CloudCache: Expanding and Shrinking Private Caches

Sangyeun Cho

Computer Science Department
University of Pittsburgh

Credits

- Parts of the work presented in this talk are from the results obtained in collaboration with students and faculty at the University of Pittsburgh:
  - Mohammad Hammoud
  - Lei Jin
  - Hyunjin Lee
  - Kiyeon Lee
  - Prof. Bruce Childers
  - Prof. Rami Melhem
- Partial support has come as:
  - NSF grant CCF-0702236
  - NSF grant CCF-0952273
  - A. Richard Newton Graduate Scholarship, ACM DAC 2008

Recent multicore design trends

- **Modular designs based on relatively simple cores**
  - Easier to validate (a single core)
  - Easier to scale (the same validated design replicated multiple times)
  - Due to these reasons, future "many-core" processors may look like this
  - Examples: Tilera TILE*, Intel 48-core chip [Howard et al., ISSCC '10]

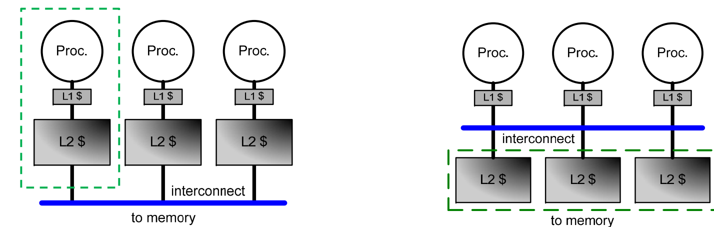
Recent multicore design trends

- **Modular designs based on relatively simple cores**
  - Easier to validate (a single core)
  - Easier to scale (the same validated design replicated multiple times)
  - Due to these reasons, future "many-core" processors will look like this
  - Examples: Tilera TILE*, Intel 48-core chip [Howard et al., ISSCC '10]
- **Design focus shifts from cores to "uncores"**
  - **Network on-chip (NoC)**: bus, crossbar, ring, 2D/3D mesh, …
  - **Memory hierarchy**: **caches**, coherence mechanisms, memory controllers, …
  - **System-on-chip components** like network MAC, cypher, …
- **Note also:**
  - In this design, aggregate cache capacity increases as we add more tiles
  - Some tiles may be active while others are inactive

L2 cache design issues

- L2 caches occupy a major portion (prob. the largest) on chip
- **Capacity vs. latency**
  - Off-chip access (still) expensive
  - On-chip latency can grow; with switched on-chip networks, **we have NUCA (Non-Uniform Cache Architecture)**
  - **Private cache vs. shared cache or a hybrid scheme?**
- **Quality of service**
  - Interference (capacity/bandwidth)
  - **Explicit/implicit capacity partitioning**
- **Manageability, fault and yield**
  - Not all caches have to be active
  - Hard (irreversible) faults + process variation + soft faults

Private cache vs. shared cache



☑ Short hit latency (always local)
☒ High on-chip miss rate
☒ Long miss resolution time (i.e., are there replicas?

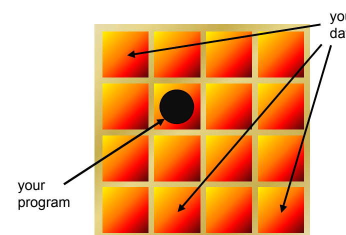
☑ Low on-chip miss rate
☑ Straightforward data location
☒ Long average hit latency
☒ Uncontrolled interference

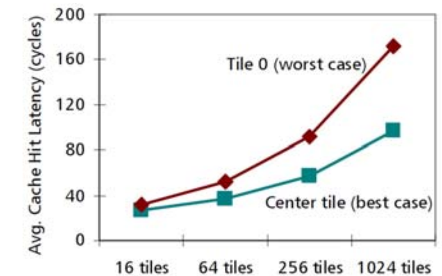
Historical perspective

- Private designs were there
  - Multiprocessors based on microprocessors
- Shared cache clustering [Nayfeh and Olukotun, ISCA '94]
- Multicores with private design more straightforward
  - Earlier AMD chips and Sun Microsystems' UltraSPARC processors
- Intel, IBM, and Sun pushed for shared caches (e.g., Xeon products, POWER-4/5/6, Niagara*)
  - A variety of workloads perform relatively well
- **More recently, private caches re-gain popularity**
  - We have more transistors
  - Interference is easily controlled
  - Match well with on-/off-chip (shared) L3 $$ [Oh et al., ISVLSI '09]

Shared $$ issue 1: NUCA

- A compromise design
  - Large monolithic shared cache: conceptually simple but very slow
  - Smaller cache slices w/ disparate latencies [Kim et al., ASPLOS '02]
    - Interleave data to all available slices (e.g., IBM POWER-*)
- **Problems**
  - Blind data distribution with no provisions for latency hiding
  - As a result, simple NUCA performance is not scalable



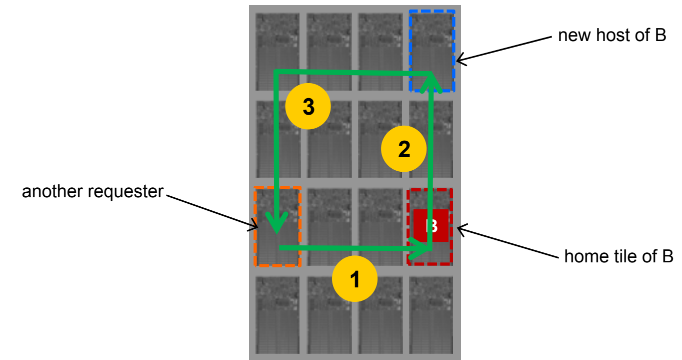


Improving program-data distance

- **Key idea: replicate or migrate**
- **Victim replication** [Zhang and Asanović, ISCA '05]
  - Victims from private L1 cache are "replicated" to local L2 bank
  - Essentially, local L2 banks incorporate large victim caching space
  - A natural hybrid scheme that takes advantages of shared & private $$
- **Adaptive Selective Replication** [Beckmann et al., MICRO '06]
  - Uncontrolled replication can be detrimental (shared cache degenerates to private cache)
  - Based on benefit and cost, control replication level
  - Hardware support quite complex
- **Adaptive Controlled Migration** [Hammoud, Cho, and Melhem, HiPEAC '09]
  - Migrate (rather than replicate) a block to minimize average access latency to this block based on Manhattan distance
  - **Caveat:** book-keeping overheads

