15-740/18-740 Computer Architecture Lecture 17: Prefetching, Caching, Multi-core

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Milestone meetings
  - Meet with Evangelos, Lavanya, Vivek
  - And, me... especially if you receive(d) my feedback and I asked to meet

Last Time

- Markov Prefetching
- Content Directed Prefetching
- Execution Based Prefetchers

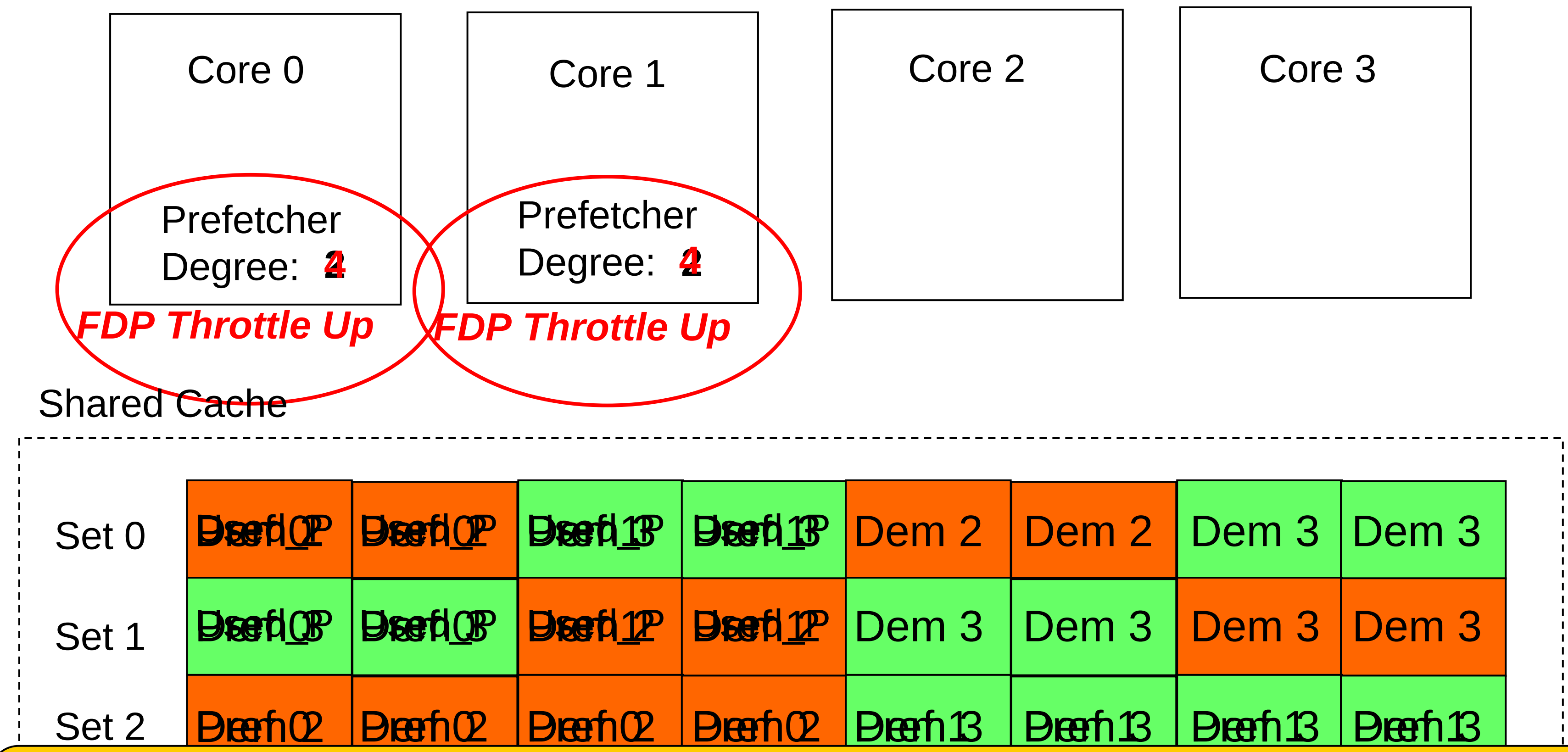
Multi-Core Issues in Prefetching and Caching

Prefetching in Multi-Core (I)

- Prefetching shared data
  - Coherence misses

- Prefetch efficiency a lot more important
  - Bus bandwidth more precious
  - Prefetchers on different cores can deny service to each other and to demand requests
    - DRAM contention
    - Bus contention
    - Cache conflicts
  - Need to coordinate the actions of independent prefetchers for best system performance
    - Each prefetcher has different accuracy, coverage, timeliness

Shortcoming of Local Prefetcher Throttling



Local-only prefetcher control techniques have no mechanism to detect inter-core interference

Prefetching in Multi-Core (II)

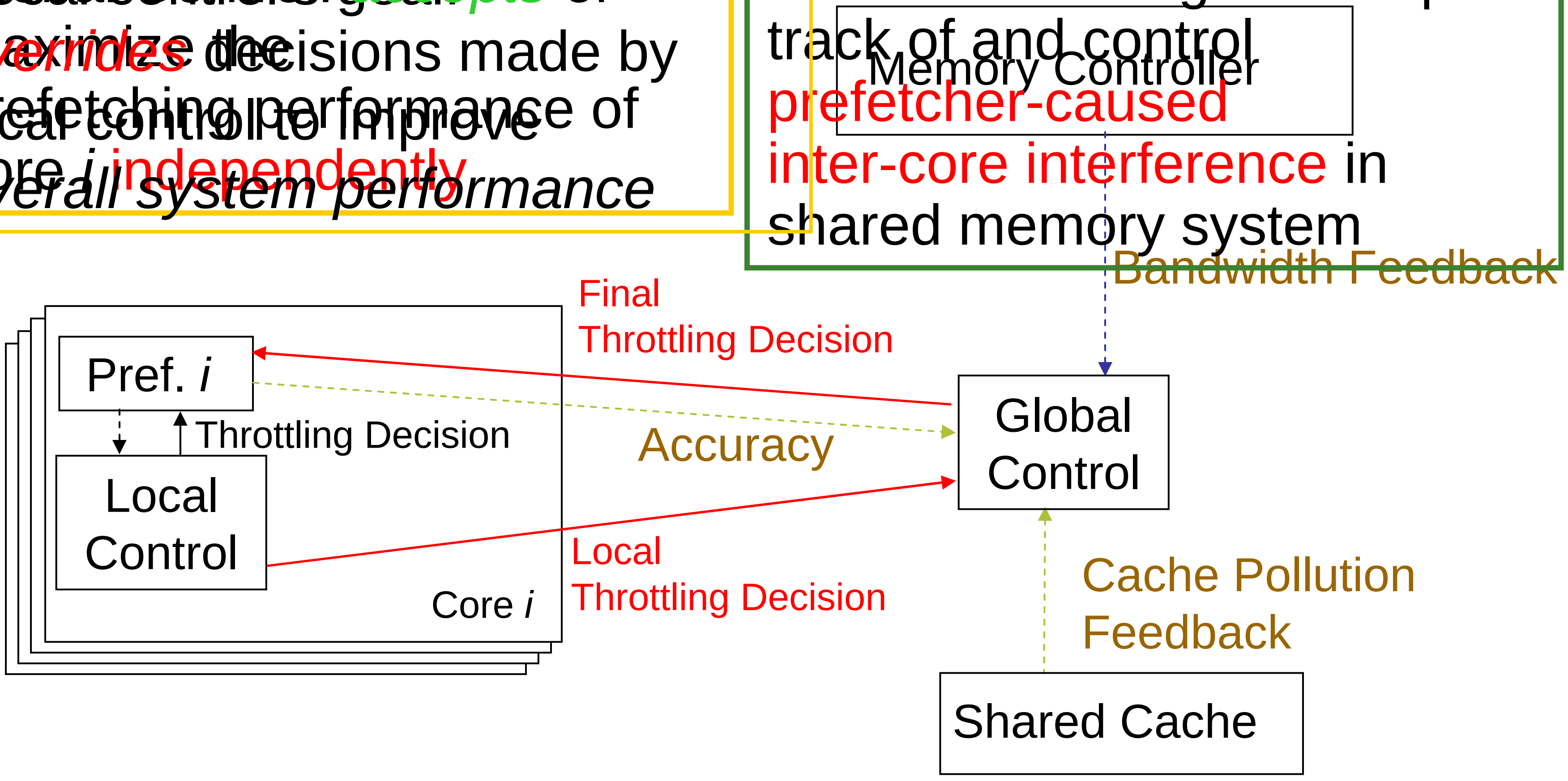
- Ideas for coordinating different prefetchers' actions
  - Utility-based prioritization
    - Prioritize prefetchers that provide the best marginal utility on system performance
  - Cost-benefit analysis
    - Compute cost-benefit of each prefetcher to drive prioritization
  - Heuristic based methods
    - Global controller overrides local controller's throttling decision based on interference and accuracy of prefetchers
    - Ebrahimi et al., "Coordinated Management of Multiple Prefetchers in Multi-Core Systems," MICRO 2009.

