Lecture Topics

- Today: The Memory Hierarchy
  (P&H 5.1-5.11)
- Next: continued

Announcements

- Calendar revised
- Milestone #6 (due 3/16)
- Milestone #7 (due 3/23)
The Memory Hierarchy

- There is limited storage in the processor (registers), so there must be additional memory available

- Ideal scenario: unlimited amount of fast memory to hold instructions and data

- Not practical, so a memory hierarchy is used to provide the illusion of a large amount of fast memory

---

The Memory Hierarchy

- **Top:**
  - fast access
  - small capacity

- **Bottom:**
  - slow access
  - large capacity

- **Illusion:**
  - speed of level 1
  - capacity of level n
The Memory Hierarchy

- Basic model:
  - registers (in processor)
  - cache (closely tied to processor)
  - primary storage (RAM)
  - secondary storage (magnetic disk)

- There may be several levels of cache (with different sizes and speeds)
Locality of Reference

- Observation: a program references only a small portion of its address space at any particular moment in time

- We can exploit locality of reference: keep the currently used instructions and data objects at the top of the memory hierarchy and keep everything else at the bottom of the hierarchy; move items within the hierarchy as needed

Temporal locality (locality in time): if an item is referenced, it is likely to be referenced again in the near future

Spatial locality (locality in space): if an item is referenced, nearby items are likely to be referenced in the near future

Both instructions and data exhibit both types of locality of reference
Locality of Reference

Example code fragment:

```c
i = 0;
while (i < 10)
{
    list[i] = 0;
    i = i + 1;
}
```

Example: C/C++ Program

From www.cse.msu.edu/~cse420/Examples/example01

```c
int sum = 0;

int main()
{
    for( int i = 1; i <= 6; i++)
    {
        sum = sum + i;
    }
}
```
ARM Assembly Language

.global main
.text
main:   push    {lr}
        mov     r0, #1
loop:   cmp     r0, #6
        bgt     end
        ldr     r2, =sum
        ldr     r1, [r2]
        add     r1, r1, r0
        str     r1, [r2]
        add     r0, r0, #1
        b       loop
end:    pop     {lr}
        mov     pc, lr

ARM Machine Language

.global main
.text
0000  E52DE004   main:   push    {lr}
0004  E3A00001   mov     r0, #1
0008  E3500006   loop:   cmp     r0, #6
000c  CA000005   bgt     end
0010  E59F2018   ldr     r2, =sum
0014  E5921000   ldr     r1, [r2]
0018  E0811000   add     r1, r1, r0
001c  E5821000   str     r1, [r2]
0020  E2800001   add     r0, r0, #1
0024  EAFFFFFF7   b       loop
0028  E49DE004   end:    pop     {lr}
002c  E1A0F00E   mov     pc, lr
Execution Trace

<table>
<thead>
<tr>
<th>Time</th>
<th>PC</th>
<th>IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00010800</td>
<td>E52DE004 *</td>
</tr>
<tr>
<td>1</td>
<td>00010804</td>
<td>E3A00001</td>
</tr>
<tr>
<td>2</td>
<td>00010808</td>
<td>E3500006</td>
</tr>
<tr>
<td>3</td>
<td>0001080c</td>
<td>CA000005</td>
</tr>
<tr>
<td>4</td>
<td>00010810</td>
<td>E59F2018 *</td>
</tr>
<tr>
<td>5</td>
<td>00010814</td>
<td>E5921000 *</td>
</tr>
<tr>
<td>6</td>
<td>00010818</td>
<td>E0811000</td>
</tr>
<tr>
<td>7</td>
<td>0001081c</td>
<td>E5821000 *</td>
</tr>
<tr>
<td>8</td>
<td>00010820</td>
<td>E2800001</td>
</tr>
<tr>
<td>9</td>
<td>00010824</td>
<td>EAFFFFF7</td>
</tr>
<tr>
<td>10</td>
<td>00010808</td>
<td>E3500006</td>
</tr>
<tr>
<td>11</td>
<td>0001080c</td>
<td>CA000005</td>
</tr>
</tbody>
</table>

main: push {lr}
mov r0, #1
loop: cmp r0, #6
bgt end
sum

Hierarchy Levels

- Each pair of adjacent levels can be viewed as an upper level (closer to processor) and lower level
  - Upper: faster, smaller
  - Lower: slower, larger
- Data transferred: block (or line)
Hierarchy Levels

- Hit rate: fraction of memory accesses found in the upper level
- Miss rate: fraction of memory accesses not found in the upper level
- Miss rate equals 1.0 - hit rate
- Ex: hit rate of 0.97, miss rate of 0.03

Hit time:
- determine if item is present in upper level
- if present, transfer item to requestor

Miss penalty:
- access block in lower level
- transmit it from lower level to upper level
- place it in the upper level
- transfer item to requestor
Average Memory Access Time

