



# CS698Y: Modern Memory Systems

## Lecture-8 (Caches)

---

**Biswabandan Panda**

**[biswap@cse.iitk.ac.in](mailto:biswap@cse.iitk.ac.in)**

**<https://www.cse.iitk.ac.in/users/biswap/CS698Y.html>**

# Flow of the Module

**Cache Management Policies**

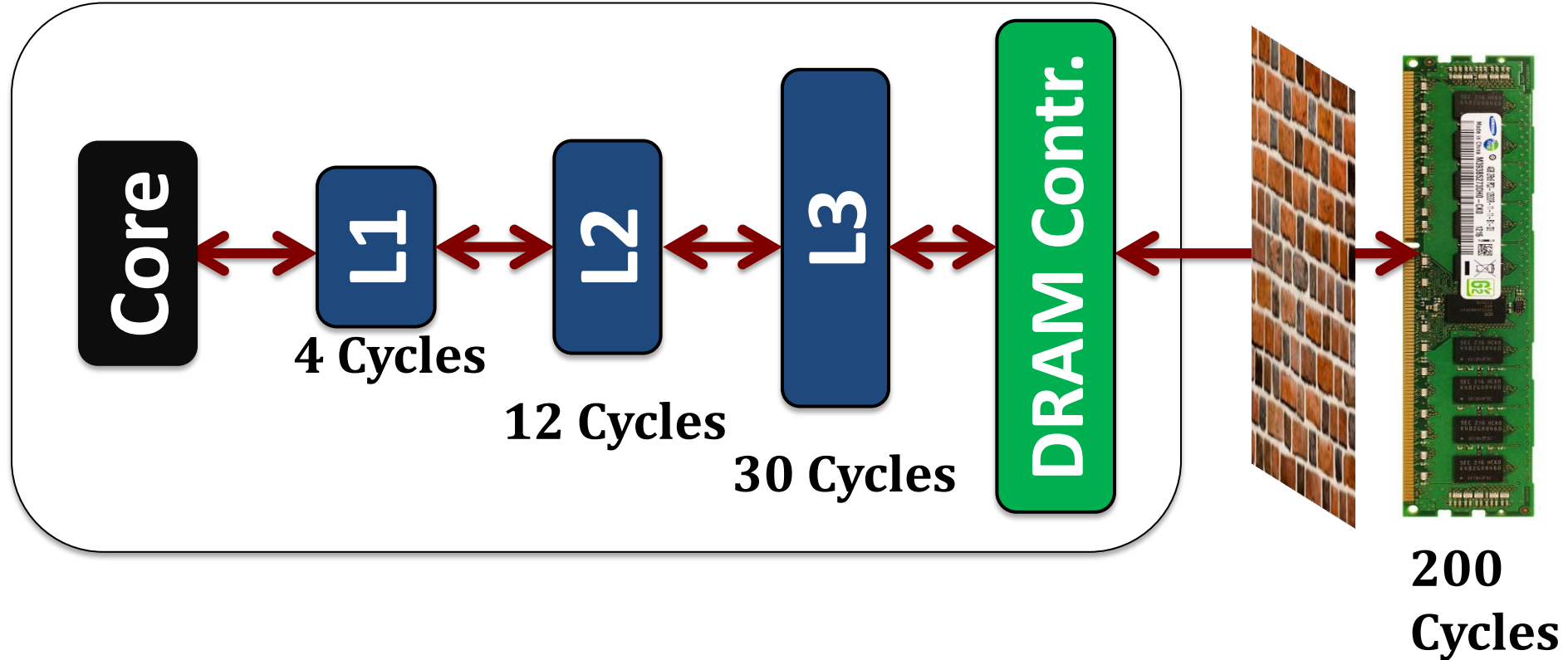
**Cache Hierarchies**

**Hardware Prefetching**

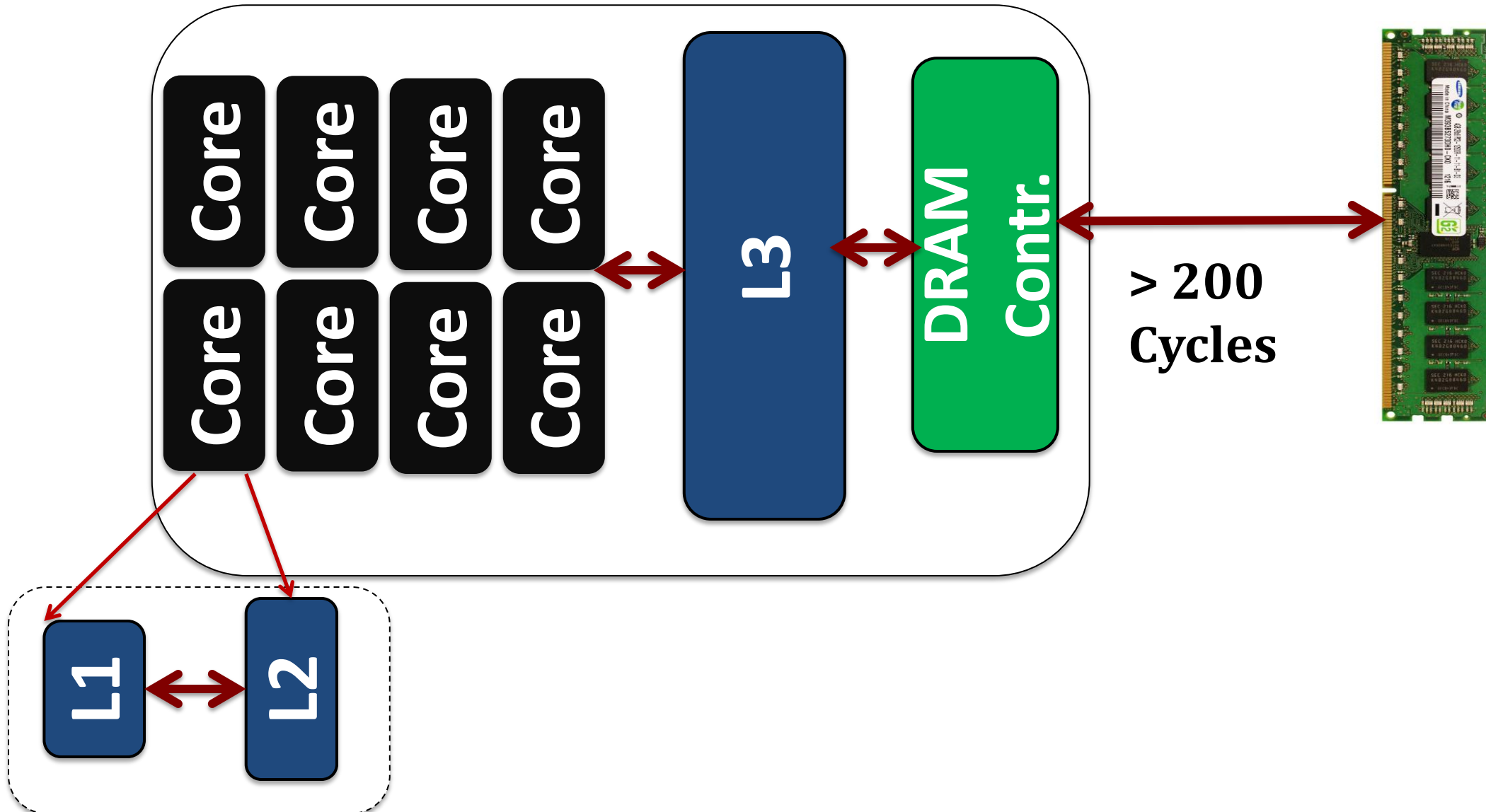
**Cache Compression**

**Non-uniform Caches**

# Caches in Single-core System



# Caches in Multi-core



# Latency Numbers

**L1**

**Few Cycles**

**L2**

**Tens of Cycles**

**L3**

**Two to three  
times of L2**

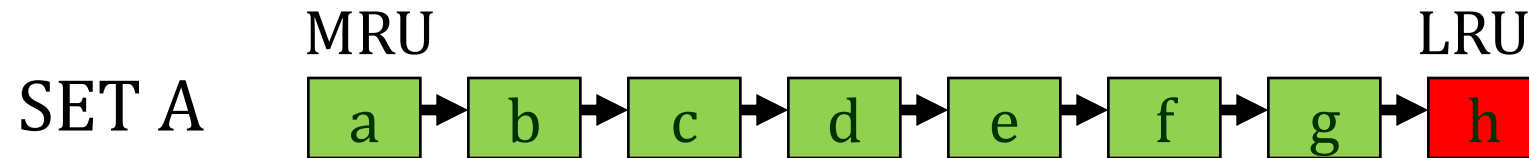


**Hundreds of cycles**

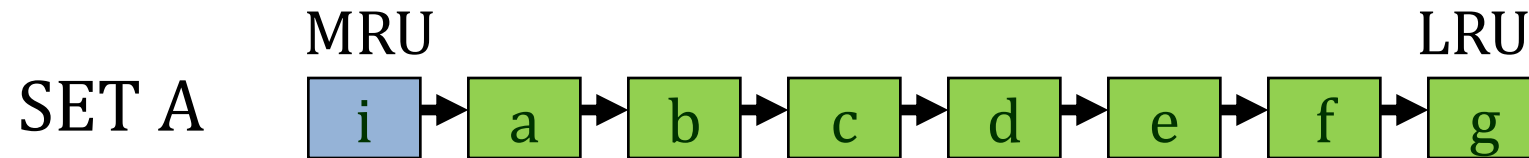
**Our Goal:**  
**To minimize off-chip DRAM accesses**

# Cache Replacement (LRU) - 101

**Cache Eviction Policy: On a miss (block  $i$ ), which block to evict (replace) ?**



**Cache Insertion Policy: New block  $i$  inserted into MRU.**



**Cache Promotion Policy: On a future hit (block  $i$ ), promote to MRU**

**LRU causes thrashing when working set  $>$  cache size**

# Common Access Patterns [RRIP, ISCA 10]

**Recency friendly**  $(a_1, a_2, \dots, a_k, a_{k-1}, \dots, a_2, a_1)^N$

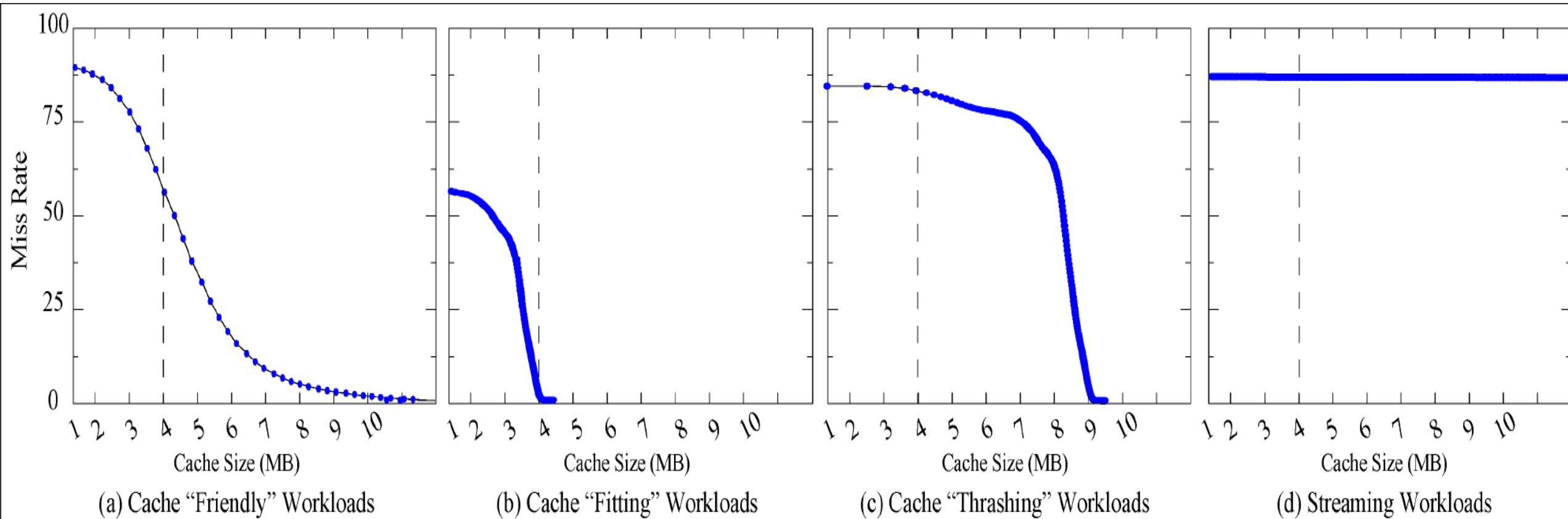
**Thrashing**  $(a_1, a_2, \dots, a_k)^N$   $[k > \text{cache size}]$

**Streaming**  $(a_1, a_2, \dots, a_\infty)^N$

**Combination of above three**



# Types of Workloads (Baseline 4MB Cache)



# Limitations of LRU

LRU exploits **temporal locality**

**Streaming data ( $a_1, a_2, a_3, \dots, a_\infty$ ):**

No temporal locality,  
No temporal reuse

**Thrashing data ( $a_1, a_2, a_3, \dots, a_n$ ) [ $n > c$ ]**

Temporal locality exists. However, LRU fails to capture.

