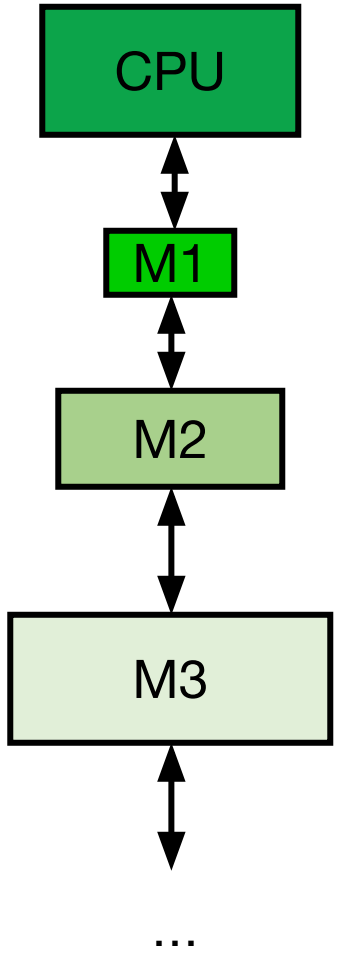
Module 7 – Advanced Topics in Cache

(ECE 209) Secure and Advanced Computer Architecture

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Memory Hierarchy

★ ***Idea:*** *have multiple layers of memory to balance between size and speed!*



Cache Performante

- Average Access Time = HitTime + MissRate × MissPenalty

★ *How to optimize this?*

Summary

- Miss Rate
  - Increase block size
  - Increase associativity
  - Increase cache size
  - Prefetching
  - Victim cache
  - Compiler
  - Replacement Policy
- Miss Penalty
  - Write buffer
  - Early restart with critical block first
  - Adding more levels
  - Sub-blocking
- Hit Time
  - Set associative cache
  - Add more levels
  - Parallel lookup
  - Speculative loads

Today's Plan

- Cache replacement policies
- Multicore caching

What to do when cache is full?

Replacement Policy

- Strategies to maximize hit rate.
- The basic idea is that we should evict something that is least useful.
- Invalid lines first, but what if are lines are valid and potentially useful?

Three Main Questions

- How to *select* which line to evict?

- What position to *install* the new line
  - LRU
  - MRU
  - Somewhere in between?

- What to do on a hit?
  - *Promote?* how much?

Examples

- LRU
  - Eviction
  - Install
  - Promotion

- Random
  - Eviction
  - Install
  - Promotion

Tradeoffs

- More sophisticated replacement policies need more storage for tracking and more logic for decision making.
  - More power and area
  - Larger latency

- Better replacement policies improve cache performance (AAT).

Types of Misses

- Compulsory
  - Larger line-size, prefetching
- Conflict
  - Higher associativity, skewed cache
- Capacity
  - Compression, software optimization to reduce working set
- Coherence
  - Better coherence algorithms, avoid wasteful communication
- Clairvoyance
  - Better replacement algorithms (view/prediction of future?)

Types of Workloads

- Cache-friendly
  - Small workset
  - Large workset
- Thrashing
  - [1…n] for n>> cache size
- Streaming
  - Almost no reuse

★ One application could have all of these (dynamic change).

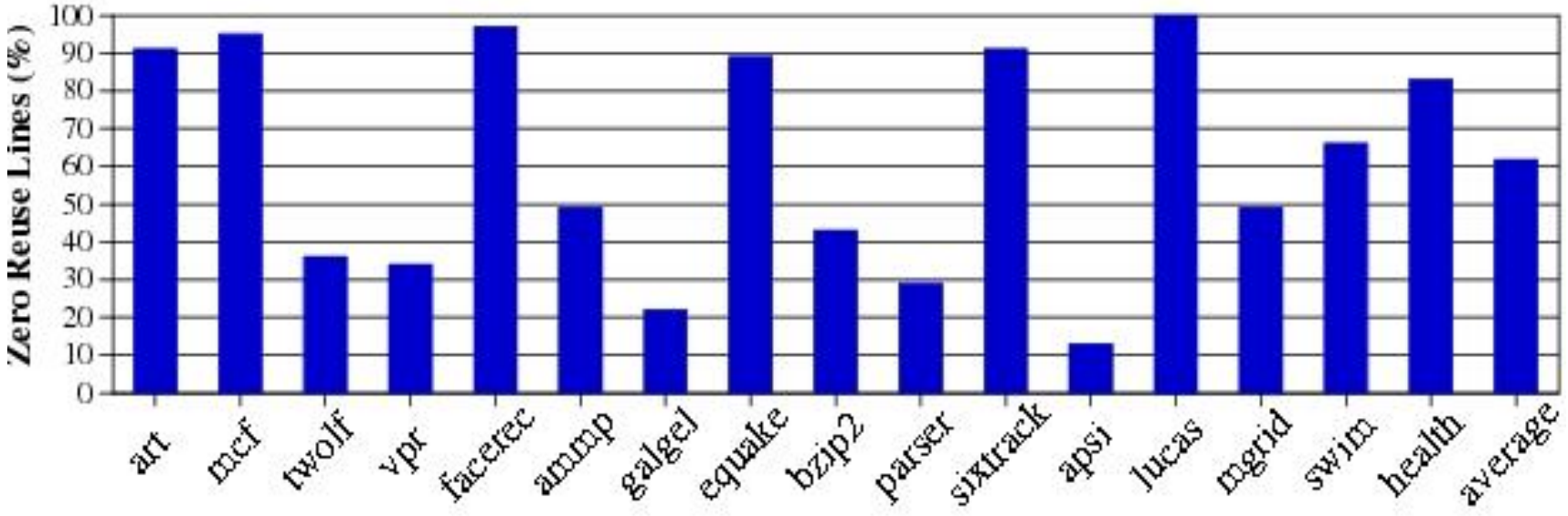
Is LRU good enough?

- Why it is a bad idea to install at MRU?

Is LRU good enough?

- Why it is a bad idea to install at MRU?
  → Dead on arrival lines!