Hierarchical Prefetcher Throttling

Local control's goal: Maximize the prefetching performance of core *i* independently

Global Control: *accepts* or *overrides* decisions made by local control to improve *overall system performance*

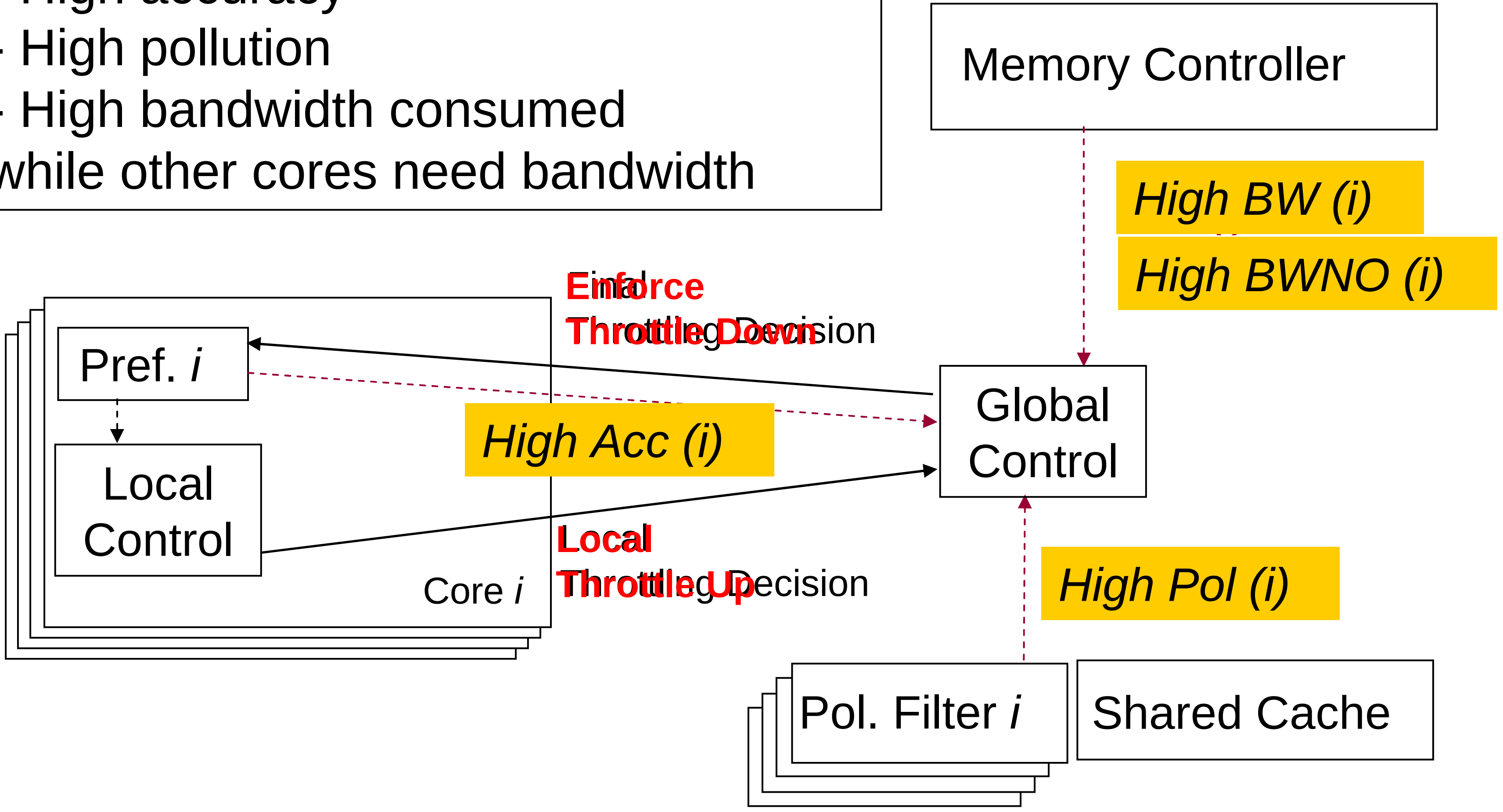
Global control's goal: Keep track of and control prefetcher-caused inter-core interference in shared memory system



Hierarchical Prefetcher Throttling Example

- High accuracy
- High pollution
- High bandwidth consumed

while other cores need bandwidth



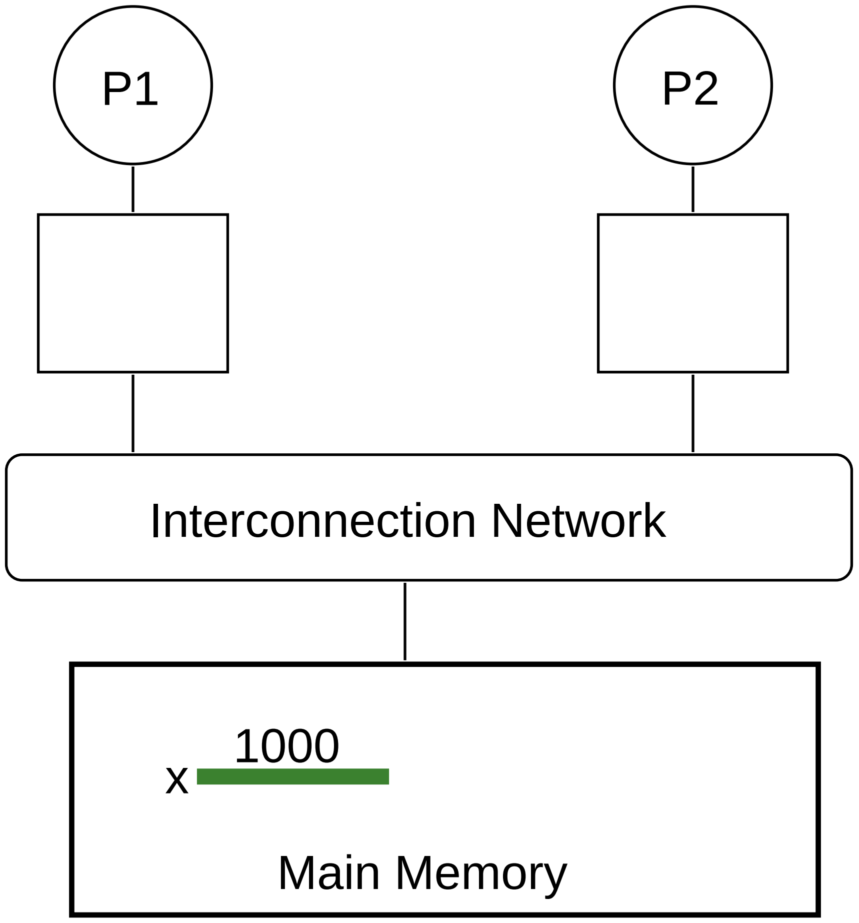
Multi-Core Issues in Caching

- Multi-core
  - More pressure on the memory/cache hierarchy → cache efficiency a lot more important
  - Private versus shared caching
  - Providing fairness/QoS in shared multi-core caches
  - Migration of shared data in private caches
  - How to organize/connect caches:
    - Non-uniform cache access and cache interconnect design

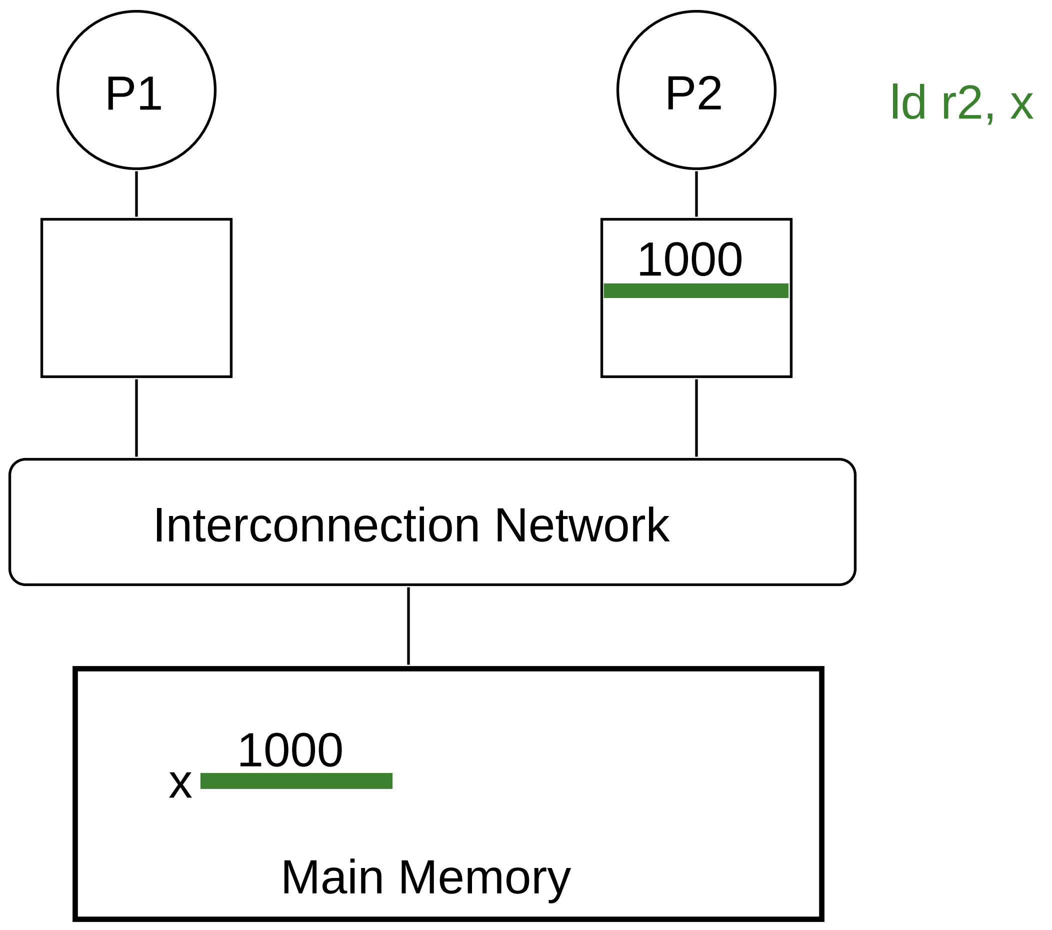
- Placement/insertion
  - Identifying what is most profitable to insert into cache
  - Minimizing dead/useless blocks
- Replacement
  - Cost-aware: which block is most profitable to keep?

