18-742
Parallel Computer Architecture
Lecture 6: Cache Coherence II

Chris Craik (TA)
Carnegie Mellon University
Readings: Coherence

- Required for Review

- Required

- Recommended
Last Time...
Coherence: Guarantees

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order.

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate
  - **Write serialization**: provide a consistent global order seen by all processors

- Need a *point of serialization* for this global store ordering

- Ordering between writes to *different* locations is a *memory consistency model* problem: separate issue
Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman83, Papamarcos84]
  - Bus-based, single point of serialization
  - Processors observe other processors’ actions and infer ownership
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier78, Lenoski92, Laudon97]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
Snoopy Bus vs. Directory Coherence

- **Snoopy**
  + Critical path is short: miss → bus transaction to memory
  + Global serialization is easy: bus provides this already (arbitration)
  + Simple: adapt bus-based uniprocessors easily
  - Requires single point of serialization (bus): *not scalable*
    - (not quite true that snoopy *needs* bus: recent work on this later)

- **Directory**
  - Requires extra storage space to track sharer sets
    - Can be approximate (false positives are OK)
  - Adds indirection to critical path: request → directory → mem
  - Protocols and race conditions are more complex
  + Exactly as scalable as interconnect and directory storage
    (*much more scalable than bus*)
Meso State Machine

[Culler/Singh96]
Directory-Based Coherence
Directory-Based Protocols

- Required when scaling past the capacity of a single bus
  - Can be distributed across the different nodes
- Distributed, but:
  - Coherence still requires single point of serialization (for write serialization)
  - This can be different for every block (striped across nodes)

- We can reason about the protocol for a single block: one server (directory node), many clients (private caches)

- Directory receives Read and ReadEx requests, and sends Invl requests: invalidation uses explicit messages (as opposed to snoopy buses)
Key operation to support is *set inclusion test*

- False positives are OK: want to know which caches *may* contain a copy of a block, and spurious invals are ignored
- False positive rate determines *performance*

Most accurate (and expensive): full bit-vector

- What other methods may exist?

Compressed representation, linked list, Bloom filter

- [Zebchuk09] are all possible

Here, we will assume directory has perfect knowledge
Directory: Basic Operations

- Follow *semantics* of snoop-based system
  - but with explicit messages are sent

- Directory:
  - Receives *Read, ReadEx, Upgrade* requests from nodes
  - Sends *Inval/Downgrade* messages to sharers if needed
  - Forwards request to memory if needed
  - Maintain sharing state
  - Generally co-located with data copy (last level cache/memory)

- Protocol design is flexible
  - Exact forwarding paths depend on implementation
  - For example, do cache-to-cache transfer?
P0 acquires an address for reading:

1. Read

P0

1. DatEx

Home

2. DatShr

2. DatEx

P1

2. Read

P2
Rd with Former Exclusive

1. Rd

2a. Downgrade

2b. DatSpec

3a. Ack

3b. Ack

P0

Home

P1
Rd with Former Modified

- **P0**
  - 1. Rd
  - 2b. DatSpec

- **Home**
  - 2a. Downgrade
  - 3a. Rev

- **P1**
  - 3b. DatShr
RdEx with Former Exclusive

1. RdEx

P0

2a. Inv

2b. DatSpec

Home

3. Ack

P1
RdEx with Former Modified

1. RdEx

2a. Invl

2b. DatSpec

3. DatEx

P0 [Path Diagram]

Home

P1
Contestion Resolution (for Write)

P0

1a. RdEx

2a. DatEx

4. Invl

P1

1b. RdEx

3. RdEx

2b. NACK

5b. DatEx
Issues with Contention Resolution

- Need to escape race conditions by:
  - NACKing requests to busy (pending invalidate) entries
    - Original requestor retries
  - OR, queuing requests and granting in sequence
  - (Or some combination thereof)

- Fairness
  - Which requestor should be preferred in a conflict?
  - Interconnect delivery order, and distance, both matter

- We guarantee that *some* node will make forward progress

- Ping-ponging is a higher-level issue
  - With solutions like combining trees (for locks/barriers) and better shared-data-structure design
Protocol and Directory Tradeoffs

- Forwarding vs. strict request-reply
  - Increases complexity
  - Shorten critical path by creating a chain of request forwarding

- Speculative replies from memory/directory node
  - Decreases critical path length in best case
  - More complex implementation (and potentially more network traffic)

