Token Coherence

Milo M. K. Martin

Dissertation Defense

Wisconsin Multifacet Project
http://www.cs.wisc.edu/multifacet/
University of Wisconsin—Madison
Overview

• **Technology and software trends** are changing multiprocessor design
  – Workload trends ➔ snooping protocols
  – Technology trends ➔ directory protocols

• **Three desired attributes**
  – Fast cache-to-cache misses
  – No bus-like interconnect
  – Bandwidth efficiency (moderate)

• **Our approach:** *Token Coherence*
  – Fast: directly respond to unordered requests (1, 2)
  – Correct: count tokens, prevent starvation
  – Efficient: use prediction to reduce request traffic (3)
Key Insight

• 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.
Contributions

1. Token counting rules for enforcing safety

2. Persistent requests for preventing starvation

3. Decoupling correctness and performance in cache coherence protocols
   - Correctness Substrate
   - Performance Policy

4. Exploration of three performance policies
Outline

• **Motivation: Three Desirable Attributes**
  • Fast but Incorrect Approach
  • Correctness Substrate
    – Enforcing Safety with Token Counting
    – Preventing Starvation with Persistent Requests
  • Performance Policies
    – TokenB
    – TokenD
    – TokenM

• Methods and Evaluation
• Related Work
• Contributions
Motivation: Three Desirable Attributes

- Low-latency cache-to-cache misses
- No bus-like interconnect
- Bandwidth efficient

Dictated by workload and technology trends
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 → snooping protocols
Workload Trends

Low-latency cache-to-cache misses

No bus-like interconnect  Bandwidth efficient
Workload Trends ➔ Snooping Protocols

Low-latency cache-to-cache misses

(Yes: direct request/response)

No bus-like interconnect
(No: requires a “virtual bus”)

Bandwidth efficient
(No: broadcast always)
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
Technology Trends

Low-latency cache-to-cache misses

No bus-like interconnect

Bandwidth efficient
Technology Trends → Directory Protocols

Low-latency cache-to-cache misses

(No: indirection through directory)

No bus-like interconnect
(Yes: no ordering required)

Bandwidth efficient
(Yes: avoids broadcast)
Goal: All Three Attributes

- Low-latency cache-to-cache misses
- No bus-like interconnect
- Bandwidth efficient

Step #1

Step #2
Outline

• Motivation: Three Desirable Attributes
• **Fast but Incorrect Approach**
• Correctness Substrate
  – Enforcing Safety with Token Counting
  – Preventing Starvation with Persistent Requests
• Performance Policies
  – TokenB
  – TokenD
  – TokenM
• Methods and Evaluation
• Related Work
• Contributions
Basic Approach

- Fast cache-to-cache misses
  - Broadcast with direct responses
  - As in snooping protocols

- Fast interconnect
  - Use unordered interconnect
  - As in directory protocols
  - Low latency, high bandwidth, low cost

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
Basic approach… but not yet correct

P₀

No Copy

P₁

Read-only

P₂

Read-only

Read/Write

1

2

3

4

• P₂ responds with data to P₁
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
Outline

• Motivation: Three Desirable Attributes
• Fast but Incorrect Approach
• **Correctness Substrate**
  – **Enforcing Safety with Token Counting**
  – Preventing Starvation with Persistent Requests
• Performance Policies
  – TokenB
  – TokenD
  – TokenM
• Methods and Evaluation
• Related Work
• Contributions
Enforcing Safety with Token Counting

• Definition of safety:
  – All reads and writes are coherent
  – i.e., maintain the coherence invariant
  – Processor uses this property to enforce consistency

• Approach: token counting
  – Associate a fixed number of tokens for each block
  – At least one token to read
  – All tokens to write
  – Tokens in memory, caches, and messages

• Present rules as successive refinement
  …but first, revisit example
Token Coherence Example

1. P₀ issues a request to write (delayed to P₂)
2. P₁ issues a request to read

Max Tokens: T=16 (R/W)
Token Coherence Example

- P₀
  - T=0
  - 1

- P₁
  - T=0
  - 2
  - T=1(R)

- P₂
  - T=15(R)
  - T=16 (R/W)

- P₂ responds with data to P₁

• P₂ responds with data to P₁
Token Coherence Example

- \( P_0 \)'s delayed request arrives at \( P_2 \)
Token Coherence Example

- $P_0$ responds to $P_0$

- $P_1$ responds to $P_2$

- $P_2$ responds to $P_0$
Token Coherence Example
Before addressing the starvation issue, more depth on safety

Now what? (P₀ still wants all tokens)
Simple Rules

• **Conservation of Tokens**: Components do not create or destroy tokens.

