Cache Coherence Protocols for Chip Multiprocessors - II

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Context

- Thus far
  - —chip multiprocessors
  - —hardware threading strategies
  - —future microprocessor issues and trends
  - —cache coherence and victim replication
- Today: more cache coherence protocols for current and future chip multiprocessors

Today's References

- **ASR: Adaptive selective replication for CMP caches.** B. Beckmann, M. Marty, and D. Wood. MICRO-39, Dec. 2006.
- **Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors.** E. Herrero, Jose Gonzalez, and Ramon Canal. ICSA '10, Saint-Malo, France, June 2010.
- **Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory.** Xiangyao Yu and Srinivas Devadas. 2015. In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15). IEEE Computer Society, Washington, DC, USA, 227-240.

Tiled Architectures

Organize multicore processor as a set of tiles

- **Each tile**
  - —one or more cores
  - —some private cache
  - —some shared cache
- **Issues**
  - —wire delay: tens of clock cycles across chip
  - —long delays data accesses by a core is often in other tiles

Adaptive Selective Replication

Adaptive Selective Replication

- **Demonstrate that cache replication policies should focus on shared read-only blocks.**
  - **for commercial workloads, shared read-only blocks account for 42-71% of L2 requests, but consume—without replication—only 10-21% of the L2 capacity**
  - **replicating relatively few shared read-only blocks significantly reduces L2 access time due to their tremendous locality: the top 3% of shared read-only blocks account for 70% of requests**
  - **aggressive replication degrades some workloads' performance due to increased off-chip misses**
- **Selective Probabilistic Replication (SPR)**
  - **no designated home node for a cache line**
  - **assumes private L2 caches and selectively limits replication on L1 evictions**
  - **on an L1 cache eviction, SPR writes a shared block back to its local L2 if**
    - **(i) the block was already allocated in the local L2**
    - **(ii) the replication policy (below) allocates a new block**
    - **otherwise, uses a ring writeback to merge the block with an existing remote L2 copy**
  - **on L1 cache writebacks, SPR uses probabilistic filtering to decide when to replicate a block**

B. Beckmann, M. Marty, and D. Wood. ASR: Adaptive selective replication for CMP caches. MICRO-39, Dec. 2006.

Table 2: SPR Replication Levels

Level	0	1	2	3	4	5
Probability	0	1/64	1/16	1/4	1/2	1
Threshold	0	4	16	64	128	256

Elastic Cooperative Caching

Background: Caching Strategies

- Static partitioning of cache resources
  - —private: unable to give all cache lines to a single thread
  - —shared: all threads have same priority and compete for lines
    - threads may interfere: a needy thread can evict another's data
- Dynamic repartitioning of cache resources per thread
  - —desirable to reduce off-chip traffic
  - —flavors
    - software-based dynamic reconfiguration (OS manages resource)

      most organizations divide resources into independent sets for QoS

      e.g. cooperative cache partitioning

      partition resources in both space and time

      multiple time-sharing partitions to manage current capacity
    - hardware dynamic reconfiguration

      use performance counters to measure benefit of increasing cache size for each thread

Motivation

Streaming applications don't exploit cache close to processor

- **—access lots of data that won't be reused**
- **—when combined with other applications**
  - **may needlessly evict blocks that might be reused**

Elastic Cooperative Caching Goals

- Want memory hierarchy that exercises intelligent control
  - —distributes resources fairly
  - —exploits differences between applications
- Should be managed by hardware rather than software
- Should provide elastic tradeoff between
  - —low latency of private caches
  - —low off-chip miss rate of shared caches

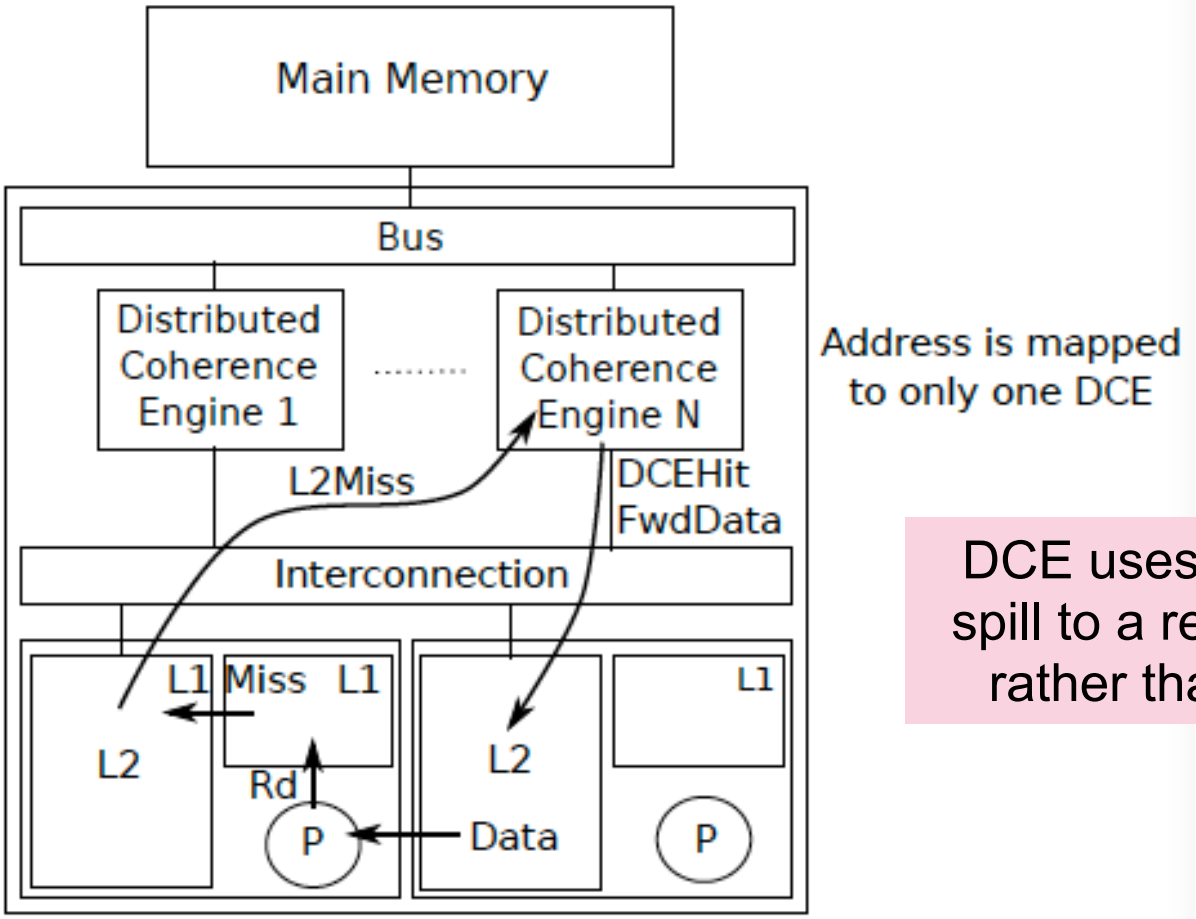
Elastic Cooperative Caching

- No centralized structure
- First distributed cache repartitioning mechanism
  - —uses only local info
  - —distributed cache partitioning units
    - support redistribution of cache resources
    - operate autonomously with only local information
- Shared/private caches + repartitioning unit
  - —shared cache: stores evicted blocks from active private regions
  - —private regions: allow big local private caches to meet appl needs

Distributed Coherence Engine (DCE)

DCEs are directory caches responsible for coherence over part of the address space

—uses a "home node" mapping scheme for addresses



DCE uses 1-FWD: spill to a remote tile rather than evict

Figure credit: Elastic Cooperative Caching ..., E. Herrero et al. ICSA, June 2010..

