November 2003

Token Coherence: A New Framework for Shared-Memory Multiprocessors

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

Mark D. Hill
University of Wisconsin

David A. Wood
University of Wisconsin

At the time of publication, author Milo M.K. Martin was affiliated with the University of Wisconsin. Currently, November 2006, he is a faculty member in the Department of Computing and Information Sciences at the University of Pennsylvania.

Token Coherence: A New Framework for Shared-Memory Multiprocessors

Abstract

Commercial workload and technology trends are pushing existing shared-memory multiprocessor coherence protocols in divergent directions. Token Coherence provides a framework for new coherence protocols that can reconcile these opposing trends.

At the time of publication, author Milo M.K. Martin was affiliated with the University of Wisconsin. Currently, November 2006, he is a faculty member in the Department of Computing and Information Sciences at the University of Pennsylvania.

Token Coherence: A New Framework for Shared-Memory Multiprocessors

COMMERCIAL WORKLOAD AND TECHNOLOGY TRENDS ARE PUSHING EXISTING SHARED-MEMORY MULTIPROCESSOR COHERENCE PROTOCOLS IN DIVERGENT DIRECTIONS. TOKEN COHERENCE PROVIDES A FRAMEWORK FOR NEW COHERENCE PROTOCOLS THAT CAN RECONCILE THESE OPPOSING TRENDS.

Milo M.K. Martin
Mark D. Hill
David A. Wood
University of Wisconsin-Madison

The performance of database and Web servers is important because the services they provide are increasingly becoming part of our daily lives. Many of these servers are shared-memory multiprocessors, because most commercial workloads have abundant thread-level parallelism. Multiprocessors commonly use private per-processor caches that buffer blocks of the shared memory to improve both effective memory latency and bandwidth.

Cache coherence protocols

A cache coherence protocol manages the read and write permissions of data in the caches to ensure all processors observe a consistent view of shared memory. As described in the "Cache Coherence Review" sidebar, most protocols enforce a coherence invariant that permits each memory block to have either multiple read-only copies or a single writable copy, but never both at the same time. Current coherence protocols enforce this invariant indirectly via a subtle combination of local actions and request ordering restrictions. Unfortunately, emerging workload and technology trends reduce the attractiveness of these existing solutions.

We propose the Token Coherence framework, which directly enforces the coherence invariant by counting tokens (requiring all of a block's tokens to write and at least one token to read). This token-counting approach enables more obviously correct protocols that do not rely on request ordering and can operate with alternative policies that seek to improve the performance of future multiprocessors.

Workload trends favor snooping protocols

Commercial workloads represent an important market for shared-memory multiprocessors. These workloads frequently share data, and thus they favor cache coherence protocols that minimize the latency of transferring data directly between caches (that is, by minimizing the latency of cache-to-cache misses).[1]

Snooping protocols achieve low-latency cache-to-cache-misses by broadcasting coher-

Published by the IEEE Computer Society

Cache Coherence Review

Cache coherence protocols keep caches transparent to software, even for write-back caches that defer updating memory until a block is replaced.[1]

Many coherence protocols tag cache blocks using the MOSI (modified, owned, shared, invalid) states;[2] each state conveys rights and responsibilities. A cache with a block in state modified (M) allows processor reads and writes and must provide data when another cache requests the block. Owned (O) allows processor reads and must provide data when requested. Shared (S) also allows processor reads, but does not provide data to other requesters. Invalid (I) has no rights or responsibilities.

Coherence protocols coordinate cached copies by restricting the MOSI states of a given block across all caches. Common write-invalidate coherence protocols ensure the coherence invariant of a single writer or multiple readers. Single writer means that if one cache is in state M, the rest are in I. Multiple readers allow at most one cache in O, zero or more in S, and the rest in I.

Consider, for example, a request to get a block in M when the requester is currently in I, and three other caches hold the block in O, S, and S. Most protocols honor this request with the cache in O providing data and the O, S, and S caches invalidating (they all go into state I).

Coherence protocols face two key challenges. First, they must maintain the coherence invariant (often by establishing an order of requests to resolve requests racing for the same block). Second, they must ensure freedom from starvation (that is, all memory references must eventually complete).

Cache coherence protocols enforce ordering restrictions on reads and writes to a single block (the coherence invariant). Processors use coherent caches as part of implementing a multiprocessor's memory consistency model, which specifies ordering restrictions to reads and writes among many blocks.[1] Because Token Coherence enforces the coherence invariant, it does not affect the implementation of standard memory consistency models.