• **Write Rule**: A processor can write a block only if it holds all the block’s tokens.

• **Read Rule**: A processor can read a block only if it holds at least one token.

• **Data Transfer Rule**: A message with one or more tokens must contain data.
Deficiency of Simple Rules

- **Tokens must always travel with data!**
  - Bandwidth inefficient

1. **When collecting many tokens**
   - Much like invalidation acknowledgements

2. **When evicting tokens in “shared”**
   - (Token Coherence does not support silent eviction)
   - Simple rules require data writeback on all evictions

3. **When evicting tokens in “exclusive”**

**Solution: distinguish clean/dirty state of block**
Revised Rules (1 of 2)

• **Conservation of Tokens**: Tokens may not be created or destroyed. *One token is the owner token that is clean or dirty.*

• **Write Rule**: A processor can write a block only if it holds all the block’s tokens *and has valid data*. *The owner token of a block is set to dirty when the block is written.*

• **Read Rule**: A processor can read a block only if it holds at least one token *and has valid data*.

• **Data Transfer Rule**: A message with *a dirty owner token* must contain data.
Revised Rules (2 of 2)

• **Valid-Data Bit Rule:**
  – Set valid-data bit when data and token(s) arrive
  – Clear valid-data bit when it no longer holds any tokens
  – The memory sets the valid-data bit whenever it receives the **owner token** (even if the message does not contain data).

• **Clean Rule:**
  – Whenever the memory receives the owner token, the **memory sets the owner token** to clean.

  Result: reduced traffic, encodes all MOESI states
Token Counting Overheads

• Token storage in caches
  – 64 tokens, owner, dirty/clear = 8 bits
  – 1 byte per 64-byte block is 2% overhead
• Transferring tokens in messages
  – Data message: similar to above
  – Control message: 1 byte in 7 bytes is 13%
• Non-silent eviction overheads
  – Clean: 8-byte eviction per 72-byte data is 11%
  – Dirty: data + token message = 2%
• Token storage in memory
  – Similar to a directory protocol, but fewer bits
  – Like directory: ECC bits, directory cache
Other Token Counting Issues

• **Stray data**
  – Tokens can arriving at any time
  – Ingest or redirect to memory

• **Handling I/O**
  – DMA: issue read requests and write requests
  – Memory mapped: unaffected

• **Block-write instructions**
  – Send clean-owner without data

• **Reliability**
  – Assumes reliable delivery
  – Same as other coherence protocols
Outline

• Motivation: Three Desirable Attributes
• Fast but Incorrect Approach
• Correctness Substrate
  – Enforcing Safety with Token Counting
  – Preventing Starvation with Persistent Requests
• Performance Policies
  – TokenB
  – TokenD
  – TokenM
• Methods and Evaluation
• Related Work
• Contributions
Preventing Starvation via Persistent Requests

• Definition of starvation-freedom:
  – All loads and stores must eventually complete

• Basic idea
  – Invoke after timeout (wait 4x average miss latency)
  – Send to all components
  – Each component remembers it in a small table
  – Continually redirect all tokens to requestor
  – Deactivate when complete

• As described later, not for the common case
Back to the example…
Token Coherence Example

T=15(R)

$P_0$

T=1(R)

$P_1$

T=0

$P_2$

$P_0$ still wants all tokens
Token Coherence Example

- \( P_0 \) issues persistent request
- \( P_1 \) responds with a token
• \( P_0 \)'s request completed
• \( P_0 \)'s deactivates persistent request
Persistent Request Arbitration

• **Problem**: many processors issue persistent requests for the same block

• **Solution**: use starvation-free arbitration
  – Single arbiter (in dissertation)
  – Banked arbiters (in dissertation)
  – Distributed arbitration (my focus, today)
Distributed Arbitration

