15-740/18-740 Computer Architecture Lecture 12: Advanced Caching

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Chuck Thacker (Microsoft Research) Seminar Tomorrow
  - RARE: Rethinking Architectural Research and Education
  - October 7, 4:30-5:30pm, GHC Rashid Auditorium
- Ben Zorn (Microsoft Research) Seminar Friday
  - Performance is Dead, Long Live Performance!
  - October 8, 11am-noon, GHC 6115
- Guest lecture Friday
  - Dr. Ben Zorn, Microsoft Research
  - Fault Tolerant, Efficient, and Secure Runtimes

Announcements

- Homework 2 due
  - October 10

- Midterm I
  - October 11
  - Sample exams online
  - You can bring one letter-sized cheat sheet

- Project proposals
  - Meetings with some groups Wednesday, October 13
  - Contact TAs assigned → Meet with them
  - Come to office hours for feedback

Last Time …

- Dual-core Execution, Slipstream Idea
- Store-Load Dependency Handling
- Speculative Execution and Data Coherence
- Open Research Issues in OoO Execution
- Asymmetric vs. Symmetric Cores
- ACMP
- Accelerated Critical Sections using ACMP

- Advanced Caching
  - Inclusion vs. Exclusion
  - Multi-level caches in Pipelined Designs

Topics in (Advanced) Caching

- Inclusion vs. exclusion, revisited
- Handling writes
- Instruction vs. data
- Cache replacement policies
- Cache performance
- Enhancements to improve cache performance
- Enabling multiple concurrent accesses
- Enabling high bandwidth caches

Readings

- Required:
  - Hennessy and Patterson, Appendix C.1-C.3
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

- Recommended:
  - Seznec, "A Case for Two-way Skewed Associative Caches," ISCA 1993.
  - Chilimbi et al., "Cache-conscious Structure Layout," PLDI 1999.
  - Chilimbi et al., "Cache-conscious Structure Definition," PLDI 1999.

Handling Writes (Stores)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back
  - -- Need a bit in the tag store indicating the block is “modified”
  - + Can consolidate multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy

- Write-through
  - + Simpler
  - + All levels are up to date. Consistency: Simpler cache coherence because no need to check lower-level caches
  - -- More bandwidth intensive

Handling Writes (Stores)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss

  + Can consolidate writes instead of writing each of them individually to next level

  + Simpler because write misses can be treated the same way as read misses

  -- Requires (?) transfer of the whole cache block

- No-allocate

  + Conserves cache space if locality of writes is low

Instruction vs. Data Caching

- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)
  - -- Instructions and data can conflict with each other (i.e., no guaranteed space for either)
  - -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split
  - for the last reason above
- Second and higher levels are almost always unified

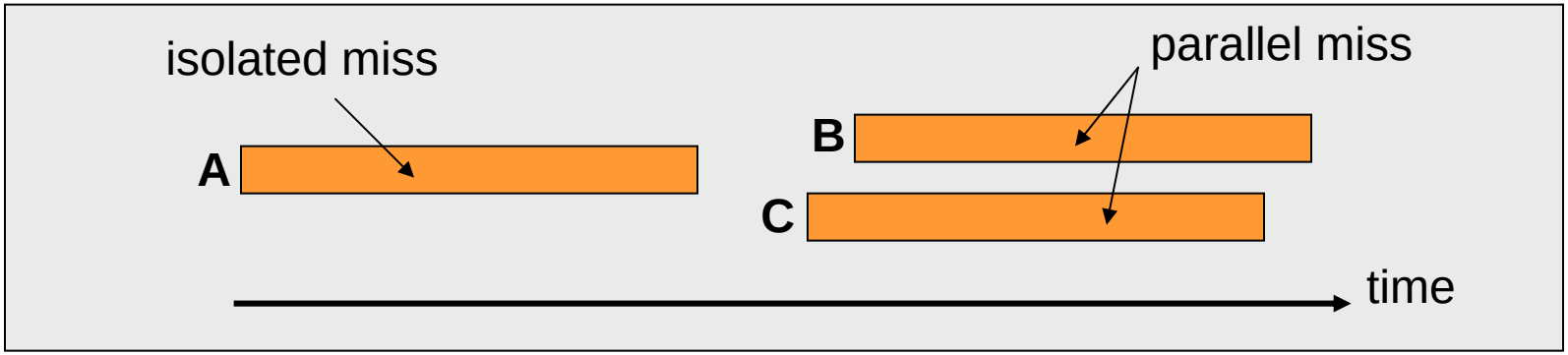
Set-Associative Caches (I)

