Token Coherence: Low-Latency Coherence on Unordered Interconnects

Milo M. K. Martin
Mark D. Hill
David A. Wood

Technical Report #1459

November 2002

Technical Report #1459, Computer Sciences Department, University of Wisconsin-Madison — November 2002

Token Coherence: Low-Latency Coherence on Unordered Interconnects

Milo M. K. Martin, Mark D. Hill, and David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
{milo, markhill, david}@cs.wisc.edu

Abstract

Future shared-memory multiprocessor servers will target commercial workloads using highly-integrated "glueless" designs. Commercial workloads, which exhibit frequent sharing misses, benefit from the direct communication of snooping protocols. Unfortunately, snooping systems require a totally-ordered interconnect, which is difficult to efficiently implement in glueless designs. The standard alternative, directory protocols, are a poor match for commercial workloads because the indirection through the directory increases the latency of common sharing misses. An ideal coherence protocol would have processors communicate directly with one another, without indirections or fixed ordering point. Such an approach, however, introduces numerous races that are hard to resolve.

We propose a new coherence framework to enable such protocols by separating performance from correctness. A performance protocol can optimize for the common case (i.e., absence of races) and rely on the underlying correctness substrate to provide safety and liveness. We call the combination Token Coherence, *since it resolves races using the direct exchange of tokens to control coherence permissions.*

Token Coherence provides a framework that can support a wide variety of coherence protocols. This paper develops TokenB, *a specific performance protocol that uses broadcast, but not snooping, for a 16-processor glueless multiprocessor with a high-bandwidth unordered interconnect. Simulations of commercial workloads (using a detailed memory system and out-of-order processor models) show that our new protocol significantly outperforms both snooping and directory protocols.*

1 Introduction

The performance and cost 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 high-performance shared-memory multiprocessors. In our view, software and technology trends point toward a new design space that enables opportunities to increase performance and reduce cost of these servers.

Software trends. Many commercial workloads exhibit abundant thread-level parallelism, making using multiple processors an attractive approach for increasing performance. To efficiently support the frequent communication and synchronization of these workloads [9], these servers should optimize for low-latency sharing misses (i.e., those misses that cause data to move directly between caches). Multiprocessors with 4-16 processors suffice for many workloads, because larger services tend to cluster machines to increase both throughput and availability.

Traditionally, many multiprocessor commercial servers have used snooping cache coherence. Snooping uses totally-ordered broadcast to efficiently satisfy sharing misses in small- to medium-sized multiprocessors. Due to the increasing difficulty of scaling the bandwidth of shared-wire buses [14], some recent snooping systems build "virtual buses" with indirect switched interconnects (e.g., Figure 1a). These interconnects can provide higher bandwidth than buses, at the cost of added switch chips.

Technology trends. The increasing number of transistors per chip predicted by Moore's Law has encouraged and will continue to encourage higher levels of integration. Many current and future multiprocessors will seek to integrate processor(s), cache(s), coherence logic, switch logic, and memory controller(s) on a single die (e.g., Alpha 21364 [34] and AMD's Hammer [5]). Directly connecting these highly-integrated nodes leads to a high-bandwidth, low-cost, low-latency "glueless" interconnect (e.g., Figure 1b).

These glueless interconnects are fast but do not easily provide the virtual bus behavior required by snooping protocols. Instead, most such systems use directory protocols that provide coherence without an ordered interconnect. These systems maintain a directory at the home node (i.e., memory) that orders requests on a per-cache-block basis. However, directory protocols send all requests to the home node, adding an indirection to the critical path of sharing misses—a poor match for commercial workloads.

Token Coherence. Ideally, a coherence protocol would allow processors to communicate directly with one another without indirections or a fixed ordering point. This

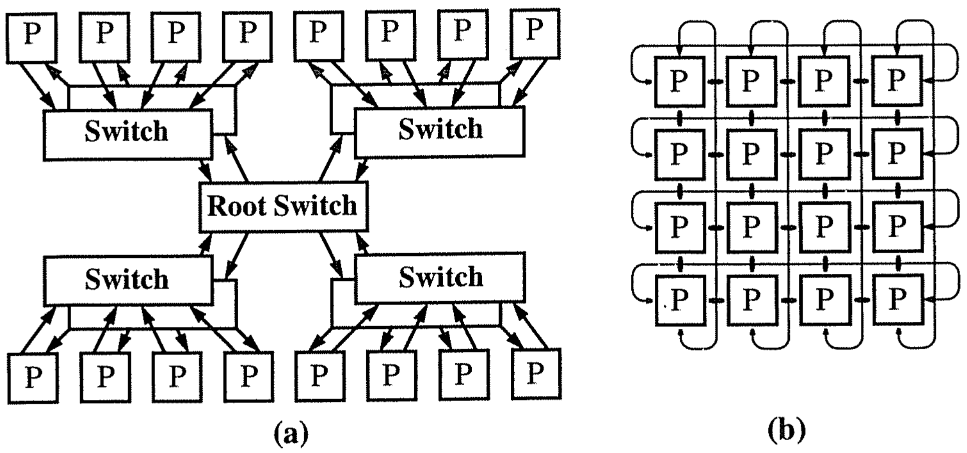


Figure 1. (a) 16 processor two-level tree switched interconnect and (b) 16 processor (4x4) bi-directional torus interconnect. The boxes marked "P" represent highly integrated nodes that include a processor, caches, memory controller, and all coherence controllers. The indirect broadcast tree uses discrete switches, while the torus is a directly connected interconnect. In this example, the torus has lower latency (two vs. four chip-crossings on average) and does not require any glue chips; however, unlike the indirect tree, the torus provides no request total order, making it unsuitable for snooping.

approach has not been used, however, because it suffers from numerous race cases that are difficult to make correct (Section 2). Rather than abandoning this fast approach, we use it to make the common case fast, but we back it up with a substrate to ensure correctness.

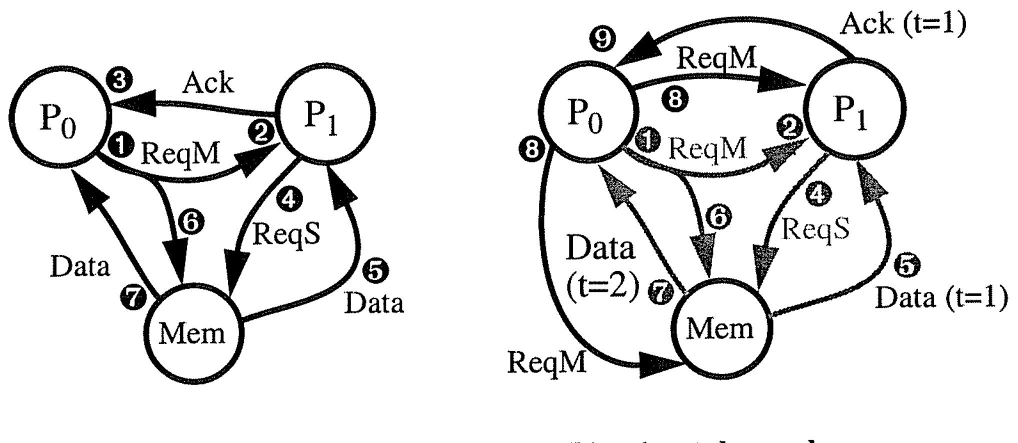
To this end, we propose *Token Coherence,* which has two parts:

