SFU

CMPT 450/750: Computer Architecture
Fall 2022

Cache Management

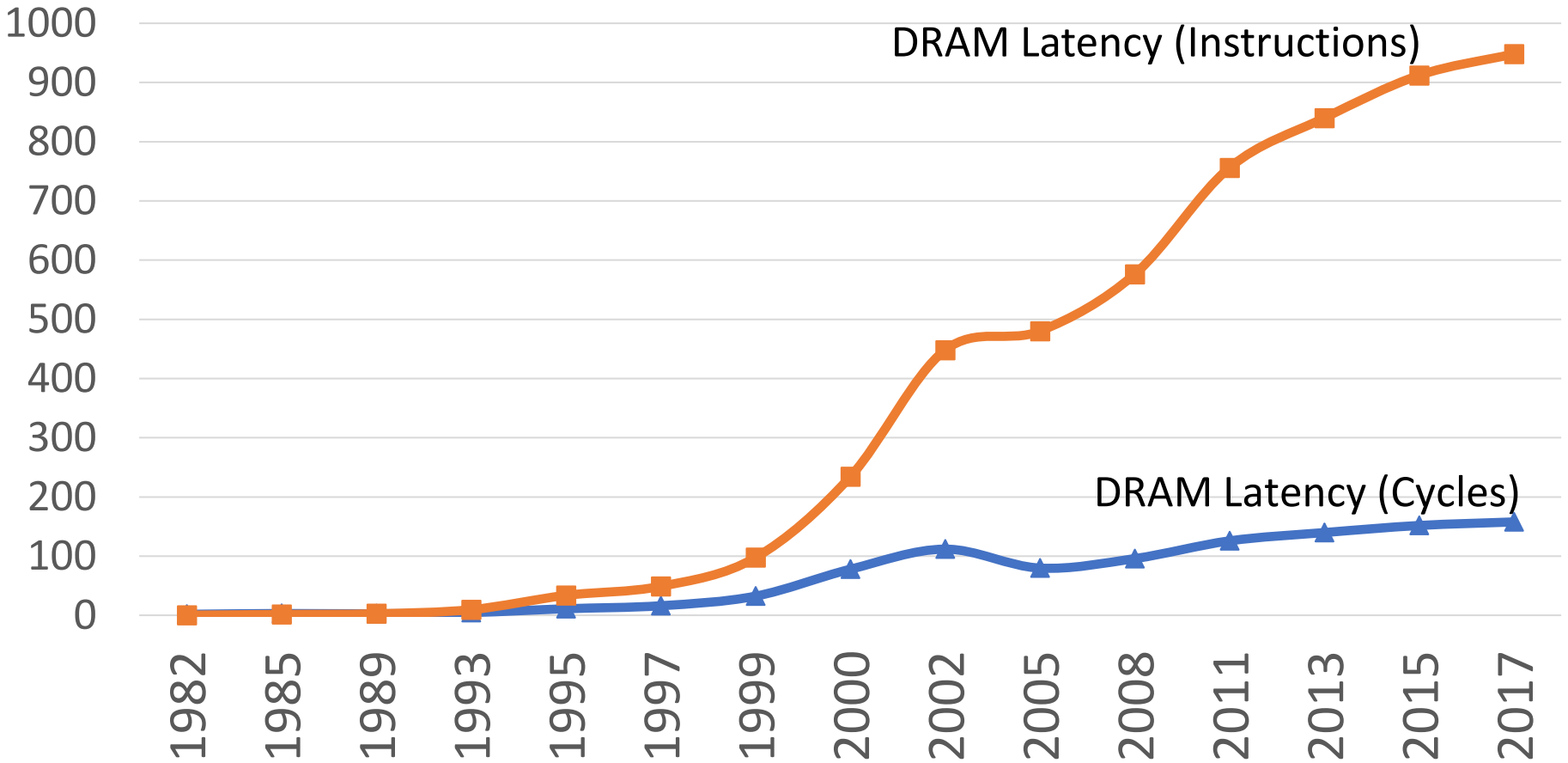
Alaa Alameldeen & Arrvindh Shriraman

Recall: Cache Misses are Expensive

- **Cache misses result in large performance and energy losses**
- **Cache Miss Types:**
  - **Compulsory:** Misses in an infinite cache
  - **Capacity:** Misses in a fully-associative cache
  - **Conflict:** Misses due to limited number of ways per set
- **To reduce cache misses, we can use:**
  - High associativity or/and victim caching (conflict misses)
  - Prefetching (compulsory and capacity)
  - Effective replacement algorithms (conflict and capacity misses)
  - Insertion policies (conflict and capacity misses)
  - Dead block prediction (conflict and capacity misses)

Impact of Cache Misses

- **Why are cache misses expensive?**
  - Blocking cache: Severely reduce performance
  - Non-blocking cache: Load stalls in ROB, can prevent instruction issue or fetch



Cache Hit Time Tradeoffs

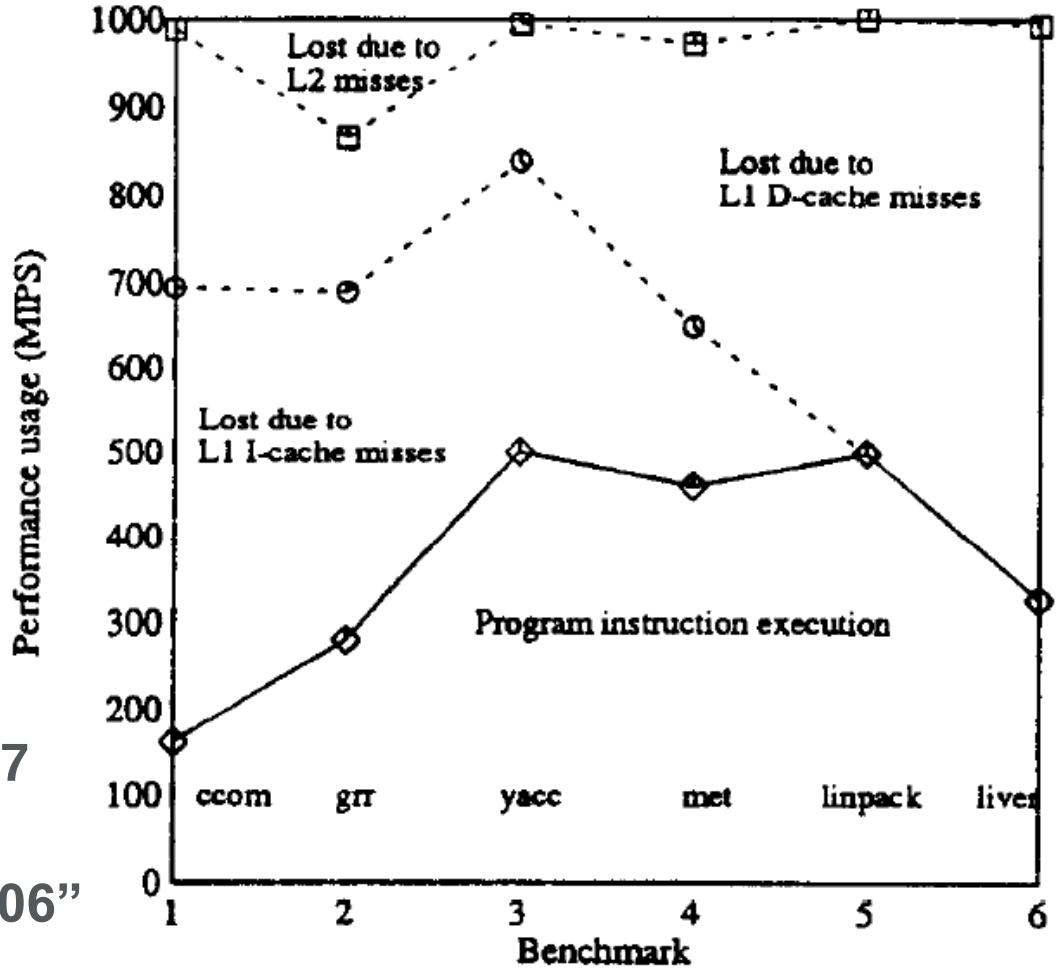
- **Cache hit time is important to reduce average memory access time**
- **However, reducing hit time comes at the expense of higher miss rates or higher energy consumption**
- **Cache Size Tradeoff**
  - Smaller caches are faster to access
  - However, smaller caches have higher capacity misses
- **Associativity Tradeoff**
  - Direct-mapped cache: faster access time, more conflict misses
  - Set-associative cache: slower access time, fewer conflict misses
- **Tag and Data Access Tradeoff**
  - Parallel tag and data access reduces hit time but wastes energy due to extra dynamic power
  - Sequential tag then data access is slower but saves energy
    - Typically done in L2, L3 etc.

Example: 4KB Direct-Mapped Cache with 16B Lines

program name	baseline miss rate instr.	data
ccom	0.096	0.120
grr	0.061	0.062
yacc	0.028	0.040
met	0.017	0.039
linpack	0.000	0.144
liver	0.000	0.273

Miss rates: Jouppi 1990 Table 2-2

- More recent analysis from SPEC 2006 and SPEC 2017 workloads: "A Reusable Characterization of the Memory System Behavior of SPEC2017 and SPEC2006" https://dl.acm.org/doi/fullHtml/10.1145/3446200



Performance: Jouppi 1990 Figure 2-2

Many Misses Caused by Conflicts

- **In direct-mapped caches, conflict misses represent significant percentage**
  - Average: 39% for D-cache, 29% for I-cache