• One persistent request per processor
  – One table entry per processor

• Lowest processor number has highest priority
  – Calculated per block
  – Forward all tokens for block (now and later)

• When invoking
  – “mark” all valid entries in local table
  – Don’t issue another persistent request until
    “marked” entries are deactivated

• Based on arbitration techniques (FutureBus)
Distributed Arbitration System

Interconnect

Processor Component (CPU and caches)

Persistent Request Table

Memory Module Component (memory controller and DRAM)

Persistent Request Table

requires associative search by address

One Entry Per Processor

Valid Bit

Address

Marked Bit
Other Persistent Request Issues

- **All tokens, no data problem**
  - Bounce clean owner token to memory

- **Persistent read requests**
  - Keep only one (non-owner) token
  - Add read/write bit to each table entry

- **Preventing reordering of activation and deactivation messages**
  1. Point-to-point ordering
  2. Explicit acknowledgements
  3. Acknowledgement aggregation
  4. Large sequence numbers

- **Scalability of persistent requests**
Outline

• Motivation: Three Desirable Attributes
• Fast but Incorrect Approach
• Correctness Substrate
  – Enforcing Safety with Token Counting
  – Preventing Starvation with Persistent Requests

• **Performance Policies**
  – TokenB
  – TokenD
  – TokenM

• Methods and Evaluation
• Related Work
• Contributions
Performance Policies

• Correctness substrate is sufficient
  – Enforces safety with token counting
  – Prevents starvation with persistent requests

• A performance policy can do better
  – Faster, less traffic, lower overheads
  – Direct when and to whom tokens/data are sent

• With no correctness requirements
  – Even a random protocol is correct
  – Correctness substrate has final word
Decoupled Correctness and Performance

Cache Coherence Protocol

Correctness Substrate (all cases)

- Safety (token counting)
- Starvation Freedom (persistent requests)

Performance Policy (common cases)

Safety

Starvation Freedom

Just some of the implementation choices

Simple rules
Refined rules
Centralized
Distributed
TokenB
TokenD
TokenM
TokenB Performance Policy

• **Goal**: snooping without ordered interconnect

• Broadcast unordered *transient requests*
  – **Hints** for recipient to send tokens/data
  – Reissue requests once (if necessary)
    After 2x average miss latency
  – Substrate invokes a persistent request
    As before, after 4x average miss latency

• **Processors & memory respond to requests**
  – As in other MOESI protocols
  – Uses migratory sharing optimization
    (as do our base-case protocols)
TokenB Potential

- Low-latency cache-to-cache misses
  - (Yes: direct request/response)

- No bus-like interconnect
  - (Yes: unordered protocol)

- Bandwidth efficient
  - (No: broadcast always)
Beyond TokenB

- Broadcast is **not required**

- **TokenD**: directory-like performance policy
- **TokenM**
  - Multicast to a predicted *destination-set*
  - Based on past history
  - Need not be correct (fall back on persistent request)

- **Enables larger or more cost-effective systems**
TokenD Performance Policy

• **Goal**: traffic & performance of directory protocol

• **Operation**
  – Send all requests to soft-state directory at memory
  – Forwards request (like directory protocol)
  – Processors respond as in MOESI directory protocol

• **Reissue requests**
  – Identical to TokenB

• **Enhancement**
  – Pending set of processors
  – Send *completion message* to update directory
TokenD Potential

Low-latency cache-to-cache misses

(No: indirection through directory)

No bus-like interconnect
(Yes: unordered protocol)

Bandwidth efficient
(Yes: avoids broadcast)
TokenM Performance Policy

• **Goals:**
  – Less traffic than TokenB
  – Faster than TokenD

• **Builds on TokenD, but uses prediction**
  – Predict a destination set of processors
  – Soft-state directory forwards to missing processors
Destination-Set Prediction

• Observe past behavior to predict the future
  – Leverage prior work on coherence prediction

• Training events
  – Other requests
  – Data responses

• Mostly subsumes
  – TokenD
  – TokenB
Destination-Set Predictors

• Three predictors (ISCA ‘03 paper)
  – Broadcast-if-shared
  – Group
  – Owner

• All simple cache-like (tagged) predictors
  – 4-way set-associative
  – 8k entries (32KB to 64KB)
  – 1024-byte macroblock-based indexing