For a two-level memory hierarchy, where M1 is faster than M2, the average memory access time can be calculated using:

\[ \text{AMAT} = H \times T1 + (1-H) \times (T1 + T2) \]

\[ = T1 + (1-H) \times T2 \]

H = hit rate
T1 = access time for M1
T2 = access time for M2

Example

- Assume the following:
  - Clock cycle: 1 ns
  - Hit rate: 0.95
  - Hit time: 1 clock cycle
  - Miss penalty: 20 clock cycles

- Average Memory Access Time (AMAT):
  \[ \text{AMAT} = 1 \text{ ns} + 0.05 \times 20 \text{ ns} \]
  \[ = 2 \text{ ns} \]
Memory Technologies

- Rule of thumb: the larger the capacity, the longer time to access an item

- Main technologies (2012):

<table>
<thead>
<tr>
<th>Technology</th>
<th>Access Time</th>
<th>Cost per GiB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>0.5 to 2.5 ns</td>
<td>$500 to $1000</td>
</tr>
<tr>
<td>DRAM</td>
<td>50 to 70 ns</td>
<td>$10 to $20</td>
</tr>
<tr>
<td>Disk</td>
<td>5,000,000 to 20,000,000 ns</td>
<td>$0.05 to $0.10</td>
</tr>
</tbody>
</table>

Cache Basics

- Level between processor and primary storage (RAM) – introduced because of gap between speed of registers and RAM

- As gap has increased, additional levels introduced:
  - Level 1 (instruction cache and data cache)
  - Level 2
  - Level 3
Cache Basics

- Assume one level of cache (split between instruction cache and data cache)

- Processor checks cache first for desired item (instruction or data); if not present, fetch it from primary storage (RAM)

Cache Basics

- Where can a block be placed?
- How is a block found?
- Which block is replaced on a miss?
- How are writes handed?
Read Operation

Read: copy data from RAM to CPU

Check cache first – if desired item is already present in the cache, simply copy item from cache to CPU

If desired item is not already present in the cache, copy a block (item and its neighbors) from RAM to the cache and copy the item to the CPU

Write Operation

Write: copy data from CPU to RAM

Check cache first – if desired item is already present in the cache, simply copy item from CPU to cache

If desired item is not already present in the cache, copy item (and its neighbors) from RAM to the cache and copy the item from the CPU
Write Policies

After a store instruction, cache and RAM are inconsistent: contents of block in cache and RAM are different

Two strategies:

• Write through
• Write back

Write Policies

- Write through: whenever a cache block is changed, the block is copied to RAM

- Write back: cache block is only written (copied) to RAM when the cache line is evicted (replaced)
  - multiple store instructions can occur before block has to be written to RAM
  - modified bit used to indicate that block has been changed (and must be written to RAM)
Direct Mapped Cache

- With direct mapping, each RAM slot maps to a specific cache slot.
- Since RAM is much larger than cache, multiple RAM slots map to the same cache slot.
- Mapping formula:
  \[(\text{address}) \mod (\text{number of cache slots})\]

Example

- Assume 8 cache slots, 32 RAM slots.
Direct Mapped Cache

- Where can a block be placed?
  - only one possible cache slot

- How is a block found?
  - check valid bit (cache slot contains some block)
  - check tag (part of address) to see if it is the correct block

Example Configuration

- Cache: 64 KB
- RAM: 16 MB (24-bit addresses)
- Block size: 4 bytes
- Cache is organized as $2^{14}$ lines, where each line holds 4 bytes
- RAM is viewed as 4M blocks of 4 bytes each
Example (1)

- Address (24 bits) viewed as three fields:
  - Word: 2 bits to identify byte within word
  - Line: 14 bits to identify cache line
  - Tag: 8 bits (remaining bits)

Example (2)

Address: 16339C

in binary:

```
0101100011001110011100
```

- Tag: 00010110 (16)
- Line: 011100111001111 (0CE7)
- Word: 00 (0)
Example (3)

<table>
<thead>
<tr>
<th>Cache line</th>
<th>Addresses of RAM blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000, 010000, ..., FF0000</td>
</tr>
<tr>
<td>1</td>
<td>000004, 010004, ..., FF0004</td>
</tr>
<tr>
<td>2</td>
<td>000008, 010008, ..., FF0008</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$2^{14} - 1$</td>
<td>00FFFC, 01FFFC, ..., FFFFFFC</td>
</tr>
</tbody>
</table>

Block Size

- To exploit spatial locality, a cache slot must hold more than one item (one word)
- Block size is always a multiple of 2 (use least significant bits of address to identify specific byte within block)
- Typical block sizes are 32 to 256 bytes
Block Size

- Miss rate vs. block size for one benchmark

![Graph showing miss rate vs. block size for different block sizes.](image)

