Appears in the proceedings of the
39th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-39), 2006

Coherence Ordering for Ring-based Chip Multiprocessors

Michael R. Marty and Mark D. Hill
Computer Sciences Department
University of Wisconsin—Madison
{mikem, markhill}@cs.wisc.edu

Abstract

Ring interconnects may be an attractive solution for future chip multiprocessors because they can enable faster links than buses and simpler switches than arbitrary switched interconnects. Moreover, a ring naturally orders requests sufficiently to enable directory-less coherence, but not in the total order that buses provide for snooping coherence. Existing cache coherence protocols for rings either establish a (total) ordering point (ORDERING-POINT) or use a greedy order (GREEDY-ORDER) with unbounded retries.

In this work, we propose a new class of ring protocols, RING-ORDER, in which requests complete in ring position order to achieve two benefits. First, RING-ORDER improves performance relative to ORDERING-POINT by activating requests immediately instead of waiting for them to reach the ordering point. Second, it improves performance stability relative to GREEDY-ORDER by not using retries.

Thus, the new RING-ORDER combines the best of ORDERING-POINT (good performance stability) with the best of GREEDY-ORDER (good average performance).

1. Introduction

Chip-Multiprocessors (CMPs) present microarchitects with new challenges and opportunities. In particular, because the primary means of communication between the individual processors is via shared-memory, the memory system is now a first-order design issue at the chip level instead of the system level. A key function of the memory system is to keep caches coherent.

Cache coherence has been well-studied in the context of building larger systems from uniprocessors. Small to moderate-sized symmetric multiprocessors [11, 43] used ordered bus interconnects for coherence with a snooping protocol. Larger multiprocessors [30, 40] required more complex interconnection networks to offer better scalability, such as the use of packet-switching in general topologies like a grid or torus. Although recent advances now allow a broadcast-based protocol to function correctly on an unordered network [25, 36], the default solution for scalable systems remains directory protocols even though they have costly indirections and state requirements.

Existing cache coherence techniques developed for previous multiprocessors can be applied to current CMPs [7, 26]. However, the technology of future CMPs and the realities of increasing design complexity [8] may require new solutions. On-chip networks can offer very high bandwidth because of the abundant availability of wires. However, the relative delay of these wires is rapidly increasing [21].

Implementing a logical bus in a CMP will likely entail a pipelined design with centralized arbitration and ordering points. To initiate a request on the CMP bus fabric described by Kumar et al. [28], a processor must first access a centralized arbiter and then send its request to a queue to create the total order. Ordered requests are resent on different snoop links, and snoop results collect at another queue and are again resent. Thus implementing bus ordering in a future CMP may incur significant latency and area costs. Like a logical bus, a crossbar is also a centralized structure that faces similar performance challenges in many-core CMPs.

Microarchitects can move to a packet-switched network using a general topology, like a grid or torus, to create a scalable on-chip interconnect. However this “route packets, not wires” approach [13] also comes with significant costs. First, implementing the interconnect itself requires the correct design and verification of all queues, routers, and algorithms for routing with deadlock avoidance. Second, more state and area overhead may be devoted to buffers and routers. Third, a directory coherence protocol designed to operate with an unordered network is usually needed for most topologies, requiring costly indirections and acknowledgement messages.

Ring interconnects offer a viable intermediate solution, because they use short point-to-point wires with distributed control, have trivial routers with less area and design overhead, and offer some ordering properties exploitable by the coherence protocol. Given the centralized nature of buses and crossbars, and the complexity and overhead associated with a packet-switched network, a ring-based interconnect may offer a preferable compromise. Rings are already used for intra-CMP coherence in the IBM Power4 [48] and Power5 [44], the on-chip interconnect for the IBM/Sony/Toshiba Cell [23], and are being considered by Intel for next-generation CMPs consisting of 8 to 16 cores [24]. Larger ring-based systems can be built by using a hierarchy, such as clustering or even a ring-of-rings (KSR-1 [9, 17]). The Scalable Coherence Interconnect (SCI) [20] sent messages on rings, but the coherence protocol did not exploit ring ordering properties.