- Directory storage can imply protocol design
  - E.g., linked list for sharer set

---

Token Coherence: 
Decoupling Performance and Correctness 

Milo Martin, Mark Hill, and David Wood

Wisconsin Multifacet Project
http://www.cs.wisc.edu/multifacet/
University of Wisconsin—Madison
We See Two Problems in Cache Coherence

1. Protocol ordering bottlenecks
   - Artifact of conservatively resolving racing requests
   - “Virtual bus” interconnect (snooping protocols)
   - Indirection (directory protocols)

2. Protocol enhancements compound complexity
   - Fragile, error prone & difficult to reason about
   - Why? A distributed & concurrent system
   - Often enhancements too complicated to implement (predictive/adaptive/hybrid protocols)

Performance and correctness tightly intertwined
Rethinking Cache-Coherence Protocols

- Goal of invalidation-based coherence
  - Invariant: many readers -or- single writer
  - Enforced by globally coordinated actions

- Enforce this invariant directly using tokens
  - Fixed number of tokens per block
  - One token to read, all tokens to write

- Guarantees safety in all cases
  - Global invariant enforced with only local rules
  - Independent of races, request ordering, etc.
Technology Trends

- High-speed point-to-point links
  - No (multi-drop) busses

- Desire: low-latency interconnect
  - Avoid "virtual bus" ordering
  - Enabled by directory protocols

Technology trends → unordered interconnects
Workload Trends

- **Commercial workloads**
  - Many cache-to-cache misses
  - Clusters of small multiprocessors

- **Goals:**
  - Direct cache-to-cache misses
    (2 hops, not 3 hops)
  - Moderate scalability

Workload trends → avoid indirection, broadcast ok
Basic Approach

- Low-latency protocol
  - Broadcast with direct responses
  - As in snooping protocols

- Low-latency interconnect
  - Use unordered interconnect
  - As in directory protocols

Fast & works fine with no races...
...but what happens in the case of a race?
Basic approach… but not yet correct

- $P_0$ issues a request to write (delayed to $P_2$)
- $P_1$ issues a request to read

Diagram:

1. Request to write
2. No Copy
3. Request to read

Delayed in interconnect

$P_0$ No Copy $P_1$ No Copy $P_2$ Read/Write
Basic approach… but not yet correct

- \( P_0 \) responds with data to \( P_1 \)
Basic approach… but not yet correct

- $P_0$'s delayed request arrives at $P_2$
Basic approach… but not yet correct

- $P_2$ responds to $P_0$
Basic approach… but not yet correct

Problem: $P_0$ and $P_1$ are in inconsistent states
Locally “correct” operation, globally inconsistent
Contribution #1: Token Counting

- Tokens control reading & writing of data
  - At all times, **all blocks have** \( T \) **tokens**
    E.g., one token per processor
  - **One or more to read**
  - **All tokens to write**

- Tokens: in caches, memory, or in transit
  - Components exchange tokens & data

Provides **safety** in all cases
Token Coherence Example

- $P_0$ issues a request to write (delayed to $P_2$)
- $P_1$ issues a request to read

$P_0$ issues a request to write (delayed to $P_2$)

Request to read
Token Coherence Example

- $P_0$ at $T=0$
- $P_1$ at $T=0$
- $P_2$ at $T=15 (R)$, $T=16 (R/W)$

- $P_2$ responds with data to $P_1$
Token Coherence Example

- $P_0$'s delayed request arrives at $P_2$
Token Coherence Example

- $P_2$ responds to $P_0$
Token Coherence Example

\[ P_0 \quad T=15(R) \]
\[ P_1 \quad T=1(R) \]
\[ P_2 \quad T=0 \]

\[ T=16 (R/W) \]

\[ T=15(R) \]
\[ T=0 \]
Token Coherence Example

T=15(R)

Now what? (P₀ wants all tokens)
Basic Approach (Re-Revisited)

- As before:
  - Broadcast with direct responses (like snooping)
  - Use unordered interconnect (like directory)
  - Track tokens for safety

- **Reissue requests as needed**
  - Needed due to racing requests *(uncommon)*
  - Timeout to detect failed completion
    - Wait twice average miss latency
    - Small hardware overhead
  - All races handled in this uniform fashion
Token Coherence Example

- $P_0$ reissues request
- $P_1$ responds with a token
Token Coherence Example

T=16 (R/W)  