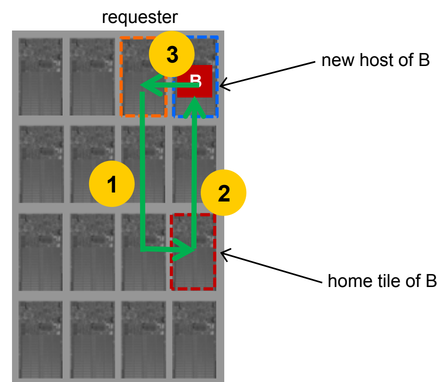
3-way communication

- Once you migrate a cache block away from its home, how do you locate it? [Hammoud, Cho, and Melhem, HiPEAC '09]



3-way communication

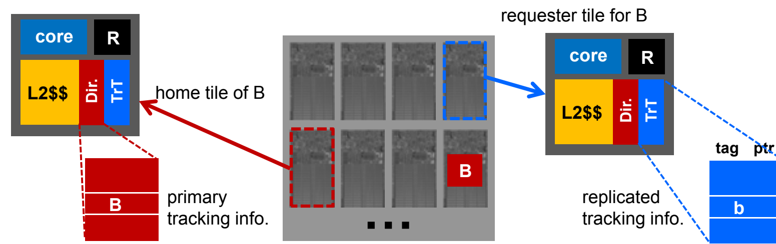
- Once you migrate a cache block away from its home, how do you locate it? [Hammoud, Cho, and Melhem, HiPEAC '09]



Cache block location strategies

- **Always-go-home**
  - **"Go to the home tile (directory) to find the tracking information"**
  - Lots of 3-way cache-to-cache transfers
- **Broadcasting**
  - **"Don't remember anything"**
  - Expensive
- **Caching of tracking information at potential requesters**
  - **"Remember (locally) where previously accessed blocks are"**
  - Quickly locate the target block locally (after first miss)
  - Need maintenance (coherence) of distributed information
  - We proposed and studied one such technique [Hammoud, Cho, and Melhem, HiPEAC '09]

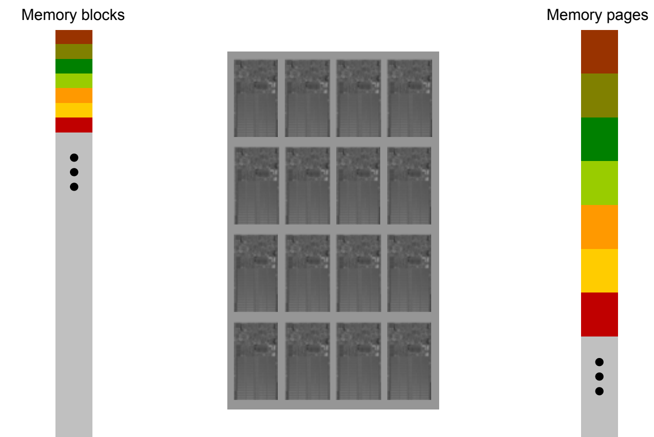
Caching of tracking information



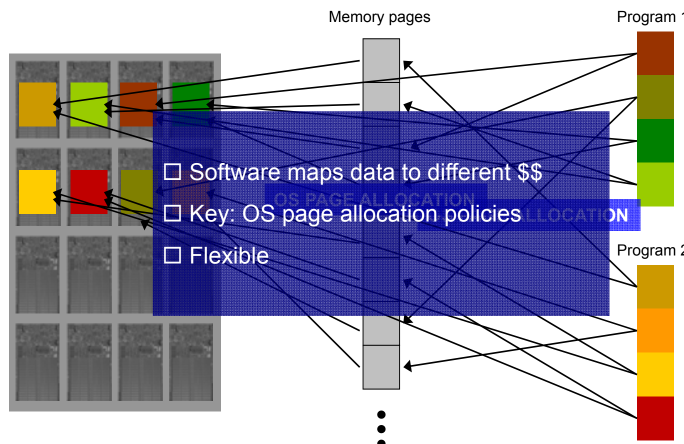
- **Primary tracking information at home tile**
  - Shows where the block has been migrated to
  - Keeps track of who has copied the tracking information (bit vector)
- **Tracking information table (TrT)**
  - A copy of the tracking information (only one pointer)
  - Updated as necessary (initiated by the directory of the home tile)

A flexible data mapping approach

- Key idea: What if we distribute memory pages to cache banks instead of memory blocks? [Cho and Jin, MICRO '06]

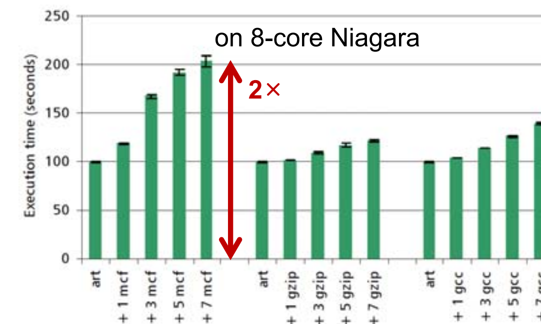


Observation



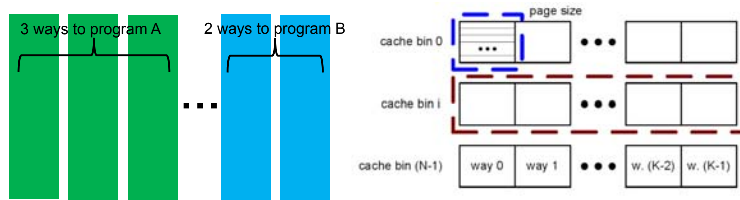
Shared $$ issue 2: interference

- Co-scheduled programs compete for cache capacity freely ("capitalism")
- Well-behaving programs get damages if another program vie for more cache capacity w/ little reuse (e.g., streaming)
- Performance loss due to interference hard to predict



Explicit/implicit partitioning

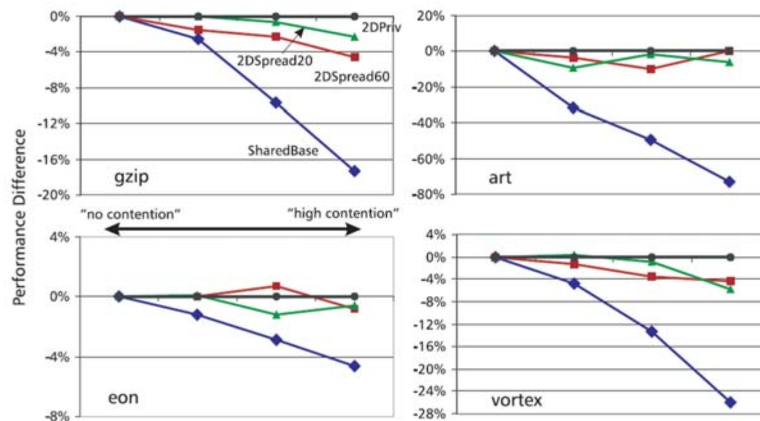
- Strict partitioning based on system directives
  - E.g., program A gets 768KB and program B 256KB
  - System must allow a user to specify partition sizes [Iyer, ICS ‘04][Guo et al., MICRO ‘07]
- Architectural/system mechanisms
  - Way partitioning (e.g., 3 vs. 2 ways) [Suh et al., ICS ‘01][Kim et al., PACT ‘04]
  - Bin/bank partitioning: maps pages to cache bins (no hardware support needed) [Liu et al., HPCA ‘04][Cho, Jin and Lee, RTCSA ‘07][Lin et al., HPCA ‘08]