The ordering of a ring is *not* the same as a totally ordered bus. Figure 1 illustrates how processors may see a different order of message arrivals depending on ring position. Cache coherence protocols *for rings* must create their own ordering of requests. One straightforward approach establishes an *ordering point* on the ring in which a total ordering of requests is created. The downside is that request messages are not active until they reach the ordering point, costing both latency and bandwidth. A second approach, used by

This work is supported in part by the National Science Foundation (NSF), with grants EIA/CNS-0205286, CCR-0324878, and CNS-0551401, as well as donations from Intel and Sun Microsystems. Hill has significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Intel, or Sun Microsystems.

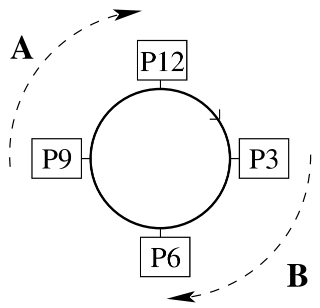


Figure 1: For unidirectional rings that allow simultaneous transmitters, the order of received messages may depend on ring position. Here, P12 receives messages in {A,B} order whereas P6 sees them in {B,A} order.

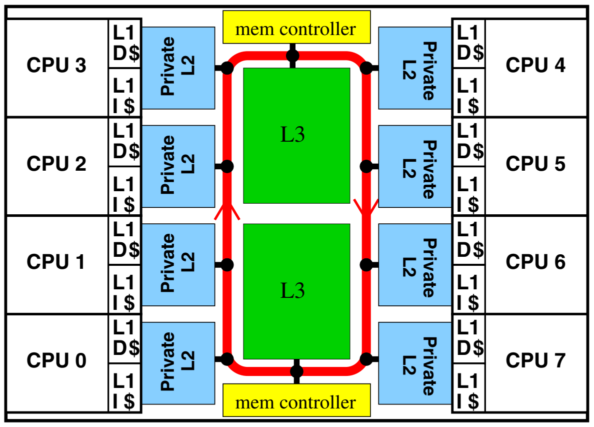


Figure 2: Baseline CMP

other systems including the IBM Power4/5 [29, 44, 48], greedily orders requests by making them active immediately and completing the first request that reaches the current owner. Unfortunately, the current greedy protocols resort to unbounded retries to handle conflicts and races [5, 12, 29].

Deploying a ring in a modern CMP requires aggressive implementation techniques. In particular, to minimize latency a ring-based system should immediately forward request messages to the next node (i.e., eager forwarding [47]) instead of first performing a snoop and then forwarding the message. Furthermore, a ring lacks shared lines to inhibit a memory response, yet a system should access the slow DRAM as soon as possible when necessary. Our baseline, eight-processor CMP with a ring interconnect is shown in Figure 2. In this CMP, the ring attaches to the processors' private cache hierarchies, shared L3 cache banks, and on-chip memory controllers.

An ideal ring protocol should (a) minimize latency and use of bandwidth, and (b) provide stable and predictable performance (e.g., no use of retries or negative acknowledgements). We develop such a protocol in this paper and summarize our contributions as follows:

- We present ring-based protocols based on an ordering point, ORDERING-POINT (Section 2.1), and a greedy-based approach, GREEDY-ORDER (Section 2.2). We create a new class of ring protocols that completes requests in order of ring position, RING-ORDER (Section 2.3), without incurring the overhead of an ordering point and without retries. Furthermore, RING-ORDER has no requirements on the timing of snoop responses or the ring.
- We apply ORDERING-POINT, GREEDY-ORDER, and RING-ORDER to the eight-processor CMP in Figure 2, using a memory interface cache (MIC) to improve memory latency and bandwidth characteristics (Section 3).
- We evaluate the three classes of protocols, using full-system simulation, with workloads including OLTP, Apache, SpecJBB, Zeus, and SpecOMP benchmarks (Sections 4 and 5). We find RING-ORDER performs up to 52% faster than ORDERING-POINT, and that it can outperform GREEDY-ORDER by up to 13%. RING-ORDER also offers significant improvements in performance stability over GREEDY-ORDER. Hence our new protocol combines the best of ORDERING-POINT (good performance stability) with the best of GREEDY-ORDER (good average performance).