- **Correctness Substrate**: This substrate provides a correct coherence protocol for unordered interconnects (Section 3). It associates a fixed number of *tokens* with each logical block and physically stores tokens with copies of the block in caches and coherence messages. It provides safety by ensuring (a) that a processor may only read a cache block if it holds at least one token, and (b) a processor may only write a cache block if it holds all tokens (allowing for a single writer or many readers, but not both). The substrate provides liveness and starvation freedom via *persistent requests*, which a processor invokes when it detects lack of forward progress. Persistent requests always succeed in obtaining data and tokens, because once activated they persist in forwarding data and tokens until the request is satisfied. Finally, the substrate enables performance protocols by allowing caches and memory to exchange data and tokens at any time.
- **Performance Protocol***:* Performance protocols have no correctness requirements, but they use *transient requests* as "hints" to direct the correctness substrate to exchange data and tokens (Section 4.1). In the common case, a transient request succeeds in obtaining the requested data and tokens. However, it may fail to complete, principally due to races. When the protocol suspects a transient request may have failed to complete, it can reissue a transient request one or more times. Since processors will eventually invoke persistent requests directly, performance protocol bugs may hurt performance, but they cannot affect correctness.

Since Token Coherence never incorrectly changes architected state, Token Coherence is **not** a speculative execution technique, and it requires no rollback or recovery mechanism.

TokenB. We target small- to medium-sized glueless multiprocessors with a specific performance protocol called *Token-Coherence-using-Broadcast* or *TokenB* (Section 4.2). In *TokenB*, processors broadcast transient requests and respond like a traditional MOSI protocol. This performance protocol directly finds the data in the common case of no races, allowing for low-latency sharing misses. When transient requests fail (due to races), the protocol reissues them until the processor times out and invokes a persistent request to guarantee liveness.

For selected commercial workloads on a full-system simulation of a 16-processor system (described in Section 5), we find that (a) reissued and persistent requests are rare (3.0% and 0.2% of requests, respectively), (b) *TokenB* is faster than snooping, because it allows use of an unordered interconnect (26-65% faster), (c) *TokenB* is faster than a directory protocol, because it avoids directory indirections (17-54%), and (d) a directory protocol uses less bandwidth than *TokenB* (21-25%), but this additional bandwidth is not a problem for the high-bandwidth glueless interconnects that will be common in future systems. We present these and other results in Section 6.



(a) fast but incorrect **(b) using token coherence**

Figure 2. Motivating Example: A shared and modified request race.

While *TokenB* provides an attractive alternative for small-to medium-sizes systems, Token Coherence permits other performance protocols that can reduce traffic for larger systems, use prediction to push data, and support hierarchy with low complexity (Section 7).

2 A Motivating Example

In this section, we first present a simple coherence race to illustrate that naively sending requests without an ordering point is incorrect. We then review how the race is handled by snooping protocols (by ordering requests in the interconnect) and directory protocols (by ordering requests at the home node). Finally, we forecast how Token Coherence handles this and other races.

Fast (but incorrect) approach. Invalidation-based coherence protocols provide the illusion of a single memory shared by all processors by allowing either a single writer or many readers, but not both at the same time. A fast way to obtain read or write permission to a cache block would allow requesters to broadcast requests to all other processors over a low-latency unordered interconnect. Doing this naively, however, can cause coherence errors when processors act on racing requests in different orders.

Consider the example illustrated in Figure 2a, in which processor P_0 desires read/write access to the block (i.e., MOESI [44] state *modified* or *M*), and processor P_1 desires read-only access (i.e., state *shared* or *S*). P_0 broadcasts its request at time ❶, which the interconnect promptly delivers to P_1 at time ❷ but belatedly to memory at time ❻ (e.g., due to contention on the unordered interconnect). Processor P_1 handles P_0's request but takes no action other than an acknowledgment (Ack) at time ❸, since it lacks a valid copy. Now at time ❹, P_1 issues its request, which the memory quickly satisfies at time ❺. Finally, P_0's delayed request arrives at memory at time ❻, and the memory satisfies the request at time ❼. After receiving the response, P_0 believes (erroneously) that it has a writable copy of the block, but P_1 still holds a read-only copy. If this situation arises, a violation of the memory consistency model—the definition of memory access correctness in a multiprocessor system—is almost certain to occur.

Snooping protocols. Snooping protocols resolve this and other races by relying on the interconnect—a virtual bus—to provide a total order of all requests. This ordering ensures that all processors (and memories) observe requests in the same order (including their own requests, to establish their place in the total order). In our example race, request ordering was not consistent with a total order. P_1 observed its request as occurring *after* P_0's request, while memory observed P_1's request *before* P_0's request. A total order would guarantee correct operation because either P_0's invalidation would have occurred before P_1's request (and thus P_0 would respond to P_1's request), or P_0's request would have arrived at P_1 after P_1's request (invalidating P_1's shared copy of the block). Unfortunately, interconnects that enforce a total order may have higher latency or cost.

Directory protocols. Directory protocols resolve this and other races without an ordered interconnect by providing a per-block ordering point at the directory. The directory prevents the race by determining which request will be processed first, using forwarded requests, invalidations, and explicit acknowledgements to enforce ordering (and possibly blocking or using negative acknowledgments in some cases). In our example, P_1's request arrives at the home memory/directory first, and the home provides data. P_0's request arrives later, and the home forwards an invalidation message to P_1. P_0's request completes after receiving an acknowledgment from P_1 and data from the home, knowing that no other copies are present in the system. Unfortunately, the cost of this solution is an added level of indirection on the critical path of sharing misses.

Token Coherence. Token Coherence allows this and other races to occur but provides correctness with a *correctness* *substrate* (described in Section 3) that ensures processors only read/write coherent blocks appropriately (safety), that guarantees processors eventually obtain a needed block (liveness and starvation avoidance), and allows blocks to move at any time. *Performance protocols* (described in Section 4) seek to make the common case fast with requests for data movement that do not always succeed.

3 Correctness Substrate

The correctness substrate enforces safety via token counting (do no harm), and it provides liveness via persistent requests (do some good). These two mechanisms allow the substrate to move data around the system without regard to order, allowing processors to read and write the block as appropriate, but still allows requests for a block to eventually succeed in all cases.

