CS698Y: Modern Memory Systems Lecture-6 (Caches)

Biswabandan Panda

biswap@cse.iitk.ac.in

https://www.cse.iitk.ac.in/users/biswap/CS698Y.html
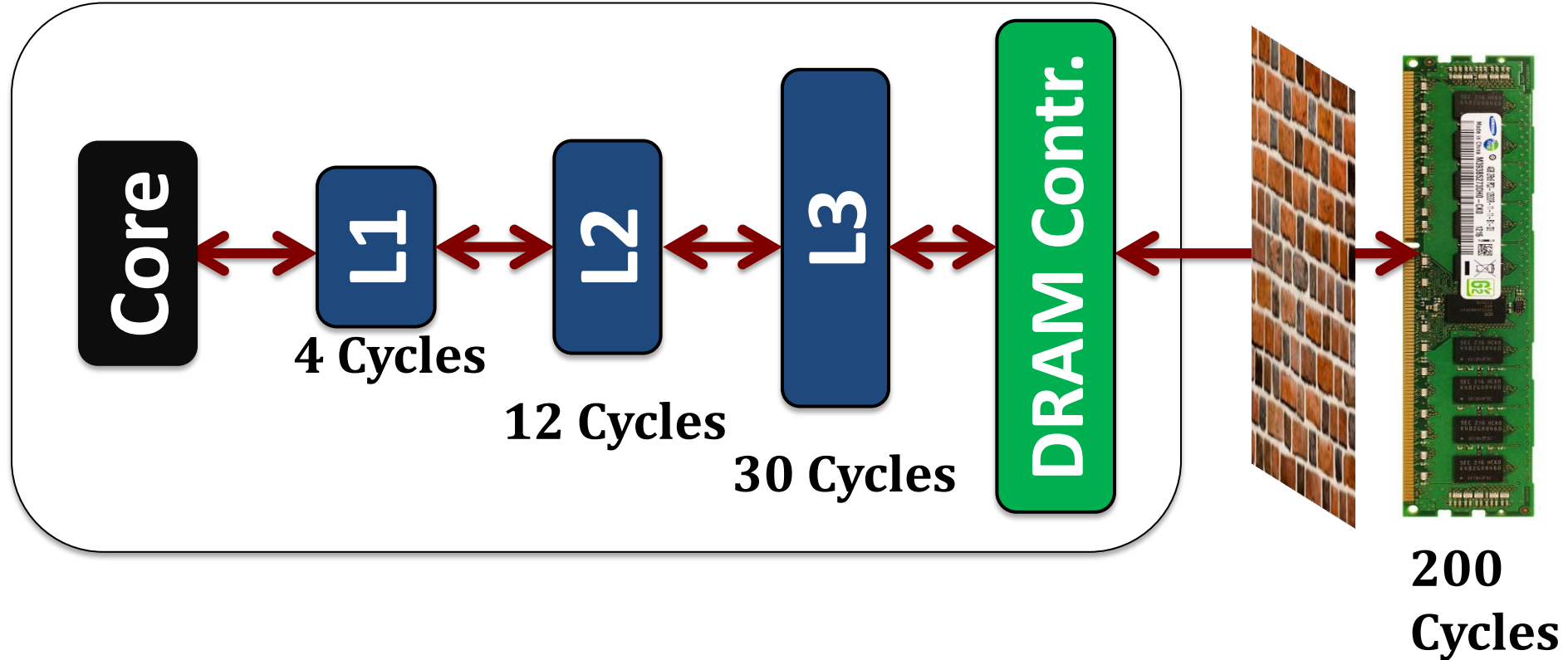
Flow of the Module