2. Ring-based Cache Coherence Protocols

In this section, we discuss three different classes of coherence protocols for a ring and concentrate on *how requests are ordered*. For readers interested in more detailed specifications, table-based state-transition tables [45] are available in the Appendix for GREEDY-ORDER and RING-ORDER. We simplify the development of the protocols by first assuming a ring comprised only of minimal nodes with a processor and cache. In Section 3, we apply the protocols to a more-realistic CMP with integrated memory controllers, and discuss more requirements of the ring architecture. While our simulations assume sequential consistency, all coherence protocols in this paper can also support more relaxed memory consistency models.

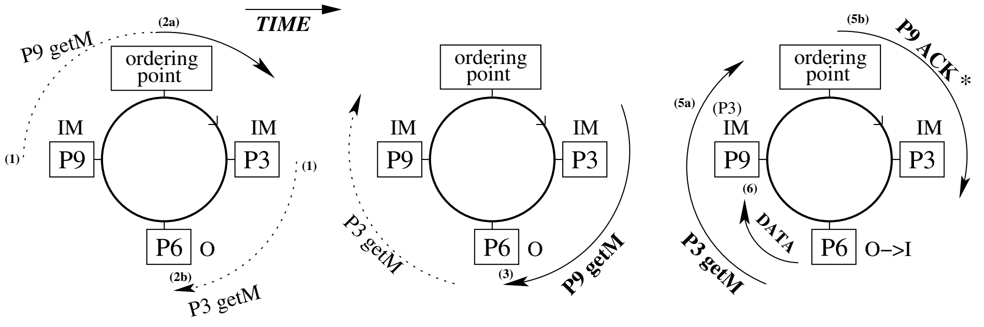
For all protocols, we require a unidirectional ring in which messages for the same block address cannot bypass one another on any ring link (i.e., parallel rings interleaved by address are allowed). When a request message is received at each ring interface, it is copied to the node's snoop queue and eagerly forwarded on the ring, minimizing latency by *not* forcing the *serialization of snoops*. The interface may also remove the request message from the ring, depending on the protocol. For data messages, the interface examines each to determine if it should be pulled from the ring or forwarded to the next node.

2.1 ORDERING-POINT

The first class of protocols recreates the global ordering of an atomic bus by establishing a point on the ring where all requests are ordered. The disadvantage of this approach is that requests are not active until they reach the ordering point, thus increasing both latency and bandwidth.

Ordering points are commonly used to deal with unordered interconnects, ranging from the directory-less AMD Opteron [1, 25, 50] to directory-based systems [7, 18, 30, 31, 20]. Because our ring-based ORDERING-POINT protocol does not store a list of sharers or a pointer (e.g., SCI) in some directory structure, it is logically most similar to the Opteron protocol. However unlike these other approaches, ORDERING-POINT exploits the order of a ring so that requests are pipelined and do not block.

Figure 3 shows an example of how ORDERING-POINT works. A processor's request message is initially inactive and ignored by other nodes until it reaches the ordering point. The ordering point



Glossary: getM = get modified, O = owned, I = invalid, IM = issued request for modify
***final ack message can be omitted with additional assumptions discussed in Section 2.1*

The example depicts two exclusive requestors in the ORDERING-POINT protocol (some actions not labeled in figure):

(1) P9 and P3 issue get modified requests to a block owned by P6.

(2a) P9's request reaches the ordering point and is made active. The active request will invalidate caches and locate the owner.

(2b) P3's inactive request is ignored by P6 and P9.

(3) P6 receives P9's active request, performs a snoop, sends data to P9.

(4) P9 forwards its own active request to potentially invalidate other processors. P9 sets a bit indicating its own active request is received.

(5a) P3's request reaches P9 (already seen own request). P9 commits to service it upon completion.

(5b) The ordering point removes P9's active request, sends final ack message indicating all caches are invalidated.

(6) P9 receives data from P6.

(7) P9 receives final ack message and sends data to P3.

Figure 3: Example of ORDERING-POINT

activates the request, thereby creating a consistent order of *active* request messages seen by other processors. The owning processor will eventually complete a snoop and send data to the requesting processor. The ordering point removes the request message from the ring and, for an exclusive request, sends a final acknowledgement message to the requestor indicating that all snoops (invalidations) have completed. The final acknowledgement message can be elided with some additional complexity and constraints, such as ensuring that all requests are fully buffered, but we found this has little impact on performance because it overlaps with data access and transfer.

