Memory: Cache-Main Memory

CIT 595
Spring 2010

Motivation

- Main memory and registers are only storage CPU can access directly
- Mismatch in processor and memory speed
  - 3 GHz processor can execute an “add” operation in 0.33ns
  - Main Memory (DRAM) speed ~50ns
- Reduce the speed mismatch at the same time have memory that is cost effective

RAM

- RAM
  - Random Access Memory
    - Access time is consistent across all locations and is independent of previous access
  - Volatile

- Types
  - SRAM – Static RAM
    - 1-bit -> cross-coupled NOR gates (1 NOR gate requires 4 transistors)
  - DRAM – Dynamic RAM
    - 1 bit -> 1 capacitor (to store charge) + 1 transistor (to be able to change the state)
    - Denser than SRAM – more bits per unit area
    - Slower in access
      - Need to sense 1 or 0 based on capacitor charge stored
      - Capacitor leaks charge over time, periodic refresh required

Time/Cost/Capacity

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- Magnetic disk (hard drive)
  - 5ms – 20ms, $0.20 – $2 per GB

Ideal memory

- Access time of SRAM
- Capacity and cost/GB of disk

How do I realize the ideal memory?
Memory Hierarchy
- To provide the best performance at the lowest cost
- Small, fast storage elements are kept near the CPU
  - E.g. Registers, Caches (SRAM technology)
- Larger, slower memory is accessed through the data bus
  - Bus: shared communication channel
    - Parallel set of wires for information (addn/data) and synchronized of information transfer
  - Volatile: E.g. Main Memory (DRAM technology)
  - Non-Volatile: E.g. Magnetic Disk Drives, Optical Media (CD, DVD)

Cache
- A cache is a small amount of fast memory
- What makes small fast?
  - Simpler decoding logic
  - More expensive SRAM technology
  - Is close proximity to processor
    - Cache sits between normal main memory and CPU
      - Dedicated connection (not shared unlike for other storage)
      - Modern computers have at least two levels
        - L1 is on CPU chip or module
        - L2 is on the motherboard accessed via interconnect (bus)

Memory Lookup
- Each level of memory keeps a subset of the data contained in the lower memory-level (i.e. from larger memory)
- To access a particular piece of data, the CPU first sends a request to its nearest memory, i.e. cache
- If the data is not found then next level is queried
- Once the data is located at a level, then the data, and a number of its nearby data elements are brought into the cache
  - E.g. if data from address x is requested, then data from address X + 1, X + 2, etc. is also sent
  - A block (data from multiple blocks) of data is transferred

Why is a block of data transferred?
- Data between levels is transferred using a bus(bundles of wire)
  - Bus itself takes sometime to transfer data
  - Effective to get some more data you might require in the future during one bus transaction
- So why get data that is nearby?
  - Temporal Locality: referenced memory is likely to be referenced again soon (e.g. code within a loop)
  - Spatial Locality: memory close to referenced memory is likely to be referenced soon (e.g., data in a sequentially access array)
  - Sequential locality: Instructions tend to be accessed sequentially
- The above three are known as Principles of Locality
Basic Terminology

- Memory is divided into **blocks**
- Each block contains **fixed numbers of words**
  - Word = size of data stored in one location e.g. 8 bits, 16 bits etc..
- One block is used as the minimum unit of transfer
- Hence, each location in the cache stores 1 block

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 Word 0</td>
<td>Block 0</td>
</tr>
<tr>
<td>01 Word 1</td>
<td></td>
</tr>
<tr>
<td>10 Word 2</td>
<td>Block 1</td>
</tr>
<tr>
<td>11 Word 3</td>
<td></td>
</tr>
</tbody>
</table>

Address Conversion

- Main memory address generated by the processor cannot be used to access the cache
- Hence a **mapping scheme** is required that converts the generated main memory address into a cache location
- Address Conversion is done by giving special significance to the bits of the main memory address
- The address is split into distinct groups called **fields**
  - The group fields are a way to find:
    - Which cache location?
    - Which word in the block?
    - Whether it is the right data are looking for? Some kind of unique identifier

Mapping Scheme 1: Direct Mapped Cache

- In a direct mapped cache consisting of \( N \) blocks of cache (i.e. \( N \) locations), block \( X \) of main memory maps to cache block \( Y = X \mod N \).
- E.g. if we have 10 blocks of cache, block 7 of cache may hold blocks 7, 17, 27, 37, . . . of main memory.