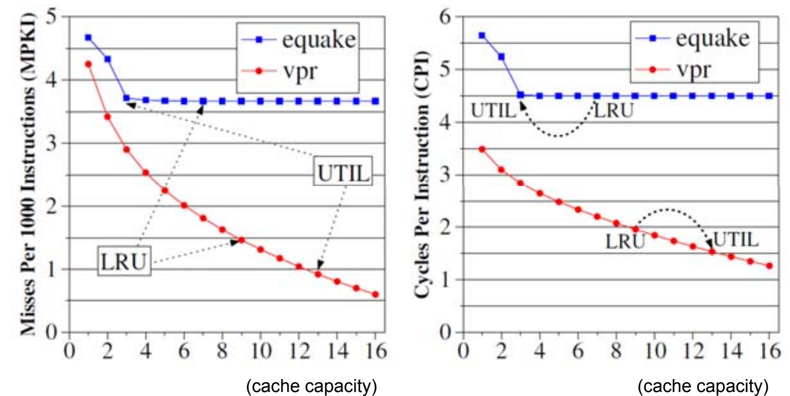
Explicit/implicit partitioning

- Strict partitioning based on system directives
  - E.g., program A gets 768KB and program B 256KB
  - System must allow a user to specify partition sizes [Iyer, ICS ‘04][Guo et al., MICRO ‘07]
- Architectural/system mechanisms
  - Way partitioning (e.g., 5 vs. 3 ways) [Suh et al., ICS ‘01][Kim et al., PACT ‘04]
  - Bin/bank partitioning: maps pages to cache bins (no hardware support needed) [Liu et al., HPCA ‘04][Cho, Jin and Lee, RTCSA ‘07][Lin et al., HPCA ‘08]
- Implicit partitioning based on utility (“utilitarian”)
  - Way partitioning based on marginal gains [Suh et al., ICS ‘01][Qureshi et al., MICRO ‘06]
  - Bank level [Lee, Cho and Childers, HPCA ’10]
  - Bank + way level [Lee, Cho and Childers, HPCA ’11]

Explicit partitioning



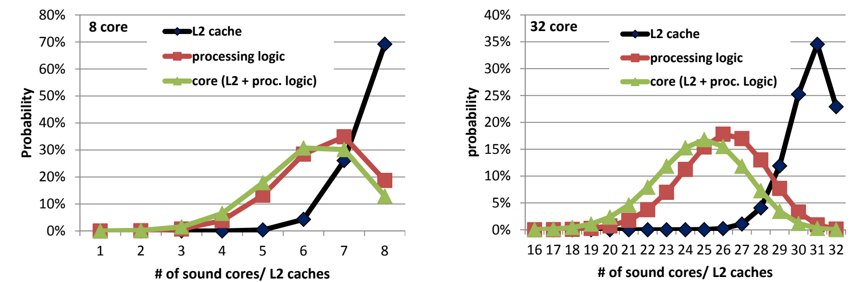
Implicit partitioning



Private $$ issue: capacity borrowing

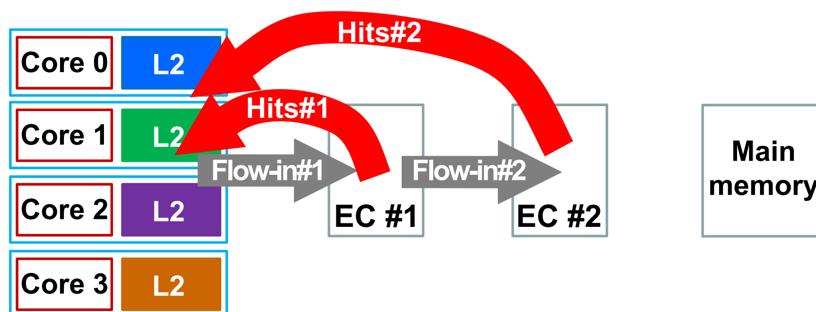
- The main drawback of private caches is fixed (small) capacity
- Assume that we have an underlying mechanism for enforcing correctness of cache access semantics;
- **Key idea: borrow (or steal) capacity from others**
  - How much capacity do I need and opt to borrow?
  - Who may be a potential capacity donor?
  - Do we allow performance degradation of a capacity donor?
- **Cooperative Caching (CC)** [Chang and Sohi, ISCA '06]
  - Stealing coordinated by a centralized directory (is not scalable)
- **Dynamic Spill Receive (DSR)** [Qureshi, HPCA '09]
  - Each node is a spiller or a receiver
  - On eviction from a spiller, randomly choose a receiver to copy the data into; then, for data access later, use broadcasting
- **StimulusCache** [Lee, Cho, and Childers, HPCA '10]
  - It's likely we'll have "excess cache" in the future
  - Intelligently share the capacity provided by excess caches based on utility

Core disabling *uneconomical*



- **Many unused (excess) L2 caches exist**
- **Problem exacerbated with many cores**

