# 15-740/18-740
# Computer Architecture
# Lecture 18: Caching in Multi-Core

Prof. Onur Mutlu
Carnegie Mellon University

# Last Time

- Multi-core issues in prefetching and caching
  - Prefetching coherence misses: push vs. pull
  - Coordinated prefetcher throttling
  - Cache coherence: software versus hardware
  - Shared versus private caches
  - Utility based shared cache partitioning

# Readings in Caching for Multi-Core

- Required
  - Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.

- Recommended (covered in class)
  - Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.
  - Qureshi et al., "Adaptive Insertion Policies for High-Performance Caching," ISCA 2007.

# Software-Based Shared Cache Management

- Assume no hardware support (demand based cache sharing, i.e. LRU replacement)
- How can the OS best utilize the cache?

- Cache sharing aware thread scheduling
  - Schedule workloads that "play nicely" together in the cache
    - E.g., working sets together fit in the cache
    - Requires static/dynamic profiling of application behavior
    - Fedorova et al., "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," PACT 2007.

- Cache sharing aware page coloring
  - Dynamically monitor miss rate over an interval and change virtual to physical mapping to minimize miss rate
    - Try out different partitions

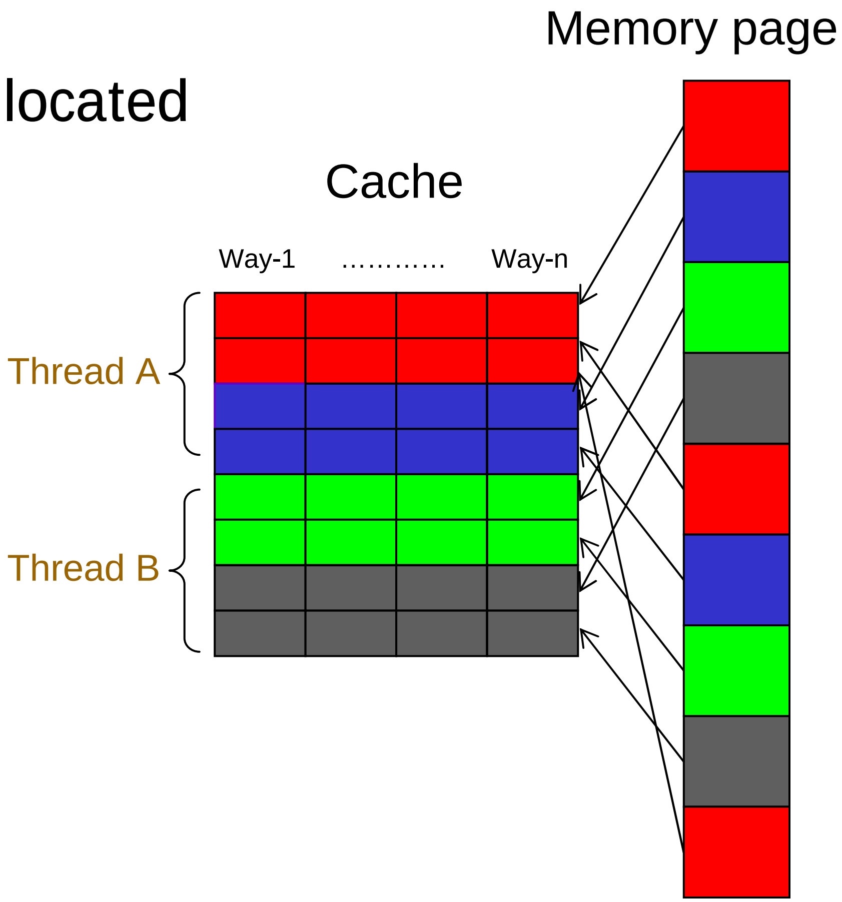
# OS Based Cache Partitioning

- Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.
- Cho and Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," MICRO 2006.

- Static cache partitioning
  - Predetermines the amount of cache blocks allocated to each program at the beginning of its execution
  - Divides shared cache to multiple regions and partitions cache regions through OS-based page mapping
- Dynamic cache partitioning
  - Adjusts cache quota among processes dynamically
  - Page re-coloring
  - Dynamically changes processes' cache usage through OS-based page re-mapping

