CS211
Computer Architecture
The Memory Hierarchy

- Topics
  - Storage technologies and trends
  - Locality of reference
  - Caching in the memory hierarchy
Memory until now...

- We’ve relied on a very simple model of memory for most this class
  - Main Memory is a linear array of bytes that can be accessed given a memory address
  - Also used registers to store values
- Reality is more complex. There is an entire memory system.
  - Different memories exist at different levels of the computer
  - Each vary in their speed, size, and cost
Random-Access Memory (RAM)

- **Key features**
  - RAM is packaged as a chip.
  - Basic storage unit is a cell (one bit per cell).
  - Multiple RAM chips form a memory.

- **Static RAM (SRAM)**
  - Each cell stores bit with a six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.

- **Dynamic RAM (DRAM)**
  - Each cell stores bit with a capacitor and transistor.
  - Value must be refreshed every 10-100 ms.
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.
Memory Modules... real life

- In reality,
- Several DRAM chips are bundled into Memory Modules
  - SIMMS - Single Inline Memory Module
  - DIMMS - Dual Inline Memory Module
- DDR - Dual data Read
  - Reads twice every clock cycle
- Quad Pump: Simultaneous R/W

Source for Pictures: http://en.kioskea.net/contents/pc/ram.php3
SDR, DDR, Quad Pump
Memory speeds

- Processor Speeds: 1 GHz processor speed is 1 nsec cycle time.
- Memory Speeds (50 nsec)

<table>
<thead>
<tr>
<th>DIMM Module Chip Type</th>
<th>Clock Speed[MHz]</th>
<th>Bus Speed[ MHz]</th>
<th>Transfer Rate [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1600 DDR200</td>
<td>100</td>
<td>200</td>
<td>1,600</td>
</tr>
<tr>
<td>PC2100 DDR266</td>
<td>133</td>
<td>266</td>
<td>2,133</td>
</tr>
<tr>
<td>PC2400 DDR300</td>
<td>150</td>
<td>300</td>
<td>2,400</td>
</tr>
</tbody>
</table>

- Access Speed gap
  - Instructions that store or load from memory
Memory Hierarchy (Review)

- **L0:** CPU registers hold words retrieved from L1 cache.
- **L1:** On-chip L1 cache (SRAM)
- **L2:** Off-chip L2 cache (SRAM)
- **L3:** Main memory (DRAM)
- **L4:** Local secondary storage (local disks)
- **L5:** Remote secondary storage (distributed file systems, Web servers)

Storage devices:
- Smaller, faster, and costlier (per byte)
- Larger, slower, and cheaper (per byte)

Storage hierarchy:
- L0: CPU registers
- L1: On-chip L1 cache
- L2: Off-chip L2 cache
- L3: Main memory
- L4: Local secondary storage
- L5: Remote secondary storage

---

Local disks hold files retrieved from disks on remote network servers.

Main memory holds disk blocks retrieved from local disks.

L1 cache holds cache lines retrieved from the L2 cache memory.

L2 cache holds cache lines retrieved from main memory.

CPU registers hold words retrieved from L1 cache.
Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory

- CPU looks first for data in L1, then in L2, then in main memory.

- Typical bus structure:
How to Exploit memory hierarchy

- Availability of memory
  - Cost, size, speed
- Principle of locality
- Memory references are bunched together
  - A small portion of address space is accessed at any given time
- This space in high speed memory
- Problem: not all of it may fit
Types of locality

- **Temporal locality**
  - Tendency to access locations recently referenced

- **Spatial locality**
  - Tendency to reference locations around recently referenced
  - Location $x$, then others will be $x-k$ or $x+k$
Sources of locality

- Temporal locality
  - Code within a loop
  - Same instructions fetched repeatedly

- Spatial locality
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)
What does locality buy?

- Address the gap between CPU speed and RAM speed
- Spatial and temporal locality implies a subset of instructions can fit in high speed memory from time to time
- CPU can access instructions and data from this high speed memory
- Small high speed memory can make computer faster and cheaper
- Speed of 1-20 nsec at cost of $50 to $100 per Mbyte
- This is Caching!!
Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte block.