References

1. D.E. Culler and J. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.
2. P. Sweazey and A.J. Smith, "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus," *Proc. 13th Ann Int'l Symp. Computer Architecture*, ACM Press, 1986, pp. 414-423.

ence requests on a shared bus. All caches "snoop" the bus to maintain the coherence invariant and to allow other caches to directly respond to requests, speeding up cache-to-cache misses. The processors resolve racing requests (multiple concurrent requests for the same block) using the total order of requests naturally provided by the bus. Fair bus arbitration prevents starvation.

To overcome the increasingly difficult challenge of scaling the bandwidth of shared-wire buses, some snooping designs (such as the Sun E10000[2]) broadcast requests on a "virtual bus" created with an indirect switched interconnect, such as the one in Figure 1a. These interconnects use high-speed point-to-point links to provide higher bandwidth than buses, but the use of dedicated switch chips often increases system cost and latency.

Technology trends favor directory protocols

The increasing number of transistors per chip predicted by Moore's law encourages increasingly integrated designs, making glue logic (dedicated switch chips) less desirable. Many current and future systems will integrate processor(s), cache(s), coherence logic, switch logic, and memory controller(s) on a single die (such as the Alpha 21364[3] and AMD Opteron[4]). Directly connecting these highly integrated nodes leads to fast and low-cost glueless interconnects, such as the one in Figure 1b. Unfortunately, glueless interconnects do not naturally provide the virtual-bus properties required by snooping protocols. To exploit these glueless interconnects, recent highly integrated designs[3,4] have used coherence protocols—such as directory protocols—that do not rely on interconnect ordering.

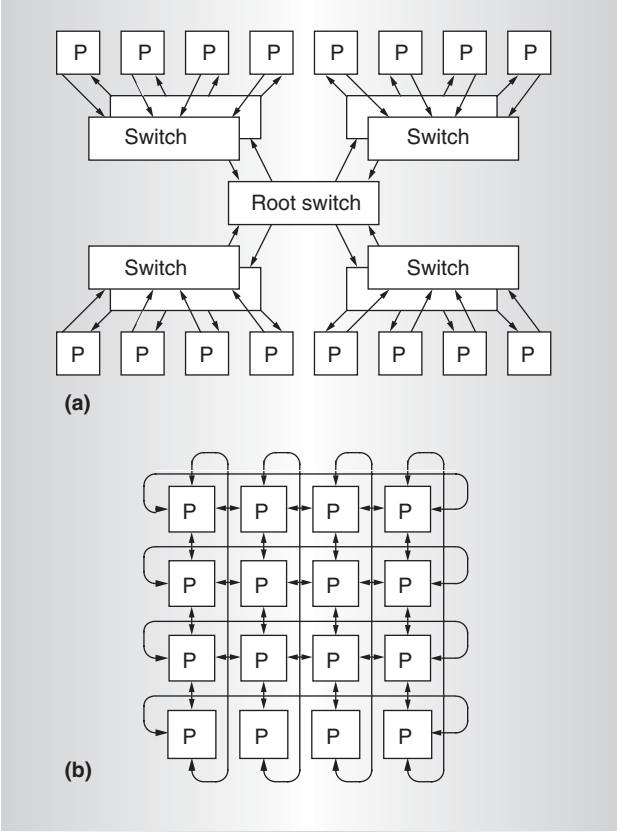

Figure 1. Multiprocessor interconnect options: two-level broadcast tree (a) and 4 × 4 bidirectional torus (b). The boxes marked "P" represent highly integrated nodes that include a processor, caches, memory controller, and coherence controllers. The indirect broadcast tree uses dedicated switches, while the torus is a directly connected interconnect. For 16-processor systems, the torus has lower latency (two versus four chip crossings on average) and does not require any glue chips. However, unlike the broadcast tree, the torus does not provide the total order necessary for conventional snooping.

Directory protocols employ a directory that tracks where each block is cached. Requests travel via an unordered interconnect to the directory (usually at a block's home memory module). The directory responds and/or forwards the request to other caches, which in turn respond with data or acknowledgments. To resolve races without relying on interconnect ordering, the directory acts as an ordering point, blocking or queuing problematic requests as necessary. Fair queuing at the directory prevents starvation. A principal disadvantage of directory protocols is that cache-to-cache misses incur the delay of a directory access and an extra interconnect traversal.