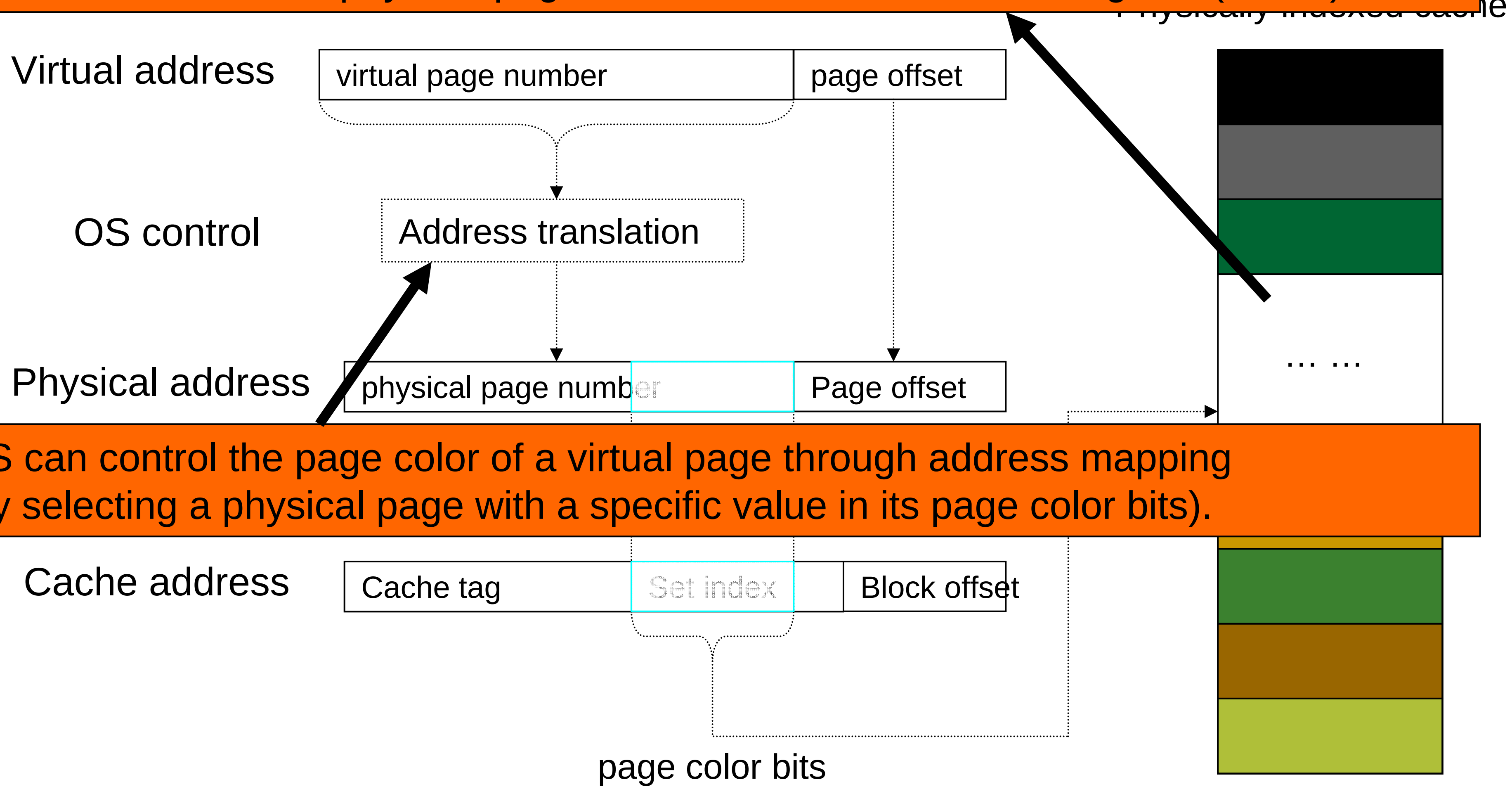
# Page Coloring

- Physical memory divided into colors
- Colors map to different cache sets
- Cache partitioning
  - Ensure two threads are allocated pages of different colors



# Page Coloring

- Physically indexed caches are divided into multiple regions (colors).
- All cache lines in a physical page are cached in one of those regions (colors).



Physically indexed cache

Virtual address

virtual page number

page offset

OS control

Address translation

Physical address

physical page number

Page offset

Cache address

Cache tag

Set index

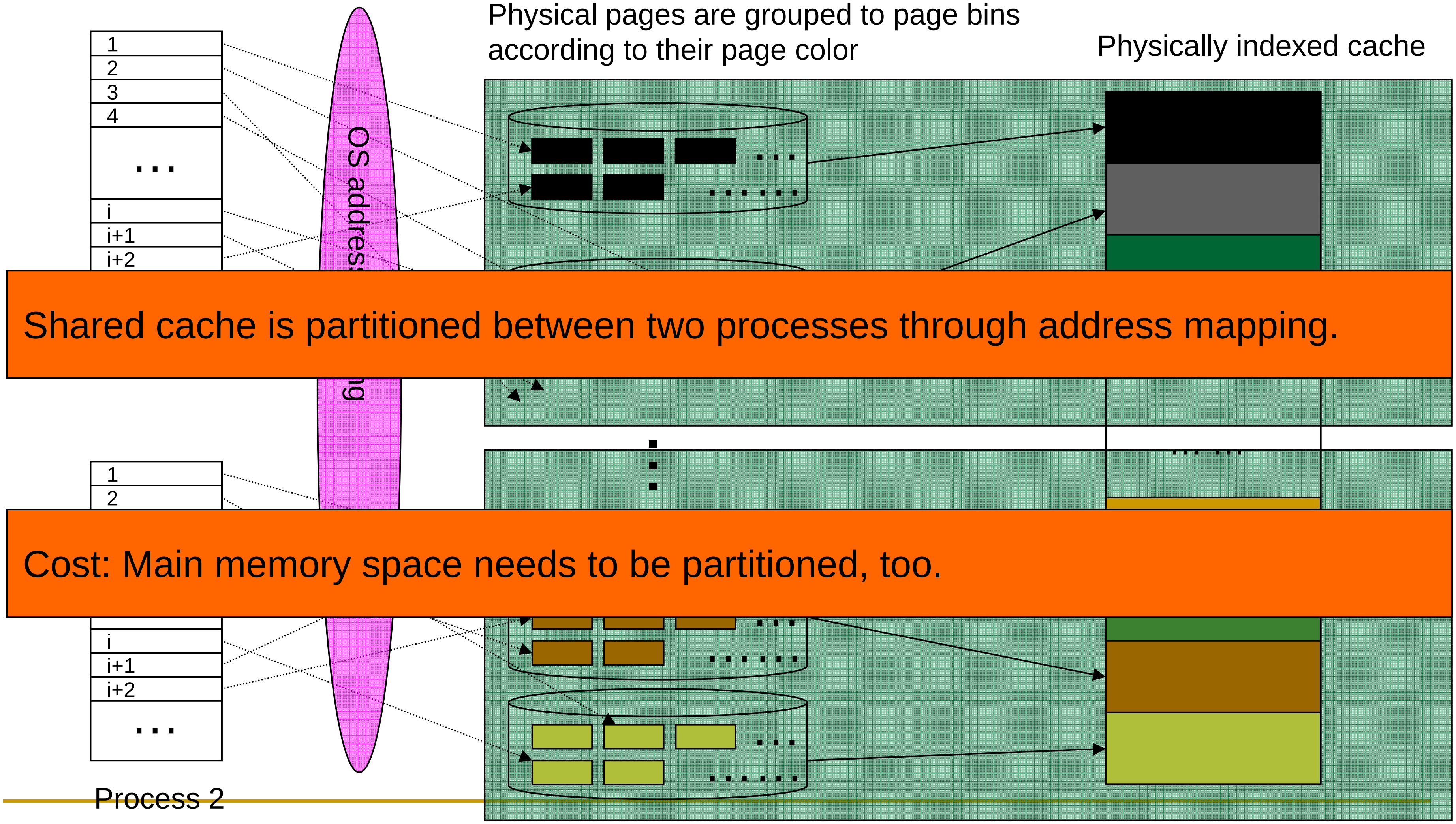
Block offset

page color bits

… …

OS can control the page color of a virtual page through address mapping (by selecting a physical page with a specific value in its page color bits).