Cache Management Policies

Cache Hierarchies

Hardware Prefetching

Cache Compression

Non-uniform Caches

Caches in Single-core System

Caches in Multi-core

> 200 Cycles
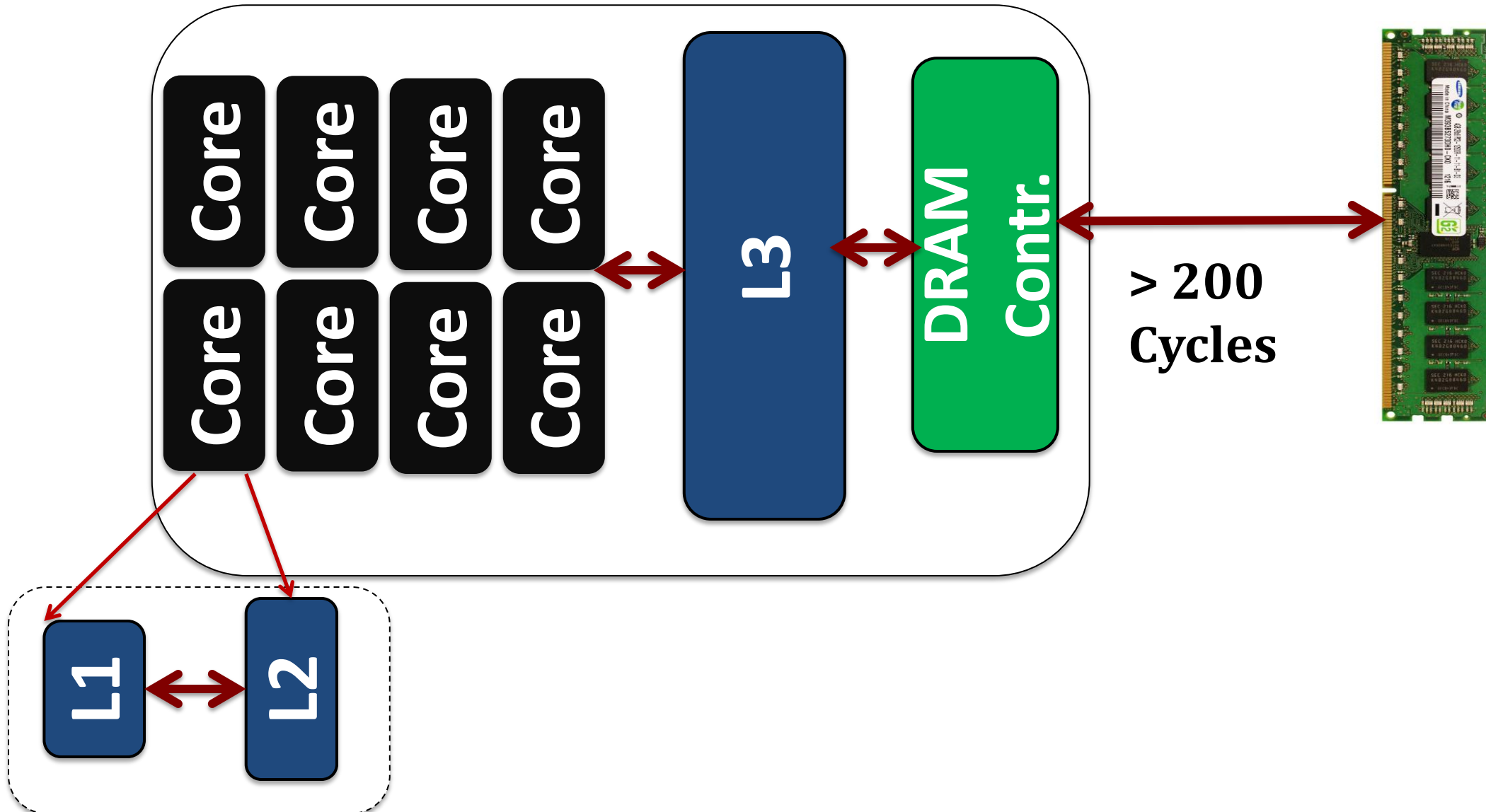
Latency Numbers

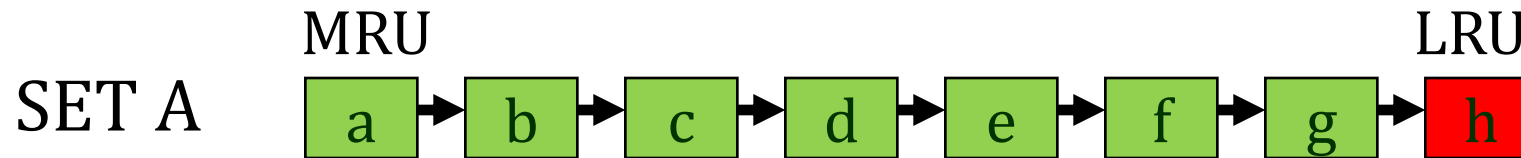**L1**   Few Cycles

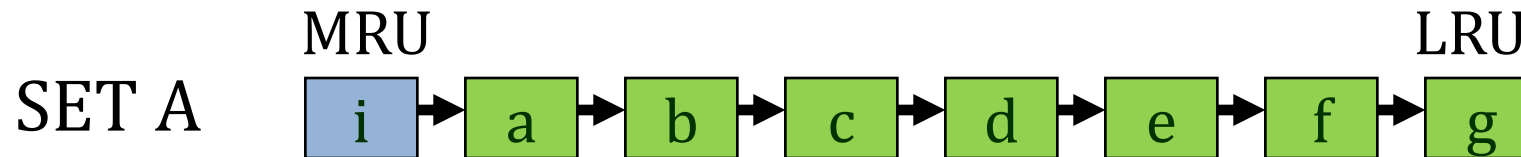**L2**   Tens of Cycles

L3   Two to three times of L2

    Hundreds of cycles

Our Goal:
To minimize off-chip DRAM accesses

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

MRU   LRU

SET A   a → b → c → d → e → f → g → h

Cache Insertion Policy: New block i inserted into MRU.

MRU   LRU

SET A   i → a → b → c → d → e → f → g

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

LRU causes thrashing when working set > cache size
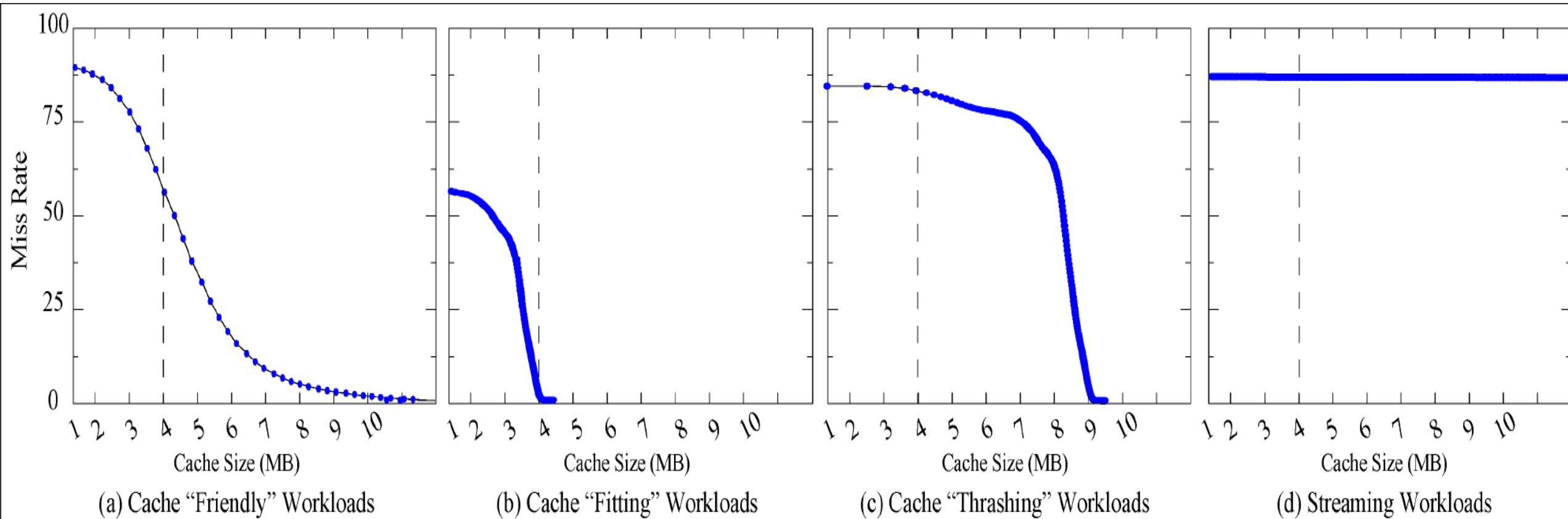
Common Access Patterns [RRIP, ISCA 10]