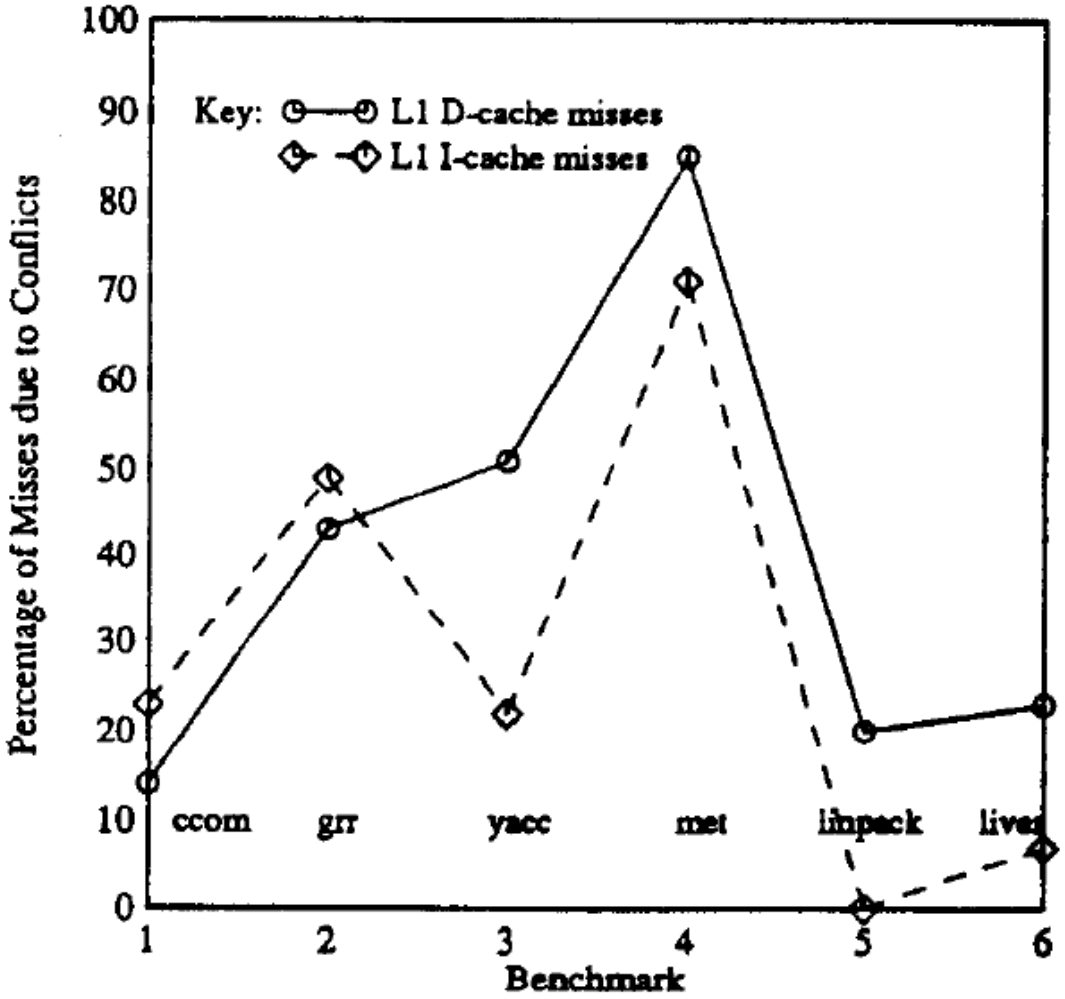
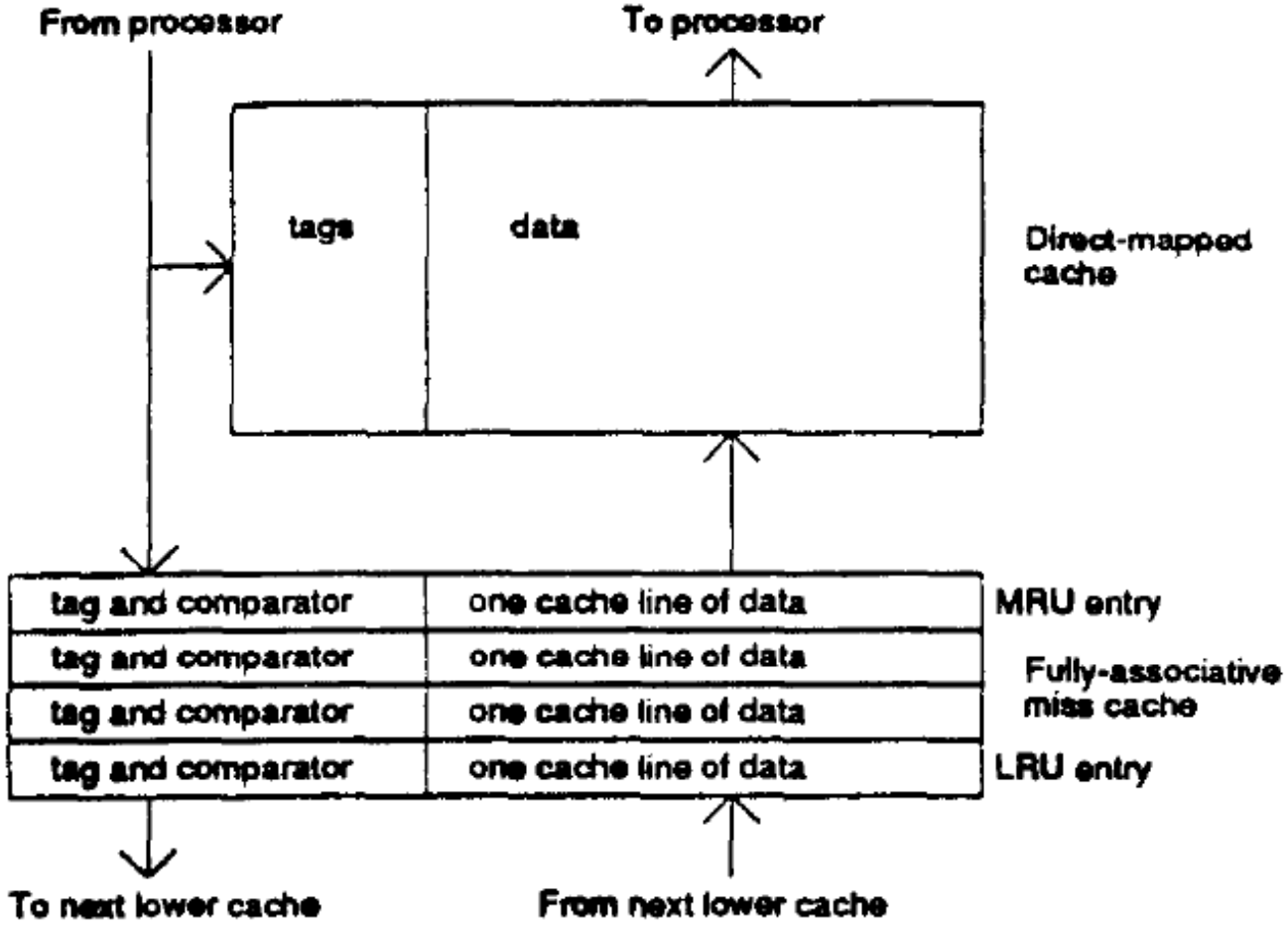


Figure 3-1: Conflict misses, 4KB I and D, 16B lines

Jouppi 1990 Figure 3-1

First Proposal: Miss Caching

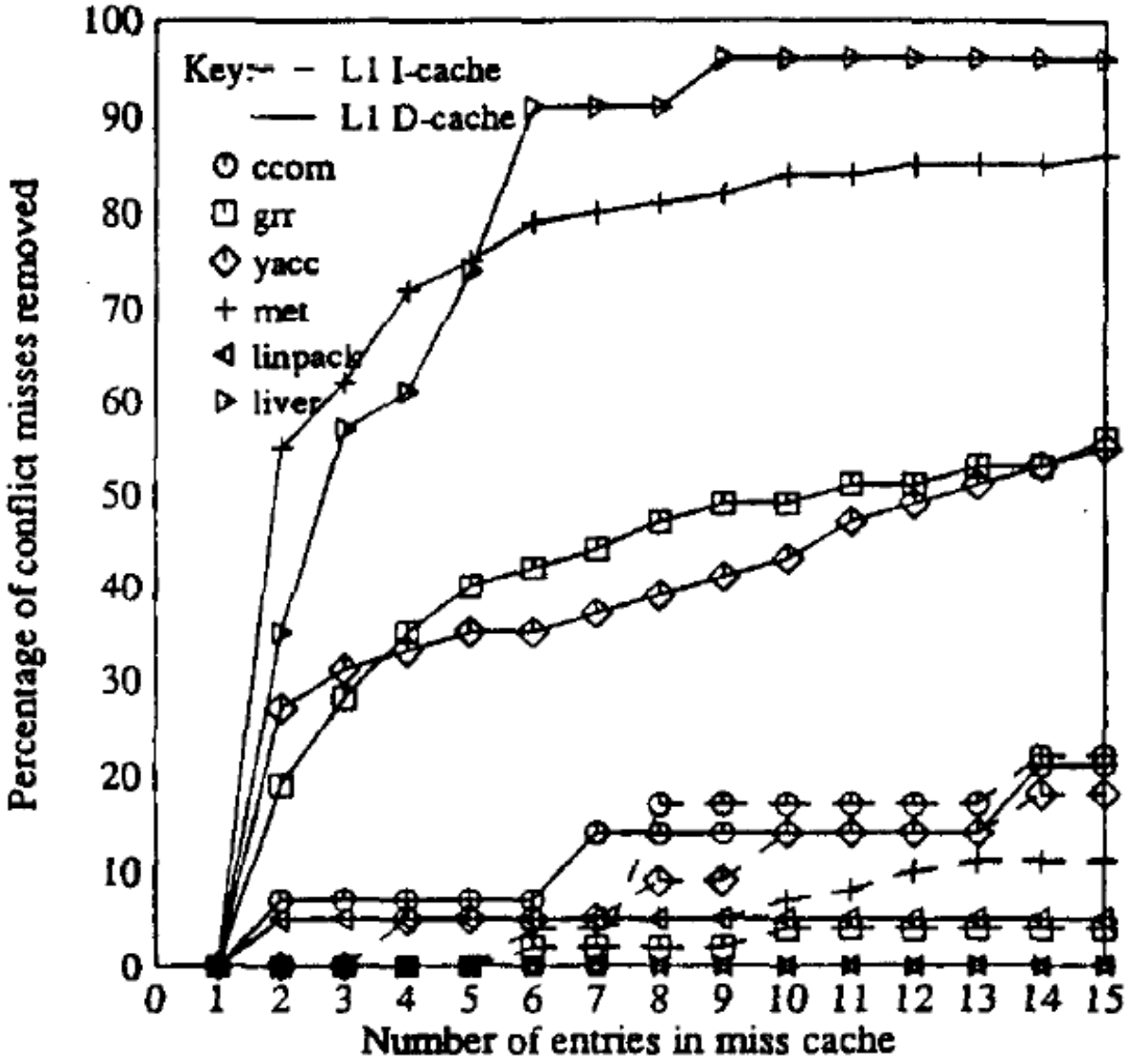
- **Miss Cache: Small cache placed between the L1 and L2 caches**
  - Provides additional associativity without increasing hit time in common case
  - Fully associative cache containing 2-5 lines
  - On a miss, data is returned to both L1 cache and miss cache



