EE 156: Advanced Topics in Computer Architecture

Spring 2022
Tufts University

Guest Lecturer: Cesar Gomes

Lecture 2B: Cache Eviction Addendum
(LRU, pLRU and RRIP)

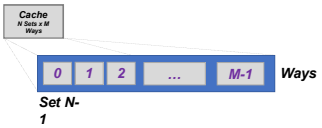
Unit 1: The Memory Hierarchy

Unit 1: The Memory Hierarchy (3-4 weeks)
- **Introduction and Performance Metrics [Chapter 1]**
- **Review of Basic Caches and Set Associativity [Appendix B]**
- Advanced Cache Optimization Techniques and **Replacement policies** [Appendix B, Ch 2]
- Prefetching [SLCA: Falsafi and Wenisch]
- Memory consistency and Cache coherence [Chapter 5]
- Software interfaces and memory consistency
- Transactional memory
- Review of Virtual Memory and TLBs [Appendix B]
- Advanced Virtual Memory [SLCA: Bhattacharjee and Lustig]
- New Non-Volatile Memory (NVM) technologies

- Textbook Reading: Appendix B

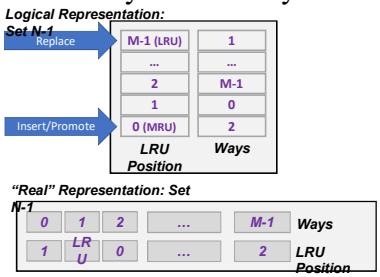
The Problem with Set Associative Caches is…

- Program access patterns don't always fit within sets
- More cache levels "filter" locality and make insertion/promotion/eviction a more difficult problem
- Larger caches are expensive from a chip-area and power perspective, so complex/inefficient policies are often pushed aside
- Solution: Better replacement policies



Common: Least Recently Used Policy

- Maintains Reuse Stack to facilitate set-wise replacement
- Real implementations keep a value with each cache line which is updated on each access to that set
- Closer to Most Recently Used (MRU) Position indicates frequent reuse
- Closer to Least Recently Used (LRU) Position indicates infrequent reuse



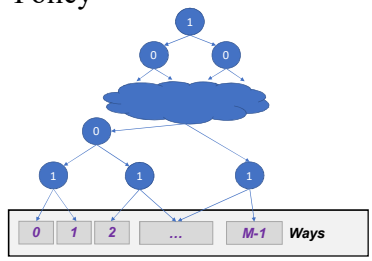
LRU Problems

Assume cache is size 256kB and 8-way set associative

- How many bits does LRU take?
- What percentage of bits are used for LRU?
- What about the logic to implement LRU?
- Trade-off: Does not scale well with higher associativity or capacity since each cache line requires a unique stack position
- Short-coming: does not work well in larger caches closer to main memory since locality filters out applicable reuse patterns; promotions and evictions move non-accessed data closer to LRU without knowledge of the respective reuse

Approximate: Pseudo Least Recently Used Policy

- Approximates LRU Reuse Stack with binary tree to facilitate set-wise replacement
- Real implementations modify replacement logic to represent the "tree;" victim selection
- Each node below the "root" (top) node contains a "subtree" of cache lines.
- Traversal of "pointers" to each subtree is chosen based on the current node value
- Follow the path down the "tree" to find the "LRU" cache line
- Pointer value of 1 indicates movement to the right child node; 0 indicates movement to the left child node
- Promotion is done by changing node values along the path to an updated cache line so that they all point away from the related subtree

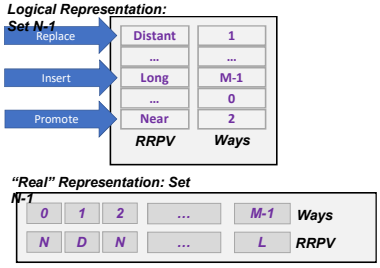


PLRU Problems

- How many bits does PLRU take?
- What percentage of bits are used for PLRU?
- What about the logic to implement PLRU?
- Trade-off: trades chip area for performance given that it only approximates LRU and can have false victim selection
- Short-coming: is still LRU, just has some errors introduced

Advanced: Re-Reference Interval Policy

- Approximates the reuse stack with a re-reference prediction value (RRPV)
- RRPV
  - Near-Immediate: Predicted to be reused soon
  - Long: Predicted to be reused longer than it will exist in cache
  - Distant: Predicted to be reused too far in the future to be useful soon
- Encodes reuse, or the probability that a line is used again
- Adds and decouples frequency, or the number of times a line is used



