Cache Replacement Policies

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

ECE/CS 752 Spring 2016
Cache Design: Four Key Issues

* These are:
  - 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
## Placement

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Placement</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Anywhere; Int, FP, SPR</td>
<td>Compiler/programmer manages</td>
</tr>
<tr>
<td>Cache (SRAM)</td>
<td>Fixed in H/W</td>
<td>Direct-mapped, set-associative, fully-associative</td>
</tr>
<tr>
<td>DRAM</td>
<td>Anywhere</td>
<td>O/S manages</td>
</tr>
<tr>
<td>Disk</td>
<td>Anywhere</td>
<td>O/S manages</td>
</tr>
</tbody>
</table>
Placement

- **Address Range**
  - Exceeds cache capacity
- **Map address to finite capacity**
  - Called a *hash*
  - Usually just masks high-order bits
- **Direct-mapped**
  - Block can only exist in one location
  - Hash collisions cause problems
Identification

- **Fully-associative**
  - Block can exist anywhere
  - No more hash collisions

- **Identification**
  - How do I know I have the right block?
  - Called a *tag check*
    - Must store address tags
    - Compare against address

- Expensive!
  - Tag & comparator per block
 Placement

- **Set-associative**
  - Block can be in $a$ locations
  - Hash collisions:
    - $a$ still OK

- **Identification**
  - Still perform *tag check*
  - However, only $a$ in parallel
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: 3 C’s [Hill]

- **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*
Replacement

● How do we choose *victim*?
  – Verbs: *Victimize, evict, replace, cast out*

● 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 (yes, really!)
  – Optimal
  – Etc
Optimal Replacement Policy?

[Belady, IBM Systems Journal, 1966]

- Evict block with longest reuse distance
  - i.e. next reference to block is farthest in future
  - Requires knowledge of the future!

- Can’t build it, but can model it with trace
  - Process trace in reverse
  - [Sugumar&Abraham] describe how to do this in one pass over the trace with some lookahead (Cheetah simulator)

- Useful, since it reveals opportunity
  - (X,A,B,C,D,X): LRU 4-way SA $, 2\textsuperscript{nd} X will miss
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
Practical 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
Practical Pseudo-LRU In Action

Partial Order Encoded in Tree:

Z < A  Y < X  B < C  J < F
A > X  C < F
A > F

J Y X Z B C F A

011: PLRU
Block B is here

110: MRU
block is here
Practical Pseudo-LRU

- Binary tree encodes PLRU *partial order*
  - At each level *point* to LRU half of subtree
- Each access: flip nodes along path to block
- Eviction: follow LRU path
- Overhead: \((a-1)/a\) bits per block

Refs: J,Y,X,Z,B,C,F,A

011: PLRU
Block B is here

110: MRU
block is here
True LRU Shortcomings

- Streaming data/scans: $x_0, x_1, \ldots, x_n$
  - 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
Segmented or 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
Protected LRU: LIP

- Simplified variant of this idea: LIP
  - Qureshi et al. ISCA 2007
- Insert new blocks into LRU position, not MRU position
  - Filter list of size 1, reuse list of size (a-1)
- Do this adaptively: DIP
- Use set dueling to decide LIP vs. LRU
  - 1 (or a few) set uses LIP vs. 1 that uses LRU
  - Compare hit rate for sets
  - Set policy for all other sets to match best set
Not Recently Used (NRU)

- 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
RRIP [Jaleel et al. ISCA 2010]

- Re-reference Interval Prediction
- Extends NRU to multiple bits
  - Start in the middle, promote on hit, demote over time
- Can predict *near-immediate*, *intermediate*, and *distant* re-reference
- Low overhead: 2 bits/block
- Static and dynamic variants (like LIP/DIP)
  - Set dueling
Least Frequently Used

- Counter per block, incremented on reference
- Evictions choose lowest count
  - Logic not trivial ($a^2$ comparison/sort)
- Storage overhead
  - 1 bit per block: same as NRU
  - How many bits are helpful?
Pitfall: Cache Filtering Effect

- 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
Cache Replacement Championship

- Held at ISCA 2010
- [http://www.jilp.org/jwac-1](http://www.jilp.org/jwac-1)
- Several variants, improvements
- Simulation infrastructure
  - Implementations for all entries
Recap

- Replacement policies affect *capacity* and *conflict* misses
- Policies covered:
  - Belady’s optimal replacement
  - Least-recently used (LRU)
  - Practical pseudo-LRU (tree LRU)
  - Protected LRU
    - LIP/DIP variant
    - *Set dueling* to dynamically select policy
  - Not-recently-used (NRU) or *clock* algorithm
  - RRIP (re-reference interval prediction)
  - Least frequently used (LFU)
- Contest results
References


References