UnorderedB: Fast but incorrect

To reconcile these opposing workload and technology trends, the unordered-using-broadcast (UnorderedB) coherence protocol broadcasts coherence requests (like snooping protocols), but uses a fast, unordered interconnect (like directory protocols). UnorderedB can be faster than either directory protocols (by avoiding directory indirections on cache-to-cache misses) or snooping protocols (by avoiding slow, ordered interconnects). We show, however, that UnorderedB is incorrect: It fails to enforce the coherence invariant or prevent starvation. Fortunately, Token Coherence will allow us to restore correctness to UnorderedB, while retaining its performance potential.

UnorderedB acts like a standard MOSI (modified, owned, shared, invalid) snooping protocol, but without any request ordering. A cache can request a block in state S (ReqS) or state M (ReqM). Each cache and memory checks the state of the requested block. If the block is in state I, a cache or memory ignores all requests. If the block is in state S, it ignores ReqS requests, but transitions to state I on ReqM requests. If the block is in state O, it responds with data and either stays in O on ReqS requests or transitions to I on ReqM requests. If the block is in state M, it responds as in state O.

Unfortunately, UnorderedB is not always correct in the presence of protocol races. Consider the example illustrated in Figure 2a, in which processor P_0 initially has a block in state M, processor P_1 seeks the same block in S (ReqS), and processor P_2 seeks it in M (ReqM). P_1 and P_2 both broadcast their requests at time ❶, which the interconnect promptly delivers to each other at time ❷ but belatedly delivers to P_0 at times ❸ and ❺ (because of contention on the unordered interconnect). Processors P_1 and P_2 handle each other's request at time ❷ but take no action because each lacks a valid copy. P_0 satisfies P_1's request at time ❹, by responding with data and transitioning to state O. Finally, P_2's delayed request arrives at P_0 at time ❺, and P_2 satisfies the request at time ❻, by sending data and transitioning to state I. After receiving their responses, P_2 believes that it has the single, writable copy of the block, and P_1 believes it holds a valid read-only copy. This resulting sit-

uation is unsafe: It violates the coherence invariant and can result in erroneous program behavior. Starvation can also result, if repeated races prevent a processor from ever making forward progress (for example, a request might miss the owner, and thus never receive a data response).

This simplified example illustrates the fundamental difficulty with traditional coherence protocols. Even though each processor takes a locally correct action, collectively the processors violate the coherence invariant. Current coherence protocols avoid this particular race by ordering requests, either in the interconnect or at a directory. (In the example, if all processors handle ReqS before ReqM, then P_1 will invalidate its shared copy, thereby maintaining the invariant.) However, ordering requests introduces significant overhead.

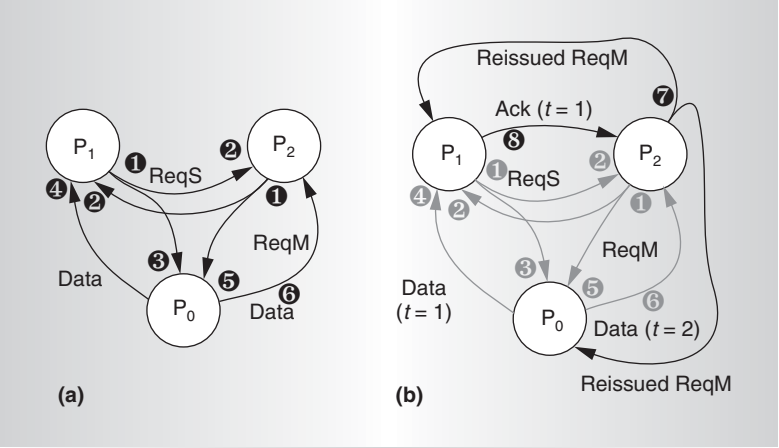

Figure 2. Example race: fast but incorrect (a) and using Token Coherence (b). A request for shared (ReqS) races with a request for modified (ReqM) to violate the coherence invariant unless Token Coherence checks safety.

Enforcing correctness with Token Coherence

Instead of relying on request ordering, Token Coherence uses two mechanisms to enforce correctness. These mechanisms—token counting and persistent requests—form the correctness substrate, illustrated in Figure 3.