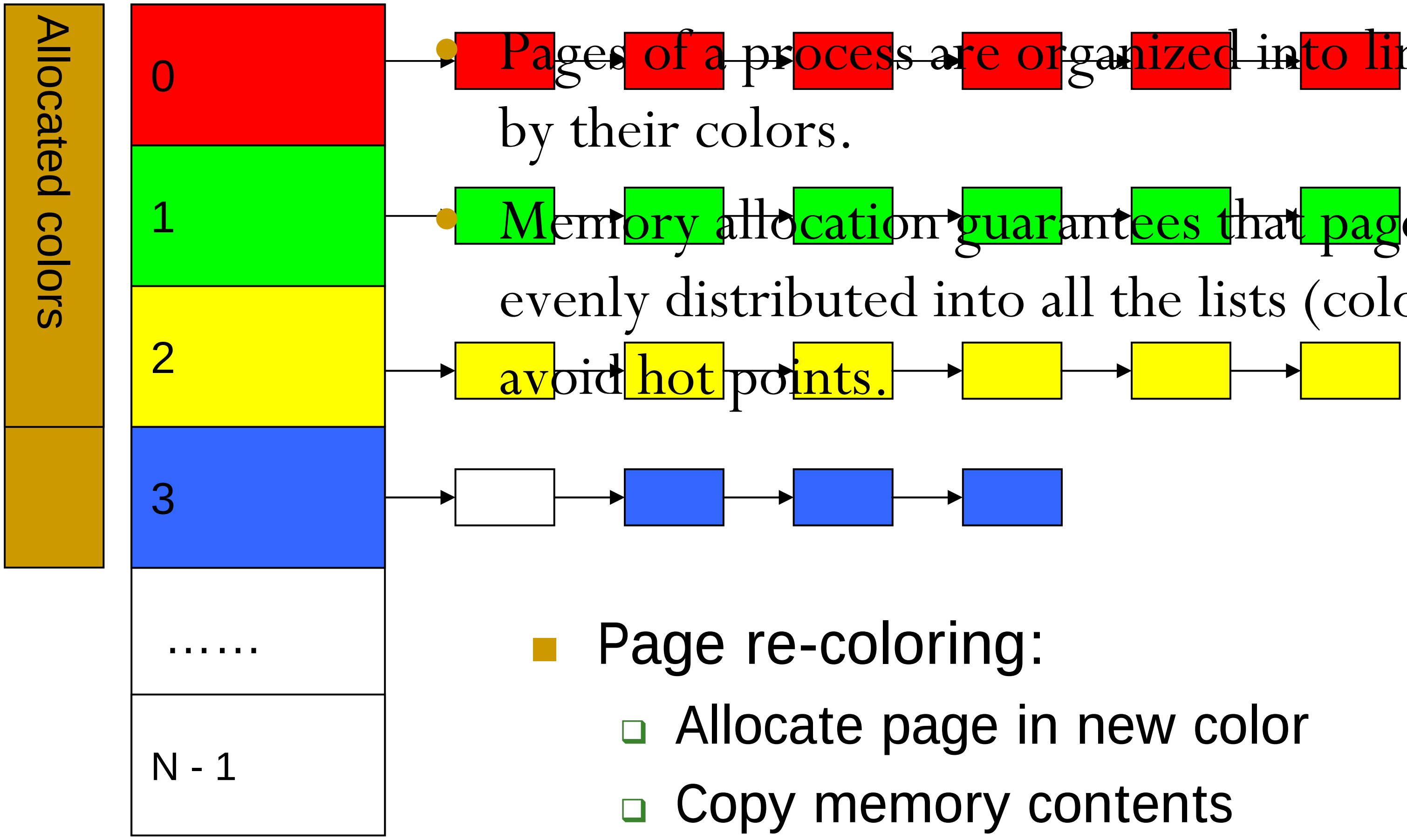
# Static Cache Partitioning using Page Coloring



Shared cache is partitioned between two processes through address mapping.

Cost: Main memory space needs to be partitioned, too.