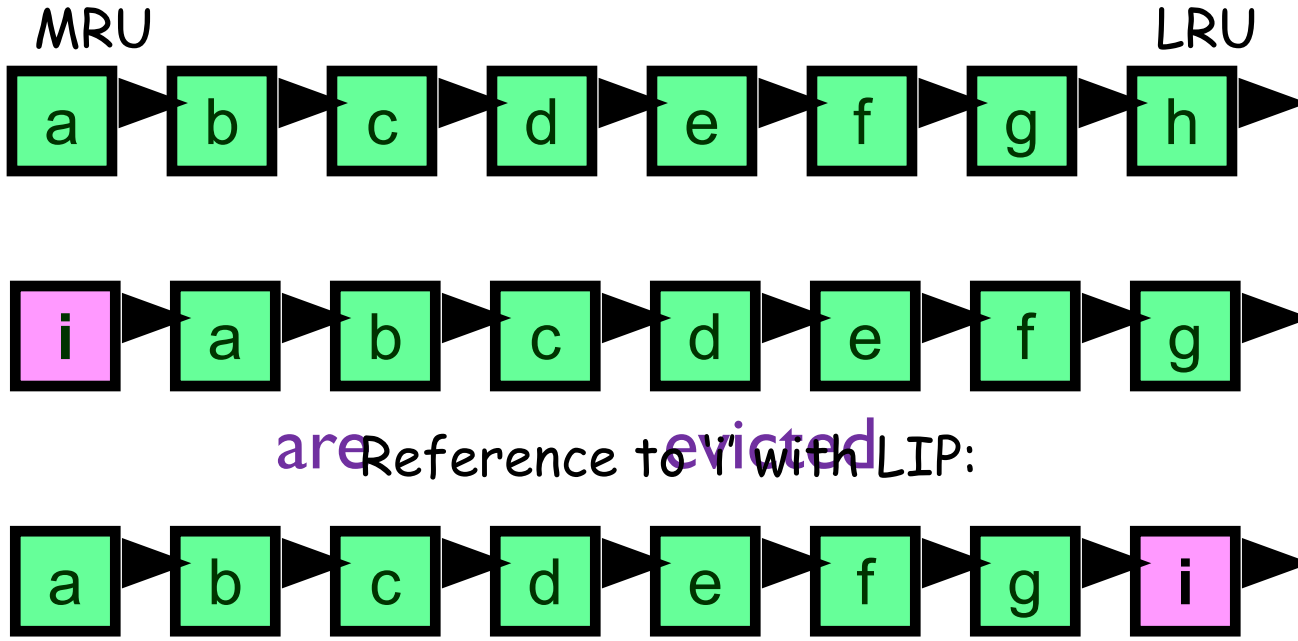
- Streaming data



How to fix this?

- LRU insertion policy (LIP)
  - Install on LRU instead of MRU
  - Promote to MRU only on a hit

→ DOA lines are evicted much faster.



- Problems?

Problem with LIP?

- Unfair to some lines since they have smaller chance to survive!
- What if the behavior of the program changes
  - Switching between thrashing (lots of DOA) and spatial locality
- What can we do?

BIP

- Bimodal-Insertion Policy (BIP)
  - Infrequently insert lines in MRU position.
  - Allows dynamic and fair change.

- Let ε = Bimodal throttle parameter

```
if   ( rand() < ε )
   Insert at MRU position;
else
   Insert at LRU position;
```

Problems with BIP?

Problems with BIP?

- What if the program is only LRU-friendly?
  - Lots of spatial and temporal locality.
  - Not many DOA lines.

- How to dynamically change between BIP and LRU?

Hybrid Methods?

- Use the policy which performs best at runtime for the given workload

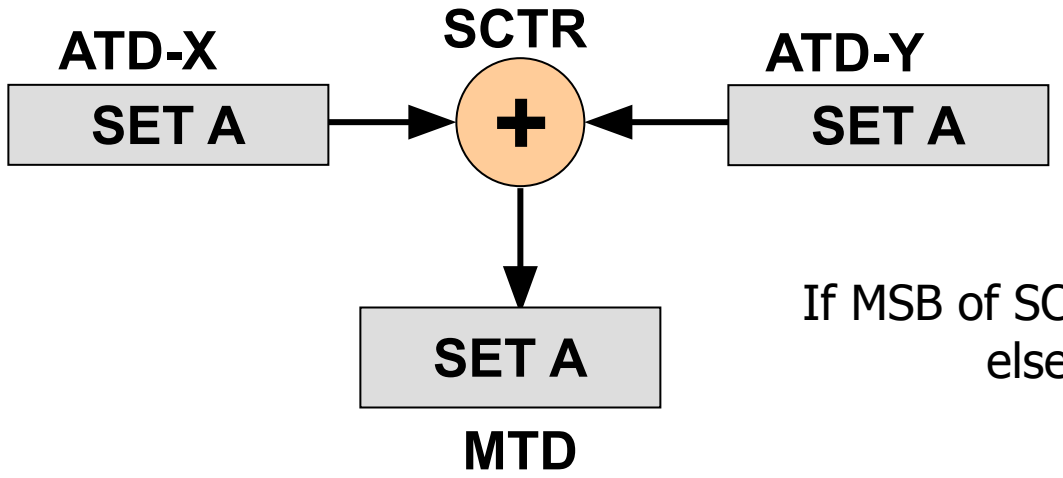
- We need to:
  - Estimate performance of the component policy
  - Choose the policy that reduces miss-rate
  - Apply the winner policy to the cache

Hybrid Methods

- How about implementing three caches: One that uses Policy-X, the second that uses Policy-Y, and third is the winner (main cache)

- For estimating misses we need only tag and not data → Auxiliary Tag Directory (ATD)