ElasticCC Tile Structure

- **Several independent L2 cache memories**
  - **—shared vs. private regions compete for the cache space**
- **Private regions**
  - **—store blocks evicted from local L1**
- **Shared regions**
  - **—store blocks spilled from neighboring caches**
- **Operation**
  - **—shared data is replicated into private regions**
  - **—shared region stores only unique blocks**

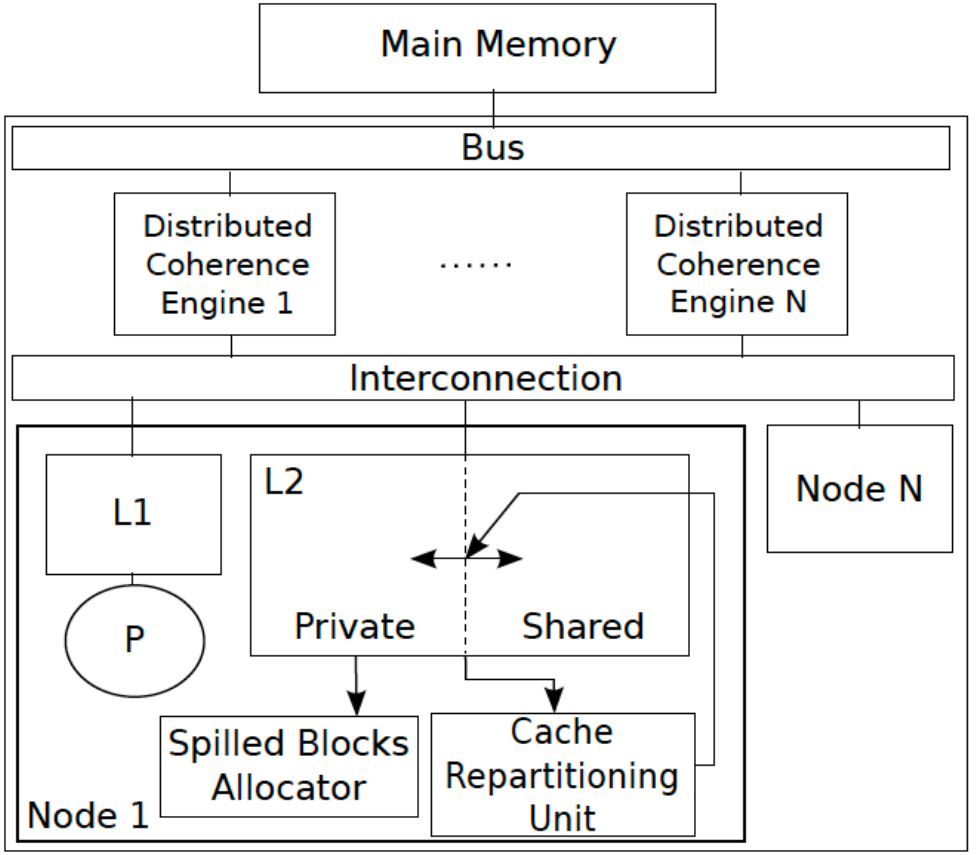
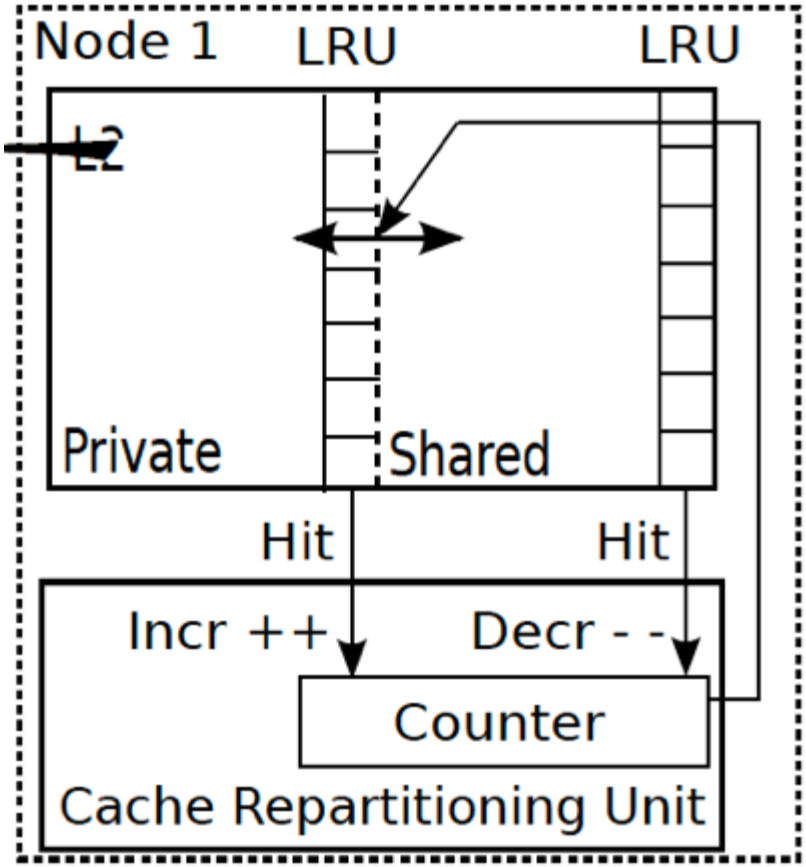


Figure credit: Elastic Cooperative Caching ..., E. Herrero et al. ICSA, June 2010..

