# Lecture 9
# Advanced Multicore Caching

Daniel Sanchez and Joel Emer

[Based on EE382A Material from Kozyrakis & Sanchez]

6.888 Parallel and Heterogeneous Computer Architecture
Spring 2013

Massachusetts Institute of Technology

CSAIL

# Administrivia



- Project proposal due next week
  - 2-3 pages
  - Idea, motivation, expected results

# Caches? Again?



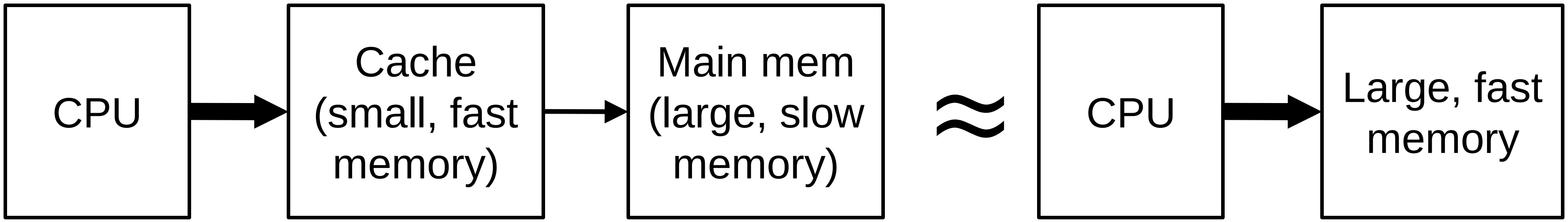
| | 45nm | 11nm | |
|---|---|---|---|
| 16b integer multiply | 2 pJ | 0.4 pJ | |
| 64b FP multiply-add | 50 pJ | 8 pJ | Compute |
| 64b read, 8KB SRAM | 14 pJ | 2 pJ | |
| 256b read, 1MB SRAM | 566 pJ | 94 pJ | |
| 256b 10nm wire | 310 pJ | 174 pJ | Memory |
| 256b DRAM interface | 5,120 pJ | 512 pJ | |
| 256b read DRAM | 2,048 pJ | 640 pJ | |

- Caches set performance and power of multi-core chips
  - Why?
- Caches take ~50% of multi-core chips
- Our focus today: last-level caches (LLC)

# Motivations for Caching



- Main benefit in uniprocessors
  - Reduce average memory access time (latency)




- Additional crucial benefits in CMPs
  - Memory bandwidth amplification
  - Energy efficiency
  - Faster inter-thread communication