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 0</td>
<td>Block 1</td>
</tr>
<tr>
<td>Block 2</td>
<td></td>
</tr>
<tr>
<td>Block 3</td>
<td></td>
</tr>
</tbody>
</table>

Direct Mapped Scheme: Address Conversion

<table>
<thead>
<tr>
<th>Tag</th>
<th>Block</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>Word 0</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>Word 2</td>
</tr>
</tbody>
</table>

Word = which word in block?
Block = Which location in Cache?
Tag = unique identifier w.r.t one block

Note: Tag is used to distinguish whether main memory block 7 or 17 is stored in cache block 7
### Cache with 4 blocks and 8 words per block

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example of Direct Mapped Scheme

- Suppose our memory consists of \(2^{14}\) words, and cache has \(16 = 2^4\) blocks, and each block holds 8 words
- Thus main memory is divided into \(2^{14} / 2^3 = 2^{11}\) blocks
- Of the 14 bit address, we need 4 bits for the block field, 3 bits for the word, and the tag is what's left over

![Address bits](image)

### Direct Mapped Cache with 16 blocks

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Calculating Cache Size

- Suppose our memory consists of \(2^{14}\) locations (or words), and cache has \(16 = 2^4\) blocks, and each block holds 8 words
- There are 16 locations in the cache
- Each row has 7 bits for tag + 8 words
- Assume 1 word is 8 bits, the total bits in row \((8 \times 8) + 7 = 71\)
- 71 is approximately 9 bytes
- Cache size = \(16 \times 9\) bytes = 144 bytes
Cache Indexing and Data Retrieval

Block No. | Tag | Data
---|---|---
0 | | 1AA
1 | | 
2 | | 
3 | | 
4 | | 
5 | 0000011 | 
- | | 
13 | | 
14 | | 
15 | | 

Example of Direct Mapped Cache (contd..

- Suppose a program generates the address 1AA
  - In 14-bit binary, this number is: 000001 1010 1010

- The first 7 bits of this address go in the tag field, the next 4 bits go in the block field, and the final 3 bits indicate the word within the block.

Direct Mapped Cache Example (contd..)

- However, if the program generates the address, 3AB
  - 3AB also maps to block 0101, but we will not find data for 3AB in cache
    - Tags will not match i.e. 0000111 (3AB) is not equal to 0000011 (1AB)
    - This is known as cache miss
  - Hence we get it from main memory

- The block loaded for address 1AA would be replaced by the blocks associated with the 3AB reference
## Direct Mapped Cache with address 3AB

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0000111</td>
<td>3AB</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Disadvantage of Direct Mapped Cache
- Suppose a program generates a series of memory references such as: 1AB, 3AB, 1AB, 3AB, ...
  - The cache will continually evict and replace blocks
- The theoretical advantage offered by the cache is lost in this extreme case
- Other cache mapping schemes are designed to prevent this kind of thrashing

## Size of block
- How many words must a block contain i.e. block size?
  - If block size is too small
    - Cannot hold enough data this mostly likely needed in the future
    - Need to get data from main memory frequently
  - If block size is too large
    - Might benefit data if you are using lot of arrays
    - Might not benefit instructions as usually every few instructions a branch might be encountered and then block brought in will be wasted
    - Also, larger the block size, more circuitry is needed for comparison and muxing data words – will slow down cache access

## Valid Cache block
- Determining the block in cache is valid or not?
  - E.g. The cache will be empty and tag fields in each location will be meaningless at start up
    - Thus tag fields must be ignored initially when the cache is starting to fill up
  - For validity, another bit called valid bit is added to the cache indicate whether the block contains valid information
    - 0 – not valid, 1 – valid
    - All blocks at start up would be not valid
    - If data from main memory is got into cache for a particular block, then valid bit for that field is set
    - Valid bit will contribute to the cache size
**Direct Mapped Cache with Valid (V) Field**

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0000111</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Address 3AB referenced for the first time. Entire block is brought into cache block 5.

**Scheme 2: Fully Associative Cache**

- Avoids the collision problem of direct mapped
- Allow a block to *go anywhere* in cache
- This way, cache would have to fill up before any blocks are evicted
- Disadv: have to choose the block to evict when cache is full
- A memory address is partitioned into only two fields: the tag and the word

**Fully Associative Cache: Address Conversion**

- E.g. 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:

  11 bits | 3 bits
  Tag | Word

- When the cache is searched, all tags are searched in parallel to retrieve the data quickly
- This requires evaluation hardware
  - Basically need a comparator circuit for each block in cache
  - Costly: because we need "n" comparators where n = # of blocks in cache

**Scheme 3: Set Associative**

- Combine the ideas of direct mapped cache and fully associative cache
- A set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache
- But that cache location can *hold more than one* main memory block
- The cache location is then called a *set*
  - Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache
Scheme 3: Set Associative Example

- E.g. 2-way set associative cache can be conceptualized as shown in the schematic below
- Each set contains two different memory blocks

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Block 0 of set</th>
<th>Valid</th>
<th>Tag</th>
<th>Block 1 of set</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>Words A, B, C, ...</td>
<td>1</td>
<td>--------</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11110101</td>
<td>Words L, M, N, ...</td>
<td>1</td>
<td>--------</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>--------</td>
<td>0</td>
<td>10111011</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>--------</td>
<td>0</td>
<td>11111100</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- K-way set associate cache will have K blocks per set