$P_0$'s request completed

T=0  

One final issue: What about starvation?
Contribution #2: Guaranteeing Starvation-Freedom

- Handle pathological cases
  - **Infrequently invoked**
  - Can be slow, inefficient, and simple

- When normal requests fail to succeed (4x)
  - Longer timeout and issue a **persistent request**
  - Request persists until satisfied
  - Table at each processor
  - “Deactivate” upon completion

- Implementation
  - Arbiter at memory orders persistent requests
Outline

- Overview
- Problem: ordering bottlenecks
- Solution: Token Coherence (TokenB)
- **Evaluation**
- Further exploiting decoupling
- Conclusions
Evaluation Goal: Four Questions

1. Are reissued requests rare?  
   \textbf{Yes}

2. Can Token Coherence outperform snooping?  
   \textbf{Yes: lower-latency unordered interconnect}

3. Can Token Coherence outperform directory?  
   \textbf{Yes: direct cache-to-cache misses}

4. Is broadcast overhead reasonable?  
   \textbf{Yes (for 16 processors)}

Quantitative evidence for qualitative behavior
Workloads and Simulation Methods

- **Workloads**
  - **OLTP** - On-line transaction processing
  - **SPECjbb** - Java middleware workload
  - **Apache** - Static web serving workload
  - All workloads use **Solaris 8** for SPARC

- **Simulation methods**
  - **16 processors**
  - Simics full-system simulator
  - Out-of-order processor model
  - Detailed memory system model
  - Many assumptions and parameters (see paper)
Q1: Reissued Requests

(percent of all L2 misses)

<table>
<thead>
<tr>
<th>Outcome</th>
<th>OLTP</th>
<th>SPECjbb</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Reissued</td>
<td>98%</td>
<td>98%</td>
<td>96%</td>
</tr>
<tr>
<td>Reissued Once</td>
<td>2%</td>
<td>2%</td>
<td>3%</td>
</tr>
<tr>
<td>Reissued &gt; 1</td>
<td>0.4%</td>
<td>0.3%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Persistent Requests (Reissued &gt; 4)</td>
<td>0.2%</td>
<td>0.1%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Yes; reissued requests are rare (these workloads, 16p)
Q2: Runtime: Snooping vs. Token Coherence (Hierarchical Switch Interconnect)

Similar performance on same interconnect

“Tree” interconnect
Q2: Runtime: Snooping vs. Token Coherence

Direct Interconnect

<table>
<thead>
<tr>
<th></th>
<th>OLTP</th>
<th>SPECjbb</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token Snooping</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Not applicable</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

"Torus" interconnect

Snooping not applicable
Q2: Runtime: Snooping vs. Token Coherence

Yes; Token Coherence can outperform snooping (15-28% faster)

Why? Lower-latency interconnect
Q3: Runtime: Directory vs. Token Coherence

Yes; Token Coherence can outperform directories (17-54% faster with slow directory)

Why? Direct “2-hop” cache-to-cache misses
Q4: Traffic per Miss: Directory vs. Token

Yes; broadcast overheads reasonable for 16 processors
(directory uses 21-25% less bandwidth)
Q4: Traffic per Miss: Directory vs. Token

Yes; broadcast overheads reasonable for 16 processors (directory uses 21-25% less bandwidth)

Why? Requests are smaller than data (8B v. 64B)
Outline

- Overview
- Problem: ordering bottlenecks
- Solution: Token Coherence (TokenB)
- Evaluation
- Further exploiting decoupling
- Conclusions
Contribution #3: Decoupled Coherence

- Cache Coherence Protocol
- Correctness Substrate (all cases)
- Safety (token counting)
- Starvation Freedom (persistent requests)
- Performance Protocol (common cases)

Many Implementation choices
Example Opportunities of Decoupling

- **Example #1**: Broadcast is not required

  **Predict a destination-set** [ISCA ‘03]
  - Based on past history
  - Need not be correct (rely on persistent requests)
  - Enables **larger** or more **cost-effective** systems

- **Example #2**: **predictive push**

Requires no changes to correctness substrate
Conclusions

- **Token Coherence (broadcast version)**
  - Low cache-to-cache miss latency (no indirection)
  - Avoids “virtual bus” interconnects
  - Faster and/or cheaper

- **Token Coherence (in general)**
  - Correctness substrate
    - Tokens for **safety**
    - Persistent requests for **starvation freedom**
  - Performance protocol for **performance**
  - **Decouple correctness from performance**