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

Thrashing $(a_1, a_2, \ldots a_k)^N$ **[k > cache size]**

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

Combination of above three

(a) Cache "Friendly" Workloads

(b) Cache "Fitting" Workloads

(c) Cache "Thrashing" Workloads

(d) Streaming Workloads

Limitations of LRU

LRU exploits **temporal locality**

Streaming data (a_1, a_2, a_3,....a_∞):
No temporal locality,
No temporal reuse

Thrashing data (a_1, a_2, a_3,....a_n) [n>c]
Temporal locality exists. However, LRU fails to capture.

```
if ( rand() < ε ) ε=1/16,1/32,1/64
        Insert at MRU position;
else
        Insert at LRU position;
```
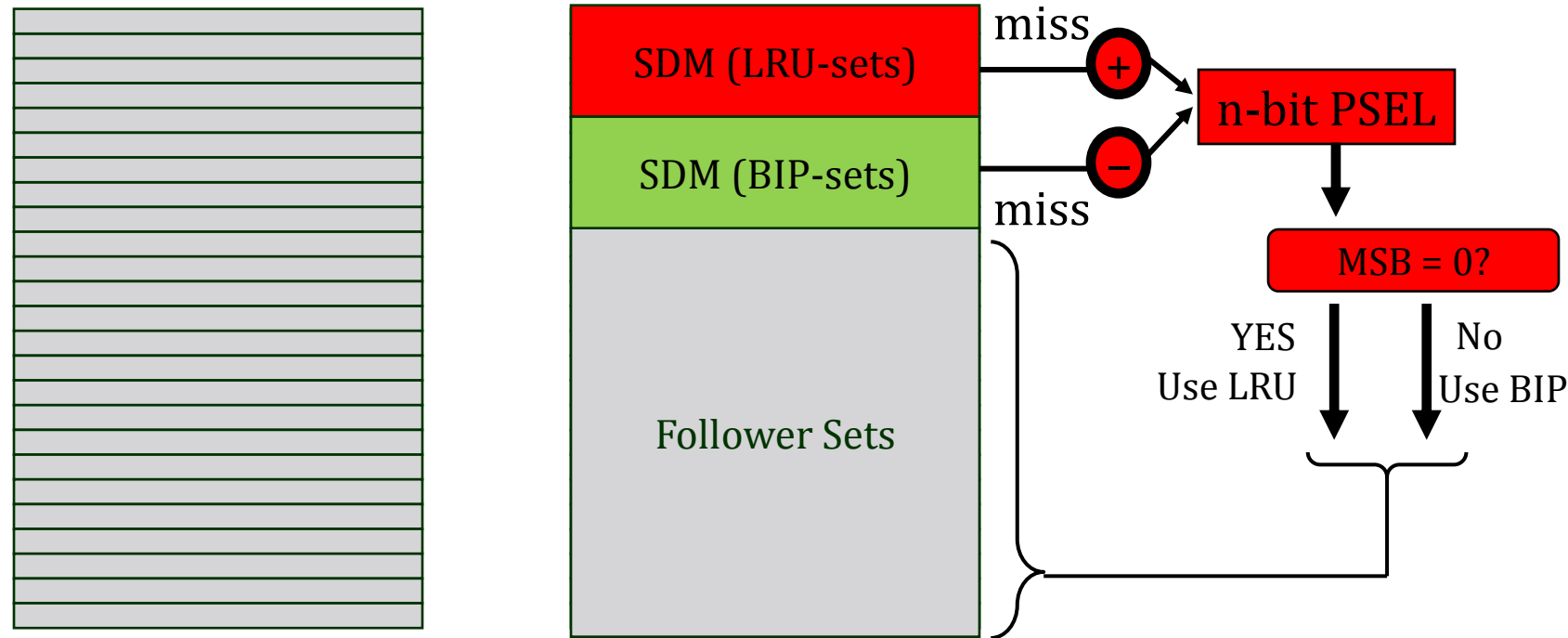
For small ε: BIP retains thrashing protection of LRU insertion policy.
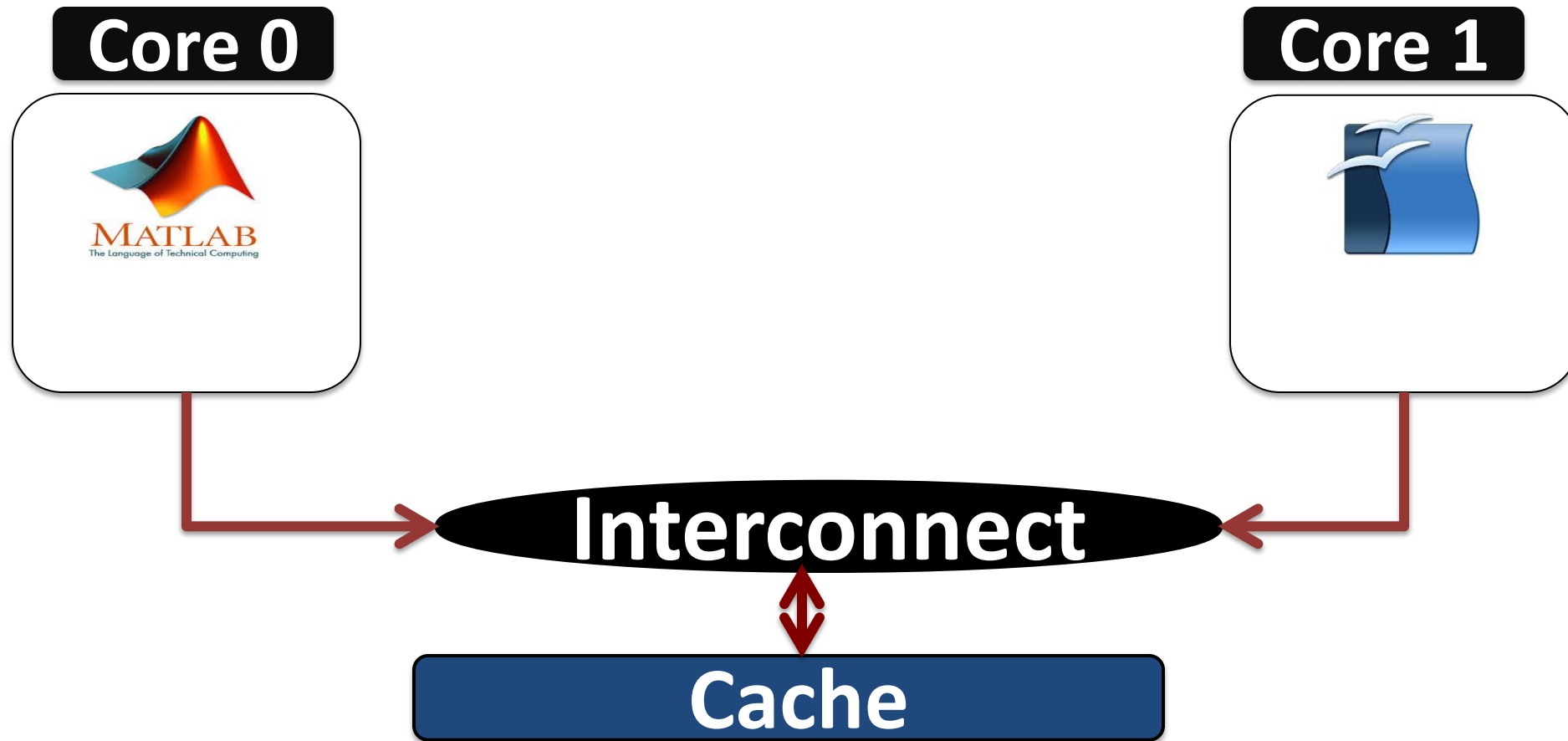
Infrequently insert lines in MRU position

Dynamic Insertion Policy (DIP) [ISCA '07]