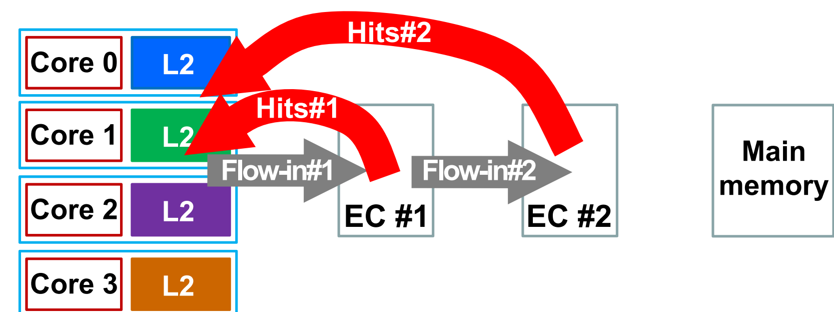
Dynamic sharing policy



Flow-in#N: # data blocks flowing to EC#N

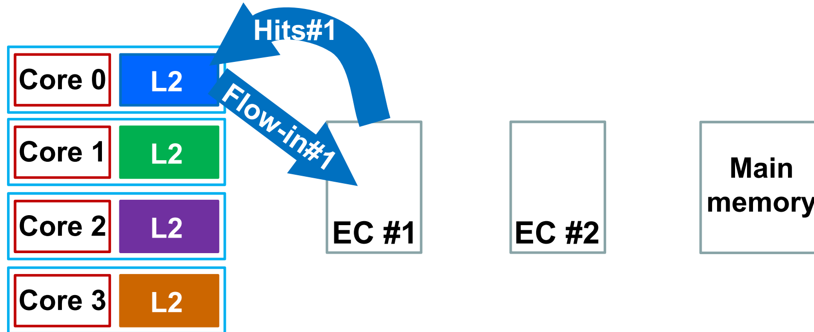
Hit#N: # data blocks hit at EC#N

Dynamic sharing policy



Hit/Flow-in ↑ ➔ More ECs
Hit/Flow-in ↓ ➔ Less ECs

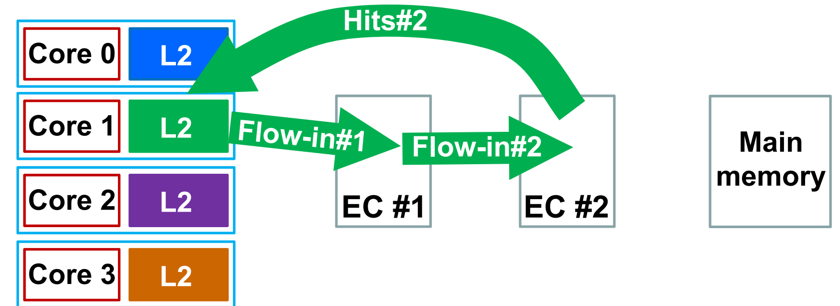
Dynamic sharing policy



Core 0: At least 1 EC
No harmful effect on EC#2

➔ Allocate 2 ECs

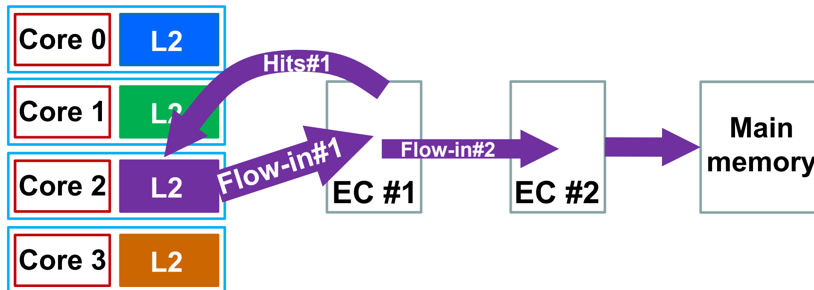
Dynamic sharing policy



Core 1: At least 2 ECs

➔ Allocate 2 ECs

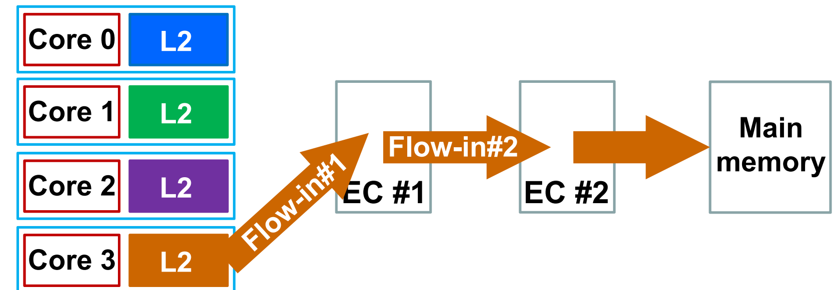
Dynamic sharing policy



Core 2: At least 1 EC
Harmful effect on EC#2

➔ Allocate 1 EC

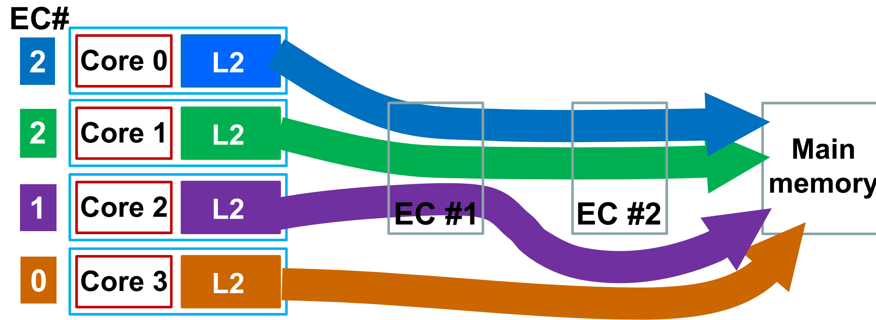
Dynamic sharing policy



Core 2: No benefit with ECs

➔ Allocate 0 EC

Dynamic sharing policy



Maximized capacity utilization

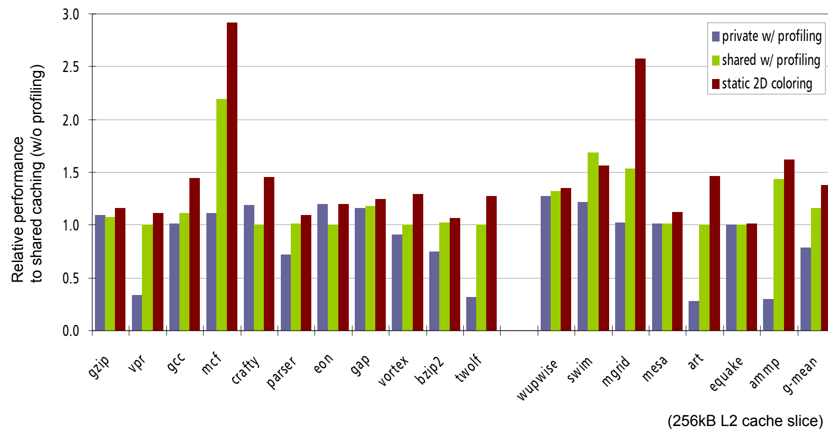
Minimized capacity interference

Summary: priv. vs. shared caching

- For a small- to medium-scale multicore processor, shared caching appears good (for its simplicity)
  - A variety of workloads run well
  - **Care must be taken about interferences**
  - **Because of NUCA effects, this approach is not scalable!**
- For larger-scale multicore processors, private cache schemes appear more promising
  - Free performance isolation
  - Less NUCA effect
  - Easier fault isolation
  - **Capacity stealing a necessity**