Cache Coherence

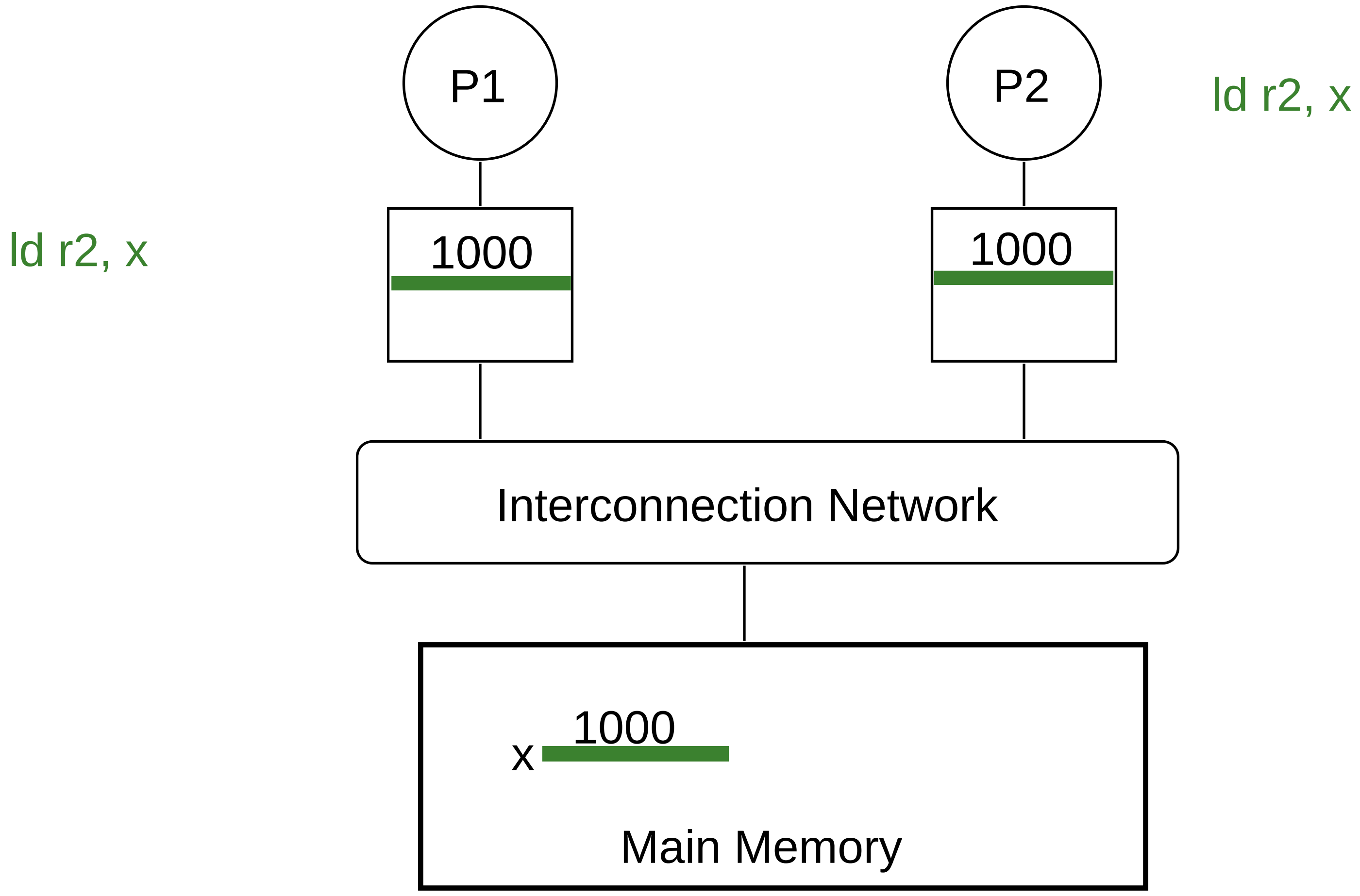
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