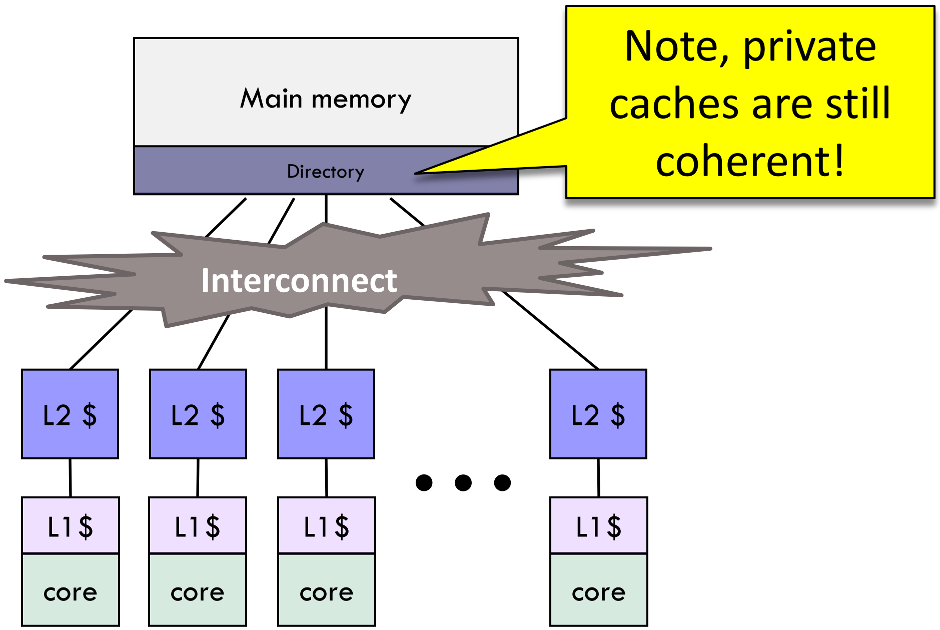
# Outline

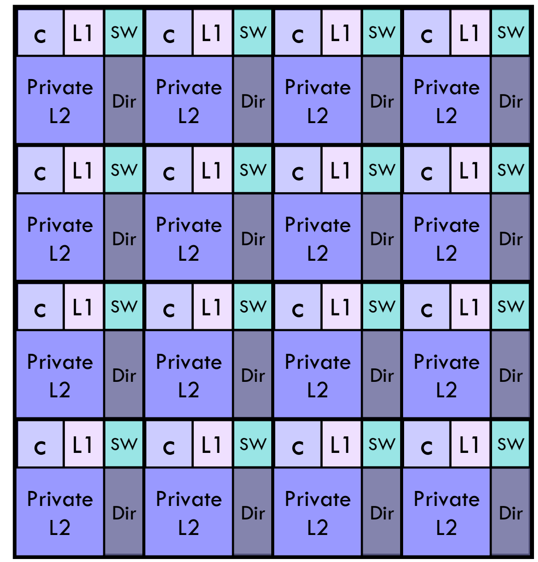


- Shared vs private CMP caches
- Addressing CMP caching issues
  - High access latency [shared]: placement, migration, replication
  - Lost capacity [private]: controlled replication
  - Interference [shared]: cache partitioning, replacement policies for shared caches
  - Underutilization [private]: capacity sharing

# Private Caches



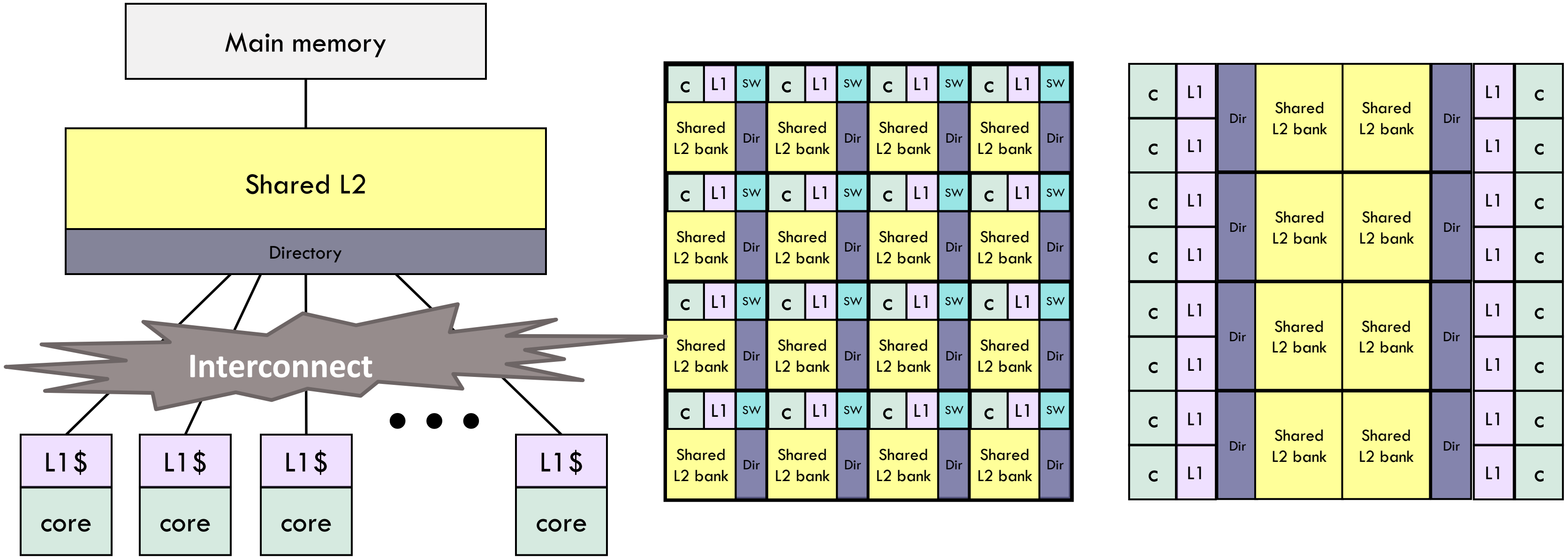



- ✓ Low access latency
- ✓ Isolation (capacity, bandwidth)
- ✓ Lower bandwidth interconnect
- ✗ Underutilization of resources (capacity, replicated data)
- ✗ Expensive coherence, slow inter-core communication

# Shared Caches



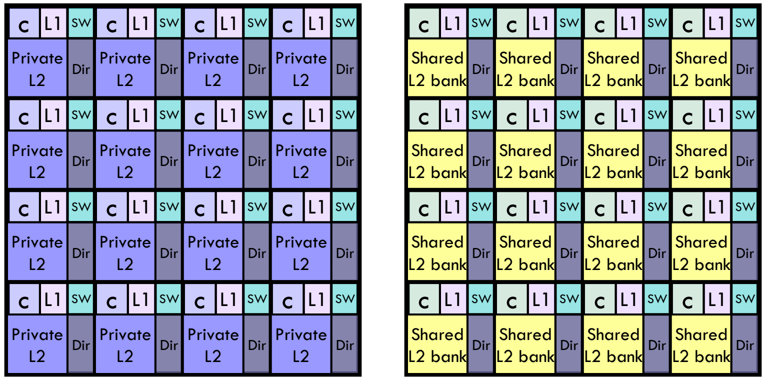



- ✓ Resource sharing (capacity, bandwidth)
- ✓ Cheaper coherence, fast inter-core communication
- × High L2 avg. access latency
- × Requires high-bandwidth interconnect
- × Destructive interference (capacity)

# Notes



- Can also have hybrid models (hierarchical cache)
  - E.g., parts of the LLC shared between a group of cores
- Note difference between logical and physical origination
  - E.g., shared cache with private-like chip layout
  - Notice anything interesting with this distributed way of implementing shared caches?