Jouppi 1990 Figure 3-2

Miss Cache Performance

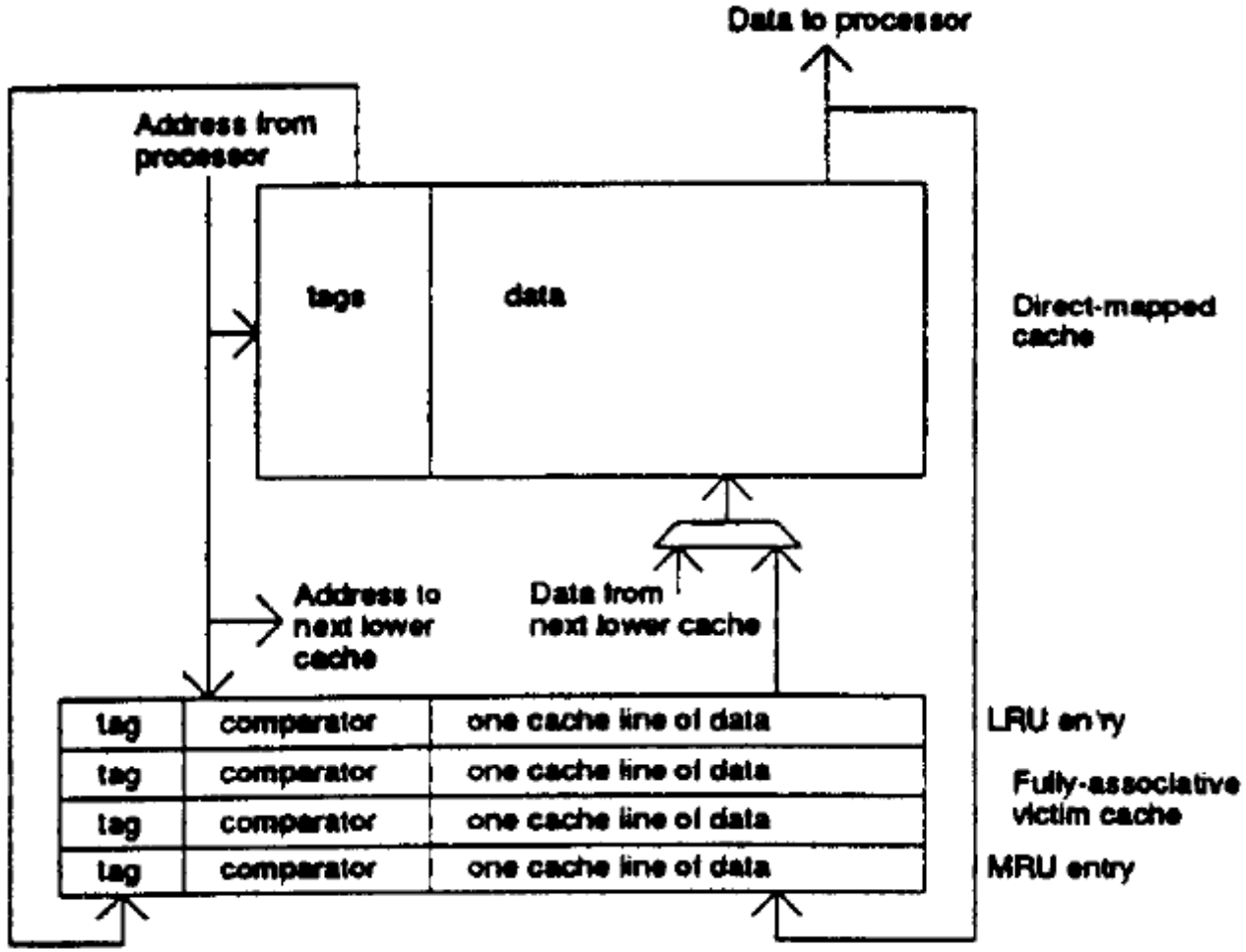
- More effective when %Conflict misses is high
- More effective for D-cache than I-cache. Why?
- Why do we need at least 2 entries?



Jouppi 1990 Figure 3-3

Second Proposal: Victim Caching

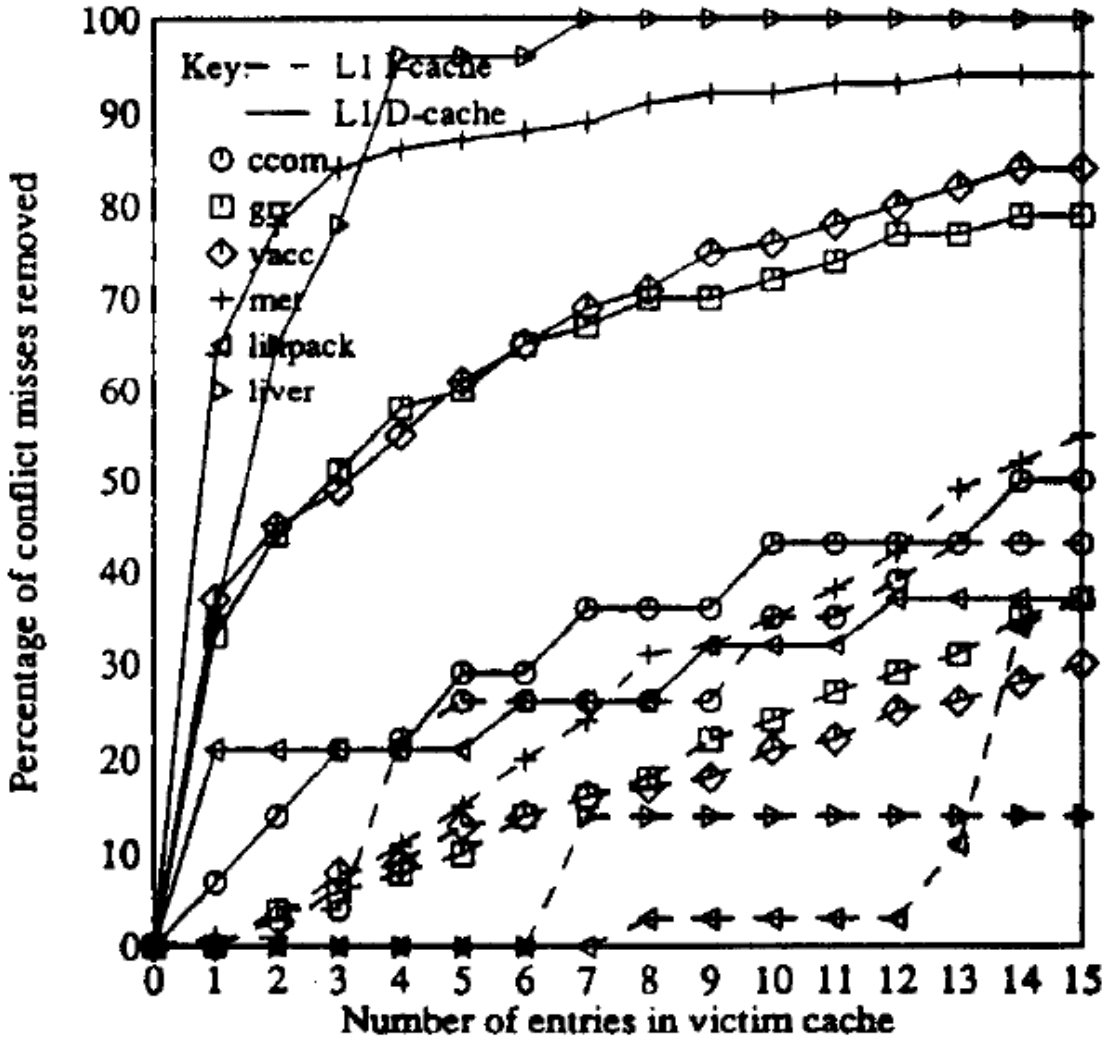
- **Disadvantage of Miss Cache: data redundancy**
  - Fill line inserted in both regular cache and miss cache
  - Needs at least two lines to be effective (i.e., increase the associativity of one cache set from 1-way to 2-way)
- **Victim Cache: On a miss, replacement victim line is placed in the victim cache**
  - Provides additional associativity without increasing hit time in common case
  - Even a single line can be effective
  - Always an improvement over miss caching