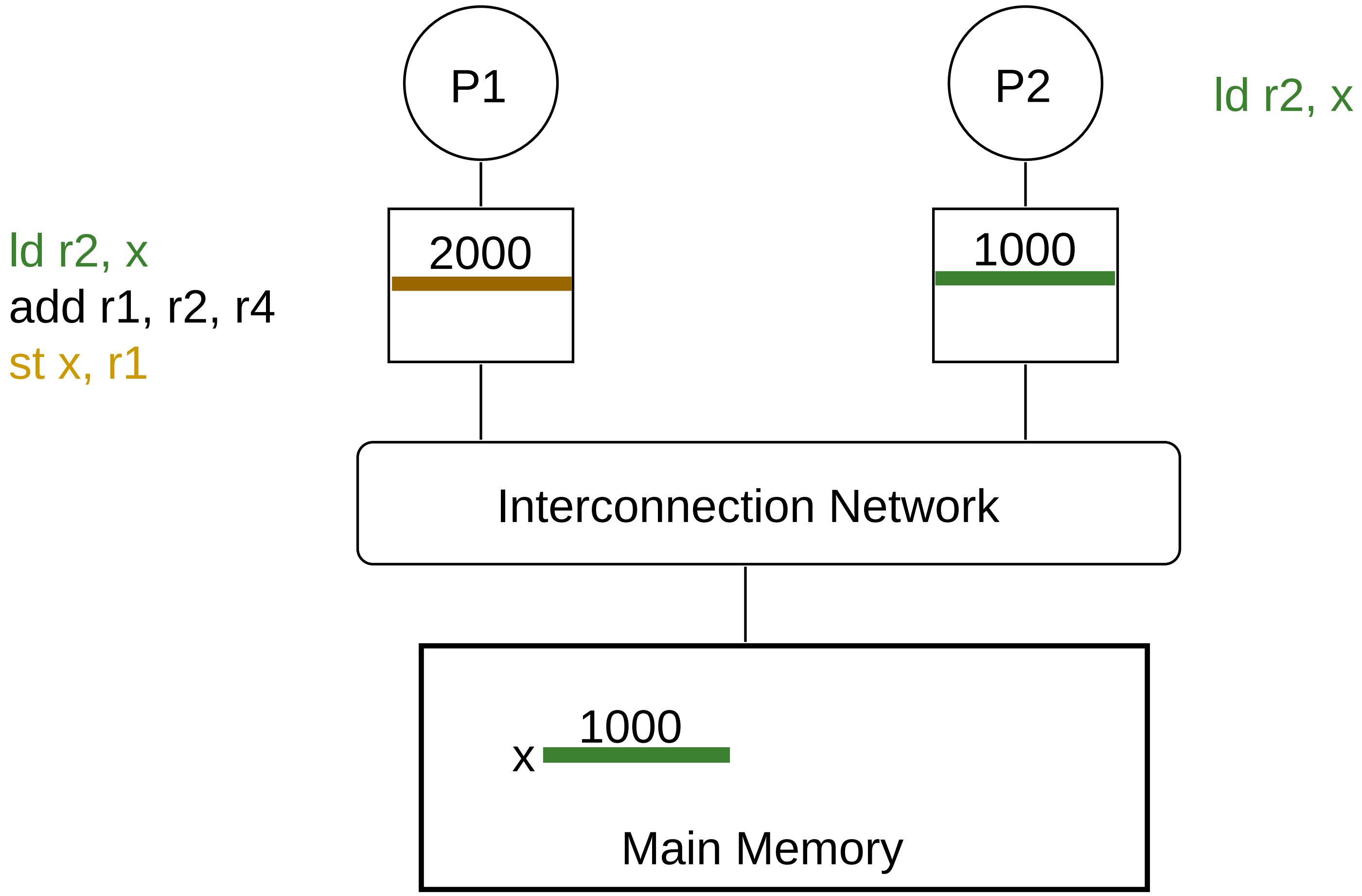
The Cache Coherence Problem



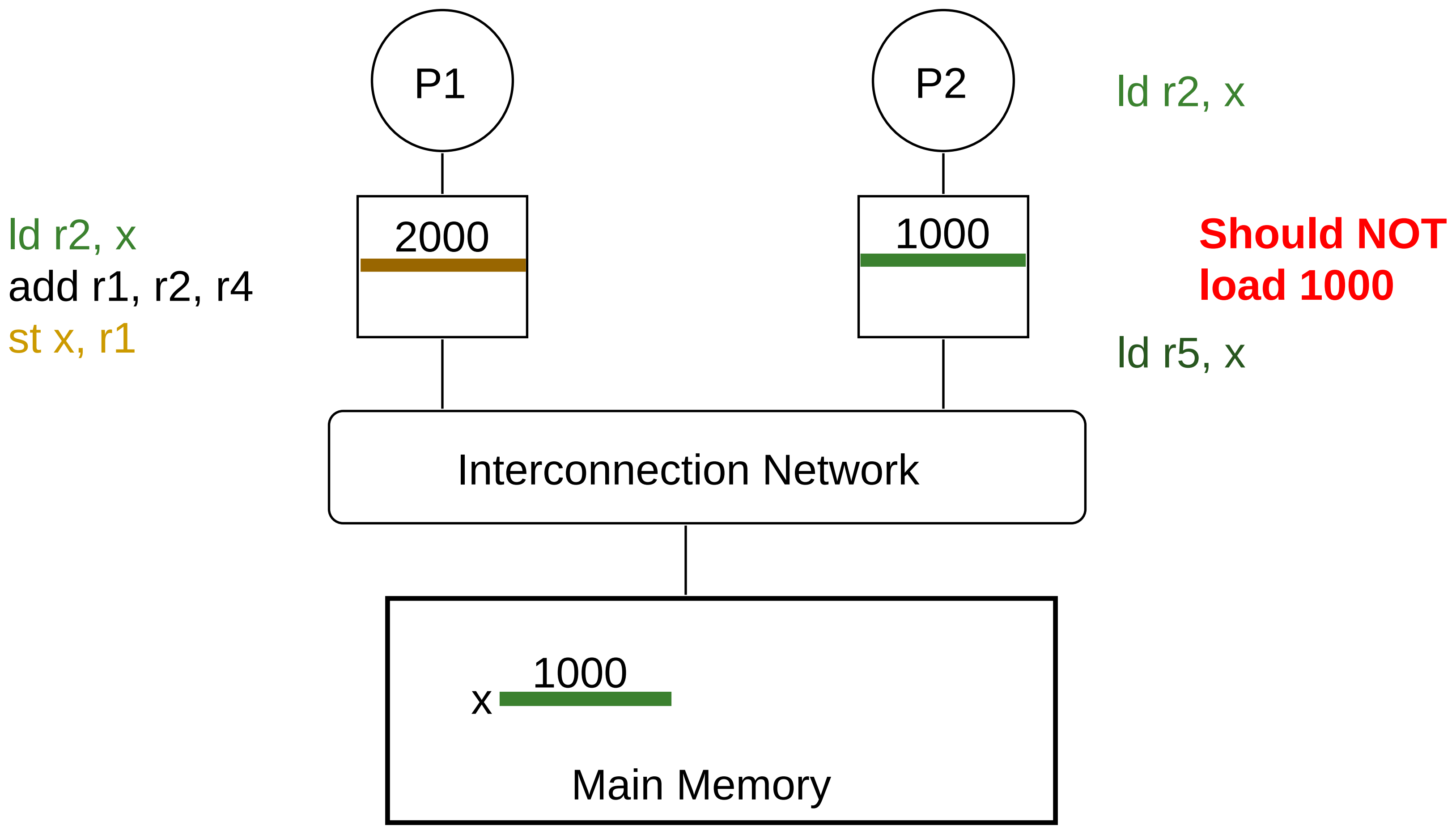
The Cache Coherence Problem



The Cache Coherence Problem



The Cache Coherence Problem



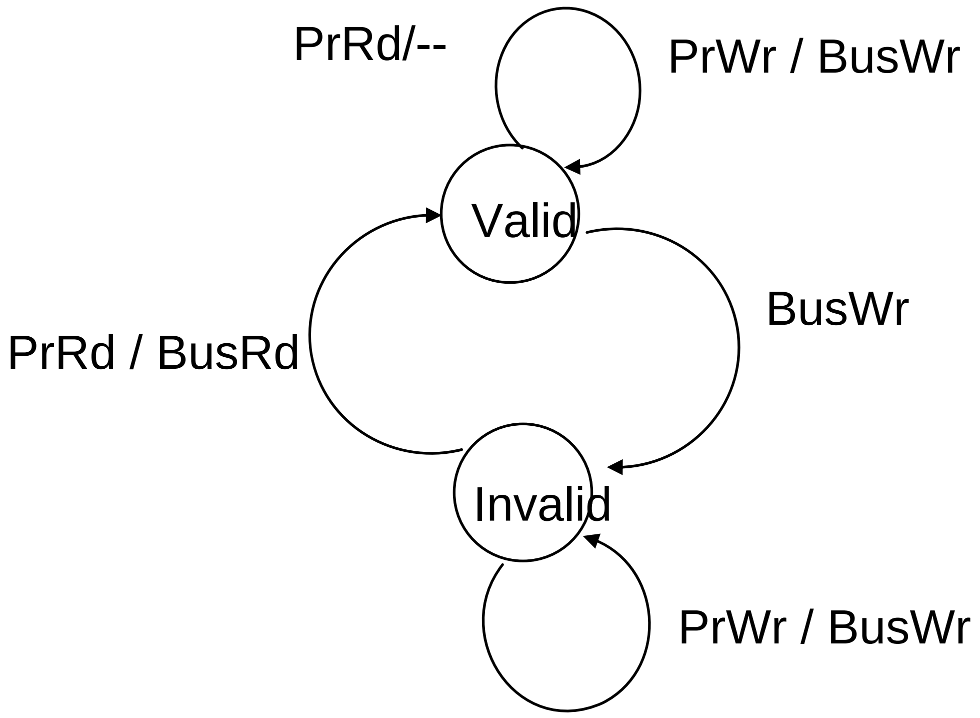
Cache Coherence: Whose Responsibility?

- Software
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided the following instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache
    - When does the programmer need to FLUSH-LOCAL an address?
  - What if the ISA provided the following instruction?
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches
    - When does the programmer need to FLUSH-GLOBAL an address?

- Hardware
  - Simplifies software's job
  - One idea: Invalidate all other copies of block A when a processor writes to it

Snoopy Cache Coherence

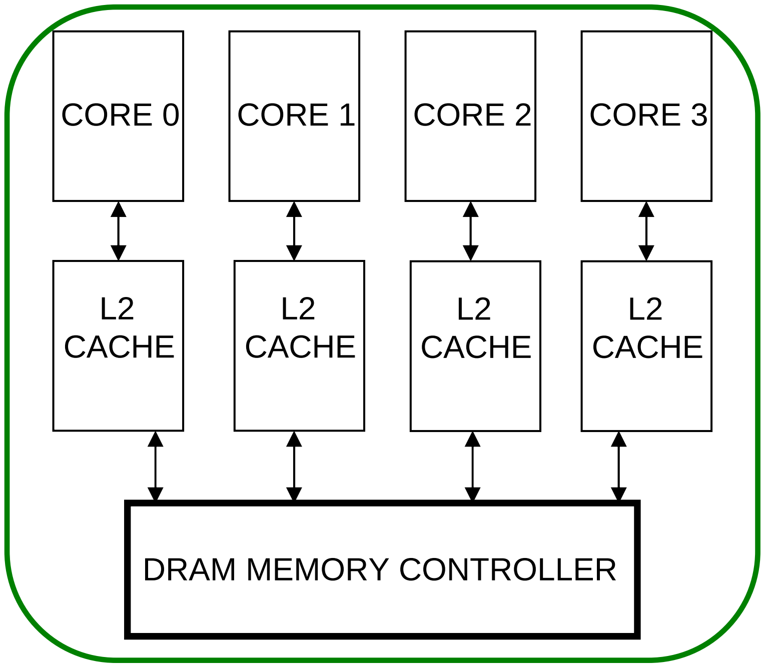
- Caches "snoop" (observe) each other's write/read operations
- A simple protocol:

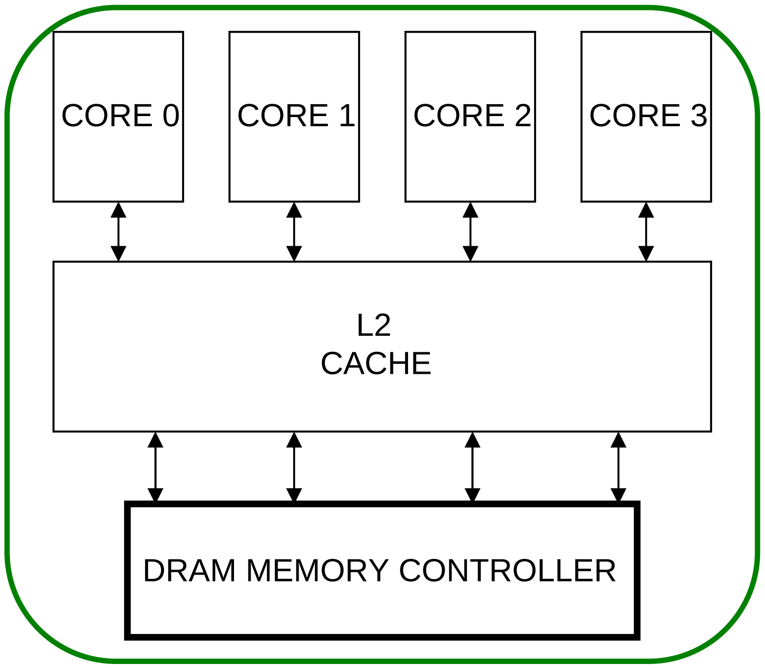


- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

Multi-core Issues in Caching

- How does the cache hierarchy change in a multi-core system?
- Private cache: Cache belongs to one core
- Shared cache: Cache is shared by multiple cores