# Dynamic Cache Partitioning via Page Re-Coloring

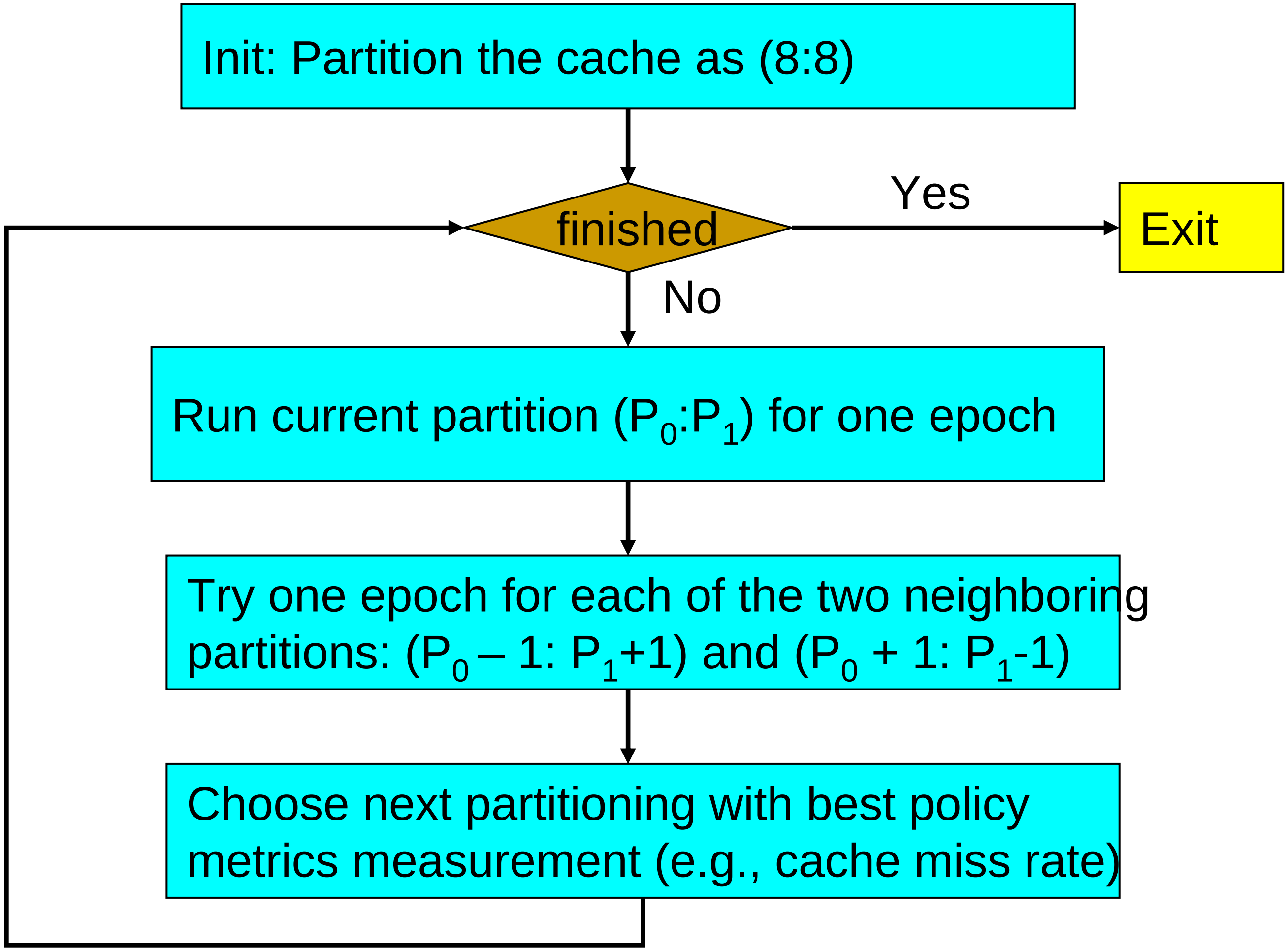


page color table

- Pages of a process are organized into linked lists by their colors.
- Memory allocation guarantees that pages are evenly distributed into all the lists (colors) to avoid hot points.

- Page re-coloring:
  - Allocate page in new color
  - Copy memory contents
  - Free old page

# Dynamic Partitioning in Dual Core



Init: Partition the cache as (8:8)

finished

Yes

Exit

No

Run current partition ($P_0$:$P_1$) for one epoch

Try one epoch for each of the two neighboring partitions: ($P_0$ – 1: $P_1$+1) and ($P_0$ + 1: $P_1$-1)

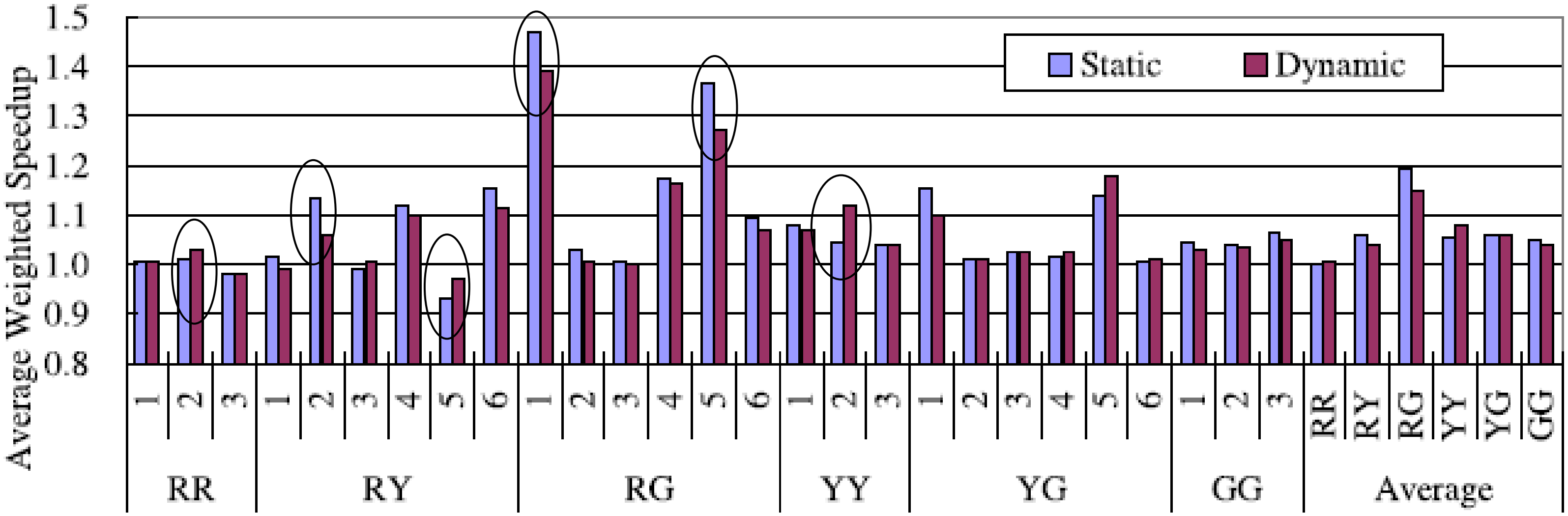
Choose next partitioning with best policy metrics measurement (e.g., cache miss rate)

# Experimental Environment

- Dell PowerEdge1950
  - Two-way SMP, Intel dual-core Xeon 5160
  - Shared 4MB L2 cache, 16-way
  - 8GB Fully Buffered DIMM

- Red Hat Enterprise Linux 4.0
  - 2.6.20.3 kernel
  - Performance counter tools from HP (Pfmon)
  - Divide L2 cache into 16 colors

# Performance – Static & Dynamic



Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.

# Software vs. Hardware Cache Management

- Software advantages

  + No need to change hardware

  + Easier to upgrade/change algorithm (not burned into hardware)

- Disadvantages

  - Less flexible: large granularity (page-based instead of way/block)

  - Limited page colors → reduced performance per application (limited physical memory space!), reduced flexibility

  - Changing partition size has high overhead → page mapping changes

  - Adaptivity is slow: hardware can adapt every cycle (possibly)

  - Not enough information exposed to software (e.g., number of misses due to inter-thread conflict)

# Handling Shared Data in Private Caches

- Shared data and locks ping-pong between processors if caches are private

  -- Increases latency to fetch shared data/locks

  -- Reduces cache efficiency (many invalid blocks)

  -- Scalability problem: maintaining coherence across a large number of private caches is costly

- How to do better?
  - Idea: Store shared data and locks only in one special core's cache. Divert all critical section execution to that core/cache.
    - Essentially, a specialized core for processing critical sections
    - Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.