ElasticCC Repartitioning Unit



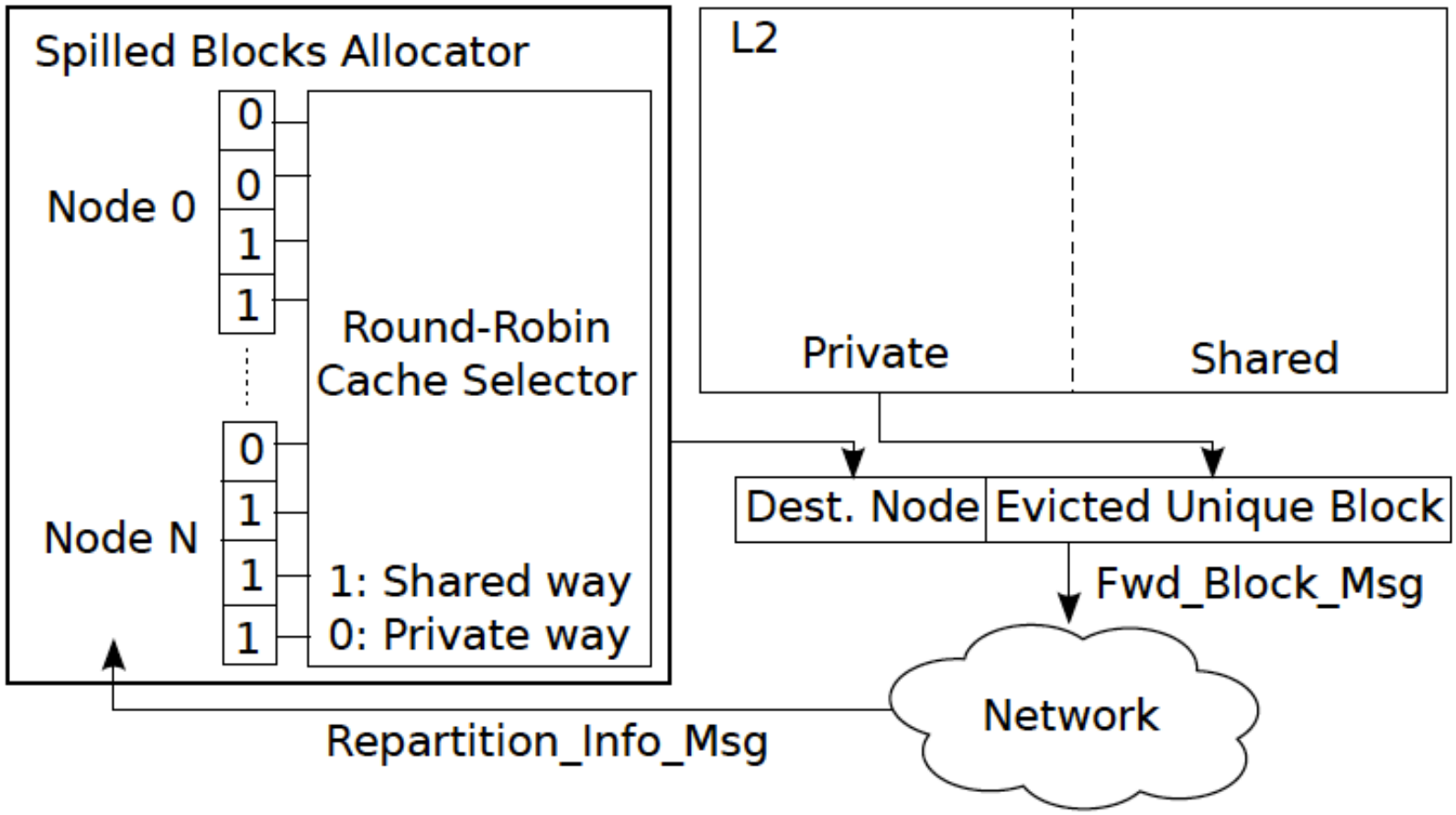

```
If Private_LRU_Hit then
   Increase Counter
EndIf
If Shared_LRU_Hit then
   Decrease Counter
EndIf
If Repartition_Cycle then
   If Counter > Upper_Threshold then
      Add_Private_Way
      Send Repartition_Info_Msg
   ElseIf Counter < Lower_Threshold then
      Add_Shared_Way
      Send Repartition_Info_Msg
   EndIf
   Clear Counter
EndIf
```

- Each tile has its own repartioning unit
- Each time a cache is repartitioned, broadcast partition info

Figure credit: Elastic Cooperative Caching ..., E. Herrero et al. ICSA, June 2010..

ElasticCC Spilled Block Allocator

- One spilled block allocator per tile
- Uses partition info from each tile to send more evicted blocks to tiles with more shared cache
- Can use stale partitioning info



ElasticCC on a Chip

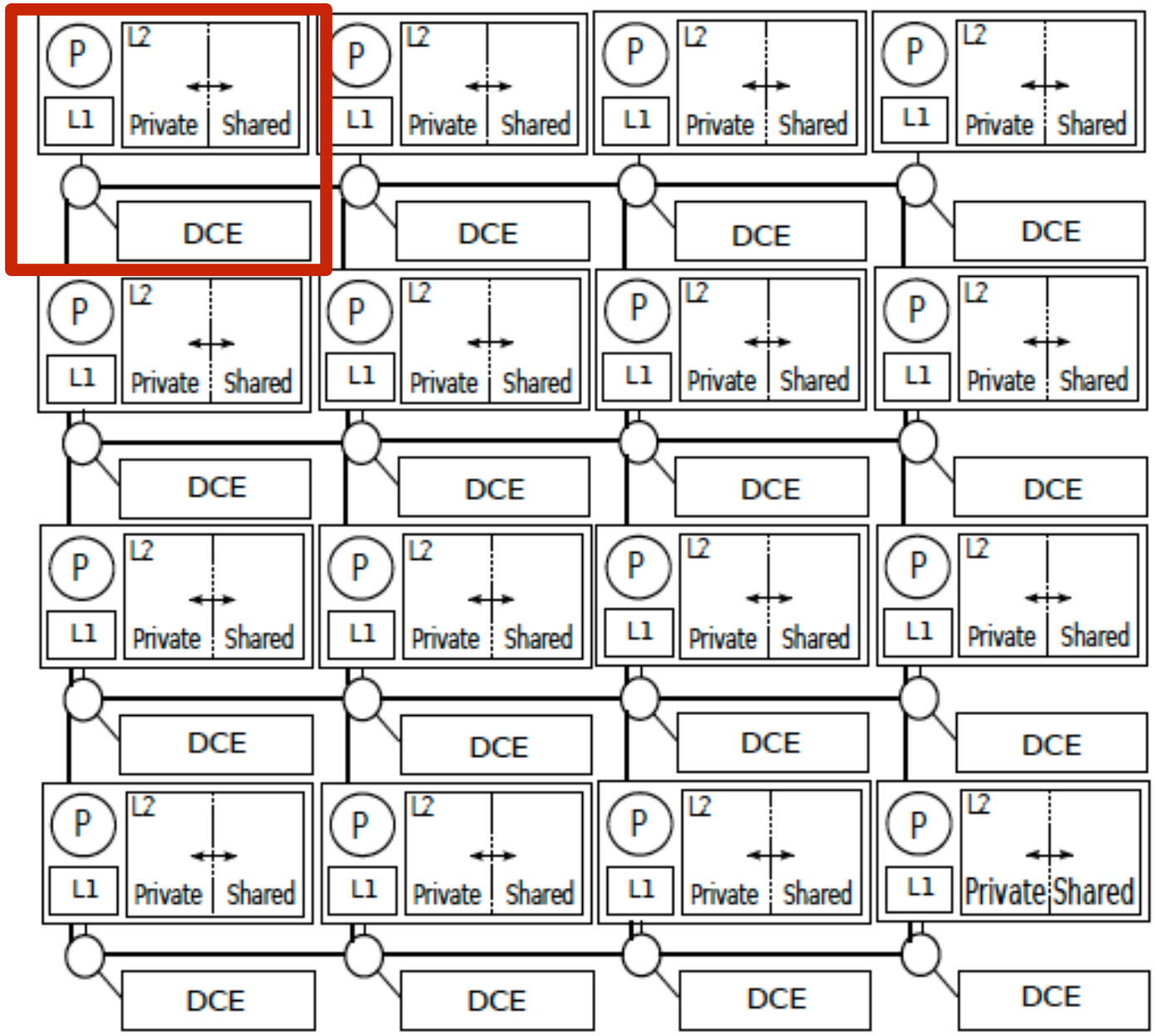


Figure credit: Elastic Cooperative Caching ..., E. Herrero et al. ICSA, June 2010..