Shared Caches Between Cores

- Advantages:
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
  - Easier to maintain coherence
  - Shared data and locks do not ping pong between caches

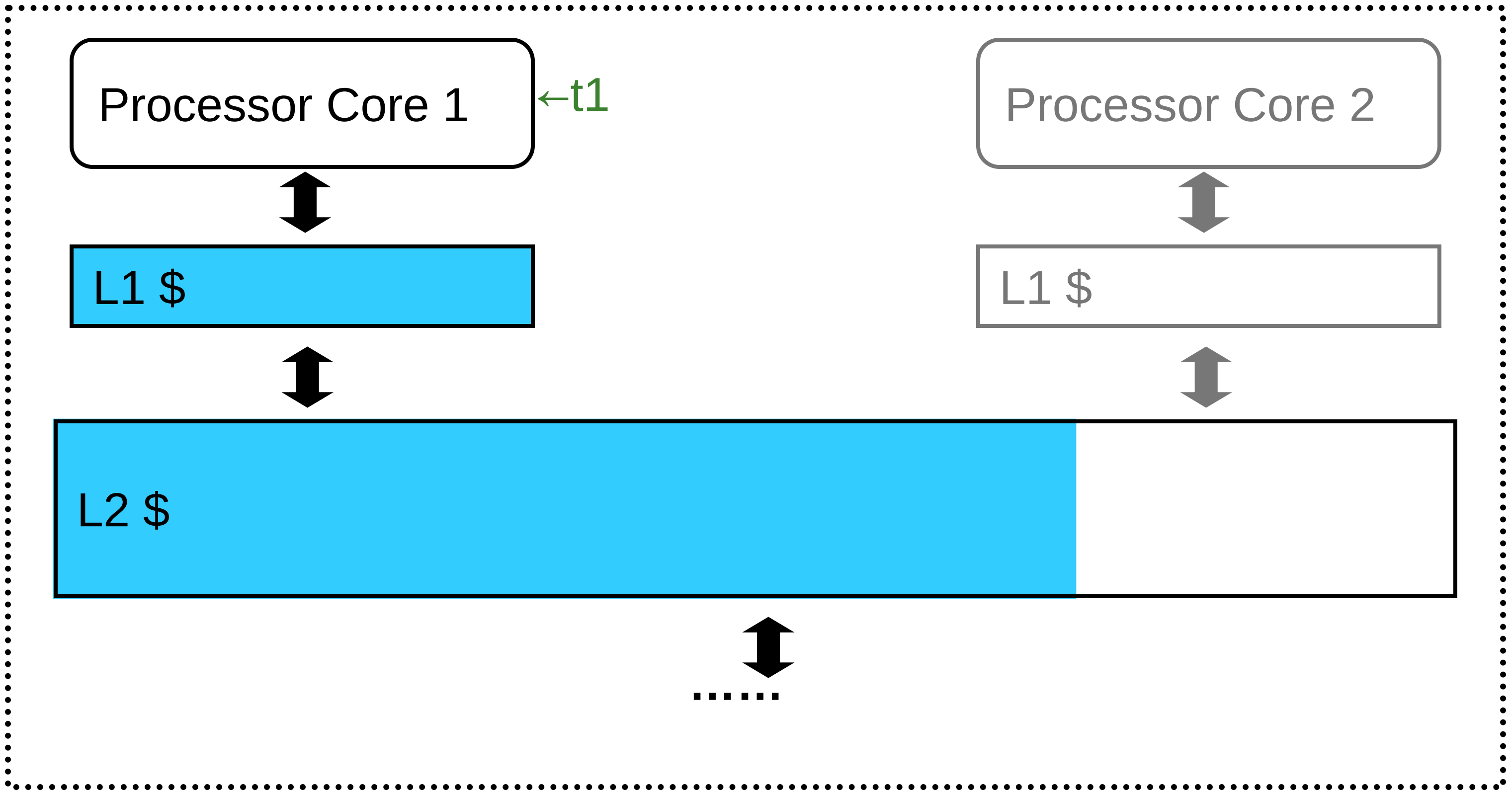
- Disadvantages
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
      - What kind of access patterns could cause this?
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)
  - High bandwidth harder to obtain (N cores → N ports?)

Shared Caches: How to Share?

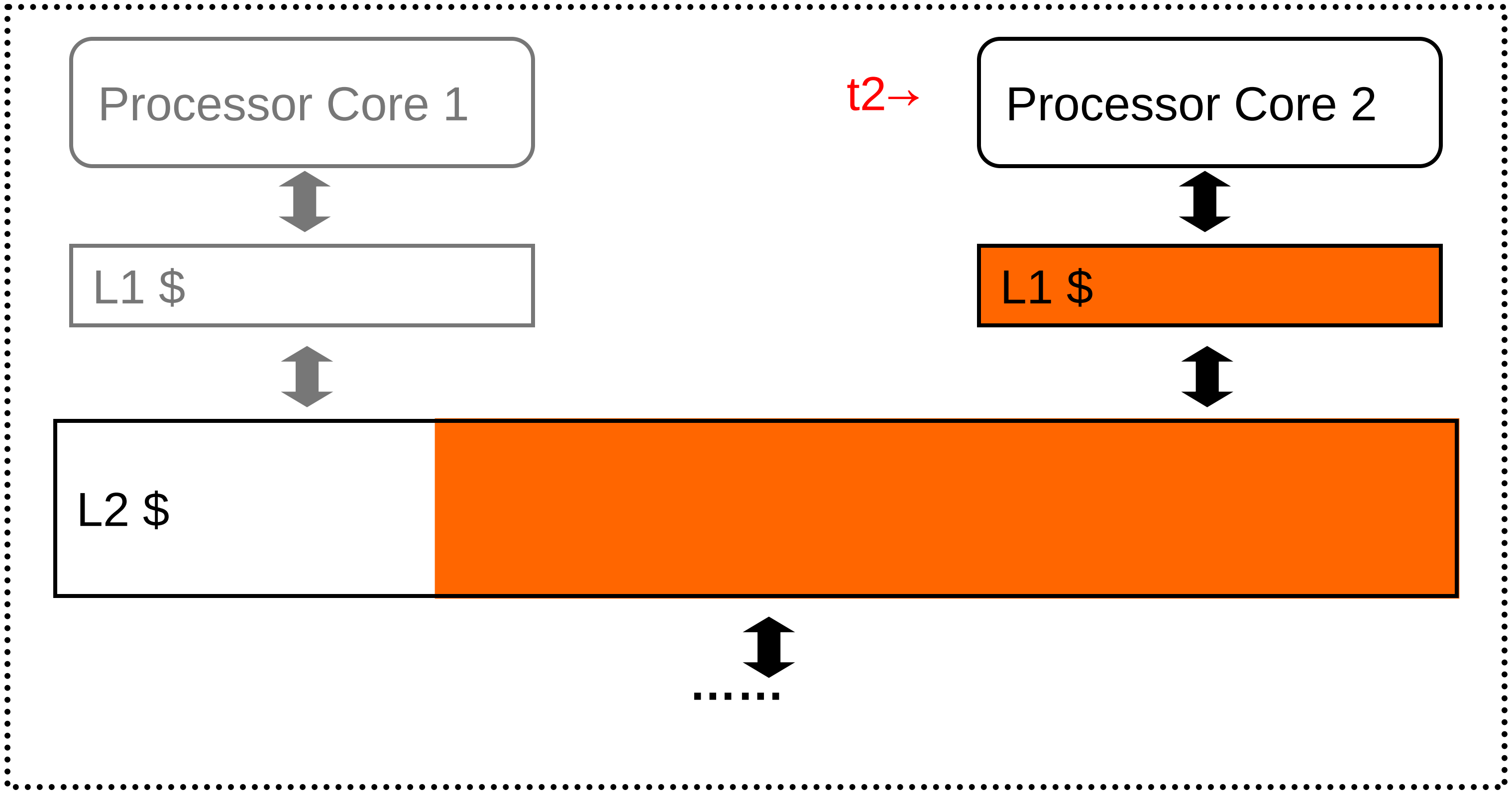
- Free-for-all sharing
  - Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
  - Not thread/application aware
  - An incoming block evicts a block regardless of which threads the blocks belong to

- Problems
  - A cache-unfriendly application can destroy the performance of a cache friendly application
  - Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
  - Reduced performance, reduced fairness