To prevent the ordering point from blocking subsequent requests before each has completed, a requestor must record the first active request message received after observing and forwarding its own active request. By doing so, it commits to satisfy one subsequent request, thereby forming a linked chain of coherence service. If any of the subsequent requests are for exclusive access (get modified), the requestor will also invalidate itself upon completing its own request and servicing the next.

On average, a request must traverse half the ring (N/2 hops) to reach the ordering point, then traverse the entire ring (N) while active for a total of N+N/2 hops. Assuming a final acknowledgement message is used, the total control traffic is 2N. Although the protocol creates a total order of requests with a bounded latency, strictly ordering requests at the ordering point imposes additional latency.

2.2 GREEDY-ORDER

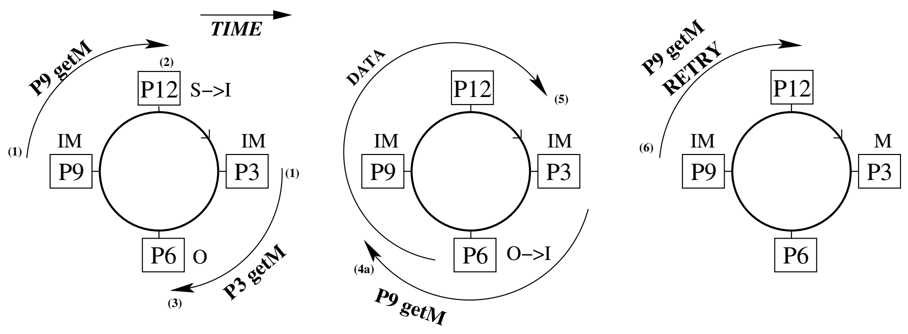
In a greedily ordered protocol, requests are immediately active and ordered by which request reaches the current owner first. In the common case, this improves latency and reduces bandwidth because a request does not incur extra hops to reach an ordering point. However, when conflicts (races) occur, a node may be required to issue an unbounded number of retries. GREEDY-ORDER is derived from Barroso et al.'s Express Ring [5] protocol and the IBM Power4/5 protocols [29, 44, 48].

Figure 4 illustrates GREEDY-ORDER with an example. A processor's request message is active immediately and acknowledged by the owning node in a combined response that follows the request (not illustrated). If multiple requests issue near-simultaneously, the first request that reaches the owner is acknowledged and wins the race. Otherwise, the request is not acknowledged and the losing requestor issues a retry after inspecting the combined response.

The example in Figure 4 shows a conflict situation with multiple exclusive requestors. If an exclusive request reaches any node with a shared request outstanding, we adopt Barroso's policy in which the shared requestor must abort the request and issue a retry, even though a data response may already be in flight. The shared request must be aborted because the owner may respond to an exclusive request while data travels to a shared request resulting in a coherence violation. An alternate approach prevents this case of possible incoherence by transferring ownership on any shared request. However, we found this policy resulted in more pathological starvation because of the increased likelihood of a shared request missing the in-flight owner. GREEDY-ORDER's cache controller is specified in the Appendix (Table 6), with shaded cells to indicate the state-transitions resulting in a retry.

A system that uses retries to handle contention avoids starvation only if future system conditions eventually allow a processor's retry to succeed in all cases. Probabilistic systems are acceptable in other domains of computing, such as Ethernet [39]. But feedback from industry indicates chip designers prefer stronger, non-probabilistic guarantees of liveness for a coherence protocol. Furthermore, a system like Ethernet exploits the carrier sense property to prove its liveness [42] whereas we are not aware of a general proof, for a greedily ordered protocol, which will always avoid a pathological retry scenario.

We considered other techniques to address retries in GREEDY-ORDER. Exponential backoff or adding randomness to retries can increase the probability of success, but does not guarantee freedom of starvation. Attaching a priority (such as the age of the request)



Glossary: getM = get modified, M = modified, O = owned, S = shared, I = invalid, IM = issued request for modify

The example depicts two exclusive requestors in the GREEDY-ORDER protocol (some actions not labeled in figure):

(1) P9 and P3 issue get modified requests to a block owned by P6.