Application Classes

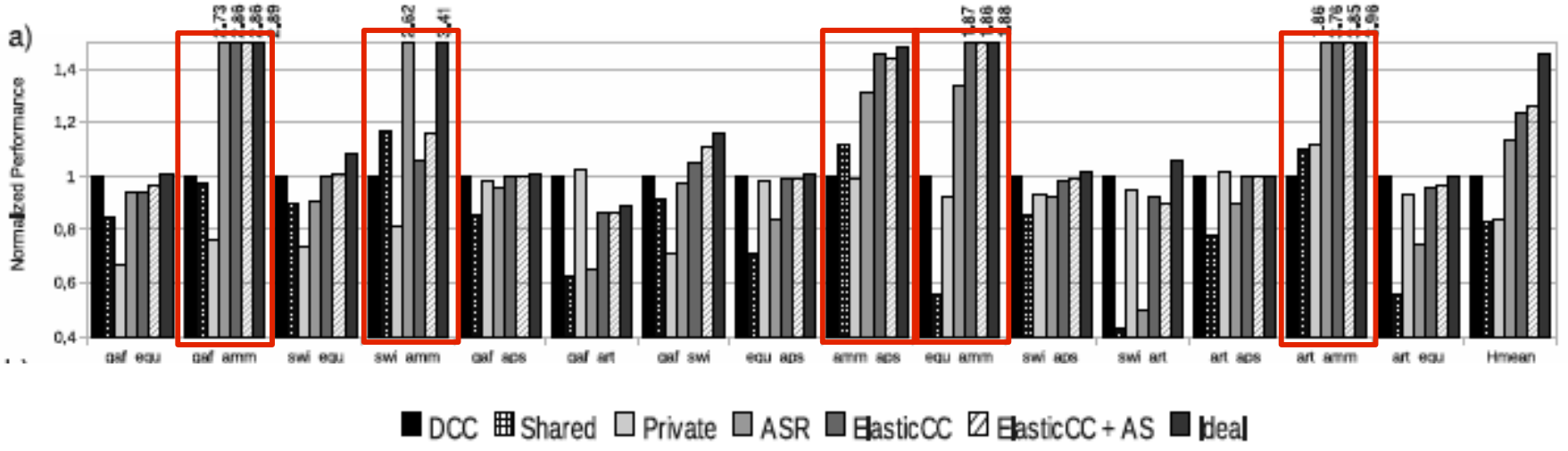
- **Saturating utility**
  - —small working set that fits in cache
  - —characterized by improving performance until working set fits
  - —e.g. equake
- **Low utility**
  - —intense use of memory but no reuse
  - —e.g. Gafort
- **Shared high utility**
  - —several threads share a large number of blocks
  - —e.g. ammp
- **Private high utility**
  - —benefit from larger memory hierarchy, but do not share data
  - —e.g. swim

Adaptive Spilling Based on Appl Type

Type	Working Set size	Sharing	Local Reuse	Private Cache size	Spilling
Saturating Utility	Small	H/L	H/L	Small	No
Low Utility	Big	Low	Low	Small	No
Shared High Utility	Big	High	H/L	Small	Yes
Private High Utility	Big	Low	High	Big	Yes

- **Saturating utility applications don't need extra space**
- **Low utility applications don't have reuse, so forbid them to spill to remote tiles**
- **High utility applications can benefit from spilling**
  - **—PHU - allow spilling when 75% of cache (6 ways) is private**
  - **—SHU - detect by high cache-to-cache transfers**
    - **have DCE track block sharing with one bit per block**
    - **spill only shared blocks**

ElasticCC Evaluation: Performance



Higher is better

Performance with AMMP (shared high utility)

- **Elastic Cooperative Caching outperforms**
  - **—private caches by 52%**
  - **—distributed shared cache by 53%**
  - **—Distributed Cooperative Caching by an average of 27%**
    - **like ECC, but static partitioning of private and shared**
  - **—distributed Adaptive Selective Replication (ASR) by 12%**
    - **monitors workload and replicates only when benefit (lower L2 hit latency) estimated to outweigh costs (more L2 misses)**

Note: ASR claims 12% better than victim replication

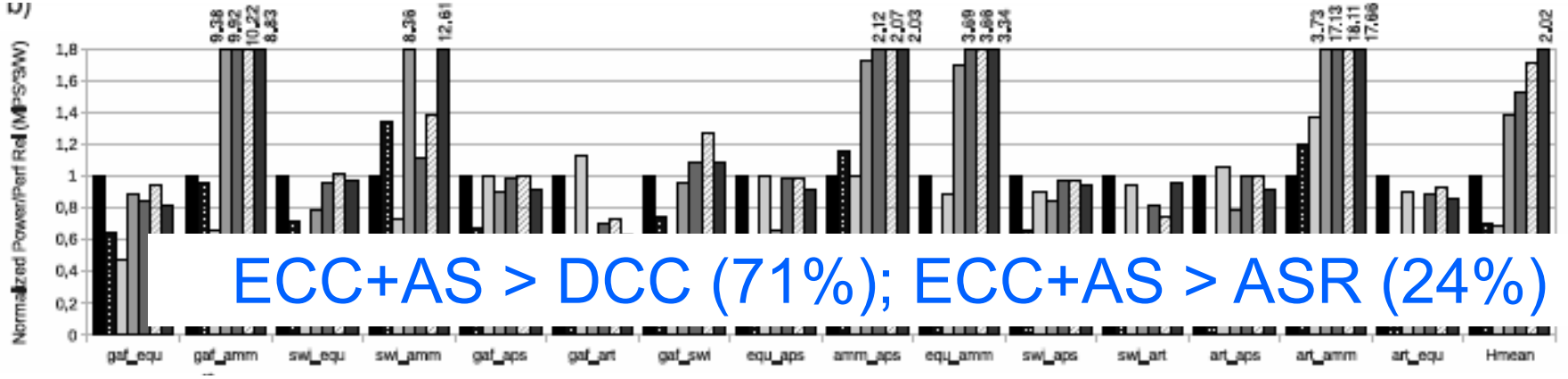
Figure credit: Elastic Cooperative Caching ..., E. Herrero et al. ICSA, June 2010..

