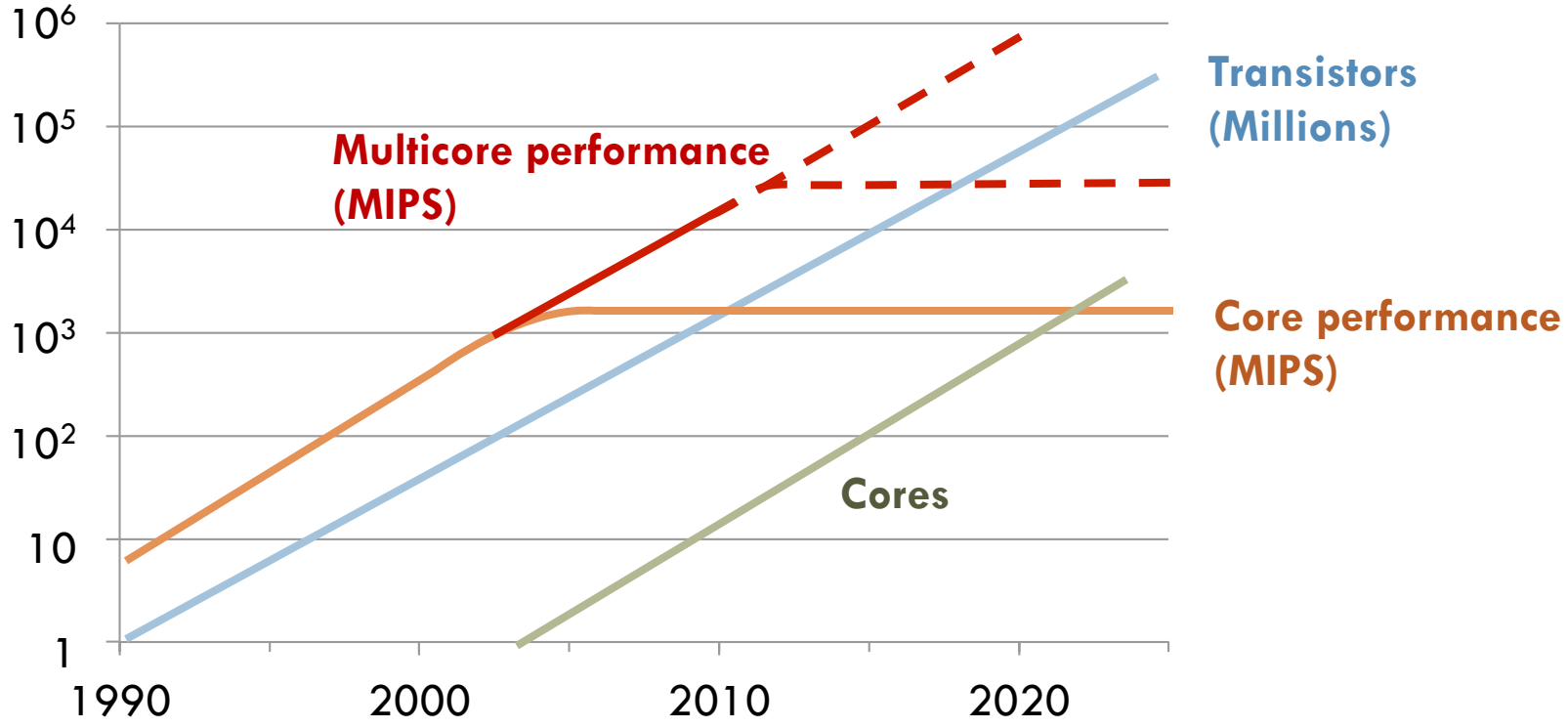
Scaling Hardware and Software for Thousand-Core Systems

Daniel Sanchez

Electrical Engineering
Stanford University

Multicore Scalability

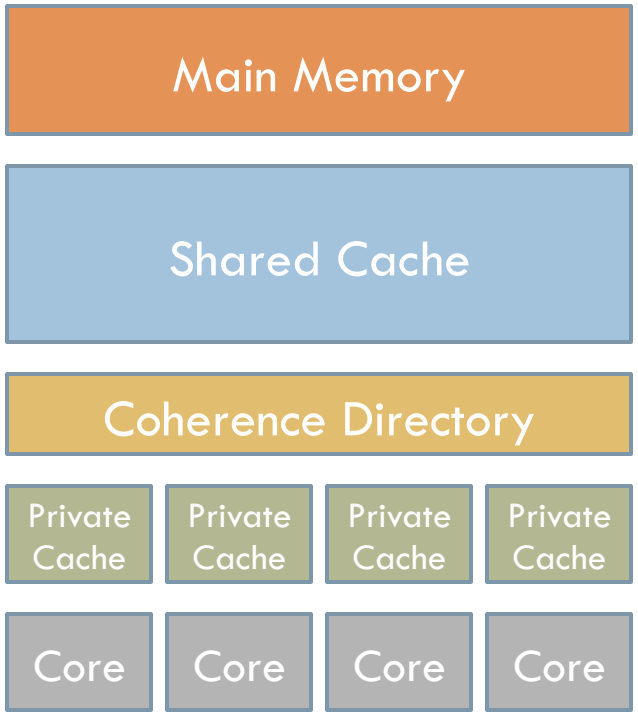


- Multicore is key to future of computing
- Scaling performance is hard, even with a lot of parallelism

Memory is Critical

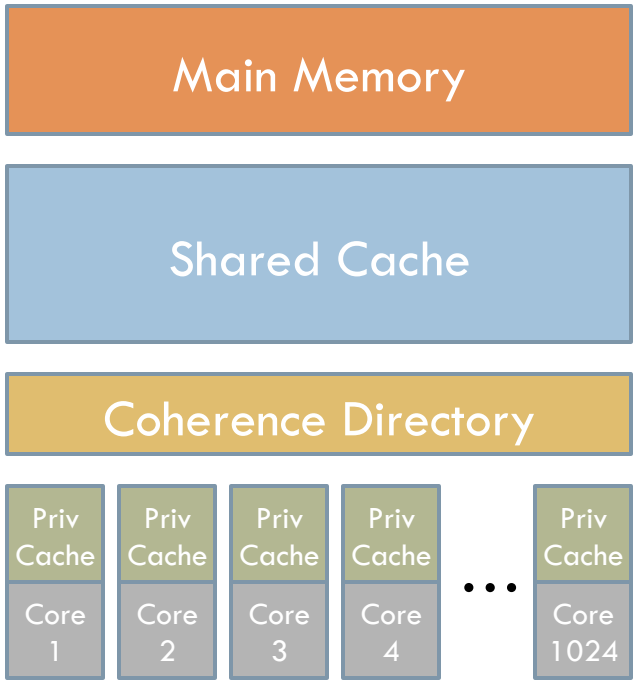
- Memory limits performance and energy efficiency
- Basic indicators:
  - 64-bit FP op: ~1ns latency, ~20pJ energy
  - Shared cache access: ~10ns latency, ~1nJ energy
  - DRAM access: ~100ns latency, ~20nJ energy
- HW & SW must optimize memory performance

Multicore Memory Hierarchy



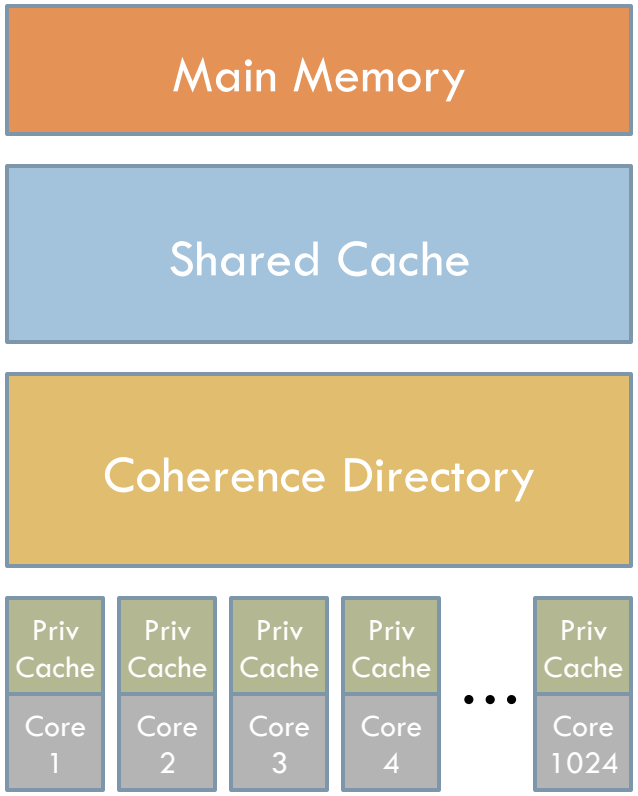
- Per-core private caches
  - Fast access to critical working set
  - Should satisfy most accesses
- Shared last-level cache
  - Increases utilization
  - Accelerates communication
  - Can be partitioned for isolation
- Coherence protocol
  - Makes caches transparent to SW
  - Uses directory to track sharers

Memory Hierarchy Challenges at 1K Cores



- Cache hierarchy is hard to scale

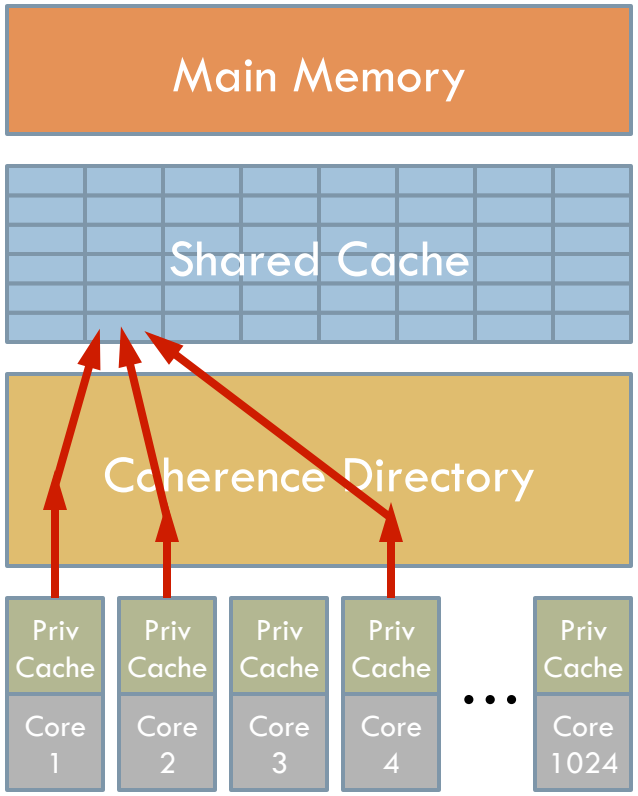
Memory Hierarchy Challenges at 1K Cores



- Cache hierarchy is hard to scale

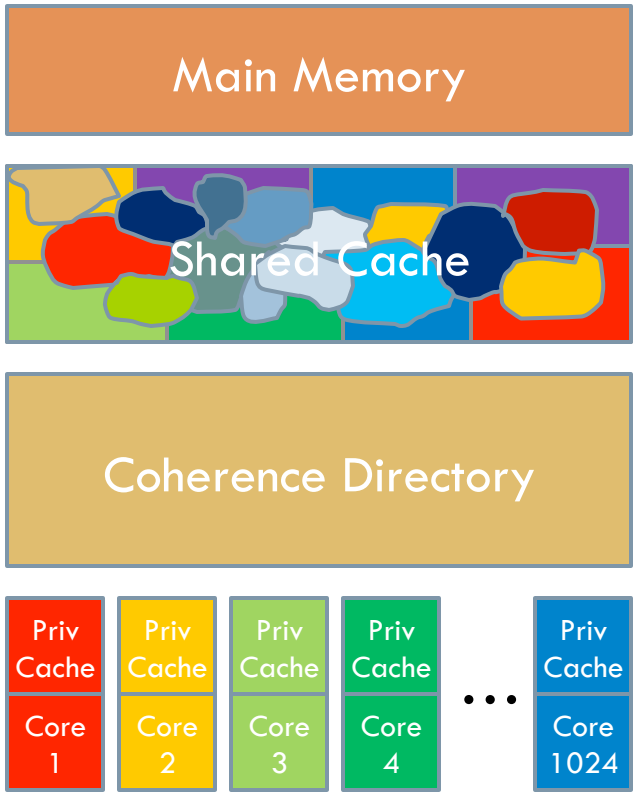
1. Directories scale poorly

Memory Hierarchy Challenges at 1K Cores



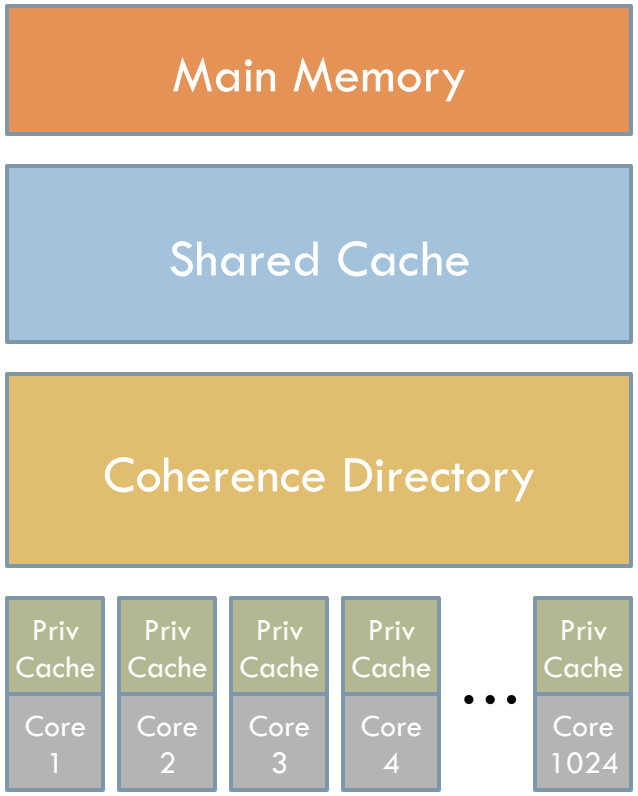
- Cache hierarchy is hard to scale
  1. Directories scale poorly
  2. Conflicts in caches & directory are more frequent