# Shared/Private Pros & Cons



| | Private | Shared |
|---|---|---|
| Access latency | Low | High |
| Duplication of read-shared data | Yes | No |
| Destructive interference | No | Yes |
| Resource underutilization | Yes | No |
| Interconnect bandwidth | Low | High |
| Coherence & communication cost | High | Low |

# Addressing Limitations



- Shared cache limitations
  - High latency: line placement, migration, and replication
  - Interference: controlled sharing
- Private cache limitations
  - Duplication of shared data: controlled replication
  - Underutilization: capacity stealing

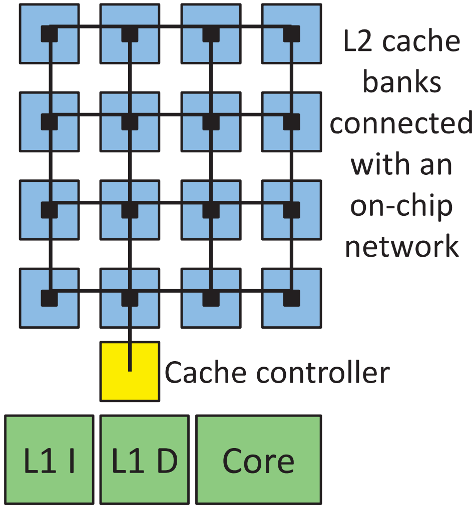
# Shared Caches: Latency Reduction Techniques



- Placement: make line→bank mapping flexible
  - Normally, line address determines bank
  - Instead, cache line in bank close to cores that use it
- Migration: move cache lines to close banks
  - Adapts to changing access patterns
  - Power-hungry, has pathological behavior
- Replication: enable multiple copies (replicas) of frequently-accessed read-shared lines
  - Lower access latency
  - Reduces total capacity

# NUCA: Non-Uniform Cache Access

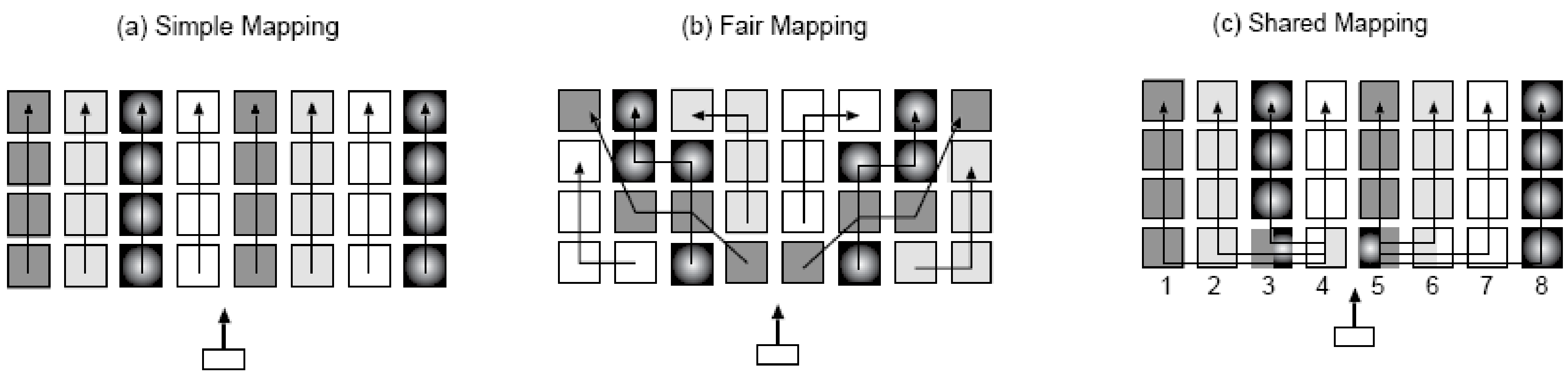





- Idea: accept & manage differences in access latencies
  - Some banks are closer than other
- From static to dynamic placement
  - Static: address bits determine bank
  - Dynamic: allow lines to migrate
- Hopefully, important data are mostly in the nearby banks