Performance Graph Notes

- **Performance improvement is highly dependent on the characteristics of all the applications being executed simultaneously**
- **Performance improvements can only come from High Utility benchmarks and in the other cases the adaptive mechanism must find the lowest amount of dedicated resources that does not degrade performance**

ElasticCC Evaluation: Energy, Misses

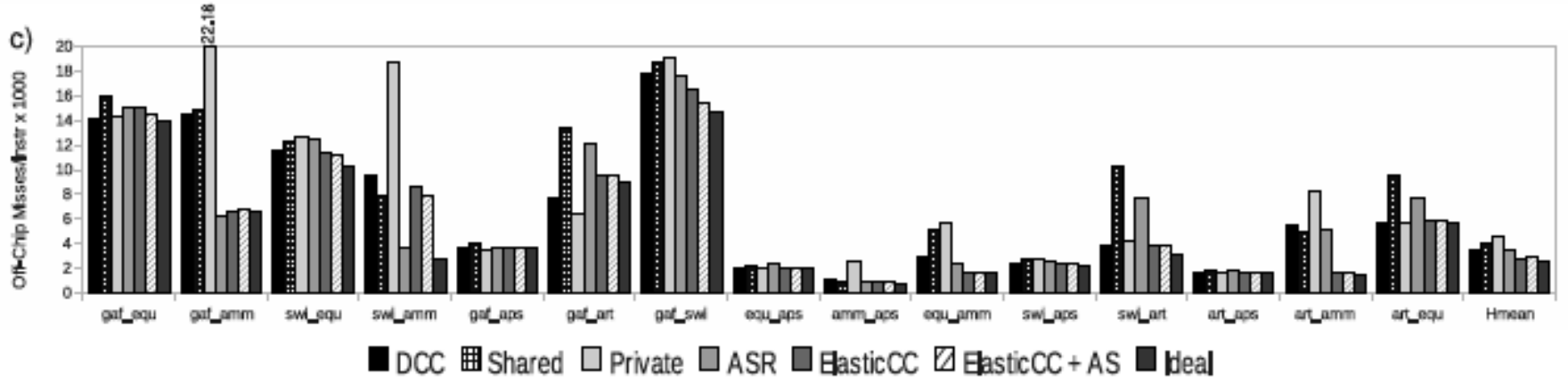
Energy Efficiency



ECC+AS > DCC (71%); ECC+AS > ASR (24%)

Higher is better

Cache Misses



Lower is better

ECC+AS < DCC (18.6%); ECC+AS < ASR (16.4%)

Figure credit: Elastic Cooperative Caching ..., E. Herrero et al. ICSA, June 2010..

Summary of ElasticCC

- Outperforms
  - —private caches by 52%
  - —distributed shared cache by 53%
  - —other proposed approaches
    - Distributed Cooperative Caching by 27%
    - Active Selective Replication by 12%
- Reduces number of off chip misses vs.
  - —distributed cooperative caching by 19%
  - —active selective replication by 16%
- Increases energy efficiency vs.
  - —Distributed Cooperative Caching by 71%
  - —Active Selective Replication by 24%
    - by avoiding reallocation of non-reused cache blocks

Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory

Tardis Motivation

- Correctness of shared memory systems depends upon memory consistency model
  - —defines legal interleavings of memory ops by different actors
- Shared-memory systems depend on cache coherence
- Coherence protocol: important for performance and scalability
- Well-known coherence protocols discussed last class
  - —snooping: requires broadcast communication medium
  - —directory: maintain a list of sharers
- Concerns
  - —broadcast doesn't scale as number of nodes increases
  - —storing sharer information doesn't scale either
  - —long waits for invalidation acknowledgements

Sequential Consistency

A system consisting of one or more processors with multiple cores is sequentially consistent if both of the following conditions hold:

1. **The result of any execution is the same as if the operations of all the cores were executed in some sequential order**
2. **The operations of each individual core appear in this sequence in the order specified by its program**

L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," Computers, IEEE Transactions on, vol. 100, no. 9, pp. 690–691, 1979.

Tardis Overview

- Goal
  - —simple scalable protocol
  - —equivalent in performance to directory protocol
- Idea
  - —express memory consistency model by enforcing global memory order using timestamp counters that represent logical and physical time
- Advantages
  - —satisfies sequential consistency
  - —no requirement of globally synchronized clock
    - – unlike prior timestamp coherence schemes
  - —no multicast/broadcast support
    - – unlike prior directory coherence schemes
  - —storage of timestamp + owner ID is O(log N) for N cores
    - – no O(N) sharer information as for directory presence bits
    - – insight: writer can jump ahead to a time when sharer copies have expired and perform write without violating sequential consistency

Tardis Timestamp Ordering

- **Directory protocols enforce global memory order ($<_M$) through physical time order, e.g. for two operations X and Y on memory location A**

$$X <_m Y \implies X <_{pt} Y$$

- **Tardis: break the correlation between global memory order and the physical time order for *write after read* (WAR) dependencies while maintaining the correlation for *write after write* (WAW) and *read after write* (RAW) dependencies**

$$S_1(A) <_m S_2(A) \implies S_1(A) <_{pt} S_2(A) \qquad \text{(WAW)}$$

$$S(A) <_m L(A) \implies S(A) <_{pt} L(A) \qquad \text{(RAW)}$$

$$L(A) <_m S(A) \not\Longrightarrow L(A) <_{pt} S(A) \qquad \text{(WAR)}$$

Tardis Global Memory Order

- **Tardis global memory order: combination of physical time and logical timestamp order, i.e., physi-logical time order**
  - **—AKA physiological time order**

$$X <_m Y \quad := \quad X <_{ts} Y \ or \ (X =_{ts} Y \ and \ X <_{pt} Y)$$

- **Operations without dependency (e.g., two concurrent read operations) or with obvious relative ordering (e.g., accesses to private data from the same core)**
  - **—can share the same timestamp**
  - **—global memory order is implicitly expressed using the physical time order**

Sequential Consistency Rules with Tardis

- **Rule 1:** $X <_p Y \Rightarrow X <_{ts} Y \vee (X =_{ts} Y \wedge X <_{pt} Y)$
  - **assuming in-order commits** $X <_p Y \Rightarrow X <_{pt} Y$.
    Tardis only needs to guarantee $X <_p Y \Rightarrow X \leq_{ts} Y$.
    - **namely, operations from the same processor have monotonically increasing timestamps in program order**
- **Rule 2:**
  - **guarantee that a load observes the correct store in the global memory order as defined by** $X <_m Y \;:=\; X <_{ts} Y \; or \; (X =_{ts} Y \; and \; X <_{pt} Y)$
  - **Correct store is the latest store – either the one with the largest logical timestamp or the latest physical time among the stores with the largest logical timestamp**

Tardis without Private Cache

- **Core timestamp**
  - —PTS - program timestamp: timestamp of last operation in program order
    - not equivalent to processor clock
      - not incremented every cycle
      - not globally synchronized
- **Cache line**
  - —RTS - read timestamp: largest timestamp among all loads of cache line so far
  - —WTS - write timestamp: timestamp of latest store to cache line
- **Invariant**
  - —a cache line's data must be valid between current WTS and RTS

Timestamp Management w/out Private Cache

Request Type	Load Request	Store Request
Timestamp Operation	$pts \Leftarrow Max(pts, wts)$ $rts \Leftarrow Max(pts, rts)$	$pts \Leftarrow Max(pts, rts + 1)$ $wts \Leftarrow pts$ $rts \Leftarrow pts$