# Bimodal Insertion Policy (BIP) [ISCA '07]

```
if ( rand() <  $\epsilon$  )  $\epsilon=1/16, 1/32, 1/64$   
    Insert at MRU position;  
else  
    Insert at LRU position;
```

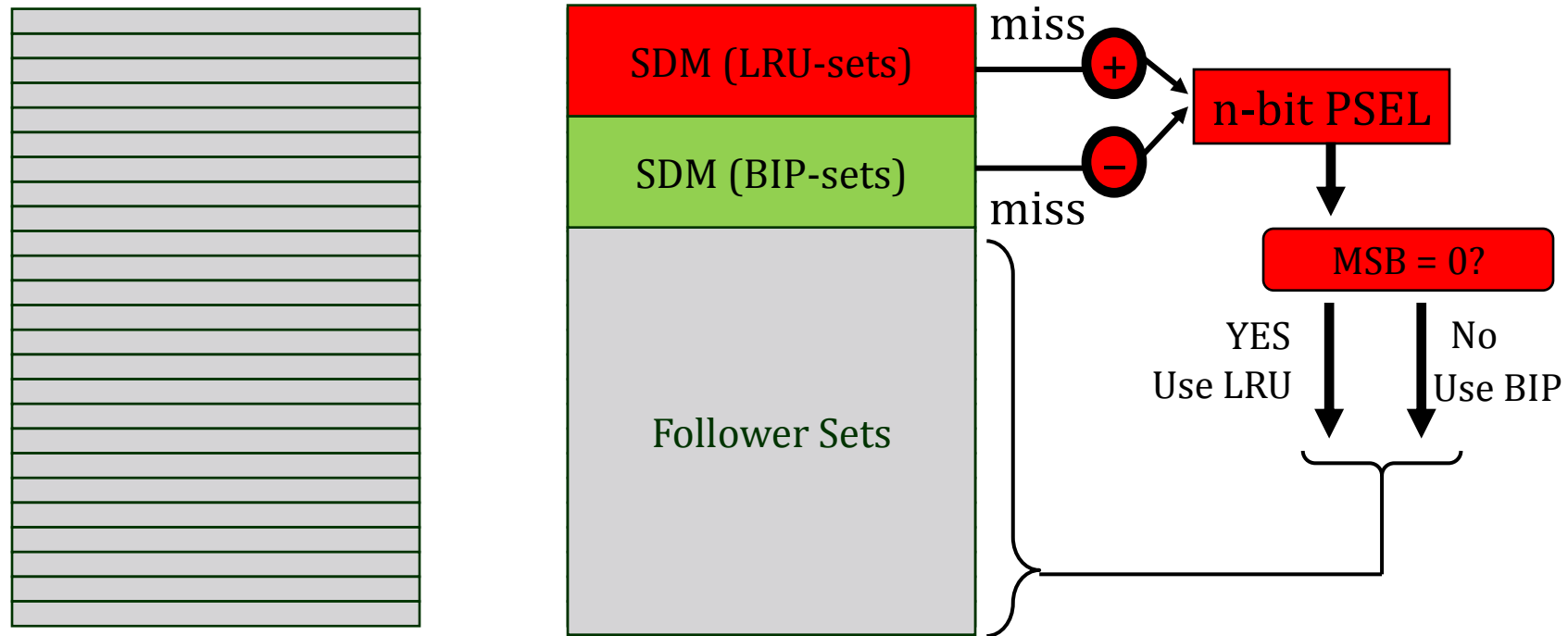
For small  $\epsilon$ : BIP retains thrashing protection of LRU insertion policy.

Infrequently insert lines in MRU position

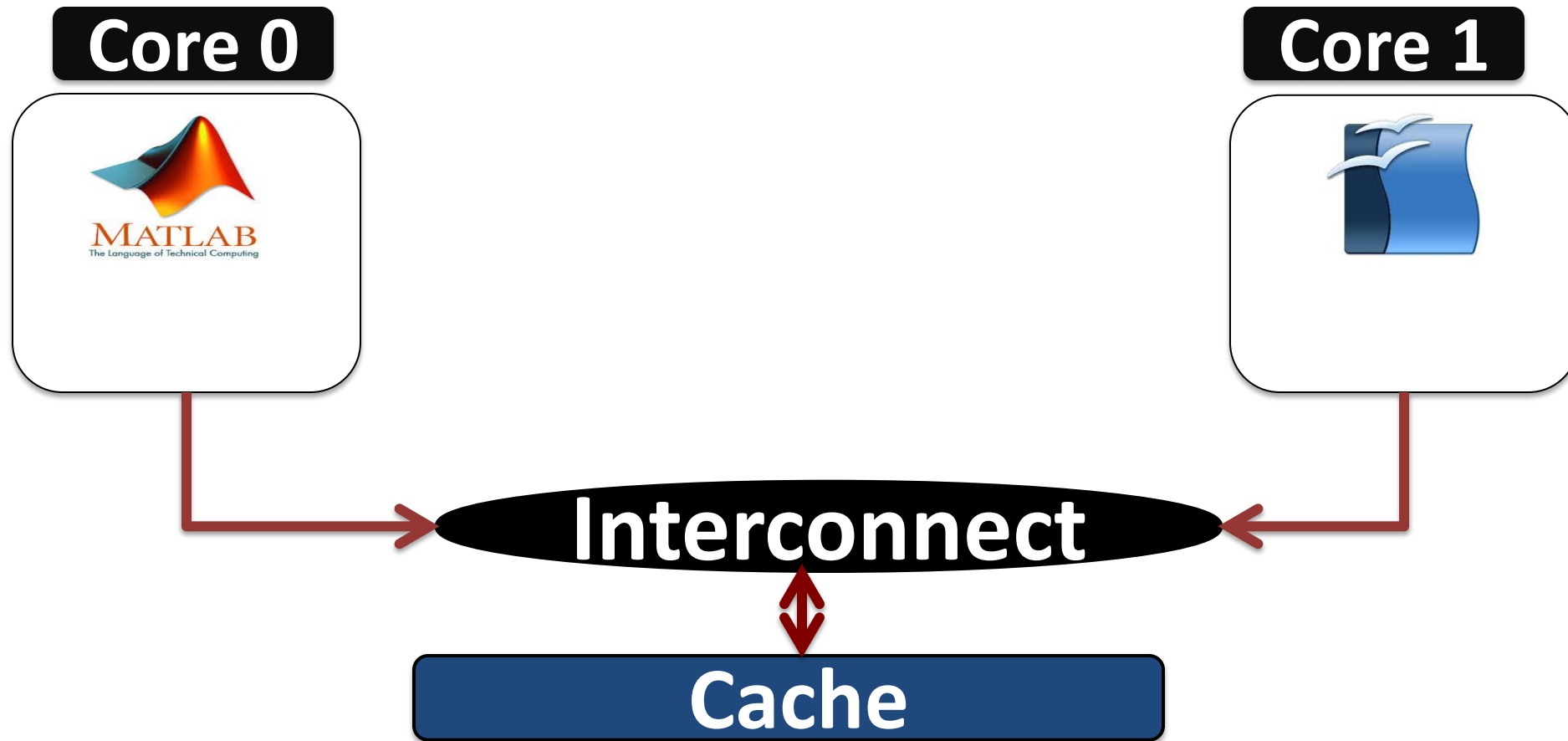
# Dynamic Insertion Policy (DIP) [ISCA '07]

SDM – Set Dueling monitors

PSEL – n-bit saturating counters for deciding a policy



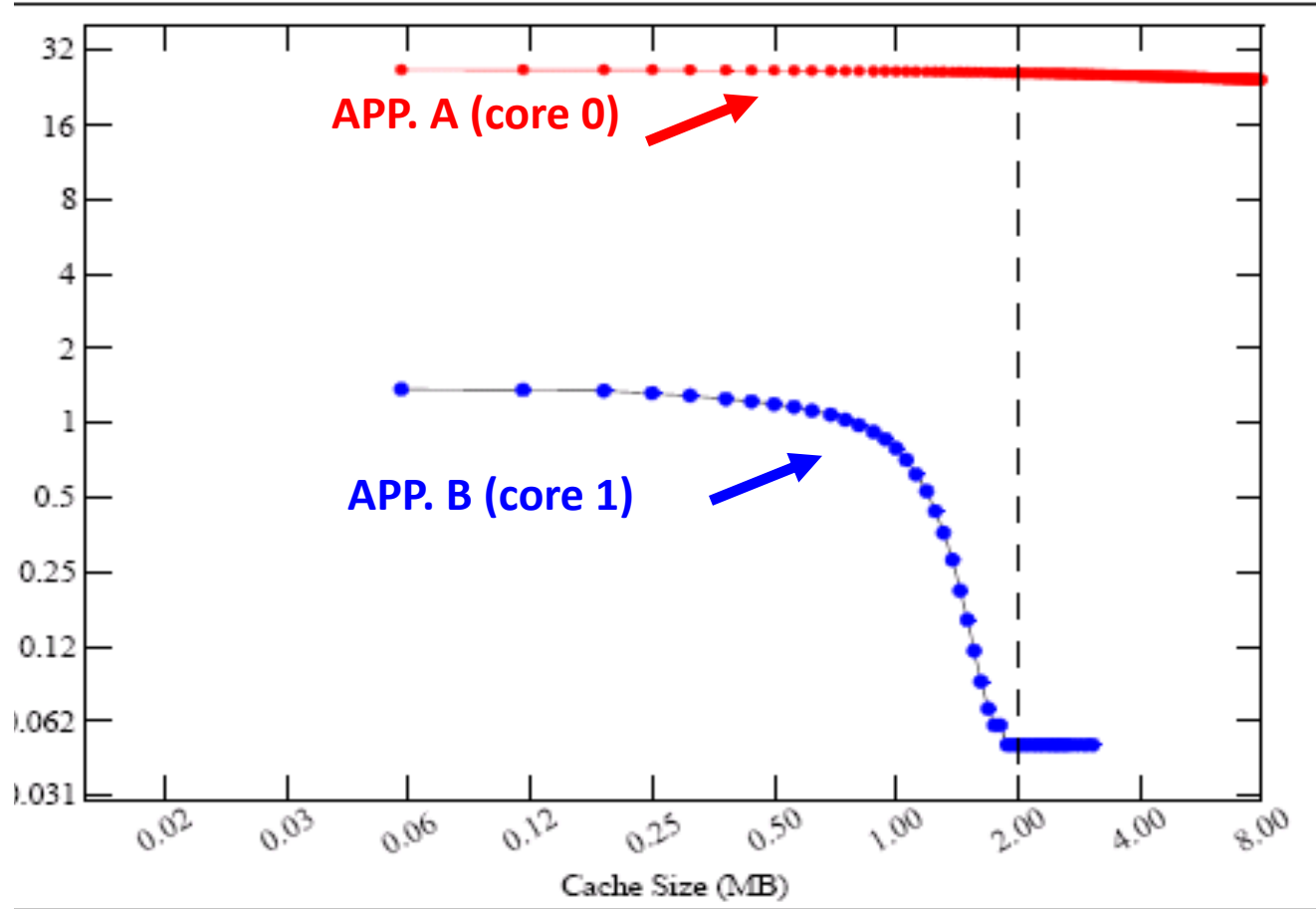
# What about DIP for shared Caches?



What about the learning process for 2-core? N-core? BIP or LRU ?

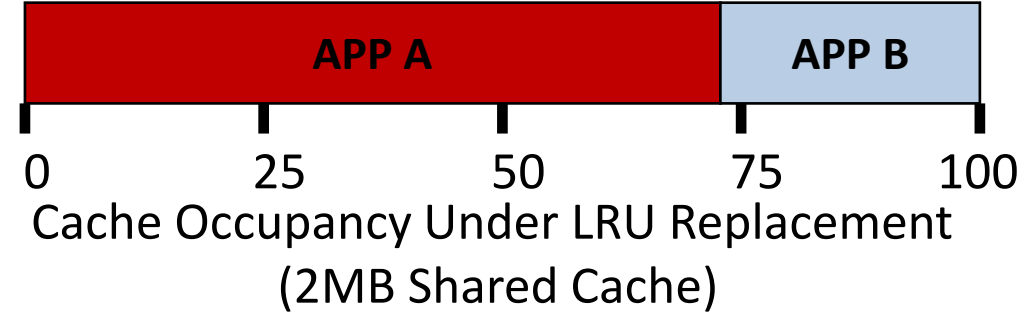
# DIP for Shared Caches [PACT '08]

Misses Per 1000 Instr (under LRU)



Source: TADIP, PACT '08 (Adapted and Modified)

DIP does not distinguish between apps. Learning is not adaptive.

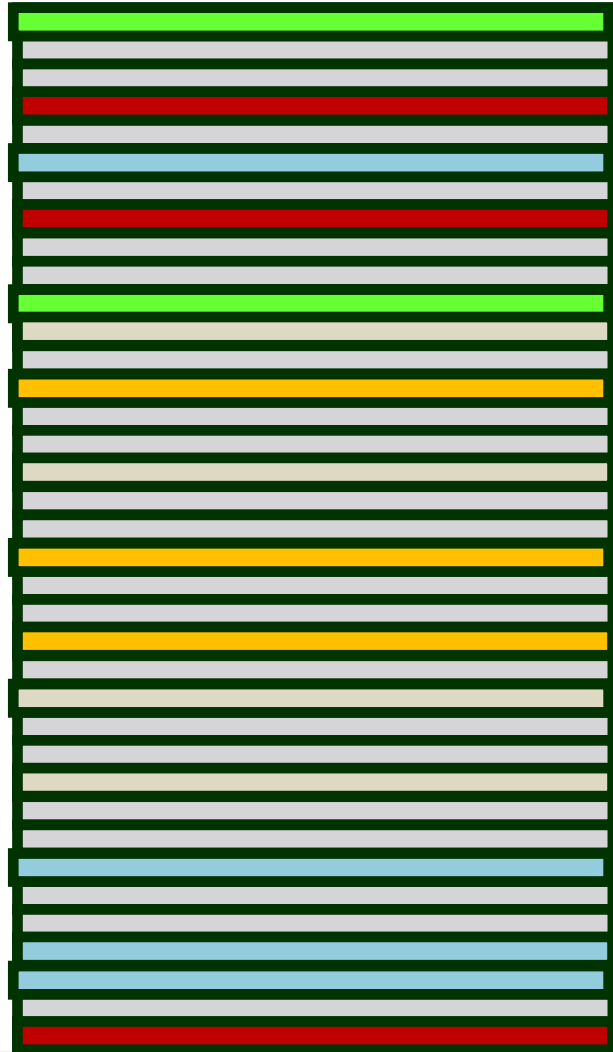


What Should be done?