# NUCA Management

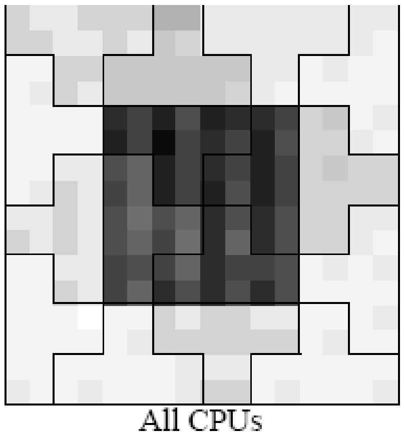





- Approach: organize cache banks into *bank sets*
  - Bank group determined by address bits
  - Banks within the group provide cache associativity
    - Need to look in all the banks in bank group
  - Cache lines can move within a group to get closer to requesting CPU
    - Works because of LRU, most hits normally happen to first cache ways
- Mechanisms: mapping, searching, migration
  - Mapping: simple, fair, shared
  - Searching: incremental, multicast, smart
  - Migration: data moves closer as it is accessed, evicted data moved further

# NUCA & Multi-core

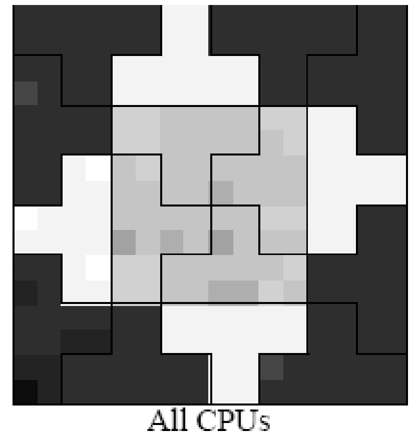





Dark → more accesses

←OLTP (on-line transaction processing)

Ocean → (scientific code)



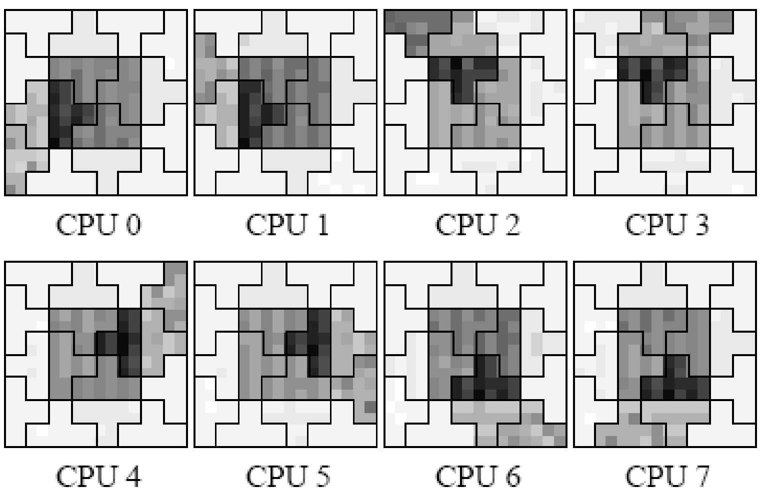




**Figure 10. oltp L2 Hit Distribution**

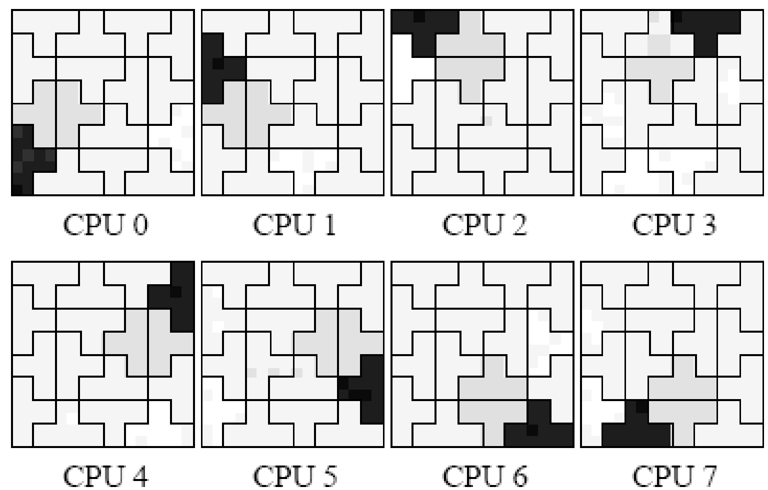



**Figure 11. ocean L2 Hit Distribution**

# NUCA Discussion & Ideas



- What are the complication of dynamic NUCA?

- Ideas for improvements
  - Centralized tags but distributed data
  - Prediction of bank search

- See syllabus for additional refs

# Victim Replication



- Idea: use local L2 bank as victim cache
  - Each line has a single home L2 bank
  - When evicting from L1, write data in local L2 bank
  - Victim can evict invalid lines, replicas and unshared lines
  - Can't evict actively shared blocks that have local L2 as home
- Implementation: simple modifications to shared L2
  - On a miss, search local L2 slice before remote L2 slices
  - Directory or banking structure does not change
    - Victim does not change sharer's info (still as if in local L1)
    - Invalidations need to check both L1 and local L2 bank
- Pros/cons over shared and private?