Memory Hierarchy Challenges at 1K Cores



- Cache hierarchy is hard to scale
  1. Directories scale poorly
  2. Conflicts in caches & directory are more frequent
  3. Shared cache cannot be partitioned efficiently

Memory Hierarchy Challenges at 1K Cores

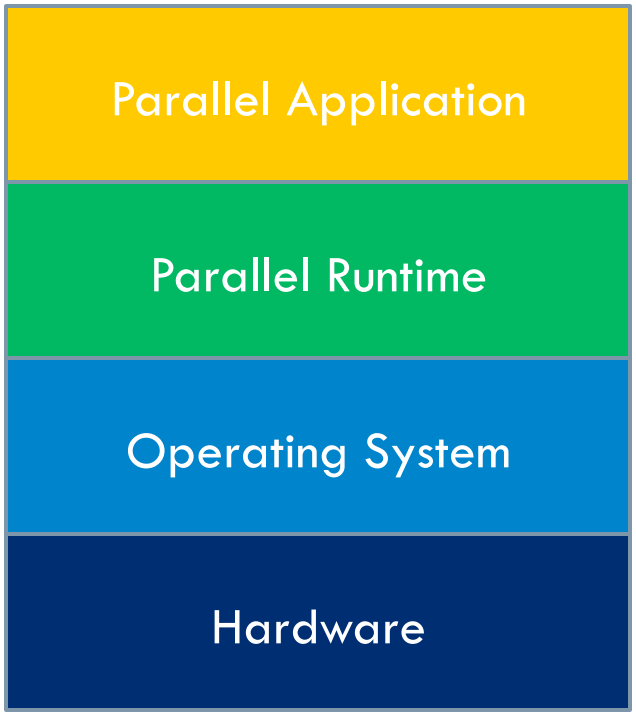


- Cache hierarchy is hard to scale

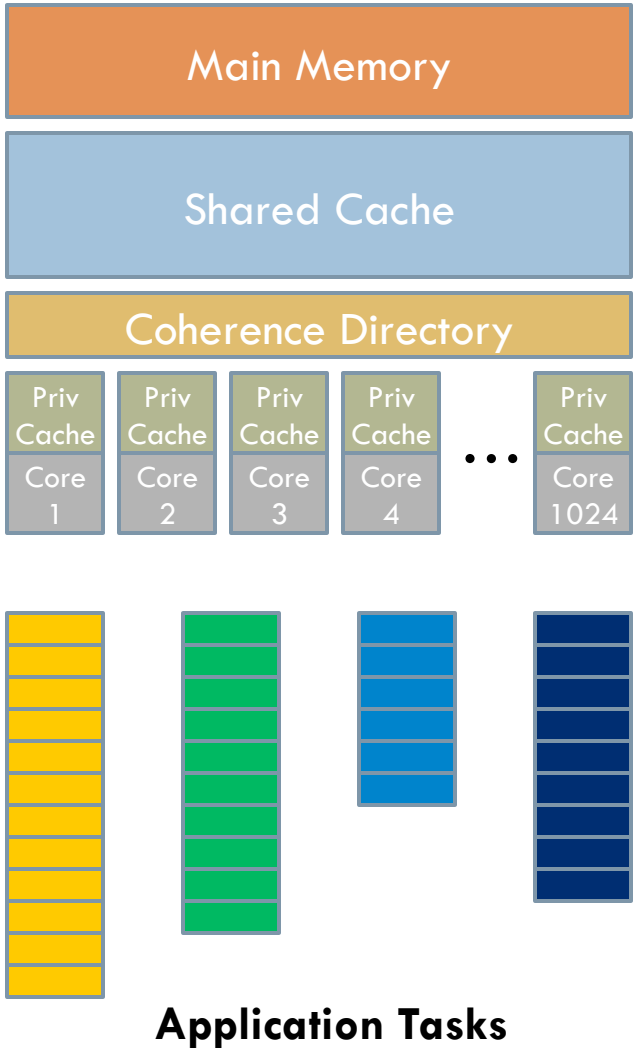
1. Directories scale poorly
2. Conflicts in caches & directory are more frequent
3. Shared cache cannot be partitioned efficiently
4. No isolation or QoS due to shared cache and directory

Scaling Parallel Runtimes

- Parallel runtime maps application to hardware
  - Resource management
  - Scheduling
- Runtime is fundamental to scale with manageable complexity

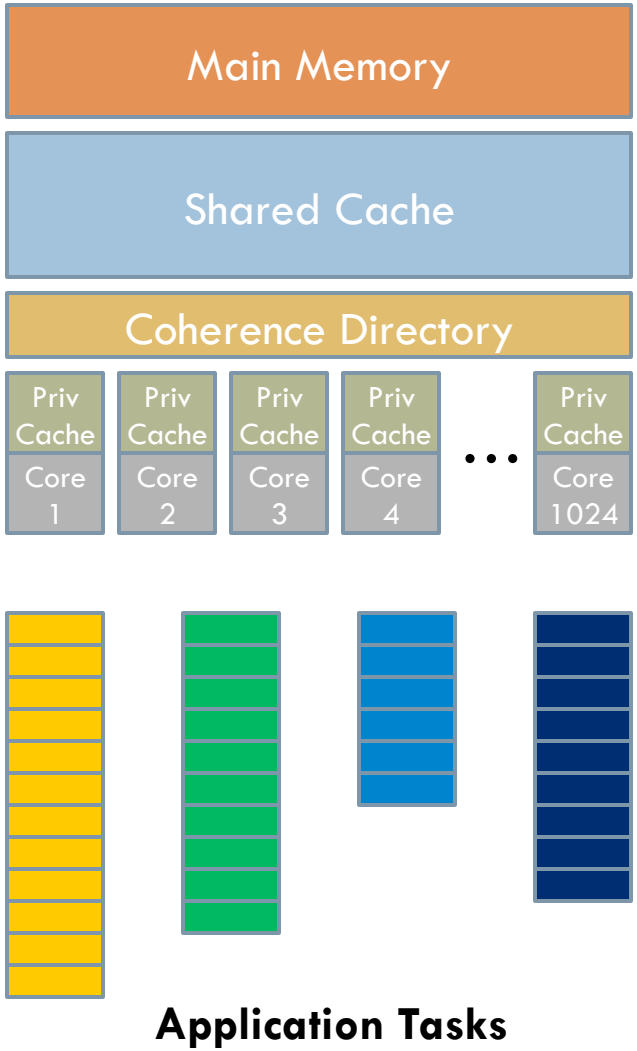


Scheduling Parallel Applications



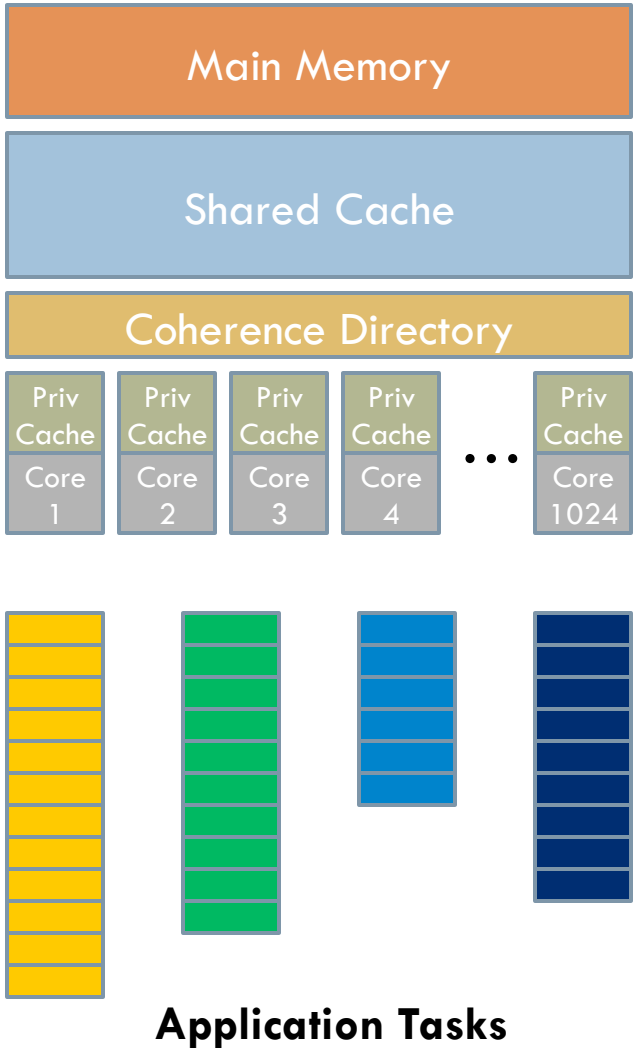
- Application → Parallel tasks
  - Different requirements
  - May have dependences

Scheduling Parallel Applications



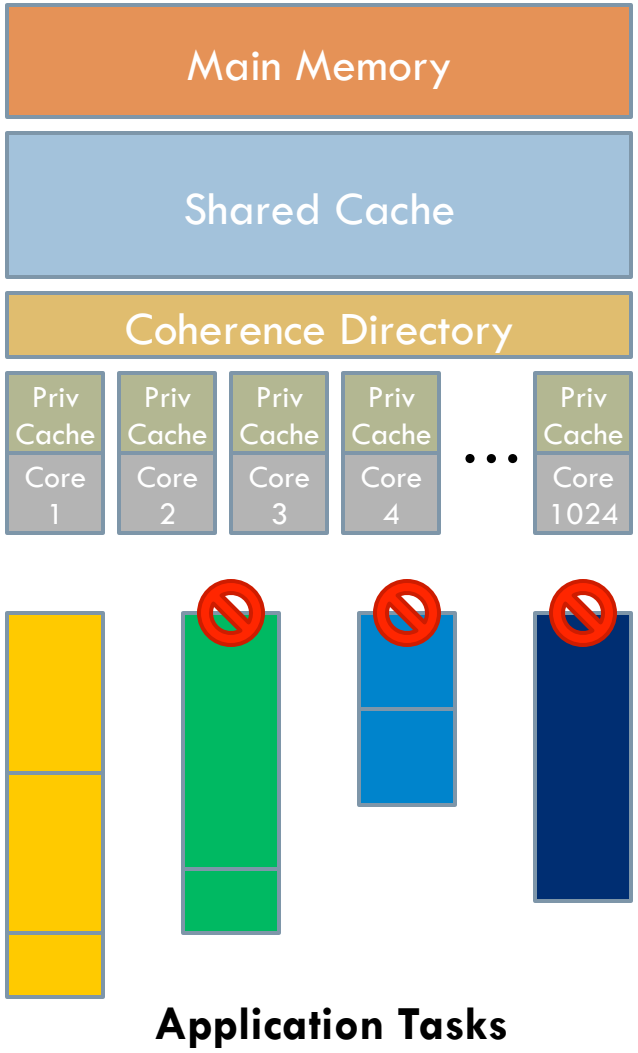
- Application → Parallel tasks
  - Different requirements
  - May have dependences
- Scheduler assigns tasks to cores

Runtime & Scheduling Challenges



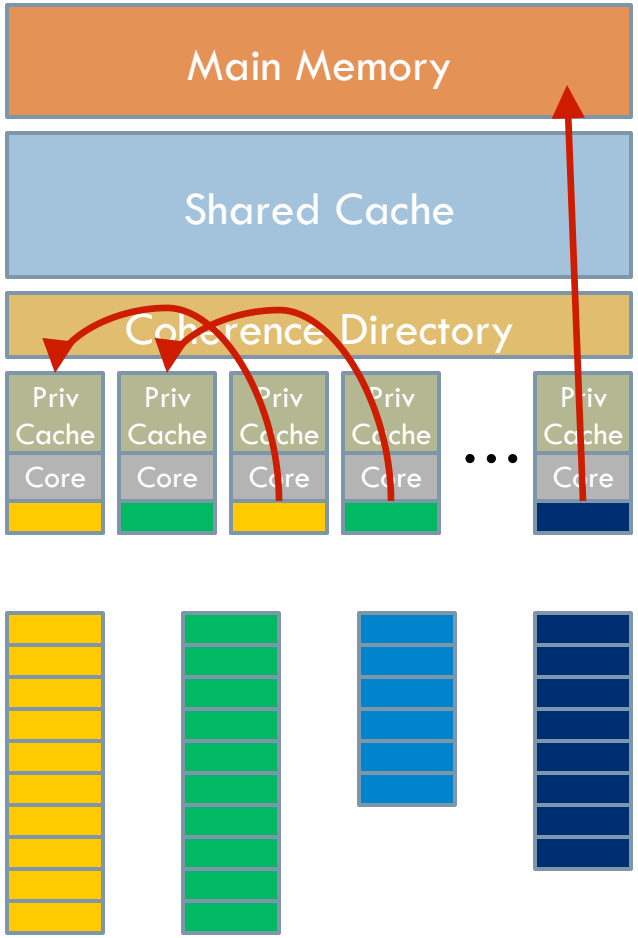
- Constrained parallelism

Runtime & Scheduling Challenges



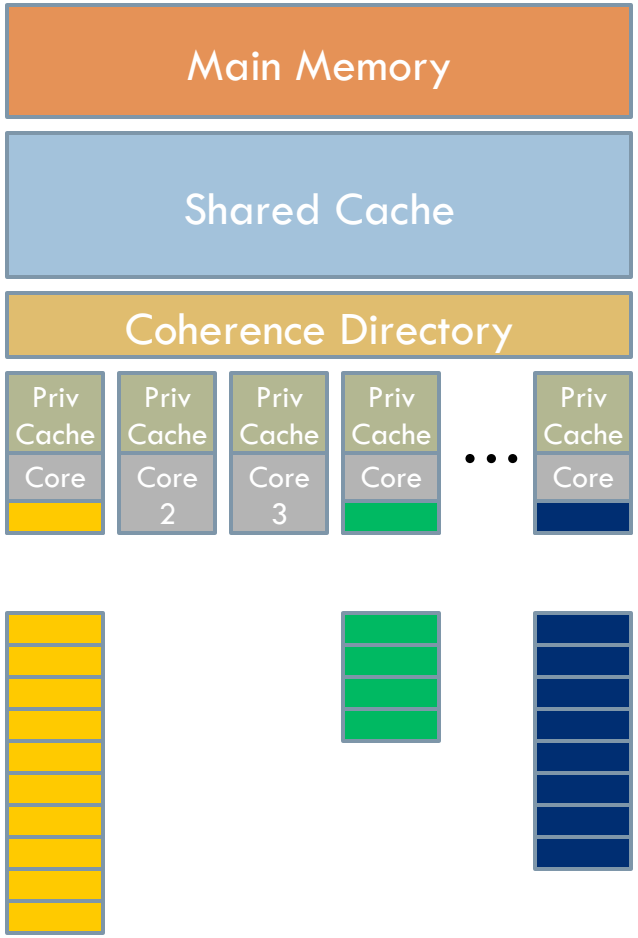
- Constrained parallelism
  - Coarser tasks
  - Unneeded serialization