# Non-Uniform Cache Access

- Large caches take a long time to access
- Wire delay
  - Closeby blocks can be accessed faster, but furthest blocks determine the worst-case access time

- Idea: Variable latency access time in a single cache
- Partition cache into pieces
  - Each piece has different latency
  - Which piece does an address map to?
    - Static: based on bits in address
    - Dynamic: any address can map to any piece
      - How to locate an address?
      - Replacement and placement policies?
- Kim et al., “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” ASPLOS 2002.

# Multi-Core Cache Efficiency: Bandwidth Filters

- Caches act as a filter that reduce memory bandwidth requirement
  - Cache hit: No need to access memory
  - This is in addition to the latency reduction benefit of caching
  - GPUs use caches to reduce memory BW requirements

- Efficient utilization of cache space becomes more important with multi-core
  - Memory bandwidth is more valuable
    - Pin count not increasing as fast as # of transistors
      - 10% vs. 2x every 2 years
  - More cores put more pressure on the memory bandwidth

- How to make the bandwidth filtering effect of caches better?

# Revisiting Cache Placement (Insertion)

- Is inserting a fetched/prefetched block into the cache (hierarchy) always a good idea?
  - No allocate on write: does not allocate a block on write miss
  - How about reads?

- Allocating on a read miss

  -- Evicts another potentially useful cache block

  + Incoming block potentially more useful

- Ideally:
  - we would like to place those blocks whose caching would be most useful in the future
  - we certainly do not want to cache never-to-be-used blocks

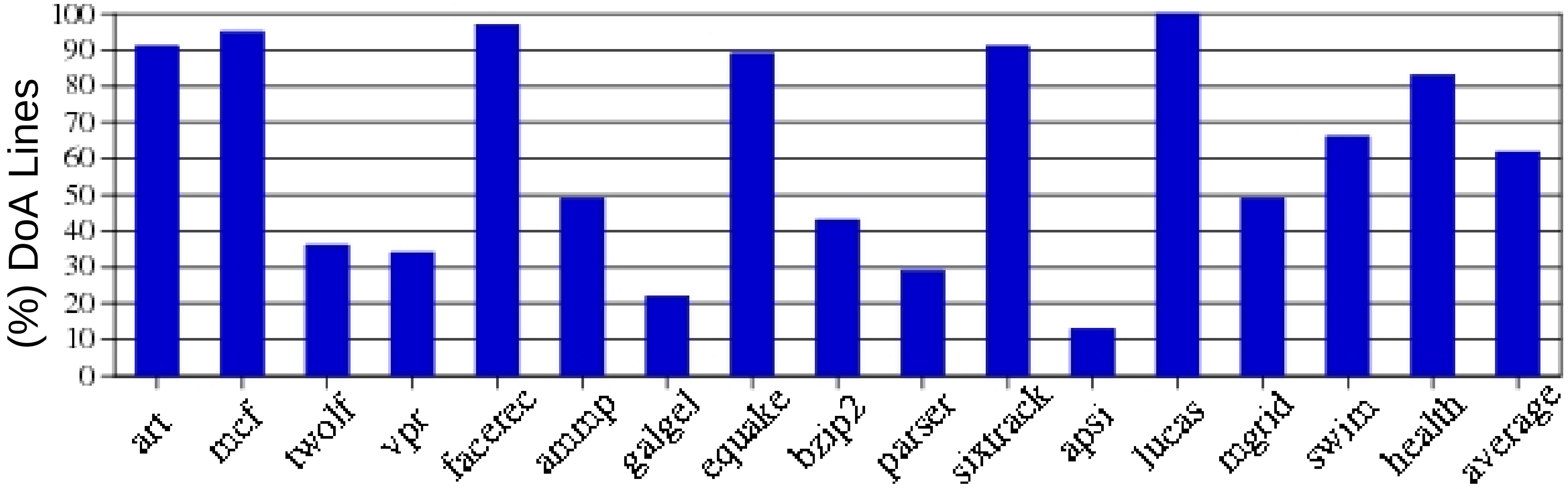
# Revisiting Cache Placement (Insertion)

- Ideas:
  - Hardware predicts blocks that are not going to be used
    - Lai et al., "Dead Block Prediction," ISCA 2001.
  - Software (programmer/compiler) marks instructions that touch data that is not going to be reused
    - How does software determine this?

- Streaming versus non-streaming accesses
  - If a program is streaming through data, reuse likely occurs only for a limited period of time
  - If such instructions are marked by the software, the hardware can store them temporarily in a smaller buffer (L0 cache) instead of the cache