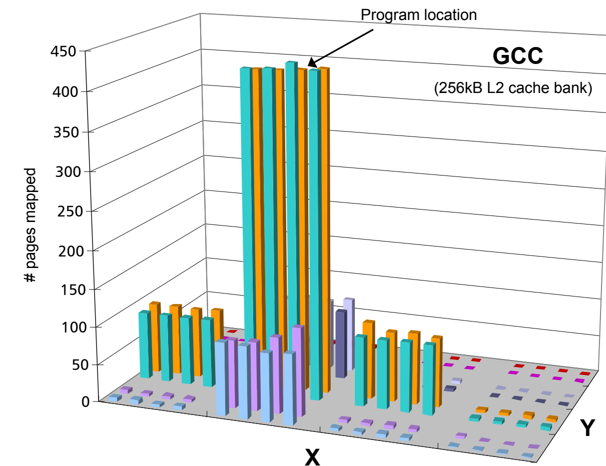
Summary: hypothetical caching

- **Tackle both miss rate and locality through judicious data mapping!** [Jin and Cho, ICPP '08]

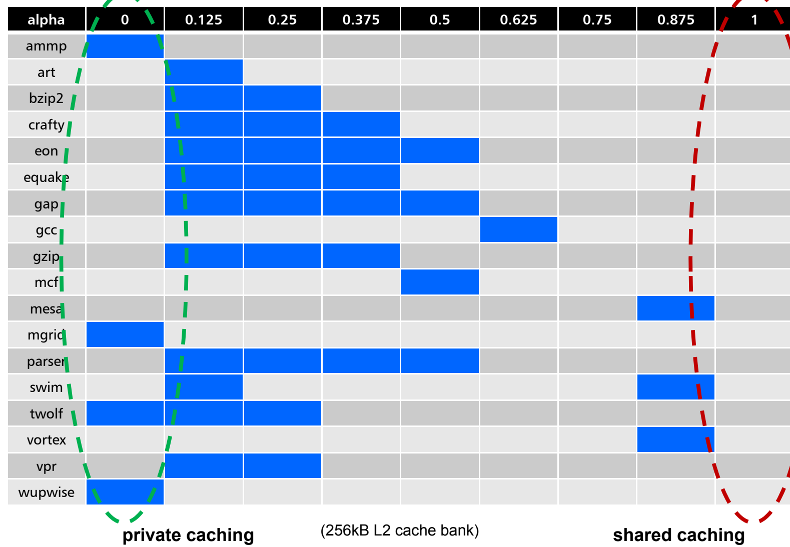


(256kB L2 cache slice)

Summary: hypothetical caching



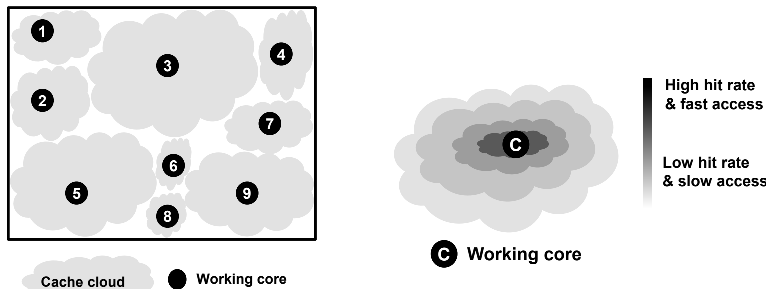
Summary: hypothetical caching



In the remainder of this talk

- **CloudCache**
  - Private cache based scheme for large-scale multicore processors
  - **Capacity borrowing at way/bank level & distance awareness**

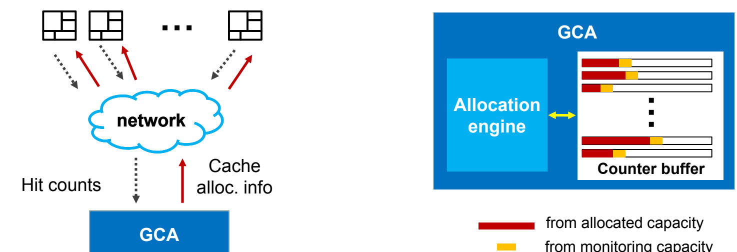
A 30,000-feet snapshot



- Each cloud is an exclusive private cache for a working core
- Clouds are built with nearby cache ways organized in a chain
- Clouds are built in a two-step process: **Dynamic global partitioning** and **cache cloud formation**

Step 1: Dynamic global partitioning

- **Global capacity allocator** (**GCA**) runs a partitioning algorithm periodically using "hit counts" information
  - Each working core sends GCA *hit counts* from "allocated capacity" and additional "monitoring capacity" (of 32 ways)
- Our partitioning considers both utility [Qureshi & Patt, MICRO '06] and QoS
- This step computes how much capacity to give to each core



Step 2: Cache cloud formation

1. Local L2 cache first
2. Threads w/ larger capacity demand first
3. Closer L2 banks first
4. Allocate capacity as much as possible

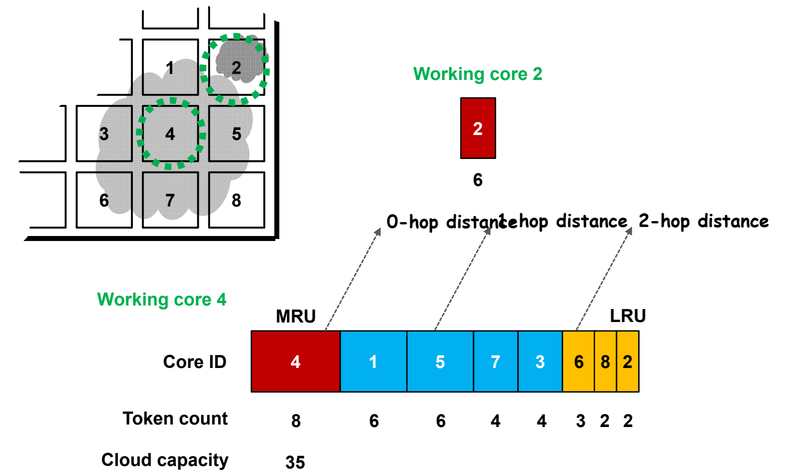
Repeat!