Example

- Address (32 bits) viewed as three fields:
  - Byte offset: 8 bits to identify byte within block
  - Line: 4 bits to identify cache line
  - Tag: 20 bits (remaining bits)

- Example: **FFF7C408**

  ![Binary representation of example address.](image)
Example (2)

- How many lines in the cache?
  \[2^4 = 16 \text{ lines}\]
- How many bytes in one block?
  \[2^8 = 256 \text{ bytes}\]
- How many control bits in one line?
  \[V + M + \text{Tag} = 1 + 1 + 20 = 22 \text{ bits}\]
- How many total bits in one line?
  \[\text{control + data} = 22 + 2048 = 2070 \text{ bits}\]

Example (3)

<table>
<thead>
<tr>
<th>I</th>
<th>V</th>
<th>M</th>
<th>Tag</th>
<th>I</th>
<th>V</th>
<th>M</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
<td>FF641</td>
<td>1 1 0</td>
<td>00014</td>
<td>8 0 0</td>
<td>0004A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>00014</td>
<td>9 1 0</td>
<td>00028</td>
<td>A 1 0</td>
<td>00028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 1 0</td>
<td>0003A</td>
<td>1 1 0</td>
<td>FFF7C</td>
<td>B 1 1</td>
<td>FFF7C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0 1</td>
<td>FFF7C</td>
<td>1 1 0</td>
<td>00014</td>
<td>C 0 1</td>
<td>00EA1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 1</td>
<td>FFF7C</td>
<td>1 1 0</td>
<td>00014</td>
<td>D 1 0</td>
<td>00028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 1 0</td>
<td>00014</td>
<td>E 1 1</td>
<td>0003A</td>
<td>F 1 1</td>
<td>0003A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example (4)

- Index – line number (not stored)
- Valid bit (V) – initially 0, set to 1 when that entry in the cache is in use
- Modified bit (M) – set to 1 when at least one byte in the block has been modified by a "write" operation (aka dirty bit)
- Tag bits – compared to tag bits from address
- Block – 256 bytes (not shown)

Example (5)

- Consider the cache entry at index 4:
  1 1 FFF7C
- What are the addresses of the first and last bytes in that cache entry?
  first byte: FFF7C400
  last byte: FFF7C4FF
- Has the contents of that cache block been modified?
  Yes, M = 1
Example (6)

- Request to read from address **00028A14**
  
  Line in address is A, so check cache line at index A:
  
  1 0 00028
  
  Hit: $V = 1$ and tag in cache line matches tag in address
  
  Transfer 4 bytes (14, 15, 16, 17) from cache block to CPU

Example (7)

- Request to read from address **0007260C**
  
  Line in address is 6, so check index 6:
  
  0 0 00014
  
  Miss: $V = 0$
  
  Transfer 256 bytes from RAM to cache
  
  Set V bit to 1
  
  Set M bit to 0
  
  Set tag to 00072
  
  Transfer 4 bytes (0C, 0D, 0E, 0F) from cache to CPU
Example (8)

- Request to write to address \texttt{0003AED8}

  Line in address is E, so check index E:

  \begin{verbatim}
  1 1 0003A
  \end{verbatim}

  Hit: \(V = 1\) and tag in cache line matches tag in address

  Transfer 4 bytes from CPU to cache (D8, D9, DA, DB)
  Set M bit to 1

- Note: some of the 256 bytes in the cache block are no longer the same as the corresponding bytes in RAM (copy block to RAM later)

Example (9)

- Request to write to address \texttt{0003A344}

  Line in address is 3, so check index 3:

  \begin{verbatim}
  0 1 FF593
  \end{verbatim}

  Miss: \(V = 0\)

  Transfer 256 bytes from RAM to cache
  Set V bit to 1
  Set M bit to 1
  Set tag to \texttt{0003A}
  Transfer 4 bytes (44, 45, 46, 47) from CPU to cache
Example (10)

- Request to read from address \texttt{002C5934}
  
  Line in address is 9, so check index 9:
  \begin{verbatim}
  1 0 00028
  \end{verbatim}
  
  Miss: \texttt{V = 1}, but tags don’t match
  
  Transfer 256 bytes from RAM to cache
  Set \texttt{V} bit to 1
  Set \texttt{M} bit to 0
  Set tag to \texttt{002C5}
  Transfer 4 bytes (34, 35, 36, 37) from cache to CPU

Example (11)

- Request to read from address \texttt{002D1F98}
  
  Line in address is F, so check index F:
  \begin{verbatim}
  1 1 0003A
  \end{verbatim}
  
  Miss: \texttt{V = 1}, but tags don’t match
  
  Transfer 256 bytes from cache to RAM (write back)
  Transfer 256 bytes from RAM to cache
  Set \texttt{V} bit to 1
  Set \texttt{M} bit to 0
  Set tag to \texttt{002D1}
  Transfer 4 bytes (98, 99, 9A, 9B) from cache to CPU