Tardis with Private Cache

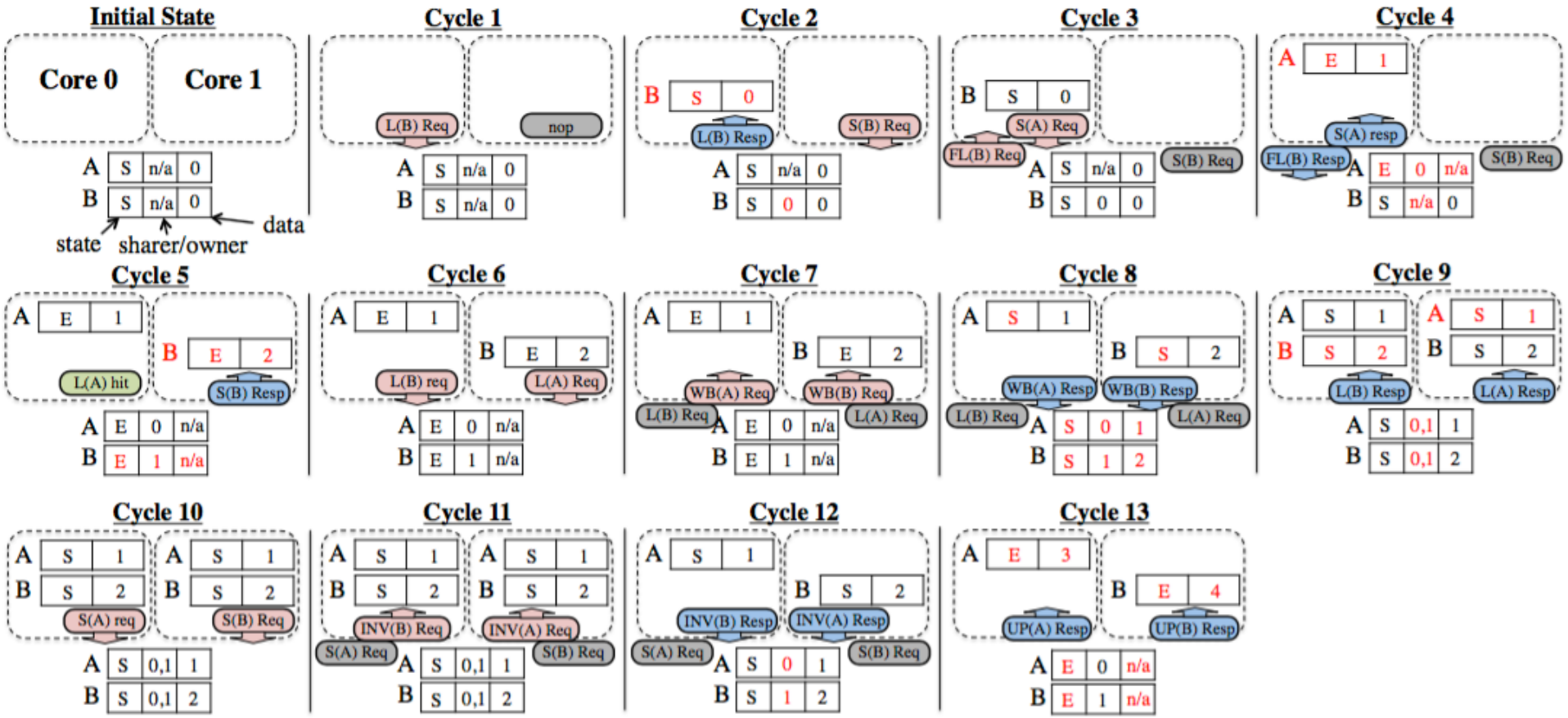
- **Timestamp manager timestamp (MTS)**
  - —maximal read timestamp of all cache lines mapped to this timestamp manager but evicted to DRAM
    - when evicting cache line c: MTS = MAX(c.RTS, MTS)
- **Timestamp reservation**
  - —allows a load to reserve a cache line in private cache for a period of logical time (the lease)
  - —end timestamp of a reservation is stored in RTS
  - —cache line can be read until timestamp expires (PTS > RTS)
  - —read of a cache line with an expired lease -> request timestamp manager to extend the lease
- **Exclusive ownership**
  - —modified cache line can be exclusively cached in private cache
    - timestamp manager records line is in exclusive state and owner (log N bits)

State Transitions for Private Cache

States	Core Event			Network Event		
	Load	Store	Eviction	SH_REP or EX_REP	RENEW_REP or UPGRADE_REP	FLUSH_REQ or WB_REQ
Invalid	send SH_REQ to TM M.wts⇐0, M.pts⇐pts	send EX_REQ to TM M.wts⇐0		Fill in data **SH_REP** D.wts⇐M.wts D.rts⇐M.rts state⇐Shared **EX_REP** D.wts⇐M.wts D.rts⇐M.rts state⇐Excl.	**RENEW_REP** D.rts⇐M.rts **UPGRADE_REP** D.rts⇐M.rts state⇐Excl.	
Shared *pts* ≤ *rts*	Hit pts⇐Max(pts, D.wts)	send EX_REQ to TM M.wts⇐D.wts	state⇐Invalid No msg sent.			
Shared *pts* > *rts*	send SH_REQ to TM M.wts⇐D.wts, M.pts⇐pts					
Exclusive	Hit pts⇐Max(pts, D.wts) D.rts⇐Max(pts, D.rts)	Hit pts⇐Max(pts, D.rts+1) D.wts⇐pts D.rts⇐pts	state⇐Invalid send FLUSH_REP to TM M.wts⇐D.wts, M.rts⇐D.rts			**FLUSH_REQ** M.wts⇐D.wts M.rts⇐D.rts send FLUSH_REP to TM state⇐Invalid **WB_REQ** D.rts⇐Max(D.rts, D.wts+lease, reqM.rts) M.wts⇐D.wts M.rts⇐D.rts send WB_REP to TM state⇐Shared

States	SH_REQ	EX_REQ	Eviction	DRAM_REP	FLUSH_REP or WB_REP
Invalid	Load from DRAM			Fill in data D.wts⇐mts D.rts⇐mts state⇐Shared	
Shared	D.rts⇐Max(D.rts, D.wts+lease, reqM.pts+lease) **if reqM.wts=D.wts** send RENEW_REP to requester M.rts⇐D.rts **else** send SH_REP to requester M.wts⇐D.wts M.rts⇐D.rts	M.rts⇐D.rts state⇐Excl. **if reqM.wts=D.wts** send UPGRADE_REP to requester **else** M.wts⇐D.wts send EX_REP to requester	mts⇐Max(mts, D.rts) Store data to DRAM if dirty state⇐Invalid		
Exclusive	send WB_REQ to the owner M.rts⇐reqM.pts+lease	send FLUSH_REQ to the owner			Fill in data D.wts⇐M.wts, D.rts⇐M.rts state⇐Shared