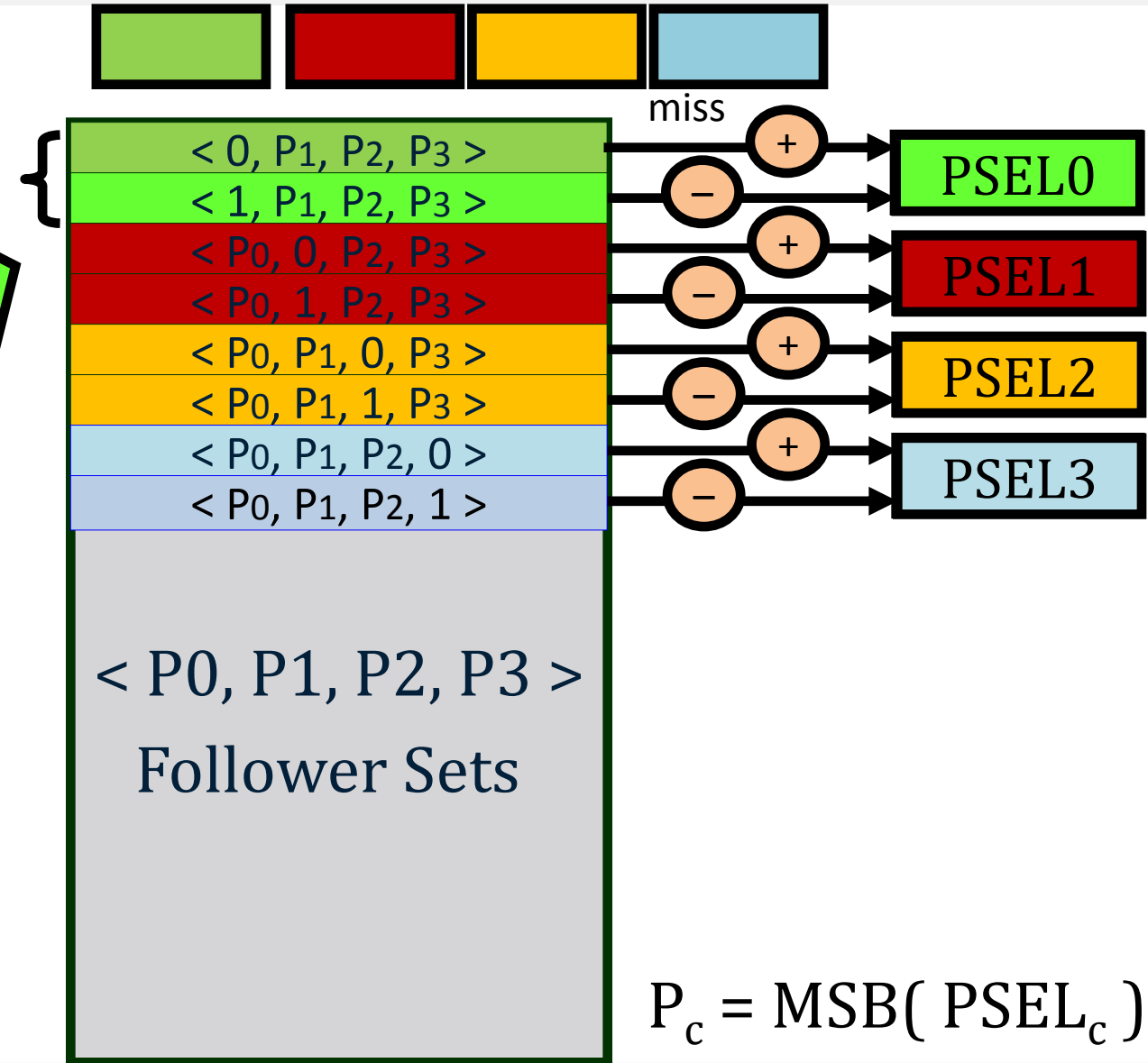
APP A- LRU/BIP? APP B – LRU/BIP?

What about an N-core system?

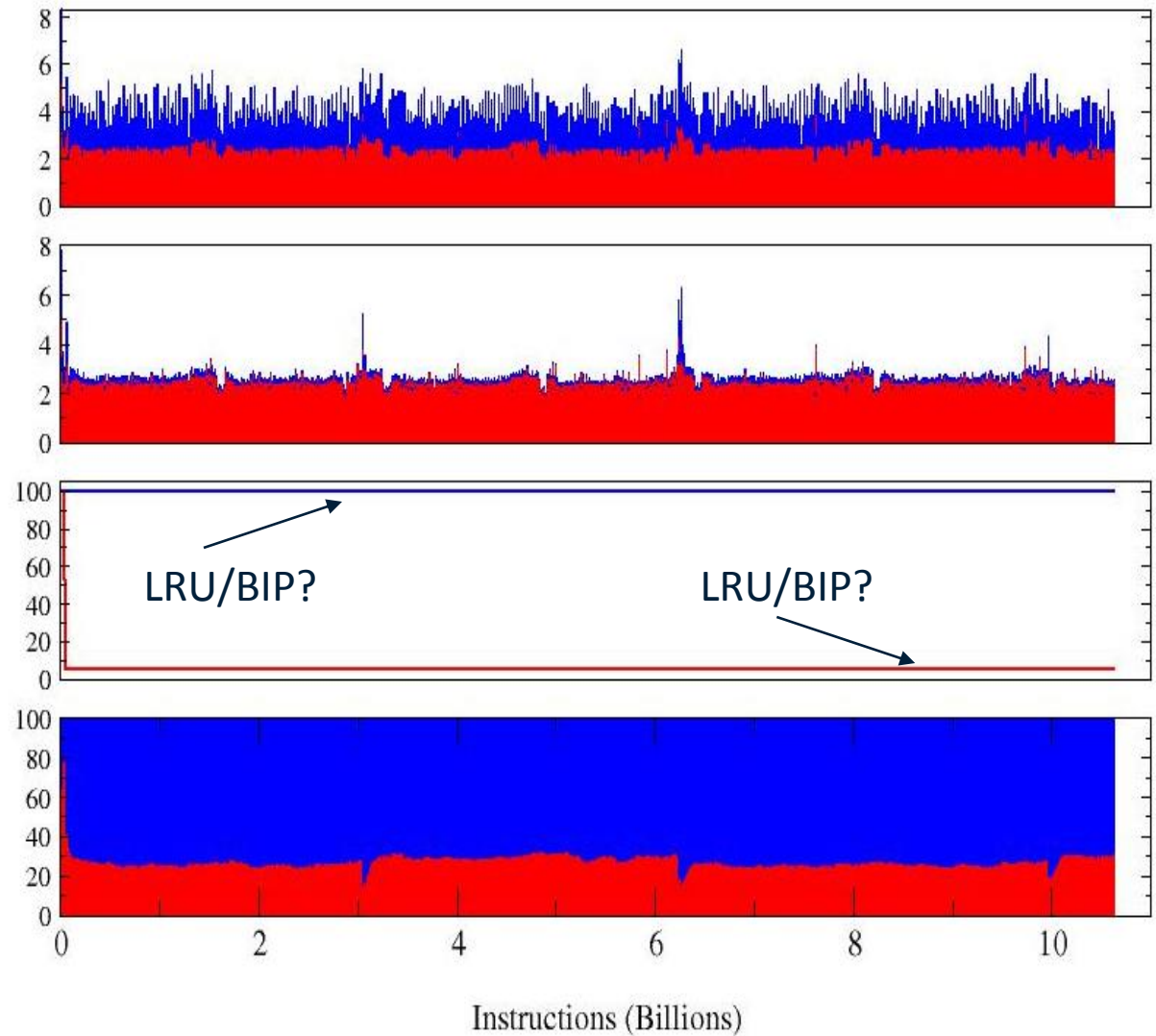
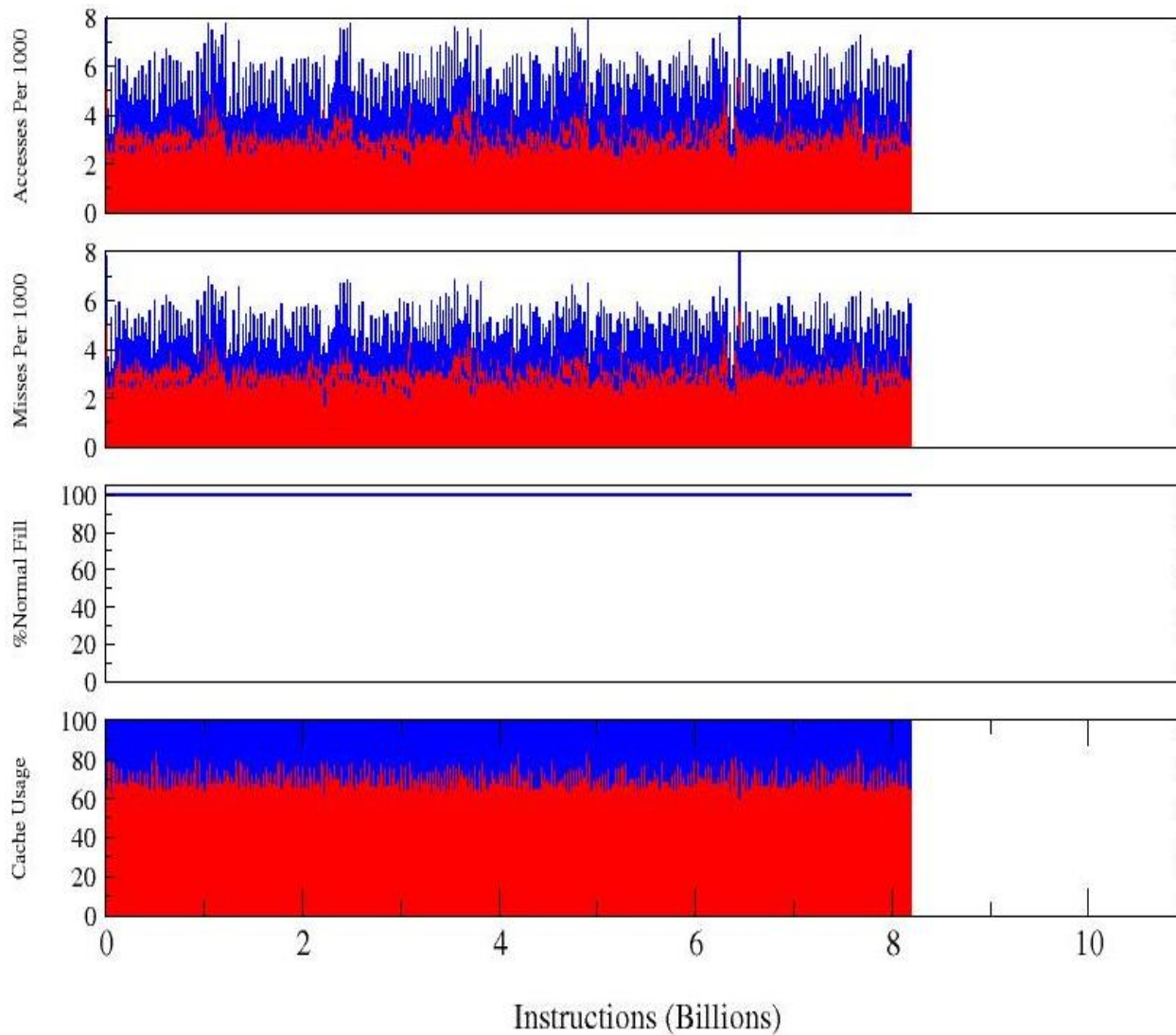
# Thread-Aware DIP (TA-DIP) [PACT '08]



In the presence of other apps, does APP0 doing LRU or BIP improve cache performance?





# DIP vs TA-DIP





# Still Miles to Go

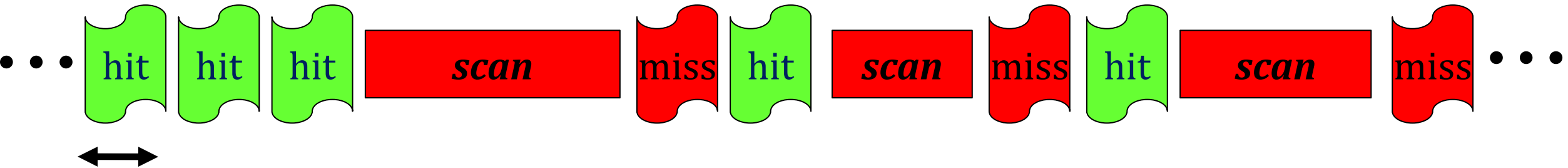
$LLC_{size}$    
 $W_{size}$  

Working set larger than the cache causes thrashing



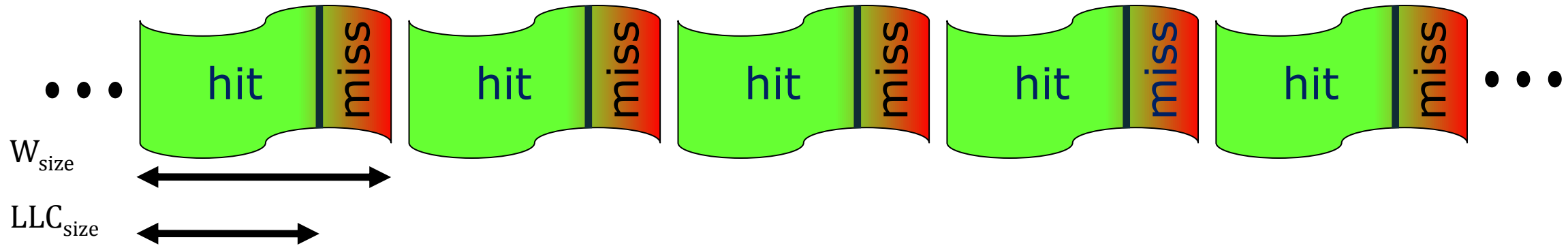
---

References to non-temporal data (*scans*) discards frequently referenced working set

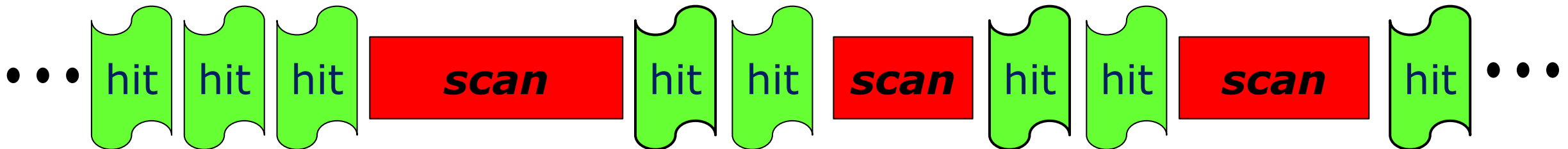


# Still Miles to Go

Working set larger than the cache →  
Preserve some of working set in the cache



Recurring *scans* (bursts of non-temporal data) → Preserve frequently referenced working set in the cache




# Still Miles to Go



Source: Software Technology Forum

Replaces block that **will** be re-referenced furthest in future

victim block 

Physical Way # →  
Cache Tag →

0	1	2	3	4	5	6	7
a	c	b	h	f	d	g	e
4	13	11	5	3	6	9	1

“Time” when block  
will be referenced  
next

# What About NRU?

Is it better than  
LRU?