# Adaptive Selective Replication



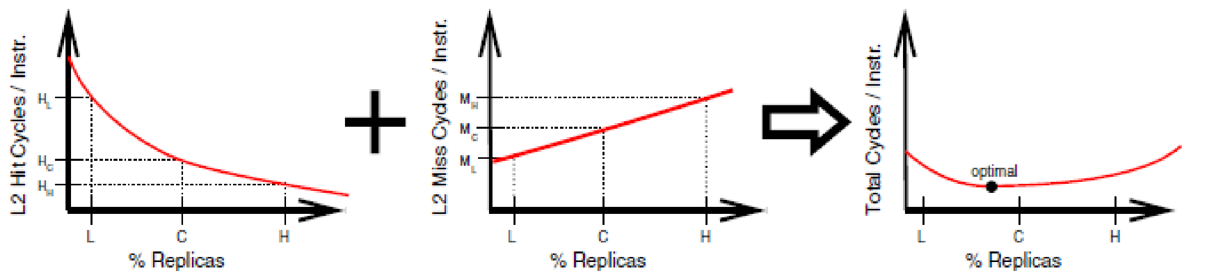



Figure 4: a) Replication Benefit — Figure 4: b) Replication Cost — Figure 4: c) Replication Effectiveness

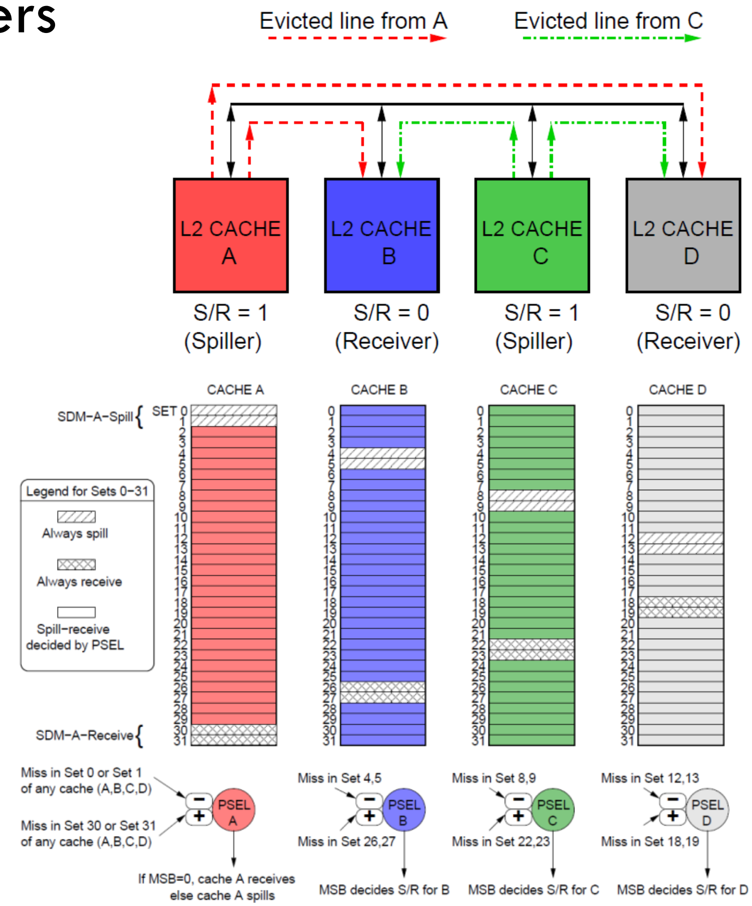
- Private caches always replicate, lose capacity
- Idea: cost/benefit analysis to decide how much to replicate
  - Benefit: faster hits on replicas
  - Cost: more misses due to lost capacity
- Implementation:
  - Choose to keep block or not in L1 eviction probabilistically
  - Adapt replication probability
  - Small victim tag buffer to profile extra misses
  - Count hits on replicas to estimate gains on hit latency