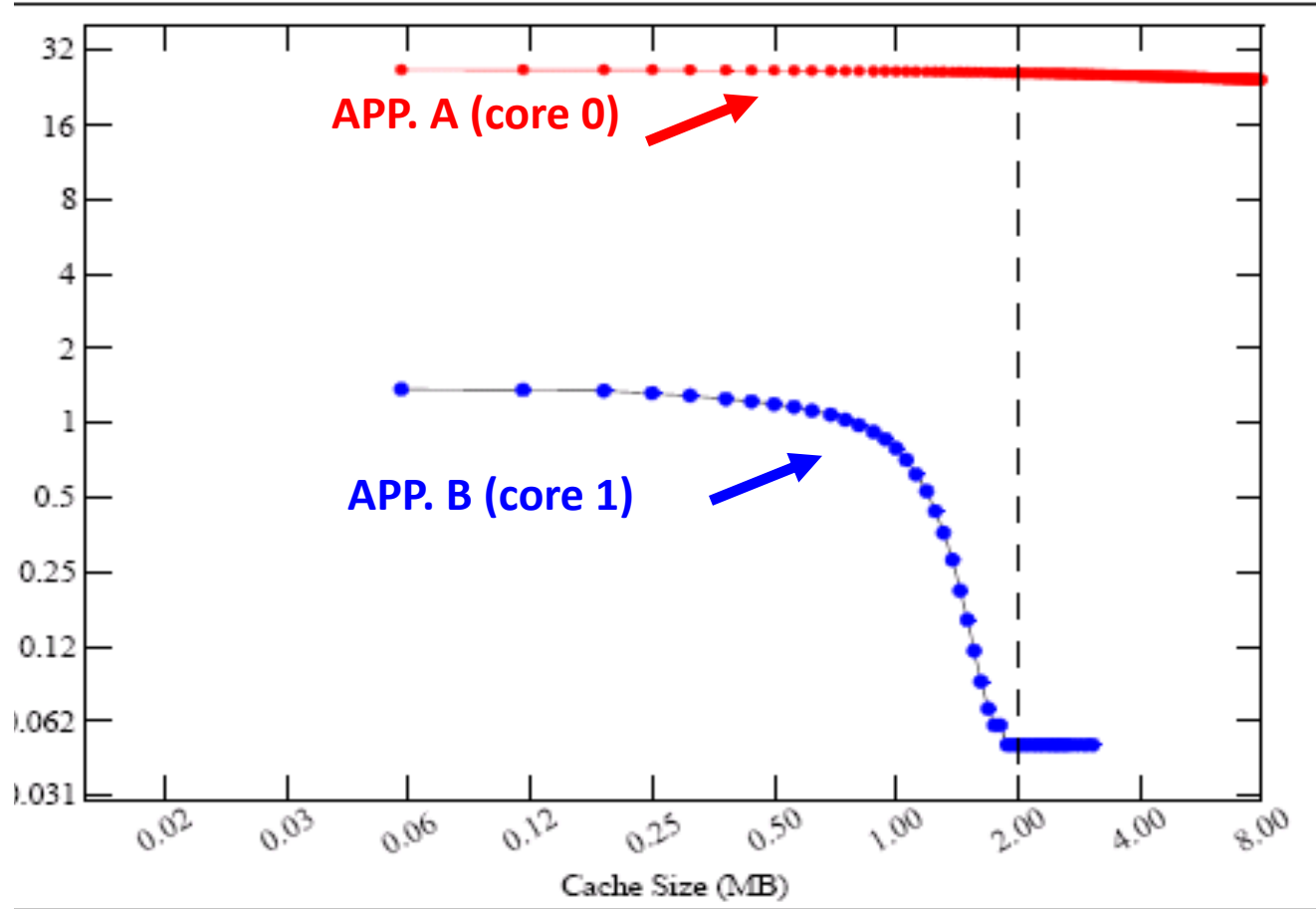SDM – Set Dueling monitors   PSEL – n-bit saturating counters for deciding a policy

What about DIP for shared Caches?

Core 0

Core 1

Interconnect

Cache

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

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

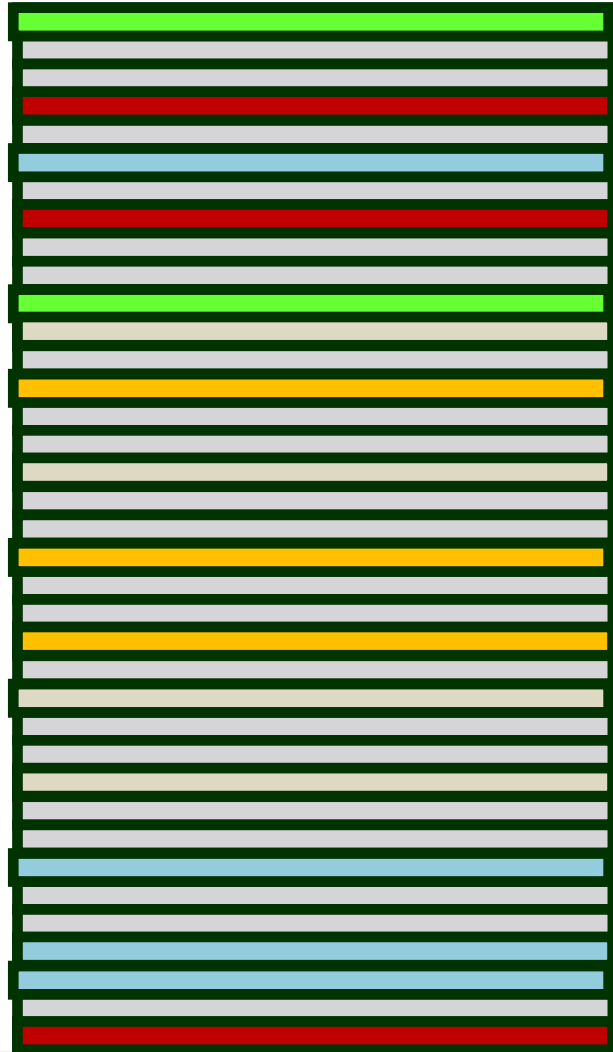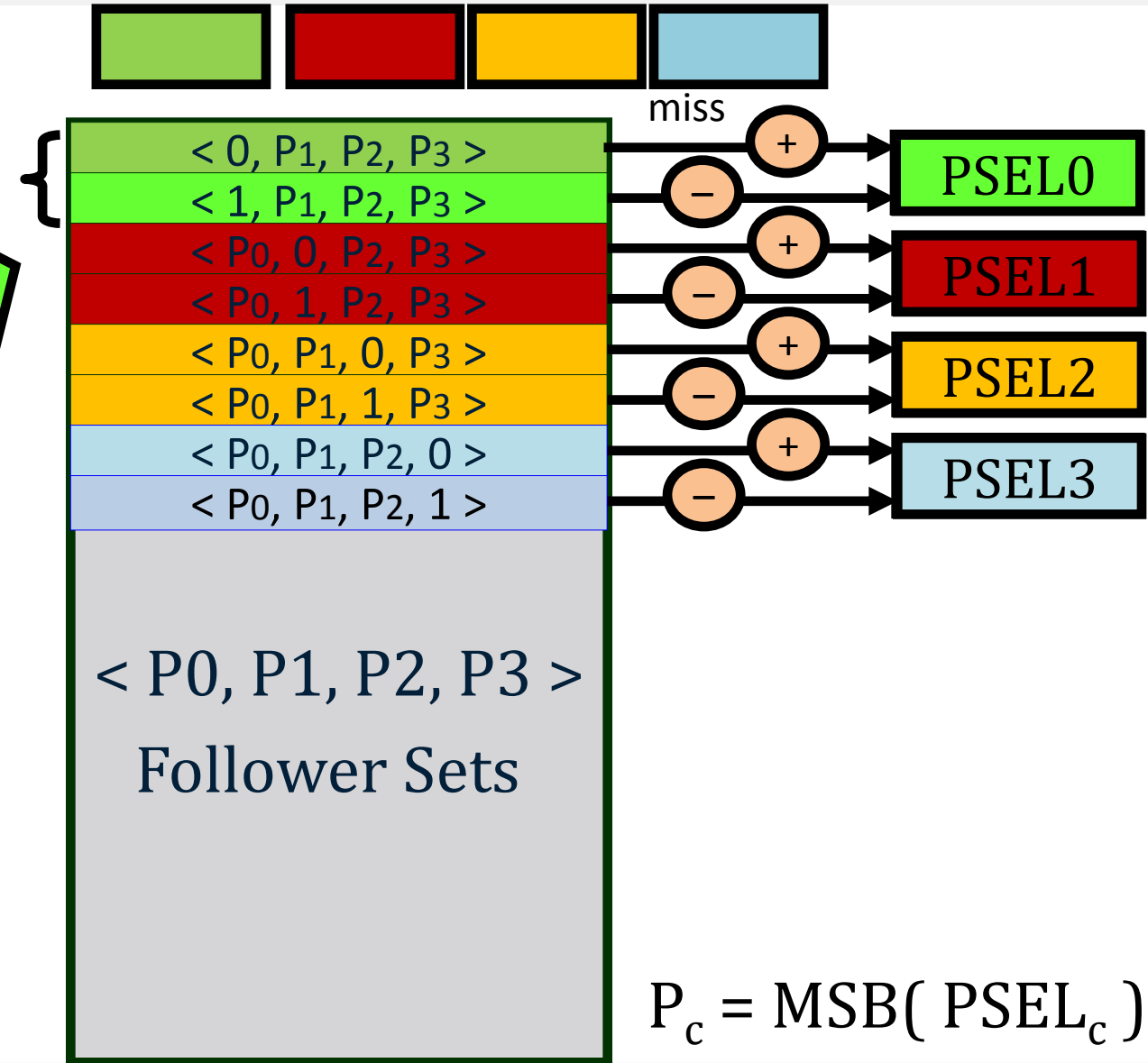DIP does not distinguish between apps. Learning is not adaptive.