- Diminishing returns in hit rate from higher associativity
- Longer access time with higher associativity
- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - **Least costly to re-fetch?**
      - **Why would memory accesses have different cost?**
    - Hybrid replacement policies
    - Optimal replacement policy?

Set-Associative Caches (II)

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?
  - Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Memory Level Parallelism (MLP)



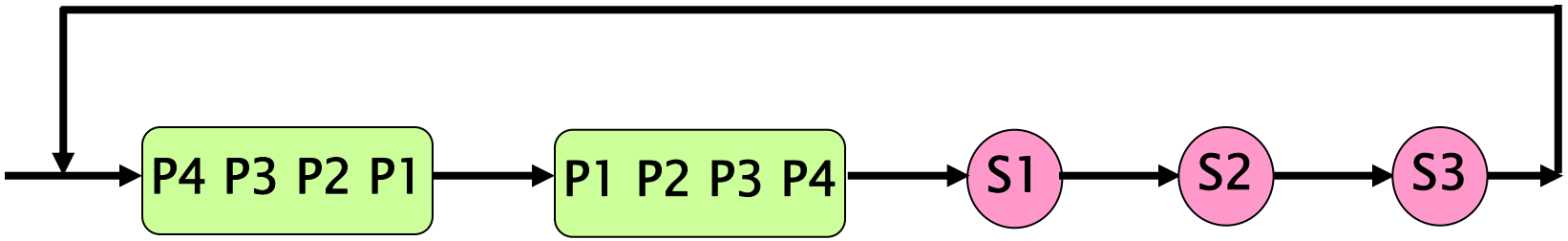
- Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew'98]
- Several techniques to improve MLP (out-of-order, runahead etc.)
- MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache replacement policies try to reduce miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP breaks this assumption!
- Eliminating an isolated miss helps performance more than eliminating a parallel miss
- Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



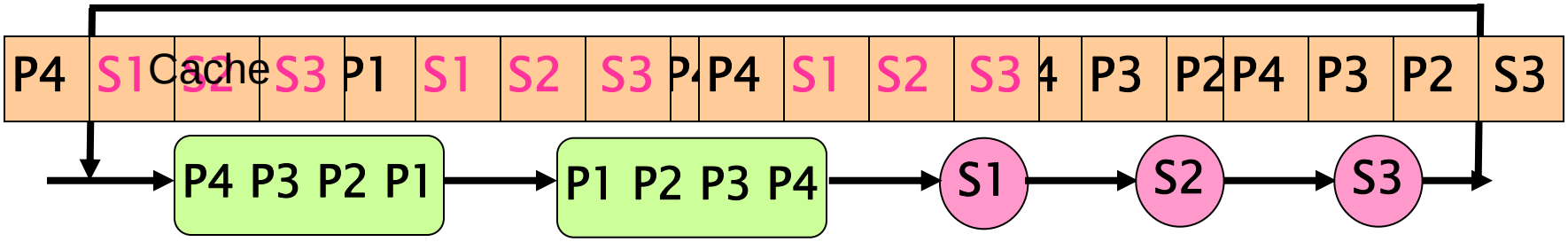
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