	Goal	Capacity to allocate
0	2.75	0.75
1	1.25	0.25

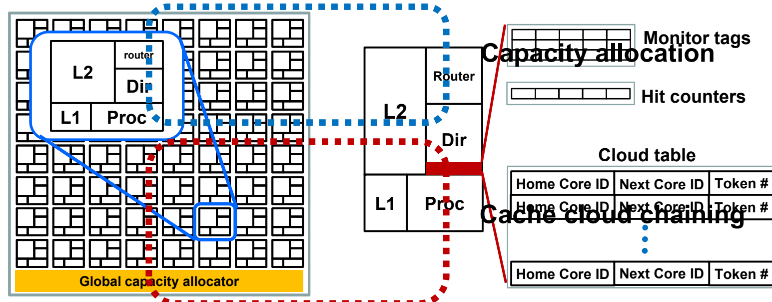
0 1
2 3

- **Allocation is done only for sound L2 banks**

Cache cloud chaining example

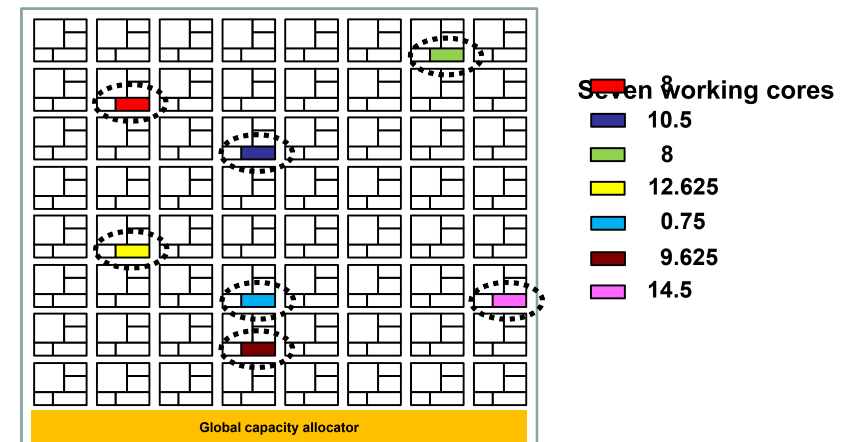


Architectural support

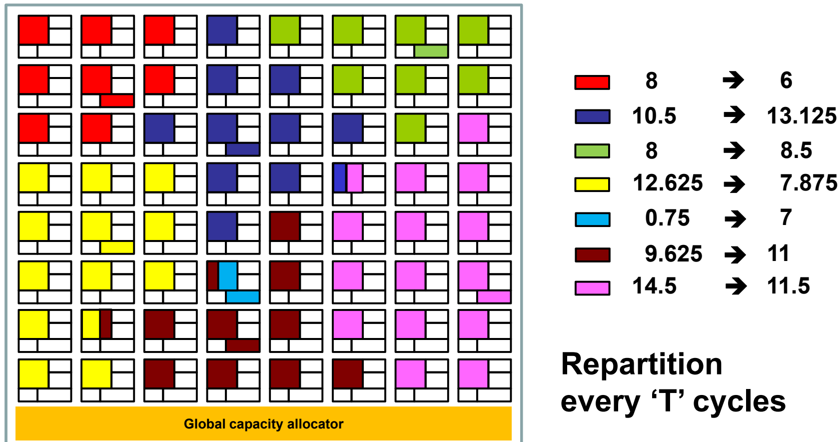


- Cloud table resembles StimulusCache's NECP
- Additional hardware is per-way hit counters and monitor tags
- We do not need core ID tracking information per block

Capacity allocation example

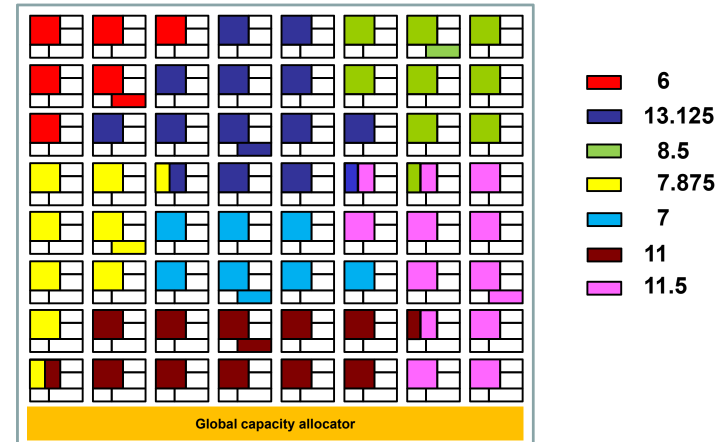


Capacity allocation example



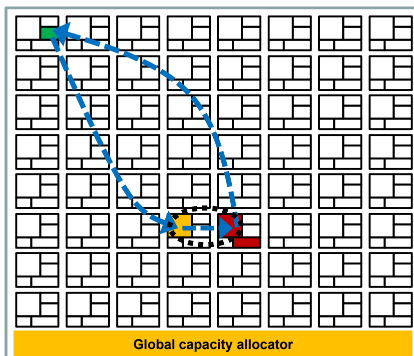
Repartition every ‘T’ cycles

Capacity allocation example



CloudCache is fast?

- Remote L2 access has 3-way communication

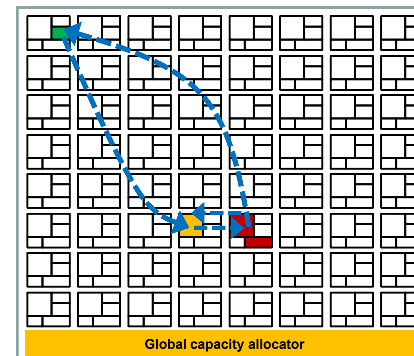


(1) Directory lookup
(2) Request forwarding
(3) Data forwarding

Distance-aware cache cloud formation tackles only (3)!