MTD: Main Tag Directory



If MSB of SCTR is 1, MTD uses Policy-X
else MTD use Policy-Y

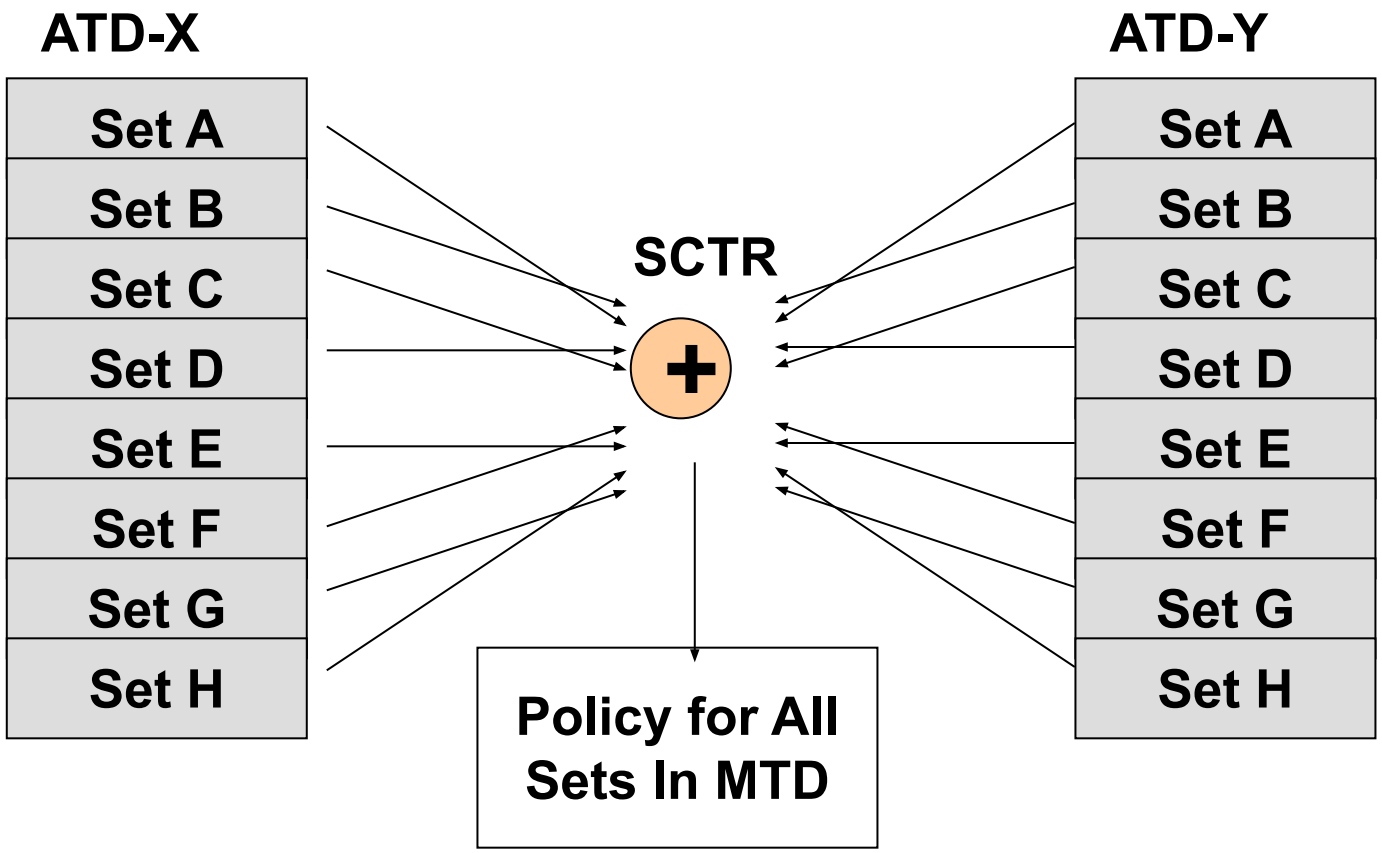
ATD-X	ATD-Y	Saturating Counter (SCTR)
HIT	HIT	Unchanged
MISS	MISS	Unchanged
HIT	MISS	++
MISS	HIT	--

Overhead?

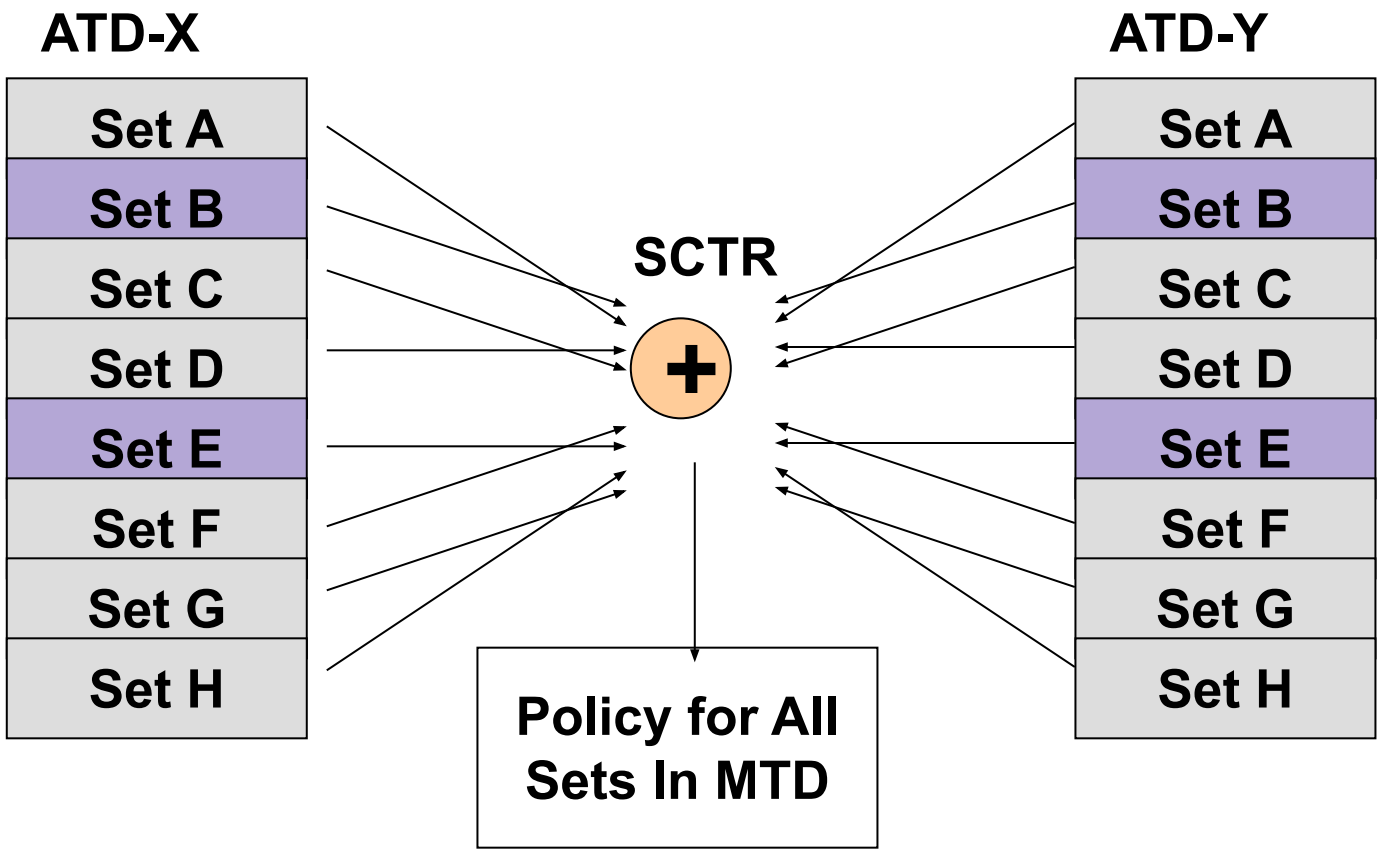
- Having two tags and one counter per set is quite expensive.