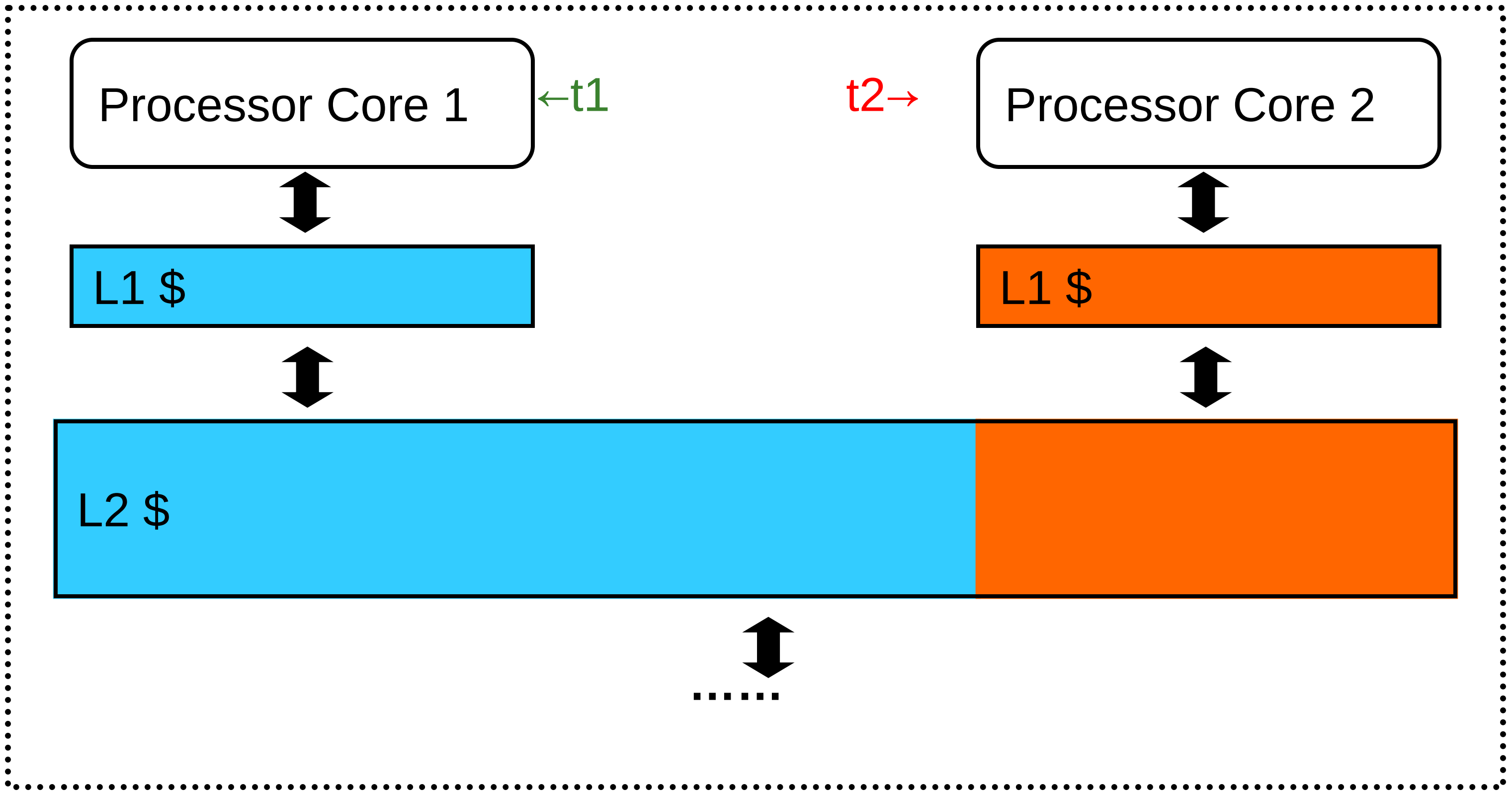
Problem with Shared Caches



Problem with Shared Caches



Problem with Shared Caches



t2's throughput is significantly reduced due to unfair cache sharing.

Controlled Cache Sharing

- Utility based cache partitioning
  - Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.
  - Suh et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA 2002.

- Fair cache partitioning
  - Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

- Shared/private mixed cache mechanisms
  - Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.
  - Hardavellas et al., "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," ISCA 2009.

Utility Based Shared Cache Partitioning

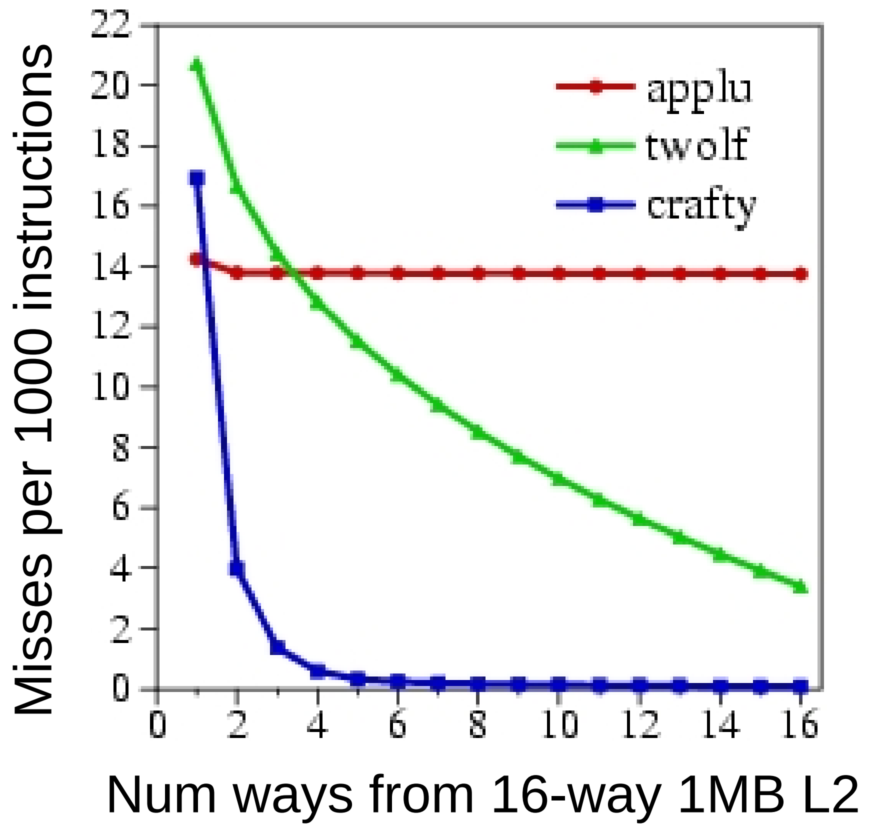
- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

- The high-level idea can be applied to other shared resources as well.

- Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.
- Suh et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA 2002.

Utility Based Cache Partitioning (I)

