The Memory Hierarchy

**The BIG Picture**

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching

- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy
Block Placement

- **Determined by associativity**
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location

- **Higher associativity reduces miss rate**
  - Increases complexity, cost, and access time
Finding a Block

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Location method</th>
<th>Tag comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>n-way set associative</td>
<td>Set index, then search entries within the set</td>
<td>n</td>
</tr>
<tr>
<td>Fully associative</td>
<td>Search all entries</td>
<td>#entries</td>
</tr>
<tr>
<td></td>
<td>Full lookup table</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Hardware caches**
  - Reduce comparisons to reduce cost

- **Virtual memory**
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate
Replacement

- **Choice of entry to replace on a miss**
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement

- **Virtual memory**
  - LRU approximation with hardware support
Write Policy

- **Write-through**
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer

- **Write-back**
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state

- **Virtual memory**
  - Only write-back is feasible, given disk write latency
Sources of Misses (the “3 Cs”)

- **Compulsory misses (aka cold start misses)**
  - First access to a block

- **Capacity misses**
  - Due to finite cache size
  - A replaced block is later accessed again

- **Conflict misses (aka collision misses)**
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size
# Cache Design Trade-offs

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effect on miss rate</th>
<th>Negative performance effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase cache size</td>
<td>Decrease capacity misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase associativity</td>
<td>Decrease conflict misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase block size</td>
<td>Decrease compulsory misses</td>
<td>Increases miss penalty. For very large block size, may increase miss rate due to pollution.</td>
</tr>
</tbody>
</table>
Miss Penalty Reduction

- **Return requested word first**
  - Then back-fill rest of block

- **Non-blocking miss processing**
  - Hit under miss: allow hits to proceed
  - Miss under miss: allow multiple outstanding misses

- **Hardware prefetch: instructions and data**

- **Opteron X4: bank interleaved L1 D-cache**
  - Two concurrent accesses per cycle
Can the programmer improve a program’s cache hit rate?

- Memory access patterns can be influenced by the programmer.

- Example: In C, arrays are stored in “row-major” order, and in Fortran arrays are stored in “column-major” order:

  \[
  \begin{bmatrix}
  1 & 2 & 3 \\
  4 & 5 & 6 \\
  \end{bmatrix}
  \]

  - In C (row-order): In memory: 1,2,3,4,5,6
  - In Fortran (column-order): In memory: 1,4,2,5,3,6
Access memory such that the cache is filled with data you will use next.

- In C: 1,2,3,4,5,6

- A C-program that accesses rows and then columns will have more cache hits (for a bigger data set):

  ```c
  for (i = 0; i < 2; i++)
      for (j = 0; j < 3; j++)
          printf("%d\n", A[i][j]);
  ```

- The first row will be loaded into the cache on the first access, and the next accesses will be cache hits.
Access memory such that the cache is filled with data you will use next

In C: 1,2,3,4,5,6

A C-program that accesses columns and then rows will have more cache misses (for a bigger data set):

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

for (j = 0; j < 3; j++)
for (i = 0; i < 2; i++)
printf("%d\n", A[i][j]);

The first row will be loaded into the cache on the first access, but the second row will be needed immediately. The cache blocks will be replaced frequently, leading to more misses.
Example: Matrix Multiply (MATSIZE=768)

\[
AB = \begin{bmatrix}
a & b \\
c & d \\
\end{bmatrix} \begin{bmatrix}
e & f \\
g & h \\
\end{bmatrix} = \begin{bmatrix}
ae + bg & af + bh \\
ce + dg & cf + dh \\
\end{bmatrix}
\]

- **Program 1:** Time =

```c
for(i=0;i<MATSIZE;i++)
    for(j=0;j<MATSIZE;j++)
        for(k=0;k<MATSIZE;k++)
            C[i][j] += A[i][k] * B[k][j];
```

- **Program 2:** Time =

```c
for(j=0;j<MATSIZE;j++)
    for(j=0;j<MATSIZE;j++)
        for(i=0;i<MATSIZE;i++)
            C[i][j] += A[i][k] * B[k][j];
```
Virtual Machines

- **Host computer emulates guest operating system and machine resources**
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources

- **Virtualization has some performance impact**
  - Feasible with modern high-performance computers

- **Examples**
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC
Virtual Machine History

- It all started in the early 1960s, at IBM...

The IBM “Stretch:”

The first transistorized Computer

“Batch” processing for one thing at a time
Virtual Machine History

- In 1963, MIT proposed “Project MAC” (Mathematics and Computation): a computer that could run more than one program at a time. AT&T Bell Labs wanted a similar system for research.

- IBM responded with the CP-40 research computer, including the first virtual machine environment. The CP-40 could support 14 simultaneous virtual machines, which each ran privileged instructions and were able to catch exceptions.
Virtual Machine History

- The CP-40 evolved into the CP-67, which was the first commercial Main Frame to support virtualization.

