Computer Architecture
Memory Hierarchy &
Virtual Memory

Some diagrams from *Computer Organization and Architecture 5th edition* by William Stallings
Memory Hierarchy

- **CPU Registers**
  - 3-10 access/cycle
  - 32-64 words

- **On-Chip Cache**
  - 1-2 access/cycle
  - 5-10 ns  1KB-2MB

- **Off-Chip Cache (SRAM)**
  - 5-20 cycles/access
  - 10-40 ns  1MB – 16MB

- **Main Memory (DRAM)**
  - 20-200 cycles/access
  - 60-120ns
  - 64MB-many GB

- **Disk or Network**
  - 1M-2M cycles/access
  - 4GB – many TB

**Costs**
- $0.137/MB
- $1.30/GB
## Generalized Caches

## Movement of Technology

<table>
<thead>
<tr>
<th>Machine</th>
<th>CPI</th>
<th>Clock (ns)</th>
<th>Main Memory (ns)</th>
<th>Miss Cycles</th>
<th>Penalty / Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX 11/780</td>
<td>10</td>
<td>200</td>
<td>1200</td>
<td>6</td>
<td>0.6</td>
</tr>
<tr>
<td>Alpha 21064</td>
<td>0.5</td>
<td>5</td>
<td>70</td>
<td>14</td>
<td>28</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>0.25</td>
<td>2</td>
<td>60</td>
<td>30</td>
<td>120</td>
</tr>
<tr>
<td>Pentium IV</td>
<td>??</td>
<td>0.5</td>
<td>~5 (DDR, &lt; RMBS)</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>
Cache and Main Memory.

Problem: Is that Main Memory is slow compared to CPU. Solution: Store the most commonly used data in a smaller, faster memory. Good trade off between $$ and performance.
At any time some subset of the Main Memory resides in the Cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache.
CPU generates an address, RA, that it wants to read a word from. If the word is in the cache then it is sent to the CPU. Otherwise, the block that would contain the word is loaded into the cache, and then the word is sent to the processor.
Generalized Caches

Elements of Cache Design

**Cache Size**

**Mapping Function**
- Direct
- Associative
- Set Associative

**Replacement Algorithm**
- Least recently used (LRU)
- First in first out (FIFO)
- Least frequently used (LFU)
- Random

**Write Policy**
- Write through
- Write back

**Line Size**

**Number of caches**
- Single or two level
- Unified or split
**Mapping Function**

**Direct Mapping** – map each block of main memory into only one possible cache line.

\[ i = j \mod m \]

Where

- \( i \) = cache line number
- \( j \) = main memory block number
- \( m \) = number of lines in the cache
**Mapping Function**

"Fully Associative"

**Associative Mapping** – more flexible than direct because it permits each main memory block to be loaded into any line of the cache. Makes it much more complex though.
Set Associative Mapping – compromise that has the pros of both direct and associative while reducing their disadvantages.

\[ m = v \times k \], where the cache is divided into \( v \) sets, each of which contain \( k \) lines, given us the cache, \( m \).

\[ i = j \mod v \], where \( i \) = cache set number, \( j \) = main memory block number, \( m \) = number of lines in the cache.
Replacement Algorithms

**Least Recently Used (LRU)** – probably the most effective. Replace the line in the cache that has been in the cache the longest with no reference to it.

**First-In-First-Out (FIFO)** – replace the block that has been in the cache the longest. Easy to implement.

**Least Frequently Used (LFU)** – replace the block that has had the least references. Requires a counter for each cache line.

**Random** – just randomly replace a line in the cache. Studies show this gives only slightly worse performance than the above ones.
Write Policy

Before a block resides in the cache can be replaced, you need to determine if it has been altered in the cache but not in the main memory. If so, you must write the cache line back to main memory before replacing it.

Write Through – the simplest technique. All write operations are made to main memory as well as to the cache, ensuring that memory is always up-to-date. Cons: Generates a lot of memory traffic.

Write Back – minimizes memory writes. Updates are only made to the cache. Only when the block is replaced is it written back to main memory. Cons: I/O modules must go through the cache or risk getting stale memory.
**Example Cache Organizations**

**Cache Structure**
Has two L1 caches, one for data, one for instructions. The instruction cache is four-way set associative, the data cache is two-way set associative. Sizes range from 8KB to 16KB.