Enforcing safety via token counting

The correctness substrate uses token counting to explicitly enforce the coherence invariant. During system initialization, the system assigns each block T tokens, distinguishes one as the owner token, and stores them at the block's home memory. Processor caches and coherence messages can also hold tokens. T is at least as large as the number of processors.

Tokens and data move throughout the system obeying four rules:

- *Rule 1.* At all times, each block has exactly T tokens in the system, one of which is the owner token.

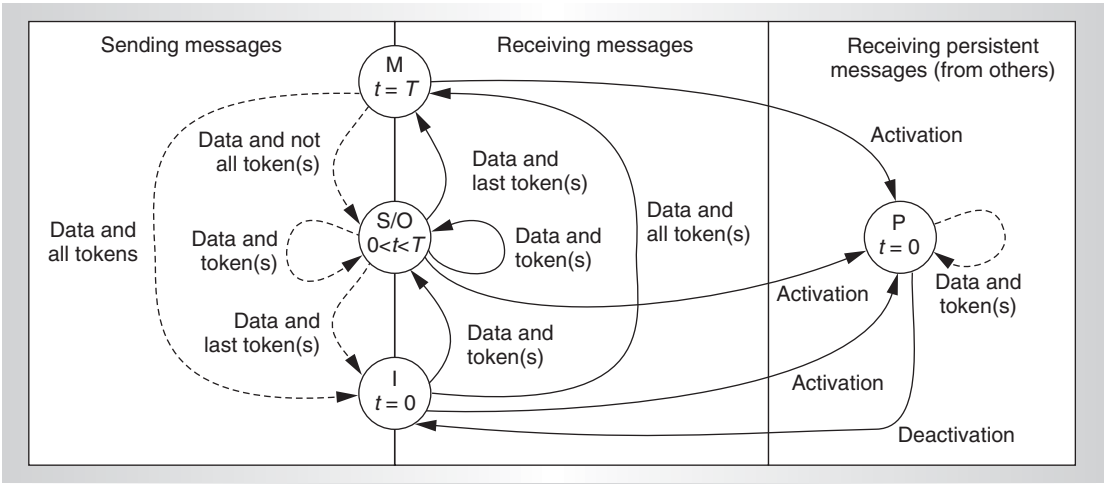

Figure 3. Correctness substrate state transitions for a processor. As a simplification, the figure shows only tokens sent with data. Symbol t represents the current token count, and T represents all the tokens. Solid arcs are transitions in response to incoming messages. Dashed arcs are transitions that occur when a processor sends an outgoing message. The P state occurs when a node receives another processor's persistent request from the arbiter. Each processor must also remember its own persistent request; for simplicity this figure does not show these transitions.

- *Rule 2.* A processor can write a block only if it holds all T tokens for that block.
- *Rule 3.* A processor can read a block only if it holds at least one token for that block and has valid data.
- *Rule 4.* If a coherence message contains the owner token, it must contain data.

Rules 1 through 3 ensure safety by directly enforcing the coherence invariant of a single writer or multiple readers. A processor can write a block if it has all T tokens, because no other processor can have a token for reading. A processor with one or more tokens can safely read the block, because holding a token prevents another processor from holding all the tokens and thus from writing the block. These rules enable designers to reason about safety without concern for subtle interactions among protocol states, multiple concurrent requests, explicit acknowledgments, write-back messages, interconnect ordering, or system hierarchy.

Rule 4 requires that a valid copy of the data must always travel with the owner token to ensure that

- the current data for a block is never lost and
- a processor that has collected all the tokens will have received at least one data response (when it received the owner token).

However, Rule 4 allows for non-owner tokens to travel without data (to provide a bandwidth-efficient means of collecting non-owner tokens from many processors). Because this rule allows processors and memory to receive tokens without valid data, caches and memory use separate valid bits for data and tokens.

Because Rule 1 requires that tokens be indestructible, caches return tokens to memory when evicting blocks. Rule 4 requires that evictions of the owner token must include data (much like the eviction of an M or O block in a traditional protocol). Caches also send evicted non-owner tokens to memory (rather than performing a silent eviction). Because this message is small (that is, it contains no data), it adds only a small traffic overhead.

The system holds tokens in processor caches (for example, part of tag state), memory (for example, efficiently encoded in error-correcting code bits), and coherence messages. Because the system counts tokens, but does not track which processors hold them, tokens can be stored in $2 + \lceil \log_2 T \rceil$ bits (the valid bit, owner-token bit, and non-owner token count). For example, encoding 64 tokens requires one byte, a 1.6 percent overhead for 64-byte blocks.

