CS3350B Computer Architecture
Memory Hierarchy: Why?

Marc Moreno Maza

http://www.csd.uwo.ca/~moreno/cs3350_moreno/index.html
Department of Computer Science
University of Western Ontario, Canada

Thursday January 10, 2018
Components of a Computer

- Processor
  - Control
  - Datapath
- Memory
- Devices
  - Input
  - Output
- Cache
- Main Memory
- Secondary Memory (Disk)
The “Memory Wall”

- Processor vs DRAM speed disparity continues to grow

Graph showing the performance gap between microprocessors (μProc) and DRAM over time. The gap is labeled as growing at 50% per year for microprocessors and 7% per year for DRAM.
Principles of Locality

- A program is likely to access a relatively small portion of the address space at any instant of time. This program has:
  - **Temporal Locality** (locality in time): If, whenever a memory location is referenced, then this location is likely to be referenced again soon.
  - **Spatial Locality** (locality in space): If, whenever a memory location is referenced, then the locations with nearby addresses are likely to be referenced soon.

- What program structures lead to temporal and spatial locality in code? In data?

**Locality Example:**

- Reference to array elements in succession (stride-1 reference pattern):
  - Spatial locality

- Reference to `sum` each iteration:
  - Temporal locality

```c
sum = 0;
for (i=0; i<n; i++)
    sum += a[i];
return sum;
```
Question: Does this function in C have good locality? If yes, which type?

```c
int sumarrayrows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```
Question: Does this function in C have good locality? If yes, which type?

```c
int sumarraycols(int a[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```
**Question:** Can you permute the loops so that the function scans the 3D array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d(int a[M][N][N]) {
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];