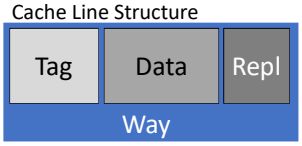
RRIP Problems

- How many bits does RRIP take?
- What percentage of bits are used for RRIP?
- What about the logic to implement RRIP?
- Trade-off: Less state and better performance than LRU; decouples line promotion so other blocks are not effected by individual data reuse
- Short-coming: Patterns fair better with RRIP, but still assumes static insertion, or insertion of line at the same RRPV

RRIP & Cache Relevant Metadata State

```
int NEARIMM   =   0;
int LONG      =   1;
int DISTANT   =   2;
int ASSOC     =   8;
```

Cache Line Structure



RRIP Line Management Policy - Eviction

On any given access, if a requested cache line is not found in the cache ("miss"), the RRIP replacement algorithm executes as follows:

```
int RRIPEviction( int S ):
    int candidate  = ASSOC;
    bool update    = 1;
    While( candidate == ASSOC ):
      for(int w = 0; w < ASSOC; w++ ) {
        if(cache[S][w].RRPV == DISTANT) {
          candidate = w;
          update    = 0;
        }
      }
      if(candidate == ASSOC) {
        for( int w = 0; w < ASSOC; w++) {
          if(cache[S][w].RRPV < DISTANT)
            cache[S][w].RRPV = cache[S][w].RRPV + 1;
        }
      }
    }
    return candidate;
}
```

Line Replacement Search 1

Line Replacement Prediction 1

Line Replacement Pass 2

RRIP Line Management Policy - Eviction

- On any given access, if a requested cache line is not found in the cache ("miss"), the RRIP replacement algorithm checks the RRPV of each cache line in search of one with a "distant" re-reference prediction.
- If one is not found, all RRPVs in the set are incremented by 1.
- This cycle continues until a line with RRPV that represents a "distant" re-reference prediction is found.

Line Replacement Search 1

Line Replacement Prediction 1

Line Replacement Pass 2

RRIP Line Management Policy - Insertion

On an access that results in a miss, new data is placed in a cache line in the following way:

```
void RRIPInsertion(int64 tag, int64 data, int set, int way) {
    cache[set][way].TAG    = tag;
    cache[set][way].DATA   = data;
    cache[set][way].VALID  = 1;
    cache[set][way].RRPV   = LONG;
}
```

Before Line Insertion

After Line Insertion

RRIP Line Management Policy - Insertion

- On an access that results in a miss, new data is placed in a cache line that is predicted to have "distant" reuse-distance (RRPV = 2).
- The RRPV of said cache line is set to reflect a "long" reuse-distance prediction (RRPV = 1)

Before Line Insertion

After Line Insertion

RRIP Line Promotion

On any given access, if a requested cache line is found in the cache ("hit") the cache line RRPV is updated in the following manner:

```
void RRIPPromotion(int set, int way) {
 cache[set][way].RRPV= NEARIMM;
}
```

Before Line Promotion

After Line Promotion

RRIP Line Management Policy - Promotion

On any given access, if a requested cache line is found in the cache ("hit") the cache line RRPV is updated to reflect a near-immediate reuse-distance prediction (RRPV = 0).

Summary

- Set associative caches need better replacement algorithms to better handle diverse access patterns
- LRU alone does a poor job of this by coupling reuse of a single block to reuse of every block in a set
- PLRU is the same as LRU but with errors introduced
- RRIP decouples line dependency, reduces overhead, encodes reuse and frequency, and improves performance
- Is there more to it? Let's discuss this in the paper reading!

Appendix

- Reuse is abstractly if something comes into cache, would it be used again?
- Frequency is abstractly how often it would be reused
- Depends on how/how often other lines are reused/evicted

- Why do we care a/b reuse? Approx. very large/fast memory
- What are the touch points?
  - Approx. very large/fast memory
  - Point of replacement is to prioritize
  - Abstract reuse and frequency
  - In intel caches
  - Benefits in on chip space and performance

RRIP vs LRU

RRIP vs LRU

- Study access patterns and resulting HR
- Discuss reasons policies act accordingly
- Discuss trade-offs

Accesses Patterns

a b c d w x y z a b c d w x y z a b c d w x y z a b c d w x y z
a b w x c y d z a b w x c y d z a b w x c y d z a b w x c y d z
a w b x c y d z a w b x c y d z a w b x c y d z a w b x c y d z
a b w x c d y z a b w x c d y z a b w x c d y z a b w x c d y z
a b c d a b c d a b w c d x a b y c d z a b c x d a b y c d a z