Avoiding starvation via persistent requests

Although token counting ensures that races do not violate the coherence invariant, it does not ensure that a request is eventually satisfied. Thus the correctness substrate provides persistent requests to prevent starvation. When a processor detects possible starvation (such as via a time-out), it initiates a persistent request. The substrate then activates at most one persistent request per block, using a fair-arbitration mechanism. Each system node remembers all activated persistent requests (for example, in a table at each node) and forwards all tokens for the block—those tokens currently present and received in the future—to the request initiator. Finally, when the initiator has sufficient tokens, it performs a memory operation (a load or store instruction) and deactivates its persistent request.

We know of two approaches to implementing persistent requests. First, Martin, Hill, and Wood (and our performance results later) use an arbiter at each home memory module.[5] Processors send persistent requests to the appropriate arbiter for a block. The arbiter selects a request to activate and informs all processors, which save the request in a small table at each node. A similar sequence occurs on deactivation. Second, Martin has developed an alternative persistent request implementation that uses a distributed arbitration policy to create a priority ordering.[6] This implementation improves worst-case behavior by allowing faster handoff of highly contended blocks (such as a hot lock).

TokenB: Fast and correct

Token Coherence protocols rely on the correctness substrate to ensure safety and freedom from starvation in all cases. This guarantee permits aggressive performance policies that optimize for the common case without worrying about rare corner cases. Token-using-Broadcast (TokenB) is one

example of a performance policy. TokenB behaves much like UnorderedB, allowing it to be fast in the common no-race case. However, it relies on the correctness substrate to ensure safety and prevent starvation when races do occur.

Specifically, TokenB usually operates just like UnorderedB. Caches broadcast coherence requests, which we call transient requests to differentiate them from persistent requests. When a persistent request is active in the system, it has priority over transient requests. Otherwise, caches and memory respond to transient requests in the same way as UnorderedB, identifying the MOSI states with token counts. All T tokens implies M, 1 to T–1 tokens (including the owner token) implies O, 1 to T–1 tokens (excluding the owner token) implies S, and no tokens implies I.

If a transient request fails to garner sufficient tokens within a time-out interval of twice the average miss latency (for example, because of racing requests), the cache controller reissues the request up to three times. If these requests continue to fail, the requesting processor eventually invokes a persistent request, which always succeeds.

Returning to the example race in Figure 2b, the block has three tokens initially held by P_0. P_1 receives one token in the response at time ❹, allowing it to read the block. Because of the race, P_2 only receives two tokens in the response at time ❻. Unlike UnorderedB, this does not violate the coherence invariant, because TokenB requires that a processor have all three tokens before writing the block. P_2 waits for additional tokens and, after a specified time-out interval, reissues its request at time ❼. P_1 responds with the missing token at time ❽, allowing P_2 to finally complete its request. Because only one race occurs in this example, P_2 need not issue a persistent request.

Evaluation methods

We quantitatively evaluate TokenB using methods highlighted here, but explained in detail elsewhere.[5] We use three multiprocessor commercial workloads as benchmarks:[7] a static Web serving workload (Apache), an online transaction processing workload (OLTP), and a Java middleware workload (SPECjbb).

Our target system is a 16-processor Sparc multiprocessor with highly integrated nodes. Each node includes a pipelined and dynamically scheduled processor, split L1 caches (128 Kbytes each), unified L2 cache (4 Mbytes), coherence protocol controllers, and a memory controller for part of the globally shared memory. All components maintain coherence on 64-byte blocks.

We compare TokenB against two analogous coherence protocols; all three implement the same MOSI base states, types of requests, and migratory sharing optimization. *Snooping* is a snooping protocol that requires an ordered interconnect, but allows many concurrent requests with flexible timing. *Directory* is a standard full-map directory protocol, inspired by the protocols in the Origin 2000[8] and Alpha 21364.[3] It uses an unordered interconnect but must queue requests at the directory, in some cases, to prevent races.

We evaluate systems with two interconnection networks. *Tree* is a four-ary tree (Figure 1a) that uses two levels of switch chips to deliver requests in a total order. *Torus* is a glueless two-dimensional, bidirectional torus (Figure 1b) that delivers requests as quickly as possible, without regard to order. We present results that assume links have unlimited bandwidth; Martin, Hill, and Wood[5] includes limited-bandwidth results.