Jouppi 1990 Figure 3-4

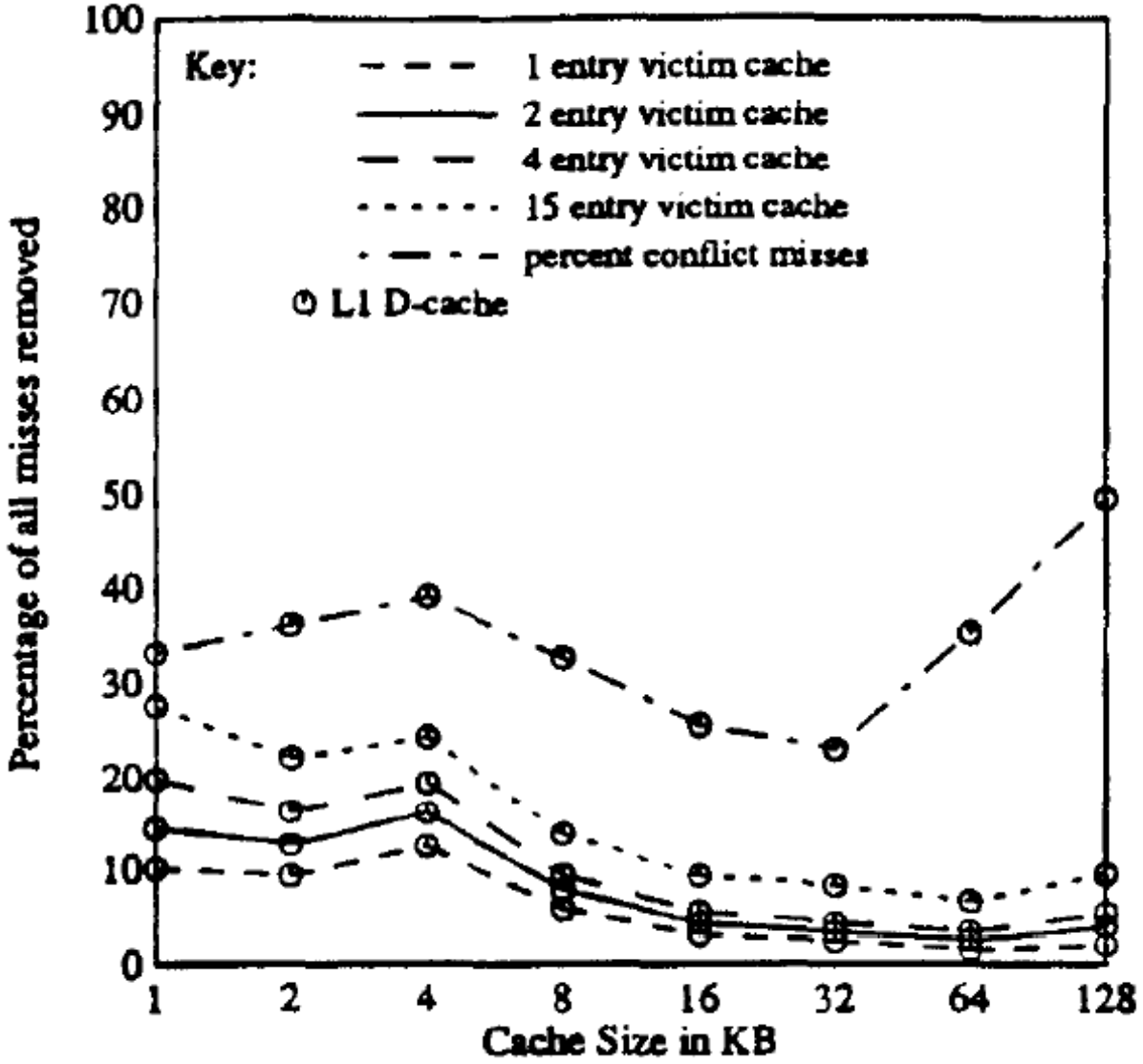
Victim Cache Performance

- More effective for D-cache than I-cache
- Always outperforms Miss cache

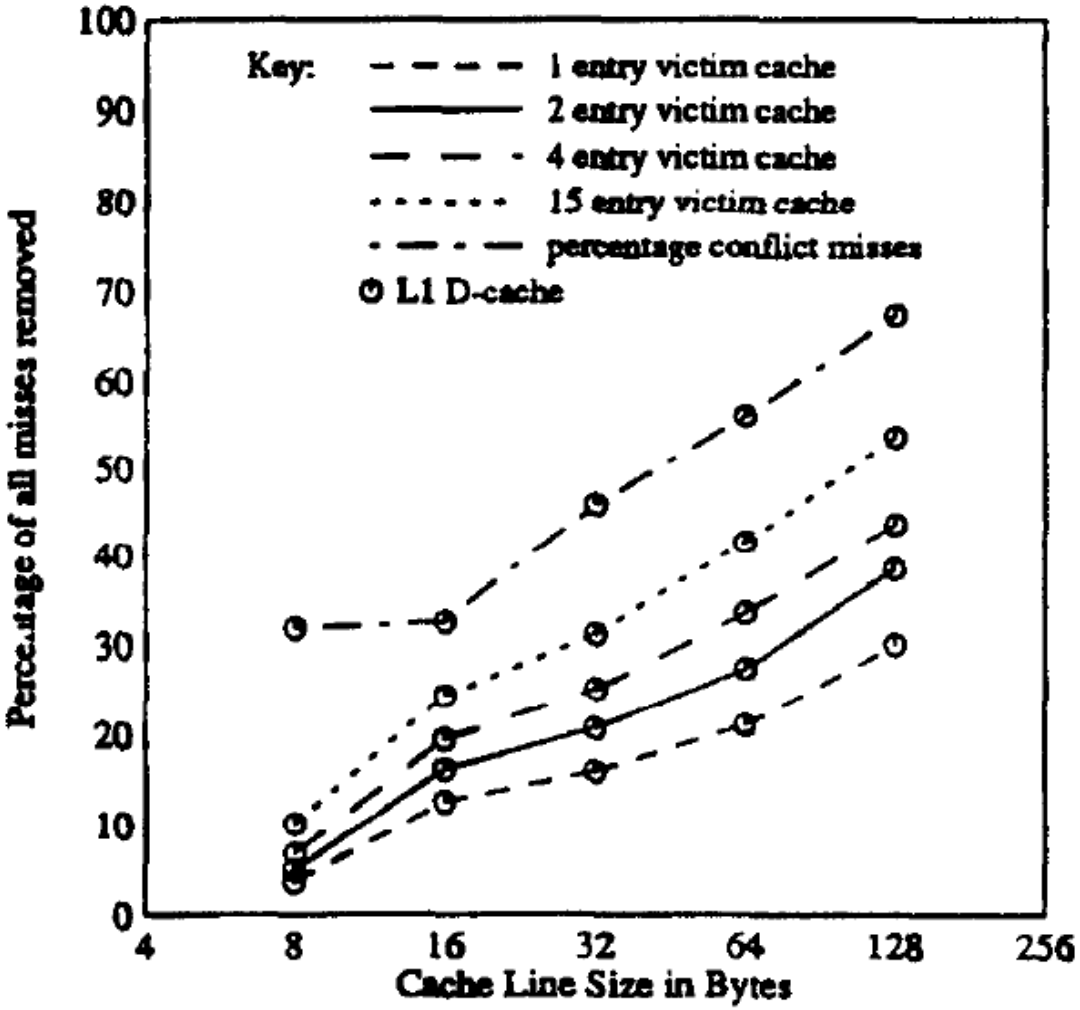


Jouppi 1990 Figure 3-5

Victim Cache Performance vs. Cache & Line Size



Jouppi 1990 Figure 3-6



Jouppi 1990 Figure 3-7

Prefetching

Prefetching

- **Bringing lines to the cache before being requested**
  - Can reduce compulsory and capacity misses
- **Requests to next level of memory hierarchy fall into two categories:**
  - **Demand miss:** Fill request due to cache miss
  - **Prefetch:** Fill request in anticipation of data request
- **Instruction and data access patterns are different (discuss)**

Prefetching Terminology

- **Timeliness: Measures whether the prefetch arrives early enough to avoid a miss**
  - ➢Even if miss is not totally avoided, miss latency is reduced
- **Prefetch Hit: Prefetched line that was hit in the cache before being replaced (miss avoided)**
- **Prefetch Miss: Prefetched line that was replaced before being accessed**
- **Prefetch rate: Prefetches per instruction (or 1000 inst.)**
- **Accuracy: Percentage of prefetch hits to all prefetches**
- **Coverage: Percentage of misses avoided due to prefetching**
  - ➢100 x (Prefetch Hits / (Prefetch Hits + Cache Misses))

Classification of Prefetched Lines

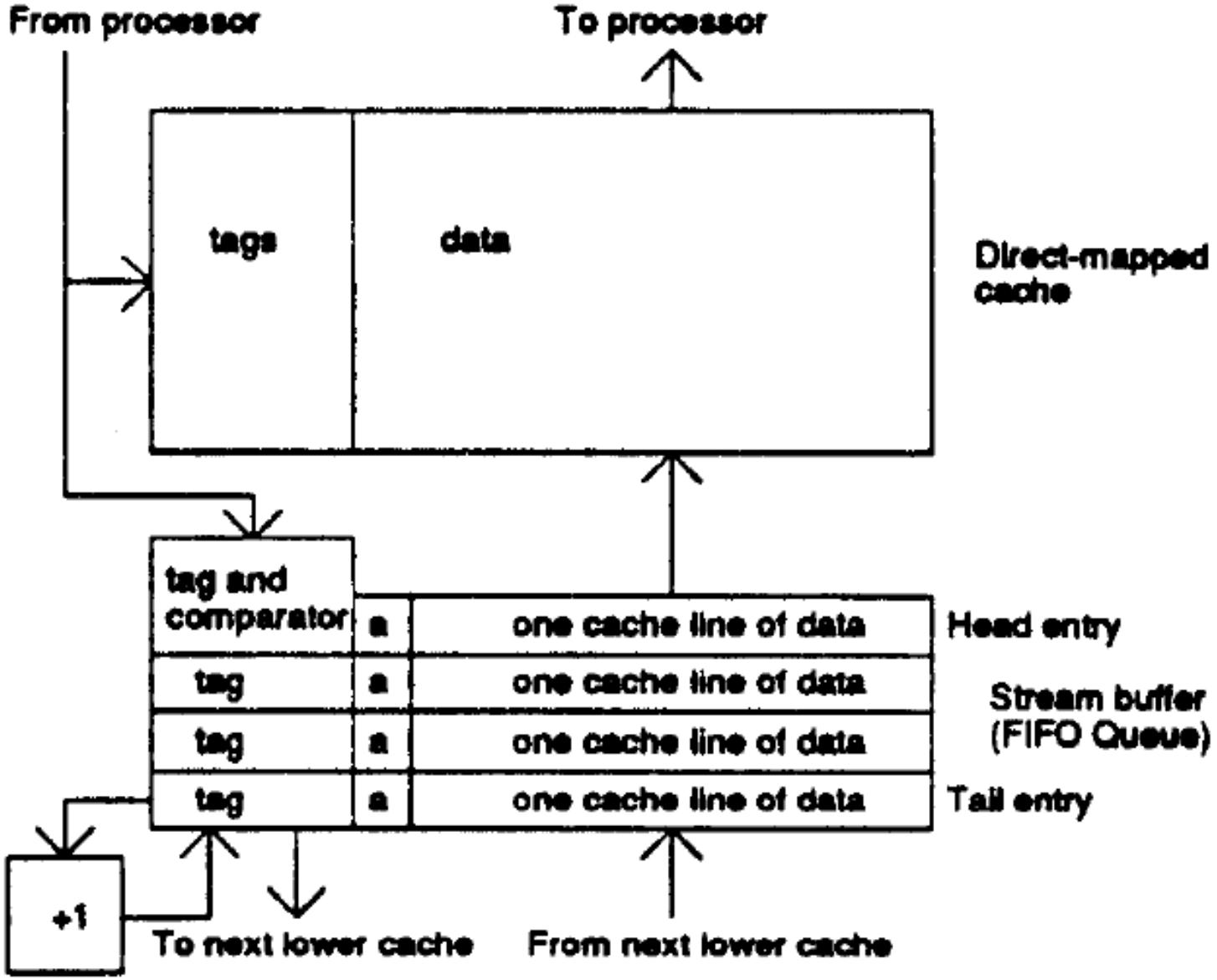
- **Useful Prefetch**
  - Prefetch hit before being replaced
  - Results in avoiding a cache miss
- **Useless Prefetch**
  - Prefetch is replaced before being accessed (prefetch miss)
  - Downside: Increases demand for cache bandwidth
- **Harmful Prefetch**
  - Prefetch is replaced before being accessed AND
  - Prefetch replaces a line that is requested later (cache pollution)
  - Results in an additional cache miss

Simple Prefetching Alternatives

- **Prefetch always**
  - Prefetch after every reference
  - Leads to significant demand on resources for next level in memory hierarchy
- **Prefetch on miss (Also called one block lookahead)**
  - On a miss, we prefetch the next sequential line as well
  - Cuts number of misses in a sequential stream in half
  - We can also implement N-block lookahead
- **Tagged Prefetch**
  - Each block has a tag status bit associated with it
  - On a prefetch, tag bit set to zero
  - On a hit, tag bit set to 1 (indicating prefetch hit)
  - When a block's status bit changes from 0 to 1, next block is prefetched

Stream Buffers

- Tagged prefetch may not be timely if cache lines are consumed faster than they are prefetched
- Need to start prefetching before a tag status bit transition takes place