➢ *What can we do to optimize this?*

Counter overhead can be reduced by using a global counter



Tag overhead can be reduce by only *sampling* a few sets!



Dynamic Set Sampling

- How many sets are required to choose best performing policy?
  - Bounds using analytical model and simulation.
  - DSS with 32 sample sets performs similar to having all sets.
  - Last-level cache typically contains 1000s of sets, thus ATD entries are required for only 2%-3% of the sets.

DIP

- Dynamic Insertion Policy (DIP):
  - Two types of workloads: LRU-friendly or BIP-friendly → use DSS to choose between them dynamically.

- DIP can be implemented by:
  - Monitor both policies (LRU and BIP)
  - Choose the best-performing policy
  - Apply the best policy to the cache

Divide the cache in three:

- Dedicated LRU sets
- Dedicated BIP sets
- Follower sets (winner of LRU,BIP)

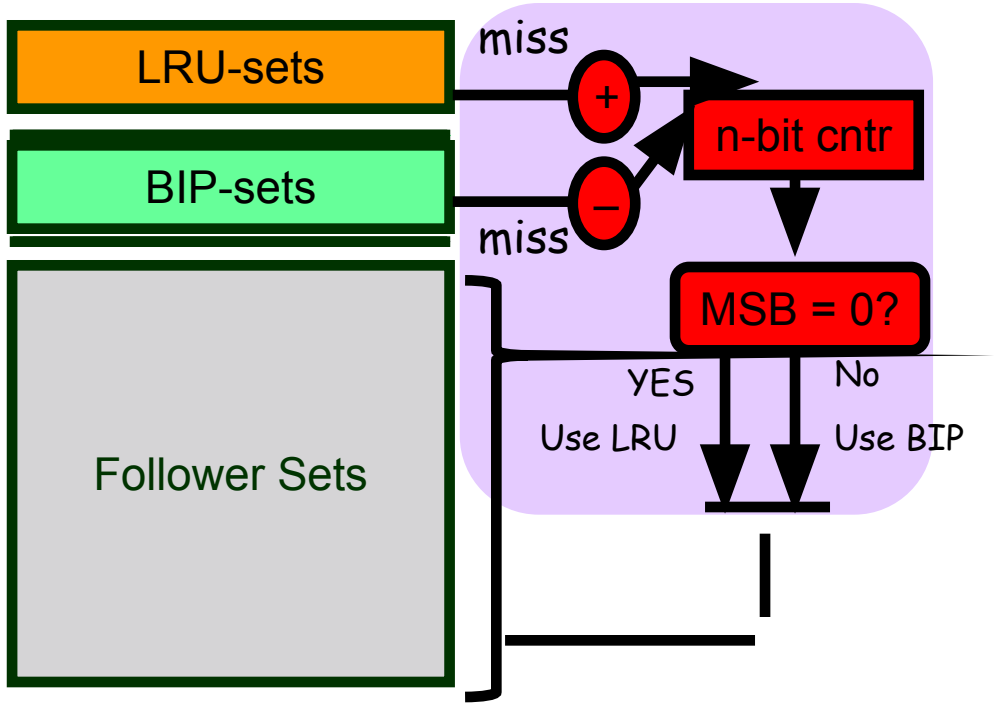
n-bit saturating counter

misses to LRU-sets: counter++

misses to BIP-set: counter--

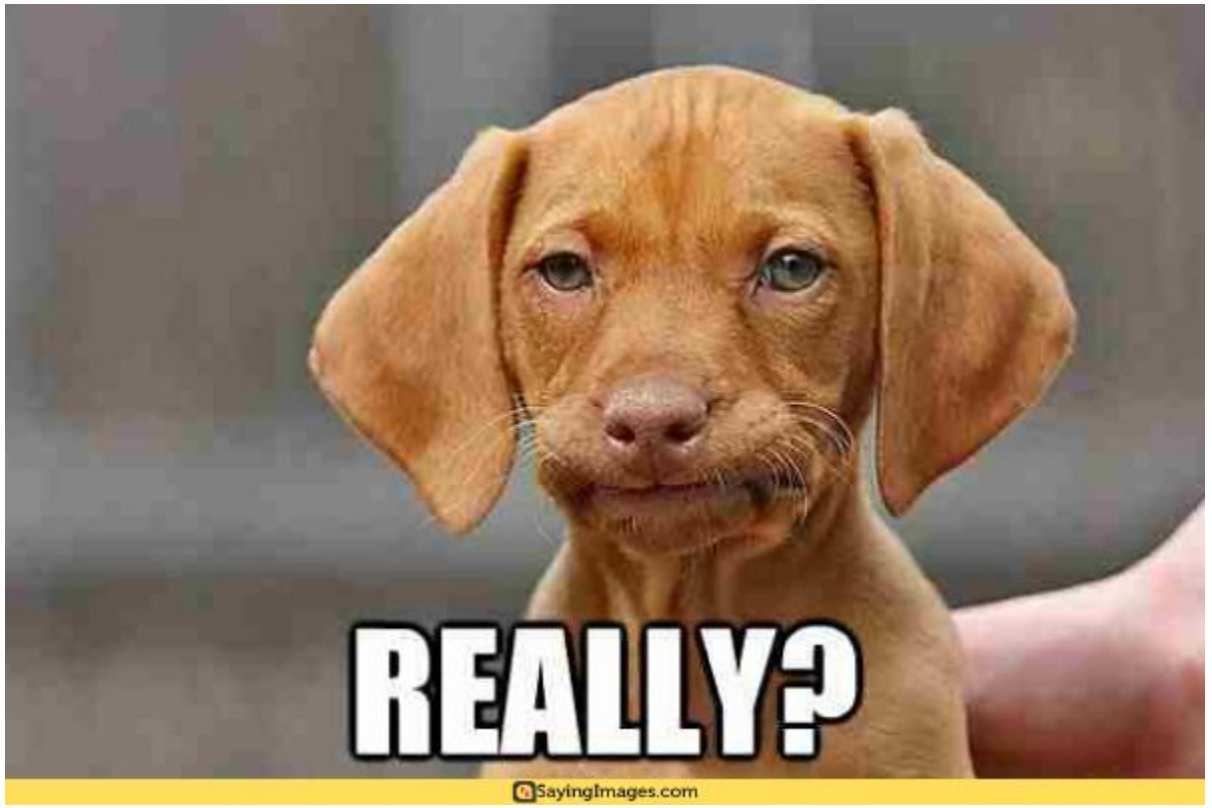
Counter decides policy for Follower sets:

- MSB = 0, Use LRU
- MSB = 1, Use BIP



monitor → choose → apply
(using a single counter)

Problems with DIP?



Three Main Questions

- How to *select* which line to evict?

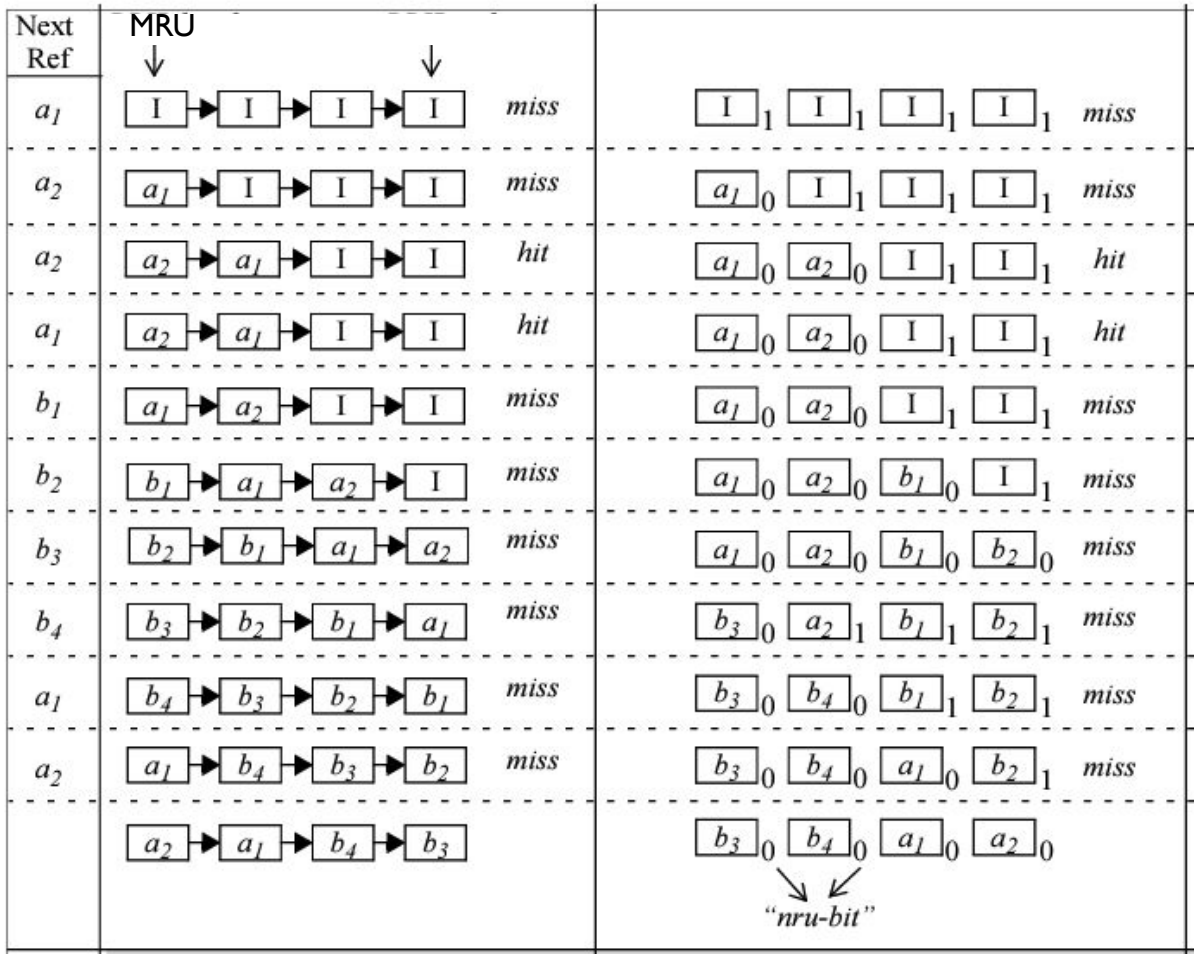
- What position to *install* the new line
  - LRU
  - MRU
  - Somewhere in between?

- What to do on a hit?
  - *Promote?* how much?

How to select the best line?

- LRU leverages a timestamp but it is very expensive.
  - Why? → updating and checking takes time + storage overhead.