Cache Occupancy Under LRU Replacement (2MB Shared Cache)
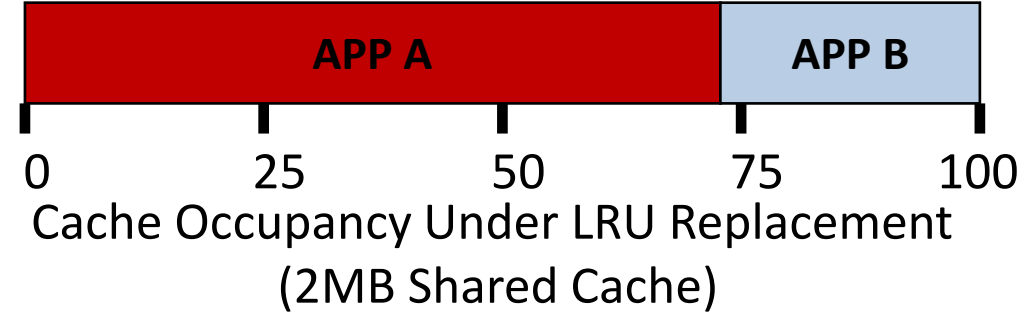
What Should be done?

APP A- LRU/BIP? APP B – LRU/BIP?

What about an N-core system?
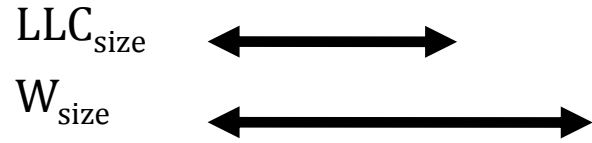
In the presence of other apps, does APP0 doing LRU or BIP improve cache performance?

$< 0, P_1, P_2, P_3 >$

$< 1, P_1, P_2, P_3 >$

$< P_0, 0, P_2, P_3 >$
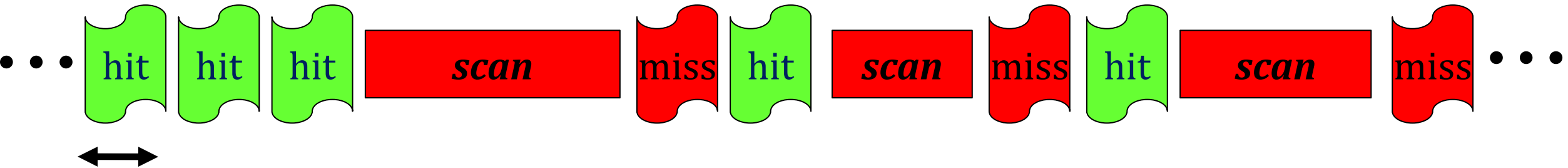
$< P_0, 1, P_2, P_3 >$

$< P_0, P_1, 0, P_3 >$

$< P_0, P_1, 1, P_3 >$

$< P_0, P_1, P_2, 0 >$

$< P_0, P_1, P_2, 1 >$

$< P_0, P_1, P_2, P_3 >$
Follower Sets

miss

PSEL0

PSEL1

PSEL2

PSEL3

$P_c = MSB(PSEL_c)$

DIP vs TA-DIP

Still Miles to Go

LLC_{size} ⟷

W_{size} ⟷

Working set larger than the cache causes thrashing

• • • | miss | miss | miss | miss | miss | • • •

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

• • • | hit | hit | hit | *scan* | miss | hit | *scan* | miss | hit | *scan* | miss | • • •

⟷

Still Miles to Go

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

W_{size}

LLC_{size}
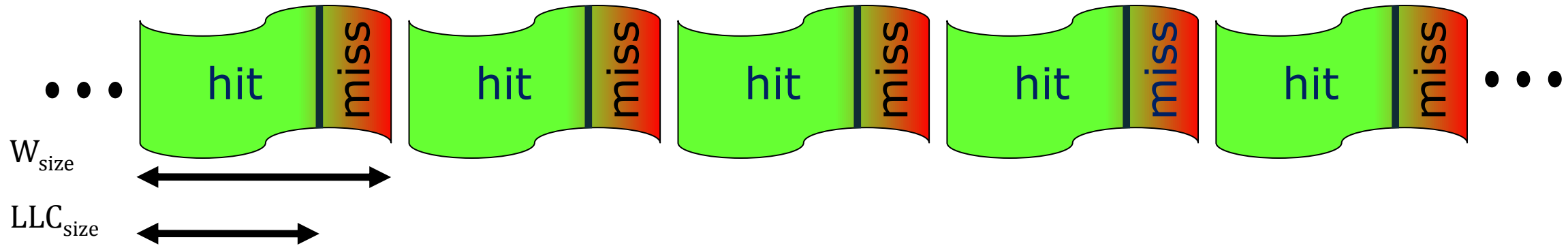
Recurring *scans (bursts of non-temporal data)* ➔ Preserve frequently referenced working set in the cache
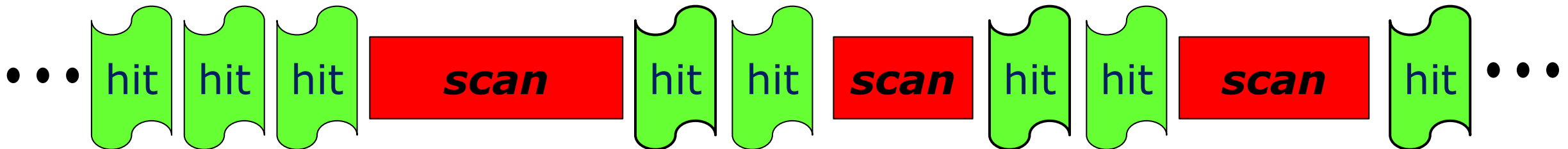
Source: Software Technology Forum