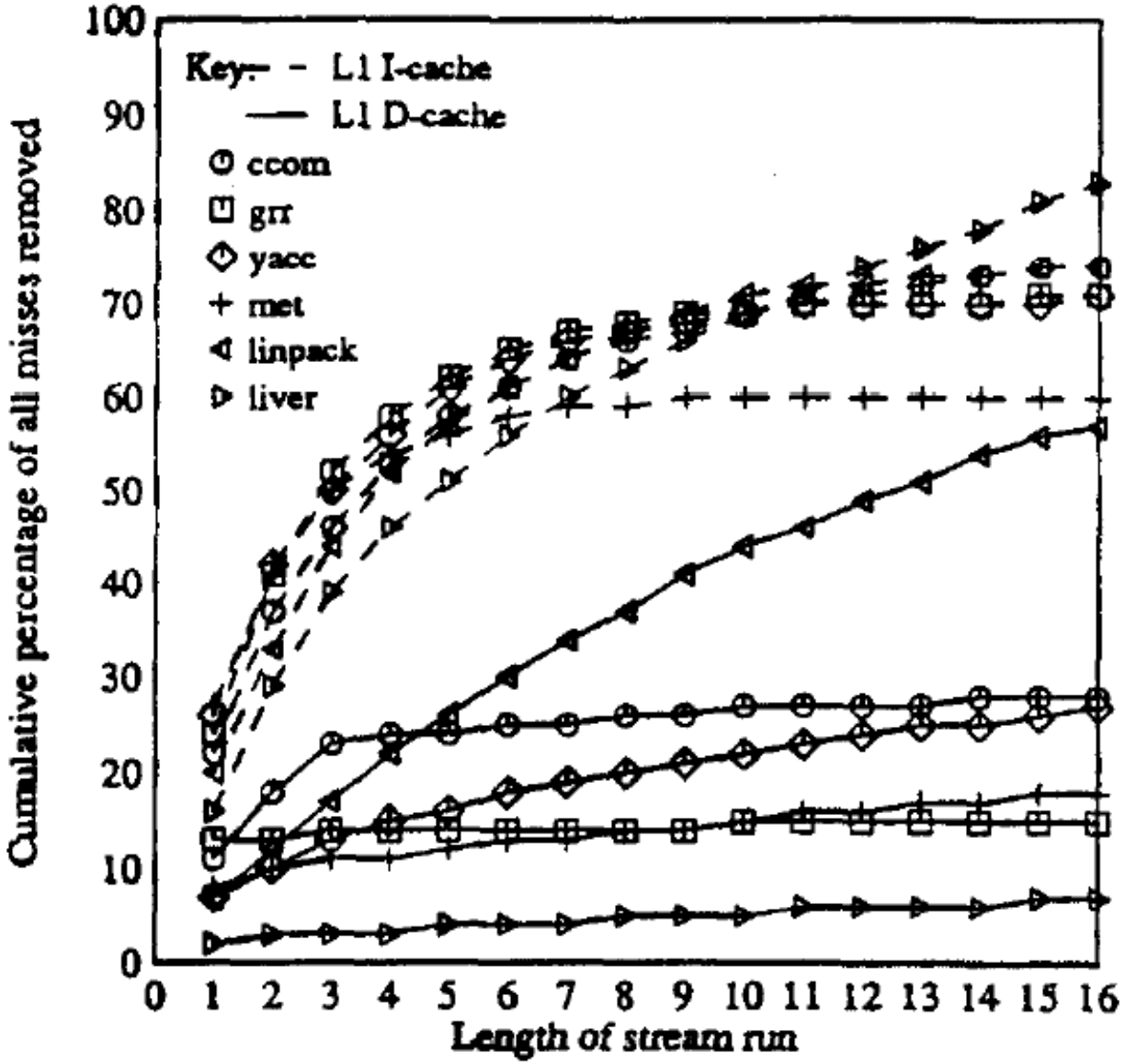
Jouppi 1990 Figure 4-2

Stream Buffer Operation

- **On a cache miss**
  - Stream buffer prefetches successive lines starting at the miss address
  - As each prefetch is sent out, we allocate an entry in the stream buffer and set available bit to false
  - When prefetch data returns, it is placed in buffer entry; available bit set to true
  - Prefetch lines are stored in the stream buffer not the cache to avoid cache pollution
- **On a cache miss and buffer hit**
  - Data loaded from stream buffer in one cycle
  - All buffer entries shift by one, new line prefetched to vacant entry
- **On a non-sequential miss**
  - Stream buffer flushed
  - Prefetching starts from new miss address (even if miss is present in another stream buffer entry)

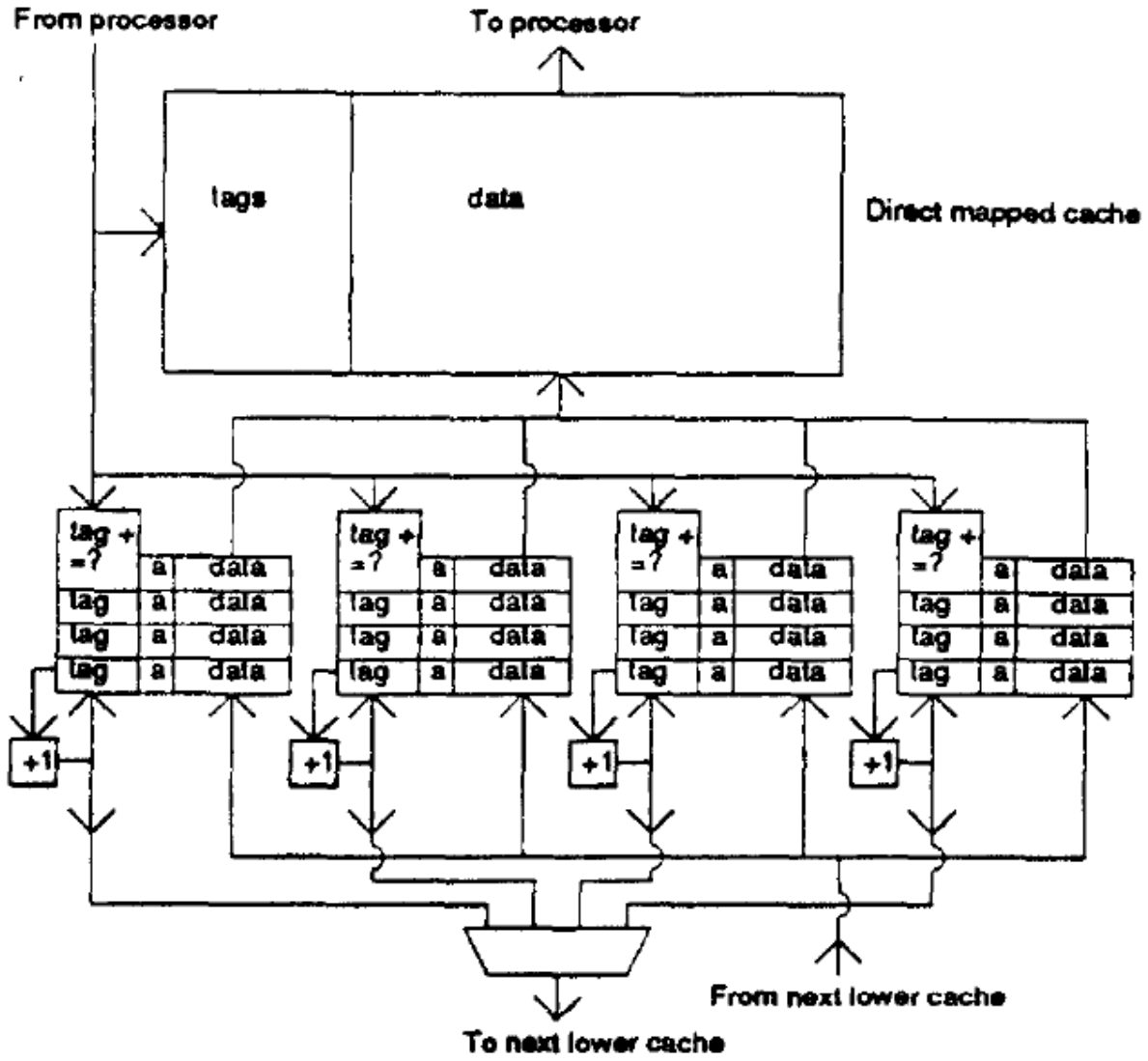
Stream Buffer Performance

- More successful for instructions compared to data. Why?