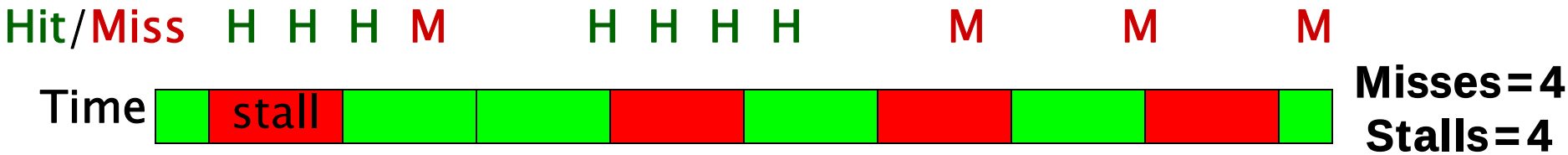
Two replacement algorithms:

1. Minimizes miss count (Belady's OPT)
2. Reduces isolated miss (MLP-Aware)

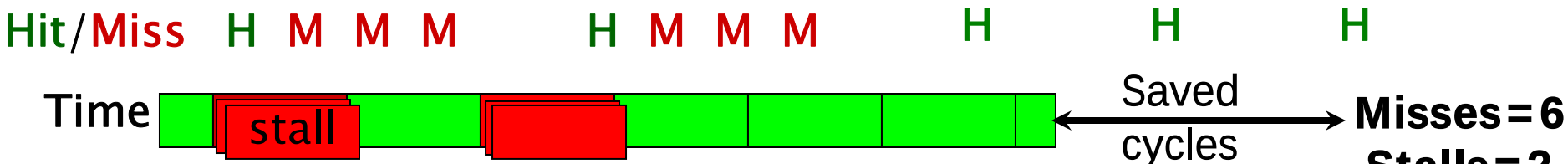
For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance





Belady's OPT replacement



MLP-Aware replacement

Cache Performance

Improving Cache "Performance"

- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency
- Reducing hit latency

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Software approaches

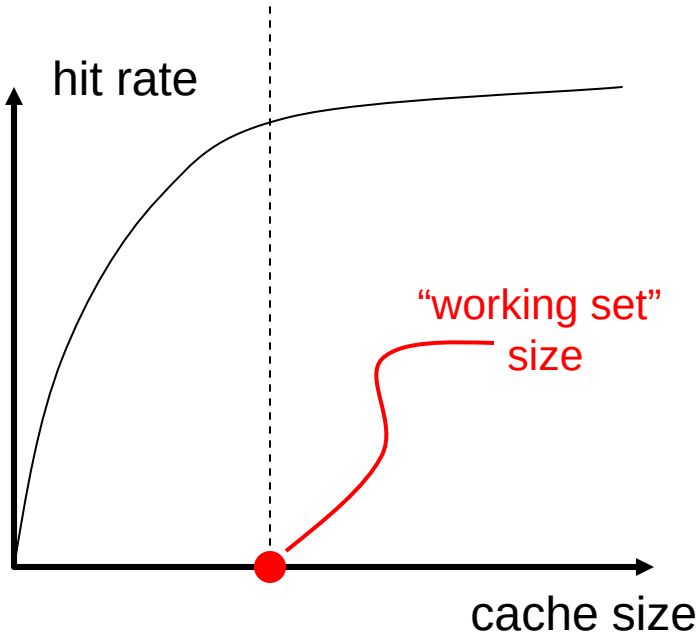
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking
  - Non-blocking caches
  - Multiple accesses per cycle
  - Software approaches