We use the Virtutech Simics full-system, functional-execution-driven simulator augmented to simulate memory hierarchies and out-of-order processors.[7] We pseudorandomly perturb simulations to calculate 95 percent confidence intervals, as described in this issue by Alameldeen and Wood.[9]

TokenB evaluation

We present evidence that Token Coherence can improve performance via four questions. Figure 4 presents the normalized runtime (shorter is better) of TokenB on the tree and torus interconnects, Snooping on the tree interconnect, and Directory on the torus interconnect.

Is the number of reissued and persistent requests small?

Yes; on average for our workloads, less than 3 percent of TokenB's cache misses are reissued, with only 0.2 percent causing persistent requests, as Table 1 shows. Thus, TokenB handles the common case like UnorderedB, while

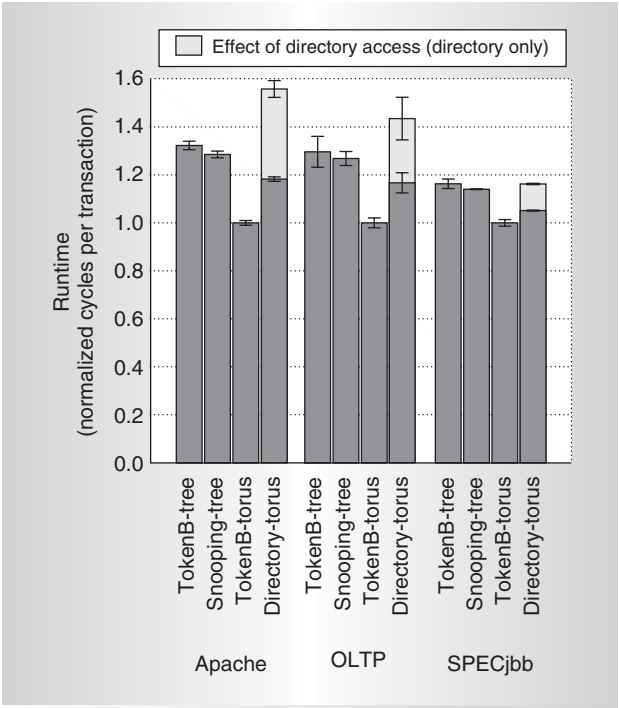

Figure 4. Runtime normalized to TokenB on the torus interconnect.

rarely invoking the correctness substrate's starvation prevention mechanism.

Does TokenB outperform Snooping?

Yes; for the ordered tree interconnect, both protocols perform similarly (Figure 4) with similar traffic (not shown), but TokenB becomes 15 to 28 percent faster by using the lower-latency unordered torus, which Snooping cannot use.

Does TokenB outperform Directory?

Yes; by removing the directory access latency and extra interconnect traversal from the critical path of cache-to-cache misses, TokenB is faster than Directory by 17 to 54 percent using the torus interconnect (Figure 4). The lower portion of the bar shows Directory's runtime with an unimplementable 0-cycle directory access latency. TokenB is still faster by 6 to 18 percent because it avoids the extra interconnect traversal.

How does TokenB's traffic compare to Directory's?

For this specific 16-processor system, Directory uses 21 to 25 percent less traffic than TokenB (not shown). The extra traffic of TokenB over Directory is not as large as you might expect, because

- both protocols send a similar number of messages that contain 64-byte data blocks (81 percent of Directory's traffic on average),
- 8-byte request messages are significantly smaller than data messages, and
- our torus interconnect supports bandwidth-efficient broadcast tree routing.

As the number of processors in the system increases, however, TokenB's traffic grows relative to Directory. Experiments using a simple microbenchmark indicate that, for a 64-processor system, TokenB uses twice the interconnect bandwidth of Directory, making TokenB less attractive for large or bandwidth-starved systems.

Token Coherence as a coherence framework

TokenB is only one specific example of a Token Coherence performance policy. For example, alternative performance policies can conserve bandwidth by

- using a soft-state directory that is fast and usually correct or
- replacing TokenB's broadcasts with predictive multicasts using destination-set prediction.[6]

Table 1. Overhead due to reissued requests.

