Lecture 27
Cache Replacement Policies

CSC208A-Computer Organisation and Architecture

B. Tech. 2016

Course Leader:

Chaitra S.

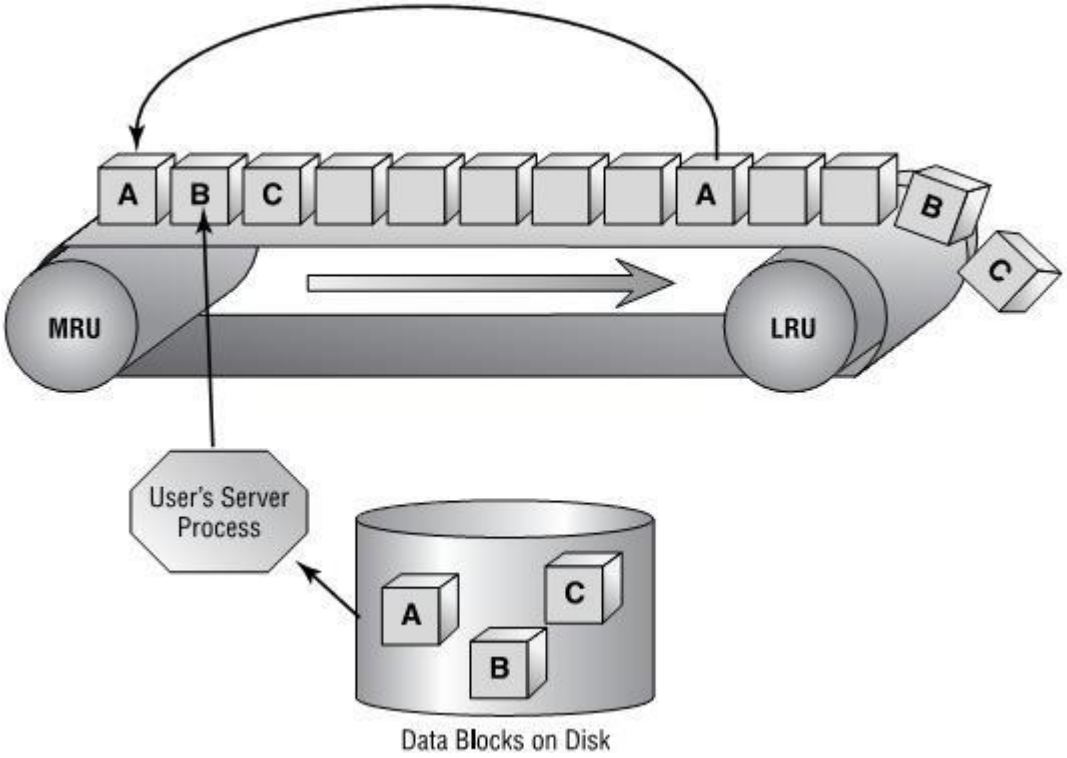
Naveeta

Objectives

- At the end of this lecture, students will be able to
  - Identify the need for having cache replacement policies
  - Describe the majorly used replacement policies with their variants

Contents

- Cache Design
  - Cache Replacement
- Optimal
- Least Recently Used
  - Pseudo LRU
  - True LRU
  - Protected LRU
- Not Recently Used
- Least Frequently Used



Cache Design

- Key issues
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?

- Consider these for caches
  - Usually SRAM

- Also apply to main memory, disks

Cache Replacement

- Cache has finite size
  - What do we do when it is full?

- Analogy: desktop full?
  - Move books to bookshelf to make room
  - Bookshelf full? Move least-used to library
  - Etc.

- Same idea:
  - Move blocks to next level of cache

Cache Miss Rates

- Compulsory miss or Cold miss
  - First-ever reference to a given block of memory
  - Measure: number of misses in an infinite cache model

- Capacity
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced
  - Good replacement policy is crucial!
  - Measure: additional misses in a fully-associative cache

- Conflict
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as capacity within set
  - Good replacement policy is crucial!
  - Measure: additional misses in cache of interest

Comparison

Direct Cache Example:

8 bit tag

14 bit line

2 bit word

Associate Cache Example:

22 bit tag

2 bit word

Set Associate Cache Example:

9 bit tag

13 bit set

2 bit word

Cache Replacement

- When a new block must be brought into the cache and all the positions that it may occupy are full, a decision must be made as to which of the old blocks is to be overwritten. In general, a policy is required to keep the block in cache when they are likely to be referenced in near future. However, it is not easy to determine directly which of the block in the cache are about to be referenced. The property of locality of reference gives some clue to design good replacement policy.

Replacement Algorithm 1

- The replacement policy is the technique we use to determine which line in the cache should be thrown out when we want to put a new block in from memory.