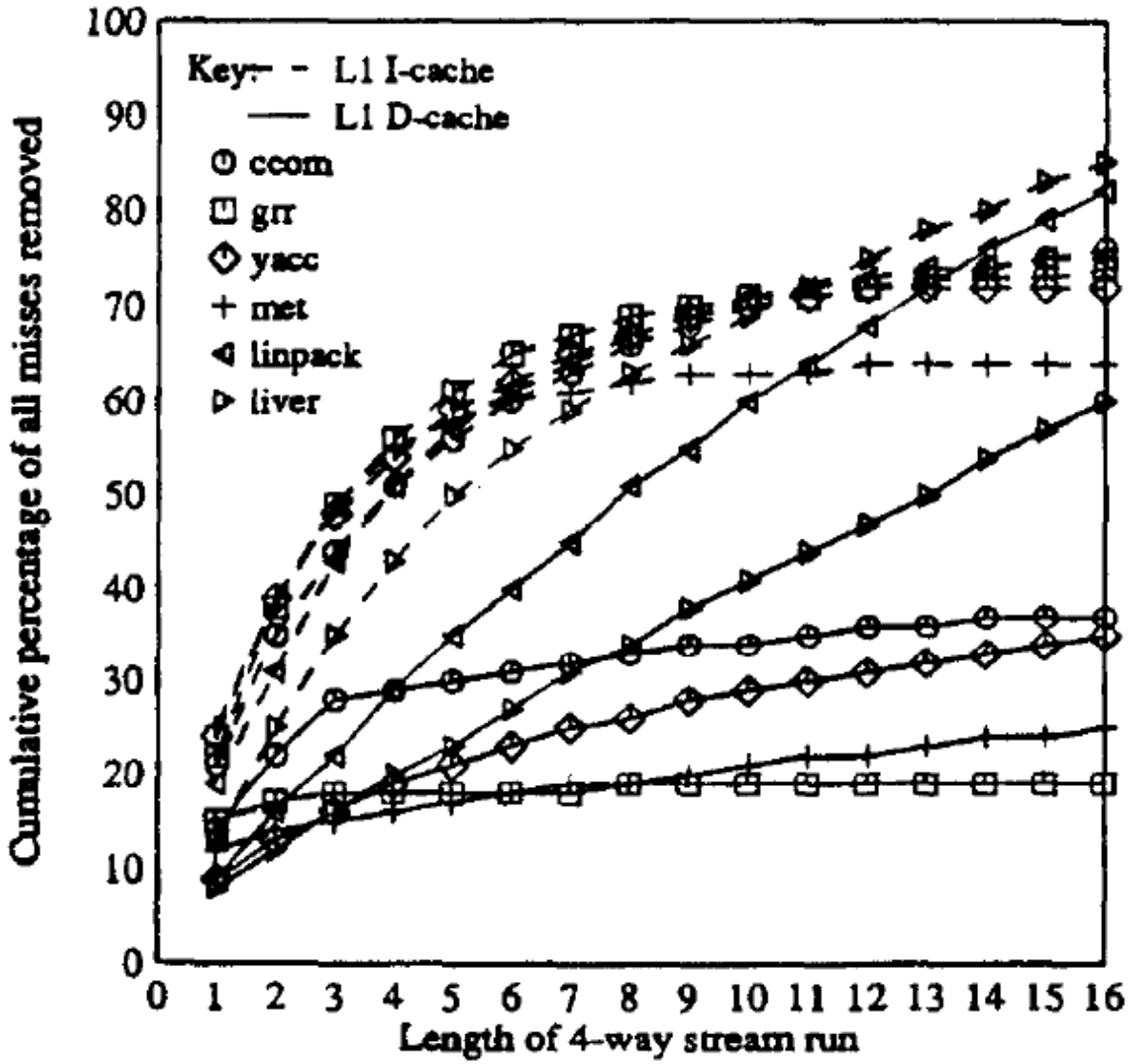


Jouppi 1990 Figure 4-3

Multi-Way Stream Buffer



Jouppi 1990 Figure 4-4



Jouppi 1990 Figure 4-5

Memory Access Patterns

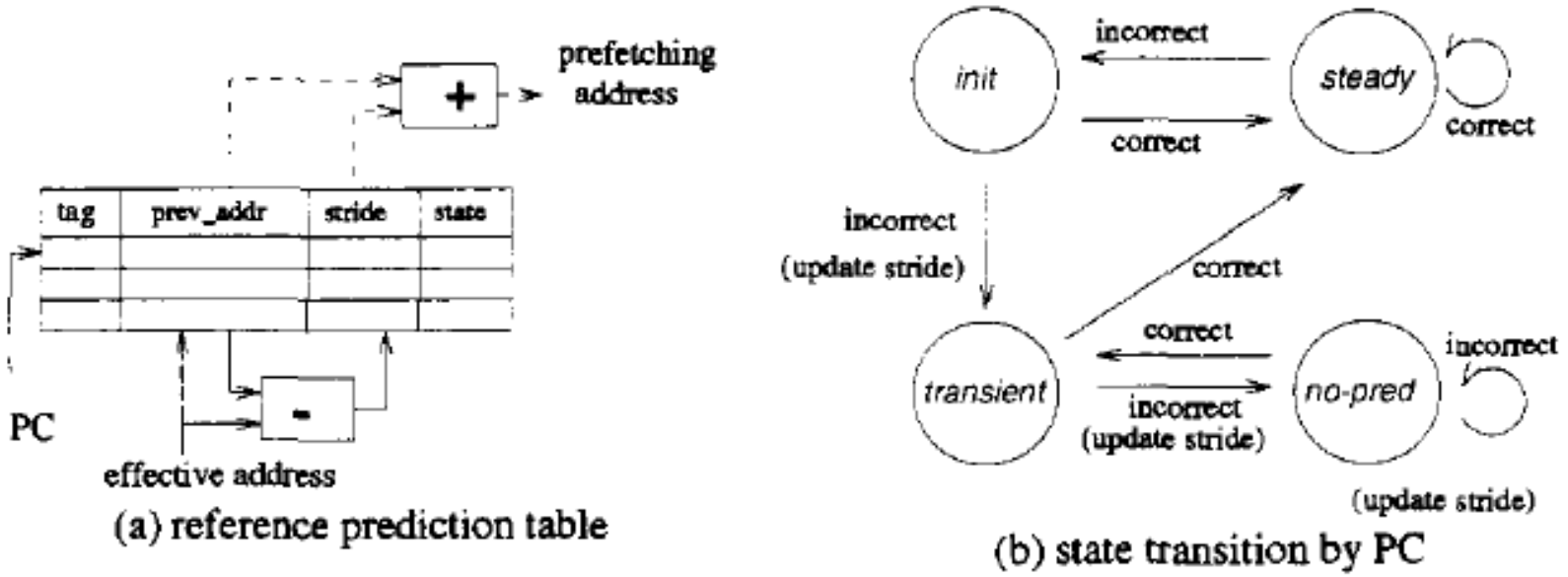
- **For successful prefetching strategies, we need to understand how programs access memory**

1. **Scalar / Zero Stride**
   - Example: simple variable references; A[i] in a loop indexed by j; A[i,j] in a loop indexed by k
   - Do not require prefetching. References will be in the cache due to temporal locality
2. **Streaming**
   - Example: Accessing cache lines A, A+1, A+2,…etc. OR A, A-1, A-2,…etc.
   - Can be prefetched using Next-Line Prefetcher or Stream Buffers
3. **Constant Stride**
   - Example: Accessing cache lines A, A+s, A+2s, …etc.; Accessing array elements A[i] in loop indexed by i; Accessing A[i,j] or A[j,i] in loop indexed by i or j
4. **Complex Access Patterns**
   - Any pattern that doesn't fit the above categories
   - Example: traversing a linked list, traversing a tree, traversing a graph

Stride-Based Prefetching (Chen & Baer)

- **Goal: Prefetching constant-stride access patterns**
- **Idea: Detect prefetching patterns based on load/store instruction PC**
- **Uses a reference prediction table to predict future memory references**
  - Tagged by PC
  - On a hit, compare current address with previous address and match stride
- **Lookahead prefetching**
  - Used to improve timeliness of prefetches
  - Uses a "lookahead PC"

Chen&Baer 1994 Fig 2



Software vs. Hardware Prefetching

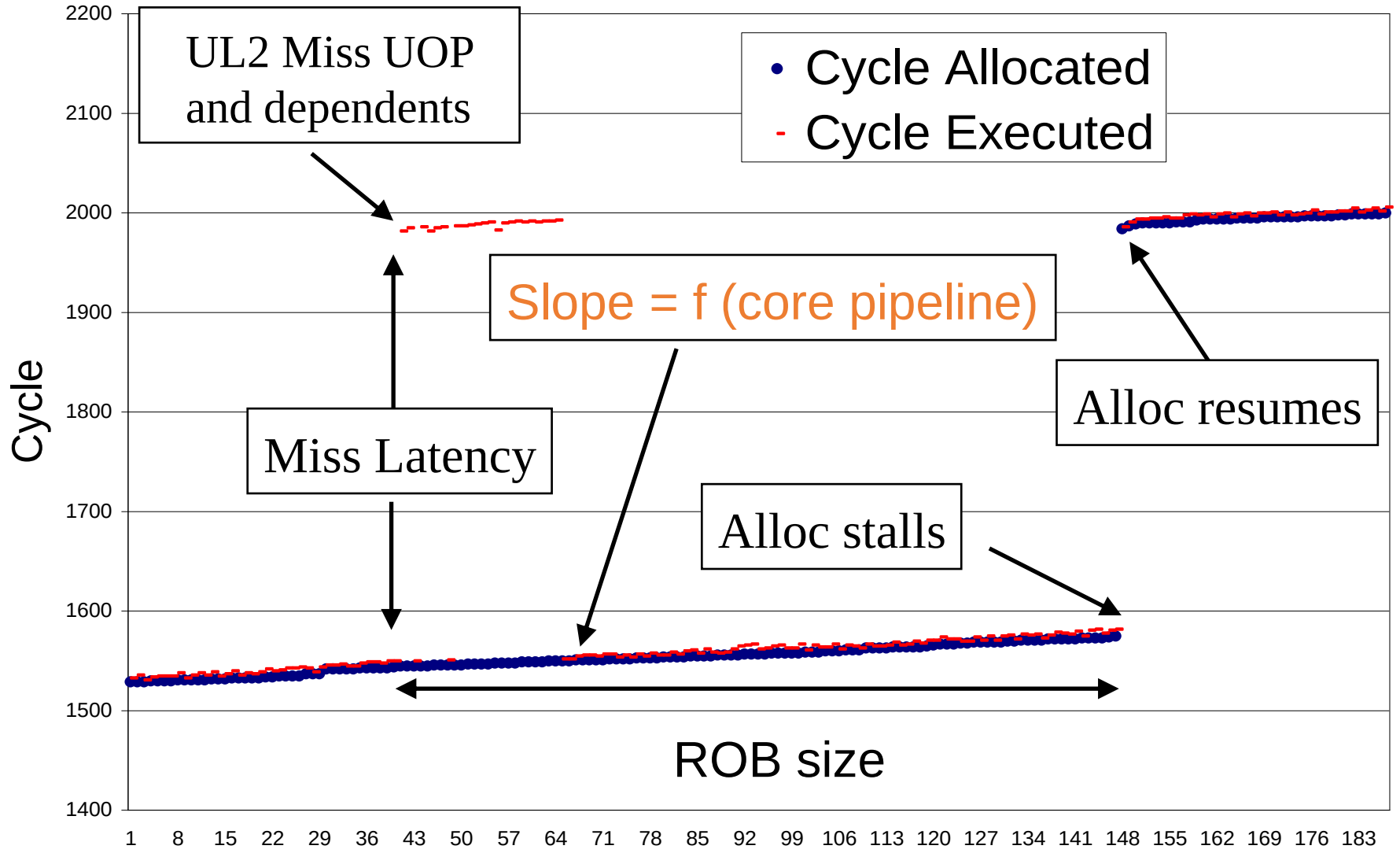
- **Some simple access patterns are easy to detect in software (e.g., streaming, constant stride)**
- **Some complex access patterns can also be prefetched by software**
  - Example: Link list traversal
    - Program can access one element and prefetch element->next
- **Software prefetching requires inserting "prefetch" instructions in the program by the compiler**
  - Advantage: Lower complexity in hardware (no prefetching structures)
  - Disadvantage: Larger programs
  - Disadvantage: Prefetches may not be timely so they arrive after demand accesses
- **Hardware prefetching can be more responsive due to knowledge of complex dynamic control flows of a program at runtime**

Prefetching Complex Patterns: Runahead Execution

- **How can we predict which address to prefetch next?**
- **Intuition: The program dynamic execution is the best predictor**
- **Idea: When the head of the ROB is a cache miss, checkpoint architectural state then continue to execute and "pseudo-retire" instructions**
  - This frees up space in the instruction window for more instructions
  - Some of these instructions may be other cache misses, triggering prefetches
  - When initial miss returns, restart the pipeline from checkpoint
  - When the next memory access occurs, the request would be already out and the data could possibly be in the cache.
- **Runahead expands effective instruction window size (more misses-under-miss, higher MLP)**