Runtime & Scheduling Challenges



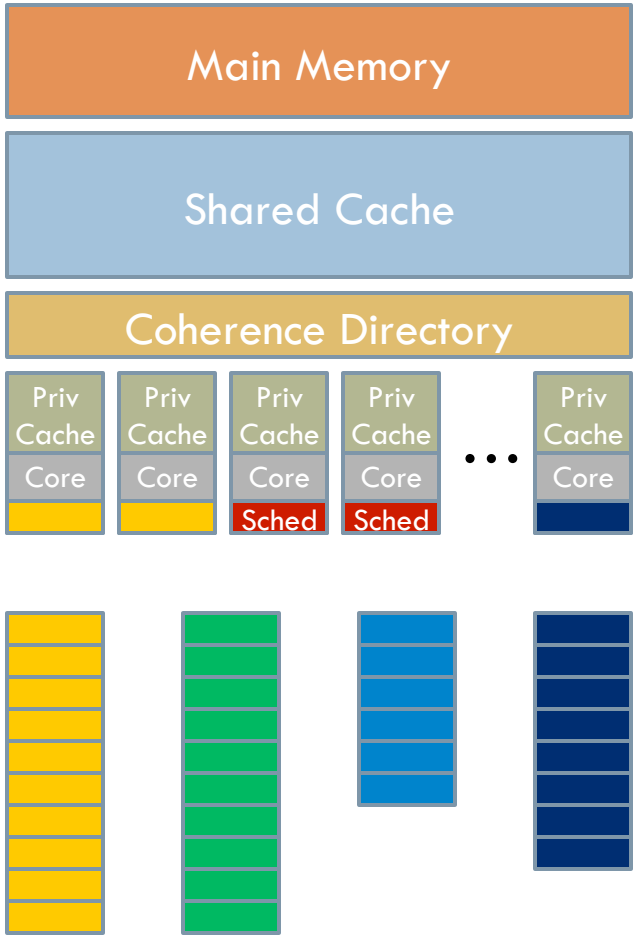
- Constrained parallelism
- Increased cache misses

Runtime & Scheduling Challenges



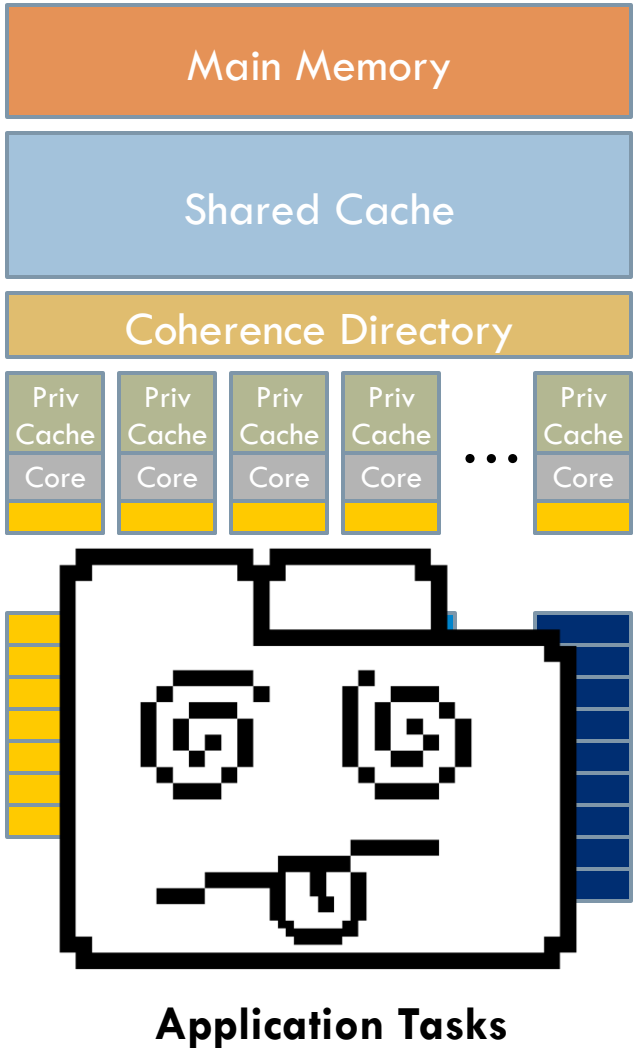
- Constrained parallelism
- Increased cache misses
- Load imbalance

Runtime & Scheduling Challenges



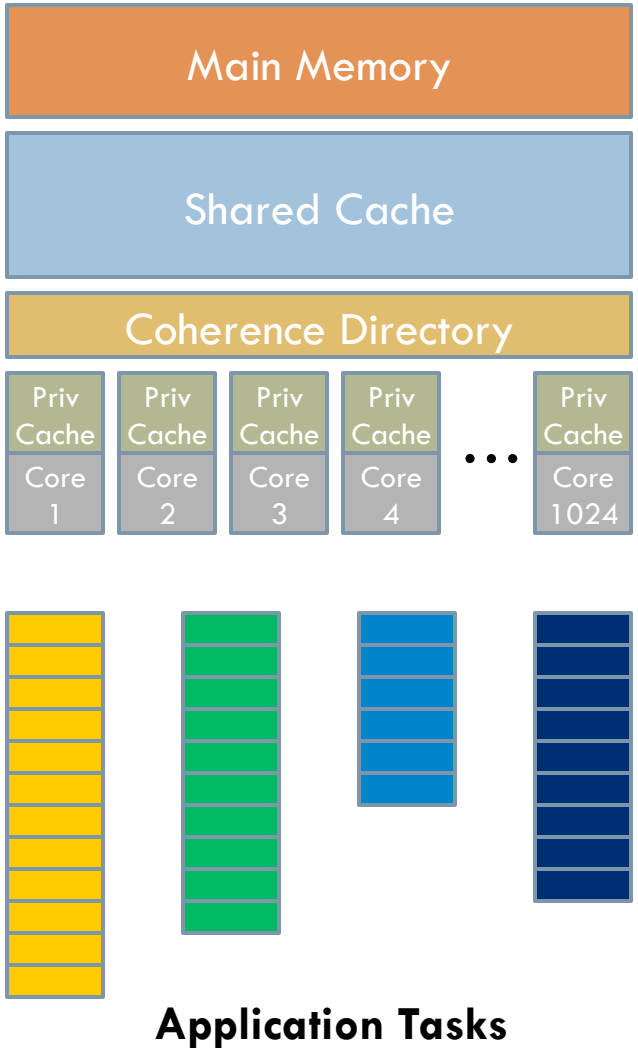
- Constrained parallelism
- Increased cache misses
- Load imbalance
- Scheduling overheads

Runtime & Scheduling Challenges



- Constrained parallelism
- Increased cache misses
- Load imbalance
- Scheduling overheads
- Excessive memory footprint (crash!)

Runtime & Scheduling Challenges



- Constrained parallelism
- Increased cache misses
- Load imbalance
- Scheduling overheads
- Excessive memory footprint (crash!)

- Conflicting issues → Need smart algorithms!

Contributions

- Scalable cache hierarchies:
  - Efficient highly-associative caches [MICRO 10]
  - Scalable cache partitioning [ISCA 11, Top Picks 12]
  - Scalable coherence directories [HPCA 12]

- Scalable scheduling:
  - Efficient dynamic scheduling by leveraging programming model information [PACT 11]
  - Hardware-accelerated scheduling [ASPLOS 10]

This Talk

- Scalable cache hierarchies:
  - Efficient highly-associative caches [MICRO 10]
  - Scalable cache partitioning [ISCA 11, Top Picks 12]
  - Scalable coherence directories [HPCA 12]

- Scalable scheduling:
  - Efficient dynamic scheduling by leveraging programming model information [PACT 11]
  - Hardware-accelerated scheduling [ASPLOS 10]

Rethinking Common-Case Design

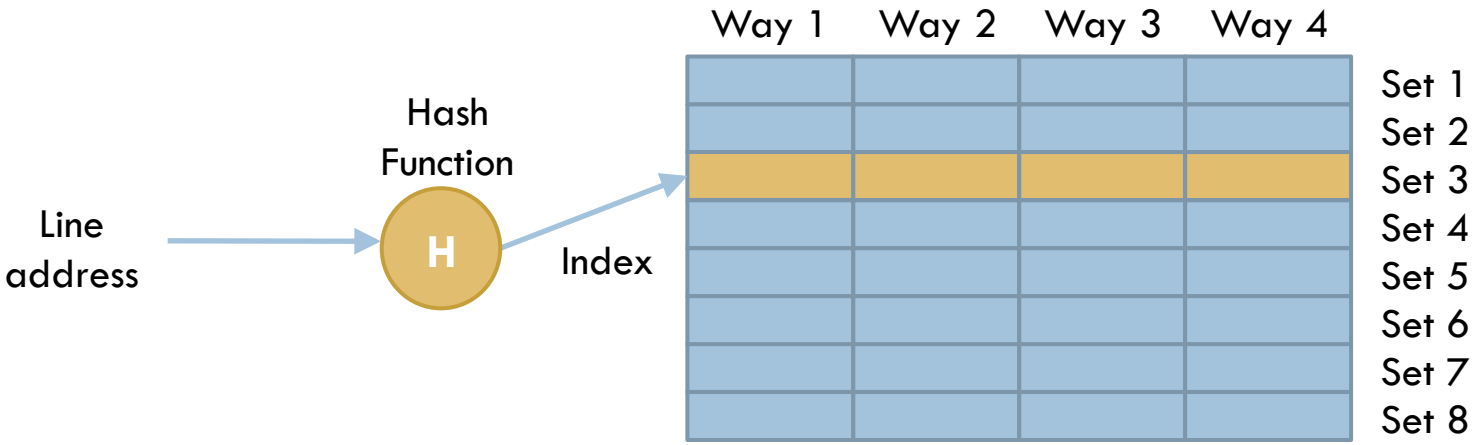
- Conventional approach: Make the common case fast
  - Based on patterns of past and current workloads
  - Overprovision to mitigate worst case or for future workloads
- Multicore demands going beyond the common case
  - Shared resources → Need guarantees on all cases
  - Overprovisioning alone is insufficient and wasteful
    - Some overprovisioning simplifies design
    - Must provide guarantees with minimal overprovisioning
  - Root cause: Empirical design → Limited understanding of system behavior

Solution: Analytical Design Approach

- Design basic components that are easily analyzable
  - Simple, accurate, workload-independent analytical models
  - Easy to understand, reason about behavior
- Use models to design systems that work well in all cases
  - Scalability and QoS guaranteed in all scenarios
  - Outperform conventional techniques in the common case
- Need to revisit fundamental aspects of our systems (associativity, coherence, …)

Set-Associative Caches

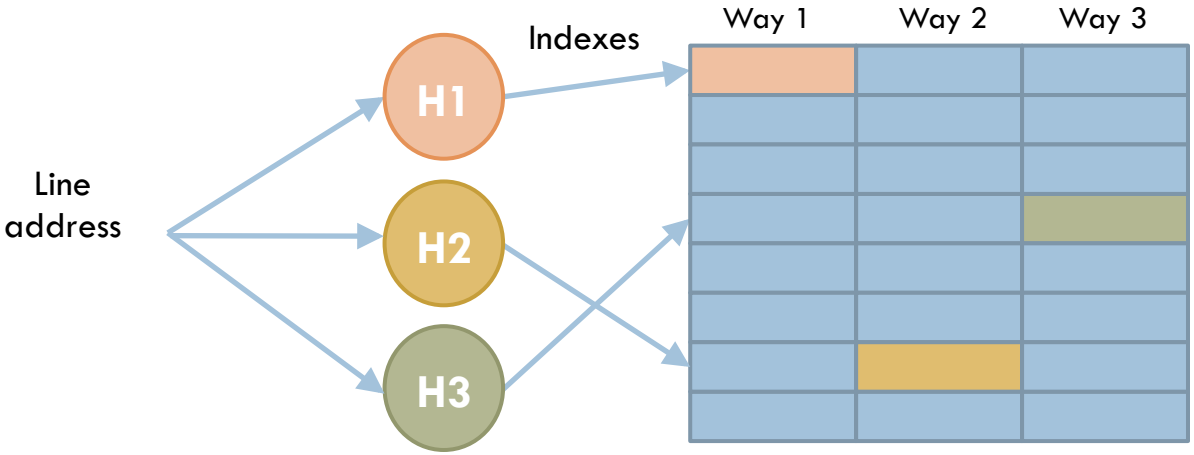
- Basic building block of caches, directories