Cache Parameters vs. Miss Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

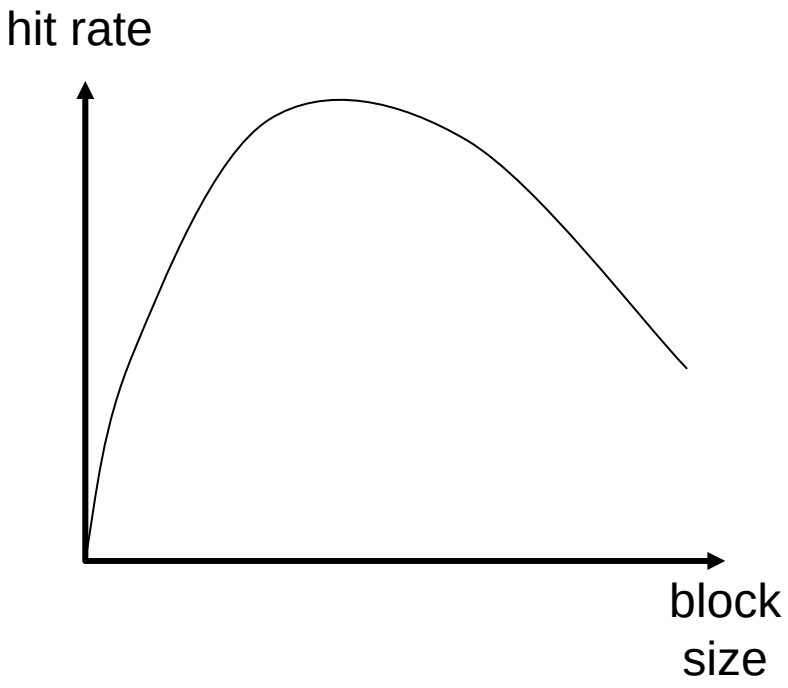
Cache Size

- Cache size in the total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data the executing application references
  - Within a time interval



Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each with V bit)
      - Can improve “write” performance
- Small blocks
  - don’t exploit spatial locality well
  - have larger tag overhead
- Large blocks
  - likely-useless data transferred
    - Extra bandwidth/energy consumed
  - too few total # of blocks
    - Useful data frequently replaced



Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line critical word first
  - restart cache access before complete fill

- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid bits for each subblock
  - When is this useful?

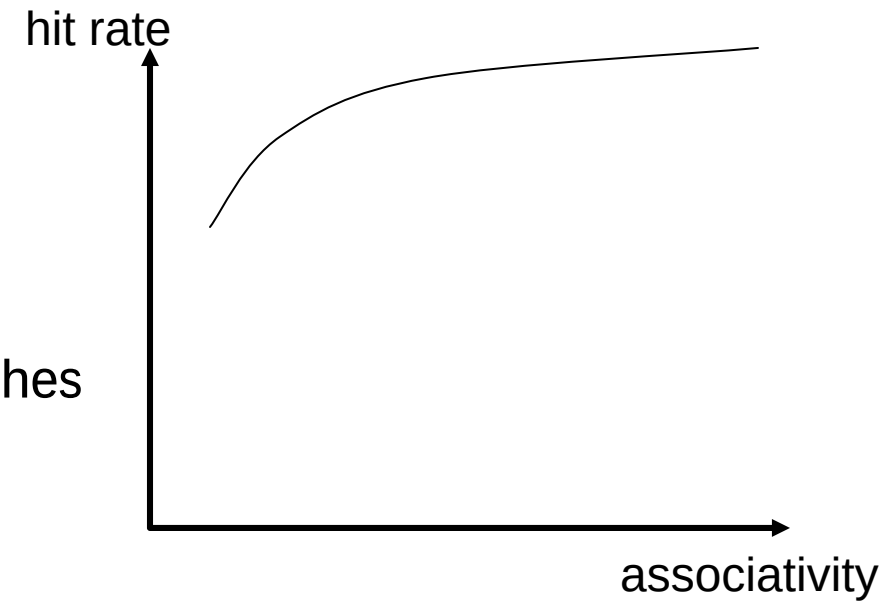
Associativity

- How many blocks can map to the same index (or set)?

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns

- Smaller associativity
  - lower hardware cost
  - faster hit time
    - Especially important for L1 caches

- Power of 2 associativity?