- The CP-67 ran an operating system called “CP/CMS” – CP (“Control Program”) created virtual machines that ran CMS (“Console Monitor System”), which was a small single-user OS that was user-active: the first time a user could interact with a program while it was running (before, you ran the program, and got output).
Virtual Machine History

- IBM’s System/360 model mainframe computer originally ran CP-67 (in 1964), and the model was one of IBM’s most successful mainframe computers. It continued to be produced into the late 1970s.

- Subsequent systems maintained backwards compatibility, and IBM mainframes today can still run System/360 code from fifty years ago.
Virtual Machine History

- The CP approach was very forward looking: each user could have their own complete operating system, effectively giving each user their own computer.

- The benefits of virtual machines over time sharing was that VMs could share resources, instead of having resources split equally. There was also better security: users ran an independent OS, and could not interfere (or crash) other systems.
Virtual Machine History

- Where do multi-user single operating systems fit into the picture (e.g., Unix?)

- Unix provides virtualization at the User or Workspace level: users share resources, but have their own profiles. Unix was also one of the first systems to encourage portability between systems: both the OS and most user programs were written in C, meaning that they were mostly portable across computing platforms.
Virtual Machine History

- Even though C programs allowed programs to run on multiple platforms, they still needed to be compiled for each platform.

- In the early 1990s, Sun Microsystems started working on a project that would move away from the dependence of compiling programs for each processor: this became Java, which was released in 1996.

- Java allows a “write-once, run-anywhere” workflow: applications are compiled into Java Byte Code, which is then run in by a “Just in Time” (JIT) compiler on a particular processor.
Virtual Machine History

- The Java Run-Time Environment (JRE) is responsible for providing the JIT capabilities. It has been enhanced over time to improve performance (which was terrible at first).
  - E.g., the first time a program is run, the JIT starts the compilation process to machine code.
  - The machine code is saved on disk.
  - When the program is run subsequently, the machine code is used again, so the JIT doesn’t need to take the time to compile the Java Byte Code.

- A significant part of the JRE is the “Java Virtual Machine,” which is a tiny OS that runs your code.
Virtual Machine History

**The Java Virtual Machine**

- The JVM interprets (or compiles) byte code and converts the byte code into processor-specific machine code.

- The JVM verifies all bytecode before it is run:
  - Branches must be to valid locations
  - Data is always initialized
  - References are for the correct types
  - Access to private data and methods is tightly controlled

- The JVM is considered “safe” because of its strict checking. The safety comes from the virtual machine aspect – nothing runs on the hardware until it has passed through the JVM.
Virtual Machine History

- **SoftPC**
  - In 1987, Unix mainframes were still in wide use (are still are, to some extent). Unix workstations were also prevalent, but they couldn’t run DOS software, which had by then become the dominant OS for desktop systems.
  - Insignia Solutions released “SoftPC” – an emulator that allowed Unix systems to run DOS programs. SoftPC ported this to the Mac OS (which, in the 1990s was running on Motorola hardware, incompatible with x86 hardware). The Mac version also emulated Windows.
Virtual Machine History

- **More Windows Emulators**
  - After SoftPC took off, other competitors came along: Virtual PC (by Connectix) ran Windows on a Mac, and eventually other emulators such as Parallels and VMWare also competed for the same market.

- **User Space -vs- protected mode**
  - Up until this point, all Windows emulators were running in user space, meaning that they could not access OS-privileged instructions, which were available to the main OS, which ran in “protected” mode.
  - “Binary Translation” was used to emulate protected mode, but this was very slow.
Virtual Machine History

- **User Space -vs- protected mode (continued)**
  - Interestingly, the IBM S/370 model allows for native virtualization, but x86 PCs originally did not.
  - In 2005/2006, Intel and AMD created processor instructions for the x86 which allowed for native emulation, and they eventually enhanced this to include Memory Management Unit (MMU) virtualization as well, to support virtual memory schemes.
  - VMWare, VirtualBox, and other x86 emulators use the hardware virtualization to speed up guest operating systems.
Virtual Machine Monitor ("hypervisor")

- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs
- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources
- Guest OS may be different from host OS
- VMM handles real I/O devices
  - Emulates generic virtual I/O devices for guest
Example: Timer Virtualization

- **In native machine, on timer interrupt**
  - OS suspends current process, handles interrupt, selects and resumes next process

- **With Virtual Machine Monitor**
  - VMM suspends current VM, handles interrupt, selects and resumes next VM

- **If a VM requires timer interrupts**
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs
Instruction Set Support

- **User and System modes**

- **Privileged instructions only available in system mode**
  - Trap to system if executed in user mode

- **All physical resources only accessible using privileged instructions**
  - Including page tables, interrupt controls, I/O registers

- **Renaissance of virtualization support**
  - Current ISAs (e.g., x86) adapting
Cache Controller FSM