# Reuse at L2 Cache Level

DoA Blocks: Blocks unused between insertion and eviction

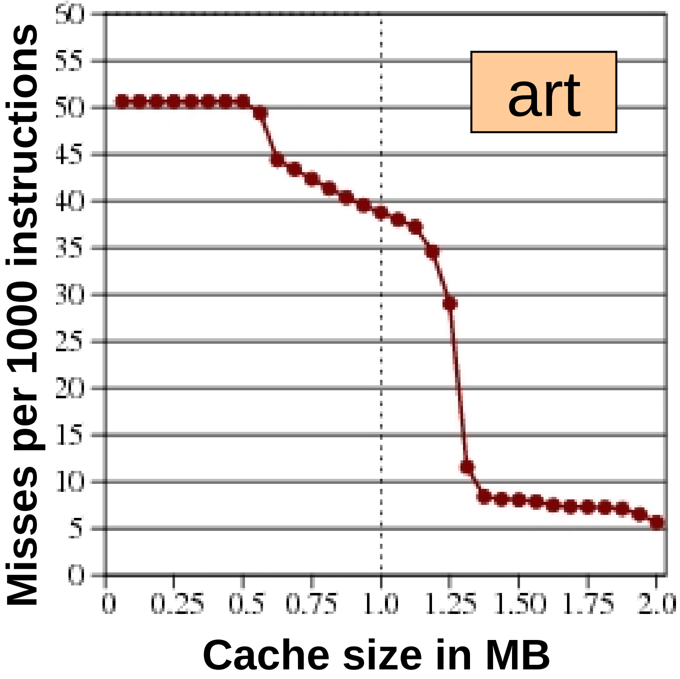


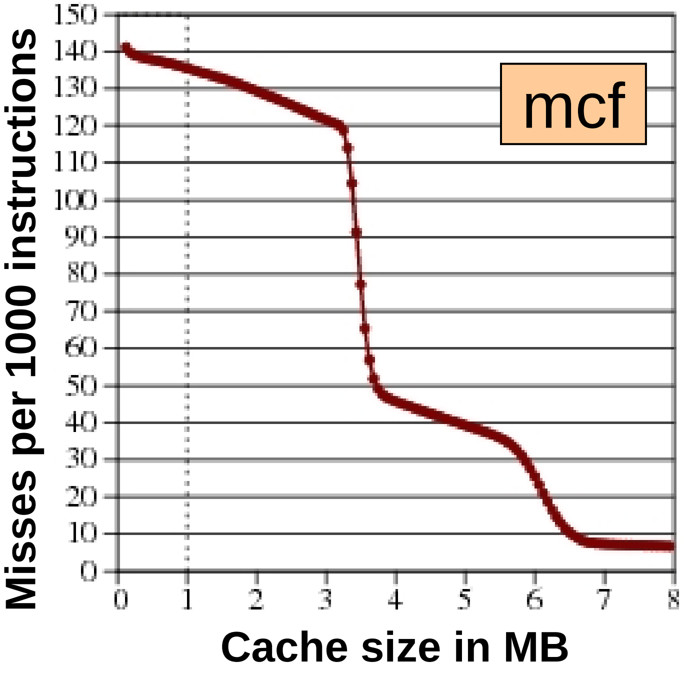
For the 1MB 16-way L2, 60% of lines are DoA

➔ Ineffective use of cache space

# Why Dead on Arrival Blocks?

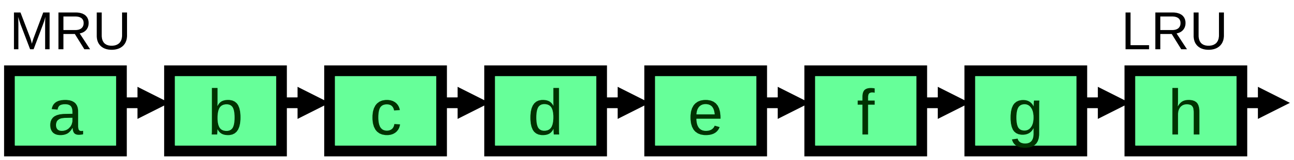
- ❑ Streaming data ➔ Never reused. L2 caches don't help.
- ❑ Working set of application greater than cache size



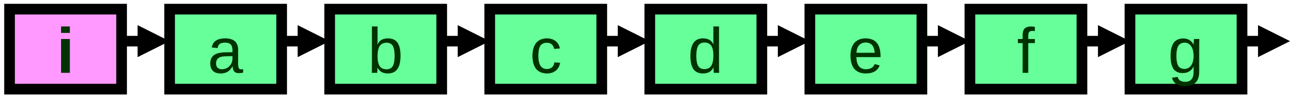


Solution: if working set > cache size, retain some working set

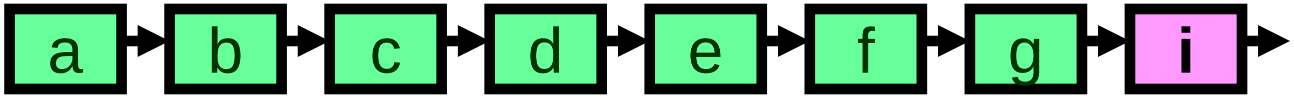
# Cache Insertion Policies: MRU vs. LRU



Reference to ‘i’ with traditional LRU policy:



Reference to ‘i’ with LIP:



Choose victim. Do NOT promote to MRU

Lines do not enter non-LRU positions unless reused

# Other Insertion Policies: Bimodal Insertion

LIP does not age older lines

Infrequently insert lines in MRU position

Let $\varepsilon$ = Bimodal throttle parameter

```
if ( rand() < $\varepsilon$ )
        Insert at MRU position;
else
        Insert at LRU position;
```

For small $\varepsilon$ , BIP retains thrashing protection of LIP while responding to changes in working set

# Analysis with Circular Reference Model

Reference stream has T blocks and repeats N times.
Cache has K blocks (K<T and N>>T)

Two consecutive reference streams:

| Policy | $(a_1\ a_2\ a_3\ \ldots\ a_T)^N$ | $(b_1\ b_2\ b_3\ \ldots\ b_T)^N$ |
|---|---|---|
| LRU | 0 | 0 |
| OPT | (K-1)/T | (K-1)/T |
| LIP | (K-1)/T | 0 |
| BIP (small ε) | ≈ (K-1)/T | ≈ (K-1)/T |

For small ε , BIP retains thrashing protection of LIP
while adapting to changes in working set