- PLRU for optimization
  - Tree-LRU
  - NRU


Next Ref
MRU
a_1 I I I I miss
a_2 a_1 I I I miss
a_2 a_2 a_1 I I hit
a_1 a_2 a_1 I I hit
b_1 a_1 a_2 I I miss
b_2 b_1 a_1 a_2 I miss
b_3 b_2 b_1 a_1 a_2 miss
b_4 b_3 b_2 b_1 a_1 miss
a_1 b_4 b_3 b_2 b_1 miss
a_2 a_1 b_4 b_3 b_2 miss
a_2 a_1 b_4 b_3
I_1 I_1 I_1 I_1 miss
$a_1{}_0$ I_1 I_1 I_1 miss
$a_1{}_0$ $a_2{}_0$ I_1 I_1 hit
$a_1{}_0$ $a_2{}_0$ I_1 I_1 hit
$a_1{}_0$ $a_2{}_0$ I_1 I_1 miss
$a_1{}_0$ $a_2{}_0$ $b_1{}_0$ I_1 miss
$a_1{}_0$ $a_2{}_0$ $b_1{}_0$ $b_2{}_0$ miss
$b_3{}_0$ $a_2{}_1$ $b_1{}_1$ $b_2{}_1$ miss
$b_3{}_0$ $b_4{}_0$ $b_1{}_1$ $b_2{}_1$ miss
$b_3{}_0$ $b_4{}_0$ $a_1{}_0$ $b_2{}_1$ miss
$b_3{}_0$ $b_4{}_0$ $a_1{}_0$ $a_2{}_0$
"nru-bit"
(a) LRU
(b) Not Recently Used (NRU)
Cache Hit:
(i) move block to MRU
Cache Miss:
(i) replace LRU block
(ii) move block to MRU
Cache Hit:
(i) set nru-bit of block to '0'
Cache Miss:
(i) search for first '1' from left
(ii) if '1' found go to step (v)
(iii) set all nru-bits to '1'
(iv) goto step (i)
(v) replace block and set nru-bit to '1'

How to improve?

- Can we track how useful a line has been so far?
- nru bit does that but with only one bit → Let's have an n-bit counter per block.
- We call this Re-Reference Interval Prediction (RRIP) counter.

RRIP

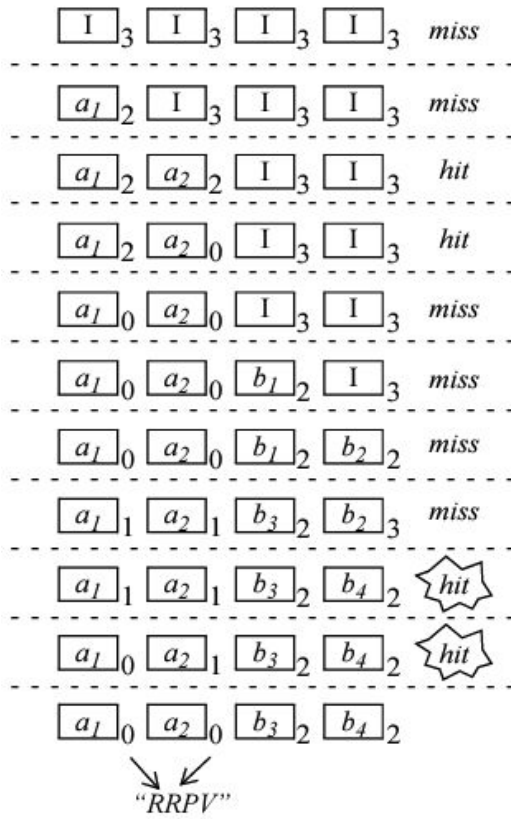
- How many bits? → 2 bit is a good balance.

- Where to install?
  - MRU?
  - LRU
  - middle?

- What to do on a hit?

(S)RRIP

- Insert on 2.
- Promote to 0 on a hit.
- Evict the line with a 3. If no line, then age.



(c) 2-bit SRRIP with Hit Promotion

Cache Hit:
(i) set RRPV of block to '0'

Cache Miss:
(i) search for first '3' from left
(ii) if '3' found go to step (v)
(iii) increment all RRPVs
(iv) goto step (i)
(v) replace block and set RRPV to '2'

How to improve?

- Use BIP with SRRIP ⟶ BRRIP
  - Most lines are installed at 2.
  - Some lines (with small probability) are installed at 1.

How to improve?

- Use BIP with SRRIP ⟶ BRRIP
  - Most lines are installed at 2.
  - Some lines (with small probability) are installed at 1.

- How to choose between SRRIP and BRRIP ⟶ use DSS.
  - Chose a subset with SRRIP. Another with BRRIP. The winner is dynamically decided. We call this DRRIP.

Is there more?

- What to do on a hit?
  - PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches (ISCA 2009).

- Many more …

CA2

- Replacement Policy Championship

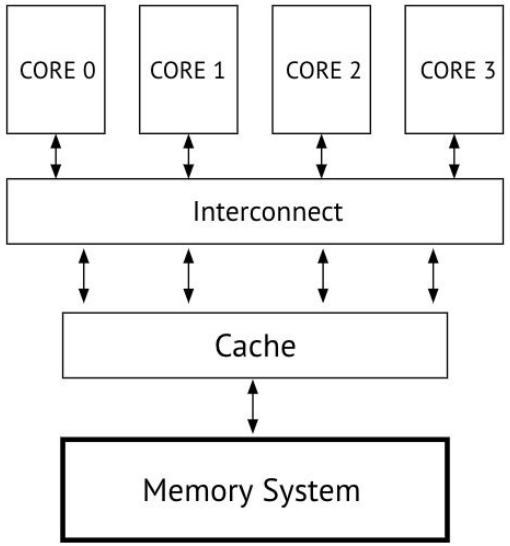
Today's Plan

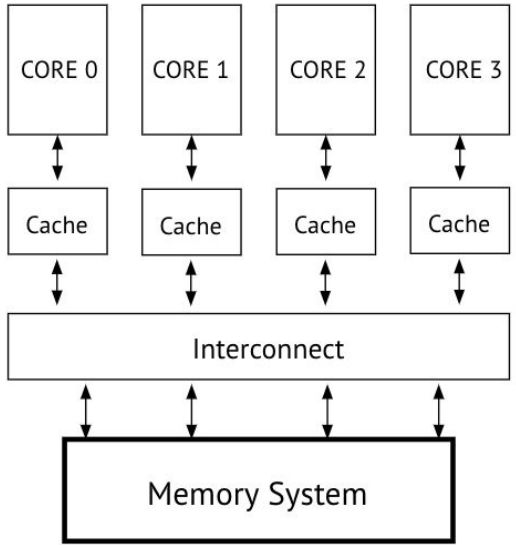
- Cache replacement policies
- Multicore caching

How to share the cache among applications and cores?

- Recall that we have superscalar hyper threaded cores so multiple application are running at the same time. They all need cache so we need to decide how to share it between them!

Shared vs. Private





Shared vs. Private

- Hittime → private is significantly faster because it is on-chip. Therefore, L1 and L2 caches are usually private.