Replacement Policy

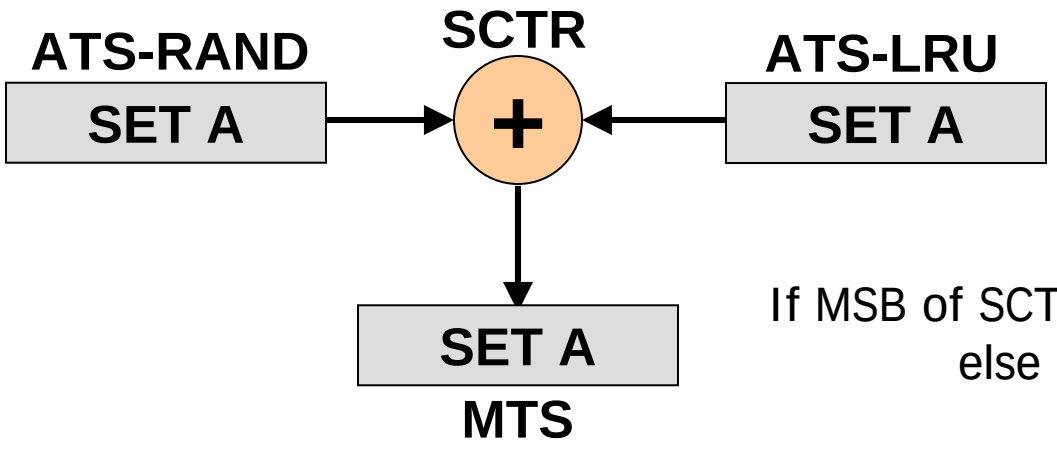
- LRU vs. Random
  - Set thrashing: When the "program working set" in a set is larger than set associativity
  - 4-way: Cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

- Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Hybrid Replacement Policies

- Idea:
  - Have 3 tag stores:
    - 2 auxiliary tag stores (ATS) dedicated to alternate policies
    - One main tag store (MTS) implementing both policies.
  - Simulate alternate policies in the ATS
  - Use the “winning” policy in the MTS

Random-LRU Hybrid Replacement (I)



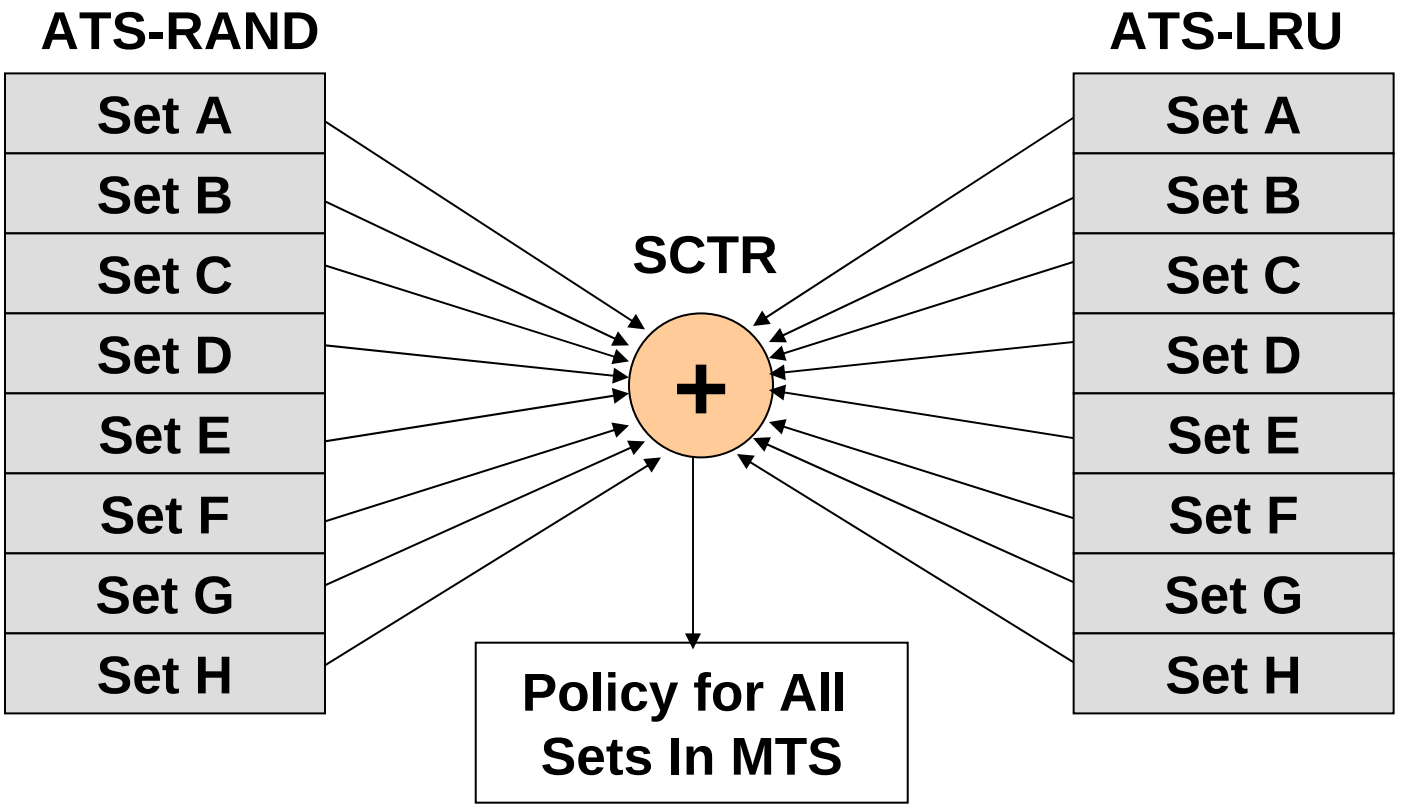
If MSB of SCTR is 1, MTS uses RAND
else MTS uses LRU