3.1 Safety via Token Counting

The correctness substrate uses tokens to ensure safety without requiring any fixed ordering point. Processors track explicit *tokens*, and a processor is only allowed to read or write a cache block when holding at least one token or all tokens, respectively. At system initialization time, the system assigns each block *T* tokens (where *T* is at least as large as the number of processors). Initially, the block's home memory module holds all tokens for a block. Later, tokens will be held by processor caches and coherence messages. An unoptimized substrate maintains these five invariants:

- **Invariant #1**: At all times, each block has *T* tokens in the system.
- **Invariant #2**: A processor can write a block only if it holds all *T* tokens.
- **Invariant #3**: A processor can read a block only if it holds at least one token.
- **Invariant #4**: If a coherence message contains data, it must contain at least one token.
- **Invariant #5**: If a coherence message contains one or more tokens, it must contain data.

Invariant #1 ensures that the substrate never creates or destroys tokens. Invariants #2 and #3 ensure that a processor will not write the block while another processor is reading it. Adding invariants #4 and #5 ensure that data and tokens travel together. In more familiar terms, token possession maps directly to traditional coherence states: holding all *T* tokens is *modified* (M); one to *T-1* tokens is *shared* (S); and no tokens is *invalid* (I).

The token-based correctness substrate enforces these invariants directly by counting tokens. The substrate maintains these invariants by induction; the invariants hold for the initial system state, and all movements of data and tokens preserve the invariants. Thus, safety is ensured without reasoning about the interactions among stable and transient protocol states, data responses, acknowledgment messages, interconnect order, and system hierarchy.

Token Coherence enforces a memory consistency model [4]—the definition of correctness for multiprocessor systems—in a similar manner to directory protocols. The above guarantee of a "single writer" or "multiple readers with no writer" is the same property provided by many invalidation-based directory protocols. For example, the MIPS R10k processors [46] in the Origin 2000 [26] use this guarantee to provide sequential consistency, even without a global ordering point[1]. The Origin protocol uses explicit invalidation acknowledgments to provide the above guarantee. We provide the same guarantee by explicitly tracking tokens for each block. As with any coherence scheme, the processors are also intimately involved in enforcing the memory consistency model.

Optimized token counting. An issue with the invariants above is that data must always travel with tokens, even when gathering tokens from shared copies (like invalidation acknowledgments in a directory protocol). To avoid this inefficiency, the substrate actually distinguishes a separate owner token, adds a data valid bit (distinct from the traditional tag valid bit), and maintains the following five invariants (changes in *italics*):

- **Invariant #1'**: At all times, each block has *T* tokens in the system, *one of which is distinguished as the owner token.*
- **Invariant #2'**: A processor can write a block only if it holds all *T* tokens.
- **Invariant #3'**: A processor can read a block only if it holds at least one token *and has valid data.*
- **Invariant #4'**: If a coherence message contains data, it must contain at least one token.
- **Invariant #5'**: If a coherence message contains *the owner token*, it must contain data.

Invariants #2' and #3' continue to provide safety. Invariants #4' and #5' allow coherence messages with non-owner tokens to omit data, but they still require that messages with the owner token contain data (to prevent all processors from simultaneously discarding data). Possession of the owner token but not all other tokens maps to the familiar MOESI state *owned* (O). System components (processors and the home memory) maintain a valid bit, to allow components to hold non-owner tokens without data. A component sets the valid bit when data arrives, and it clears the valid bit when it no longer holds any tokens.

1. The Origin protocol uses a directory to serialize some requests **for the same block**; however, since memory consistency involves the memory ordering relationship between different memory locations [4], using a distributed directory is not alone sufficient to provide memory consistency.

Tokens must be stored in processor caches (e.g., part of tag state), memory (e.g., encoded in ECC bits [18, 34, 36]), and coherence messages[2]. Tokens can be stored in $2+\lceil \log_2 T \rceil$ bits (valid bit, owner token, and non-owner token count), since we do not track which processors hold them but only count them. For example, supporting 64 tokens with 64 byte blocks adds one byte of storage (1.6% overhead).

Finally, there is important freedom in what the invariants do *not* specify. While our invariants restrict the data and token content of coherence messages, the invariants do not restrict *when* or *to whom* the substrate can send coherence messages. For example, to evict a block (and thus tokens) from a cache, the processor simply sends all its tokens (and data if the message includes the owner token) to the memory. Likewise, anytime a processor receives a message carrying tokens (with or without data), it can either choose to accept it (e.g., if there is space in the cache) or redirect it to memory (using another virtual network to avoid deadlock). We use this freedom in three additional ways. First, we define persistent requests to ensure liveness (Section 3.2). Second, we define transient requests that allow a performance protocol to "hint" to whom the substrate should send data and tokens (Section 4.1). Third, this freedom enables many performance protocols (Section 4.2 and Section 7).

3.2 Liveness via Persistent Requests

The correctness substrate provides persistent requests to guarantee liveness and starvation freedom. A processor invokes a persistent request whenever it detects lack of forward progress (e.g., it has failed to complete a cache miss within a timeout period). Since results in Section 6 show that processors resort to persistent requests on only 0.2% of misses, persistent requests must be correct but not necessarily fast. The substrate uses persistent requests to guarantee liveness by performing the following steps:

- A processor detects lack of forward progress and initiates a persistent request.
- The substrate *activates* at most one persistent request per block.
- System nodes remember all *activated* persistent requests and forward all tokens for the block—those tokens currently present and received in the future—to the initiator of the request.
- When the initiator has sufficient tokens, its performs a memory operation (e.g., load/store instruction) and *deactivates* its persistent request.

Persistent requests guarantee starvation freedom if the system implements a fair mechanism for activating persistent requests.

2. Like most coherence protocols (e.g., [12, 26, 34, 45]), we assume the interconnect provides reliable message delivery.

Implementation. Processors invoke a persistent request when a cache miss has not been satisfied within ten average miss times. The correctness substrate implements persistent requests with a simple *arbiter* state machine at each home memory module. The substrate directs persistent requests to the home node of the requested block. Requests may queue in a virtual network or at memory. The state machine activates one request and informs all nodes, who respond with acknowledgements (to avoid races). Each node remembers all active persistent requests, using a table of about 8 bytes times the number of home memory nodes (e.g., a 64-node system requires only a 512-byte table at each node). Nodes must forward all tokens (and data, if they have the owner token) to the requester. The node will also forward tokens (and data) that arrive later, because the request persists until the requester explicitly deactivates it. Once the requester is satisfied, it informs the home memory to deactivate the request. The state machine at the home memory deactivates the request by informing all nodes, who delete the entry from their table and send an acknowledgement (again, to eliminate races). Figure 3 shows the general operation of our implementation of the correctness substrate.

Although this implementation of persistent requests is simple and inefficient, it performs well for our workloads, since processors rarely invoke persistent requests (as shown in Section 6). To more efficiently handle blocks with highly-contended locks, we are exploring a more aggressive implementation that activates persistent requests using timestamps [24, 39]. The benefit of timestamps is that blocks with multiple outstanding requests can often move directly between requesters. We do not use this implementation in this paper, because its correctness relies on much more subtle reasoning than our simple state machine.

4 Performance Protocols

This section first discusses performance protocol requirements and then presents *TokenB*, a performance protocol targeting small- and medium-sized glueless multiprocessors.