- Private cache are inefficient for space utilization thus L3 is usually shared.

Treadeoffs in Shared Caches

- Advantages:
  - Dynamic partitioning of available cache space
  - No fragmentation due to static partitioning
  - Easier to maintain coherence
  - Shared data and locks do not ping pong between caches

- Disadvantages
  - Cores incur conflict misses due to other cores' accesses
  - Some cores can destroy the hit rate of other cores
  - High bandwidth interconnect to connect cores and cache

How to share?

How to share?

- Equal?

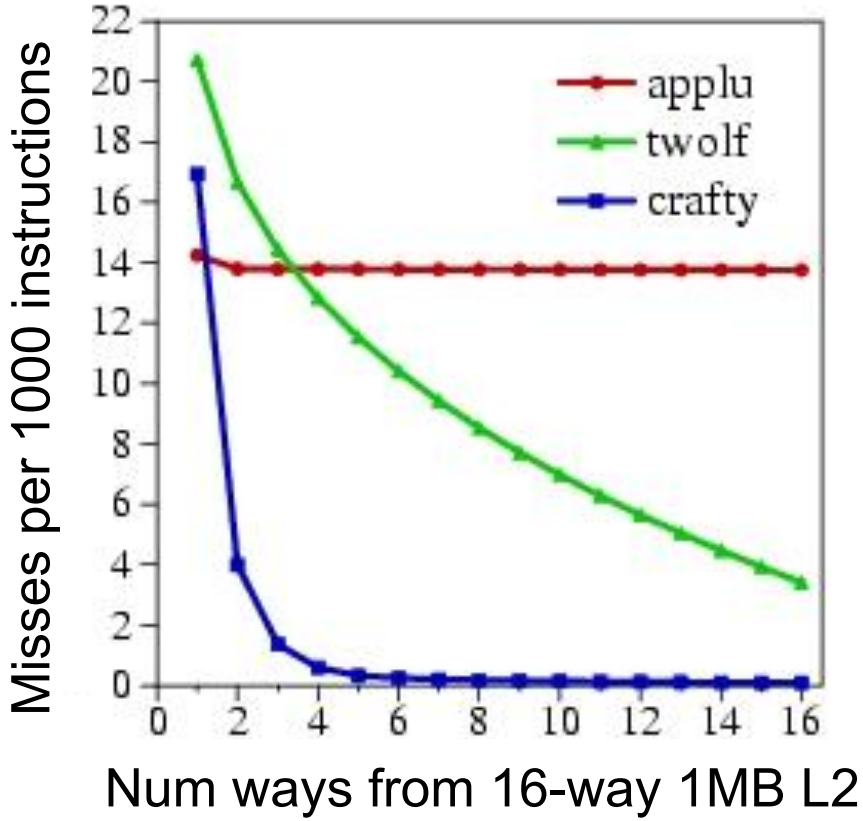
- Free for all?
  - Is cache demand == cache performance or hit rate?
  - Think about a thrashing application

Utility-Based Cache Partitioning

- Give more space to a core that could benefit from it.
  - How to track?
  - How to partition?