ATS-RAND	ATS-LRU	Saturating Counter (SCTR)
HIT	HIT	Unchanged
MISS	MISS	Unchanged
HIT	MISS	+= 1
MISS	HIT	-= 1

Random-LRU Hybrid Replacement (II)

Implementing this on a per-set basis is expensive

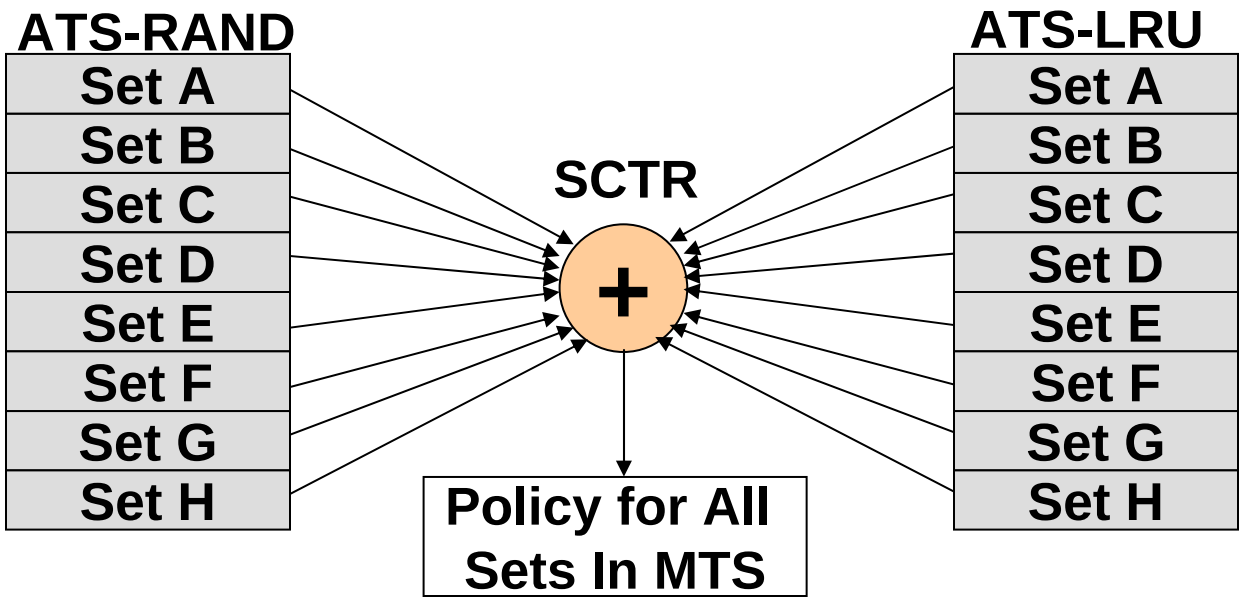
Counter overhead can be reduced by using a global counter



Random-LRU Hybrid Replacement (III)

Not all sets are required to decide the best policy

Have the ATS entries only for few sets. This is called set sampling.