(2) P12 snoops P9's request and invalidates its shared copy.

(3) P3 snoops P6's request and acknowledges it in a combined response. P3 commits to send data to P6.

(4a) P9's request passes P3 and P6 without being acknowledged.

(4b) P3 removes its request from ring. In the response following, P3 recognizes its request was acknowledged, expects data.

(5) P3 receives data from P6, completes request.

(6) P9 removes its request from ring and issues a retry because it was not acknowledged in the combined response.

Figure 4: Example of GREEDY-ORDER

does not solve the problem because it would require the processor to either remember starving requests, or to wait until multiple requests are received in order to prioritize the set of requestors. We also considered an approach that carefully constructs a distributed linked chain of requests such that a node hands off the block to the next requestor, like done in ORDERING-POINT. But correctly constructing this list without an ordering point adds significant complexity and constraints, especially when considering the effects of bank contention and interfacing with memory (discussed in Section 3).

Another disadvantage of GREEDY-ORDER is that it relies on a snoop response from every cache for every request. Implementing this efficiently on a ring (i.e., without an entire trailing response message) can use a combined response with synchrony such that response *bits* trail a request message by a fixed number of cycles. This fixed timing increases the complexity and constraints of the system. For example, the architected fixed timing must account for bank contention and the various snoop times of different-sized caches. If a request cannot be snooped within the architected fixed time, it must be negatively acknowledged (Nacked) and subsequently retried by the requestor. Increasing the delay in the fixed timing decreases the probability of a Nack, however this will negatively impact many non-delayed requests. Making the timing too aggressive will result in extra Nacks and retries, increasing the probability of pathological starvation.

We seek a better mechanism that bounds every node's coherence operation for performance stability, but does not use an ordering point. Furthermore, we seek a coherence protocol that does not rely on a synchronous snoop response for message efficiency. We now present a new class of ring protocols that orders completion of requests by the position on the ring.

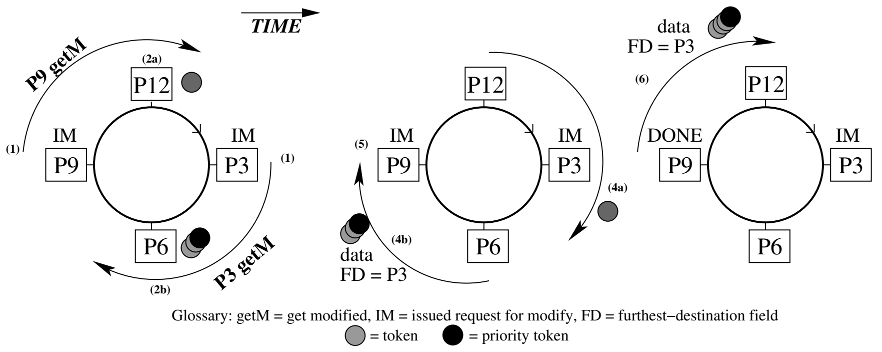
2.3 RING-ORDER

Ideally, a request in a ring protocol is active immediately, does not require retries to handle contention, and incurs minimal latency and bandwidth. We develop a new class of protocols that achieves these goals by *completing requests in ring order*. This protocol is inspired by token coherence [36], directly enforcing the coherence invariant by counting tokens. But, we do not use retries or persistent requests [37] to ensure forward progress. Instead, we exploit ring order to guarantee that initial requests always succeed. Another key advantage is that RING-ORDER does not require any fixed synchrony in the ring or when snoop responses generate.

Recall that token coherence associates a fixed number of tokens for each memory block in the system. In order to write a block, a processor must acquire all the tokens. To read a block, only a single token is needed. In this way, the coherence invariant is directly enforced by counting and exchanging tokens. Cache tags and messages encode the number of tokens using Log_2 N bits, where N is the fixed number of tokens for each block. Unlike token coherence as previously published, in RING-ORDER memory stores only a *single bit* per memory block to track whether it contains all or none of the tokens (discussed in Section 3.2). We distinguish one of the tokens as the *priority token*. Similar to the owner token used by token coherence, the priority token denotes which response should carry data. More importantly, it allows requests to complete in ring order by prioritizing the requestors as it moves around the ring.

