td>
<td>4 KB, selectable, and 256 MB</td>
</tr>
<tr>
<td>TLB organization</td>
<td>A TLB for instructions and a TLB for data&lt;br&gt;Both four-way set associative&lt;br&gt;Pseudo-LRU replacement&lt;br&gt;Instruction TLB: 32 entries&lt;br&gt;Data TLB: 64 entries&lt;br&gt;TLB misses handled in hardware</td>
<td>A TLB for instructions and a TLB for data&lt;br&gt;Both two-way set associative&lt;br&gt;LRU replacement&lt;br&gt;Instruction TLB: 128 entries&lt;br&gt;Data TLB: 128 entries&lt;br&gt;TLB misses handled in hardware</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Pentium Pro</th>
<th>PowerPC 604</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache organization</td>
<td>Split instruction and data caches</td>
<td>Split instruction and data caches</td>
</tr>
<tr>
<td>Cache size</td>
<td>8 KB each for instructions/data</td>
<td>16 KB each for instructions/data</td>
</tr>
<tr>
<td>Cache associativity</td>
<td>Four-way set associative</td>
<td>Four-way set associative</td>
</tr>
<tr>
<td>Replacement</td>
<td>Approximated LRU replacement</td>
<td>LRU replacement</td>
</tr>
<tr>
<td>Block size</td>
<td>32 bytes</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Write policy</td>
<td>Write-back</td>
<td>Write-back or write-through</td>
</tr>
</tbody>
</table>
Challenge in Memory Hierarchy

- Every change that potentially improves miss rate can negatively affect overall performance

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effects on miss rate</th>
<th>Possible effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>size ↑</td>
<td>capacity miss ↓</td>
<td>access time ↑</td>
</tr>
<tr>
<td>associativity ↑</td>
<td>conflict miss ↓</td>
<td>access time ↑</td>
</tr>
<tr>
<td>block size ↑</td>
<td>spatial locality ↑</td>
<td>miss penalty ↑</td>
</tr>
</tbody>
</table>

- Trends:
  - Synchronous SRAMs (provide a burst of data)
  - Redesign DRAM chips to provide higher bandwidth or processing
  - Restructure code to increase locality
  - Use prefetching (make cache visible to ISA)
Summary

- Caches, TLBs, Virtual Memory all understood by examining how they deal with four questions:
  1) Where can block be placed?
  2) How is block found?
  3) What block is replaced on miss?
  4) How are writes handled?
- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
Summary (cont.)

- Virtual memory was controversial: Can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to compilers, data structures, algorithms?