Very useful profiling approach

# Capacity sharing: Dynamic Spill-Receive

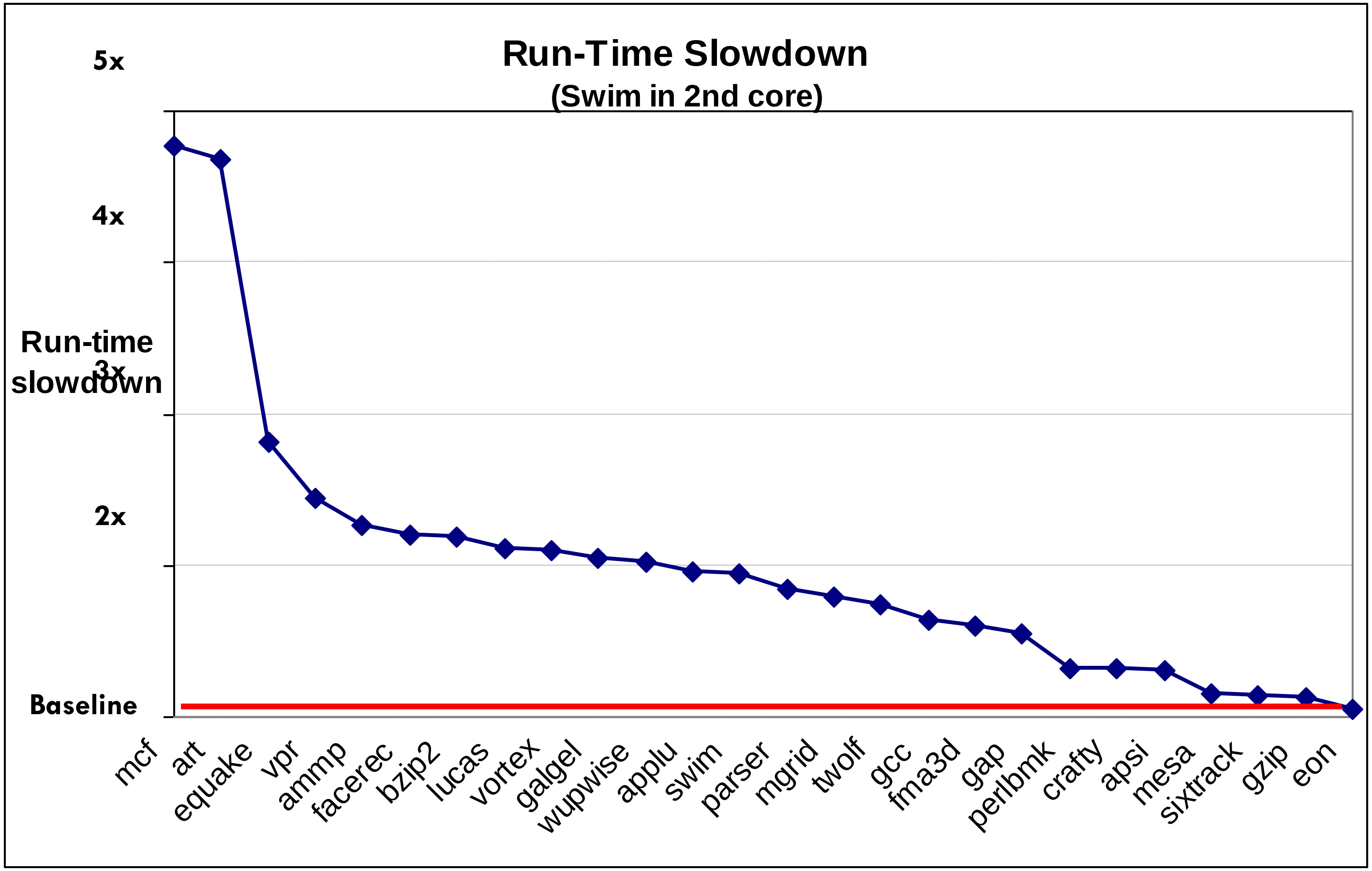


- Capacity sharing by spilling evicted lines to nearby L2s
  - Caches can be spillers or receivers
  - Spilled lines served using cache coherence
- Implementation:
  - Dedicate a few sets in each cache to always-spill or always-receive, measure which one works best