An example of RING-ORDER is shown in Figure 5, and the cache controller is specified in Table 7 of the Appendix. The key insight is that token counting allows a requestor to remove tokens off the ring to complete its request safely and potentially immediately. A request message causes tokens to move on the unidirectional ring. But a response message is not strictly sent to a particular requestor and can instead be used by other requestors on the way. Each response message includes a `furthest-destination` field to indicate the furthest relative node on the ring that desires the tokens for a coherence request. A requestor also tracks this field in its miss status holding register (MSHR) [27] so that it may hold the tokens temporarily to complete its request, but can determine if it needs to (eventually) put tokens back on the ring.

To ensure starvation avoidance, a policy must be in place to prevent multiple exclusive requestors from holding a subset of the tokens. The priority token breaks the symmetry by distinguishing which requestor should hold tokens. A requesting node must remove the incoming priority token from the ring and hold onto it



The example depicts two exclusive requestors in the RING-ORDER protocol (some actions not labeled in figure):

(1) P3 and P9 issue get modified requests to a block. P12 holds a single token, P6 holds the rest including the priority token.

(2a) P12 receives P9's request, initiates snoop.

(2b) P6 receives P3's request, initiates snoop.

(3) P6 receives P9's request while snooping, records furthest relative requestor in its snoop-tracking table.

(4a) P12 completes snoop and sends single token on ring. P3 does not remove the single token because it does not hold the priority token.

(4b) P6 completes snoop and sends data and all tokens, including the priority token, on the ring. The response is tagged with a furthest-destination field set to P3.

(5) P9 removes data and tokens from ring and is able to complete its request because it acquires all tokens.

(6) P9 honors the furthest-destination field and places data and tokens back on the ring.

Figure 5: Example of RING-ORDER

until its request completes. Other non-priority tokens, in flight due to a writeback or exclusive request, must coalesce with the priority token. Thus a requestor does not acquire non-priority tokens until it holds the priority token. If an exclusive requestor is holding the priority token, it updates the furthest-destination field in its MSHR when it receives other requests while waiting for tokens. The furthest destination field also includes a single bit to indicate if any requestor seeks all the tokens.

RING-ORDER minimizes data transfer, because *all requesting nodes complete their requests as data moves around the ring once*. One suspected negative aspect of our protocol is that a writer may need to collect tokens from multiple sharers, with a message for each. We could further optimize this by using a combined response that collects tokens. We choose not to because our observations corroborate other studies showing most invalidations are for few caches (most commonly one) [19, 35]. We do implement a simple optimization that allows a requestor to remove and hold non-priority tokens before receiving the in-flight priority token, but only if there are no other concurrent requests. To detect other concurrent requests, a bit is set in the MSHR on observing another request message, and the furthest destination of each incoming response message is examined.

RING-ORDER applies to rings with relaxed timing, or even asynchronous circuit designs [49] because it avoids a synchronous combined snoop response. The same property will also be beneficial when applied to hierarchical systems (e.g., a *ring-of-rings*) because a system-wide snoop response is unnecessary.

3. Application to CMPs

Here, we apply our three classes of ring protocols to a CMP (Figure 2) and describe the interaction with memory controllers and ring architectures.

3.1 Base System

The baseline CMP consists of eight processor cores, each with private L1 and L2 caches. Two shared L3 cache banks are each backed by an on-chip memory controller and are interleaved by the low-order bits of the block address. The L3 acts like a victim cache in that allocations only occur on L2 writebacks. The L2 and L3 cache controllers connect to a single unidirectional ring to handle all on-chip coherence and memory requests. The L2 controllers issue all requests on behalf of the processor and handle snoops. The L3 controllers also participate on the ring and tightly couple with the on-chip memory controllers via direct links. For the ORDERING-POINT protocol, the L3/memory controllers function as the ordering point. Therefore, the memory latency is the same for all three protocols.

3.2 Accessing Memory

A key challenge for non-directory coherence protocols is determining if memory should respond to a request with data. Our protocols add an owner-bit to each memory block [16] so the memory controller can recognize ownership and supply data. Because memory is slow and bandwidth constrained [15, 22], we cache these bits in a memory interface cache (MIC) located at each of the two memory controllers. The MIC is coarse-grained to reduce tag overhead and exploits spatial locality by associating several bits with each tag.