• Prediction
  – On tag miss, send only to memory
  – Otherwise, generate prediction
TokenM Potential

- Low-latency cache-to-cache misses
  - (Yes: direct request/response)

- No bus-like interconnect
  - (Yes: unordered protocol)

- Bandwidth efficient
  - (Yes: reduce broadcasts using prediction)

Bandwidth/latency tradeoff
Outline

• Motivation: Three Desirable Attributes
• Fast but Incorrect Approach
• Correctness Substrate
  – Enforcing Safety with Token Counting
  – Preventing Starvation with Persistent Requests
• Performance Policies
  – TokenB
  – TokenD
  – TokenM
• **Methods and Evaluation**
• Related Work
• Contributions
Evaluation Methods

• Non-goal: exact speedup numbers
  – Many assumptions and parameters (next slide)

• Goal: Quantitative evidence for qualitative behavior

• Simulation methods
  – Full-system simulation with Simics
  – Dynamically scheduled processor model
  – Detailed memory system model

• Multiple simulations due to workload variability
Evaluation Parameters

- **16 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**
  - 64KB L1 Instruction and Data, 4-way SA, 2 ns (4 cycles)
  - 4MB L2, 4-way SA, 6 ns (12 cycles)
  - 4GB main memory, 80 ns (160 cycles)

- **Interconnect**
  - 15ns link latency (30 cycles)
  - 4ns to enter/exit interconnect
  - Switched tree (4 link latencies) - **256 cycles**
    2-hop round trip
  - 2D torus (2 link latencies on average) - **136 cycles** 2-hop round trip

- **Coherence Protocols**
  - Aggressive snooping
  - Alpha 21364-like directory
  - **72 byte data messages**
  - **8 byte request messages**
Three Commercial Workloads

• All workloads use Solaris 9 for SPARC
• OLTP - On-line transaction processing
  – IBM’s DB2 v7.2 DBMS
  – TPCC-like workload
  – 5GB database, 25,000 warehouses
  – 8 raw disks, additional log disk
  – 256 concurrent users
• SPECjbb - Java middleware workload
  – Sun’s HotSpot 1.4.1-b21 Server JVM
  – 24 threads, 24 warehouses (~500MB)
• Apache - Static web serving workload
  – 80,000 files, 6400 concurrent users
Are reissued and persistent requests rare? (percent of all misses)

<table>
<thead>
<tr>
<th>Outcome</th>
<th>SpecJBB</th>
<th>Apache</th>
<th>OLTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Reissued</td>
<td>99.5%</td>
<td>99.1%</td>
<td>97.6%</td>
</tr>
<tr>
<td>Reissued Once</td>
<td>0.2%</td>
<td>0.7%</td>
<td>1.5%</td>
</tr>
<tr>
<td>Persistent Requests</td>
<td>0.3%</td>
<td>0.2%</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

TokenB results (TokenD/TokenM are similar)

Yes, reissue requests are rare
Runtime: Snooping vs. TokenB
“Tree” Switched Interconnect

Similar performance on same interconnect

“Tree” interconnect
Runtime: Snooping vs. TokenB
“Torus” Interconnect

Snooping not applicable

“Torus” interconnect
Runtime: Snooping vs. TokenB
“Tree” Switched Interconnect

TokenB can outperform snooping (23-34% faster)

Why? Lower latency interconnect
Runtime: Directory vs. TokenB

TokenB outperforms directory (12-64% or 7-27%)

Why? Avoids directory lookup, third hop
Interconnect Traffic: TokenB and Directory

TokenB’s additional traffic is moderate (18-35% more)

Why?