- **For Direct Mapping**
- No choice
- Each block only maps to one line
- Replace that line

Replacement Algorithms 2

- **For Associative & Set Associative**
- Hardware implemented algorithm (speed)
- Least Recently used (LRU)
- e.g. in 2 way set associative
  - Which of the 2 block is lru?
- First in first out (FIFO)
  - replace block that has been in cache longest
- Least frequently used
  - replace block which has had fewest hits
- Random

Replacement Algorithms 2

- **For Associative & Set Associative**
- Algorithm must be implemented in hardware (speed)
- Least Recently used (LRU)
  - e.g. in 2 way set associative, which of the 2 block is LRU?
    - For each slot, have an extra bit, USE. Set to 1 when accessed, set all others to 0.
  - For more than 2-way set associative, need a time stamp for each slot – expensive
- First in first out (FIFO)
  - Replace block that has been in cache longest
  - Easy to implement as a circular buffer
- Least frequently used (LFU)
  - Replace block which has had fewest hits
  - Need a counter to sum number of hits
- Random
  - Almost as good as LFU and simple to implement

Cont..

- **Least Recently Used (LRU) Replacement policy:**
- Since program usually stay in localized areas for reasonable periods of time, it can be assumed that there is a high probability that blocks which have been referenced recently will also be referenced in the near future. Therefore, when a block is to be overwritten, it is a good decision to overwrite the one that has gone for longest time without being referenced. This is defined as the least recently used (LRU) block. Keeping track of LRU block must be done as computation proceeds.

Cont..

- **First In First Out (FIFO) replacement policy:**
- A reasonable rule may be to remove the oldest from a full set when a new block must be brought in. While using this technique, no updation is required when a hit occurs. When a miss occurs and the set is not full, the new block is put into an empty block and the counter values of the occupied block will be increment by one. When a miss occurs and the set is full, the block with highest counter value is replaced by new block and counter is set to 0, counter value of all other blocks of that set is incremented by 1. The overhead of the policy is less, since no updation is required during hit.
- **Random replacement policy:**
- The simplest algorithm is to choose the block to be overwritten at random. Interestingly enough, this simple algorithm has been found to be very effective in practice.

Replacement

- How do we choose victim?

- Many policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - NMRU (not most recently used)
  - NRU
  - Pseudo-random
  - Optimal

Least Recently Used

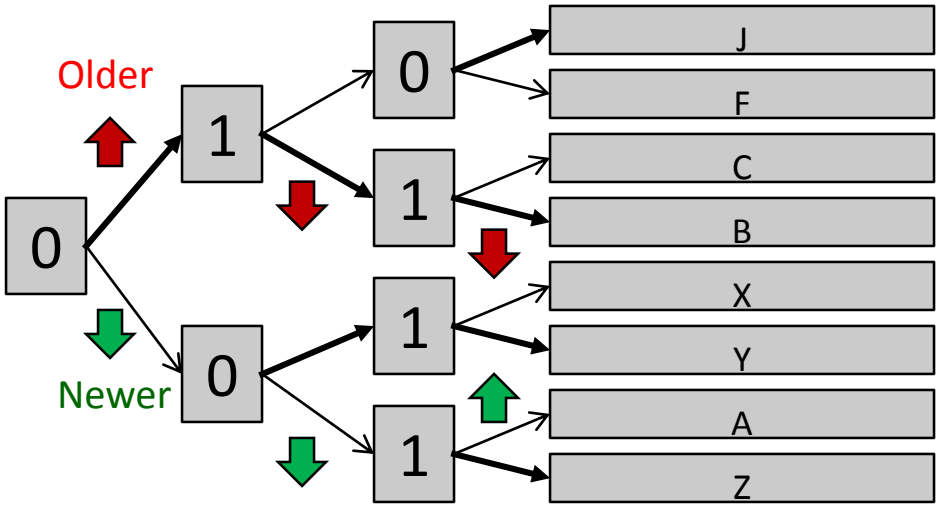
- For a=2, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access

- For a>2, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?

- List overhead: $\log_2(a)$ bits /block
  - Shift register implementation

Pseudo-LRU

- Rather than true LRU, use binary tree
- Each node records which half is older/newer
- Update nodes on each reference
- Follow older pointers to find LRU victim



True LRU Shortcomings

- Streaming data/scans: x0, x1, …, xn
  - Effectively no temporal reuse
- Thrashing: reuse distance > a
  - Temporal reuse exists but LRU fails
- All blocks march from MRU to LRU
  - Other conflicting blocks are pushed out
- For n>a no blocks remain after scan/thrash
  - Incur many conflict misses after scan ends
- Pseudo-LRU sometimes helps a little bit

Protected LRU

- Partition LRU list into filter and reuse lists
- On insert, block goes into filter list
- On reuse (hit), block promoted into reuse list
- Provides scan & some thrash resistance
  - Blocks without reuse get evicted quickly
  - Blocks with reuse are protected from scan/thrash blocks
- No storage overhead, but LRU update slightly more complicated

Not Recently Used

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim (can be pseudo-random, or rotating, or fixed left-to-right)
- Simple, similar to virtual memory clock algorithm
- Provides some scan and thrash resistance
  - Relies on “randomizing” evictions rather than strict LRU order
- Used by Intel Itanium, Sparc T2

Least Frequently Used

- Counter per block, incremented on reference
- Evictions choose lowest count
  - Logic not trivial (a2 comparison/sort)
- Storage overhead
  - 1 bit per block: same as NRU
  - How many bits are helpful?

Summary

- Upper level caches (L1, L2) hide reference stream from lower level caches
- Blocks with “no reuse” @ LLC could be very hot (never evicted from L1/L2)
- Evicting from LLC often causes L1/L2 eviction (due to inclusion)
- Could hurt performance even if LLC miss rate improves