Sets that have ATS entries (B, E, G) are called leader sets

Set Sampling for Hybrid Replacement

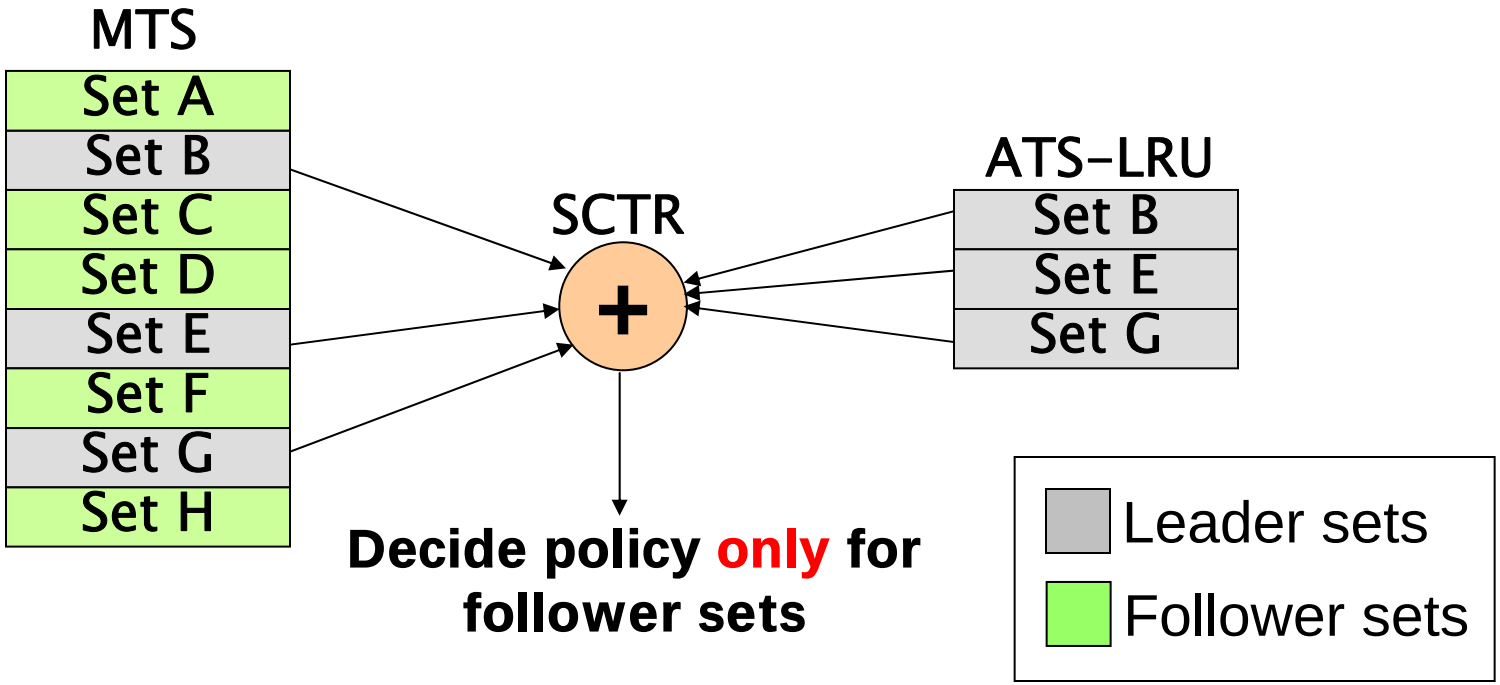
- How many sets are required to choose best performing policy?