# Example of Cache Interference

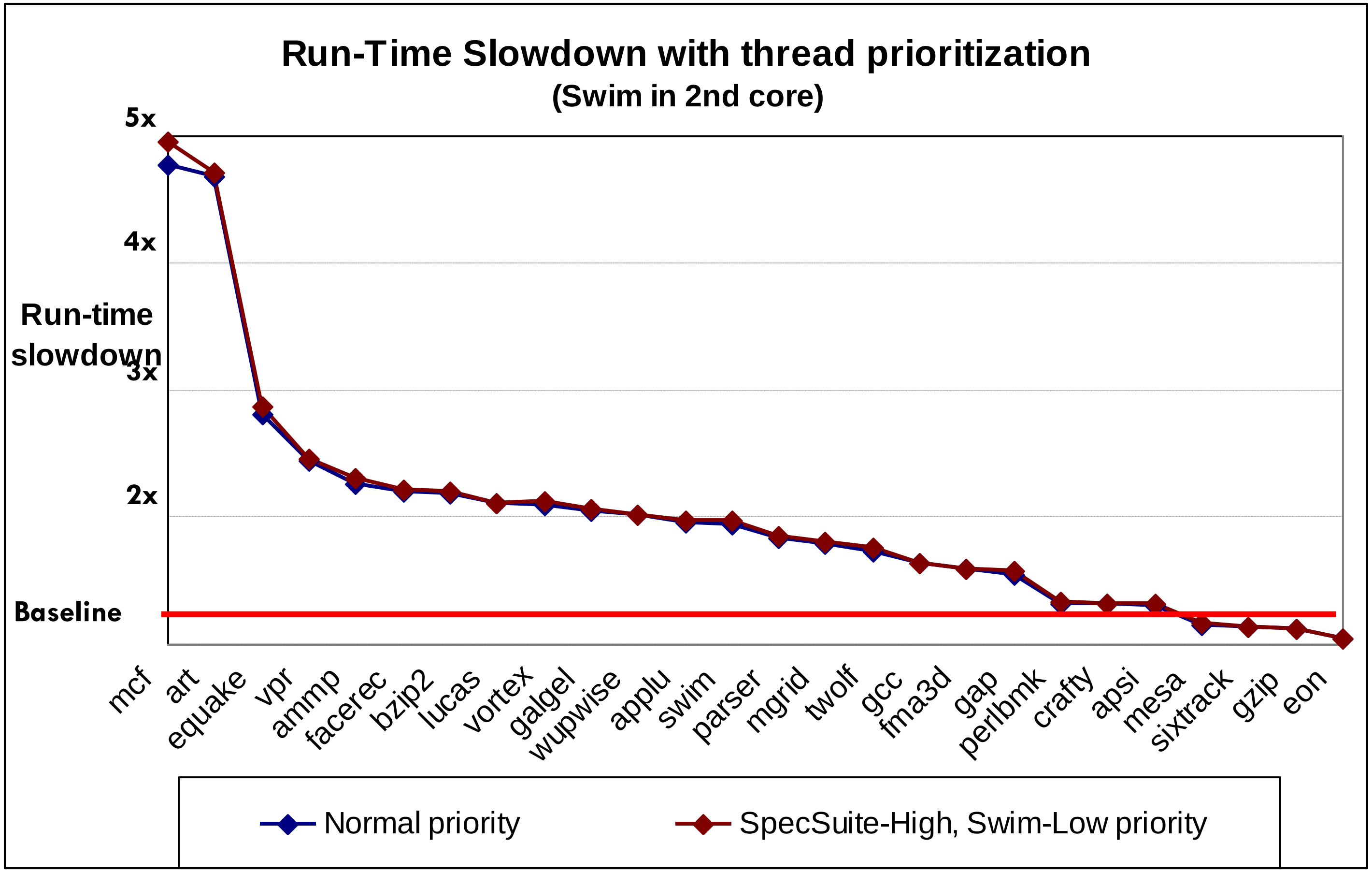





- Slowdown for SPECCPU2000 apps when running in parallel with swim, sharing the L2 cache