4.1 Obligations and Opportunities

Obligations. *Performance protocols have no obligations*, because the processors and correctness substrate ensure correctness. A null or random performance protocol would perform poorly but not incorrectly. Therefore, performance protocols may aggressively seek performance without concern for corner-case errors.

Opportunities via Transient Requests. One way in which performance protocols seek high performance is by specifying a policy for using *transient requests*. Transient requests are fast, unordered "hint" requests sent to one or more nodes that may fail to obtain a readable or writable block due to races, insufficient recipients, or being ignored by the

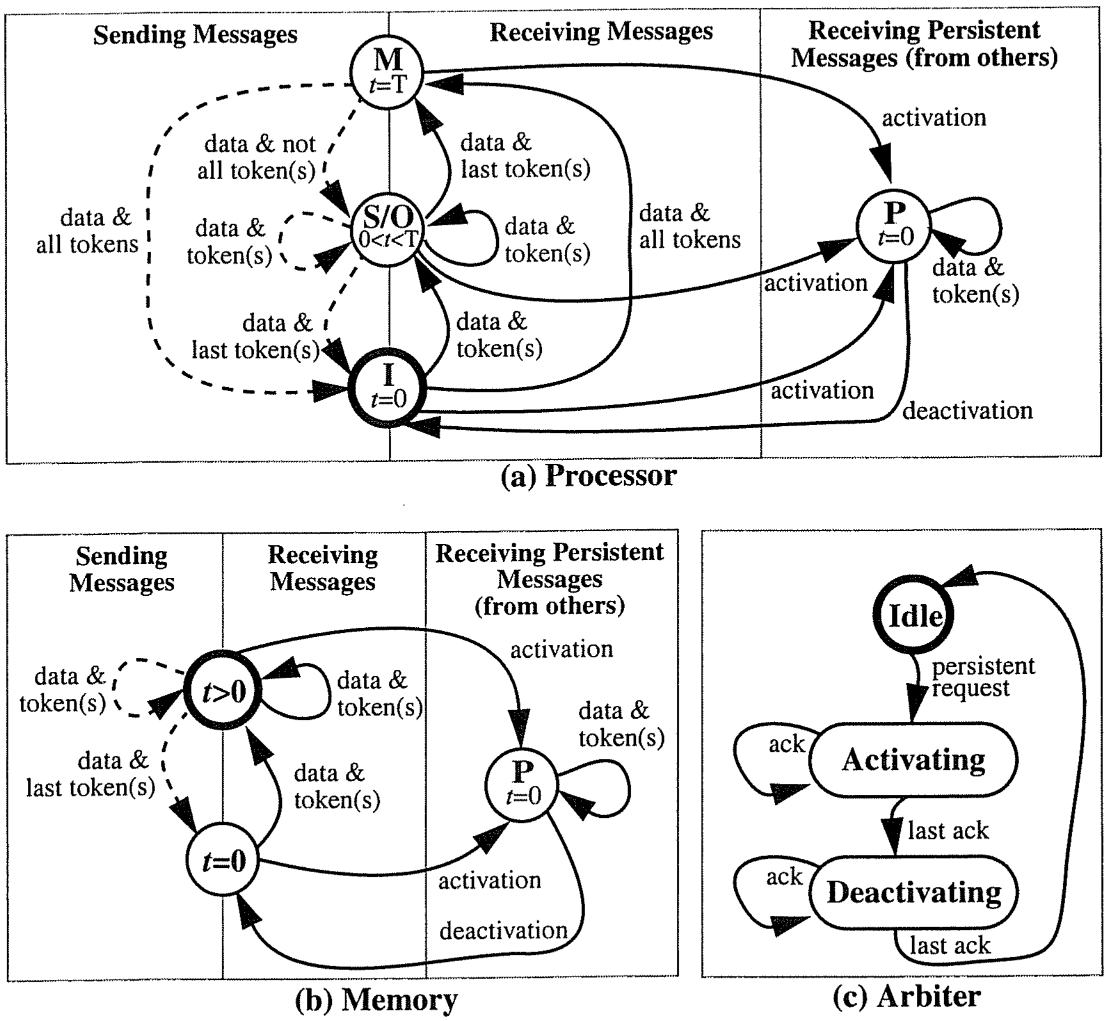


Figure 3. Protocol state transition diagram for the (a) processor, (b) memory, and (c) persistent request arbiter. As a simplification, the figure only shows tokens sent with data. The symbol *t* represents the current token count, and *T* represents all the tokens. Solid arcs are transitions in response to incoming messages. Dashed arcs are transitions a performance protocol (Section 4) can invoke at any time (e.g., when receiving a transient request). The *"P"* states occur when a node receives a persistent request from the arbiter for another processor. A processor must also remember its own persistent request, not shown explicitly in this figure. The initial states are emphasized with thick boarders.

correctness substrate. Performance protocols can detect when a request has not (yet) succeeded, because the requester has not obtained enough tokens (i.e., one token to read the block, and all tokens to write it). Performance protocols may reissue transient requests or do nothing (since the processor will eventually timeout and issue a persistent request).

A performance protocol also specifies a policy for how system components respond to transient requests. If a transient request for a shared block encounters data with all tokens, for example, a performance protocol can specify whether to reply with the data and one token or the data and all tokens (much like a migratory sharing optimization [13, 43]). Active persistent requests always override performance protocol policies to provide liveness.

A good performance protocol will use transient requests to quickly satisfy most cache misses. In the example of Section 2 and Figure 2, both processors could broadcast transient requests. Even though the requests race, frequently both processor's misses would be satisfied. In other cases, one or both may not succeed (detected by insufficient tokens), but the performance protocol can reissue those transient requests.

4.2 *TokenB*: A Performance Protocol for Glueless Multiprocessors

The *TokenB* performance protocol uses three key policies to target glueless multiprocessors with high-bandwidth unordered interconnects.

Issuing transient requests. Processors broadcast all transient requests. This policy works well for moderate-sized systems where interconnect bandwidth is plentiful and racing requests are rare.

Responding to transient requests. Components (processors and the home memory) respond[3] to (transient) requests as they would in most MOSI protocols. A component with no tokens (state *I*) ignores all requests. A component with only non-owner tokens (state *S*) ignores shared requests, but on an exclusive request it sends all its tokens in a dataless message (like an invalidation acknowledgment). A component with the owner token but not all other tokens (state *O*) sends the data with one token (usually not the owner token) on a shared request, and it sends the data and all its tokens on an exclusive request. A component with all the tokens (state *M*) responds the same way as a component in state *O*, with the exception given in the next paragraph.

To optimize for common migratory sharing patterns, we implement an optimization for migratory data [13, 43]. If a processor with all tokens (state *M*) has written the block, and it receives a shared request, it responds with the data and all tokens (instead of the data and one token). By transferring read/write permission on some read requests, this optimization halves the coherence requests for migratory sharing patterns. However, it may reduce performance if the block is later read by the same processor that last wrote it. To enable a fair comparison, we implement the analogous optimization in the all base protocols used for quantitative comparison.