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

victim block

Physical Way # →

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Cache Tag →

| a | c | b | h | f | d | g | e |

"Time" when block will be referenced next →
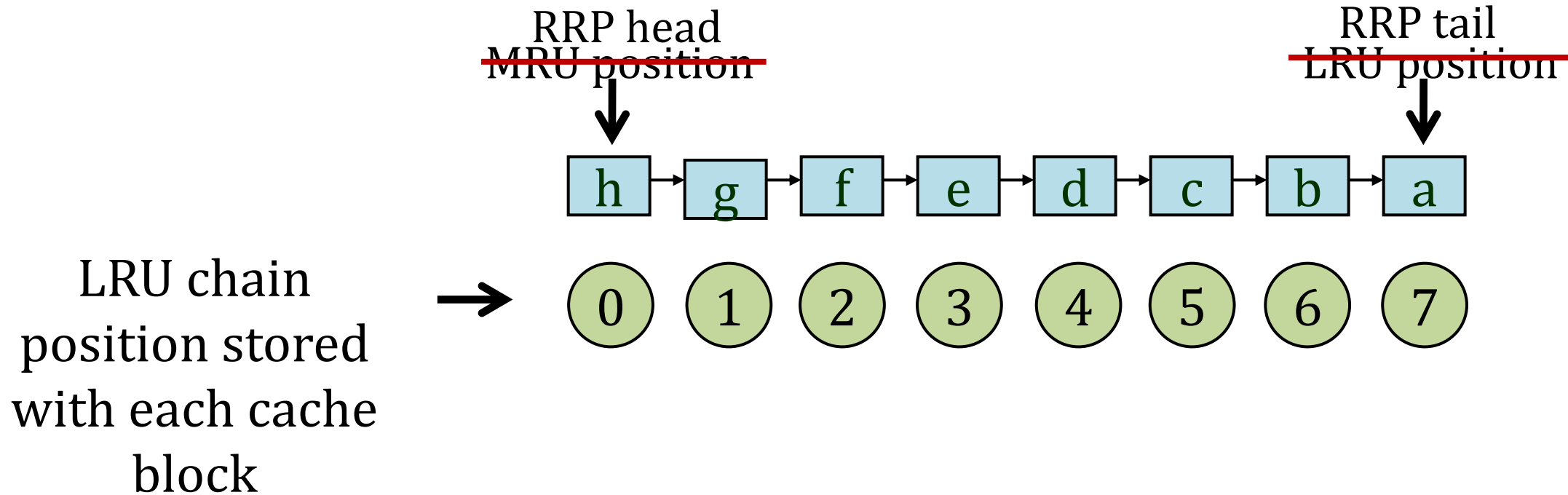
| 4 | 13 | 11 | 5 | 3 | 6 | 9 | 1 |

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

RRP head
~~MRU position~~

RRP tail
~~LRU position~~

| h | g | f | e | d | c | b | a |

LRU chain position stored with each cache block →

0 1 2 3 4 5 6 7

RRP: Re-reference prediction

RRIP

RRP Head

RRP Tail

| h | g | f | e | d | c | b | a |

Re-reference Prediction Value RRPV (n=2):

      0   1   2   3

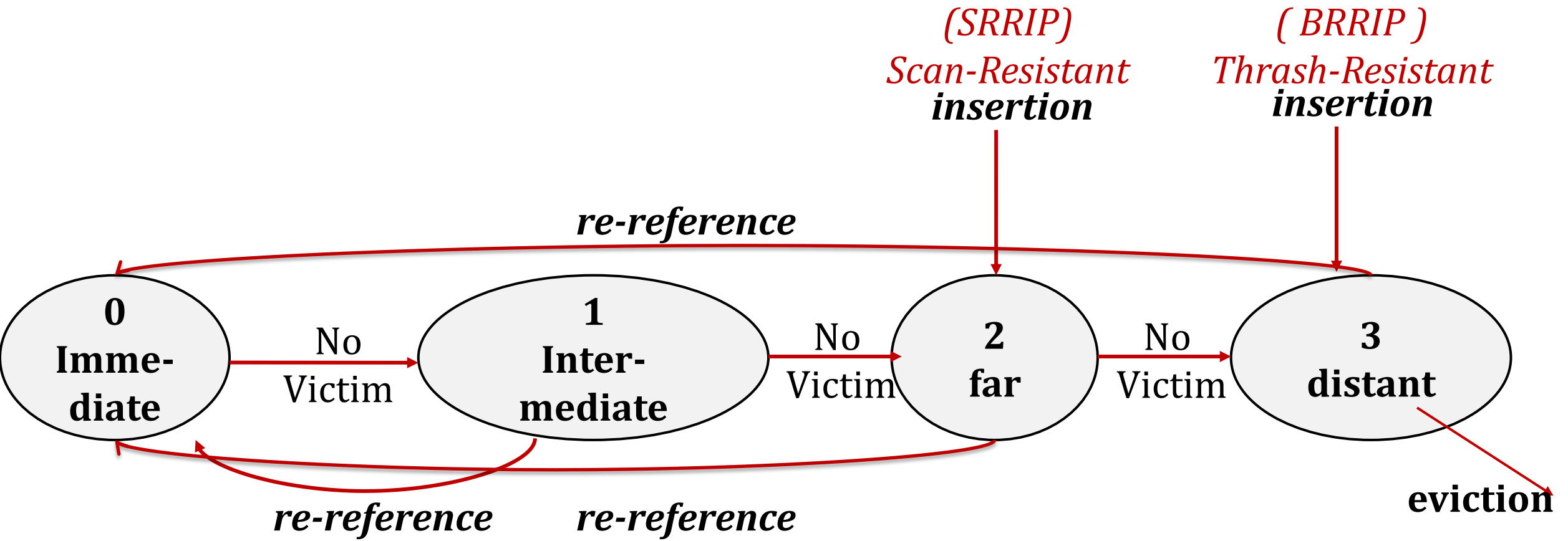Qualitative Prediction: 'near-immediate' 'intermediate' 'far' 'distant'
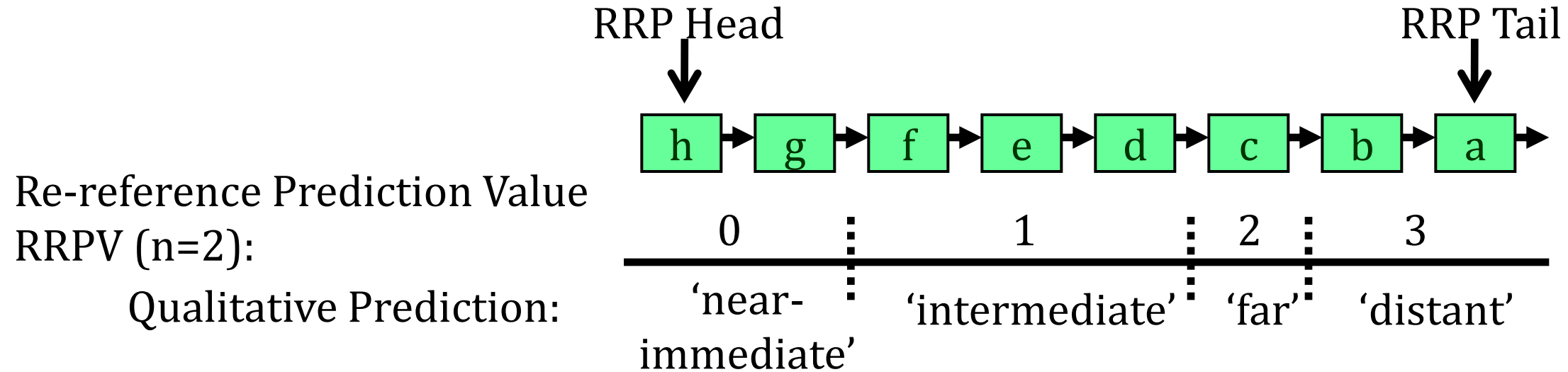
Intuition: New cache block will not be re-referenced soon. Replaces block with distant RRPV.