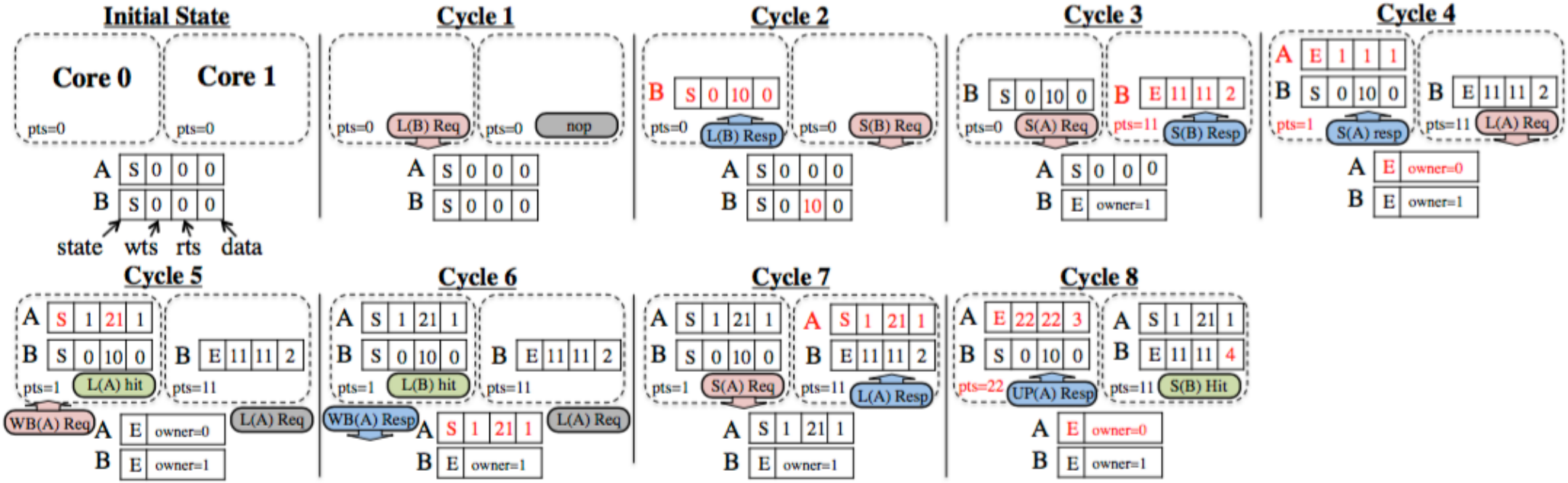
Directory Coherence Example



Listing 2. The case study parallel program

```
[Core 0]          [Core 1]
 L(B)              nop
 A = 1             B = 2
 L(A)              L(A)
 L(B)              B = 4
 A = 3
```

Tardis Coherence Example



Listing 2. The case study parallel program

```
[Core 0]          [Core 1]
 L(B)              nop
 A = 1             B = 2
 L(A)              L(A)
 L(B)              B = 4
 A = 3
```

Tardis vs. Directory Coherence

- **Invalidation**
  - **—directory**
    - **must send invalidate msgs to all sharers and await acks**
  - **—Tardis**
    - **no invalidation**
    - **exclusive ownership can be immediately returned without waiting**
    - **timestamps guarantee sequential consistency**
- **Eviction**
  - **—directory**
    - **message sent from private cache to directory where sharer stored**
    - **when evicted from LLC, all private copies must be invalidated**
  - **—Tardis**
    - **no sharer information maintained; no invalidation required**
    - **after eviction from LLC, copies in private caches can exist and be accessed**

Tardis vs. Directory Coherence

- Data renewal
  - —directory
    - load hit only requires data to exist in private cache
  - —Tardis
    - cache line lease may have expired
    - renew request sent to timestamp manager
      - incurs extra latency and network traffic
      - optimization: speculative execution
        - assume cache line with expired lease has valid data
        - if renewal fails, roll back speculative computation
- Compress timestamps using base + delta scheme
  - —only store deltas in cache line
  - —rebase whenever any delta rolls over

Tardis Advantages

- **Scalability**
  - —store only timestamps per cache line and owner ID in LLC
    - owner ID and timestamps can share bits in LLC
      - when owner ID needs to be stored, cache line is exclusively owned and manager does not maintain the timestamps
- **Simplicity**
  - —derived from definition of sequential consistency
  - —timestamps explicitly represent global memory order
    - easier to argue correctness
  - —no multicast/broadcast invalidations
  - —no acknowledgment collection
  - —fewer transient states than a directory protocol

System Configuration

System Configuration	
Number of Cores	N = 64 @ 1 GHz
Core Model	In-order, Single-issue

Memory Subsystem	
Cacheline Size	64 bytes
L1 I Cache	16 KB, 4-way
L1 D Cache	32 KB, 4-way
Shared L2 Cache per Core	256 KB, 8-way
DRAM Bandwidth	8 MCs, 10 GB/s per MC
DRAM Latency	100 ns

2-D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link)
Flit Width	128 bits

Tardis Parameters	
Lease	10
Self Increment Period	100 cache accesses
Delta Timestamp Size	20 bits
L1 Rebase Overhead	128 ns
L2 Rebase Overhead	1024 ns

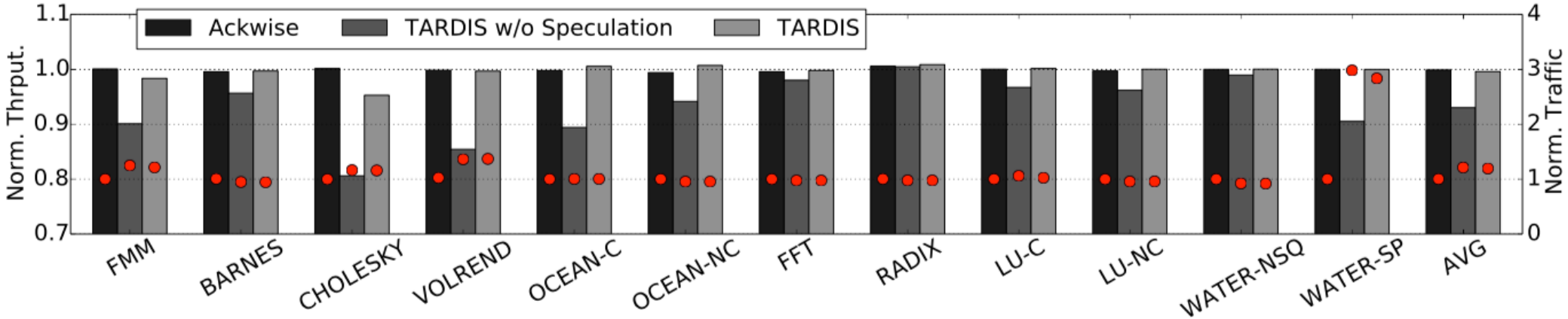
Tardis Timestamp Statistics

Benchmarks	Ts. Incr. Rate (cycle / timestamp)	Self Incr. Perc.
FMM	322	22.5%
BARNES	155	33.7%
CHOLESKY	146	35.6%
VOLREND	121	23.6%
OCEAN-C	81	7.0%
OCEAN-NC	85	5.6%
FFT	699	88.5%
RADIX	639	59.3%
LU-C	422	1.4%
LU-NC	61	0.1%
WATER-NSQ	73	12.8%
WATER-SP	363	29.1%
AVG	263	26.6%