Reissuing transient requests. If a transient request has not completed after a reasonable interval, we reissue the transient request. We continue to reissue transient requests until the processor issues a persistent request (approximately 4 times). We use both a small randomized exponential backoff (much like ethernet) and twice the recent average miss latency to calculate the reissue timeout. This policy adapts to the average miss latency of the system (to avoid reissuing too soon), and it quickly reissues requests that do not succeed due to occasional races. Since races are rare, on average only 3.0% of all misses are reissued even once (for our workloads and simulation assumptions, described next).

Example. Returning to the example in Section 2 (Figure 2b), the block has three tokens that are initially held by memory. P_1 received one token in the response at time ❻, allowing it to read the block. Due to the race, P_0 only received two tokens in the response at time ❼ but requires all three before it can write the block. After the specified interval, *TokenB* reissues P_0's request (ReqM) at time ❽. P_1 responds with a message containing the missing token (Ack) at time ❾, allowing P_0 to finally complete its request.

3. Technically, the performance protocol asks the correctness substrate to respond on its behalf.

Table 1. Benchmark descriptions

Static Web Content Serving: Apache. Web servers such as Apache are an important server application. We use Apache 2.0.39 for Solaris 8 configured to use pthread locks and minimal logging at the web server. Our experiments use a repository of 20,000 files and 160 simulated users. Our results are based on runs of 10,000 requests.
Java Server Workload: SPECjbb. SPECjbb2000 is a server-side java benchmark that models a 3-tier system, focusing on the middleware server business logic. We use Sun's HotSpot 1.4.0 Server JVM. Our experiments use 24 threads and 24 warehouses (a data size of ~500MB). Our results are based on runs of 2,000 transactions.
Online Transaction Processing (OLTP): DB2 with a TPC-C-like workload. The TPC-C benchmark models the database activity of a wholesale supplier, with many concurrent users performing transactions. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use an 800MB database with 4000 warehouses stored on five raw disks and an additional dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow for concurrency provided by a larger number of warehouses. There are 128 simulated users. Our results are based on runs of 100 transactions.

5 Evaluation Methods

To evaluate Token Coherence, we simulate a multiprocessor server running commercial workloads using multiple coherence protocols and interconnection networks. Our target is a 16-processor SPARC v9 system with highly integrated nodes that include deeply pipelined dynamically scheduled processors, two levels of cache, coherence protocol controllers, and a memory controller for part of the globally shared memory. The system implements sequential consistency using invalidation-based cache coherence and an aggressive speculative processor implementation [19, 46].

Due to the growing prevalence of information services, commercial workloads are increasingly important for high performance multiprocessor systems. Our benchmarks consist of commercial workloads, such as database and web servers, as described in Table 1. We refer interested readers to Alameldeen *et al.* [6] for more detailed description and characterization of these workloads.

We selected a number of coherence protocols, interconnection networks, latencies, bandwidths, cache sizes, and other structure sizes. Table 2 lists the system parameters for both the memory system and the processor, chosen to approximate the published parameters of systems like the Alpha 21364 [34]. The coherence protocols and interconnection networks are described below.

5.1 Coherence Protocols

We compare target systems using four distinct MOSI coherence protocols. They all implement the previously

Table 2. Target System Parameters

Coherent Memory System		Dynamically Scheduled Processor	
split L1 I & D caches	128kBytes, 4-way, 2ns lat.	clock frequency	1 Ghz
unified L2 cache	4MBytes, 4-way, 6ns lat.	reorder buffer/scheduler	128/64 entries
cache block size	64 Bytes	pipeline width	4-wide fetch & issue
DRAM/dir. latency	80ns (2 GBytes of DRAM)	pipeline stages	11
memory/dir. controllers	6ns latency	direct branch predictor	1kBytes YAGS
network link bandwidth	3.2 GBytes/s	indirect branch predictor	64 entry (cascaded)
network link latency	15ns (incl. wire, sync. & route)	return address stack	64 entry

described migratory sharing optimization that improves the performance of all the protocols. All request, acknowledgment, invalidation, and dataless token messages are 8 bytes in size (including the 40+ bit physical address and token count if needed); data messages include this 8 byte header and 64 bytes of data. We compare the *TokenB* protocol (described in Section 4) with three other coherence protocols:

Snooping. We based our snooping protocol on a modern protocol [12], but we add additional non-stable states to relax synchronous timing requirements. To avoid the complexity and latency of a snoop response combining tree to implement the "owner" signal, the protocol uses a single bit in memory to determine when the memory should respond to requests [17].

Directory. We use a standard full-map directory protocol inspired by the Origin 2000 [26] and Alpha 21364 [20]. The protocol requires no ordering in the interconnect and does not use negative acknowledgments (nacks) or retries, but it does queue requests at the directory controller during some directory transitions. The system stores the directory state in the main memory DRAM [18, 34, 36], but we also evaluate a "perfect" directory cache with experiments that use a zero cycle directory access latency.

Hammer. We use a reverse-engineered approximation of AMD's Hammer protocol [5] to represent a class of recent systems whose protocols are not described in the academic literature (e.g., Intel's E8870 "Scalability Port" [8], IBM's Power4 [45] and xSeries "Summit" [11] systems). The protocol targets small systems (where broadcast is okay) with unordered interconnects (where snooping is not possible), while avoiding directory state overhead and directory access latency. A processor first sends its request to a home node to be queued behind other requests to the same block. In parallel with the memory access, the home node broadcasts the request to all nodes who respond to the requester with data or an acknowledgment. Finally, the requester sends a message to unblock the home node. This protocol reduces the overhead and latency compared to standard directory protocols, but it still requires indirections for sharing misses.

5.2 Interconnection Networks

We selected two interconnects with high-speed point-to-point links: an ordered "virtual bus" pipelined broadcast tree (sufficient for snooping) and an unordered torus. We do not consider shared-wire (multi-drop) buses, because designing high-speed buses is increasingly difficult due to electrical issues [14, section 3.4.1]. We selected the link bandwidth of 3.2 GBytes/sec (4-byte wide links at 800 Mhz) and latency of 15 ns based on descriptions of current systems (e.g., the Alpha 21364 [34] and AMD's hammer [5]). Messages are multiplexed over a single shared interconnect using virtual networks and channels, and broadcast messages use bandwidth-efficient tree-based multicast routing [15, section 5.5].

Tree (Figure 1a). For our totally ordered interconnect, we use a two-level hierarchy of switches to form a pipelined broadcast tree with a fan-out of four, resulting in a message latency of four link crossings. This tree obtains the total order required for snooping by using a single switch at the root. To reduce the number of pins per switch, a 16-processor system using this topology has nine switches (four incoming switches, four outgoing switches, and a single root switch).

Torus (Figure 1b). For our unordered interconnect, we use a two-dimensional, bidirectional torus like that used in the Alpha 21364 [34]. A torus has reasonable latency and bisection bandwidth, especially for small to mid-sized systems. A 16-processor 2D-torus has an average message latency of two link crossings.

