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
    – simultaneous multithreading
    – fine-grain multithreading
  — future microprocessor issues and trends
  — cache coherence and victim replication

• Today: more cache coherence protocols for current and future chip multiprocessors
Today’s References


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
Elastic Cooperative Caching
Background: Caching Strategies

- **Static partitioning of cache resources**
  - *shared*: may have inter-thread interferences
  - *private*: unable to give all cache space to a single thread

- **Dynamic repartitioning**
  - *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
DCEs are directory caches responsible for coherence over part of the address space.

—uses a “home node” mapping scheme for addresses

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.
Each tile has its own repartitioning unit
Each time a cache is repartitioned, broadcast partition info
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Working Set size</th>
<th>Sharing</th>
<th>Local Reuse</th>
<th>Private Cache size</th>
<th>Spilling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saturating Utility</td>
<td>Small</td>
<td>H/L</td>
<td>H/L</td>
<td>Small</td>
<td>No</td>
</tr>
<tr>
<td>Low Utility</td>
<td>Big</td>
<td>Low</td>
<td>Low</td>
<td>Small</td>
<td>No</td>
</tr>
<tr>
<td>Shared High Utility</td>
<td>Big</td>
<td>High</td>
<td>H/L</td>
<td>Small</td>
<td>Yes</td>
</tr>
<tr>
<td>Private High Utility</td>
<td>Big</td>
<td>Low</td>
<td>High</td>
<td>Big</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- 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
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 version of Adaptive Selective Replication by 12%
  — monitors workload and replicates only when benefit (lower L2 hit latency) estimated to outweigh costs (more L2 misses)

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 > ASR (24%)

Cache Misses

ECC < DCC (18.6%); ECC < 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
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 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
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.


Rule 1: $X <_p Y \implies X <_m Y$

Rule 2:

Value of $L(a) = \text{Value of } \text{Max}_{<_m} \{ S(a) | S(a) <_m L(a) \}$

where $L(a)$ is a load to address $a$ and $S(a)$ is a store to address $a$; the $\text{Max}_{<_m}$ operator selects the most recent operation in the global memory order.
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

\[
S_1(A) <_M S_2(A) \implies S_1(A) <_{pt} S_2(A) \quad \text{(WAW)}
\]
\[
S(A) <_M L(A) \implies S(A) <_{pt} L(A) \quad \text{(RAW)}
\]
\[
L(A) <_M S(A) \not\implies L(A) <_{pt} S(A) \quad \text{(WAR)}
\]
Tardis Global Memory Order

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

\[ X <_m Y := X <_{ts} Y \text{ or } (X =_{ts} Y \text{ and } X <_{pt} Y) \]

- 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:
  \[ X < \text{p} Y \Rightarrow X < ts Y \vee (X = ts Y \land X < pt Y) \]
  • assuming in-order commits
    Tardis only needs to guarantee
    • 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
    \[ X < \text{m} Y := X < ts Y \text{ or } (X = ts Y \text{ and } X < pt Y) \]
  • 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 RTS and WTS
## Timestamp Management w/out Private Cache