- Enables further protocol innovation
Cache-Coherence Protocols

- Goal: provide a consistent view of memory
- Permissions in each cache per block
  - One read/write -or-
  - Many readers
- Cache coherence protocols
  - Distributed & complex
  - Correctness critical
  - Performance critical
- Races: the main source of complexity
  - Requests for the same block at the same time
Evaluation Parameters

- **Processors**
  - SPARC ISA
  - 2 GHz, 11 pipe stages
  - 4-wide fetch/execute
  - Dynamically scheduled
  - 128 entry ROB
  - 64 entry scheduler

- **Memory system**
  - 64 byte cache lines
  - 128KB L1 Instruction and Data, 4-way SA, 2 ns (4 cycles)
  - 4MB L2, 4-way SA, 6 ns (12 cycles)
  - 2GB main memory, 80 ns (160 cycles)

- **Interconnect**
  - 15ns link latency
  - Switched tree (4 link latencies) - **240 cycles** 2-hop round trip
  - 2D torus (2 link latencies on average) - **120 cycles** 2-hop round trip
  - Link bandwidth: 3.2 Gbyte/second

- **Coherence Protocols**
  - Aggressive snooping
  - Alpha 21364-like directory
  - 72 byte data messages
  - 8 byte request messages
Q3: Runtime: Directory vs. Token Coherence

Yes; Token Coherence can outperform directories (17-54% faster with slow directory)

Why? Direct “2-hop” cache-to-cache misses
More Information in Paper

- Traffic optimization
  - Transfer tokens without data
  - Add an “owner” token

- Note: no silent read-only replacements
  - Worst case: 10% interconnect traffic overhead

- Comparison to AMD’s Hammer protocol
Verifiability & Complexity

- **Divide and conquer complexity**
  - Formal verification is work in progress
  - Difficult to quantify, but promising
  - All races handled uniformly (reissuing)

- **Local invariants**
  - Safety is response-centric; independent of requests
  - Locally enforced with tokens
  - No reliance on global ordering properties

- **Explicit starvation avoidance**
  - Simple mechanism

- **Further innovation → no correctness worries**
Traditional v. Token Coherence

- **Traditional protocols**
  - Sensitive to request ordering
  - Interconnect or directory
  - **Monolithic**
    - Complicated
    - Intertwine correctness and performance

- **Token Coherence**
  - Track tokens (safety)
  - Persistent requests (starvation avoidance)
  - Request are only “hints”

**Separate Correctness and Performance**
Conceptual Interface

Performance Protocol

Correctness Substrate

“Hint” requests

Data, Tokens, & Persistent Requests
I’m looking for block B

Please send Block B to P1

Here is block B & one token

(Or, no response)
Snooping v. Directories: Which is Better?

- Snooping multiprocessors
  - Uses broadcast
  - “Virtual bus” interconnect
    - Directly locate data (2 hops)

- Directory-based multiprocessors
  - Directory tracks writer or readers
    - Avoids broadcast
    - Avoids “virtual bus” interconnect
  - Indirection for cache-to-cache (3 hops)

Examine workload and technology trends
Workload Trends

- Commercial workloads
  - Many cache-to-cache misses or sharing misses
  - Cluster small- or moderate-scale multiprocessors

- Goals:
  - Low cache-to-cache miss latency (2 hops)
  - Moderate scalability

Workload trends → snooping protocols
Technology Trends

- High-speed point-to-point links
  - No (multi-drop) busses

- Increasing design integration
  - "Glueless" multiprocessors
  - Improve cost & latency

- Desire: unordered interconnect
  - No "virtual bus" ordering
  - Decouple interconnect & protocol

Technology trends → directory protocols
Multiprocessor Taxonomy

Cache-to-cache latency

High
Low

Workload Trends

Technology Trends

Virtual Bus
Interconnect
Unordered

Snooping
Directories
Our Goal
“COHESION: A Hybrid Memory Model for Accelerators”

John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumeta, Sanjay J. Patel
Basic Idea

- Many processor CMP (1k+ cores)
- Hierarchical caches
- Allow switching between HW and SW coherence
- HW: strong coherence criteria
- SW: no coherence criteria

+ Reduce coherence traffic when strong coherence not required
+ Can switch schemes without copies
+ Fine granularity (cache line)
Paper outline