5.3 Simulation Methods

We simulate our target systems with the Simics full-system multiprocessor simulator [29], and we extend Simics with a processor and memory hierarchy model to compute execution times. Simics is a system-level architectural simulator developed by Virtutech AB that can run unmodified commercial applications and operating systems. Simics is a functional simulator only, but it provides an interface to support our detailed timing simulation. We use TFsim [32], configured as described in Table 2, to model superscalar processor cores that are dynamically scheduled, exploit speculative execution, and generate multiple outstanding

Table 3. Overhead due to reissued requests

	Percentage of Misses				Percentage of Traffic
Workload	**Not Reissued**	**Reissued Once**	**Reissued More Than Once**	**Persistent Requests**	**Reissued and Persistent Requests**
Apache	95.75%	3.25%	0.71%	0.29%	3.04%
OLTP	97.57%	1.79%	0.43%	0.21%	1.80%
SPECjbb	97.60%	2.03%	0.30%	0.07%	1.21%
Average	96.97%	2.36%	0.48%	0.19%	2.02%

coherence requests. Our detailed memory hierarchy simulator models the latency and bandwidth of the interconnects described above, and it also captures timing races and all state transitions (including transient states) of the coherence protocols. All workloads were warmed up and checkpointed to avoid system cold-start effects, and we ensure that caches are warm by restoring the cache contents captured as part of our checkpoint creation process. To address the variability in commercial workloads, we adopt the approach of simulating each design point multiple times with small, pseudo-random perturbations of request latencies to cause alternative operating system scheduling paths in our otherwise deterministic simulations [7]. Error bars in our runtime results represent one standard deviation from the mean in each direction.

6 Evaluation via Five Questions

We present evidence that Token Coherence can improve performance via five questions.

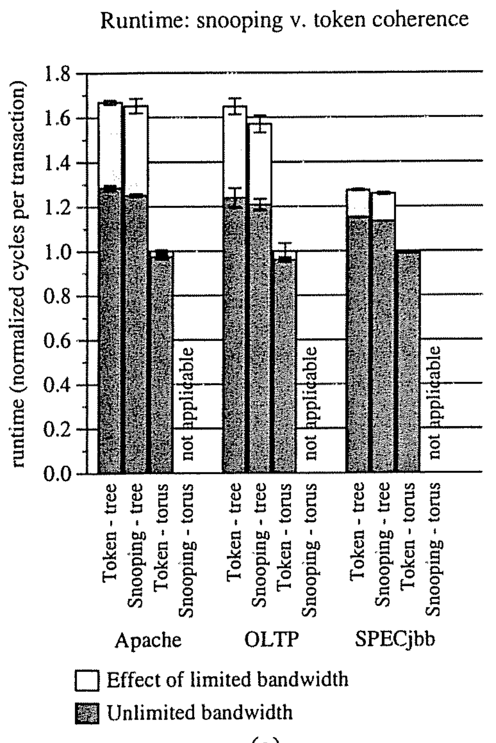
Question #1: Can the number of reissued and persistent requests be small? Answer: **Yes**; on average for our workloads, **97% of *TokenB*'s cache misses are not reissued even once**. Since reissued requests are slower and consume more bandwidth than misses that succeed on the first attempt, reissued requests must be uncommon for *TokenB* to perform well. Races are rare in our workloads, because—even though synchronization and sharing are common—multiple processors rarely access the same data simultaneously due to the large amount of shared data. The five left-most columns of Table 3 show the percentage of all *TokenB* misses that are not reissued, reissued once, reissued more than once, and that eventually use persistent requests. For our workloads, on average only 3.0% of cache misses are issued more than once and only 0.2% resort to persistent requests. The right-most column of Table 3 show the extra traffic is, unsurprisingly, also small. (Table 3 shows *Torus* interconnect results, but *Tree* results, not shown, are similar.)

Question #2: Can Token Coherence outperform a snooping protocol? Answer: **Yes**; with the same interconnect, *TokenB* and *Snooping* perform similarly for our workloads; however, by exploiting the lower-latency unordered *Torus*, ***TokenB* on the *Torus* is faster than *Snooping* on the *Tree* interconnect** (26-65% faster). Figure 4a shows the normalized runtime (smaller is better) of *TokenB* on the *Tree* and *Torus* interconnects and *Snooping* on the *Tree* interconnect. *Snooping* on the *Torus* is not applicable, because the *Torus* lacks the required total order. The dark grey bar shows the runtime when the bandwidth is changed from 3.2 GB/s to unlimited. Figure 4b shows the traffic in normalized average bytes per miss.

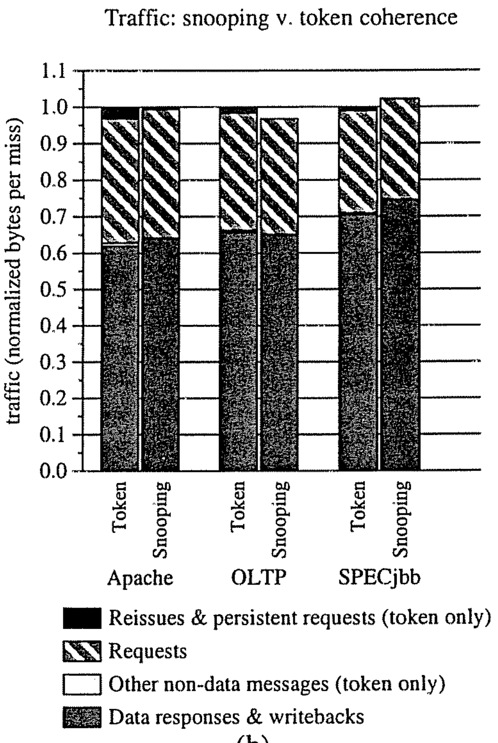
On the *Tree* interconnect, due to *TokenB*'s occasionally reissued requests, *Snooping* is slightly faster than *TokenB* (1-5% and 1-3%) with both limited and unlimited bandwidth, respectively (Figure 4a), and both protocols use approximately the same interconnect bandwidth (Figure 4b). However, since *Snooping* requires a totally ordered interconnect, only *TokenB* can exploit a lower-latency unordered interconnect. Thus, by using the *Torus*, *TokenB* is 26-65% faster than *Snooping* on *Tree* with limited bandwidth links, and 15-28% faster with unlimited bandwidth links. This speedup results from (1) lower latency for both sharing and non-sharing misses (due to lower average interconnect latency), and (2) lower contention in *Torus* (by avoiding *Tree*'s central bottleneck at the root).