The L2 cache is four-way set associative and ranged in size from 256KB to 1MB.

**Processor Core**
Fetch/decode unit: fetches program instructions in order from L1 instruction cache, decodes these into micro-operations, and stores the results in the instruction pool.

Instruction pool: current set of instructions to execute.

Dispatch/execute unit: schedules execution of micro-operations subject to data dependencies and resource availability.

Retire unit: determines when to write values back to registers or to the L1 cache. Removes instructions from the pool after committing the results.
Cache Structure

L1 caches are eight-way set associative. The L2 cache is a two-way set associative cache with 256KB, 512KB, or 1MB of memory.

Processor Core

Two integer arithmetic and logic units which may execute in parallel. Floating point unit with its own registers. Data cache feeds both the integer and floating point operations via a load/store unit.
Example cache: Alpha 21064

- 8 KB cache. With 34-bit addressing.
- 256-bit lines (32 bytes)
- Block placement: Direct map
  - One possible place for each address
  - Multiple addresses for each possible place

033

Offset

Cache Index

Tag

Offset

- Cache line include…
  - tag
  - data
Cache Example: 21064

FIGURE 5.5 The organization of the data cache in the Alpha AXP 21064 microprocessor.
Cache operation

- Send address to cache
- Parse address into offset, index, and tag
- Decode index into a line of the cache, prepare cache for reading (precharge)
- Read line of cache: valid, tag, data
- Compare tag with tag field of address
- Miss if no match
- Select word according to byte offset and read or write

If there is a miss...

- Stall the processor while reading in line from the next level of hierarchy
  - Which in turn may miss and read from main memory
  - Which in turn may miss and read from disk
Virtual Memory

<table>
<thead>
<tr>
<th>Block</th>
<th>1KB-16KB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit</td>
<td>20-200 Cycles</td>
<td>DRAM access</td>
</tr>
<tr>
<td>Miss</td>
<td>700,000-6,000,000 cycles</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Miss rate</td>
<td>1:0.1 – 10 million</td>
<td></td>
</tr>
</tbody>
</table>

Differences from cache

- Implement miss strategy in software
- Hit/Miss factor 10,000+ (vs 10-20 for cache)
- Critical concerns are
  - Fast address translation
  - Miss ratio as low as possible without ideal knowledge
Virtual Memory

Q0: Fetch strategy
  • Swap pages on task switch
  • May pre-fetch next page if extra transfer time is only issue
  • may include a disk cache

Q1: Block Placement
  • Anywhere – fully associate – random access is easily available, and time to place a block well is tiny compared to miss penalty.
Q2: Finding a block

- Page table
  - List of VPNs (Virtual Page Numbers) and physical address (or disk location)
  - Consider 32-bit VA, 30-bit PA, and 2 KB pages.
    - Page table has $2^{32}/2^{11} = 2^{21}$ entries for perhaps $2^{25}$ bytes or $2^{14}$ pages.
    - Page table must be in virtual memory (segmented paging)
    - System page table must always be in memory.
  - Translation look-aside buffer (TLB)
    - Cache of address translations
    - Hit in 1 cycle (no stalls in pipeline)
    - Miss results in page table access (which could lead to page fault). Perhaps 10-100 OS instructions.
Q3: Page replacement
- LRU used most often (really, approximations of LRU with a fixed time window).
- TLB will support determining what translations have been used.

Q4: Write policy
Write through or write back?

Write Through – data is written to both the block in the cache and to the block in lower level memory.
Write Back – data is written only to the block in the cache, only written to lower level when replaced.
Memory protection

- Must index page table entries by PID
- Flush TLB on task switch
- Verify access to page before loading into TLB
- Provide OS access to all memory, physical, and virtual
- Provide some un-translated addresses to OS for I/O buffers
TLB – typically

- 8-32 entries
- Set-associative or fully associative
- Random or LRU replacement
- Two or more ports (instruction and data)
Summary

• What is the deal with memory hierarchies? Why bother?

• Why are the caches so small? Why not make them larger?

• Do I have to worry about any of this when I am writing code?