    return sum;
}
```
Some fundamental and enduring properties of hardware and software:

- Fast storage technologies (SRAM) cost more per byte and have less capacity
- Gap between CPU and main memory (DRAM) speed is widening
- Well-written programs tend to exhibit good locality

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a memory hierarchy, to obtain the effect of a large, cheap, fast memory.
CPU looks first for data in L1, then in L2, ..., then in main memory.
Photo of a CPU: Nehalem Die

Memory Controller

Core

Core

Core

Core

Shared L3 Cache

13.6 mm (0.54 inch)

18.9 mm (0.75 inch)
Core Area Breakdown

32KB I$ per core
32KB D$ per core
512KB L2$ per core
Share one 8-MB L3$
## Cache Parameters of Two Processors

<table>
<thead>
<tr>
<th></th>
<th><strong>Intel Nehalem</strong></th>
<th><strong>AMD Barcelona</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 cache size &amp; organization</strong></td>
<td>32KB for each per core; 64B blocks; Split I$ and D$</td>
<td>64KB for each per core; 64B blocks; Split I$ and D$</td>
</tr>
<tr>
<td><strong>L1 associativity</strong></td>
<td>4-way (I), 8-way (D) set assoc.; ~LRU replacement</td>
<td>2-way set assoc.; LRU replacement</td>
</tr>
<tr>
<td><strong>L1 write policy</strong></td>
<td>write-back, write-allocate</td>
<td>write-back, write-allocate</td>
</tr>
<tr>
<td><strong>L2 cache size &amp; organization</strong></td>
<td>256MB (0.25MB) per core; 64B blocks; Unified</td>
<td>512KB (0.5MB) per core; 64B blocks; Unified</td>
</tr>
<tr>
<td><strong>L2 associativity</strong></td>
<td>8-way set assoc.; ~LRU</td>
<td>16-way set assoc.; ~LRU</td>
</tr>
<tr>
<td><strong>L2 write policy</strong></td>
<td>write-back, write-allocate</td>
<td>write-back, write-allocate</td>
</tr>
<tr>
<td><strong>L3 cache size &amp; organization</strong></td>
<td>8192KB (8MB) shared by cores; 64B blocks; Unified</td>
<td>2048KB (2MB) shared by cores; 64B blocks; Unified</td>
</tr>
<tr>
<td><strong>L3 associativity</strong></td>
<td>16-way set assoc.</td>
<td>32-way set assoc.; evict block shared by fewest cores</td>
</tr>
<tr>
<td><strong>L3 write policy</strong></td>
<td>write-back, write-allocate</td>
<td>write-back, write-allocate</td>
</tr>
</tbody>
</table>
Caches

- **Cache**: A small-and-fast storage device that acts as a staging area for a subset of data in a larger-and-slower device.

- **Fundamental idea of a memory hierarchy**:
  - For each $k$, the fast-and-small device at level $k$ serves as cache for the larger-and-slower device at level $k + 1$.

- **Why do memory hierarchies work?**
  - Programs tend to access (thus find) data at level $k$ more often than they access data at level $k + 1$.
  - Thus, storage at level $k + 1$ can be slower, and thus larger and cheaper per bit.
  - **Net effect**: Large pool of memory that costs as little as the cheap storage near the bottom, but that serves data to programs at $\approx$ rate of the fast storage near the top.
Caching in a Memory Hierarchy

- Each level of memory is partitioned into blocks of consecutive bytes and of equal size (which depends on the level).
- The smaller, faster, more expensive storage-device at level $k$ caches a subset of the blocks from level $k + 1$.
- Data is copied between levels in block-sized transfer units.
A Program needs an object \( d \), which is stored in some block \( b \)

**Cache hit (at level \( k \))**
- The Program finds \( b \) in the cache at level \( k \). e.g., block 14

**Cache miss (at level \( k \))**
- \( b \) is not at level \( k \), so the level \( k \) cache must fetch it from level \( k+1 \). e.g., block 12
- If the level \( k \) cache is full, then some current block must be replaced (evicted).
  - Which one is the “victim”?
    - **Placement (mapping) policy**: where can the new block go? e.g., \( b \mod 4 \)
    - **Replacement policy**: which block should be evicted? e.g., LRU (least recently used).
General Caching Concepts

- Types of cache misses:
  - **Cold (compulsory) miss at level** \( k \)
    - A cold miss occurs at level \( k \) for a block \( b \) when this block is missing for the first time in the level \( k \) cache
  - **Conflict miss at level** \( k \)
    - Most caches limit block positions at level \( k \) to a small subset (sometimes a singleton) of the block positions at level \( k + 1 \)
    - e.g. block \( i \) at level \( k + 1 \) must be placed in block \( i \mod 4 \) at level \( k \)
    - Conflict misses occur at level \( k \) when multiple data items from level \( k + 1 \) all map to the same block at level \( k \)
    - e.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time, with a \( i \mapsto i \mod 4 \) mapping
  - **Capacity miss at level** \( k \)
    - Occurs when the set of active blocks (that is, the data set with which the Program is working) is larger than the cache at level \( k \)
More Caching Concepts

- **Hit Rate**: the percentage of memory accesses found in a level of the memory hierarchy

- **Hit Time**: Time to access that level which consists of:
  
  Time to determine hit/miss + Time to access the block.

- **Miss Rate**: the percentage of memory accesses not found in a level of the memory hierarchy, that is, 1 - (Hit Rate).

- **Miss Penalty**: Time to replace a block in that level with the corresponding block from a lower level which consists of:
  
  Time to determine hit/miss
  
  Time to access the block in the lower level
  
  + Time to transmit that block to the level that experienced the miss
  
  + Time to insert the block in that level
  
  + Time to pass the block to the requester

  \[
  \text{Hit Time} \ll \text{Miss Penalty}
  \]
### Examples of Cache Memories in the Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What Cached</th>
<th>Where Cached</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-byte word</td>
<td>CPU registers</td>
<td>0.5</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0.5</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32-byte block</td>
<td>On-Chip L1</td>
<td>0.5</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>32-byte block</td>
<td>On/Off-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB page</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware+OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>

- The TLB (Translation lookaside buffer) stores the recent translations of virtual memory to physical memory and can be called an address-translation cache.