Question #3: Can Token Coherence outperform directory protocols? Answer: **Yes**; by removing the latency of indirection through the home node from the critical path of sharing misses, ***TokenB* is faster than both *Directory* and *Hammer*** (17-54% and 8-29% faster, respectively). Figure 5a shows the normalized runtime (smaller is better) for *TokenB*, *Hammer*, and *Directory* on the *Torus* interconnect with 3.2 GB/second links (the relative performances on *Tree*, not shown, are similar). The light grey bars illustrate the small increase in runtime due to limited bandwidth in the interconnect. The grey striped bar for *Directory* illustrates the runtime increase due to the DRAM directory lookup latency.

TokenB is faster than *Directory* and *Hammer* due to (1) avoiding the third interconnect traversal for sharing misses, (2) avoiding the directory lookup latency (*Directory* only), and (3) removing blocking states in the memory controller. Even if the directory lookup latency is reduced to zero (to approximate a fast SRAM directory or directory cache), shown by disregarding the grey striped bar in Figure 5a, *TokenB* is still faster than *Directory* by 6-18%. *Hammer* is



(a)



(b)

Figure 4. *Snooping* v. *TokenB*: runtime and traffic

7-17% faster than *Directory* by avoiding the directory lookup latency (but not the third interconnect traversal), but *Directory* with the zero-cycle directory access latency is 2-9% faster than *Hammer* due to contention in the interconnect. The performance impact of *TokenB*'s additional traffic is negligible, because (1) the interconnect has sufficient bandwidth due to high-speed point-to-point links, and (2) the additional traffic of *TokenB* is moderate, discussed next.

Question #4: How does *TokenB*'s traffic compare to *Directory* and *Hammer*? Answer: **TokenB generates less interconnect traffic than *Hammer*, but a moderate amount more than *Directory*** (*Hammer* uses 79-90% more traffic than *TokenB*, *Directory* uses 21-25% less than *TokenB*). Figure 5b shows a traffic breakdown in normalized bytes per miss (lower is better) for *TokenB*, *Hammer*, and *Directory*. The extra traffic of *TokenB* over *Directory* is not as large as one might expect, because (1) both protocols send a similar number of 72-byte data messages (81% of *Directory*'s traffic on average), (2) request messages are small (8 bytes), and (3) *Torus* supports broadcast tree routing (as stated in Section 5.2). *Hammer*, which targets smaller systems, uses much more bandwidth than *TokenB* or *Directory*, because every processor acknowledges each request (shown by the light grey striped segment).

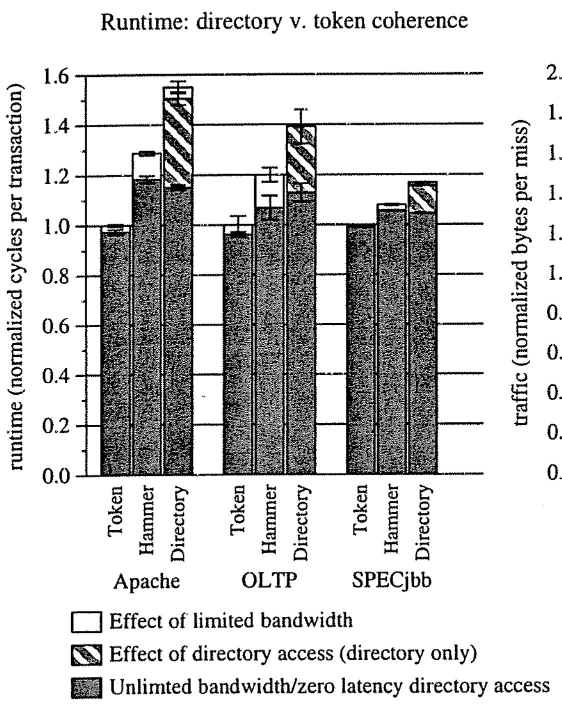
Question #5: Can the *TokenB* protocol scale to an unlimited number of processors? Answer: No; ***TokenB* relies on broadcast, limiting its scalability. However, Token Coherence is not limited to always broadcasting.** *TokenB* is more scalable than *Hammer*, because *Hammer* uses broadcast and many acknowledgment messages. *TokenB* is less scalable than *Directory*, because *Directory* avoids broadcasts. However, *TokenB* can perform well for perhaps 32 or 64 processors if bandwidth is abundant (by using high-bandwidth links [22] and coherence controllers with high throughput [35, 38] and low power [33]). Experiments (not shown) using a simple microbenchmark indicate that, for a 64 processor system, *TokenB* uses twice the interconnect bandwidth of *Directory*[4]. However, *TokenB* is a poor choice for larger or bandwidth-limited systems. For this reason, the next section discusses other potential performance protocols.

7 Opportunities for Exploiting the Generality of Token Coherence

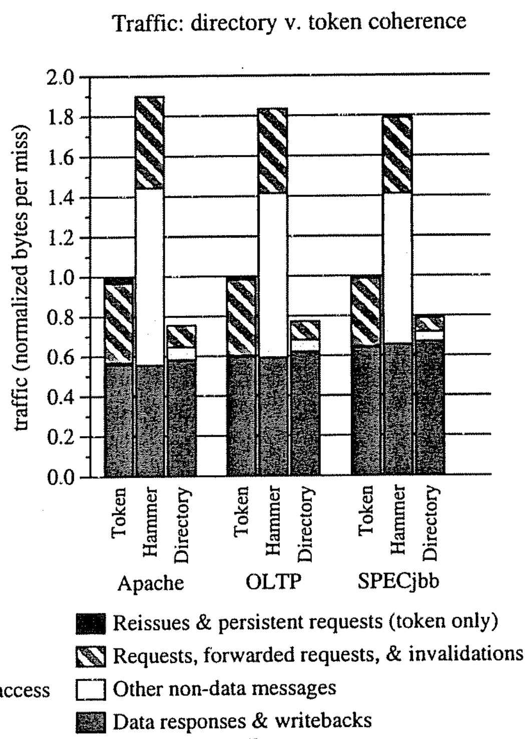
Token Coherence enables many performance protocols beyond the broadcast-always *TokenB* protocol. Furthermore, since its correctness substrate guarantees safety and liveness, performance protocol designers can innovate without fear of corner-case correctness errors.

Reducing traffic. We can reduce request traffic (by not broadcasting transient requests) in several ways. First, we can reduce the traffic to directory protocol-like amounts by constructing a directory-like performance protocol. Processors first send transient requests to the home node, and the

4. The additional cost of tree-based broadcast on *Torus* grows as Θ(sqrt(n)).



(a)



(b)

Figure 5. Directory protocols and *TokenB*: runtime and traffic

home redirects the request to likely sharers and/or the owner by using a "soft state" directory. Second, bandwidth-adaptive techniques would allow a system to dynamically adapt between *TokenB* and this directory-like mode, providing high performance for multiple system sizes and workloads [31]. Third, Token Coherence can use destination set prediction [2, 3, 10] to achieve the performance of broadcast while using less bandwidth by predicting a subset of processors to which to send requests. Previously, these proposals required complicated protocols or protocol extensions. By multicasting transient requests, Token Coherence provides a simpler implementation of these proposals, while removing the total order required by some proposals [10] and complex races in other proposals [2, 3, 10].