# Can OS Priorities Solve the Problem?

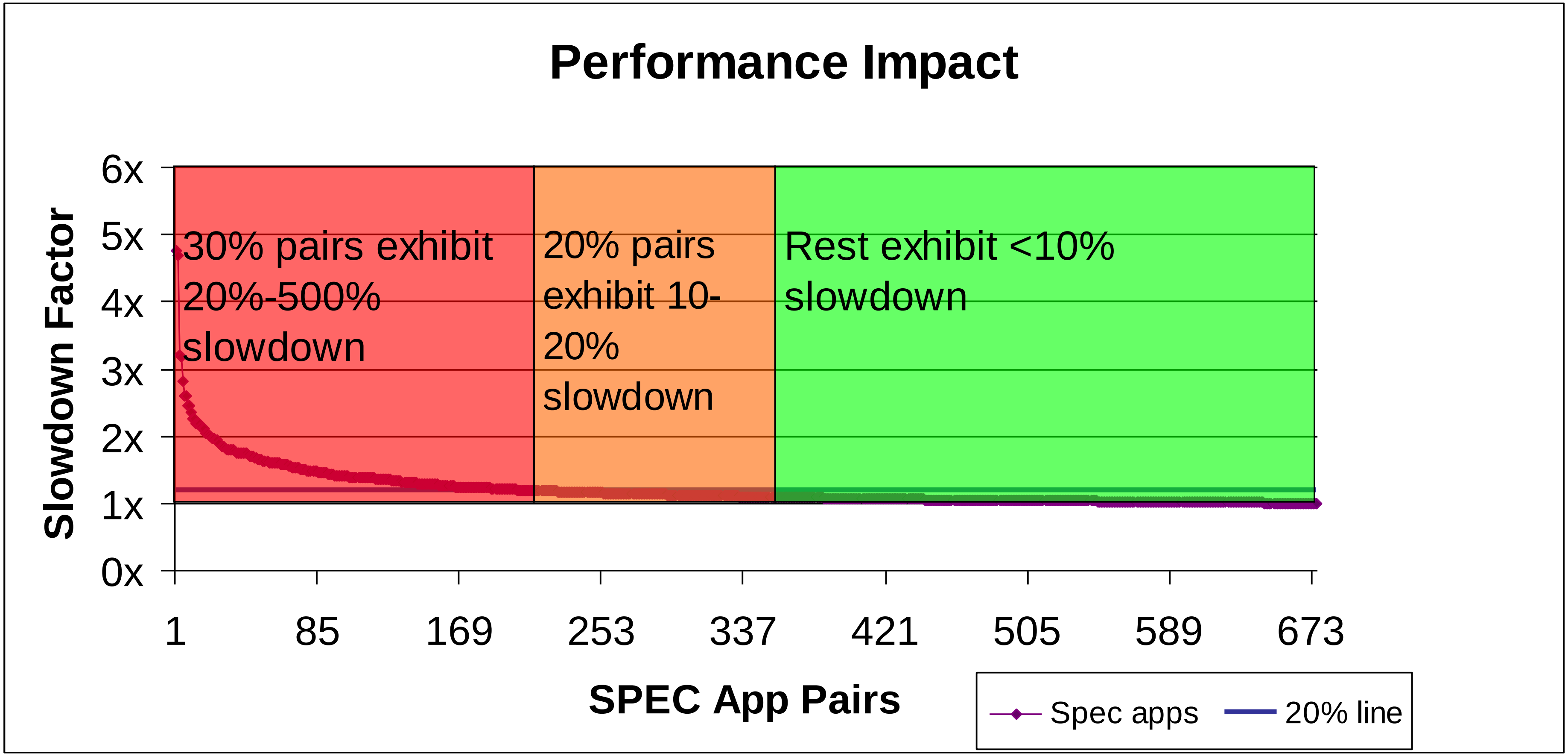





- What is the problem with OS priority mechanisms?

# Is Interference a Common Problem?






- Need mechanisms for isolation & QoS

# Isolation via Cache Partitioning



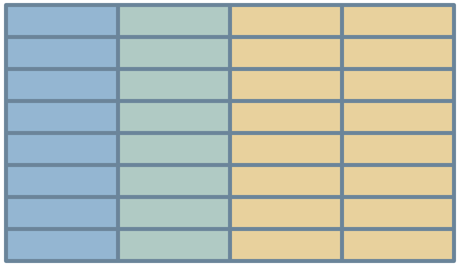
- Idea: eliminate interference by partitioning the capacity of the cache
  - Different apps and different uses get their own partition

- We need two techniques
  - A policy to assign the capacities to cores
  - A mechanism to enforce capacity assignments

# Enforcing Allocations

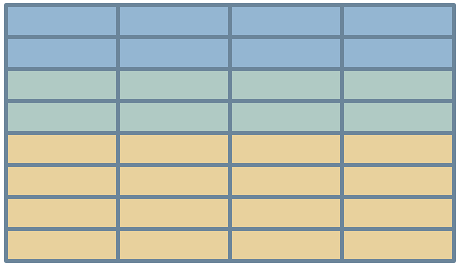


- Way partitioning: Restrict evictions/fills to specific ways
  - How many partitions can we have?
  - What happens with associativity?



- Can we partition the cache by sets?
  - Issues and challenges?

- Any other schemes?



# Capacity Management Policies

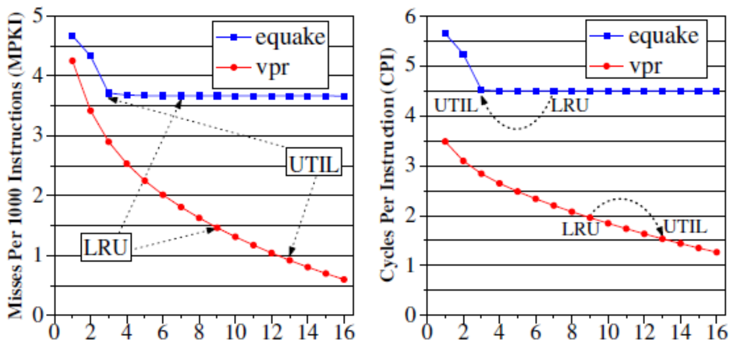


- Capitalist (most systems today)
  - No management
  - If you can generate the requests, you take over resources
- Communist
  - Equal distribution of resources across all apps
  - Guarantees fairness but not best utilization
- Elitist
  - Highest prio for one app through biased resource allocation
  - Best effort for the rest of the apps
- Utilitarian
  - Focus on overall efficiency (e.g., throughput)
  - Provide resources to whoever needs it the most

# Utility-based Cache Partitioning



- Idea: assign capacity to apps based on how well they use it
  - Maximize reduction in number of misses)




- Implementation: find utility of using each way
  - Naïve: one auxiliary set of L2 tags per core, count hits/way
  - Dynamic set sampling: simulate a small number of sets

# Replacement policies for CMPs



- Replacement policy keeps a rank of blocks
  - Select least desirable candidate on an eviction
  - Control how to change the block's rank on an insertion or hit (promotion)
- LRU
  - Select last line in LRU chain for eviction
  - Put block in head of chain (MRU) on ins/promotion
  - Does not work well with streaming/scanning applications (many lines w/o reuse) or under thrashing (working set > size of cache)

# Replacement Policies: DIP



- LRU insertion policy (LIP)
  - Insert in LRU position, promote to MRU → scan-resistance
- Bimodal insertion policy (BIP)
  - Randomly insert few lines at MRU, others LRU → thrash-resistance
- Dynamic insertion policy (DIP)
  - Profile and choose between LRU and DIP
  - Achieves good performance on LRU-friendly workloads
- Thread-aware DIP
  - Select between DIP and LRU per thread
  - Scanning/thrashing/low-utility applications use BIP, get less effective capacity → similar effects as UCP
- S/D/TAD-RRIP, SHiP, …