# Analysis with Circular Reference Model

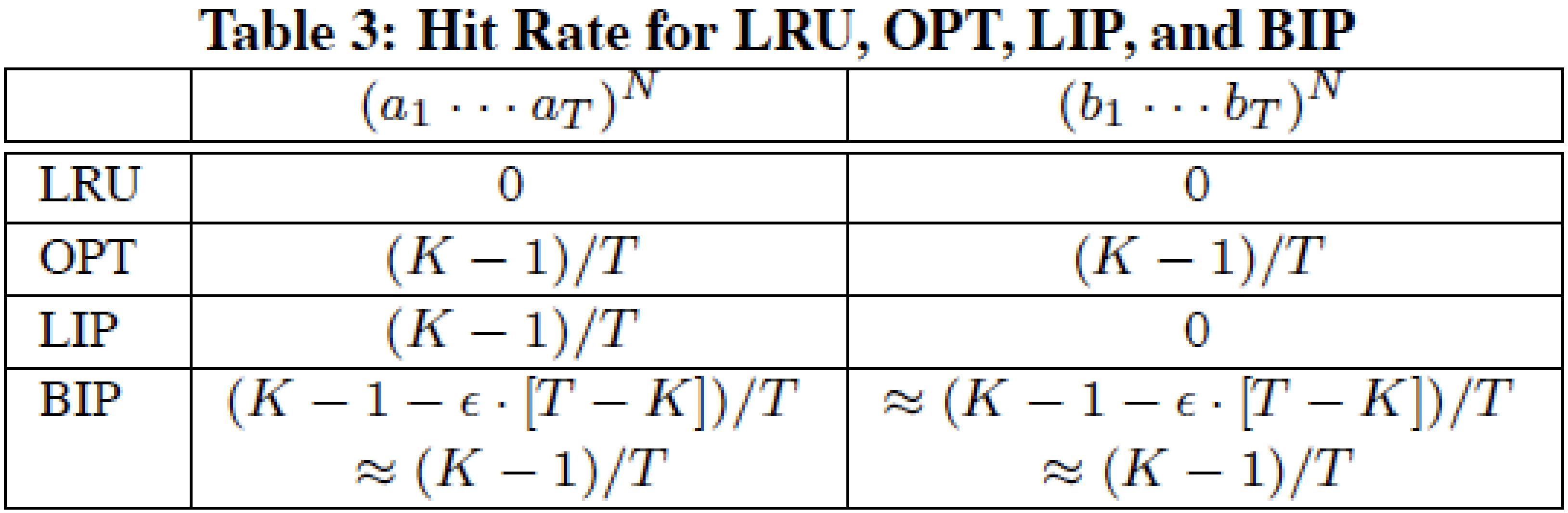
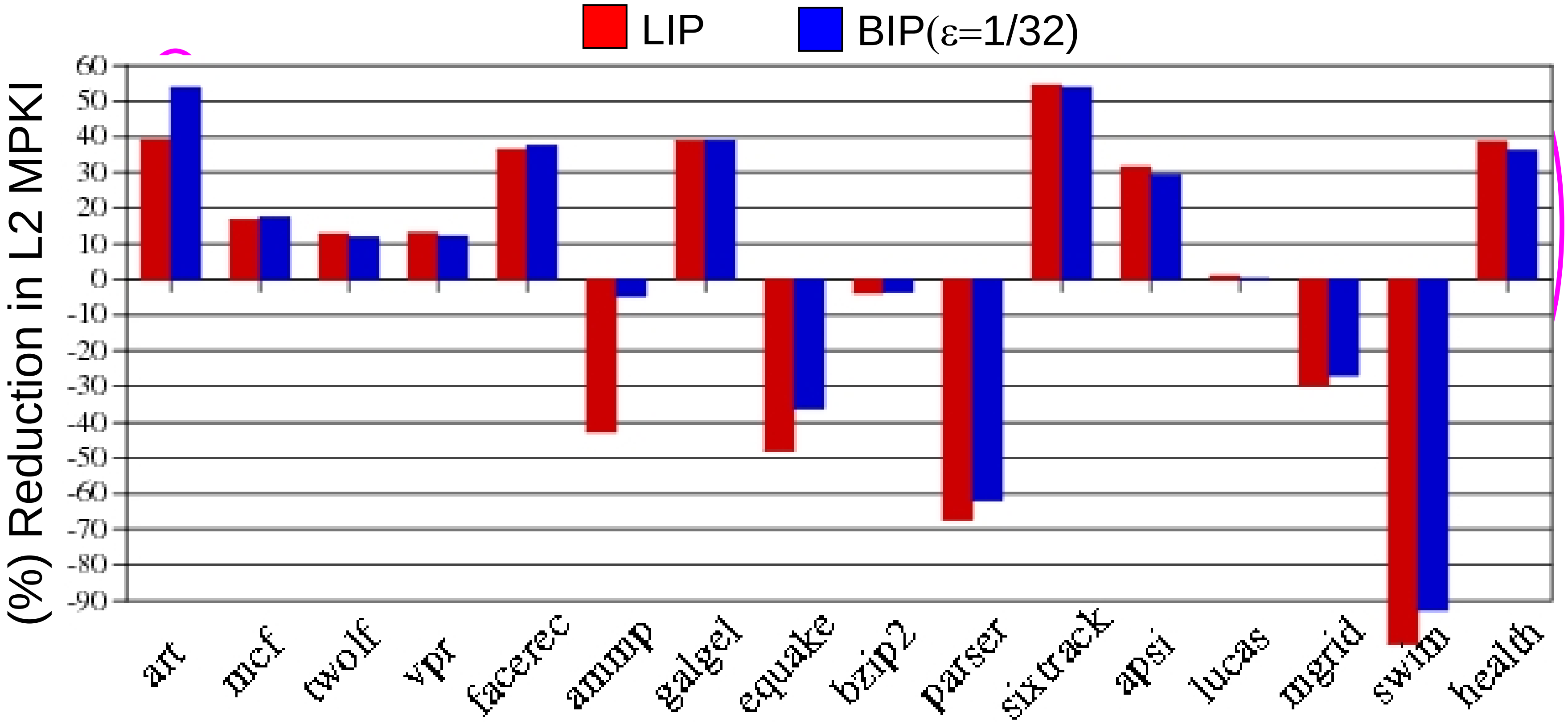


**Table 3: Hit Rate for LRU, OPT, LIP, and BIP**

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

# LIP and BIP Performance vs. LRU



Changes to insertion policy increases misses for LRU-friendly workloads

# Dynamic Insertion Policy (DIP)

- Qureshi et al., "Adaptive Insertion Policies for High-Performance Caching," ISCA 2007.