- Problems:
  - Reducing conflicts (higher associativity) → more ways
    - Higher energy, latency, area
  - Conflicts depend on workload's access patterns

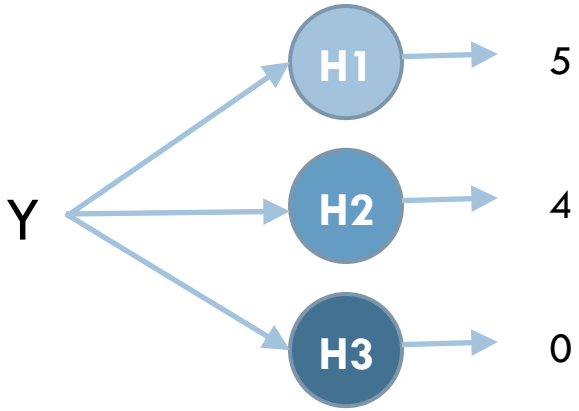
ZCache

- One hash function per way



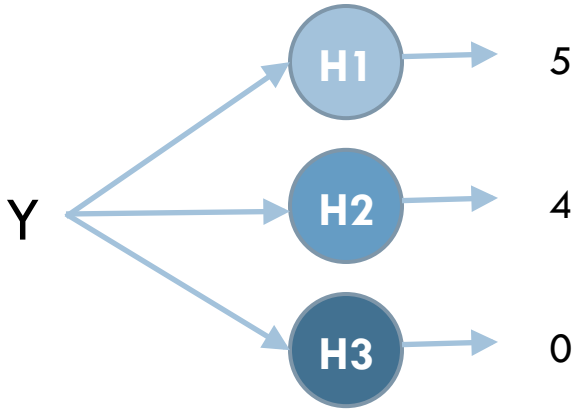
- Hits require a single lookup → low hit energy and latency
- Misses exploit the multiple hash functions to obtain an arbitrarily large number of replacement candidates
  - Multi-step process, draws on prior research on Cuckoo hashing
  - Happens infrequently (on misses) and off the critical path

ZCache Replacement



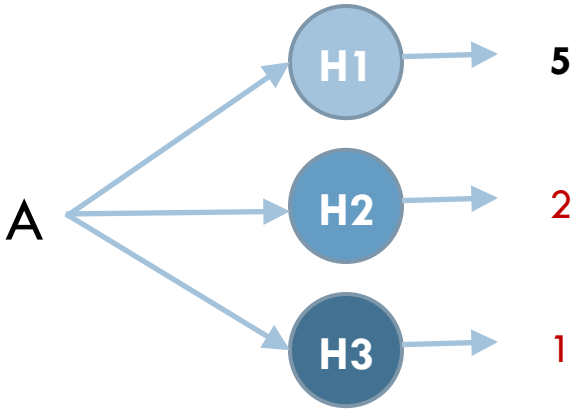
Way 1	Way 2	Way 3	
U	V	M	0
F	C	X	1
P	K	H	2
B	E	R	3
N	D	J	4
A	Z	Q	5
G	T	I	6
L	O	S	7

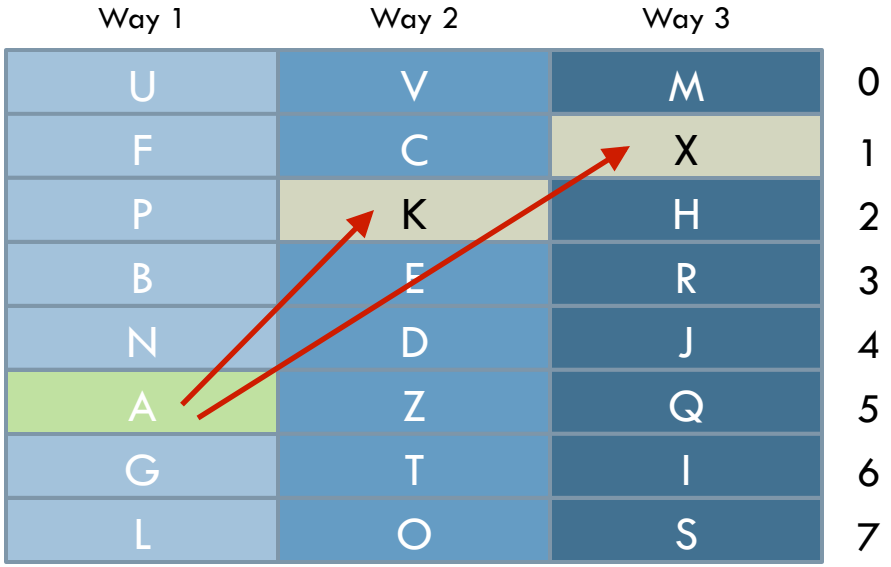
ZCache Replacement



Way 1	Way 2	Way 3	
U	V	M	0
F	C	X	1
P	K	H	2
B	E	R	3
N	D	J	4
A	Z	Q	5
G	T	I	6
L	O	S	7

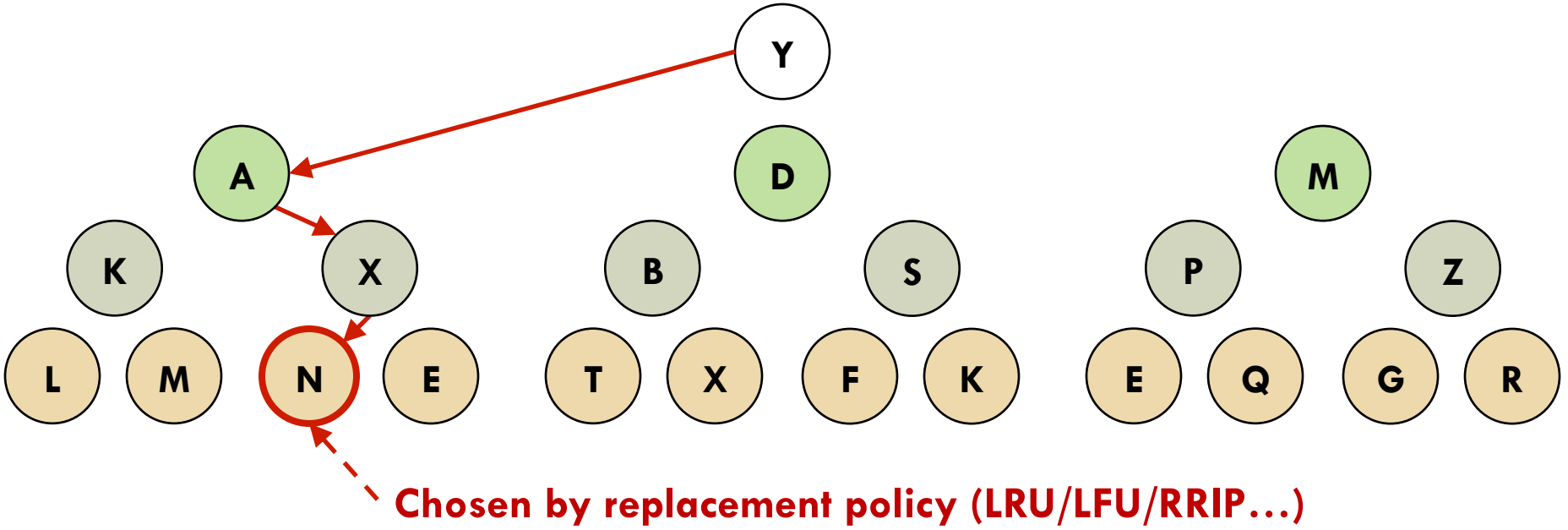
ZCache Replacement



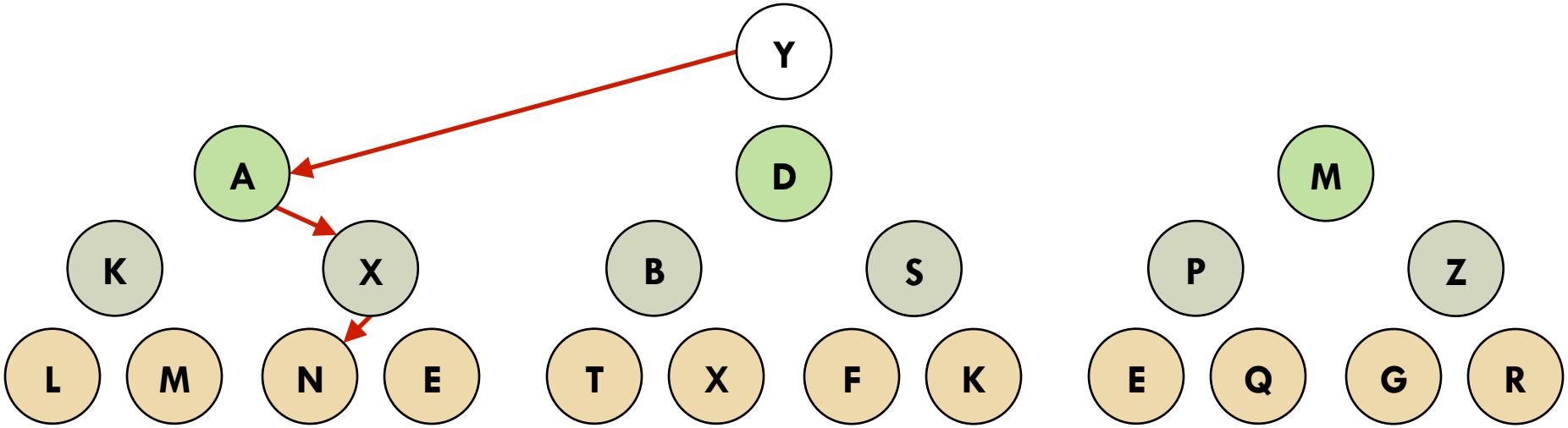


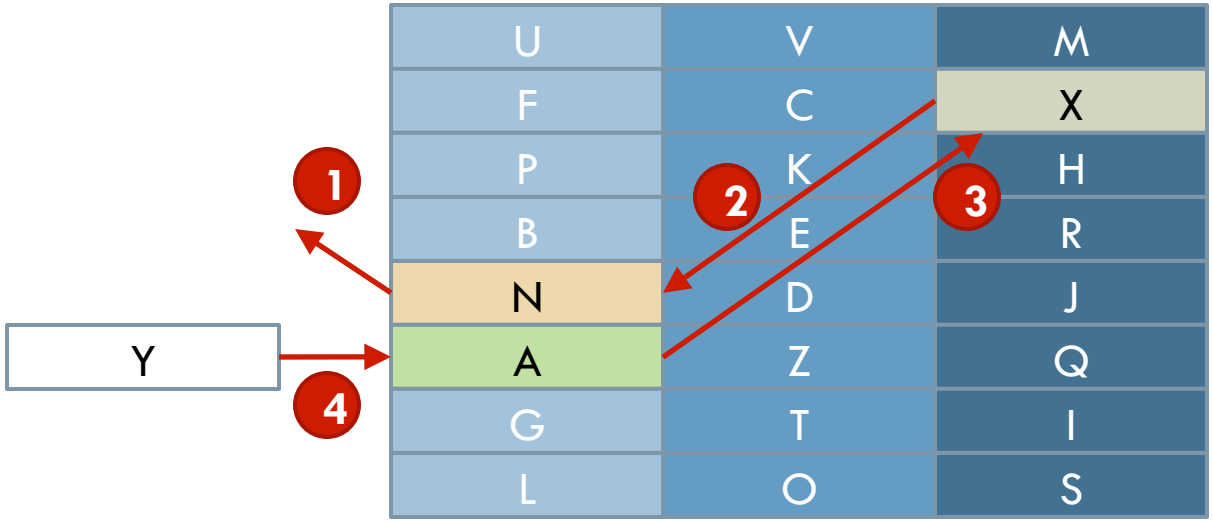
- Instead of evicting A, can move it and evict K or X
  - Similarly, can move K or X → more candidates