Idle
- Cache Hit
- Mark Cache Ready
- Valid CPU request

Compare Tag
- If Valid && Hit, Set Valid, SetTag, if Write Set Dirty

Allocate
- Read new block from Memory
- Memory not Ready
- Memory Ready

Write-Back
- Write Old Block to Memory
- Memory not Ready

Compare Tag
- Cache Miss and Old Block is Clean
- Cache Miss and Old Block is Dirty

Allocate
- Memory Ready

Write-Back
- Memory Ready
Consider a Multicore Processor
Suppose two CPU cores share a physical address space

- Write-through caches

<table>
<thead>
<tr>
<th>Time step</th>
<th>Event</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Cache Coherence Protocols

- **Operations performed by caches in multiprocessors to ensure coherence**
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access

- **Snooping protocols**
  - Each cache monitors bus reads/writes

- **Directory-based protocols**
  - Caches and memory record sharing status of blocks in a directory
Invalidating Snooping Protocols

- **Cache gets exclusive access to a block when it is to be written**
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

<table>
<thead>
<tr>
<th>CPU activity</th>
<th>Bus activity</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU A writes 1 to X</td>
<td>Invalidate for X</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CPU B read X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Consistency

- **When are writes seen by other processors?**
  - “Seen” means a read returns the written value
  - Can’t be instantaneous

- **Assumptions**
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses

- **Consequence**
  - P writes X then writes Y
    \[ \Rightarrow \] all processors that see new Y also see new X
  - Processors can reorder reads, but not writes
Multilevel On-Chip Caches

Intel Nehalem 4-core processor

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache. 8MB of L3 is shared between all four cores.
## 2-Level TLB Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Virtual addr</strong></td>
<td>48 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td><strong>Physical addr</strong></td>
<td>44 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td><strong>Page size</strong></td>
<td>4KB, 2/4MB</td>
<td>4KB, 2/4MB</td>
</tr>
</tbody>
</table>
| **L1 TLB (per core)** | L1 I-TLB: 128 entries for small pages, 7 per thread \((2 \times)\) for large pages  
L1 D-TLB: 64 entries for small pages, 32 for large pages  
Both 4-way, LRU replacement | L1 I-TLB: 48 entries  
L1 D-TLB: 48 entries  
Both fully associative, LRU replacement |
| **L2 TLB (per core)** | Single L2 TLB: 512 entries  
4-way, LRU replacement | L2 I-TLB: 512 entries  
L2 D-TLB: 512 entries  
Both 4-way, round-robin LRU |
| **TLB misses**      | Handled in hardware                               | Handled in hardware                                  |
### 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong></td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td>(per core)</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/ allocate,</td>
</tr>
<tr>
<td></td>
<td>allocate</td>
<td>hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified</strong></td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate,</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate,</td>
</tr>
<tr>
<td>cache (per core)</td>
<td>hit time n/a</td>
<td>hit time n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L3 unified</strong></td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
<tr>
<td>cache (shared)</td>
<td>n/a</td>
<td></td>
</tr>
</tbody>
</table>
More Multicore Architecture Examples

2 × quad-core
Intel Xeon e5345
(Clovertown)

2 × quad-core
AMD Opteron X4 2356
(Barcelona)
More Multicore Architecture Examples

2 × oct-core
Sun UltraSPARC T2 5140 (Niagara 2)

2 × oct-core
IBM Cell QS20
Pitfalls

- **Byte vs. word addressing**
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4

- **Ignoring memory system effects when writing or generating code**
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality
Concluding Remarks

- **Fast memories are small, large memories are slow**
  - We really want fast, large memories 😞
  - Caching gives this illusion 😊

- **Principle of locality**
  - Programs use a small part of their memory space frequently

- **Memory hierarchy**
  - L1 cache ↔ L2 cache ↔ ... ↔ DRAM memory ↔ disk

- **Memory system design is critical for multiprocessors**
The “Other” VM: Virtual Machines
Virtual Machines

- **Host computer emulates guest operating system and machine resources**
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources

- **Virtualization has some performance impact**
  - Feasible w/ modern high-performance computers

- **Examples**
  - IBM VM/370 (1970s technology!)
  - VMware
  - Microsoft Virtual PC
Virtual Machine Monitor

- Maps virtual resources to physical resources
  - Memory, I/O devices, CPUs

- Guest code runs on native machine in user mode
  - Traps to VMM on privileged instructions and access to protected resources

- Guest OS may be different from host OS

- VMM handles real I/O devices
  - Emulates generic virtual I/O devices for guest
Example: Timer Virtualization

- **In native machine, on timer interrupt**
  - OS suspends current process, handles interrupt, selects and resumes next process

- **With Virtual Machine Monitor**
  - VMM suspends current VM, handles interrupt, selects and resumes next VM

- **If a VM requires timer interrupts**
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs
Instruction Set Support

- User and System modes
- Privileged instructions only available in system mode
  - Trap to system if executed in user mode
- All physical resources only accessible using privileged instructions
  - Including page tables, interrupt controls, I/O registers
- Renaissance of virtualization support
  - Current ISAs (e.g., x86) adapting