(1) requests smaller than data (8B v. 64B)
(2) Broadcast routing

Analytical model:

64p is 2.3x
256p is 3.9x
Runtime: TokenD and Directory

Similar runtime, still slower than TokenB
Runtime: TokenD and TokenM

![Graph showing runtime comparisons between TokenD and TokenM for SPECjbb, Apache, and OLTP workloads. The x-axis represents different directories and tokens, while the y-axis shows normalized runtime. The bars indicate the effect of directory access latency.]
Interconnect Traffic: TokenD and Directory

Similar traffic, still less than TokenB
Interconnect Traffic: TokenD and TokenM

Torus traffic (normalized byte per miss)

SPECjbb  Apache  OLTP

- Writeback control messages
- Acknowledgment & miscellaneous control messages
- Reissued & persistent requests (token only)
- Requests & forwarded requests
- Data response and writeback data messages
Evaluation Summary

• **TokenB faster than:**
  – Snooping, due to faster/cheaper interconnect
  – Directories, avoids directory looking & third hop

• **TokenB uses more traffic than directories**
  – Especially as system size increases

• **TokenD is similar to directories**
  – Runtime and traffic

• **TokenM provides intermediate design points**
  – Owner is 6-16% faster than TokenD, negligible additional bandwidth
  – Bcast-if-shared is only 1-6% slower then TokenB, but 7-14% less traffic (more for larger systems)
Outline

• Motivation: Three Desirable Attributes
• Fast but Incorrect Approach
• Correctness Substrate
  – Enforcing Safety with Token Counting
  – Preventing Starvation with Persistent Requests
• Performance Policies
  – TokenB
  – TokenD
  – TokenM
• Methods and Evaluation
• Related Work
• Contributions
Architecture Related Work (1 of 2)

• Many, many previous coherence protocols
  – Including many hybrids and adaptive protocols

• Coherence prediction
  – Early work: migratory sharing optimization [ISCA93]
  – Later: DSI, LTP, Cosmos

• Destination-Set Prediction [ISCA ‘03]
  – Multicast Snooping [ISCA‘99, TPDS]
  – Acacio et al. [SC’02, Pact’02]

• Cachet [ICS‘99]
Non-Architecture Related Work (2 of 2)

- Much tangentially related work
  - …but little directly related
  - Many single-token schemes (token-base sync.)
  - Or use multiple tokens for faults (quorum commit)

- Fortran-M uses message passing of many tokens for protecting shared variables [Foster]

- Read/writers locks
  - Not implemented using tokens
Contributions

1. **Token counting** rules for enforcing safety

2. **Persistent requests** for preventing starvation

3. **Decoupling correctness and performance** in cache coherence protocols
   - Correctness Substrate
   - Performance Policy

4. Developing and evaluating **three performance policies**
Backup Slides
Centralized Arbiter System

Diagram of a Centralized Arbiter System showing the interaction between processors, memory modules, and persistent request tables connected through an interconnect.
Centralized Arbiter Example
Banked Arbiter System
Distributed Arbitration System

![Diagram of Distributed Arbitration System](image)

- Processor Component (CPU and caches)
- Persistent Request Table
- Interconnect
- Memory Module Component (memory controller and DRAM)
- Persistent Request Table
- Requires associative search by address

One Entry Per Processor

Valid Bit, Address, Marked Bit
Distributed Arbitration Example
Predictor #1: Broadcast-if-shared

- Performance of snooping, fewer broadcasts
  - Broadcast for “shared” data
  - Minimal set for “private” data

- Each entry: valid bit, 2-bit counter
  - Decrement on data from memory
  - Increment on data from a processor
  - Increment other processor’s request

- Prediction
  - If $counter > 1$ then broadcast
  - Otherwise, send only to memory
Predictor #2: Owner

- Traffic similar to directory, fewer indirections
  - Predict one extra processor (the “owner”)
  - Pairwise sharing, write part of migratory sharing

- Each entry: valid bit, predicted owner ID
  - Set “owner” on data from other processor
  - Set “owner” on other’s request to write
  - Unset “owner” on response from memory

- Prediction
  - If “valid” then predict “owner” + memory
  - Otherwise, send only to memory
Predictor #3: Group

- **Try to achieve ideal bandwidth/latency**
  - Detect groups of sharers
  - Temporary groups or logical partitions (LPAR)

- **Each entry: N 2-bit counters**
  - Response or request from another processor → Increment corresponding counter
  - Train down by occasionally decrement all counters (every 2N increments)

- **Prediction**
  - For each processor, if the corresponding counter > 1, add it in the predicted set
  - Send to predicted set + memory