ZCache Replacement

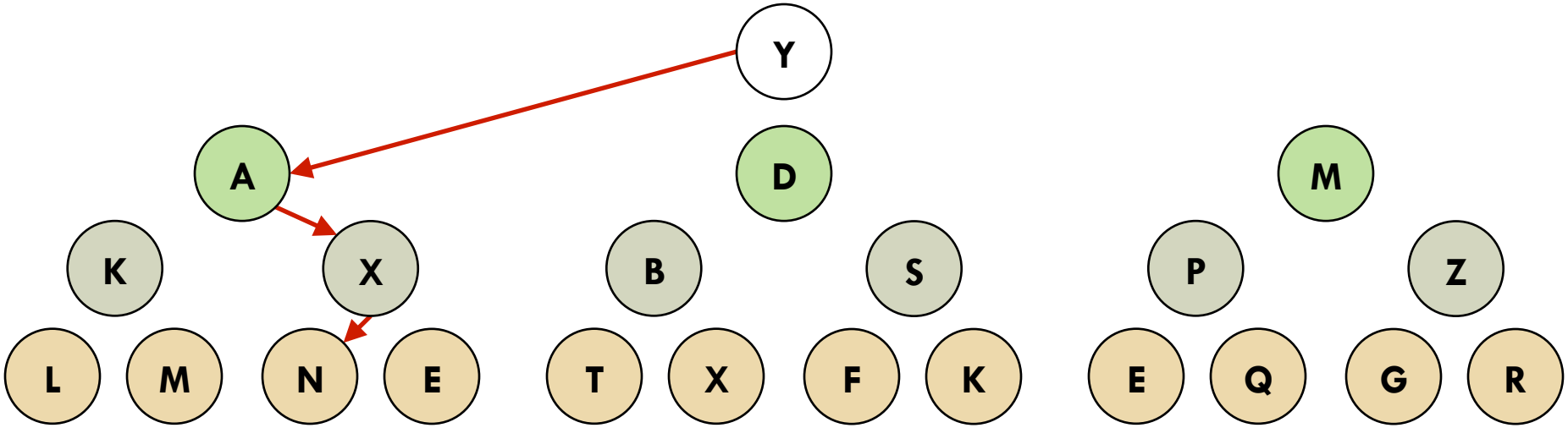


ZCache Replacement





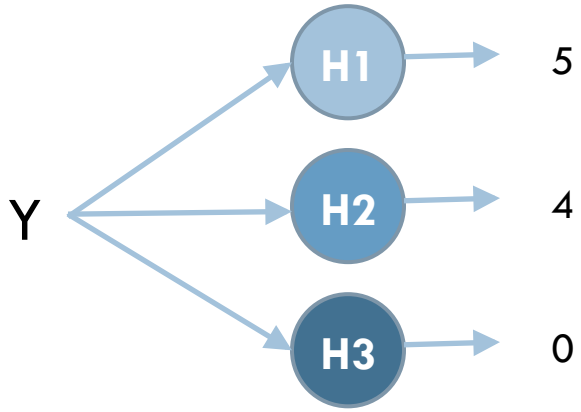
ZCache Replacement

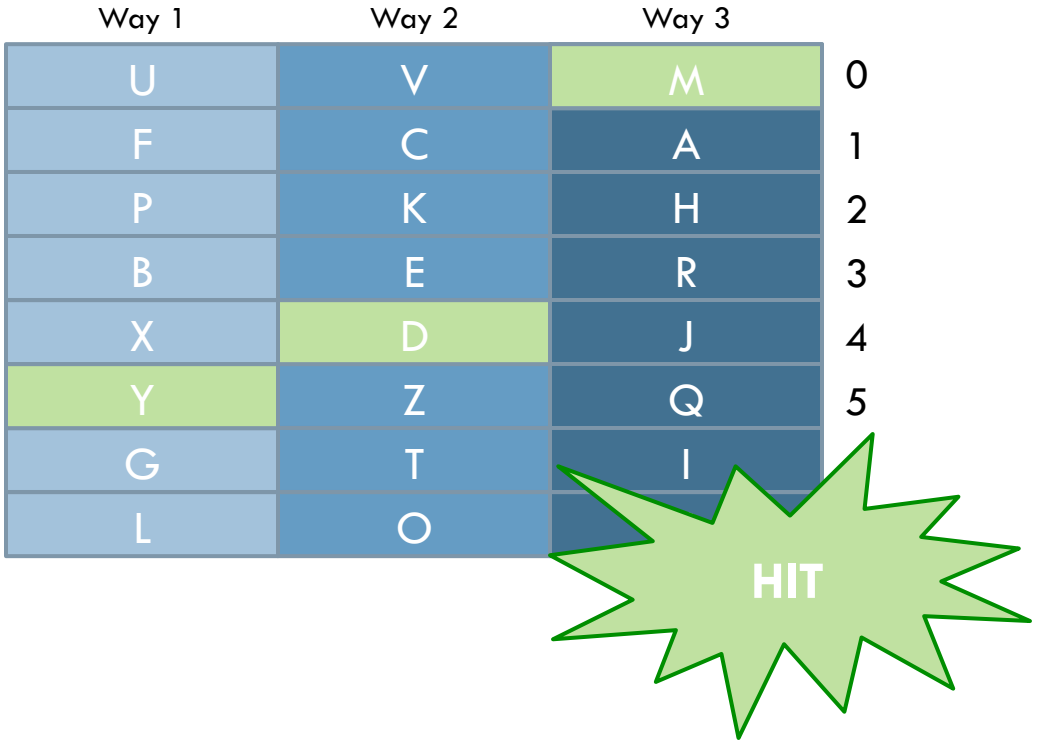


U	V	M
F	C	A
P	K	H
B	E	R
X	D	J
Y	Z	Q
G	T	I
L	O	S

ZCache Replacement

- Hits always take a single lookup





- Replacements do not affect hit latency, are simple to implement

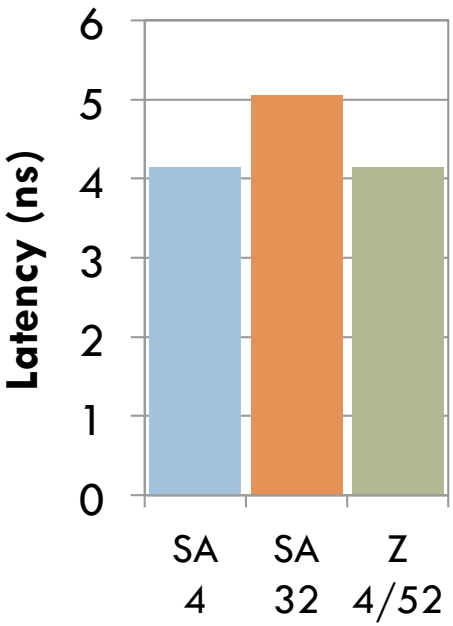
Methodology

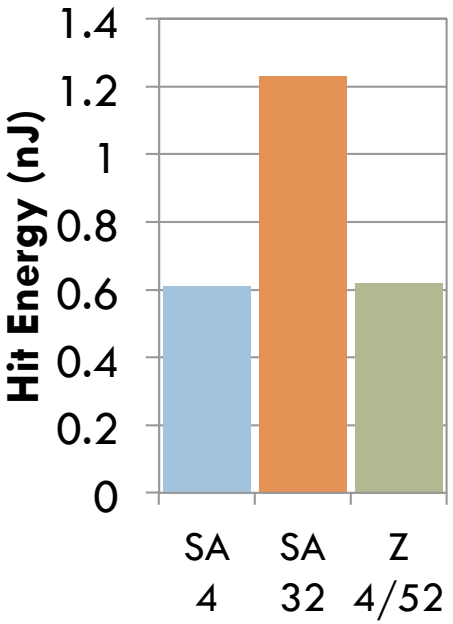
- zsim: A fast, 1000-core, microarchitectural x86 simulator
  - Fast: Parallel, leverages dynamic binary translation (Pin)
    - 15-60 Minstrs/s per host core, 600 Minstrs/s on 12-core Xeon
  - Scalable: Phase-based sync, simulates thousands of cores
  - Validated: Within 10% of Atom and Nehalem systems
  - Simple: ~20 KLoC, used in research and courses at Stanford

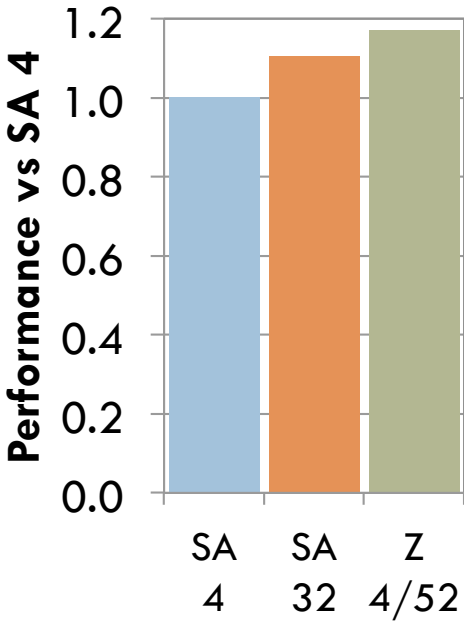
- Integrate zsim with existing area, energy, and latency models (McPAT, CACTI)

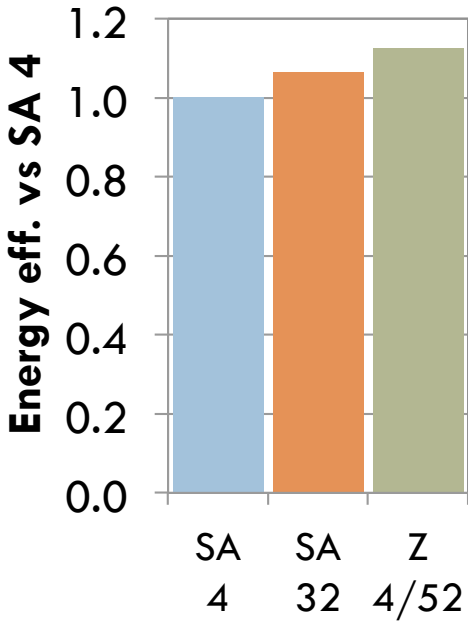
ZCache Benefits

- 8MB shared LLC optimized for area · latency · energy, 32nm:









- ZCache = Scalable associativity at low cost
  - Cost of 4-way cache
  - Associativity > 64-way cache

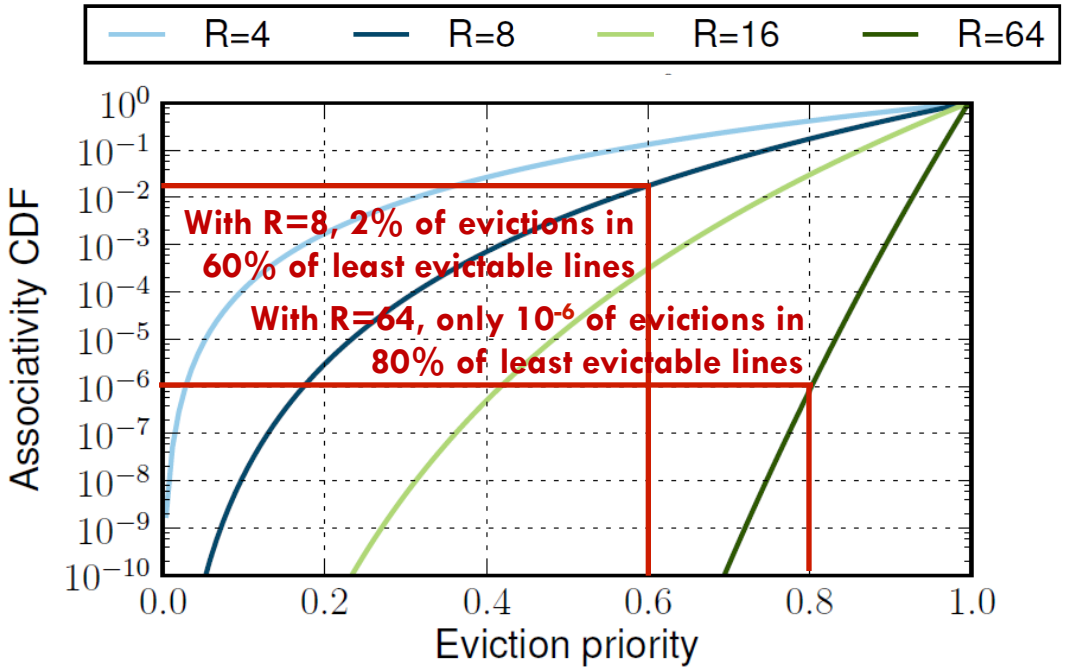
ZCache Associativity

- ZCache associativity depends only on the number of replacement candidates (R)
  - Independent of ways, workload, and replacement policy
- Problems in defining associativity: Cache array + replacement policy
- Insight 1: With ZCache, replacement candidates are very close to uniformly distributed over the array
- Insight 2: All policies do the same thing, rank cache lines
  - Eviction priority: Rank of a line normalized to [0,1]
    - Example: With LRU policy, LRU line has 1.0 priority, MRU has 0.0