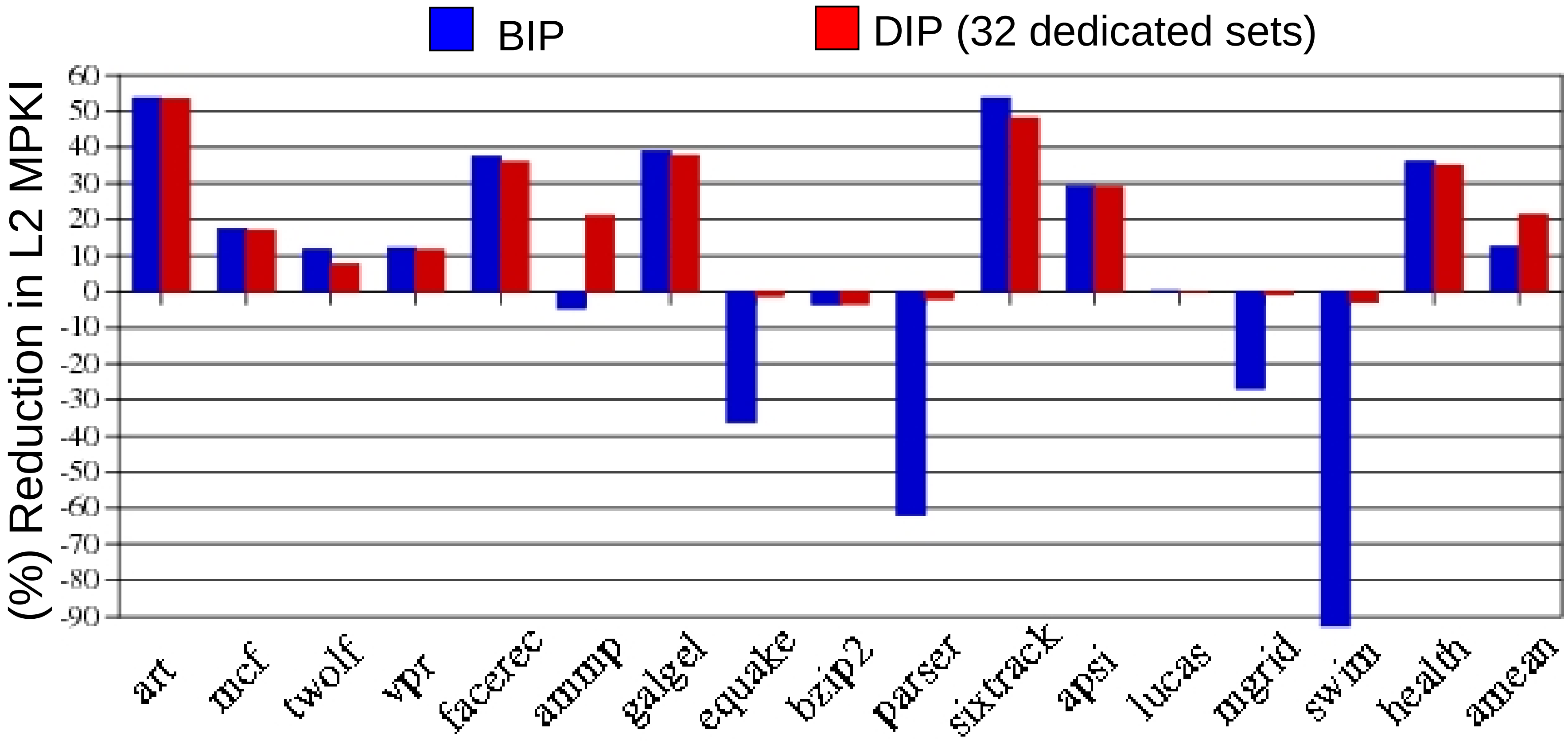
Two types of workloads: LRU-friendly or BIP-friendly

DIP can be implemented by:

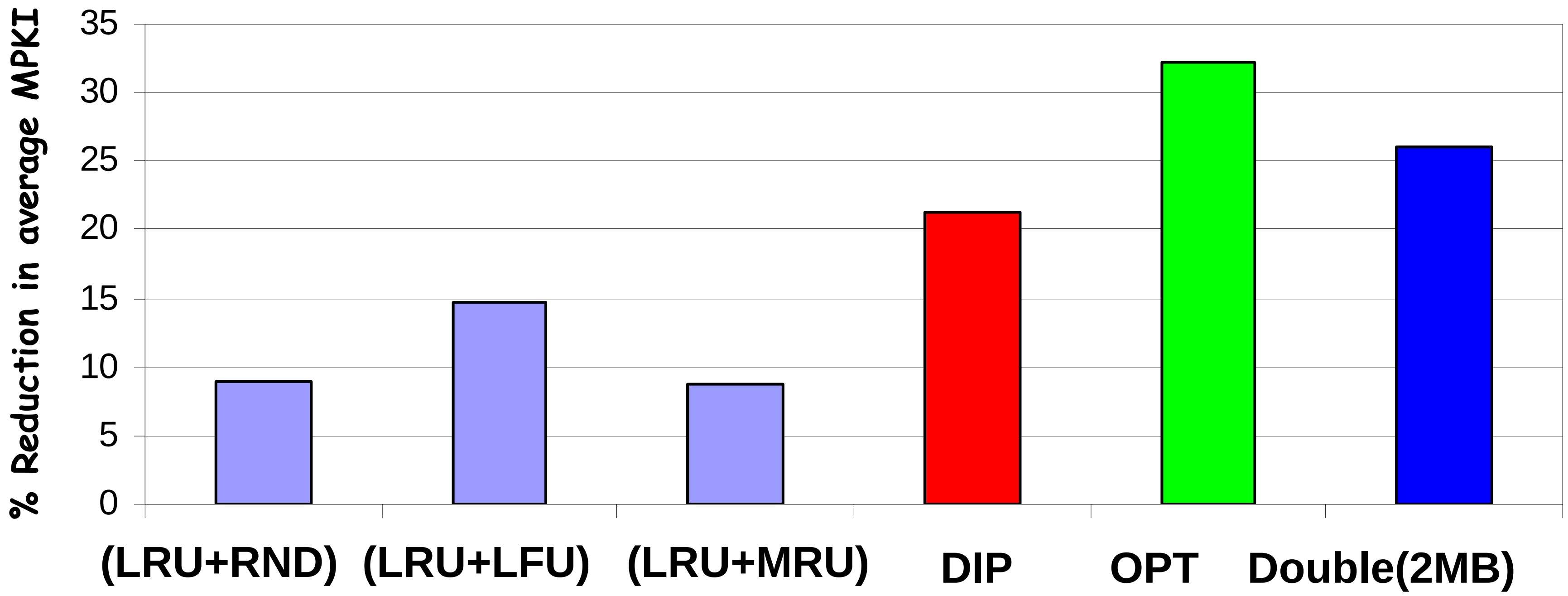
1. Monitor both policies (LRU and BIP)
2. Choose the best-performing policy
3. Apply the best policy to the cache

Need a cost-effective implementation ➔ Set Sampling

# Dynamic Insertion Policy Miss Rate



# DIP vs. Other Policies


% Reduction in average MPKI
(LRU+RND)
(LRU+LFU)
(LRU+MRU)
DIP
OPT
Double(2MB)