Limited target broadcast (LTB)

- Make common case “super fast” and rare case “not so fast”

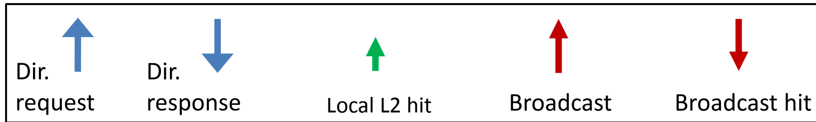


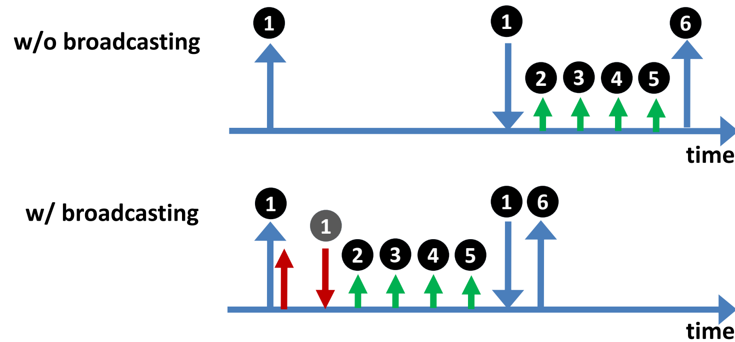
Private data:
Limited target broadcast
⇒ No wait for directory lookup

Shared data:
Directory-based coherence

Private data ≫ Shared data

LTB example

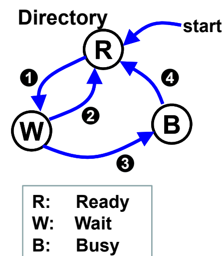




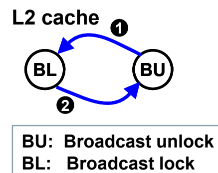
When is LTB used?

- **Shared data:** Access ordering is managed by directory
  - LTBP is NOT used
- **Private data:** Access ordering is not needed
  - Fast access with broadcast first, then notify directory
- **Race condition?**
  - When there is an access request for private data from a non-owner core before directory is updated

LTB protocol (LTBP)



	Input	Action
1	Request from other cores	Broadcast lock request
2	Ack from the owner	Process non-owner request
3	Nack from the owner	
4	Request from the owner	



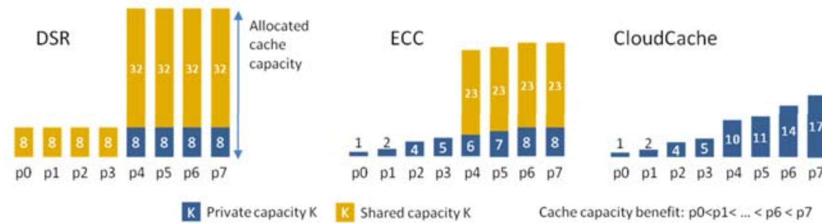
	Input	Action
1	Broadcast lock request	Ack
2	Invalidation	

Quality of Service (QoS) support

- $\text{QoS} = \text{Maximum performance degradation}$
- $F_S = \text{sampling factor,}$ 32 in this work
- $K = \text{private cache capacity,}$ 8 in this work
- $n = \text{currently allocated capacity}$
- Base cycle: Cycles (time) with private cache (estimated)
- $$\text{Base cycle} = \begin{cases} \text{current cycle} + \sum_{i=K+1}^{n} Hits(i) \times F_S \times miss\ latency & (n > K) \\ \text{current cycle} & (n = K) \\ \text{current cycle} - \sum_{i=n+1}^{K} Hits(i) \times F_S \times miss\ latency & (n < K) \end{cases}$$
- Estimated cycle: Cycles with cache capacity 'j'
- $\text{Estimated cycle(j)} = \text{Base cycle} + \sum_{i=j+1}^{K} Hits(i) \times F_S \times miss\ latency$
- $C_{QoS} = \text{Min(j)} \quad \text{s.t. Estimated cycle(j)}/(1 + \text{QoS}) < \text{Base cycle}$

DSR vs. ECC vs. CloudCache

- **Dynamic Spill Receive** [Quresh, HPCA '09]
  - Node is either spiller or receiver depending on memory usage
  - Spiller nodes randomly "spill" evicted data to a receiver node
- **Elastic Cooperative Caching** [Herrero et al., ISCA '10]
  - Each node has private area and shared area
  - Nodes with high memory demand can spill evicted data to shared area in a randomly selected node



Experimental setup

- **TPTS** [Lee et al., SPE '10, Cho et al., ICPP '08]
  - 64-core CMP with 8×8 2D mesh, 4-cycles/hop
  - Core: Intel's ATOM-like two-issue in-order pipeline
  - Directory-based MESI protocol
  - Four independent DRAM controllers, four ports/controller
  - DRAM with Samsung DDR3-1600 timing
- **Workloads**
  - SPEC2006 (10B cycles)
    - High/medium/low based on MPKI for varying cache capacity
  - PARSEC (simlarge input set)
    - 16 threads/application

Impact of global partitioning

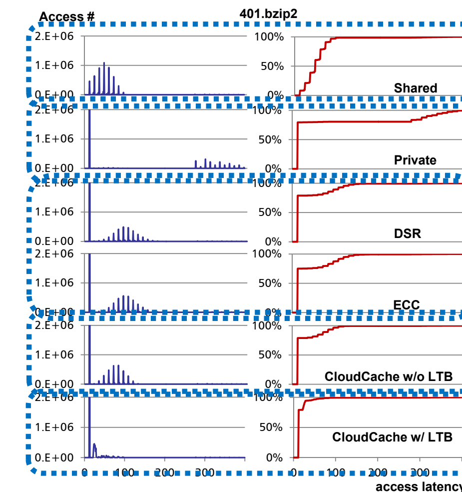
Impact of global partitioning

Impact of global partitioning

Impact of global partitioning

Impact of global partitioning

L2 cache access latency



Shared:
widely spread latency

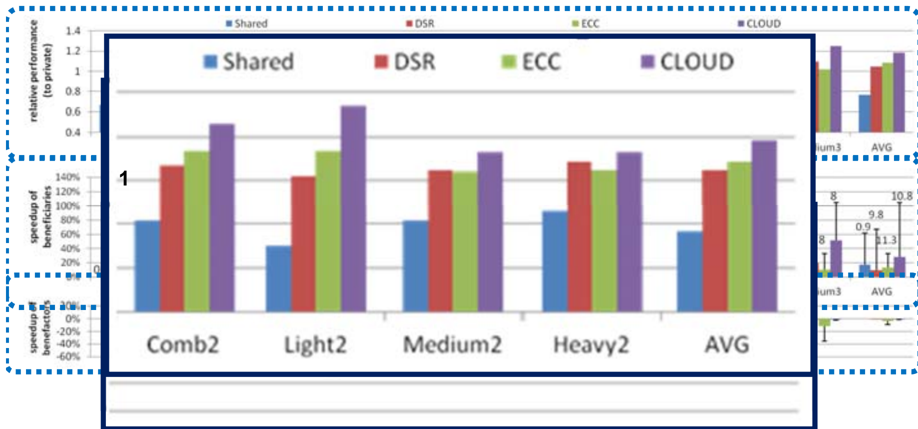
Private:
fast local access + off-chip access