Predictive push. The decoupling of correctness and performance provides an opportunity to reduce the number of cache misses by predictively pushing data between system components. This predictive transfer of data can be triggered by a coherence protocol predictor [1, 23, 37], triggered by software (e.g., the KSR1's "poststore" [40] and DASH's "deliver" [27]), or allow the memory to push data into processor caches. Since Token Coherence allows data and tokens to be transferred between system components without affecting correctness, these schemes are easily implemented correctly as part of a performance protocol.

Hierarchical system support. Token Coherence can also accelerate hierarchical systems, an increasingly important concern with the rise of chip multiprocessors CMPs (e.g., Power4 [45]). Power4 uses extra protocol states to allow neighboring processors to respond with data, reducing traffic and average miss latency. A performance protocol could achieve this more simply by granting extra tokens to requesters, and allow those processors to respond with data and tokens to neighboring processors. Other hierarchical systems connect smaller snooping based modules into larger systems (e.g., [8, 11, 27]). Token Coherence may allow for a single protocol to more simply achieve the latency and bandwidth characteristics of these hierarchical systems, without requiring the complexity of two distinct protocols and the bridge logic between them.

8 Related Work

Protocol and interconnect co-design. Timestamp Snooping [30] adds ordering sufficient for snooping to an unordered interconnect by using timestamps and reordering requests at the interconnect end points. Other approaches eschew virtual buses by using rings or a hierarchy of rings [16, 40, 45] or race-free interconnects [25]. Token Coherence addresses similar problems, but instead uses a new coherence protocol on an unordered interconnect to remove indirection in the common case.

Coherence protocols. Acacio et al. separately target read miss latency [2] and write miss latency [3] by augmenting a directory protocol with support for predicting current holders of the block. In many cases, the system grants permission without indirection, but in other cases, prediction is not allowed, requiring normal directory-based ordering and directory indirection. In contrast, we introduce a simpler and unified approach that allows for correct direct communication in all cases, only resorting to a slower mechanism for starvation prevention. Shen et al. [41] use term rewriting rules to create a coherence protocol that allows operations from any of several sub-protocols, forming a trivially-correct hybrid protocol. We similarly provide a correctness guarantee, using tokens to remove ordering from the common case. The Dir$_1$SW directory protocol [21] keeps a count of sharers at the memory for detecting the correct use of check-in/check-out annotations. In addition, Stenström [42] proposes systems that use limited directory state and state in caches to track sharers. Token Coherence uses similar state in the memory and caches to count tokens, but it uses the state to enforce high-level invariants that avoid directory indirection in the common case. Li and Hudak [28] explore a protocol in which each node tracks a probable owner, allowing requests to quickly find the current owner of the line. This approach could be used to improve a performance protocol or persistent request mechanism.

9 Conclusions

To support low-latency sharing misses on unordered interconnects, this paper introduced Token Coherence. Token Coherence resolves protocol races without order by decoupling coherence into a correctness substrate and a performance protocol. The correctness substrate guarantees correct transfer and access to blocks by tracking tokens, and it provides liveness and starvation avoidance using persistent requests. Free from the burden of correctness, the performance protocol directly requests and receives blocks without concern for races. We introduced *TokenB*, a specific performance protocol based on broadcasting transient requests and reissuing requests when occasional races occur. *TokenB* can outperform snooping by using low-latency, unordered interconnects, and it can outperform directory protocols by avoiding sharing miss indirections, using only a moderate amount of additional bandwidth.

By decoupling performance and correctness, Token Coherence may be an appealing framework for attacking other multiprocessor performance problems. Specific performance protocol can reduce request bandwidth, reduce miss frequency via predictive push, and gracefully handle hierarchical systems. By using the substrate to ensure correctness, these optimizations can be implemented with little impact on system complexity.

Acknowledgments

We thank Virtutech AB, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Alaa Alameldeen, Allan Baum, Joel Emer, Kourosh Gharachorloo, Anders Landin, Alvin Lebeck, Carl Mauer, Kevin Moore, Shubu Mukherjee, Amir Roth, Dan Sorin, Craig Zilles and the Wisconsin Computer Architecture Affiliates for their comments on this work.

This work is supported in part by the National Science Foundation (EIA-9971256, EIA-0205286, CDA-9623632, 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 sabbatical), and donations from Compaq Computer Corporation, Intel Corporation, IBM, and Sun Microsystems.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, Feb. 1997.

[2] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture. In *Proceedings of SC2002*, Nov. 2002.

[3] M. E. Acacio, J. González, J. M. García, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 155–164, Sept. 2002.

[4] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[5] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron Shared Memory MP Systems. In *Proceedings of the 14th HotChips Symposium*, Aug. 2002.

[6] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[7] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2003.

[8] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. Scalability Port: A Coherent Interface for Shared Memory Multiprocessors. In *Proceedings of the 10th Hot Interconnects Symposium*, pages 65–70, Aug. 2002.

[9] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[10] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

[11] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. EXA Cache/Scalability Controllers. In *IBM Enterprise X-Architecture Technology: Reaching the Summit*, pages 37–50. International Business Machines, 2002.

[12] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[13] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[14] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[15] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann, revised edition, 2003.

[16] K. Farkas, Z. Vranesic, and M. Stumm. Scalable Cache Consistency for Hierarchically Structured Multiprocessors. *The Journal of Supercomputing*, 8(4), 1995.

[17] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-access Times. *Electronics*, 57(1):164–169, Jan. 1984.

[18] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ECC-Based Directory Implementations for Scalable Multiprocessors. In *Proceedings of the 12th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2000)*, Oct. 2000.

[19] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, Aug. 1991.

[20] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, Oct. 1998.

[21] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. *ACM Transactions on Computer Systems*, 11(4):300–318, Nov. 1993.

[22] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-Speed Electrical Signaling: Overview and Limitations. *IEEE Micro*, 18(1), January/February 1998.

[23] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, July 1995.

[24] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[25] A. Landin, E. Hagersten, and S. Haridi. Race-Free Interconnection Networks and Multiprocessor Consistency. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[26] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[27] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[28] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[29] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[30] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, Nov. 2000.

[31] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, Feb. 2002.

[32] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, June 2002.

[33] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

[34] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. In *Proceedings of the 9th Hot Interconnects Symposium*, Aug. 2001.

[35] A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. High-Throughput Coherence Controllers. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[36] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, volume I, Aug. 1995.

[37] D. Poulsen and P.-C. Yew. Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 296–280, Aug. 1994.

[38] I. Pragaspathy and B. Falsafi. Address Partitioning in DSM Clusters with Parallel Coherence Controllers. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2000.

[39] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[40] E. Rosti, E. Smirni, T. Wagner, A. Apon, and L. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1993.

[41] X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 135–144, June 1998.

[42] P. Stenström. A Cache Consistency Protocol for Multiprocessors with Multistage Networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 407–415, May 1989.

[43] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[44] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[45] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. IBM Server Group Whitepaper, Oct. 2001.

[46] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.