	Percentages of misses that			
Workload	Are not reissued	Are reissued once	Are reissued more than once	Become persistent requests
Apache	95.75	3.25	0.71	0.29
OLTP	97.57	1.79	0.43	0.21
SPECjbb	97.60	2.03	0.30	0.07
Average	96.97	2.36	0.48	0.19

Performance policies for hierarchical systems, such as collections of chip multiprocessors, might employ mostly correct transient-request filters to reduce on- and off-chip request traffic. Finally, a performance policy could use predictors that learn from recent access patterns to predictively push data and tokens between processors.

Token Coherence makes these enhancements feasible, both individually and in arbitrary combinations. It does so because the performance policy must focus only on common-case performance, without worrying about infrequent race conditions. In all cases, the correctness substrate enforces safety using token counting rules and avoids starvation with persistent requests. To demonstrate this separation, we implemented a performance policy that continually sends transient requests for random blocks to a random set of processors. This performance policy does nothing to actually satisfy cache misses; it simply waits for the correctness substrate to invoke a persistent request. Although this policy performs poorly, the system still functions correctly.

Token Coherence enables a new family of coherence protocols that are both faster and perhaps simpler than snooping and directory protocols. A token coherence protocol is simpler to verify, because only a subset of the protocol—the correctness substrate—needs to be correct. Token coherence protocols can be faster, because decoupling performance and correctness eliminates common-case ordering overheads and encourages aggressive performance optimizations. We plan to further develop Token Coherence, exploring new performance policies, correctness substrate implementations, and verification issues. MICRO

Acknowledgments

We thank Amir Roth and Daniel Sorin for comments on this article, as well as the groups and people that helped with the original paper.[5] This work is supported in part by the National Science Foundation (CCR-0324878, EIA-9971256, EIA-0205286, and CCR-0105721); a Norm Koo Graduate Fellowship and an IBM Graduate Fellowship (Martin); two Wisconsin Romnes Fellowships (Hill and Wood); Spanish Secretaría de Estado de Educación y Universidades (Hill's sabbatical); and donations from Intel, IBM, and Sun Microsystems. Hill and Wood have significant financial interests in Sun Microsystems.

References

1. L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *Proc. 25th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1998, pp. 3-14.
2. A. Charlesworth, "Starfire: Extending the SMP Envelope," *IEEE Micro*, vol. 18, no. 1, Jan.-Feb. 1998, pp. 39-49.
3. S.S. Mukherjee et al., "The Alpha 21364 Network Architecture," *Proc. 9th Symp. High-Performance Interconnects* (HOTI 01), IEEE CS Press, 2001, pp. 113-118.
4. C. Keltcher et al., "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro*, vol. 23, no. 2, Mar.-Apr. 2003, pp. 66-76.
5. M.M.K. Martin, M.D. Hill, and D.A. Wood, "Token Coherence: Decoupling Performance and Correctness," *Proc. 30th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2003, pp. 182-193.
6. M.M.K. Martin, *Token Coherence*, doctoral dissertation, Computer Sciences Dept., Univ. Wisconsin, 2003.
7. A.R. Alameldeen et al., "Simulating a $2M Commercial Server on a $2K PC," *Computer*, vol. 36, no. 2, Feb. 2003, pp. 50-57.
8. J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1997, pp. 241-251.
9. A.R. Alameldeen and D.A. Wood, "Addressing Workload Variability in Architectural Simulations," *IEEE Micro*, vol. 24, no. 6, Nov.-Dec. 2003, pp. 94-98.

Milo M.K. Martin is an assistant professor in the Department of Computer and Information Science at the University of Pennsylvania. His research interests include the memory system performance of commercial workloads, multiprocessor cache coherence, techniques to improve multiprocessor availability, and the use of dynamic feedback to build adaptive and robust systems. Martin has a PhD in computer sciences from the University of Wisconsin-Madison. He is a member of the IEEE and the ACM.

Mark D. Hill is a professor in both the computer sciences department and the electrical and computer engineering department at the University of Wisconsin-Madison, where he also co-leads the Wisconsin Multifacet (http://www.cs.wisc.edu/multifacet) project with David Wood. His research interests include cache design, cache simulation, translation buffers, memory consistency models, parallel simulation, and parallel-computer design. Hill has a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and a member of the ACM.

The biography of **David A. Wood** appears on p. 98 of this issue.

Direct questions and comments about this article to Mark D. Hill, Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706-1685; markhill@cs.wisc.edu.