- Analyze tradeoffs between HW and SW coherence
- Authors’ hybrid design
- Evaluation
- Comments
SW coherence control

- No directories
- No tags
- False sharing can be eliminated
- Reduced message traffic between L2 and L3
- State tracked in memory
- Coherence instructions may operate on lines not present in cache
HW coherence control

- Enables speculative prefetching, data migration
- Source code portability
- Read-release not on critical path
- State tracked with hardware bits
- All coherence messages refer to valid cache lines
Design

- MSI protocol for HW coherence
- Modified Task Centric Memory Model for SW
- Sparse directory on L2
- Non-inclusive L2/L3
State transitions

(a) $HW_{cc} \Rightarrow SW_{cc}$ conversion

(b) $SW_{cc} \Rightarrow HW_{cc}$ conversion
## Evaluation Setup

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>1024</td>
<td>–</td>
<td>Line Size</td>
<td>32</td>
<td>bytes</td>
<td>Directory Size (realistic)</td>
<td>16K</td>
<td>entries</td>
</tr>
<tr>
<td>Memory BW</td>
<td>192</td>
<td>GB/s</td>
<td>Core Freq.</td>
<td>1.5</td>
<td>GHz</td>
<td>Directory Assoc. (realistic)</td>
<td>128</td>
<td>ways</td>
</tr>
<tr>
<td>DRAM Channels</td>
<td>8</td>
<td>–</td>
<td>DRAM Type</td>
<td>GDDR5</td>
<td>–</td>
<td>Directory Size (optimistic)</td>
<td>∞</td>
<td>entries</td>
</tr>
<tr>
<td>L1I Size</td>
<td>2</td>
<td>KB</td>
<td>L1I Assoc.</td>
<td>2</td>
<td>way</td>
<td>Directory Assoc. (optimistic)</td>
<td>Full</td>
<td>–</td>
</tr>
<tr>
<td>L1D Size</td>
<td>1</td>
<td>KB</td>
<td>L1D Assoc.</td>
<td>2</td>
<td>way</td>
<td>L2 Ports</td>
<td>2</td>
<td>R/W</td>
</tr>
<tr>
<td>L2 Size</td>
<td>64</td>
<td>KB</td>
<td>L2 Assoc.</td>
<td>16</td>
<td>way</td>
<td>L3 Ports</td>
<td>1</td>
<td>R/W</td>
</tr>
<tr>
<td>L2 Size (Total)</td>
<td>8</td>
<td>MB</td>
<td>L2 Latency</td>
<td>4</td>
<td>clks</td>
<td>L3 Latency</td>
<td>16+</td>
<td>clks</td>
</tr>
<tr>
<td>L3 Size</td>
<td>4</td>
<td>MB</td>
<td>L3 Assoc.</td>
<td>8</td>
<td>way</td>
<td>L3 Banks</td>
<td>32</td>
<td>–</td>
</tr>
</tbody>
</table>
Evaluation

Table 3: Timing parameters for the baseline architecture.

Traffic comparison

Directory size/perf comparison
Evaluation

Directory size req.

Runtime
Summary

+ Directory size reduction
+ Message reduction
- Programmability
Comments

- Scalability past single programs/threads?
- How handle virtual addresses?
- Is this really useful? Can same effect be achieved different way?
- How useful for caches shared btwn CPU/GPU (ala Sandy Bridge)?
Other Issues / Backup
Memory Consistency (Briefly)

- We consider only *sequential consistency* [Lamport79] here.
- Sequential Consistency gives the appearance that:
  - All operations (R and W) happen atomically in a global order.
  - Operations from a single thread occur in order in this stream.
- Thus, ordering between different mem locations exists.
- More relaxed models exist; usually require memory barriers when synchronizing.

**Proc 0**
A = 1;

**Proc 1**
while (A == 0);
B = 1;

**Proc 2**
while (B == 0);
print A
A = 1 ?
Correctness Issue: Inclusion

- What happens with multilevel caches?
- Snooping level (say, L2) must know about all data in *private hierarchy* above it (L1)
  - What about directories?

- **Inclusive** cache is one solution [Baer88]
  - L2 must contain all data that L1 contains
  - Must propagate invalidates upward to L1

- Other options
  - **Non-inclusive**: inclusion property is optional
    - Why would L2 evict if it has more sets and more ways than L1?
    - Prefetching!
  - **Exclusive**: line is in L1 xor L2 (AMD K7)