ECE 485/585
Microprocessor System Design

Lecture 9: Cache Architecture
Cache Replacement Policies

Zeshan Chishti
Electrical and Computer Engineering Dept
Maseeh College of Engineering and Computer Science

Source: Lecture based on materials provided by Mark F.
What if...

Cache miss and cache location at Cache Index occupied
- Called a “cache conflict” or “collision”

- Action:
  - “Cast Out” existing entry (“victim”)
  - Replace with new entry

- ..but what if we need that earlier entry again?
  - Thrashing

- Solution – N-way set associative caches
  - Simultaneously hold in cache two (or more) lines that would have been forced to share same place in direct mapped cache
Cache Organization

- How Does The Cache Manage the Cache Lines?
  - Associativity describes how data is stored in the cache
  - Direct Mapped
    - Associativity = 1
    - Each set has single line
    - If it’s in the cache there’s only one place it could be
  - N-way Associativity
    - Each set contains N lines
    - There are N places (“ways”) the line could be
  - Fully associative
    - All cache lines share the same possible places
Direct Mapped Cache

- Example: A $2^N$ byte cache
  - 1 KB Direct-mapped cache with 32 byte lines
  - The upper-most $(32 - N)$ bits are always the cache tag
  - The lowest $M$ bits are the byte select (line size = $2^M$)
  - On cache miss, read in complete line (block)
Set Associative Cache

- N-Way Set Associative
  - N cache entries with same index
  - N direct mapped caches operating in parallel
- Example:
  - 2 way set associative
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result
Comparison

- N-way Set Associative Cache vs. Direct Mapped Cache
  - N comparators vs. 1
  - Extra mux delay for the data
  - Data comes after hit/miss decision and set selection
  - Less thrashing (fewer “conflict” misses)

- In a direct mapped cache, cache line is available before hit/miss detection
  - Possible to assume a hit and continue. Recover later if miss.
An Implementation
Fully Associative Cache

- Fully associative cache
  - Eliminate cache index
  - Compare the cache tags of all cache entries in parallel

Example
- Line size = 32 bytes, requires N 27-bit comparators

![Diagram of 1-bit CAM example]
Cache Misses

- A cache miss occurs when the byte(s) requested by the CPU are **not** in the cache.

- Cache misses are categorized as follows:
  - **Compulsory**
    - First access to a block (reboot, cache cleared)
    - Insignificant if you’re executing trillions of instructions
    - Solution: Nothing you can do about it
  - **Capacity**
    - Cache cannot simultaneously contain all blocks required for good application performance
    - Solution: Increase cache size
  - **Conflict (Collision)**
    - Block was cast out to make room for another block at same location in cache (doesn’t occur for fully associative cache)
    - Solution: Increase associativity
    - Solution: Increase cache size
  - **Coherence (Invalidation)**
    - Other process (I/O, processor) invalidated block
    - Deal with it using additional hardware (MESI, MEI,...)
Which Line is Replaced on Miss?

- Called a **Replacement Strategy** or **Replacement Policy**
- Direct-mapped
  - Easy – only one choice
- Associative
  - Random
  - FIFO (First In First Out)
  - MRU (Most Recently Used)
  - LRU (Least Recently Used)
  - Most commonly used
Least Recently Used (LRU) Policy

Why?
- Temporal locality implies that blocks which were accessed recently are highly probable to be accessed again in near future
- Therefore, replace the block which has gone the longest time without being accessed

Implementation options:
- **Option-1**: Timestamps associated with each block
  - How many bits to encode a timestamp?
  - Must find minimum timestamp on each replacement
- **Option-2**: Sorted list
  - Need $\log_2$ (Associativity) bits per block
  - Must re-sort the list on every access

*True LRU is too expensive to implement*

Simpler (but not entirely accurate) implementations
- Pseudo-LRU and 1-bit LRU
Pseudo-LRU (PLRU)

- Use a binary tree to encode relative age
- Each node records which half of the remaining tree is older/newer
- On a replacement:
  - Follow older pointers to find the LRU victim
- On an access:
  - Flip the nodes along the path from the root to the block

Figure from: Intel® Quark Soc X1000 Developer’s Manual
1-bit LRU

- Store one status bit for each block. We call these MRU bits.
- Each bit is initialized to 0.
- Bit is set to 1, whenever the block is, either (i) brought into the cache, or (ii) accessed. This indicates that this block was “recently used”.
- When a replacement needs to be carried out, all the MRU bits in a set are scanned from left to right and the first block with MRU-bit = 0 is chosen for replacement.
- What if all the MRU bits in a set have become “1” and a replacement needs to be carried out?
  - All the bits are reset to “0” => relative recency information is lost.