ZCache Associativity

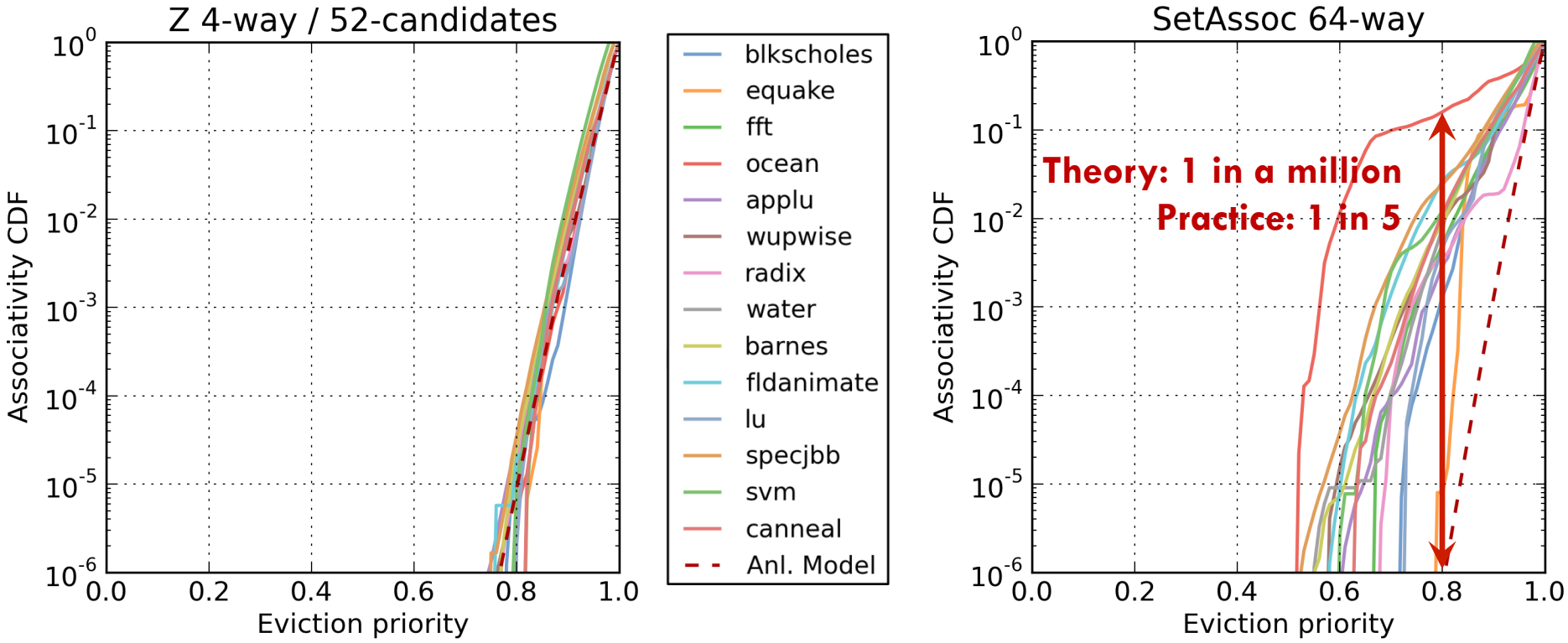
- Associativity: Probability distribution of eviction priorities of evicted lines
- ZCache associativity depends only on the number of replacement candidates (R):

$$F_A(x) = \Pr(A \le x) = x^R, x \in [0,1]$$



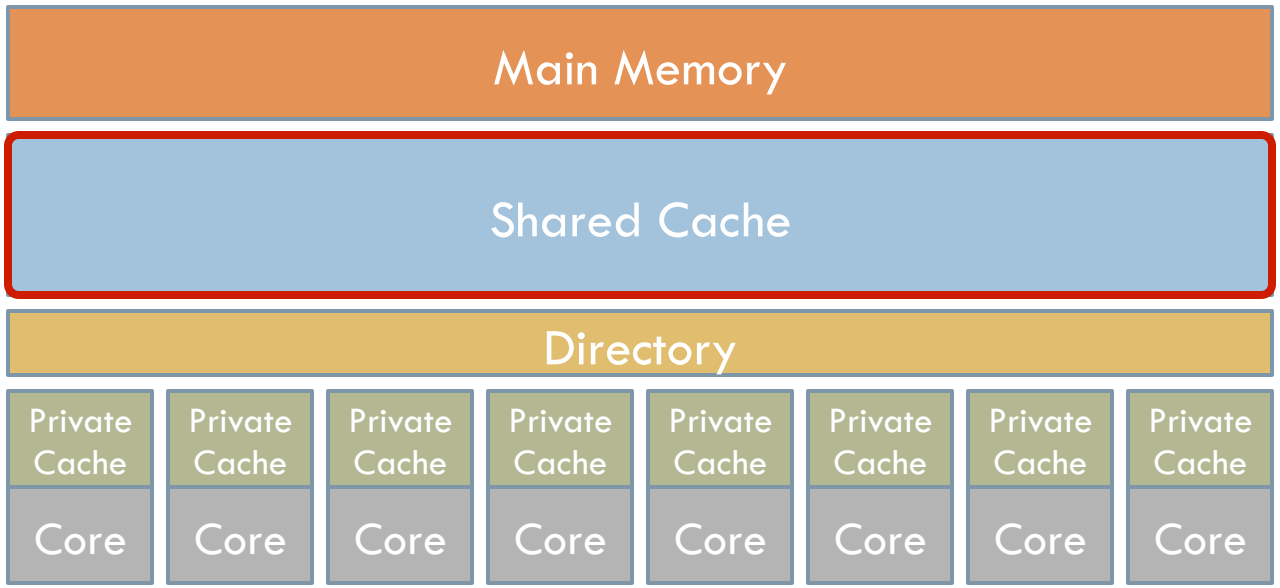
ZCache Analytical Models

- Analytical models are accurate in practice:

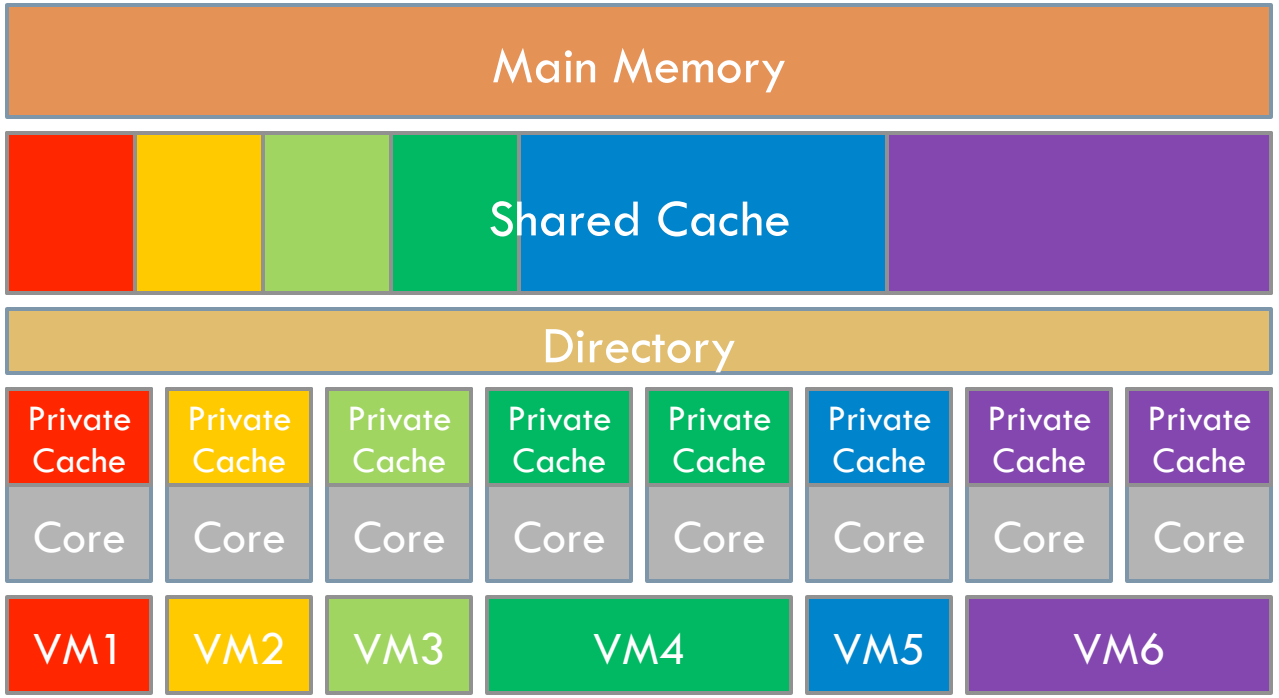


14 workloads, 1024 cores

Cache Partitioning



Cache Partitioning

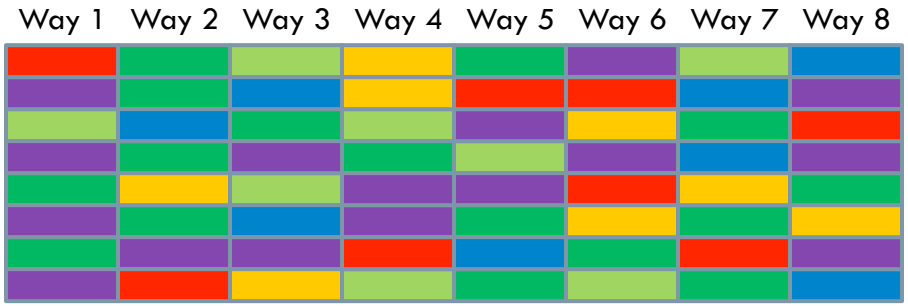


- Cache partitioning techniques divide cache space explicitly
  - Isolation: Virtualize cache among applications, VMs
  - Efficiency: Improve performance, fairness
  - Configurability: SW-controlled buffers (performance, security)

Cache Partitioning Techniques

- Strict partitioning schemes: Based on restricting line placement
  - Way partitioning: Restrict insertions to specific ways
  - Strict, but supports few partitions and degrades associativity

- Soft partitioning schemes: Based on tweaking the replacement policy
  - PIPP: Insert and promote lines in LRU chain depending on their partition
  - Simple, but approximate partitioning and degrades replacement performance



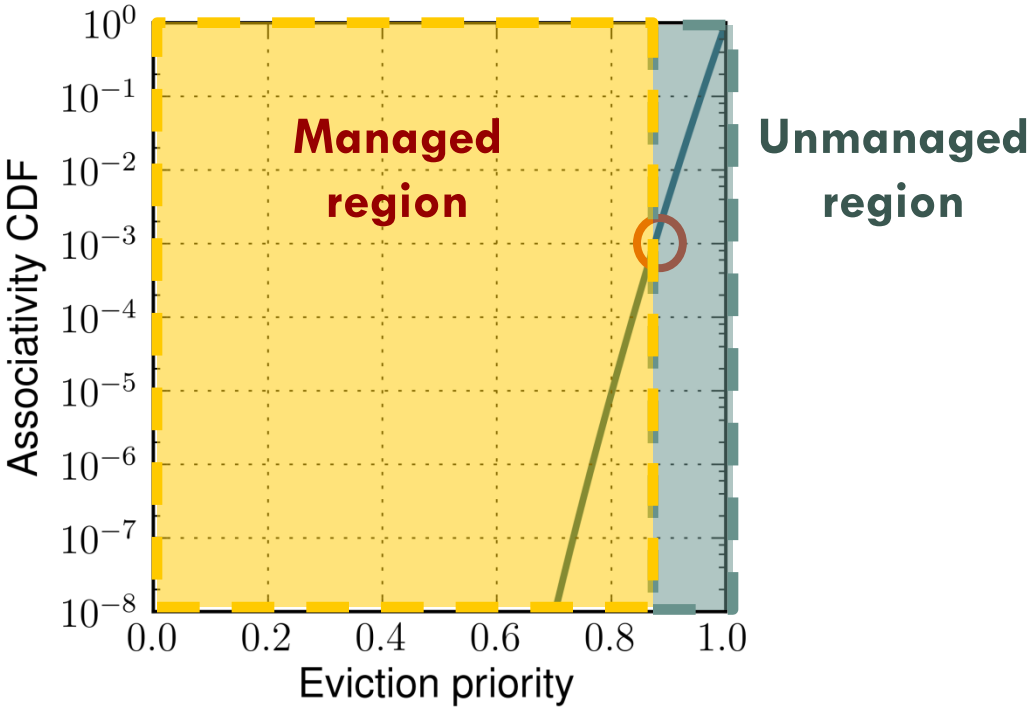
Cache Partitioning with Vantage

- Previous partitioning techniques have major drawbacks
  - Not scalable, support few partitions
  - Degrade performance

- Vantage solves deficiencies of previous techniques
  - Scalable: Supports hundreds of fine-grain partitions
  - Maintains high associativity and strict isolation among partitions (QoS)

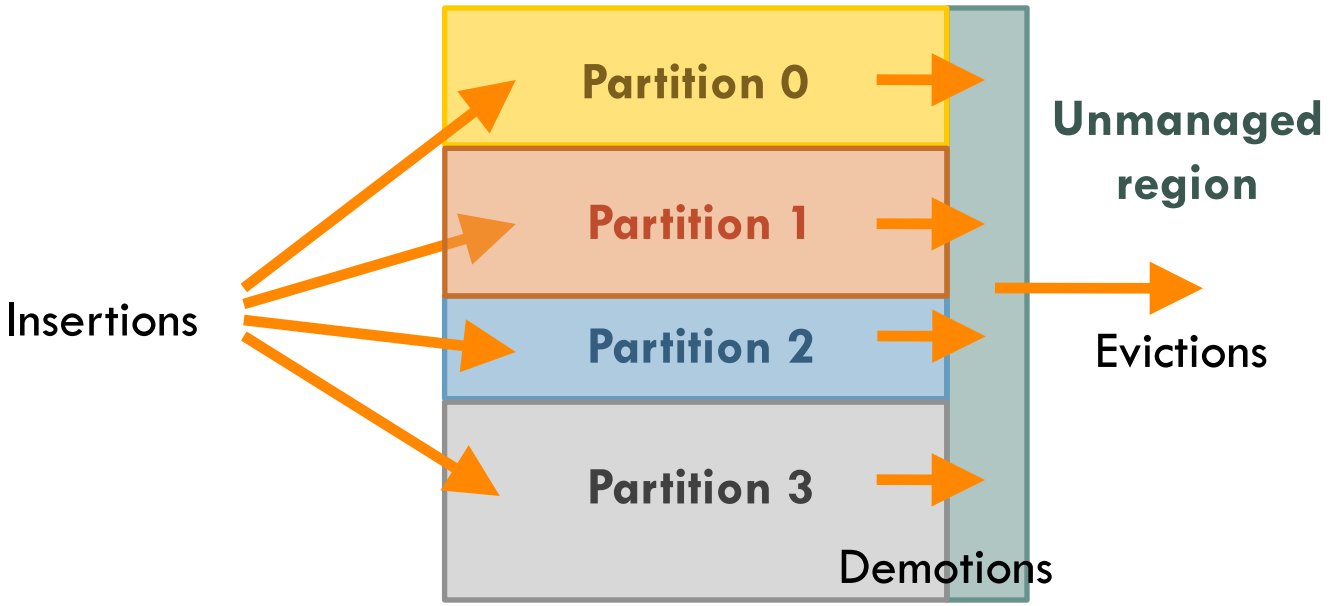
Vantage Design

- Vantage partitions most of the cache logically by modifying the replacement process
  - No restrictions on line placement