- The Andrew File System (AFS) and Network File System (NFS) are distributed file system protocols.
- A proxy server is a server (a computer system or an application) that acts as an intermediary for requests from clients seeking resources from other servers.
Being able to look at code and get a qualitative sense of its **locality** properties is a key skill for professional programmers.

**Examples of projects driven by data locality** (and other features):

- BLAS (Basic Linear Algebra Subprograms)
  http://www.netlib.org/blas/
- SPIRAL, Software/Hardware Generation for DSP Algorithms
  http://www.spiral.net/
- FFTW, by Matteo Frigo and Steven G. Johnson
  http://www.fftw.org/
- ...
Memory Performance

- **Cache Miss Rate**: number of cache misses/total number of cache references (accesses)
  Miss rate + hit rate = 1.0 (100%)

- **Miss Penalty**: the difference in access time of a given memory level, where a block is missing, and the lower level, where the block is found.

- **Average Memory Access Time (AMAT)** is the average time to access memory considering both hits and misses
  \[ \text{AMAT} = \text{Time for a Hit} + \text{Miss Rate} \times \text{Miss Penalty} \]

  What is the AMAT for a processor with a 200 ps clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?
Memory Performance

- **Cache Miss Rate**: number of cache misses/total number of cache references (accesses)
  Miss rate + hit rate = 1.0 (100%)

- **Miss Penalty**: the difference in access time of a given memory level, where a block is missing, and the lower level, where the block is found.

- **Average Memory Access Time (AMAT)** is the average time to access memory considering both hits and misses
  \[ \text{AMAT} = \text{Time for a Hit} + \text{Miss Rate} \times \text{Miss Penalty} \]

What is the AMAT for a processor with a 200 ps clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

1 + 0.02 * 50 = 2 clock cycles, or 2 * 200 = 400 ps
Assuming that the cache hit costs are included as part of the normal CPU execution cycle, we have:

\[
\text{CPU time} = IC \times \text{CPI} \times CC \\
= IC \times (\text{CPI}_{\text{ideal}} + \text{Average memory stall cycles}) \times CC
\]

\[
\text{CPI}_{\text{stall}}
\]

A simple definition for memory-stall cycles:

Memory-stall cycles = \#accesses/program × miss rate × miss penalty

This ignores extra costs of write misses.
Impacts of Cache Performance

- Relative cache miss penalty increases as processor performance improves (faster clock rate and/or lower CPI). Indeed:
  - Memory speed unlikely to improve as fast as processor cycle time.
  - When calculating CPI_{stall}, the cache miss penalty is measured in processor clock cycles needed to handle a miss.
  - The lower is CPI_{ideal}, the more pronounced is the impact of stalls

- Processor with a CPI_{ideal} of 2, a 100 cycle miss penalty, 36% load/store instr’s, and 2% instruction-cache miss rate, and 4% data-cache miss rate.
  - Memory-stall cycles = 1 × 2% × 100 + 36% × 4% × 100 = 3.44
  - So CPI_{stall} = 2 + 3.44 = 5.44
  - More than twice the CPI_{ideal}!

- What if the CPI_{ideal} is reduced to 1?
- What if the data cache miss rate went up by 1%?
Impacts of Cache Performance

- Relative cache miss penalty increases as processor performance improves (faster clock rate and/or lower CPI). Indeed:
  - Memory speed unlikely to improve as fast as processor cycle time.
  - When calculating $CPI_{stall}$, the cache miss penalty is measured in processor clock cycles needed to handle a miss.
  - The lower is $CPI_{ideal}$, the more pronounced is the impact of stalls
- Processor with a $CPI_{ideal}$ of 2, a 100 cycle miss penalty, 36% load/store instr’s, and 2% instruction-cache miss rate, and 4% data-cache miss rate.
  - Memory-stall cycles $= 1 \times 2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$
  - So $CPI_{stall} = 2 + 3.44 = 5.44$
  - More than twice the $CPI_{ideal}$!
- What if the $CPI_{ideal}$ is reduced to 1?
- What if the data cache miss rate went up by 1%?
  Memory-stall cycles $= 2\% \times 100 + 36\% \times 5\% \times 100 = 3.800$
Multiple Cache Levels

CPU → L1$ (Hit) → L2$ (Hit) → Main Memory

CPU → L1$ (Miss) → L2$ (Miss) → Main Memory

Path of Data Back to CPU
Multiple Cache Levels

With advancing technology, the CPU has more room on die for bigger L1 caches and for second level cache - normally a unified L2 cache (i.e., it holds both instructions and data,) and in some cases even a unified L3 cache.

New AMAT Calculation:

\[
\text{AMAT} = \text{L1 Hit Time} + \text{L1 Miss Rate} \times \text{L1 Miss Penalty},
\]

\[
\text{L1 Miss Penalty} = \text{L2 Hit Time} + \text{L2 Miss Rate} \times \text{L2 Miss Penalty}
\]

and so forth (final miss penalty is Main Memory access time)
1 cycle L1 hit time, 2% L1 miss rate, 
5 cycle L2 hit time, 5% L2 miss rate.
100 cycle main memory access time
Without L2 cache:

With L2 cache:
1 cycle L1 hit time, 2% L1 miss rate,
5 cycle L2 hit time, 5% L2 miss rate.

100 cycle main memory access time

Without L2 cache:
\[
AMAT = 1 + 0.02 \times 100 = 3
\]

With L2 cache:
\[
AMAT = 1 + 0.02 \times (5 + 0.05 \times 100) = 1.2
\]
Wanted: effect of a large, cheap, fast memory

Approach: Memory Hierarchy
- Successively lower levels contain “most used” data from next higher level
- Exploits temporal & spatial locality of programs
- Do the common case fast, worry less about the exceptions (RISC design principle)

Challenges to programmer:
- Develop cache friendly (efficient) programs

From Wikipedia: Reduced instruction set computing, or RISC, is a CPU design strategy based on the insight that a simplified instruction set provides higher performance when combined with a microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction.[1] A computer based on this strategy is a reduced instruction set computer, also called RISC.
C arrays allocated in row-major order
  Each row in contiguous memory locations

Stepping through columns in one row:
  for (i = 0; i < N; i++)
      sum += a[0][i];
  Accesses successive elements of size k bytes
  If block size (B) > k bytes, exploit spatial locality
      Compulsory miss rate = k bytes / B.
  Typically k = 8 and B = 8k or B = 16k.

Stepping through rows in one column:
  for (i = 0; i < n; i++)
      sum += a[i][0];
  Accesses distant elements
  No spatial locality!
      Compulsory miss rate = 1 (i.e. 100%)