K-Set Associative Cache Example

- Suppose we have a main memory of $2^{14}$ locations
- This memory is mapped to a 2-way set associative cache having 16 blocks where each block contains 8 words
- Number of Sets = Number of Blocks in cache/ Blocks per set (K)
- Since this is a 2-way cache, each set consists of 2 blocks, and there are 8 sets i.e. $16/2 = 8$
- Need 3 bits for the set, 3 bits for the word, giving 8 leftover bits for the tag bits

Advantage & Disadvantage Set Associate

- Advantage
  - Unlike direct mapped cache, if an address maps to a set, there choice for placing the new block and evicting an old block

- Disadvantage
  - Tags of each block in a set need to be matched (in parallel) to figure out whether the data is present in cache.
    - Need k comparators
    - Cost for matching is less than fully associative but it is more than direct mapped (need only one comparator)
  - If both slots are filled, then we need an algorithm that will decide which old block to evict (like fully associate)

Replacement Policy: LRU

- Evicts the block that has been unused for the longest period of time
- Goal: keep referenced memory as it is likely to be referenced again soon (locality principle)
- Requires maintaining an access history for each block
  - E.g. 2-way assoc cache requires 1 bit to indicate which line of the two has been reference more recently
  - State bits grow as the set size increase
    - N way mapping there are N! different permutations of use orders
    - It would require $\log_2(N!)$ bits to keep track
  - Reduction in bits by using Psuedo LRU (binary search approach)
    - E.g. 4-way set assoc done with 3 bits
      - 1-bit indicates upper two or lower two block that most recently used.
      - For each half another bit specifies which of the two blocks
**Replacement Policy: Random**
- It picks a block at *random* and replaces it with a new block.
- Can evict a block that will be needed often or needed soon, but it never thrashes.
- In hardware, can generate pseudo-random number by use of Linear Feedback Shift Registers (LFSR).
  - Cascaded registers (n-bit D flops), whose value is updated by linear function (XOR) and shifting bits.
  - More on this later.

**What about blocks that have been written too?**
- While your program is running, it will modify some locations.
- Need to keep main memory and cache *consistent* if we are modifying data.
  - Two options:
    - Update cache and memory at the same time.
    - Update the cache and then main memory at a later time.
  - The two choices are known *Cache Write policies*.

**Cache Write Policies: Write Through**
- Update cache and main memory simultaneously on every write.
  - Advantage
    - Keeps cache main memory consistent at the same time.
  - Disadvantage
    - All writes require main memory access (bus transaction).
    - Slows down the system.
    - This is what we were avoiding in the first place when decided to introduce the cache.

**Cache Write Policies: Write Back or Copy Back**
- Data that is modified is written back to main memory when the cache block is going to be evicted (removed) from cache.
  - Advantage
    - Faster than write-through, time is not spent accessing main memory.
    - Writes to multiple words within a block require only one write to the main-memory.
  - Disadvantage
    - Need extra bit in cache to indicate which block has been modified.
    - Like valid bit, another bit is introduced called *Dirty Bit*, to indicate a modified cache block. 0 – Not Dirty, 1 – Dirty (modified).
Direct Mapped Cache with Valid and Dirty Bit

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>####</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>####</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>####</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Dirty Words within one block

D – Dirty Bit  V- Valid Bit

Locality & Cache Performance

- Programs that exhibit bad locality
  - E.g. Spatial Locality with matrix operations
    - Suppose Matrix data kept in memory is by rows (known as row-major) i.e. offset = row*NUMCOLS + column
  - Poor code:
    - for (j = 0; j < numcols; j++)
      - for(i = 0; i < numrows; i++)
    - i.e. access pattern x[0][0], x[1][0], x[2][0] …
    - Every access is going lead into miss as the array is being accessed by column
    - Solution: switch the for loops
  - C/C++ are row-major, FORTRAN & MATLAB is column-major

Multi-level & Separate I & D Cache

- Multilevel Caches
  - Level1 cache (9KB to 64KB) is situated on the processor itself
  - Level 2 cache (64KB to 2MB) located external to the processor
  - Inclusive cache: same data may be present at multiple levels of cache
  - Strictly inclusive: caches guarantee that all data in a smaller cache also exists at the next higher level
  - Exclusive: caches permit only one copy of the data

- Separate caches for data and instructions
  - Removes structural dependence
  - Better locality

EAT

- Performance of hierarchical memory is measured by its Effective Access Time (EAT)

- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory

- EAT for a two-level memory is given by:
  - EAT = H \times \text{Access Time for Level } i + (1-H) \times \text{Access for Level } i+1
  - H is the hit rate i.e. % time data is found in level i
Example of EAT

Consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.

EAT = 0.99(10ns) + 0.01(200ns) = 9.9ns + 2ns = 11ns