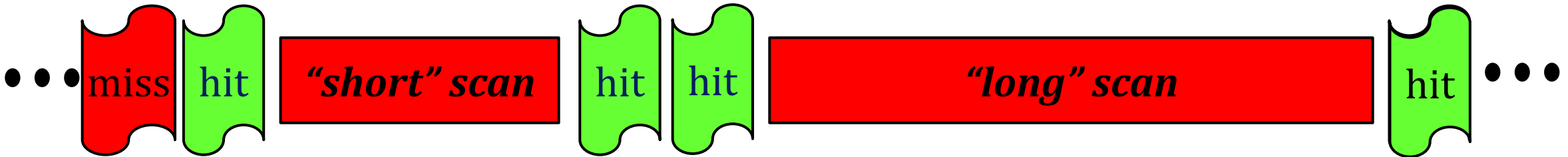Insert with RRPV=2, Evict with RRPV=3 promote blocks with RRPV=0.

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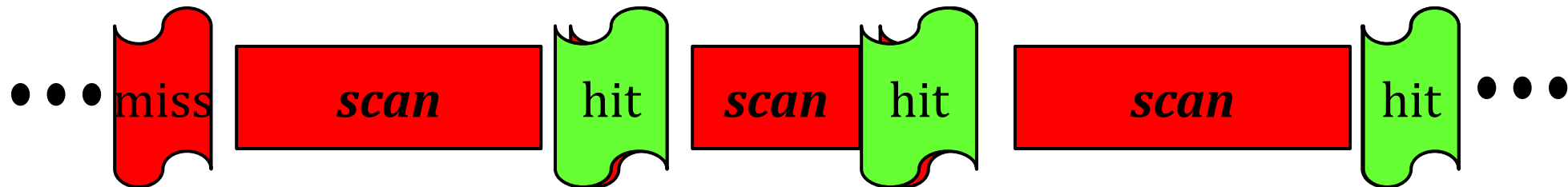
RRIP

SRRIP – Not Good Enough

LONG scans in access pattern

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

Mixed Access Patterns

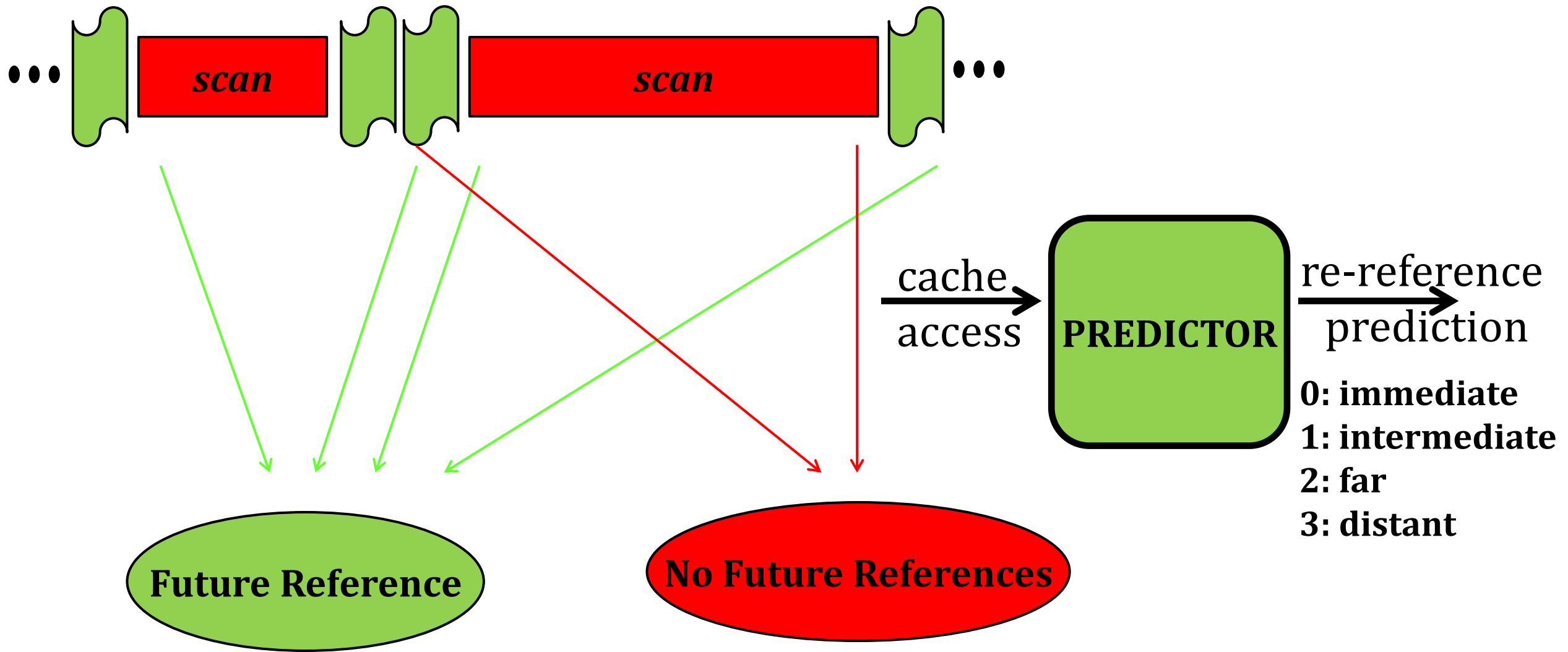**(a1, a2), (a1, a2), b1, b2, b3, (a1, a2)**