Utility U_a^b = Misses with a ways – Misses with b ways

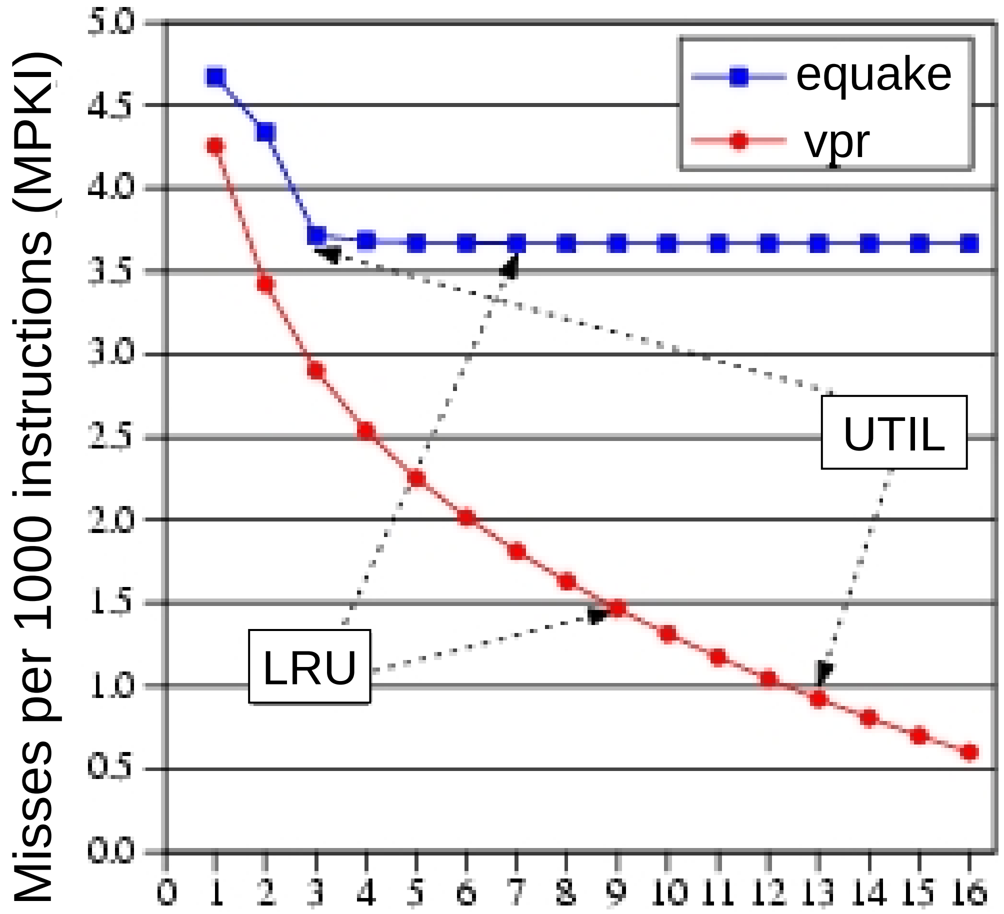


Low Utility

High Utility

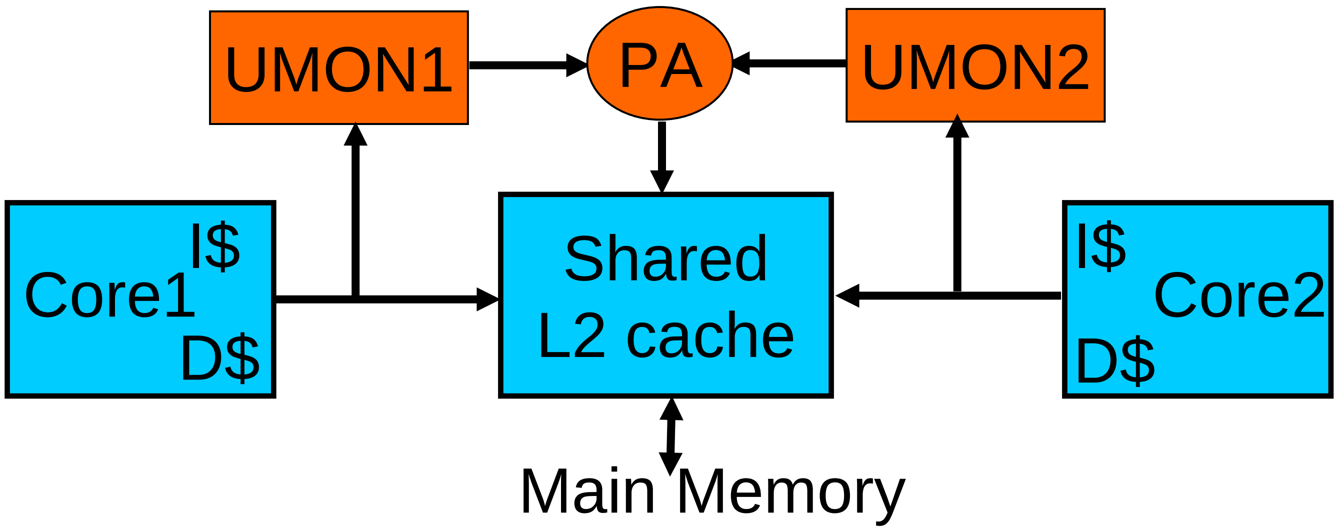
Saturating Utility

Utility Based Cache Partitioning (II)



Idea: Give more cache to the application that benefits more from cache

Utility Based Cache Partitioning (III)

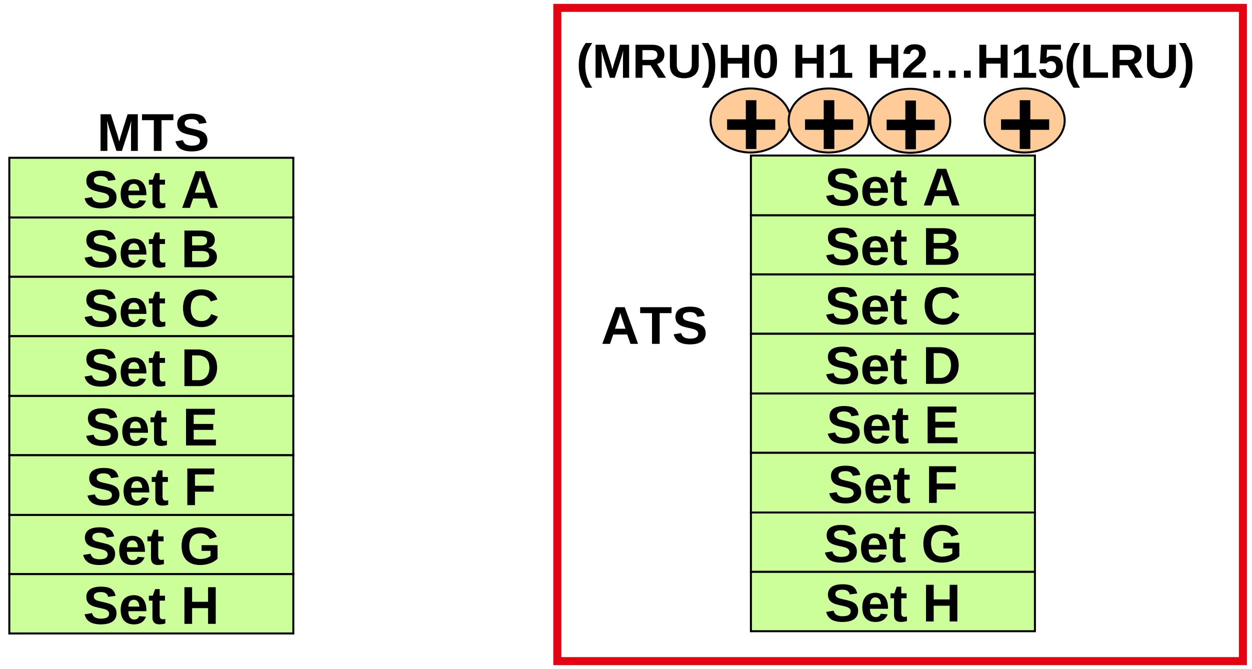


Three components:

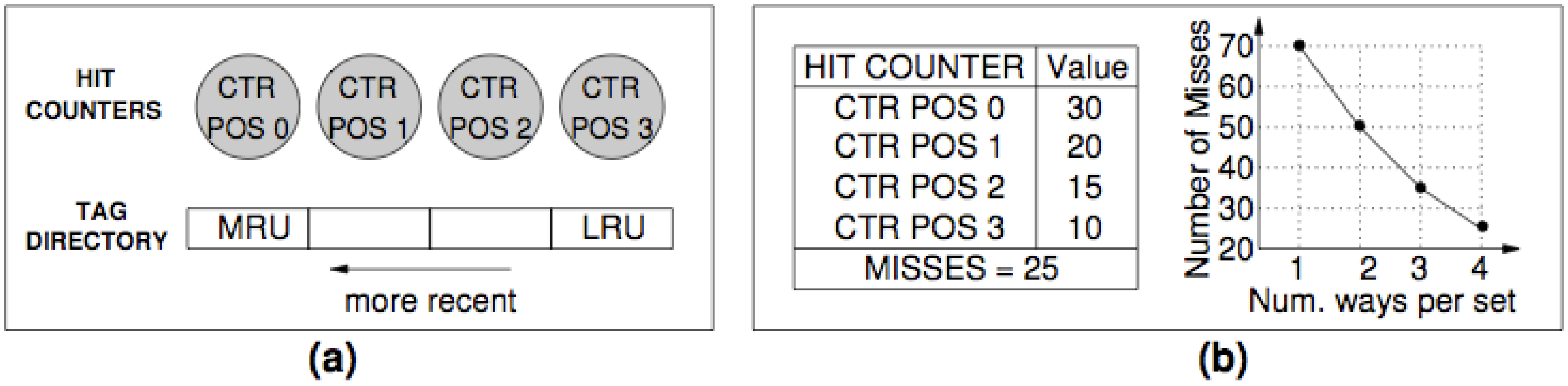
- Utility Monitors (UMON) per core
- Partitioning Algorithm (PA)
- Replacement support to enforce partitions

Utility Monitors

- ❑ For each core, simulate LRU using auxiliary tag store (ATS)
- ❑ Hit counters in ATS to count hits per recency position
- ❑ LRU is a stack algorithm: hit counts ➔ utility
  E.g. hits(2 ways) = H0+H1