Cache Miss Analysis

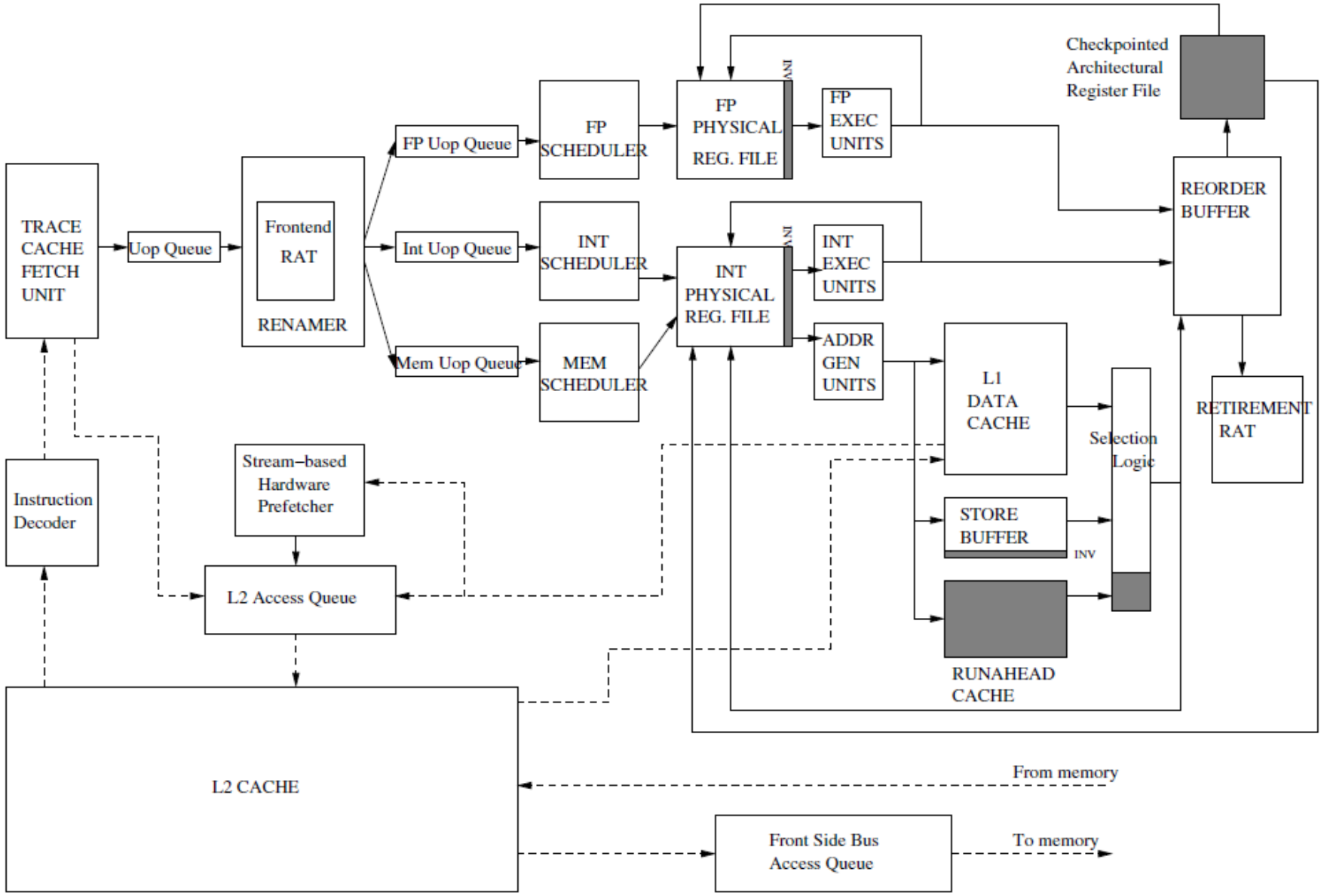
Figure from Srinivasan et al. "Continual Flow Pipelines" talk at ASPLOS 2004



Runahead Expands instruction window size, so more misses can be issued to memory simultaneously

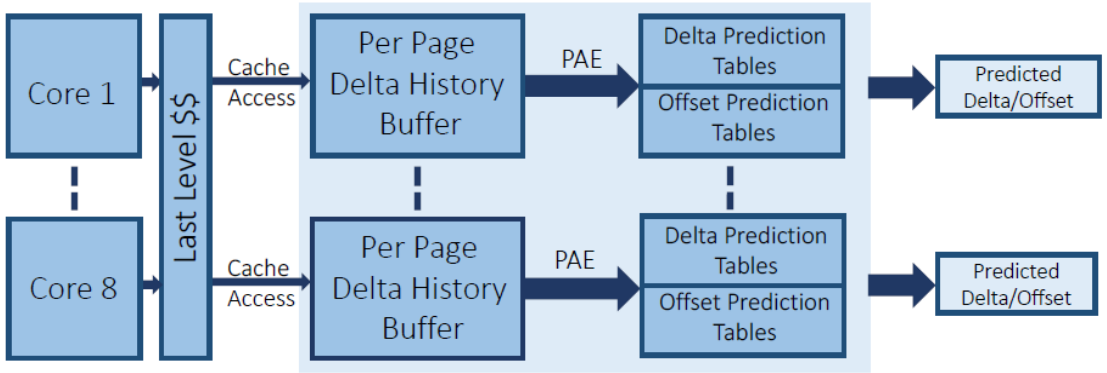
Runahead Execution: Hardware

Mutlu et al. 2003 Fig 2

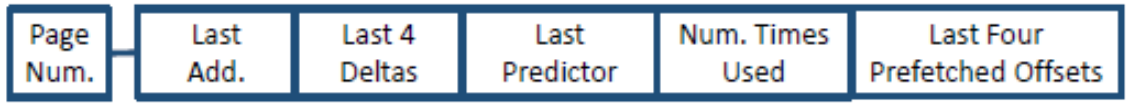


Variable Length Delta Prefetcher (VLDP)

- **Targets complex memory access patterns**
  - ➢ Example of repeated strides in real workloads: (-24, +25); (-24, -24, +49); (+2, +3, +4); (-1, +3, -1, +4)
- **Idea: Build delta (i.e., stride) histories between successive cache line misses within a page, then use history to predict accesses in other pages**
  - ➢ Uses multiple prediction tables that store predictions based on different input history lengths
    - ❑ First table uses most recent delta to predict next miss
    - ❑ Second table uses most recent two deltas to predict next miss,…etc.
    - ❑ VLDP uses table with longest history that has a matching entry to prefetch (similar to TAGE)



Shevgoor et al. 2015, Figure 1: VLDP Overview



Shevgoor et al. 2015, Figure 2: Delta History Buffer Entry

Cache Replacement and Insertion Policies

Overview

- **Cache replacement policy:**
  - ➢ On a cache line fill, which victim line to replace?
  - ➢ Only applicable to set-associative caches
    - ❑ Direct-mapped caches have only one line per set
  - ➢ Example: LRU
- **Cache insertion policy:**
  - ➢ When a cache line is filled, what would be its priority in the replacement stack?
  - ➢ LRU: fill line is inserted in “Most Recently Used” position
  - ➢ Other policies: LIP, BIP, DIP
  - ➢ Dead block prediction helps determine lines that won’t be reused (either bypassed or inserted in LRU position)

Optimal Replacement (OPT): Belady's Algorithm

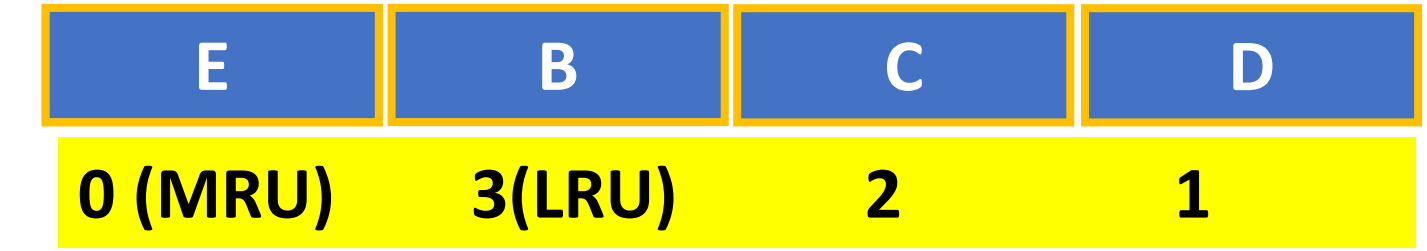
- **Replace the line that will not be needed for the longest time into the future**
- **Example (4-way cache)**
  - Access order A, B, C, D, E, A, B, D, A, B, D, A, E, B, C

Line C is furthest into the future, so replace C with E

- **Requires knowledge of future memory accesses (not practical)**
  - Other policies that do not require future knowledge (e.g., LRU) are used in real systems
  - Some recent research works attempt to predict future references and use them to approximate Belady's algorithm

Least-Recently Used (LRU) Policy

- **Replace the line that was referenced furthest in the past**
- **Example (4-way cache)**
  - Access order A, B, C, D, E, A, B, D, A, B, D, A, E, B, C



Line A is in LRU position, so replace A with E

- **Issues**
  - Requires tracking order for each line in the cache
  - Requires updating order on every access to a cache set (many read-modify-write operations)
  - Poor performance for streaming access patterns that don't fit in the cache

Other Replacement Policies

- **FIFO: Replace oldest allocated line**
- **Most Recently Used (MRU)**
  - Replaces the line that was most recently used
- **Least Frequently Used (LFU)**
  - Replaces line that has been used less often than others
  - Requires tracking frequency of access via counters (updated on every access)
  - Counters need to decay or lines will remain in cache forever
- **Random Replacement (RR)**
  - Replace a line in the set at random
  - Helps if recency of use is not a factor in predicting future use
- **Pseudo-LRU (PLRU)**
  - Replaces “one of the least recently used lines”
  - Requires fewer bits to track and update on every access
  - Explanation of how it works: https://en.wikipedia.org/wiki/Cache_replacement_policies
- **…**

Insertion Policies

- **Insertion policy determines where a fill line is inserted in the LRU stack**
- **Static Insertion Policies use the same policy always for all workloads**

1. **LRU replacement uses "MRU Insertion Policy": Insert new line in MRU position**
2. **MRU replacement uses "LRU Insertion Policy" (LIP): Insert new line in LRU position**

- **Intuition for LIP: For cyclic sequential accesses that exceed number of ways, new line will be accessed further into the future**
  - Example (4-way cache): A,B,C,D,E,A,B,C,D,E,…. (Causes thrashing in LRU)
- **LIP Adversarial Case:**
  - A,B,C,D,E,A,B,C,D,E,F,G,H,I,J,F,G,H,I,J, F,G,H,I,J, F,G,H,I,J, F,G,H,I,J,…
  - Only 3 hits, all other accesses are misses

Bimodal Insertion Policy (BIP)

- **Similar to LIP except that it occasionally inserts lines into MRU position with a small probability**
- **Bimodal throttle parameter (ε) controls the probability of inserting lines in MRU position**
  - BIP is the same as LRU when ε = 1; same as LIP when ε= 0

Table 3: Hit Rate for LRU, OPT, LIP, and BIP

	$(a_1 \cdots a_T)^N$	$(b_1 \cdots b_T)^N$
LRU	0	0
OPT	$(K-1)/T$	$(K-1)/T$
LIP	$(K-1)/T$	0
BIP	$(K-1-\epsilon \cdot [T-K])/T$ $\approx (K-1)/T$	$\approx (K-1-\epsilon \cdot [T-K])/T$ $\approx (K-1)/T$