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]

scan

scan

cache access → **PREDICTOR** → re-reference prediction

0: immediate
1: intermediate
2: far
3: distant

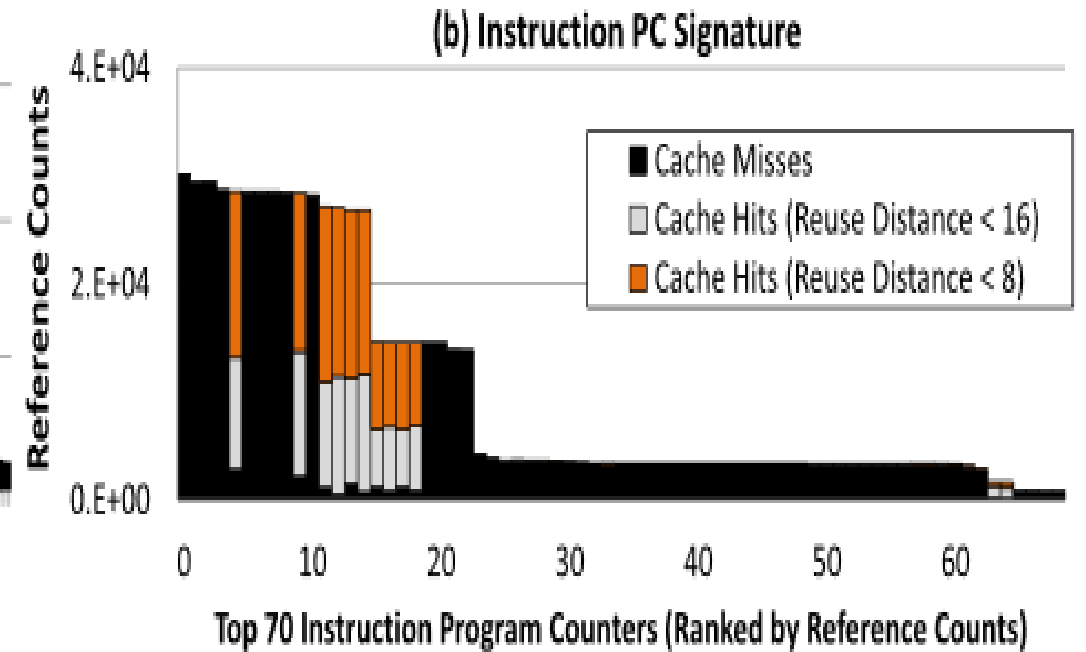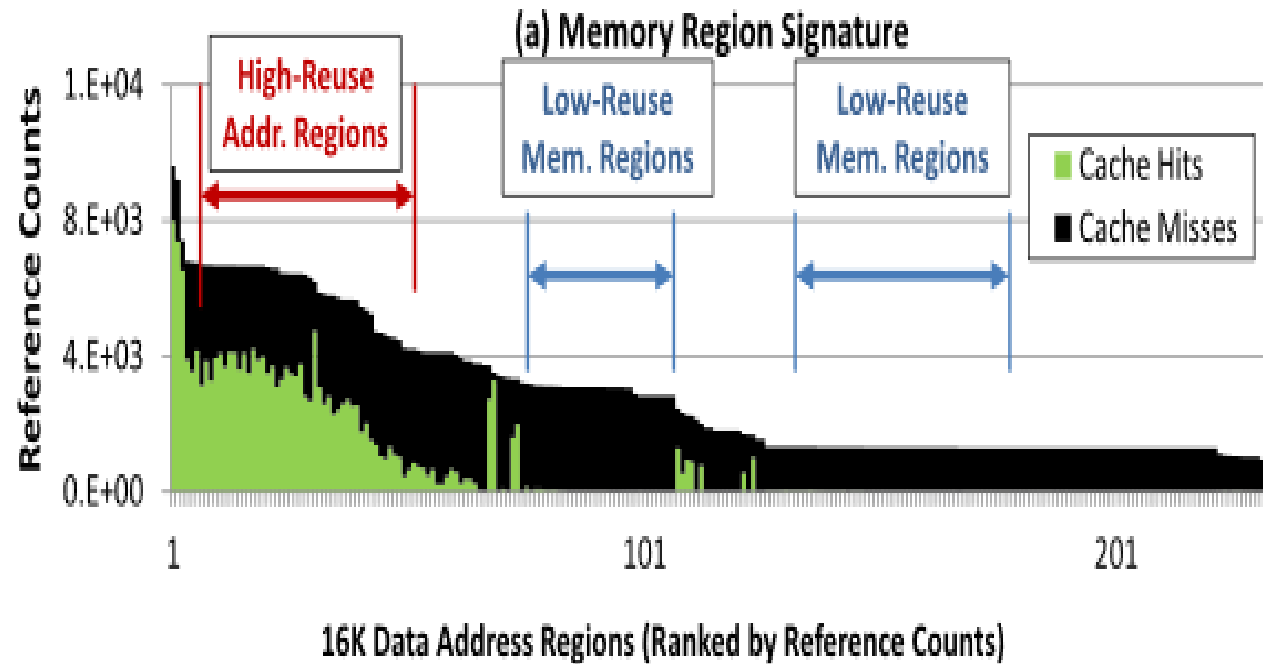**Future Reference**

No Future References

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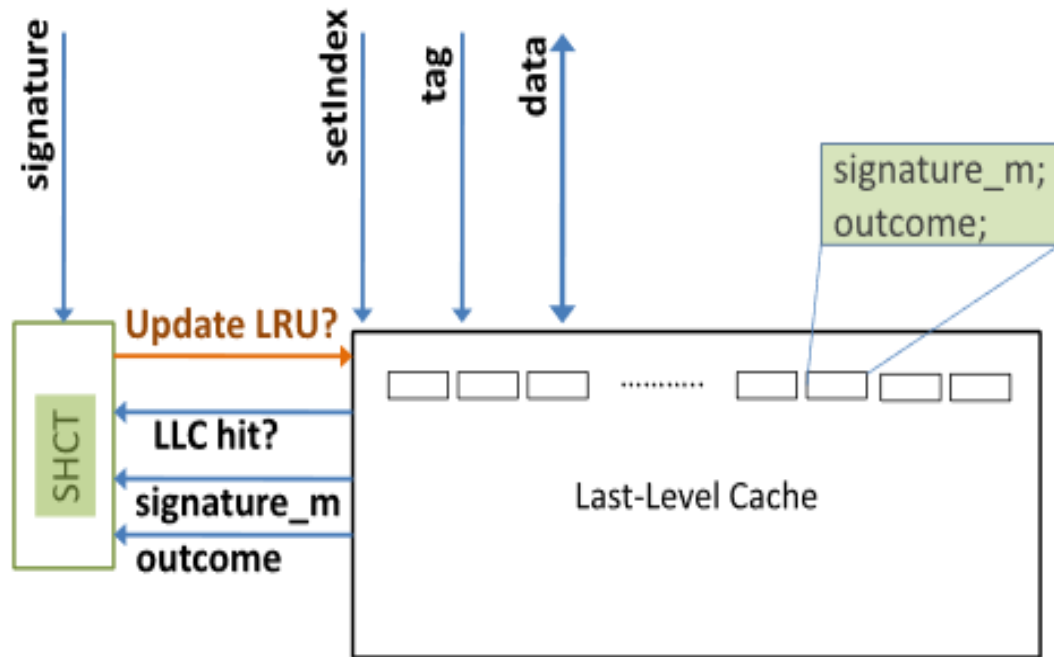
LLC accesses by the same "signature" tend to have similar re-reference patterns

Examples

(a) Memory Region Signature

(b) Instruction PC Signature

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
```