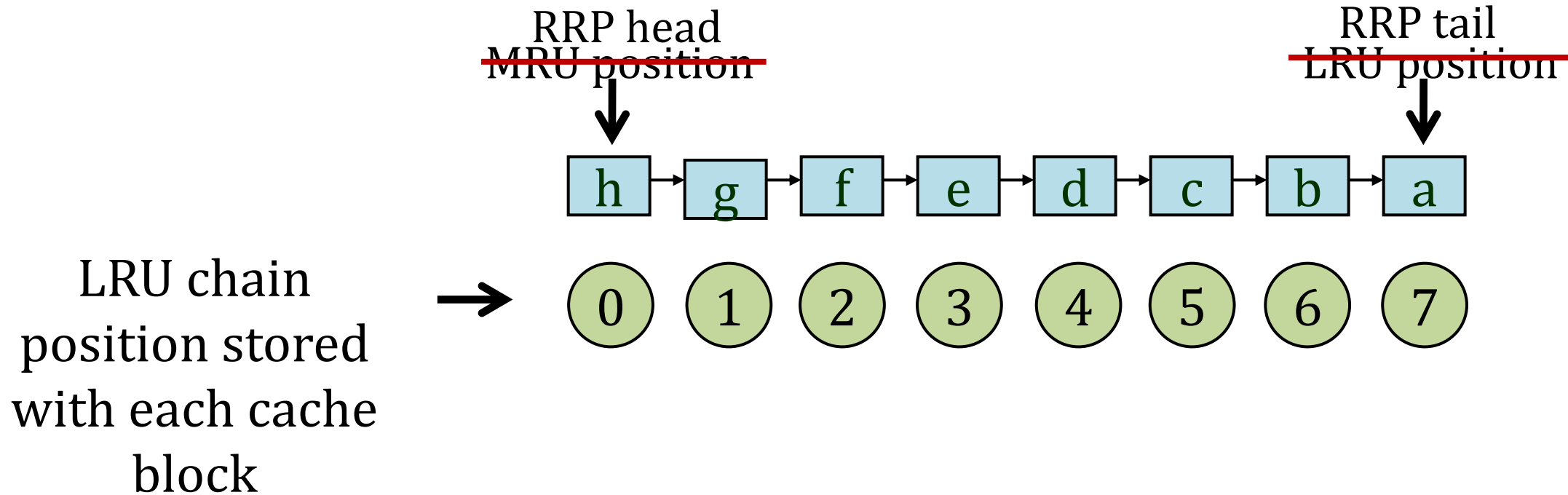
Inserted with 0

Promoted with 0

Eviction – block  
with value 1

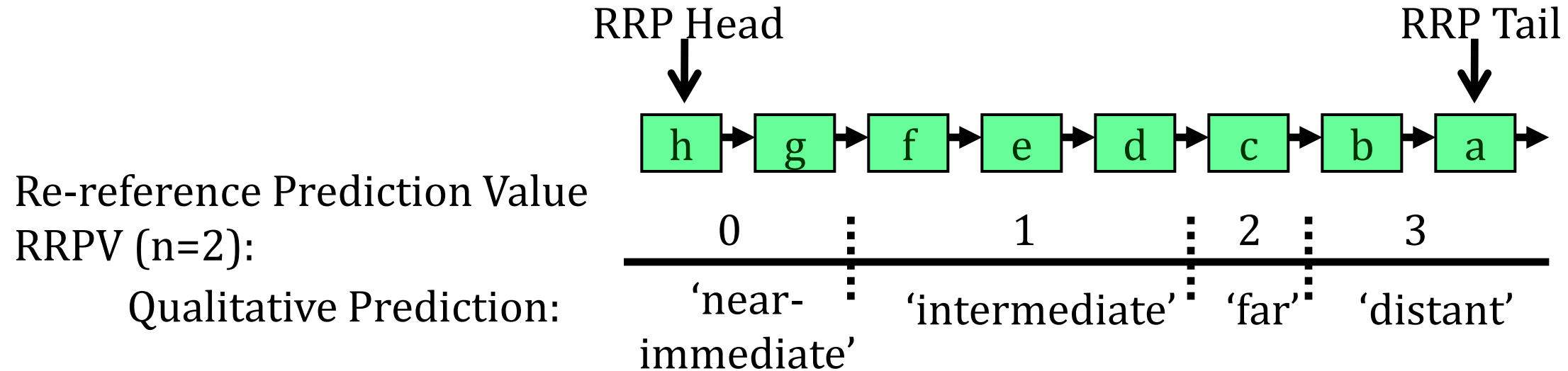
Randomization provides some  
scan and thrash resistance

# NRU to RRIP [ISCA '10]



RRP: Re-reference prediction

# RRIP

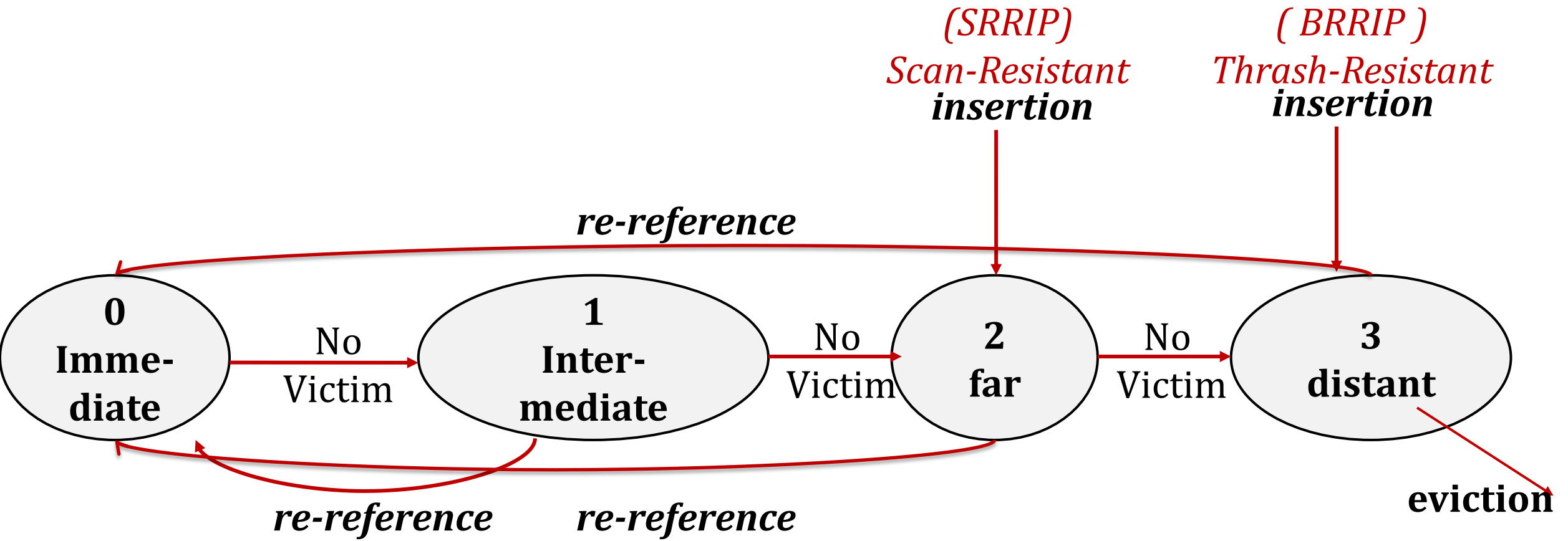


Intuition: New cache block will not be re-referenced soon.  
Replaces block with distant RRPV.

Insert with RRPV=2, Evict with RRPV=3  
promote blocks with RRPV=0.

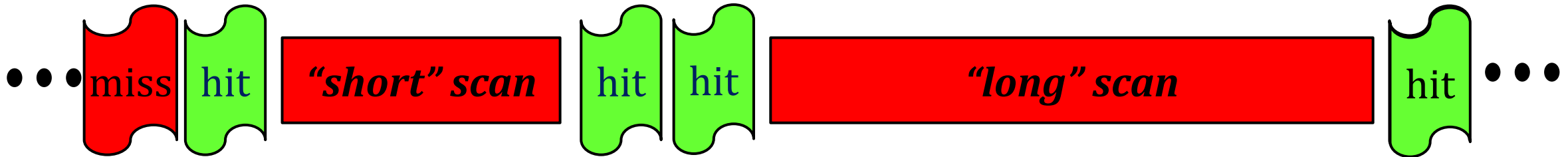
Static RRIP (Single core) and Thread-Aware Dynamic RRIP  
(SRRIP+BRRIP, multi-core, based on SDMs).

# RRIP

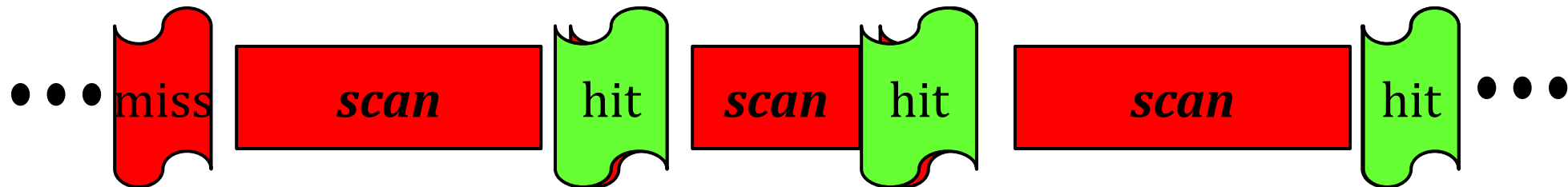


# SRRIP – Not Good Enough

LONG scans in access pattern



Active working-set **MUST** be RE-REFERENCED at least ONCE between scans





# Mixed Access Patterns

**(a1, a2), (a1, a2), b1, b2, b3, (a1, a2)**

**Short Scan**

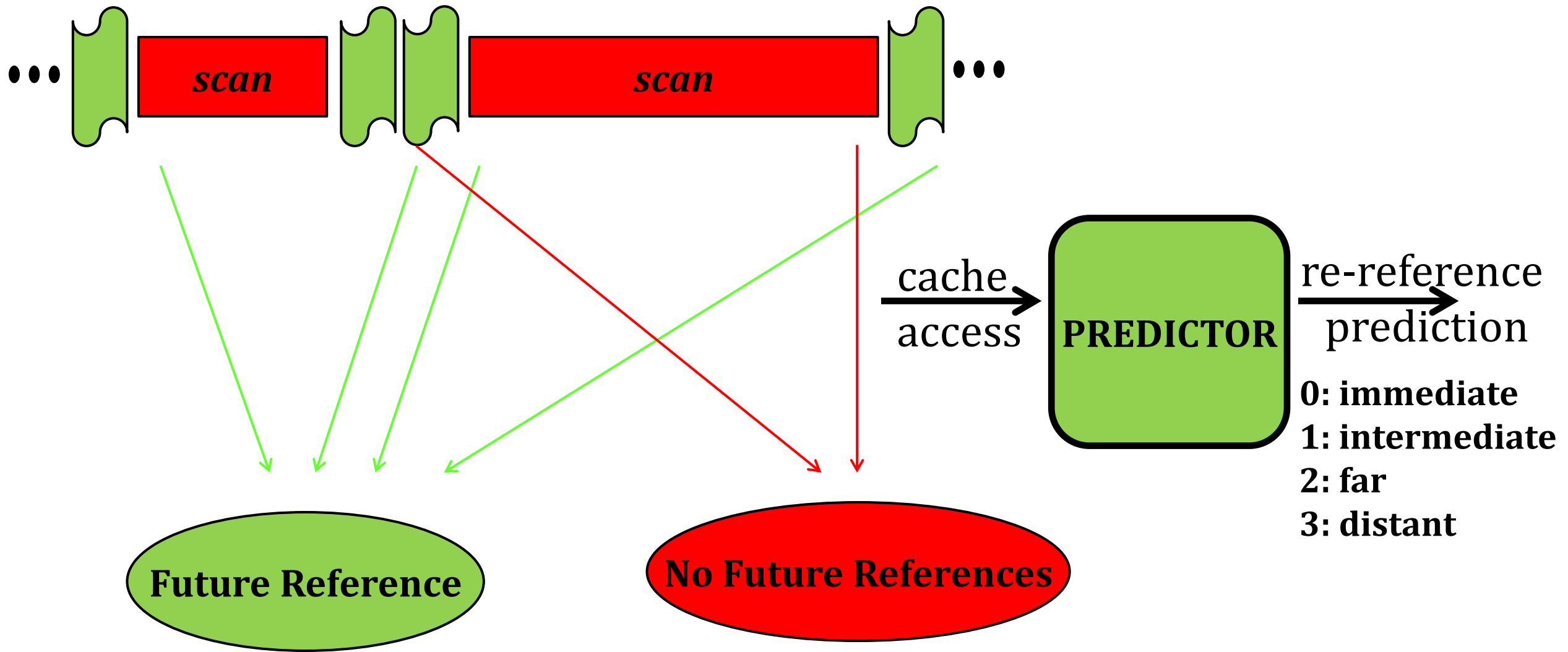
**(a1, a2), (a1, a2), b1, b2, b3, b4, b5, b6, b7,.. (a1, a2)**