Utility Monitors

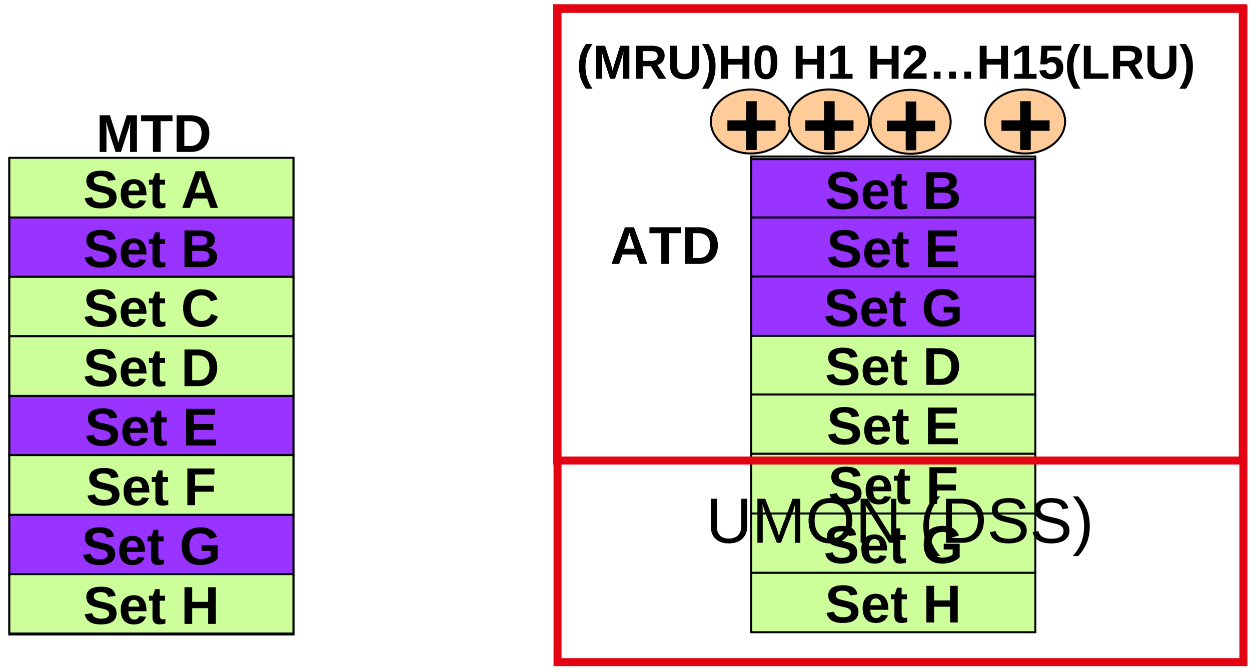


HIT COUNTER	Value
CTR POS 0	30
CTR POS 1	20
CTR POS 2	15
CTR POS 3	10
MISSES = 25	

Figure 4. (a) Hit counters for each recency position. (b) Example of how utility information can be tracked with stack property.

Dynamic Set Sampling

- Extra tags incur hardware and power overhead
- DSS reduces overhead [Qureshi+ ISCA'06]
- 32 sets sufficient (analytical bounds)
- Storage < 2kB/UMON



Partitioning Algorithm

- Evaluate all possible partitions and select the best

- With a ways to core1 and (16-a) ways to core2:

  $Hits_{core1} = (H_0 + H_1 + \ldots + H_{a-1})$ ---- from UMON1

  $Hits_{core2} = (H_0 + H_1 + \ldots + H_{16-a-1})$ ---- from UMON2

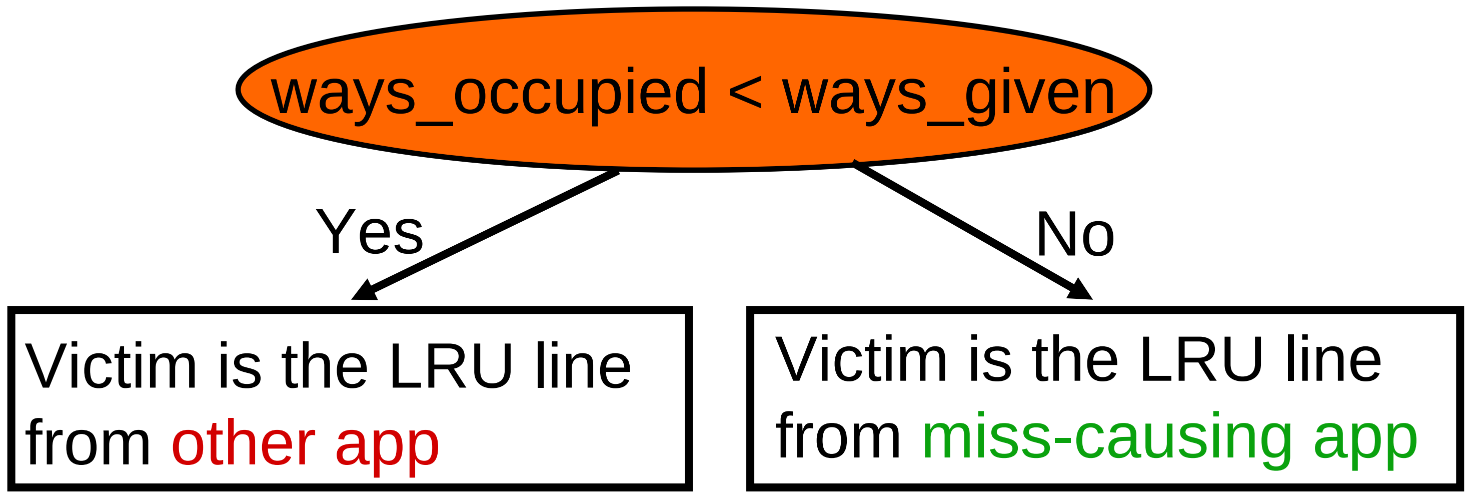
- Select a that maximizes $(Hits_{core1} + Hits_{core2})$

- Partitioning done once every 5 million cycles

Way Partitioning

Way partitioning support:

1. Each line has core-id bits
2. On a miss, count ways_occupied in set by miss-causing app


ways_occupied < ways_given
Yes
No
Victim is the LRU line from other app
Victim is the LRU line from miss-causing app

Performance Metrics

- Three metrics for performance:

1. Weighted Speedup (default metric)

   ➔ perf = $IPC_1/SingleIPC_1 + IPC_2/SingleIPC_2$

   ➔ correlates with reduction in execution time

2. Throughput

   ➔ perf = $IPC_1 + IPC_2$

   ➔ can be unfair to low-IPC application

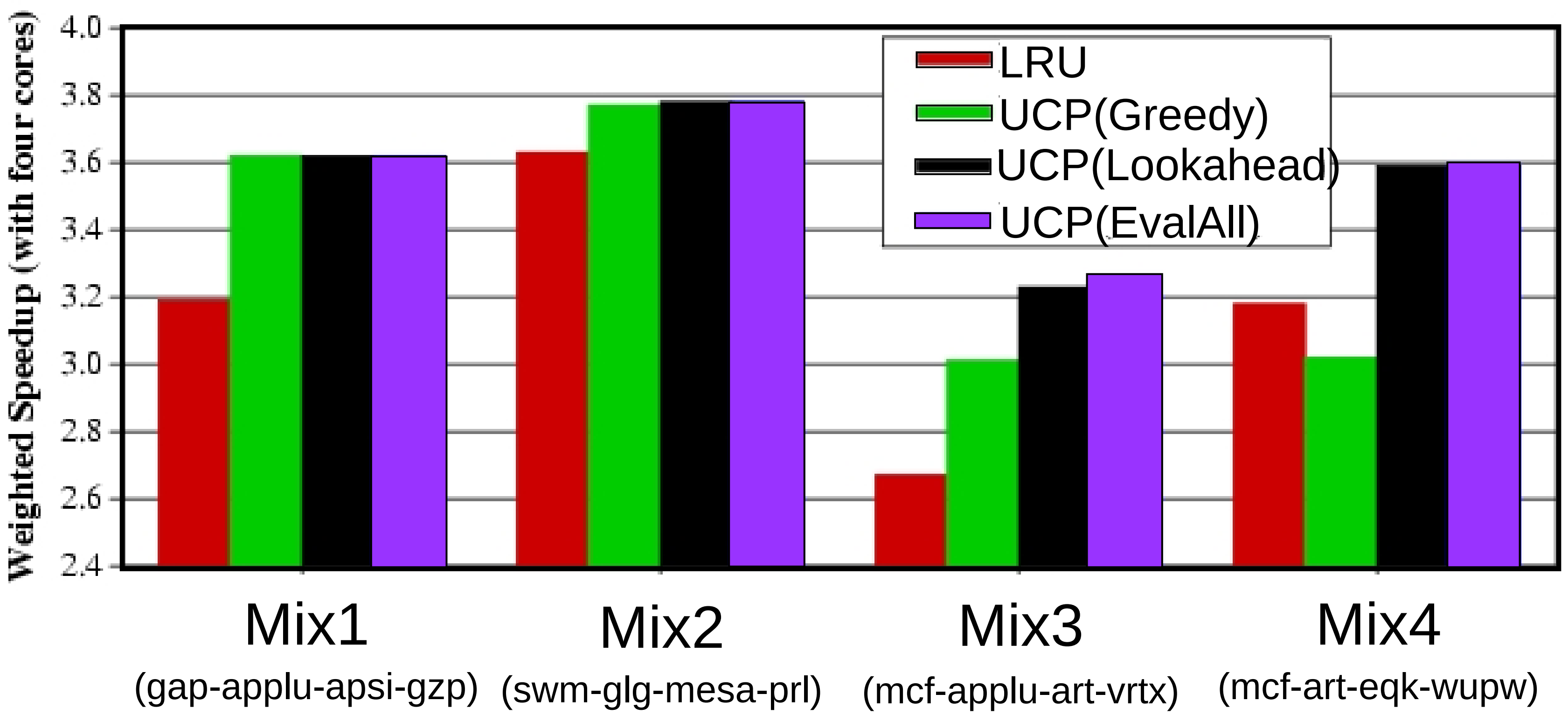
3. Hmean-fairness

   ➔ perf = hmean($IPC_1/SingleIPC_1$, $IPC_2/SingleIPC_2$)

   ➔ balances fairness and performance

Utility Based Cache Partitioning Performance

Four cores sharing a 2MB 32-way L2



Utility Based Cache Partitioning

- Advantages over LRU

  + Better utilizes the shared cache

- Disadvantages/Limitations

  - Scalability: Partitioning limited to ways. What if you have numWays < numApps?

  - Scalability: How is utility computed in a distributed cache?

  - What if past behavior is not a good predictor of utility?