The small fast L1 cache has room for two 4-word blocks.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

The big slow main memory has room for many 4-word blocks.

The tiny, very fast CPU register file has room for four 4-byte words.
What info. does a cache need

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- Your essentially allowing a **smaller** region of memory to hold data from a **larger** region. Not a 1-1 mapping.

- What kind of information do we need to keep:
  - The actual data
  - Where the data actually comes from
  - If data is even considered valid
Cache Organization

- Map each region of memory to a smaller region of cache
- Discard address bits
  - Discard lower order bits (a)
  - Discard higher order bits (b)
- Cache address size is 4 bits
- Memory address size is 8 bits
- In case of a)
  - 0000xxxx is mapped to 0000 in cache
- In case of b)
  - xxxx0001 is mapped to 0001 in cache
Finding data in cache

- Part of memory address applied to cache
- Remaining is stored as tag in cache
- Lower order bits discarded
- Need to check if 00010011
  - Cache index is 0001
  - Tag is 0011
- If tag matches, hit, use data
- No match, miss, fetch data from memory
General Org of a Cache Memory

Cache is an array of sets.
Each set contains one or more lines.
Each line holds a block of data.

\[ S = 2^s \text{ sets} \]

<table>
<thead>
<tr>
<th>set 0:</th>
<th>set 1:</th>
<th>set S-1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid</td>
<td>tag</td>
<td>0</td>
</tr>
<tr>
<td>valid</td>
<td>tag</td>
<td>0</td>
</tr>
<tr>
<td>valid</td>
<td>tag</td>
<td>0</td>
</tr>
</tbody>
</table>

1 valid bit per line \( \times \) \( t \) tag bits per line \( \times \) \( B = 2^b \) bytes per cache block

\[ E \text{ lines per set} \]

\[ \text{Cache size: } C = B \times E \times S \text{ data bytes} \]
Addressing Caches

Address A:

- **t bits**
- **s bits**
- **b bits**

The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.
Direct-Mapped Cache

- Simplest kind of cache
- Characterized by exactly one line per set.

![Diagram of Direct-Mapped Cache]

```plaintext
set 0: valid | tag | cache block
set 1: valid | tag | cache block
...  
set S-1: valid | tag | cache block
```

$E=1$ lines per set
Accessing Direct-Mapped Caches

- Set selection
  - Use the set index bits to determine the set of interest.
Accessing Direct-Mapped Caches

- Line matching and word selection
  - Line matching: Find a valid line in the selected set with a matching tag
  - Word selection: Then extract the word

(1) If (1) and (2), then cache hit, and block offset selects starting byte.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set Index</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0110</td>
<td>00</td>
</tr>
</tbody>
</table>

(2) The tag bits in the cache line must match the tag bits in the address.

(3) If (1) and (2), then cache hit, and block offset selects starting byte.
Direct-Mapped Cache Simulation

M=4 bit addresses, B=2 bytes/block, 
S=4 sets, E=1 entry/set

Address trace (reads):
0 \[0000_2\], 1 \[0001_2\], 13 \[1101_2\], 8 \[1000_2\], 0 \[0000_2\]

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

0 \[0000_2\] (miss)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
</tbody>
</table>

13 \[1101_2\] (miss)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
</tbody>
</table>

8 \[1000_2\] (miss)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
</tbody>
</table>

0 \[0000_2\] (miss)

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>M[12-13]</td>
</tr>
</tbody>
</table>
Why Use Middle Bits as Index?

<table>
<thead>
<tr>
<th>4-line Cache</th>
<th>High-Order Bit Indexing</th>
<th>Middle-Order Bit Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>01</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>10</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>11</td>
<td>0011</td>
<td>0011</td>
</tr>
</tbody>
</table>

- **High-Order Bit Indexing**
  - Adjacent memory lines would map to same cache entry
  - Poor use of spatial locality

- **Middle-Order Bit Indexing**
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time
Set Associative Caches

- Characterized by more than one line per set

Set 0:
- valid
- tag
- cache block
- valid
- tag
- cache block

Set 1:
- valid
- tag
- cache block
- valid
- tag
- cache block

... 