**Long Scan**

**(a1, a2), b1, b2, b3, b4, (a1, a2)**

**One Reuse**

# SHIP [MICRO '11]



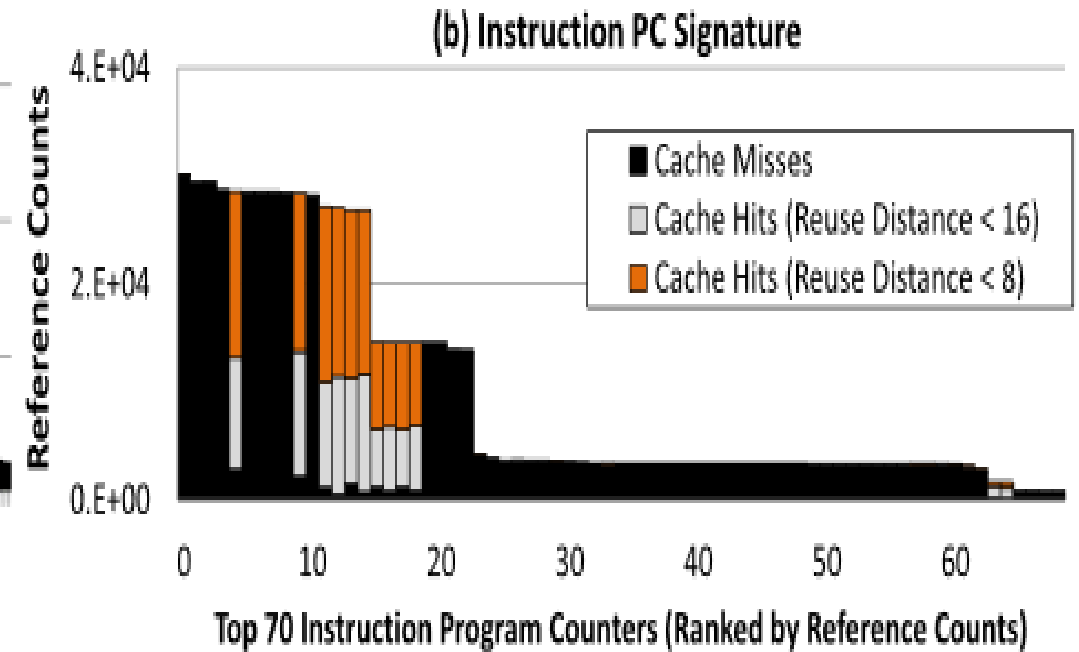
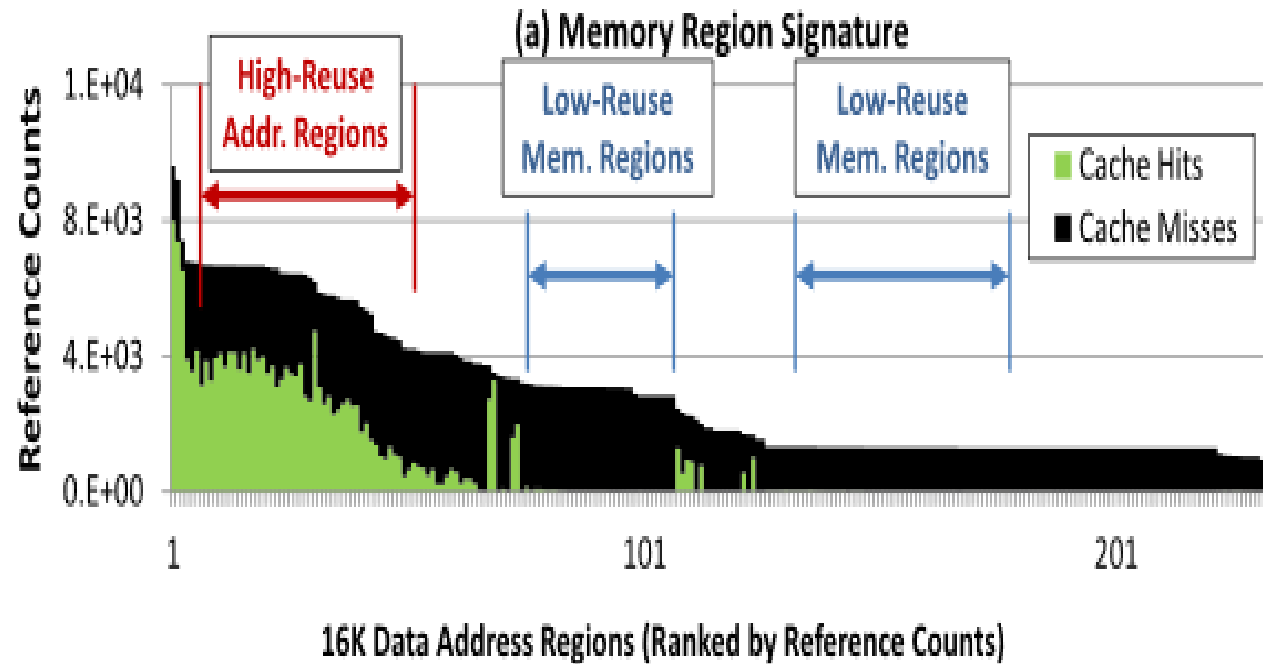
# Signatures -> Re-reference [SHiP]

Memory Region OR  
Memory Instruction Program Counter (PC)

LLC accesses by the same “signature” tend to have similar re-reference patterns

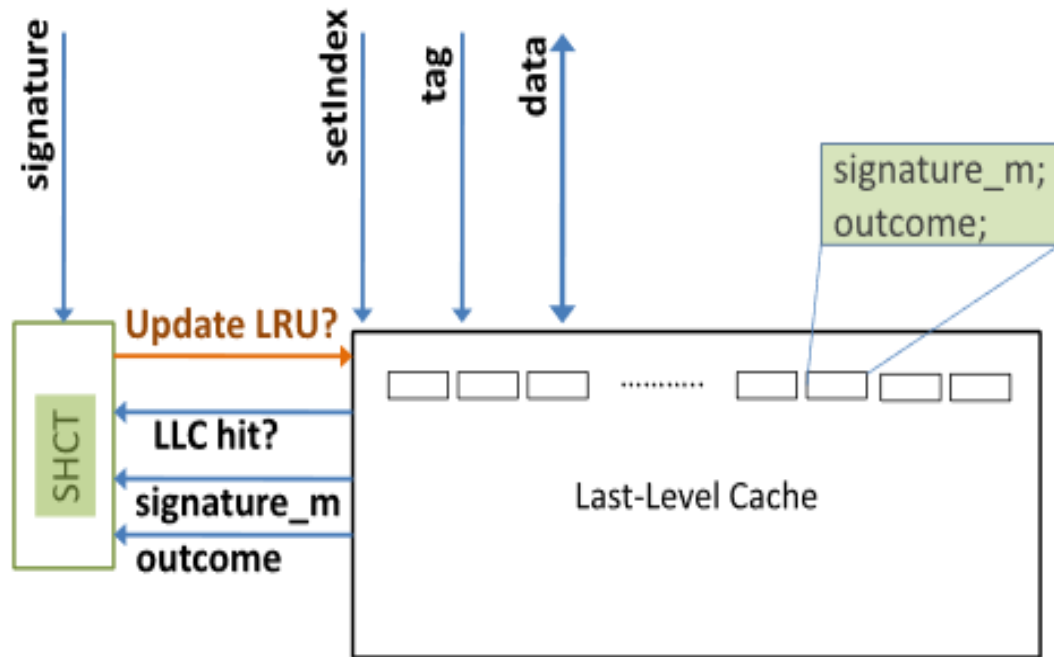
LLC accesses by the same “signature” tend to have similar re-reference patterns

# Examples



# SHiP

(a) SHiP Structure



(b) SHiP Algorithm

```
if hit then
    cache_line.outcome = true;
    Increment SHCT[signature_m];
else
    if evicted_cache_line.outcome != true
        Decrement SHCT[signature_m];
    cache_line.outcome = false;
    cache_line.signature_m = signature;
    if SHCT[signature] == 0
        Predict distant re-reference;
    else
        Predict intermediate re-reference;
end if
```

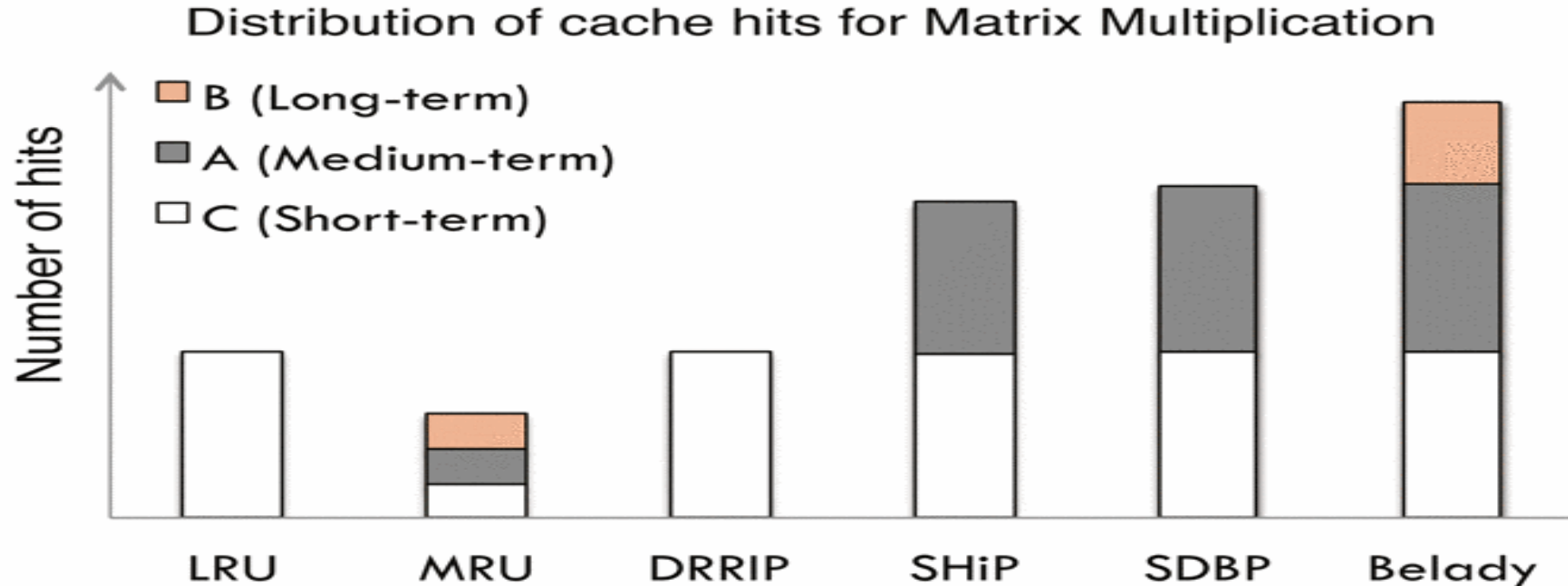
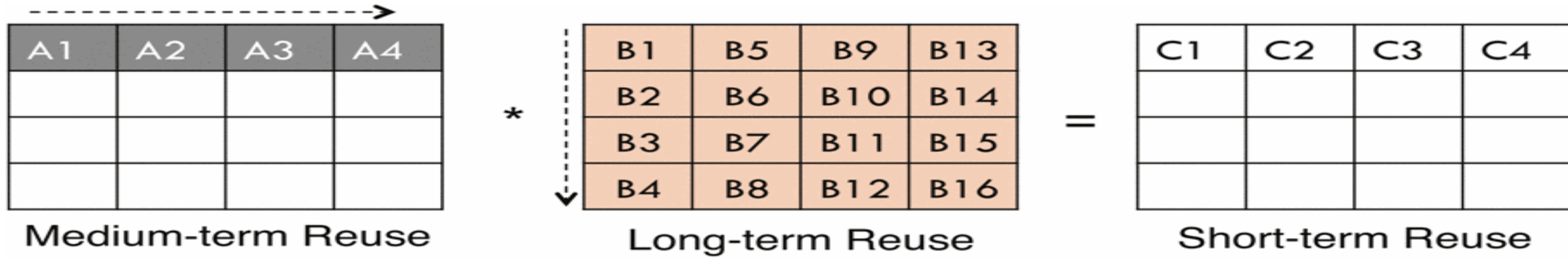
# SHiP to SHiP++ [CRC2 '17]

Improved Cache Insertion: cache block with signature with highest value of counter inserted with RRPV=0.

Training: Only on first re-reference (not on all hits) and evictions

Writebacks: Insert with RRPV=3.

