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.

Key innovation

Token Coherence: A New Framework for Cache Coherence

- Goal: Decouple performance and correctness
  - Fast in the common case
  - Correct in all cases

- To remove ordering bottlenecks
  - Ignore races (fast common case)
  - Tokens enforce safety (all cases)

- To reduce complexity
  - Performance enhancements (fast common case)
  - Without affecting correctness (all cases)
  - (without increased complexity)

Focus of this talk

Outline

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

Technology Trends

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

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

- 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

- Request to write
- Delayed in interconnect
- \( P_0 \) issues a request to write (delayed to \( P_2 \))
- \( P_1 \) issues a request to read

Basic approach... but not yet correct

- \( P_2 \) 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

Basic Approach (Revisited)
- As before:
  - Broadcast with direct responses (like snooping)
  - Use unordered interconnect (like directory)
- Track tokens for safety
  - More refinement in a moment…

Token Coherence Example
- \( P_0 \) issues a request to write (delayed to \( P_2 \))
- \( P_1 \) issues a request to read
- \( P_2 \) responds with data to \( P_1 \)

Token Coherence Example
- \( P_0 \)’s delayed request arrives at \( P_2 \)
Token Coherence – Milo Martin

Token Coherence Example

- $P_2$ responds to $P_0$

$P_2$'s request completed

$P_0$'s request completed

One final issue: What about starvation?

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

Now what? ($P_0$ wants all tokens)

- $P_0$ reissues request
- $P_1$ responds with a token

$P_0$'s request completed
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?
   Yes
2. Can Token Coherence outperform snooping?
   Yes: lower-latency unordered interconnect
3. Can Token Coherence outperform directory?
   Yes: direct cache-to-cache misses
4. Is broadcast overhead reasonable?
   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></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**

- Direct Interconnect
- Snooping not applicable

**“Torus” interconnect**

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)

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)

Many Implementation Choices

Example Opportunities of Decoupling

• Example#1: Broadcast is not required

Example#2: 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