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Load Request</th>
<th>Store Request</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timestamp</strong></td>
<td>$pts \leftarrow \max(pts, wts)$</td>
<td>$pts \leftarrow \max(pts, rts + 1)$</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>$rts \leftarrow \max(pts, rts)$</td>
<td>$wts \leftarrow pts$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$rts \leftarrow pts$</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>States</th>
<th>Core Event</th>
<th>Network Event</th>
<th>SH_REQ or EX_REQ</th>
<th>RENEW_REP or UPGRADE_REP</th>
<th>FLUSH_REQ or WB_REQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>Load</td>
<td>Send SH_REQ to TM</td>
<td>Fill in data</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td>M.wts &lt;= 0, M.pts &lt;= pts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Eviction</td>
<td>Send EX_REQ to TM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>M.wts &lt;= 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>Hit</td>
<td>Send EX_REQ to TM</td>
<td>RENEW_REP</td>
<td>D grassroots M.rts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if pts &lt;= Max(PTS, D grassroots wts)</td>
<td></td>
<td>UPGRADE_REP</td>
<td>D grassroots &lt;= M.rts</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>Send SH_REQ to TM</td>
<td>No msg sent.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if pts &gt; Max(PTS, D grassroots wts)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusive</td>
<td>Hit</td>
<td>Send FLUSH_REQ to TM</td>
<td>FLUSH_REQ</td>
<td>M.wts &lt;= D grassroots wts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if pts &lt;= Max(PTS, D grassroots wts)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D grassroots &lt;= M wts + 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D grassroots &lt;= M wts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D grassroots &lt;= D grassroots</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States</th>
<th>SH REQ</th>
<th>EX REQ</th>
<th>Eviction</th>
<th>DRAM_REQ</th>
<th>FLUSH_REQ or WB_REQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>Load from DRAM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>D grassroots &lt;= Max(D grassroots wts, D grassroots wts + lease, reqM.pts + lease)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if reqM.pts = D grassroots wts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send RENEW_REP to requester</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M grassroots &lt;= D grassroots</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Else</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send SH_REQ to requester</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M grassroots &lt;= D grassroots</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D grassroots &lt;= D grassroots</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusive</td>
<td>Send WB_REQ to the owner</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M grassroots &lt;= reqM.pts + lease</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Directory Coherence Example
Tardis Coherence Example
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>Number of Cores</td>
<td>$N = 64 @ 1 \text{ GHz}$</td>
</tr>
<tr>
<td>Core Model</td>
<td>In-order, Single-issue</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Memory Subsystem</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cacheline Size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 I Cache</td>
<td>16 KB, 4-way</td>
</tr>
<tr>
<td>L1 D Cache</td>
<td>32 KB, 4-way</td>
</tr>
<tr>
<td>Shared L2 Cache per Core</td>
<td>256 KB, 8-way</td>
</tr>
<tr>
<td>DRAM Bandwidth</td>
<td>8 MCs, 10 GB/s per MC</td>
</tr>
<tr>
<td>DRAM Latency</td>
<td>100 ns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>2-D Mesh with XY Routing</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hop Latency</td>
<td>2 cycles (1-router, 1-link)</td>
</tr>
<tr>
<td>Flit Width</td>
<td>128 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Tardis Parameters</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lease</td>
<td>10</td>
</tr>
<tr>
<td>Self Increment Period</td>
<td>100 cache accesses</td>
</tr>
<tr>
<td>Delta Timestamp Size</td>
<td>20 bits</td>
</tr>
<tr>
<td>L1 Rebase Overhead</td>
<td>128 ns</td>
</tr>
<tr>
<td>L2 Rebase Overhead</td>
<td>1024 ns</td>
</tr>
</tbody>
</table>
## Tardis Timestamp Statistics

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FMM</td>
<td>322</td>
<td>22.5%</td>
</tr>
<tr>
<td>BARNES</td>
<td>155</td>
<td>33.7%</td>
</tr>
<tr>
<td>CHOLESKY</td>
<td>146</td>
<td>35.6%</td>
</tr>
<tr>
<td>VOLREND</td>
<td>121</td>
<td>23.6%</td>
</tr>
<tr>
<td>OCEAN-C</td>
<td>81</td>
<td>7.0%</td>
</tr>
<tr>
<td>OCEAN-NC</td>
<td>85</td>
<td>5.6%</td>
</tr>
<tr>
<td>FFT</td>
<td>699</td>
<td>88.5%</td>
</tr>
<tr>
<td>RADIX</td>
<td>639</td>
<td>59.3%</td>
</tr>
<tr>
<td>LU-C</td>
<td>422</td>
<td>1.4%</td>
</tr>
<tr>
<td>LU-NC</td>
<td>61</td>
<td>0.1%</td>
</tr>
<tr>
<td>WATER-NSQ</td>
<td>73</td>
<td>12.8%</td>
</tr>
<tr>
<td>WATER-SP</td>
<td>363</td>
<td>29.1%</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td><strong>263</strong></td>
<td><strong>26.6%</strong></td>
</tr>
</tbody>
</table>
Tardis Performance

64 In-order Cores

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)
Tardis Performance

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)

(a) 16 Cores

(b) 256 Cores

use spin waiting

network traffic
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