# Hawkeye [ISCA '16]



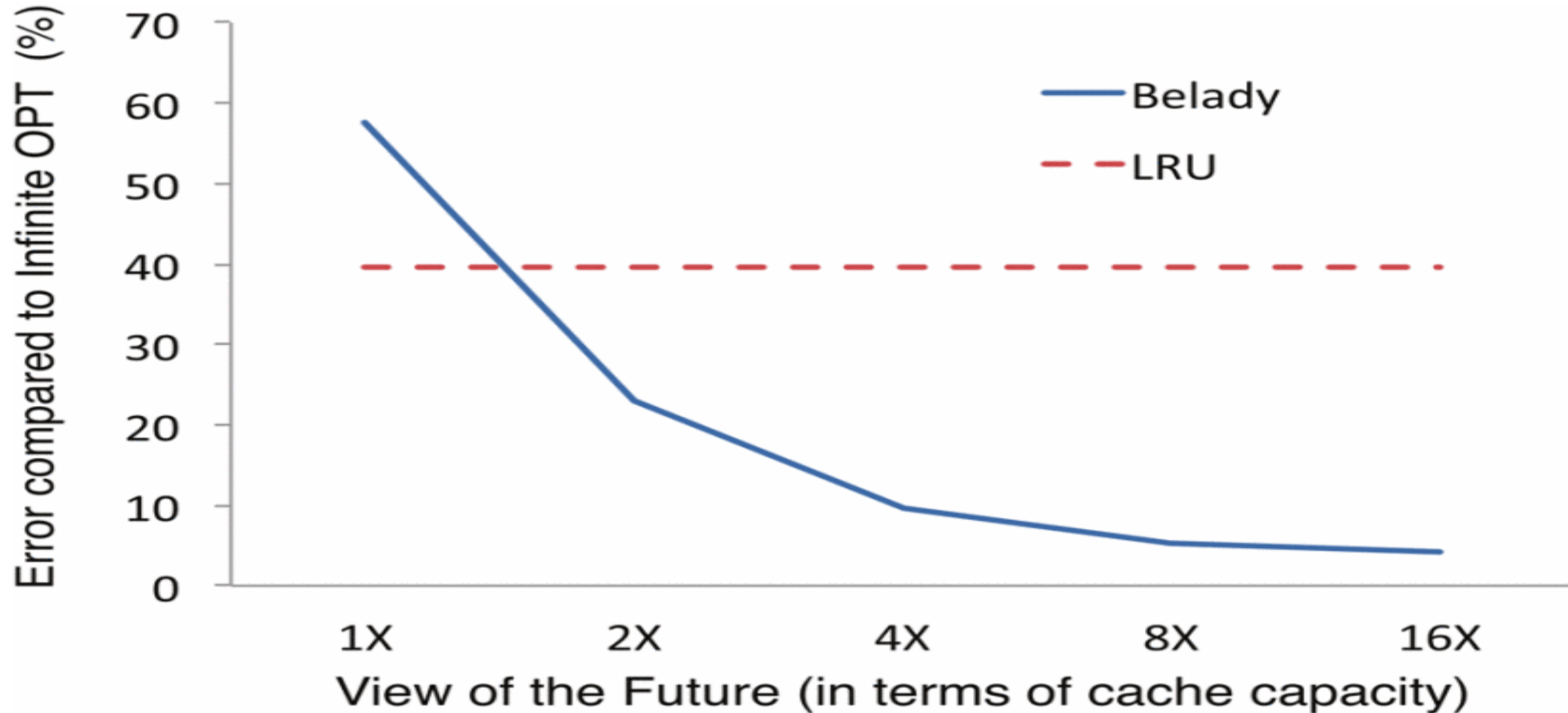


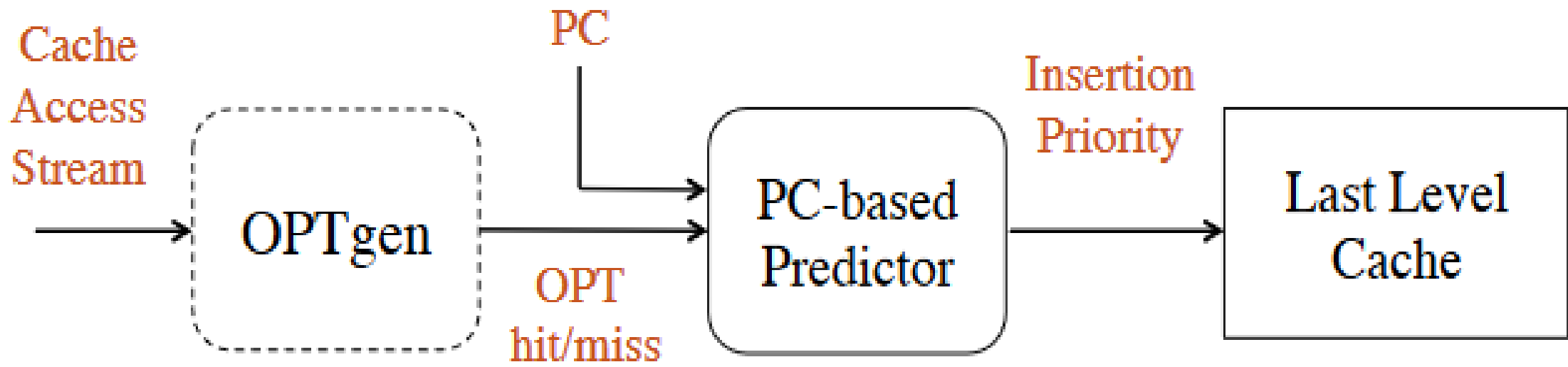
# Hawkeye





# LRU vs Belady

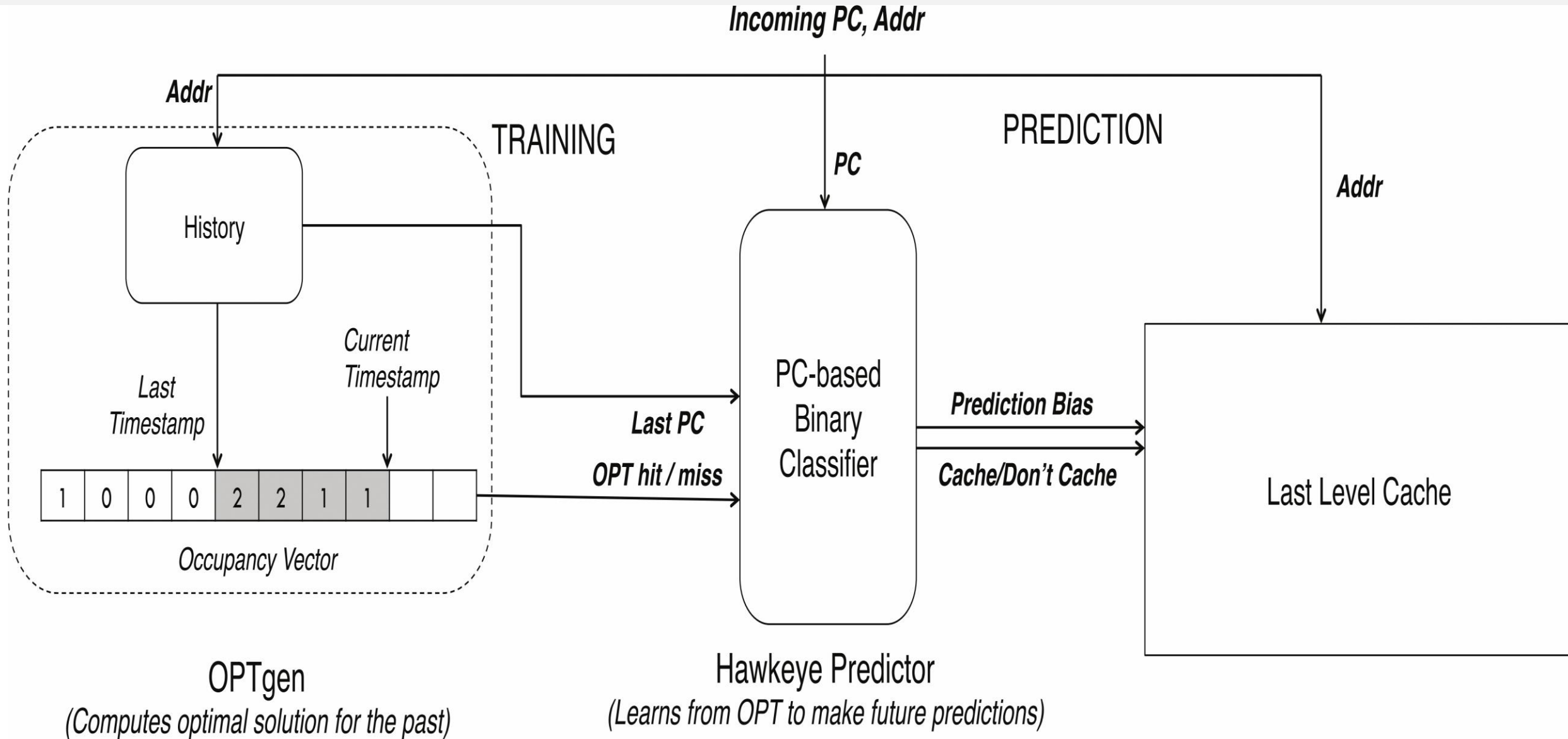




*Computes OPT's  
decisions for the past*

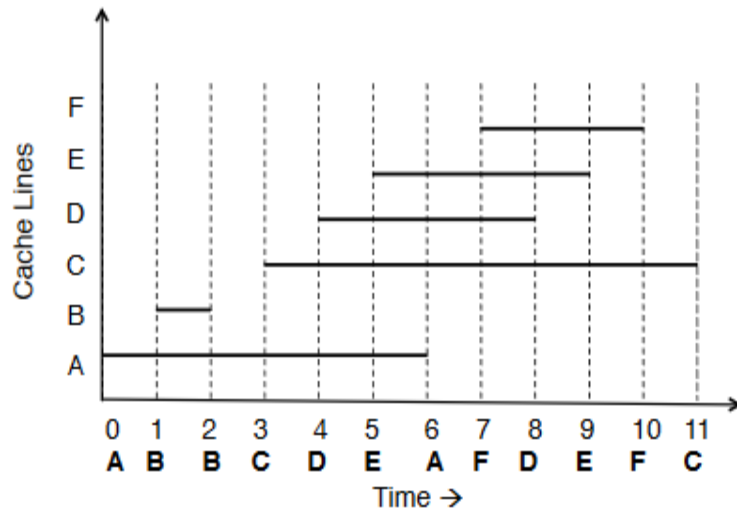
*Remembers past  
OPT decisions*

# Hawkeye in Action

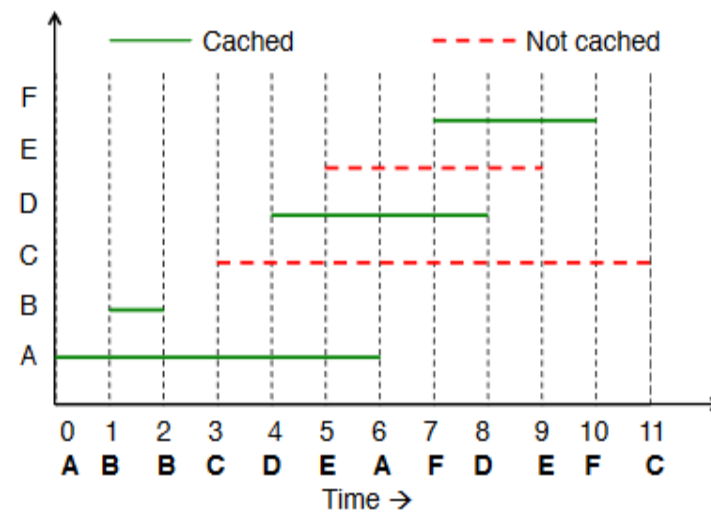


# OPTgen

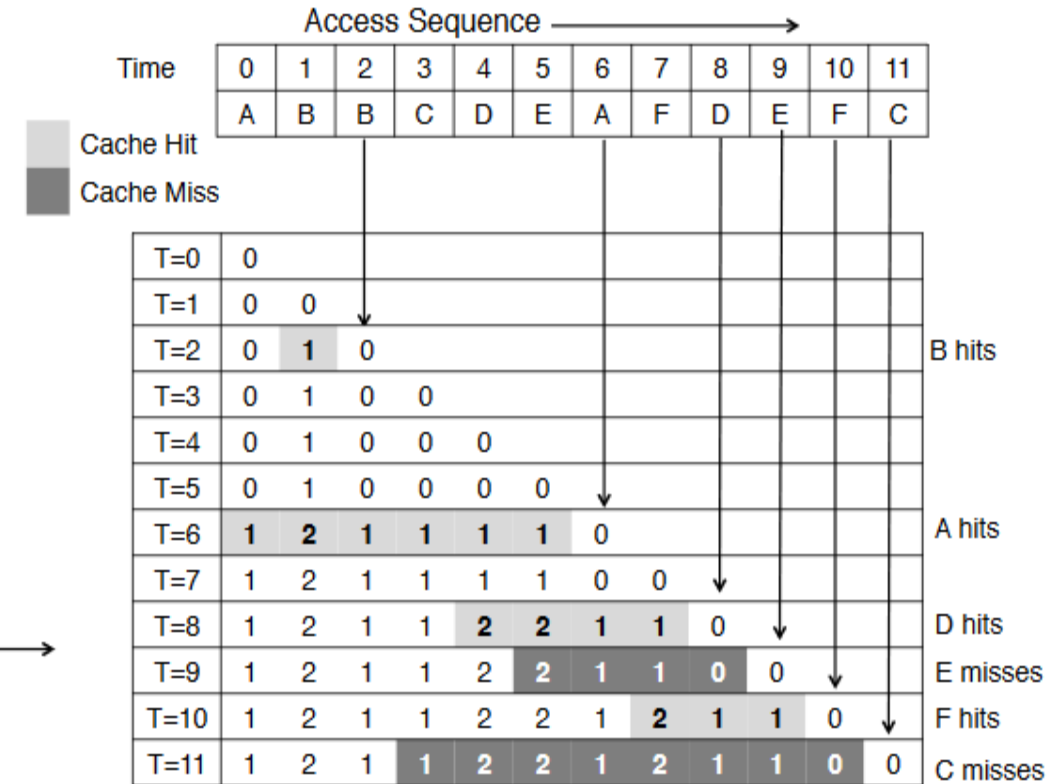
Access Sequence: A, B, B, C, D, E, A, F, D, E, F, C (Cache capacity is 2 lines)



(a) Timeline view of the Access Stream



(b) Optimal Solution (4 hits)  
[Cache hits marked as solid lines]



(c) OPTgen Solution (4hits)  
[State of the Occupancy Vector over time]

# PC Based Classifier

Cache averse vs cache friendly ?

Uses OPTgen to predict the usefulness of PC.

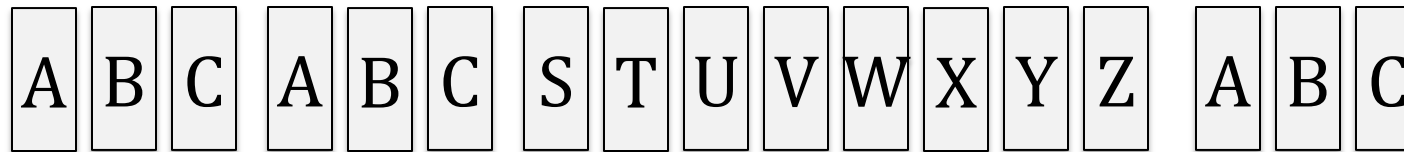
Hawkeye Prediction \ Hit or Miss	Cache Hit	Cache Miss
Cache-averse	RRIP = 7	RRIP = 7
Cache-friendly	RRIP = 0	RRIP = 0; Age all lines: if (RRIP < 6) RRIP++;

Please refer in-class discussion for a detailed Hawkeye

# Let's Break the Problem [EAF PACT'12]

Different blocks have different reuse behavior

Access Sequence:

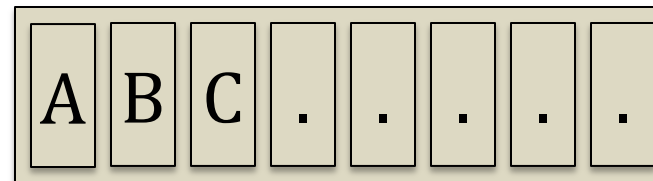


High-reuse block



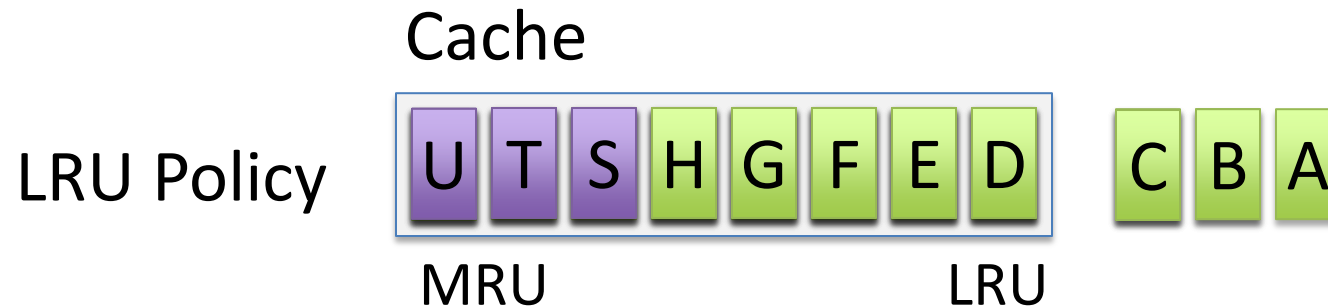
Low-reuse block

Ideal Cache

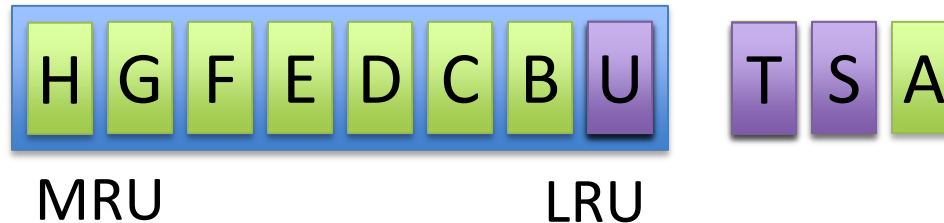


# Cache Pollution

**Problem:** Low-reuse blocks evict high-reuse blocks



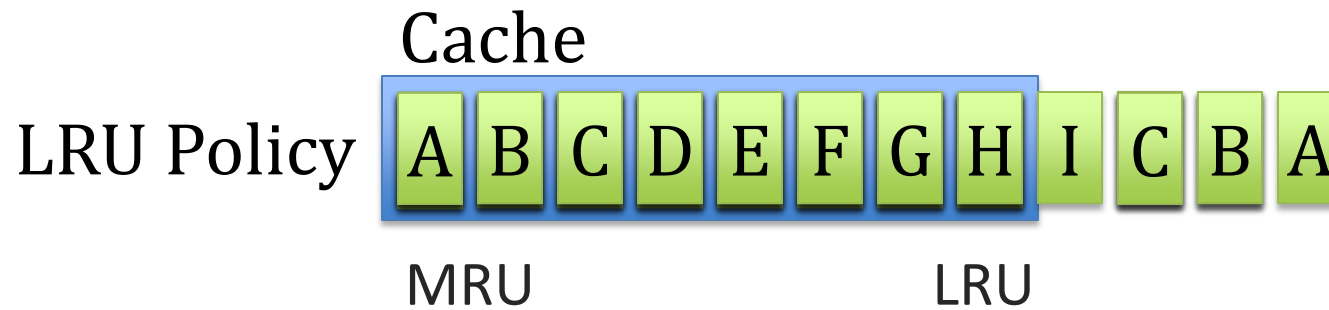
**Prior work:** Predict reuse behavior of missed blocks. Insert low-reuse blocks at LRU position.





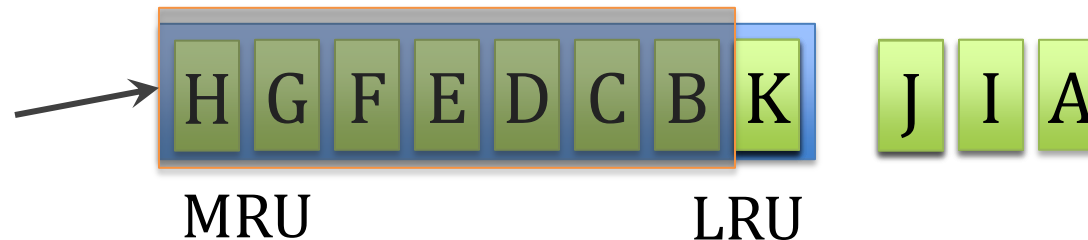
# Cache Thrashing

**Problem:** High-reuse blocks evict each other

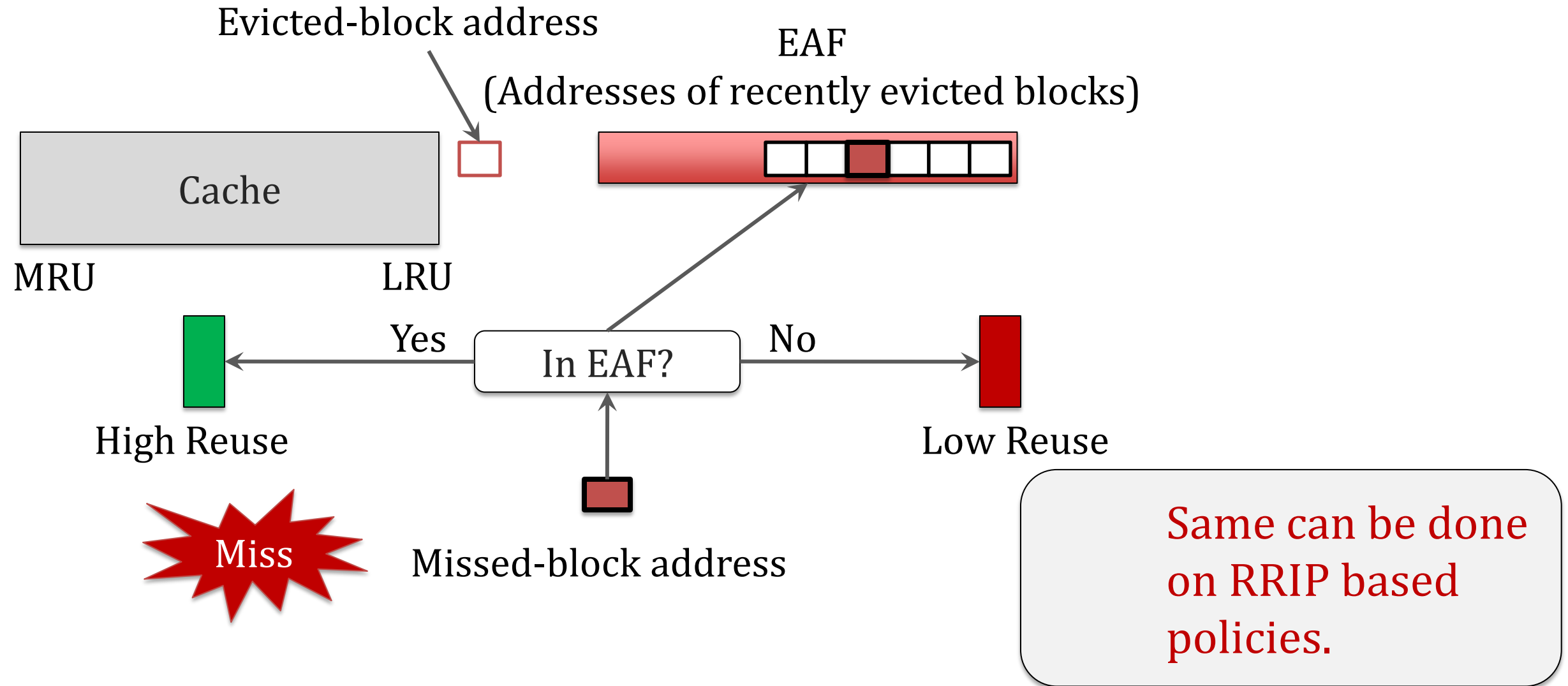


**Prior work:** Insert at MRU position with a very low probability  
(**Bimodal insertion policy**)

A fraction of  
working set  
stays in cache

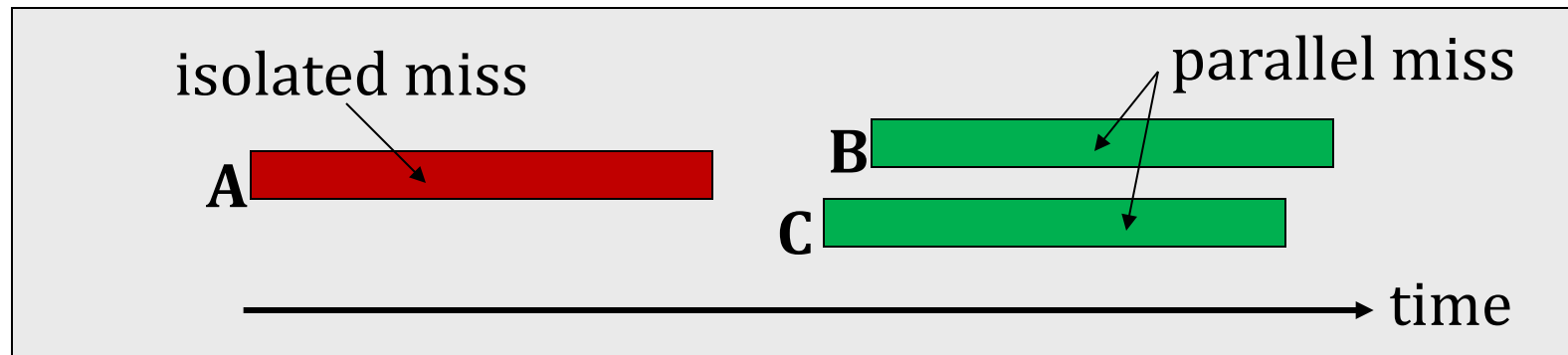


# Evicted Address Filter



# What about MLP? [ISCA '06]

## MLP: Memory Level Parallelism



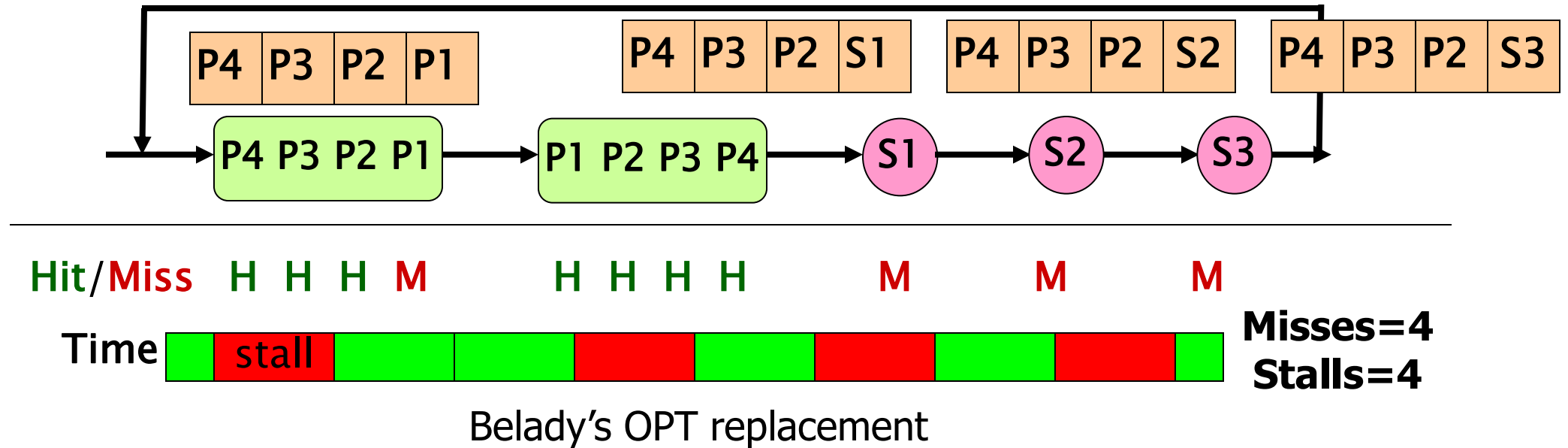
Conventional Wisdom:

Reduction in miss-count results in reduction of miss-penalty

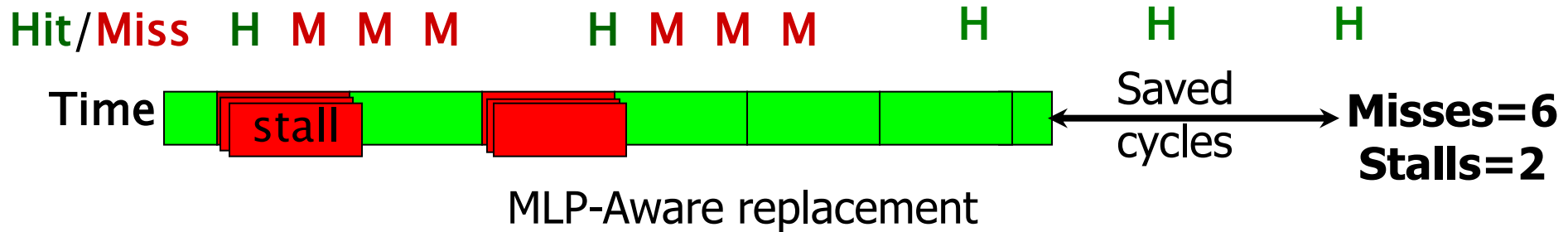
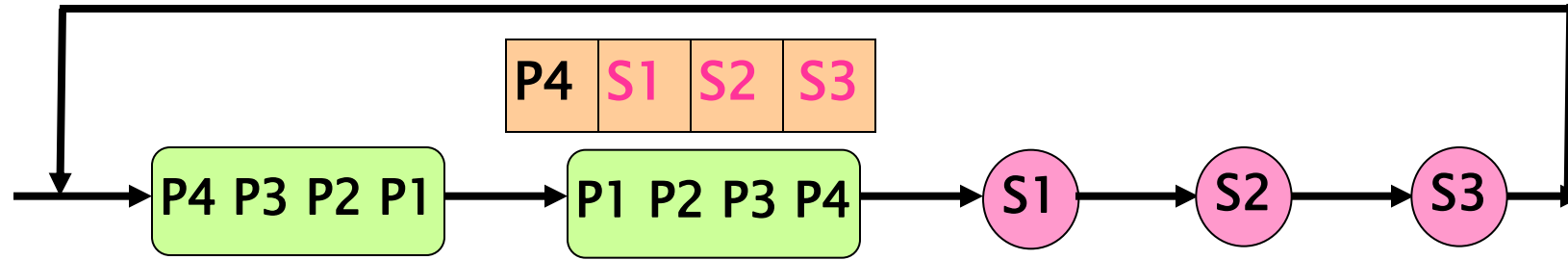
Fewest Misses != Best Performance