UCP

Utility U_a^b = Misses with a ways – Misses with b ways

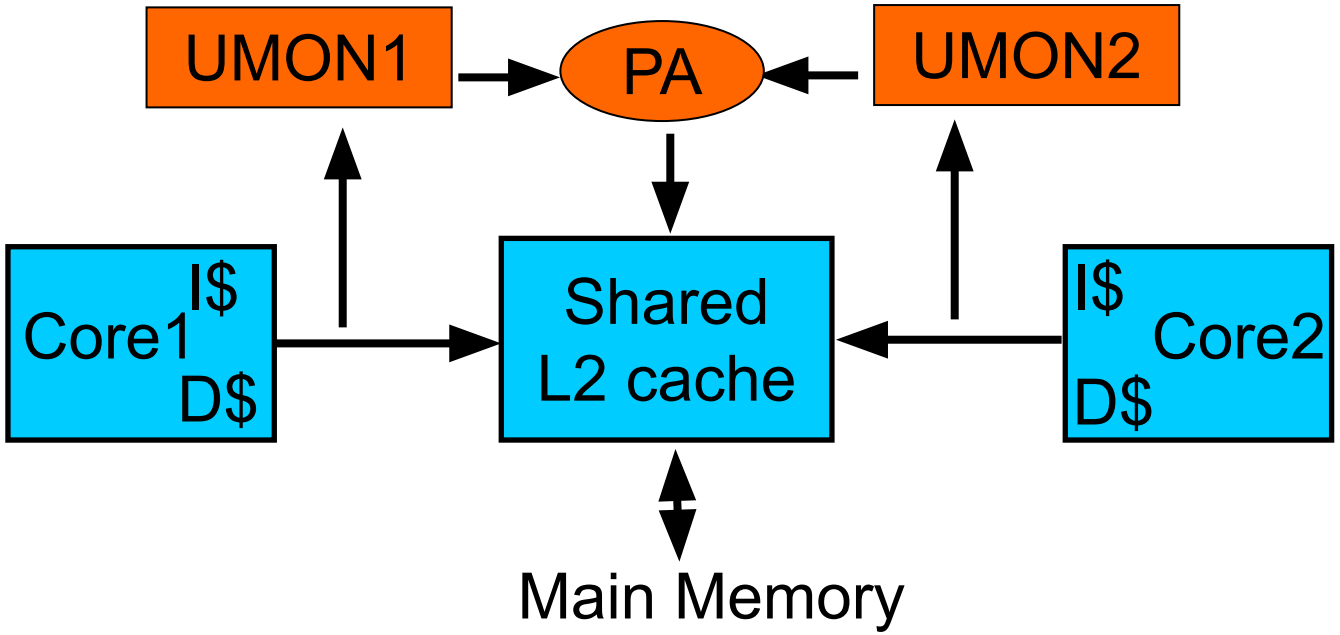


Low Utility

High Utility

Saturating Utility

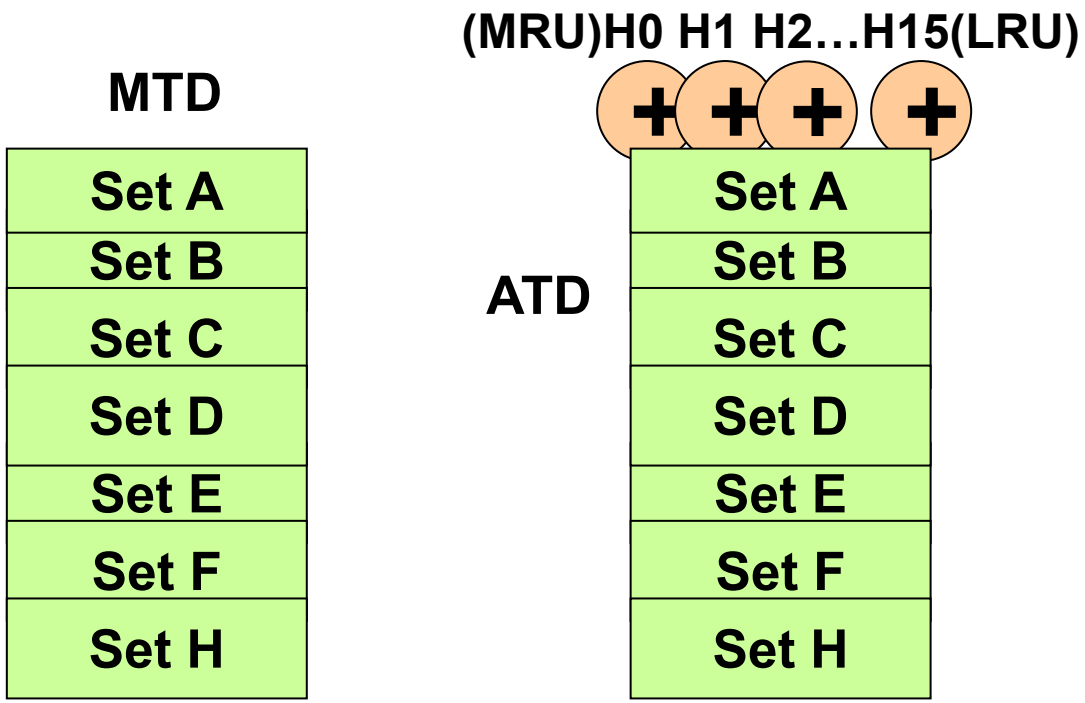
UCP



- Three components:
  - Utility Monitors (UMON) per core
  - Partitioning Algorithm (PA)
  - Replacement support to enforce partitions

UCP

- For each core, simulate LRU using auxiliary tag directory (ATD)
- Hit counters in ATD to count hits per recency position
- LRU is a stack algorithm: hit counts → utility
- E.g. hits(2 ways) = H0+H1



How to optimize UCP?

- Use DSS to only track a small subset of sets.

How to partition?

- Evaluate all possible partitions and select the best
  - (This does not scale well for more than 2 cores, why?)

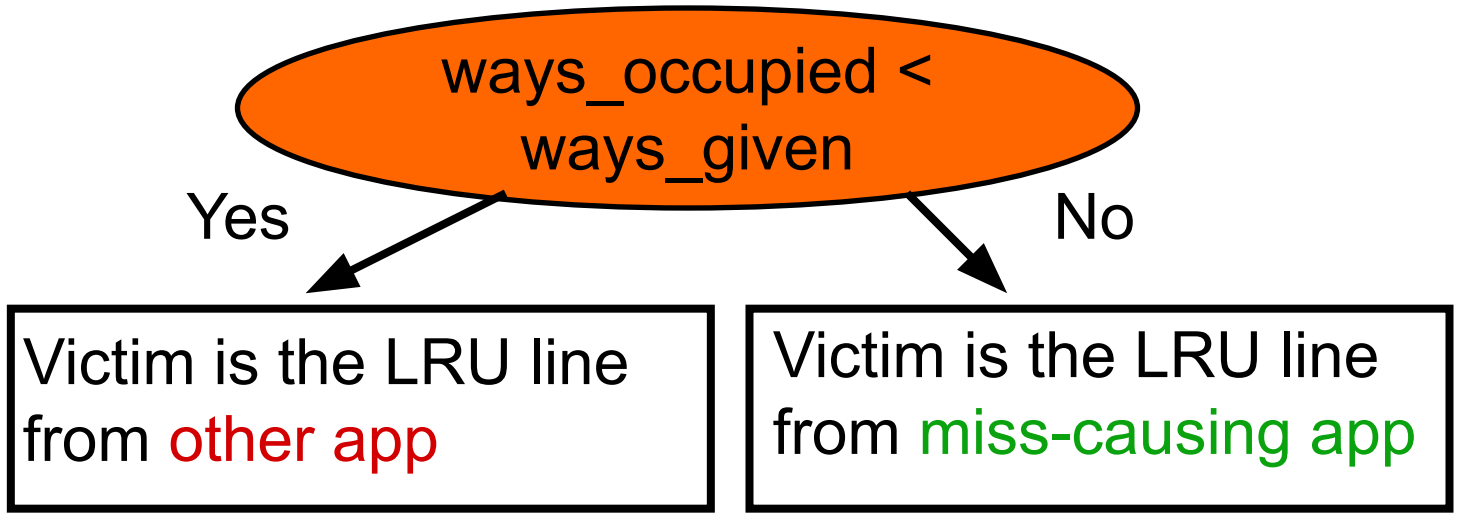
- With a ways to core1 and (16−a) ways to core2:
  - `Hitscore1 = (H0 + H1 + … + Ha-1)` ---- from UMON1
  - `Hitscore2 = (H0 + H1 + … + H16-a-1)` ---- from UMON2

- Select `a` that maximizes (`Hitscore1 + Hitscore2`)

- Partitioning done once every 5 million cycles

UCP

- Way partitioning support:
  - Each line has core-id bits
  - On a miss, count ways_occupied in set by miss-causing app



How to extend it beyond two?

- An optimization problem.

- *Read more here:* Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches.

Additional Methods

- Cache coloring

- Many dynamic and static methods
  - Software, hardware, hybrid

End of Presentation

UCLA Samueli School of Engineering

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - 6.888 MIT (Yan)
  - CS 598 UIUC (Fletcher)
  - CS 7290 Georgia Tech (Kim)
  - ECE 752 Wisconsin (Lipasti)
  - ECE 7103A Georgia Tech (Qureshi)