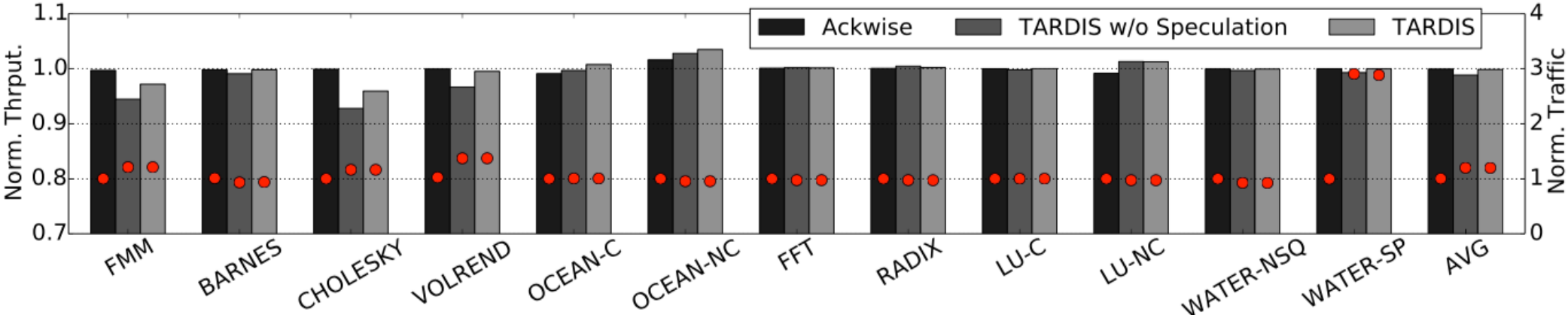
Tardis Performance

64 In-order Cores

network traffic



64 Out-of-order Cores

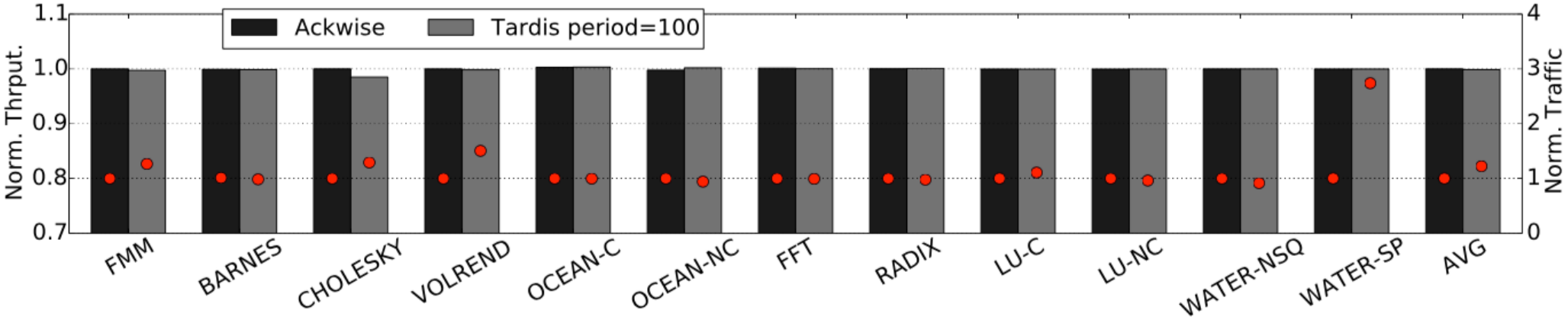


Ackwise: maintain a limited number of sharers and broadcasts invalidations to all cores when the number of sharers exceeds the limit (Tile-gx family of multicore processors," http://www.tilera.com)

Baseline is MSI: full info about all sharers

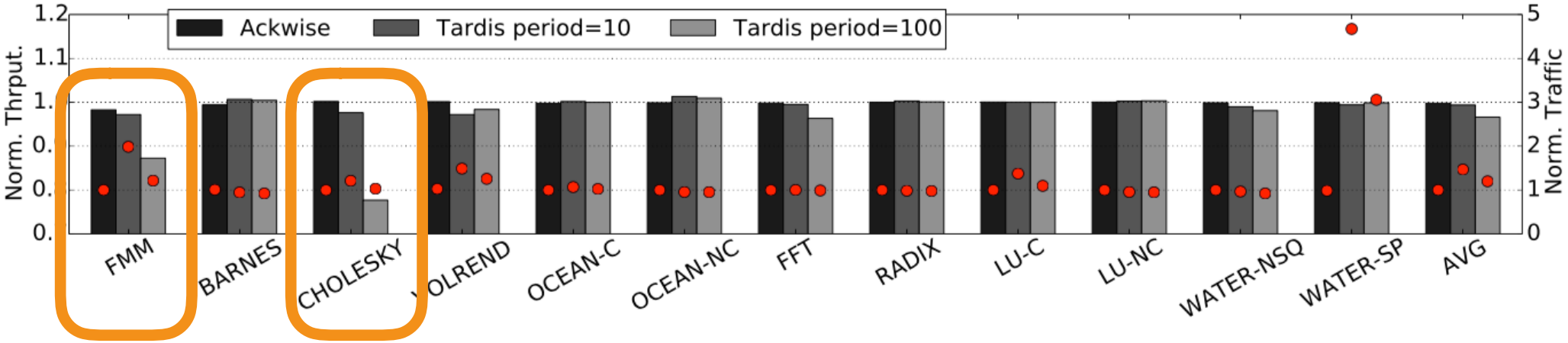
Tardis Performance

network traffic



(a) 16 Cores

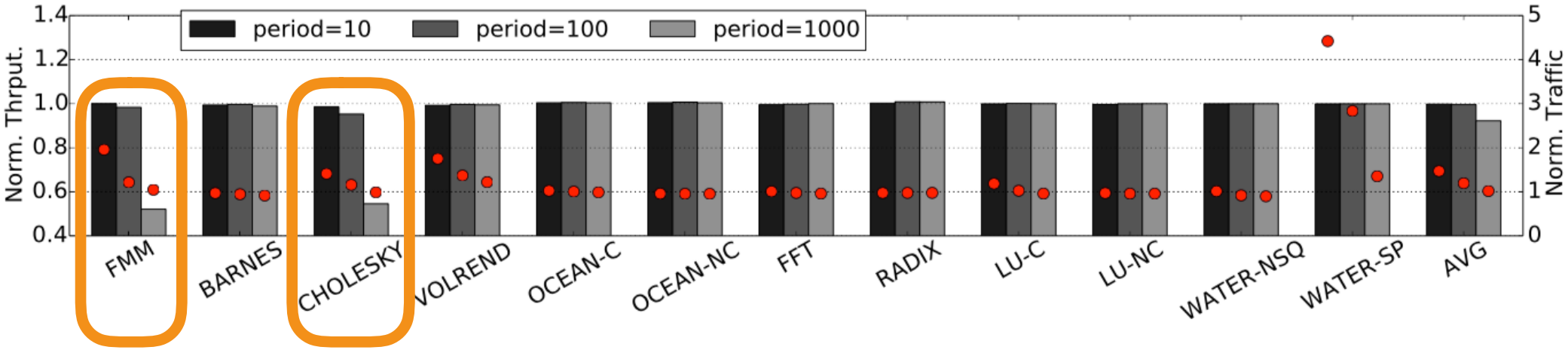
use spin waiting



(b) 256 Cores

Ackwise: maintain a limited number of sharers and broadcasts invalidations to all cores when the number of sharers exceeds the limit (Tile-gx family of multicore processors," http://www.tilera.com)

Tardis Performance vs. Increment Period



spin wait on stale values with longer self-increment period

Tardis Conclusions

- **Match baseline performance for directory protocol**
- **Better scalability for large number of cores**

Additional References

- **Efficient Timestamp-Based Cache Coherence Protocol for Many-Core Architectures.** **Yuan Yao, Guanhua Wang, Zhiguo Ge, Tulika Mitra, Wenzhi Chen, and Naxin Zhang. 2016. In *Proceedings of the 2016 International Conference on Supercomputing* (ICS '16). ACM, New York, NY, USA, , Article 19 , 13 pages. DOI=http://dx.doi.org/10.1145/2925426.2926270**