DSR & ECC:
fast local access + widely spread latency

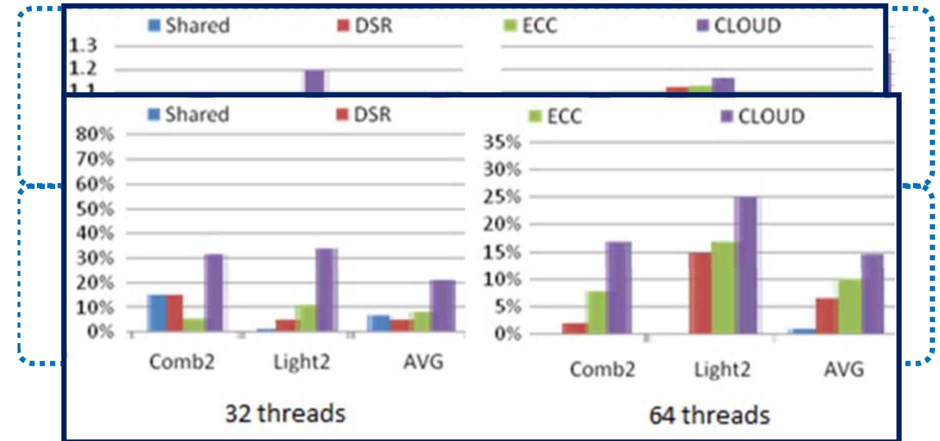
CloudCache w/o LTB
fast local access + narrowly spread latency

CloudCache w/ LTB
fast local access + fast remote access

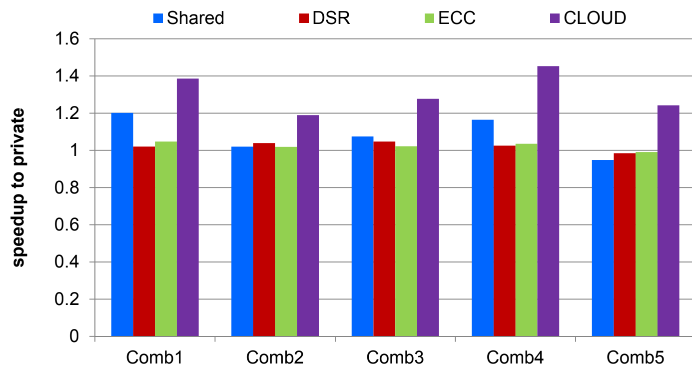
16 threads



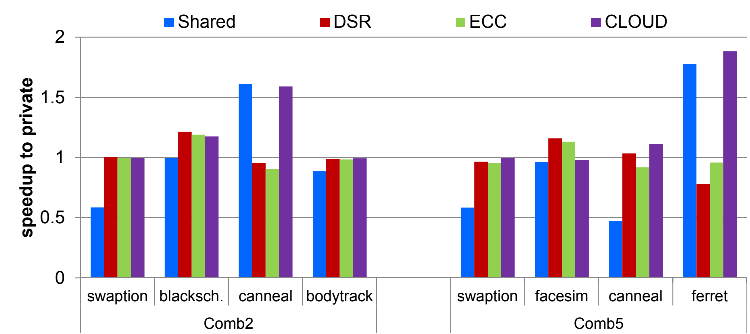
32 and 64 threads



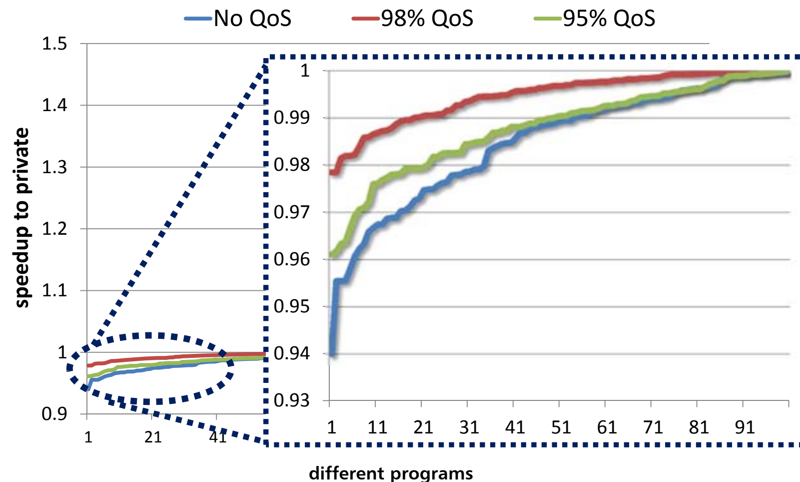
PARSEC



PARSEC



Effect of QoS enforcement



CloudCache summary

- Future processors will carry many more cores and cache resources and future workloads will be heterogeneous
- Low-level caches must become more scalable and flexible
- We proposed **CloudCache** w/ three techniques:
  - Global "private" capacity allocation to eliminate interference and to minimize on-chip misses
  - Distance-aware data placement to overcome NUCA latency
  - Limited target broadcast to overcome directory lookup latency
- Proposed techniques synergistically improve performance
- Still, we find that global capacity allocation the most effective
- QoS is naturally supported in CloudCache

Our multicore cache publications

- Cho and Jin, "**Managing Distributed, Shared L2 Caches through OS-Level Page Allocation**," *MICRO 2006*. Best paper nominee.
- Cho, Jin and Lee, "**Achieving Predictable Performance with On-Chip Shared L2 Caches for Manycore-Based Real-Time Systems**," *RTCSA 2007*. Invited paper.
- Hammoud, Cho and Melhem, "**ACM: An Efficient Approach for Managing Shared Caches in Chip Multiprocessors**," *HiPEAC 2008*.
- Hammoud, Cho and Melhem, "**Dynamic Cache Clustering for Chip Multiprocessors**," *ICS 2008*.
- Jin and Cho, "**Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches**," *ICPP 2008*.
- Oh, Lee, Lee, and Cho, "**An Analytical Model to Study Optimal Area Breakdown between Cores and Caches in a Chip Multiprocessor**," *ISVLSI 2009*.
- Jin and Cho, "**SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors**," *PACT 2009*.
- Lee, Cho and Childers, "**StimulusCache: Boosting Performance of Chip Multiprocessors with Excess Cache**," *HPCA 2010*.
- Hammoud, Cho and Melhem, "**Cache Equalizer: A Placement Mechanism for Chip Multiprocessor Distributed Shared Caches**," *HiPEAC 2011*.
- Lee, Cho and Childers, "**CloudCache: Expanding and Shrinking Private Caches**," *HPCA 2011*.