Set S-1:
- valid
- tag
- cache block
- valid
- tag
- cache block

$E=2$ lines per set
Accessing Set Associative Caches

- Set selection
  - identical to direct-mapped cache

```
set 0:
valid  tag    cache block
valid  tag    cache block
valid  tag    cache block

set 1:
valid  tag    cache block
valid  tag    cache block
valid  tag    cache block

...```

```
set S-1:
valid  tag    cache block
valid  tag    cache block
valid  tag    cache block

...```

Selected set

```
t bits  s bits  b bits
m-1     0 0 0 0 1
```

- tag
- set index
- block offset
Accessing Set Associative Caches

- Line matching and word selection
  - must compare the tag in each valid line in the selected set.

  =1?  (1) The valid bit must be set.

  (2) The tag bits in one of the cache lines must match the tag bits in the address

  (3) If (1) and (2), then cache hit, and block offset selects starting byte.
Accessing Set Associative Caches

- Line matching and word selection
  - must compare the tag in each valid line in the selected set.

1. $\texttt{valid} = 1? \quad (1) \text{ The valid bit must be set.}$

2. $\texttt{tag bits} = ? \quad (2) \text{ The tag bits in one of the cache lines must match the tag bits in the address}$

3. If (1) and (2), then cache hit, and block offset selects starting byte.

Why all of this extra work? Any other issues? Policy?
Fully Associative Caches

- Set selection is trivial.
- Accessing a line is the same as a set associative cache
  - Difference in scale

```
+---+    +---+    +---+
| v |    | t |    | c |    | b |
+---+    +---+    +---+
| v |    | t |    | c |    | b |
+---+    +---+    +---+
| v |    | t |    | c |    | b |
+---+    +---+    +---+
| v |    | t |    | c |    | b |
+---+    +---+    +---+
| v |    | t |    | c |    | b |
+---+    +---+    +---+
| v |    | t |    | c |    | b |
+---+    +---+    +---+
```

set 0:

...
Example: Direct mapped cache

- 32 bit address, 64KB cache, 32 byte block
- How many sets, how many bits for the tag, how many bits for the offset?
Example: 2-way associative cache

- 32 bit address, 64KB cache, 32 byte block
- How many sets (lines), how many bits for the tag, how many bits for the offset?

![Diagram of cache organization]

- $E=2$ lines per set

- Set 0:
  - Valid
  - Tag
  - Cache block

- Set 1:
  - Valid
  - Tag
  - Cache block

- Set S-1:
  - Valid
  - Tag
  - Cache block
Example: 2-way associative cache

- 32 bit address, 32KB cache, 16 byte block
- How many sets, how many bits for the tag, how many bits for the offset?

![Diagram of 2-way associative cache]

- **set 0:**
  - valid
  - tag
  - cache block

- **set 1:**
  - valid
  - tag
  - cache block

- **set S-1:**
  - valid
  - tag
  - cache block

\[ E=2 \text{ lines per set} \]
Write-through vs write-back

- What to do when an update occurs?
  - **Write-through**: immediately
    - Simple to implement, synchronous write
    - Uniform latency on misses
  - **Write-back**: write when block is replaced
    - Requires additional dirty bit or modified bit
    - Asynchronous writes
    - Non-uniform miss latency
    - Clean miss: read from lower level
    - Dirty miss: write to lower level and read (fill)
**Writes and Cache**