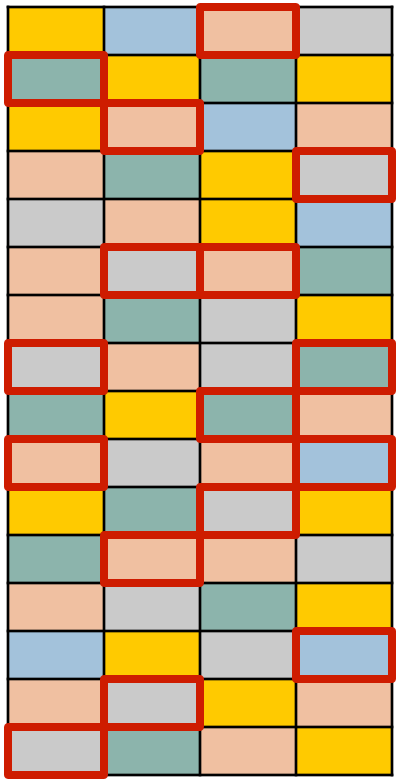


Vantage Design

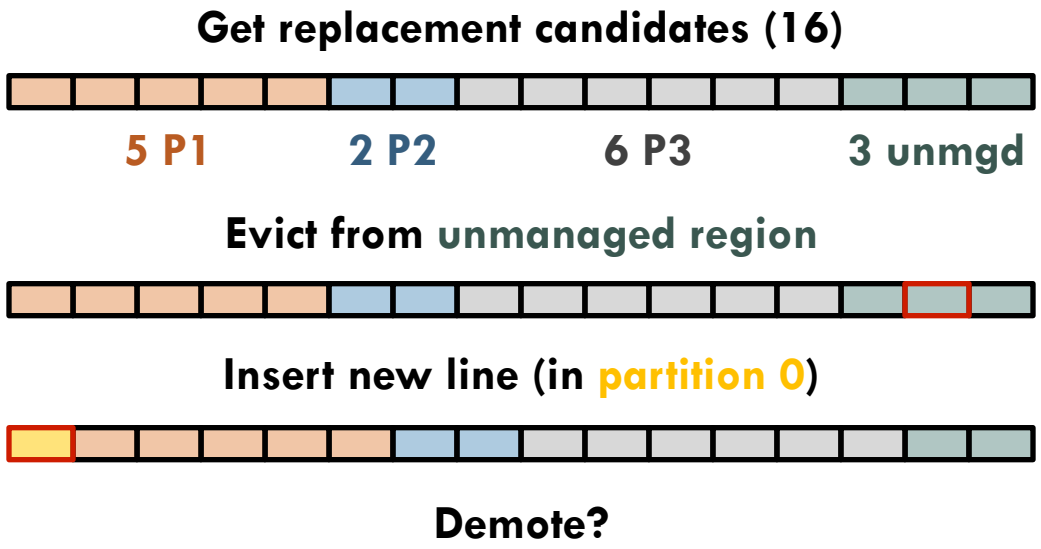
- Vantage partitions the managed region
  - Incoming lines (misses) inserted in partition
  - Each partition demotes least wanted lines to unmanaged region
  - Evict only from unmanaged region → no interference



Controlling Demotions



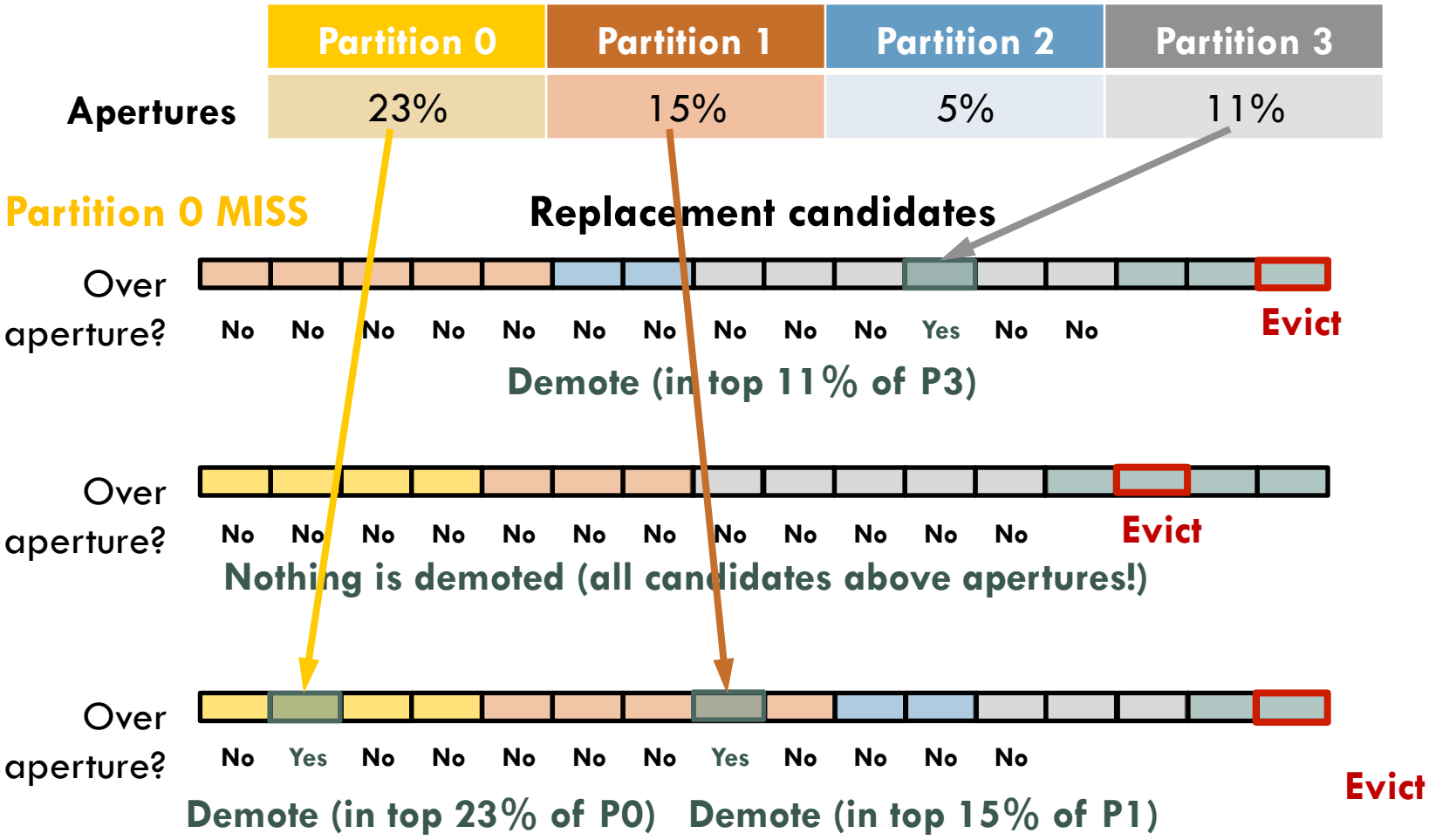
Access B (partition 0) → MISS



- Always demoting from inserting partition does not scale with number of partitions
- Instead, maintain sizes by matching demotion rate to miss rate

Demoting with Apertures

- Aperture: Portion of candidates to demote from each partition



Managing Apertures

- Partition apertures can be derived analytically:

$$A_i = \frac{M_i}{\sum_{k=1}^{P} M_k} \frac{\sum_{k=1}^{P} S_k}{S_i} \frac{1}{R \cdot m}$$

  - Intuition: Aperture ~ *miss rate (Mi)/size (Si)*

- Apertures are also capped to A_{max}
  - Higher aperture ↔ lower partition associativity
  - A_{max} ensures high minimum associativity
    - e.g., A_{max} =40% ~ R=16 associativity
  - We just let partitions that need $A_i > A_{max}$ grow

Bounds on Size and Interference

- The worst-case total growth of all partitions over their target sizes is bounded and small:

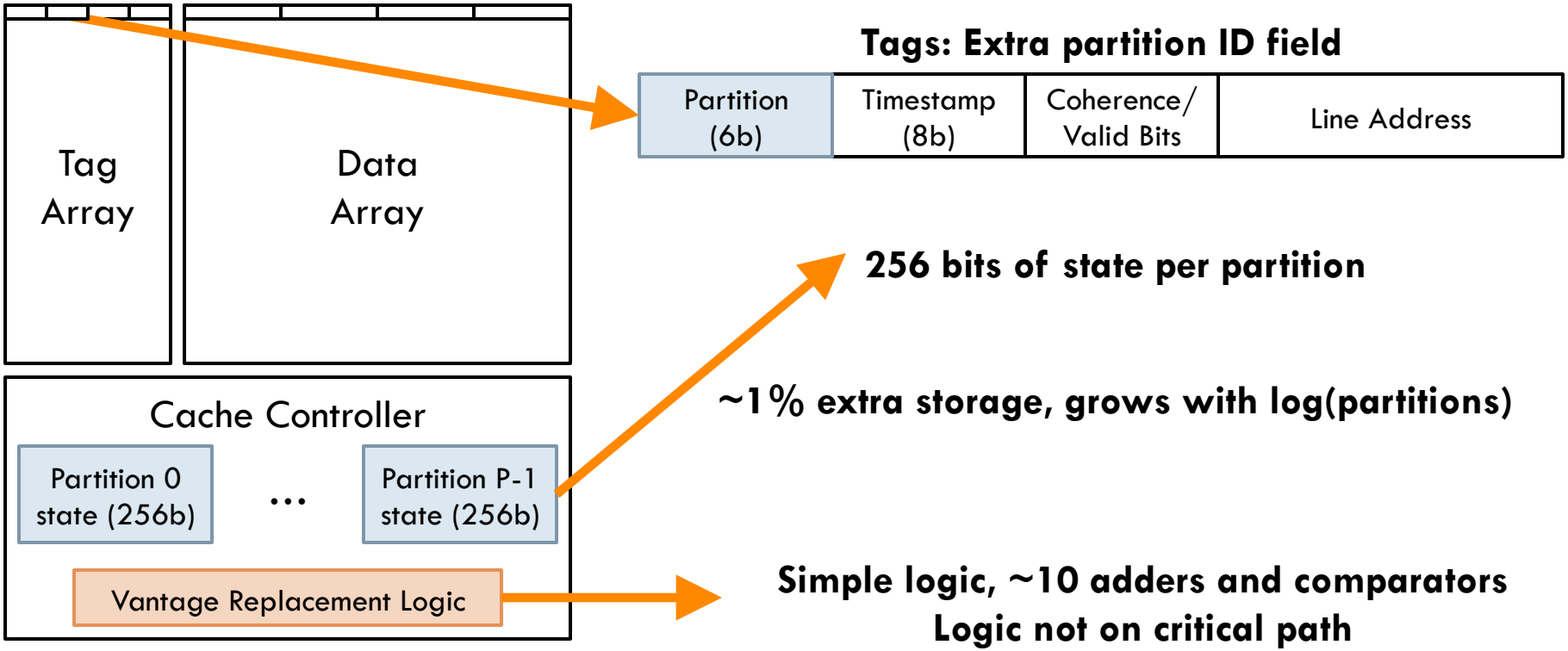
$$\Delta = \frac{1}{A_{\max}} \frac{1}{R}$$

  - Intuition: A Δ-sized partition is always stable, and multiple unstable partitions help each other demote
  - Independent of the number of partitions!