Qureshi et al. 2007, Table 3

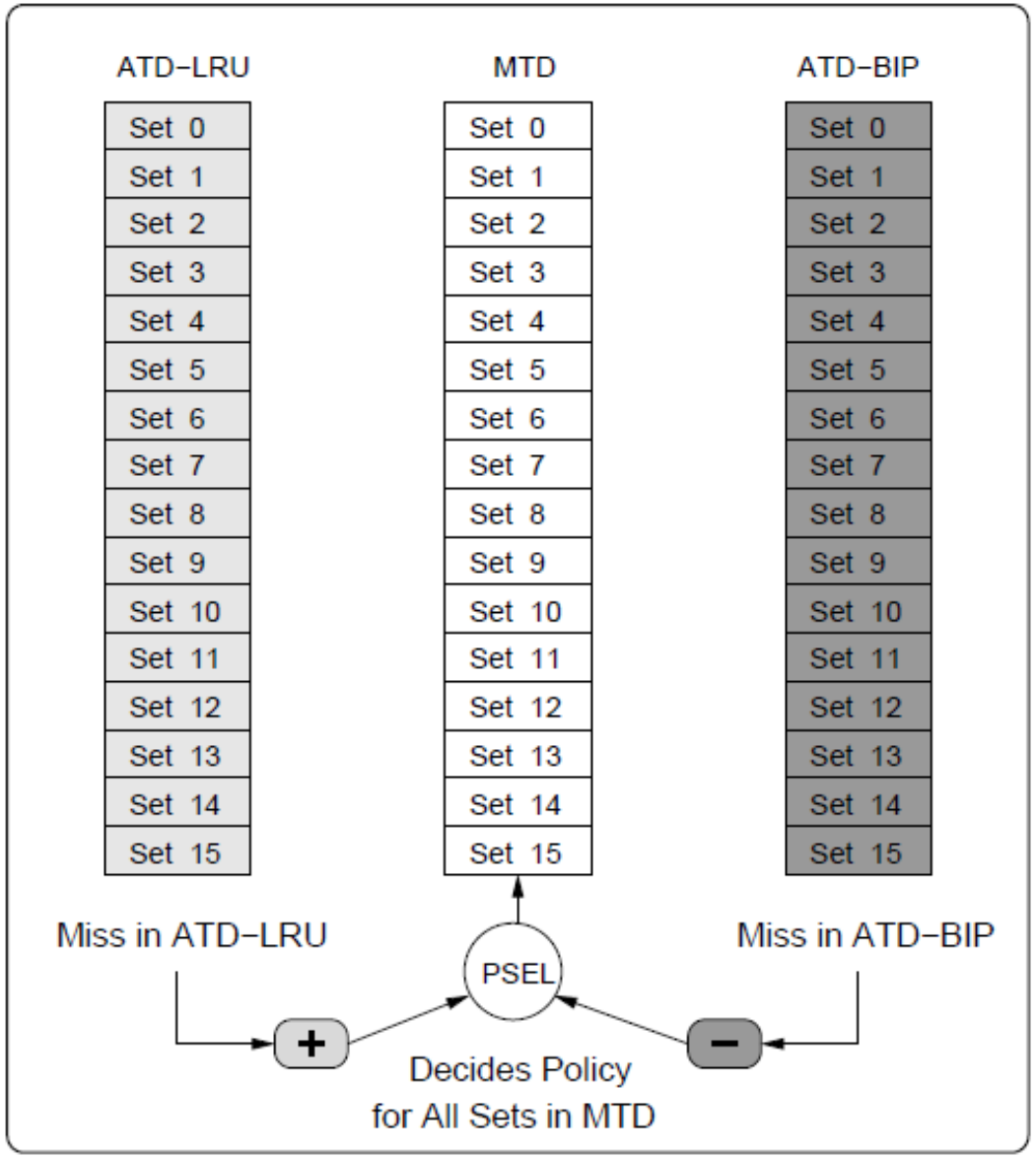
- **Workloads have different phases with different access patterns, so a more dynamic policy could be needed**

Dynamic Insertion Policy (DIP)

- **Intuition: Some workloads are LRU-friendly while others are BIP-friendly**
- **DIP: Dynamically determine at runtime which policy is better, then apply the best policy to the whole cache**
- **Tracking replacement states used to determine which policy is better**
  - Use an auxiliary tag array (ATD) that tracks cache replacement stack
  - ATD keeps track of extra tags that follow either BIP or LRU
  - ATD-BIP keeps track of lines that will be cached using BIP, ATD-LRU keeps track of lines that will be cached using LRU
  - Use saturating counter PSEL to determine which policy is better:
    - Incremented on LRU miss, decremented on BIP miss
    - Most significant bit determines which policy is better
  - Better policy used in the main tag array (MTD) for the whole cache

DIP Implementation

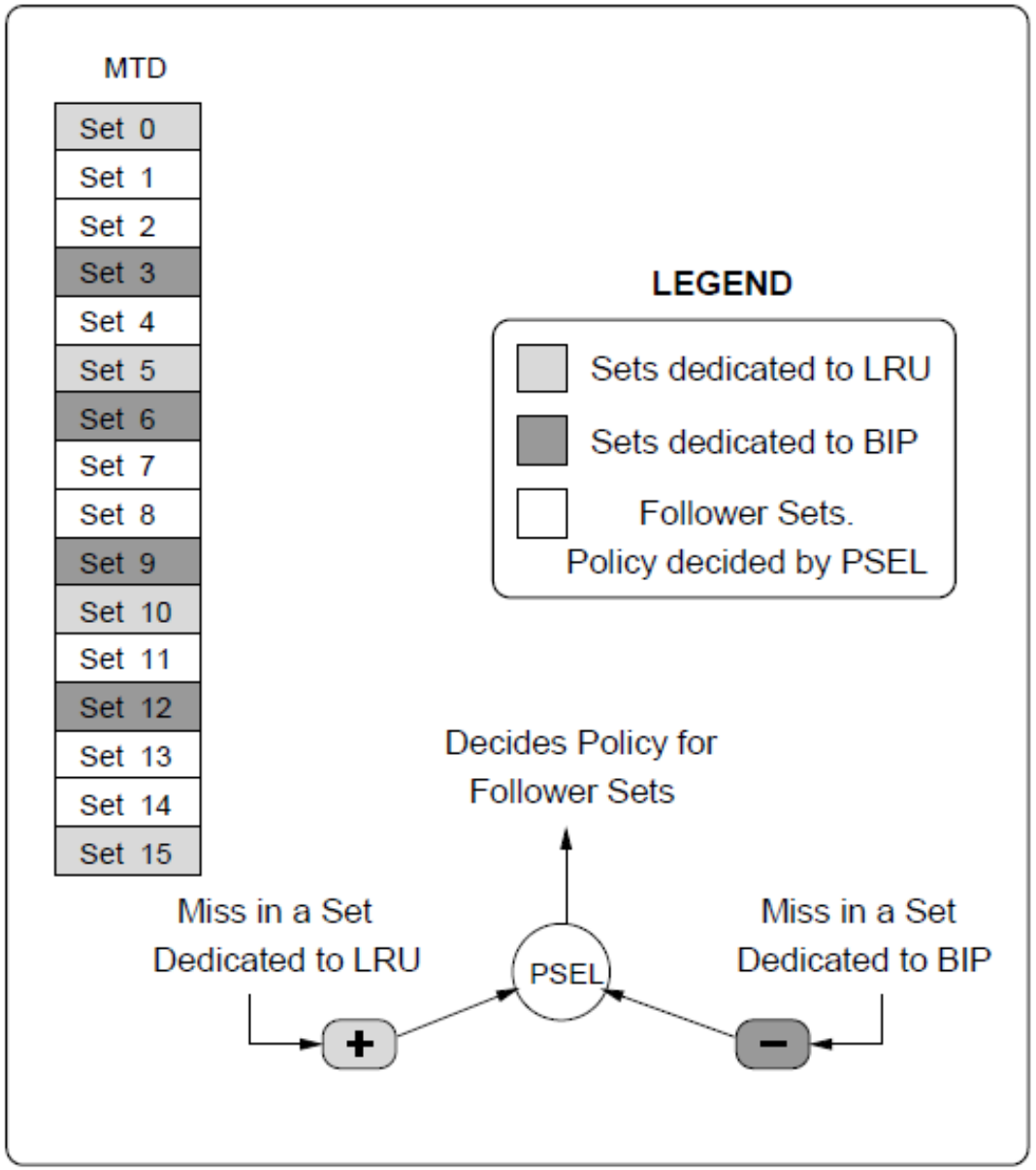
- **Track replacement stacks for both LRU and BIP**
- **Issues:**
  - Tag array increases by 3x
  - Dynamic power increases due to updating replacement state of all three tag arrays (MTD, ATD-LRU, ATD-BIP)
- **Do we really need to track all sets to decide which policy is better?**



Qureshi et al. 2007, Figure 9

DIP with Set Dueling

- **Use *Dynamic Set Sampling (DSS)* to select a few sample sets**
  - Some sample sets use BIP, others LRU
- **Best policy determined by set dueling**
  - Only sample sets used to update counter
  - Remaining cache sets follow best policy determined by counter



Qureshi et al. 2007, Figure 10

Re-Reference Interval Prediction Policy (RRIP)

- **Predict when cache lines are going to be re-referenced**
  - Each cache line has a “re-reference prediction value” (RRPV) which determines how soon it is is going to be re-referenced
  - RRPV values are quantized with n-bits (e.g., use 2-bits to quantize into 4 buckets)
  - RRPV=0 indicates near-immediate reuse, RRPV=3 indicates distant reuse
- **Idea: Predict new cache lines will not be re-referenced soon**
  - Insert new line with RRPV ≠ 0
  - On hit, RRPV updated to 0
- **Problem: Always using the same prediction for all insertions thrashes cache**
- **Dynamic Re-Reference Interval Prediction (DRRIP)**
  - Dynamically inserts new blocks between different RRPV values based on set dueling
  - RRPV = 2^n-1 could bypass cache (predicted dead on arrival)

Announcements

- **Reading Assignments**
  - N. Jouppi, “Improving Direct-Mapped Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffer,” ISCA 1990 (Read)
  - T.F. Chen and J.L Baer, “Effective Hardware-Based Data Prefetching for High-Performance Processors,” IEEE Transactions on Computers, 1995 (Skim)
  - O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003 (Skim)
  - M. Shevgoor et al., “Efficiently Prefetching Complex Address Patterns,” MICRO 2015 (Skim)
  - M. Qureshi et al., “Adaptive Insertion Policies for High-Performance Caching,” ISCA 2007 (Read)
  - A. Jaleel et al., “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP),” ISCA 2010 (Skim)
- **Assignment 3 out today. Due Oct 31.**
- **Project timeline: Teams due Oct 24. Proposals due Oct 28.**