- Reading information from a cache is straightforward.
- What about writing?
  - What if you’re writing data that is already cached (**write-hit**)?
  - What if the data is not in the cache (**write-miss**)?
- Dealing with a write-hit.
  - **Write-through** - immediately write data back to memory
  - **Write-back** - defer the write to memory for as long as possible
- Dealing with a write-miss.
  - **write-allocate** - load the block into memory and update
  - **no-write-allocate** - writes directly to memory
- Benefits? Disadvantages?
  - **Write-through** are typically **no-write-allocate**.
  - **Write-back** are typically **write-allocate**.
Multi-Level Caches

- Options: separate data and instruction caches, or a unified cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Speed</th>
<th>Cost/MB</th>
<th>Line Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 B</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>8 B</td>
<td>larger, slower, cheaper</td>
</tr>
<tr>
<td>8-64 KB</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>32 B</td>
<td></td>
</tr>
<tr>
<td>1-4MB SRAM</td>
<td>6 ns</td>
<td>$1.50/MB</td>
<td>32 B</td>
<td></td>
</tr>
<tr>
<td>128 MB DRAM</td>
<td>60 ns</td>
<td>$1.50/MB</td>
<td>8 KB</td>
<td></td>
</tr>
<tr>
<td>30 GB</td>
<td>8 ms</td>
<td>$0.05/MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Intel Pentium Cache Hierarchy

- **L1 Data**
  - 1 cycle latency
  - 16 KB
  - 4-way assoc
  - Write-through
  - 32B lines

- **L1 Instruction**
  - 16 KB, 4-way
  - 32B lines

- **L2 Unified**
  - 128KB--2 MB
  - 4-way assoc
  - Write-back
  - Write allocate
  - 32B lines

- **Main Memory**
  - Up to 4GB

**Processor Chip**

- **Regs.**
Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses/references)
  - Typical numbers:
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.

- Hit Time
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
  - Typical numbers:
    - 1 clock cycle for L1
    - 3-8 clock cycles for L2

- Miss Penalty
  - Additional time required because of a miss
    - Typically 25-100 cycles for main memory
Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```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;
}

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;
}
```

Miss rate = $\frac{1}{4} = 25\%$

Miss rate = $100\%$
Matrix Multiplication Example

- Major Cache Effects to Consider
  - Total cache size
  - Exploit temporal locality and blocking
  - Block size
    - Exploit spatial locality

- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```
Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Line size = 32BYTES (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows

- **Analysis Method:**
  - Look at access pattern of inner loop
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
- if block size (B) > 4 bytes, exploit spatial locality
  - compulsory miss rate = 4 bytes / B

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%)
Matrix Multiplication (ijk)

/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

- Misses per Inner Loop Iteration:
  - (i,*) Misses:
    - A: 0.25
    - B: 1.0
    - C: 0.0

- Inner loop:
  - Row-wise (A)
  - Column-wise (B)
  - Fixed (C)
Matrix Multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}
```

- Misses per Inner Loop Iteration:
  - A: 0.25
  - B: 1.0
  - C: 0.0
Matrix Multiplication (kij)

/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

• Misses per Inner Loop Iteration:
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Inner loop:

(A, *)  (k, *)  (i, *)

Fixed    Row-wise    Row-wise
Matrix Multiplication (ikj)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

- Misses per Inner Loop Iteration:
  - A: 0.0
  - B: 0.25
  - C: 0.25
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

- **Misses per Inner Loop Iteration:**
  - A: 1.0
  - B: 0.0
  - C: 1.0
Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

- Misses per Inner Loop Iteration:
  - A: 1.0
  - B: 0.0
  - C: 1.0

Inner loop:
- Column-wise
- Fixed
- Column-wise

A → B → C
# Summary of Matrix Multiplication

**ijk (amp; jik):**
- 2 loads, 0 stores
- misses/iter = 1.25

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
}
```

**kij (amp; ikj):**
- 2 loads, 1 store
- misses/iter = 0.5

```c
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++) {
            c[i][j] += r * b[k][j];
        }
    }
}
```

**jki (amp; kji):**
- 2 loads, 1 store
- misses/iter = 2.0

```c
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++) {
            c[i][j] += a[i][k] * r;
        }
    }
}
```
Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)