# Belady

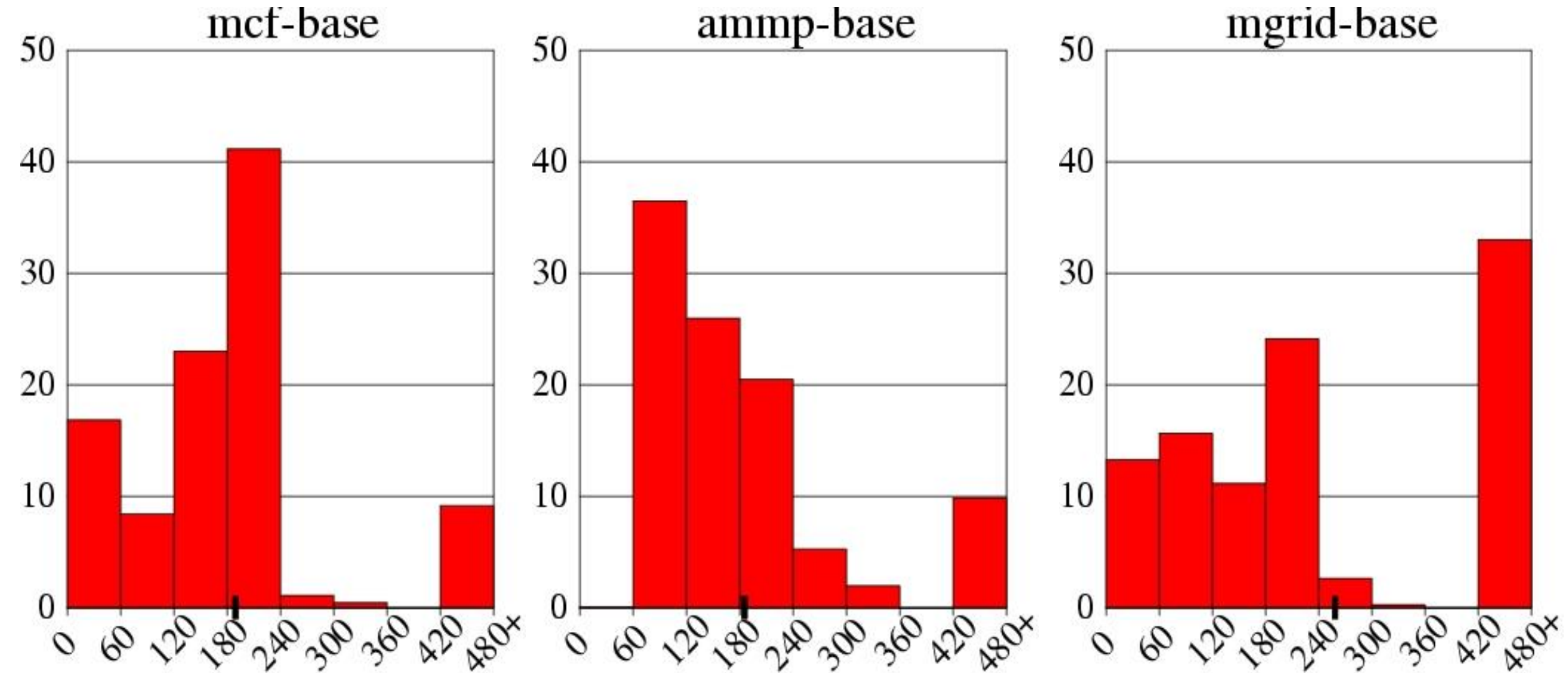
Cache



# MLP Awareness



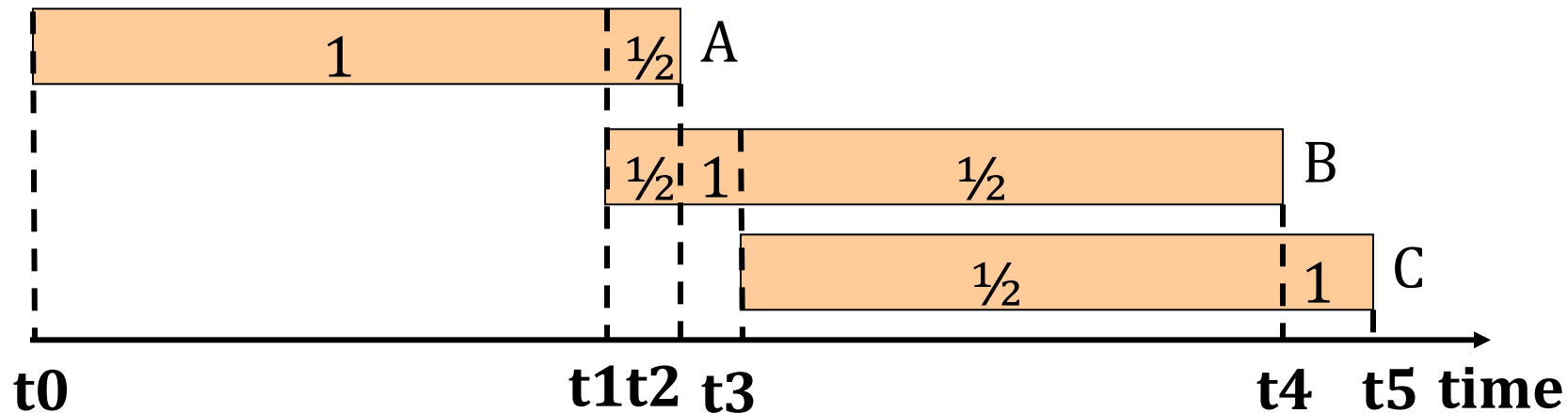
# Cost of LLC Misses (Few Examples)



# Quantifying MLP

Cost of miss is number of cycles the miss stalls the processor

Divide each stall cycle equally among all parallel misses



# How to do it at the Cache

Miss Status Holding Register (MSHR) tracks all in flight misses

Add a field **mlp-cost** to each MSHR entry

Every cycle for each demand entry in MSHR, **mlp-cost += (1/N)**

$N$  = Number of demand misses in MSHR



# Finally

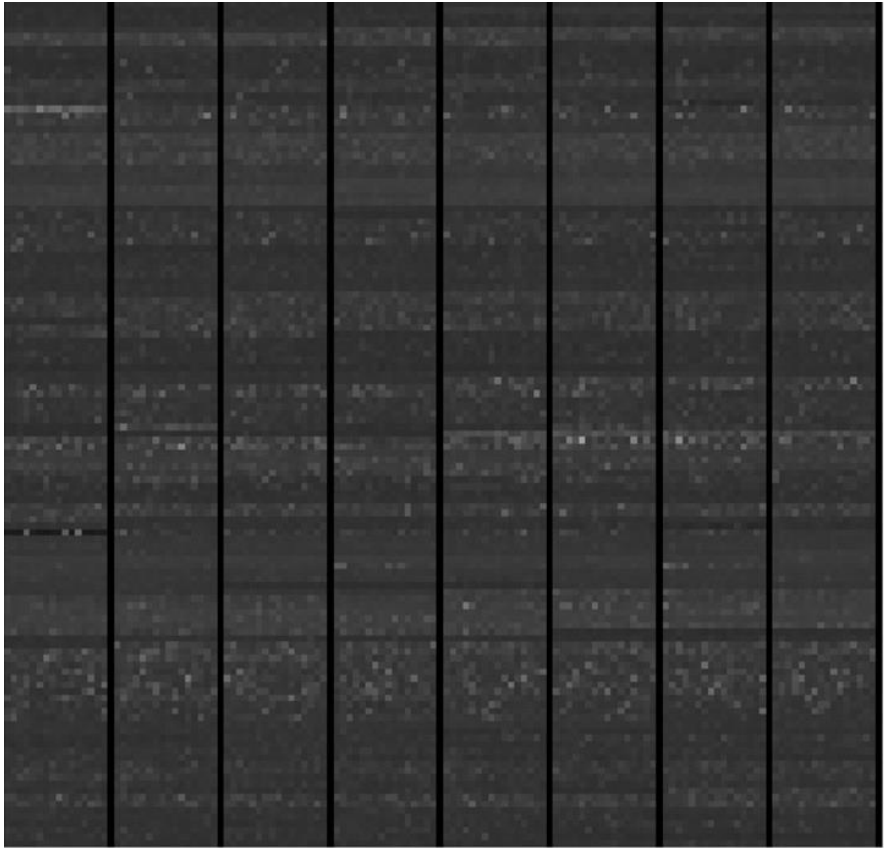
A Linear (LIN) function that considers recency and cost

$$\text{Victim-LIN} = \min \{ \text{Recency (i)} + S * \text{cost (i)} \}$$

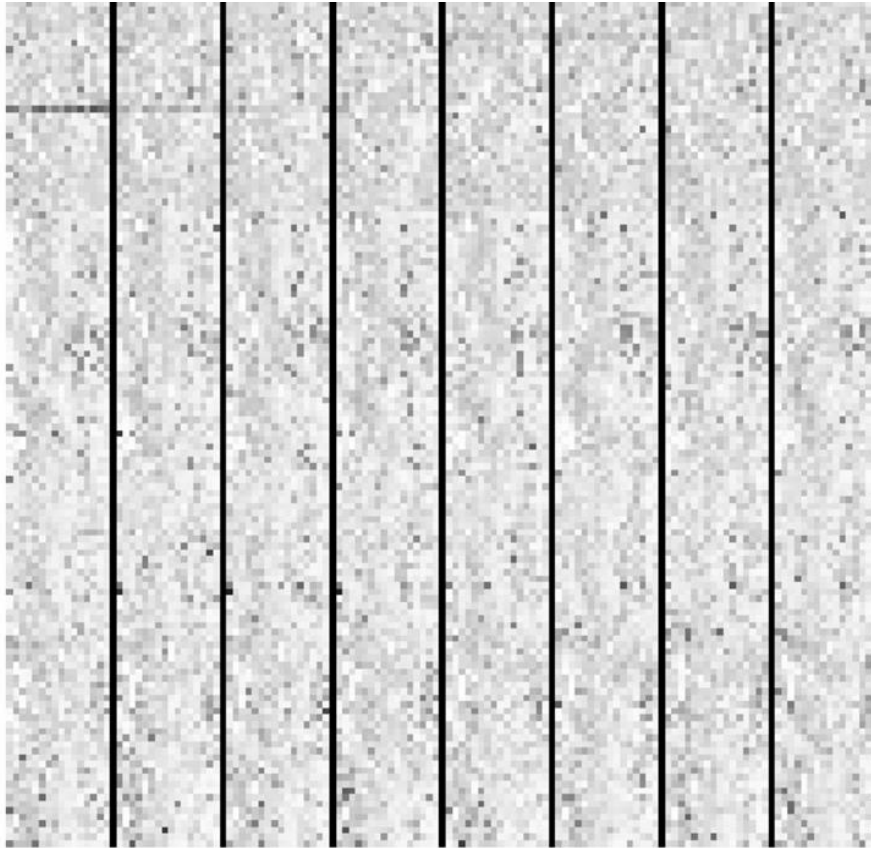
$S$  = significance of cost

$\text{Recency (i)}$  = position in LRU stack  $\text{cost(i)}$












# Predicting Dead Blocks [MICRO '10]



(a)



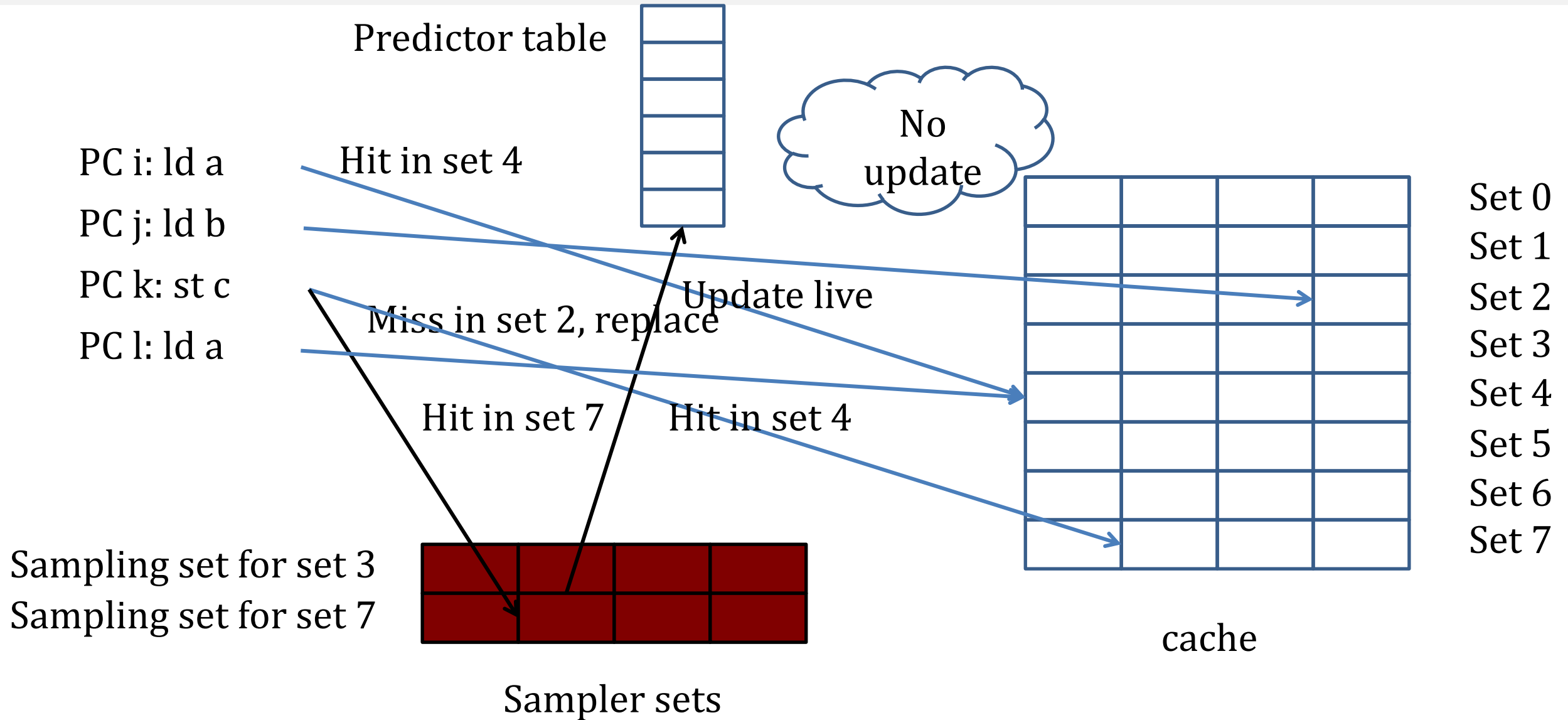
(b)

Greyscale		Efficiency
		0.0
		0.1
		0.2
		0.3
		0.4
		0.5
		0.6
		0.7
		0.8
		0.9
		1.0

**Goal: To get (b) and (a).**

Each pixel = average live time of a cache block

# Dead-block Sampler



# Next Lecture – Cache Hierarchies