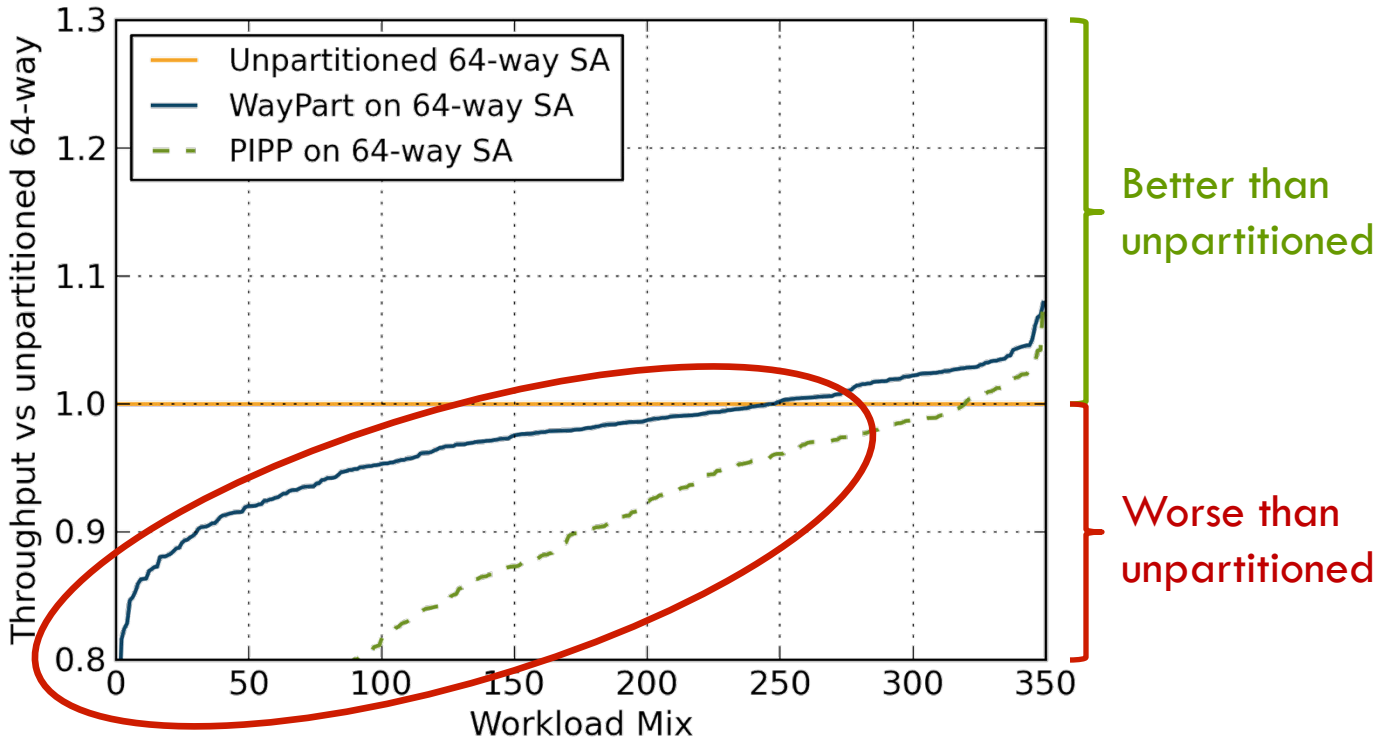
- Assign an extra Δ to unmanaged region
  - With R=52 and A_{max}=0.4, Δ=5% of the cache
  - Bounded worst-case sizes & interference

A Simple Vantage Controller

- Use negative feedback loop to derive apertures
- Use timestamps to determine lines within aperture
- Practical implementation that maintains analytical guarantees

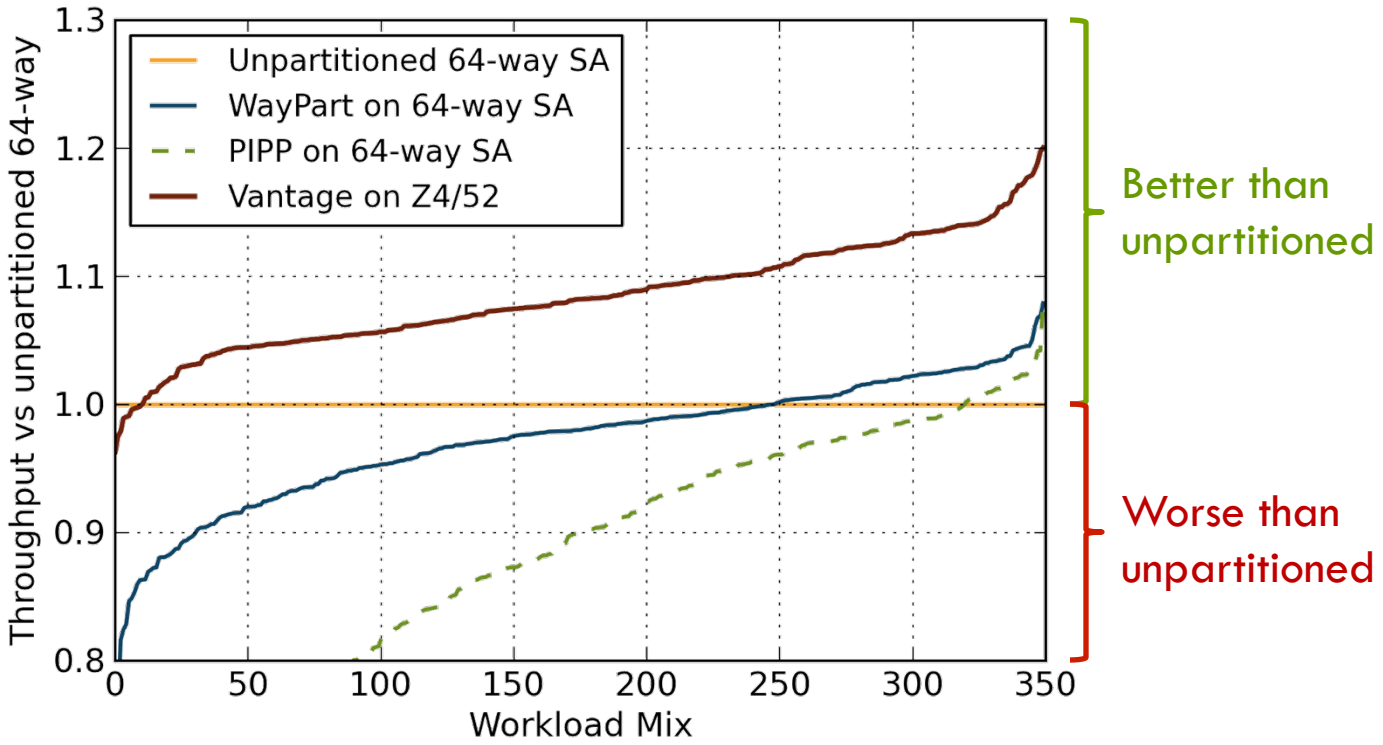


Vantage Evaluation



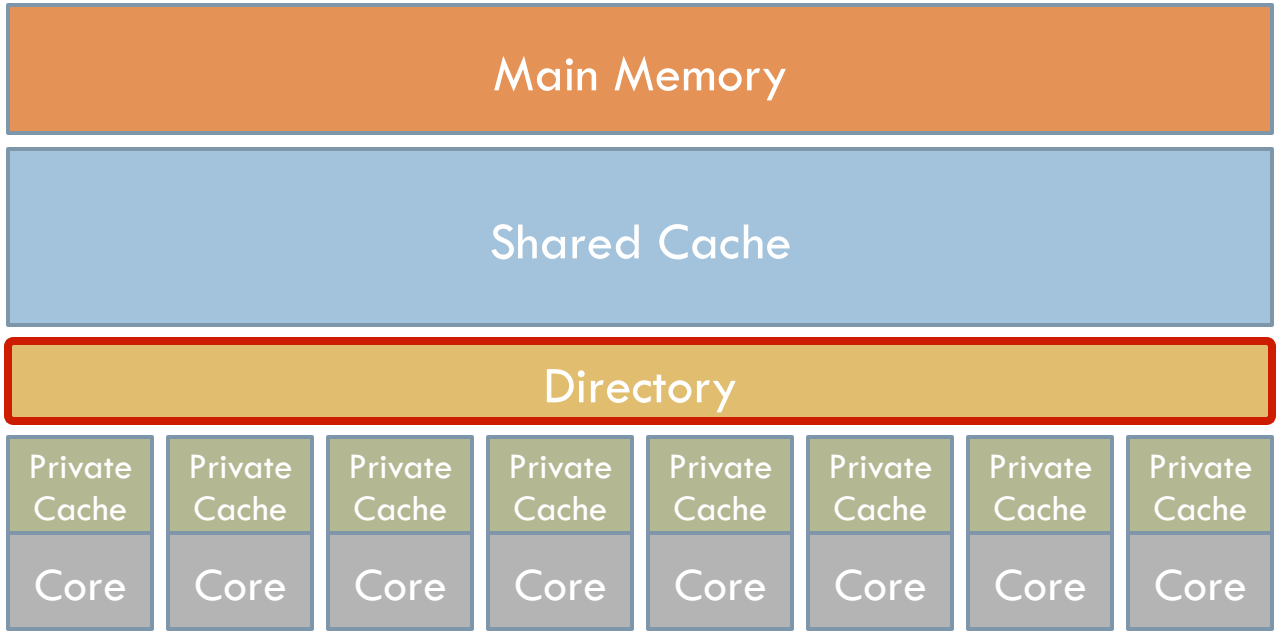
- 350 mixes on a 32-core CMP with a shared LLC (32 partitions)
- Partitions sized to maximize throughput (utility-based partitioning)
- Each line shows throughput vs unpartitioned 64-way baseline
- Way-partitioning, PIPP degrade throughput for most workloads

Vantage Evaluation



- Vantage improves throughput for most workloads using a 4-way/52-candidate Zcache
- Other schemes cannot scale beyond a few cores

Scaling Directories



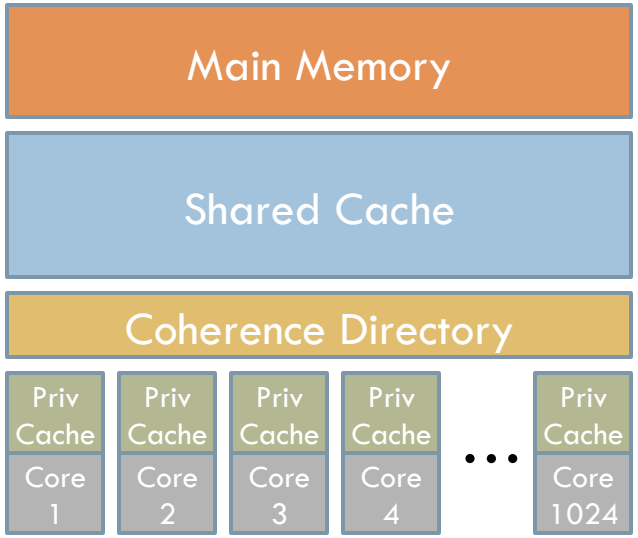
- Scaling directories is hard:
  - Excessive latency, energy, area overheads, or too complex
  - Introduce invalidations → Interference

Scalable Coherence Directory

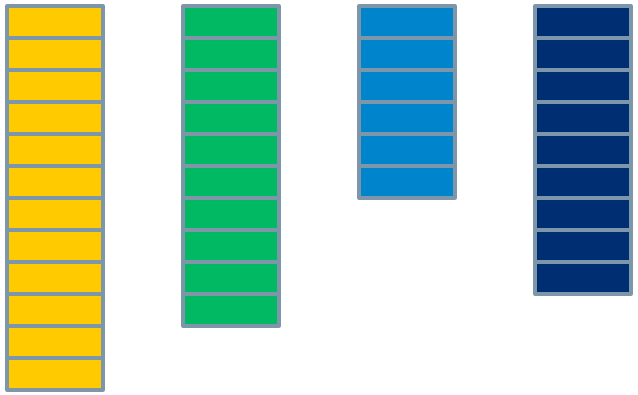
- Insights:
  - Flexible sharer set encoding: Lines with few sharers use one entry, widely shared lines use multiple entries → Scalability
  - Use ZCache → Efficient high associativity, analytical models
    - Negligible invalidations with minimal overprovisioning (~10%)

- SCD achieves scalability and performance guarantees
  - Area, energy grow with log(cores), constant latency
  - Simple: No modifications to coherence protocol
  - At 1024 cores, SCD is 13x smaller than a sparse directory, 2x smaller, faster and simpler than a hierarchical directory

Scalable Scheduling



- Scheduling requirements:
  - Expose enough parallelism
  - Locality-aware
  - Load balancing
  - Low overheads
  - Bounded memory footprint



Application Tasks

- Dynamic vs static schedulers:
  - Dynamic: Poor locality, footprint not bounded if non-trivial dependences
  - Static: Great compile-time schedules, but no load-balancing, only regular apps

Insight: Leverage Programming Model

- Solution: Dynamic fine-grain scheduling techniques that leverage programming model information to satisfy requirements
  - Expose all parallelism through fine-grain tasks
  - Locality-aware task queuing and load-balancing
  - Bounded footprint
  - Make dynamic scheduling practical in rich programming models (Streamlt, GRAMPS, Delite)
- Significant improvements over state-of-the-art schedulers on existing 12-core, 24-thread Xeon SMP:
  - Up to 17x over dynamic (more parallelism, locality-aware, footprint)
  - Up to 5.3x over static (no load imbalance)
- Scheduler choice becomes more critical as we scale up!

Hardware-Accelerated Schedulers

- Fine-grain scheduling with 100+ threads is slow in software
  - Hardware schedulers (e.g., GPUs): Fast but inflexible
- Insight: Software schedulers dominated by communication
- Solution: Accelerate communication with simple hardware
  - ADM: Asynchronous, register-register messages between threads
    - Small and scalable costs (~1KB buffers per core), virtualizable
  - ADM-accelerated fine-grain schedulers:
    - Achieve speed and scalability of HW + flexibility of SW
    - At 512 threads, 6.4x faster than SW and 70% faster than HW
  - ADM can accelerate other primitives (e.g., barriers, IPC)

Contributions

- Scalable cache hierarchies:
  - Efficient highly-associative caches [MICRO 10]
  - Scalable cache partitioning [ISCA 11, Top Picks 12]
  - Scalable coherence directories [HPCA 12]

- Scalable scheduling:
  - Efficient dynamic scheduling by leveraging programming model information [PACT 11]
  - Hardware-accelerated scheduling [ASPLOS 10]

Conclusions

- Scaling to 1000 cores requires HW and SW techniques:
  - Scale hardware with highly efficient caches with scalable partitioning and coherence
  - Scale software with dynamic, fine-grain, HW-accelerated scheduling

Acknowledgements

- Christos
- Research group: Jacob, David, Richard, Christina, Woongki, Austen, Mike, Hari
- PPL faculty: Kunle, Bill, Mark, Pat, Mendel, John, Alex
- PPL students: George, Jeremy, …
- Defense committee: Bill, Kunle, Nick
- Family & friends
  - Borja, Gemma, Idoia, Carlos, Felix, Dani, Manuel, Gonzalo, Adrian, Christina, George, Yiannis, Sotiria, Alexandros, Nadine, Martin, Elliot, Nick, Steph, Olivier, Leen, John, Sam, Mario, Nicole, Cristina, Kshipra, Robert, Erik, …

THANK YOU FOR YOUR ATTENTION

QUESTIONS?