- Bounds using analytical model and simulation (in Qureshi et al., ISCA 2006)
- Sampling 32 leader sets performs similar to having all sets
- Last-level cache typically contains 1000s of sets

  → ATS entries are required for only 3% of the sets

- ATS overhead can further be reduced by using MTS to always simulate one of the policies (say RAND)

Sampling-Based Random-LRU Hybrid



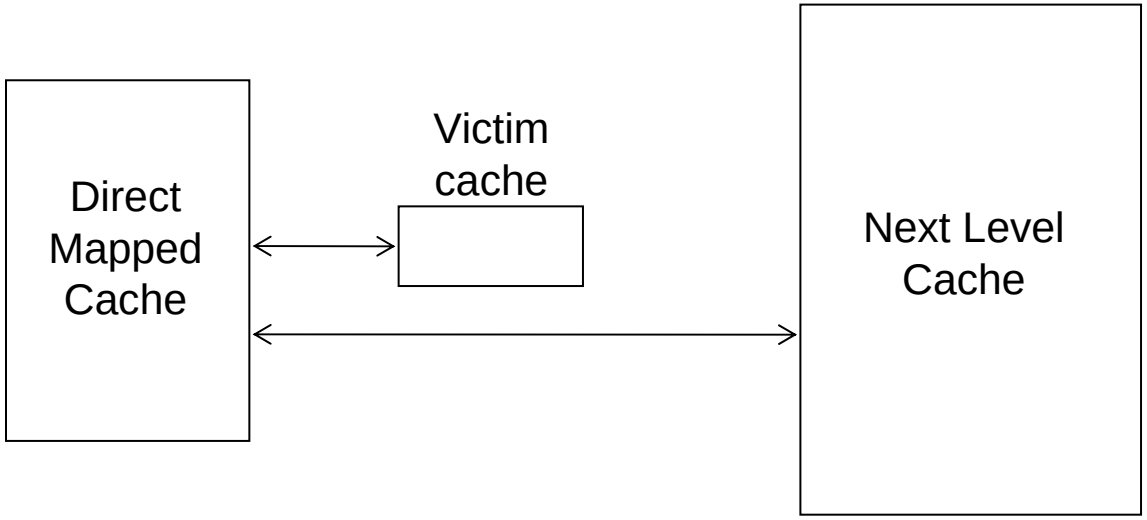
Classification of Cache Misses

- Compulsory (Cold)
  - The block was never accessed before
  - Can sole caching do anything about compulsory misses?
- Conflict
  - A same-size fully-associative cache would not have missed
- Capacity
  - The cache was too small (even if it were fully associative)
    - Neither compulsory nor conflict
- Coherence/communication (multiprocessor)
  - Another processor invalidated the block

How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Hashing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set such that each "phase" fits in cache

Victim Cache: Reducing Conflict Misses



- Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
- Idea: Use a small fully associative buffer (victim cache) to store evicted blocks
  - + Can avoid ping ponging of cache blocks mapped to the same set (if two cache blocks continuously accessed in nearby time conflict with each other)
  - -- Increases miss latency if accessed serially with L2

Victim Cache Performance

- Generally helps with
  - Temporally close conflict misses
    - Smaller associativity
  - Smaller cache size
  - Instruction caches
  - Larger block size (fewer total blocks)

Hashing and Pseudo-Associativity

- Hashing: Better "randomizing" index functions

  + can reduce conflict misses
    - by distributing the accessed memory blocks more evenly to sets
  - Example: stride where stride value equals cache size

  -- More complex to implement: can lengthen critical path

- Pseudo-associativity (Poor Man's associative cache)
  - Serial lookup: On a miss, use a different index function and access cache again
  - Given a direct-mapped array with K cache blocks
    - Implement K/N sets
    - Given address Addr, sequentially look up: {0,Addr[lg(K/N)-1: 0]}, {1,Addr[lg(K/N)-1: 0]}, … , {N-1,Addr[lg(K/N)-1: 0]}