An alternative approach is to collect a snoop response from every cache on every request (analogous to logical shared lines in bus-based systems), and only then obtain the data from memory. However, this can negatively impact memory latency due to the additional ring traversal.

For all three classes of protocols, the owner bit is accessed to determine if memory should respond to a request with data. On a MIC miss, the owner bits are fetched and the data block is prefetched in case the owner bit is ultimately set. GREEDY-ORDER must also Nack the request, causing a retry, because the timing of the synchronous snoop response cannot be met when accessing the slow DRAM. Designers of a GREEDY-ORDER protocol also need to prevent a possible pathological scenario in which the recently fetched owner bit is continually replaced before the retry reaches the MIC.

For RING-ORDER, the owner bit represents whether memory logically contains all the tokens, or none of the tokens. Thus unlike

Table 1: Memory System Parameters
(processor cycles specified)

Private L1 Caches	Split I&D, 64 KB 4-way set associative, 2-cycle access time, 64-byte line
Private L2 Caches	Unified 1MB 4-way set associative, 15-cycle data access, 8-cycle tag access, 64-byte line
Shared L3 Caches	Two 8MB shared banks, 16-way set associative, 25-cycle access time, 64-byte line
On-chip Ring Interconnect	80-byte unidirectional links, 6-cycle delay per link, 2-cycle switch delay
Memory	4GB of DRAM, 275-cycle access
Memory Interface Cache	128KB, 16-way set associative, 256 bits per tag

Table 2: Out-of-Order Core Parameters

Reorder buffer/scheduler	128/64 entries
Pipeline width	3-wide fetch & issue
Pipeline stages	15
Direct branch predictor	1kBytes YAGS
Indirect branch predictor	64 entry (cascaded)
Return address stack	64 entry

token coherence as previously published, we further innovate by reducing the token count at memory down to a single bit per block by coalescing tokens on replacements. In the common case of replacing unshared data, tokens are already coalesced causing the memory interface to immediately accept the writeback and set the owner bit. Otherwise, a cache places tokens on the ring to coalesce with other tokens. To replace the priority token and prevent possible livelock (because the sharers could simultaneously replace), the cache with the priority token must first send a control message to find a different cache willing to accept its tokens.

Alternative ordering strategies present challenges for implementing the exclusive cache state (E-state). GREEDY-ORDER achieves the E-state with an added bit to the combined response that indicates if any sharer exists (logically similar to a shared intervention signal). RING-ORDER has the equivalent of an exclusive state because any response from memory logically contains all the tokens, and clean data is omitted from token replacement messages. In ORDERING-POINT, however, an exclusive state is difficult without other significant compromises. Even if an additional exclusive bit is added to each memory block (and MIC), the memory controller cannot determine when to reset the exclusive bit without tracking a count of sharers. Hence our ORDERING-POINT protocol lacks an E-state.

The equivalent of a perfect memory interface cache is to implement a structure that summarizes all on-chip caches. We did not evaluate this approach because our coarse-grained MIC performs quite well, and avoids the very wide aggregate associativity and significant area overhead of this structure.

3.3 Ring Architecture and Buffering

Although the focus of this paper is coherence ordering, we now discuss some of the implications and requirements of the low-level ring architecture. The choice of ring architecture (commonly slotted or register-insertion) [46] is mostly independent of our three classes of ring protocols. A slotted ring is most practical for GREEDY-ORDER because acknowledgement bits can trail the request slot by a fixed number of cycles. Therefore, implementing the combined snoop response does not require a separate message. However a register-insertion ring inserts arbitrary delays by allowing a processor to immediately transmit while buffering some amount of incoming data. Since ORDERING-POINT and RING-ORDER do not require a combined snoop response on every request, a register-insertion ring is just as practical.

All three protocols require the low-level ring architecture to provide guaranteed resources (virtual networks) for request and responses. In a slotted ring, this is accomplished by using a fixed allocation of request and response slots. Register-insertion rings can use flow control and priority techniques similar to the SCI ring (i.e., idle symbols, go bits) [20]. Like most existing systems we are aware of, we assume that the ring provides fault tolerance at a lower level than the coherence protocol.

In all protocols, once a response message is placed on the ring, it can always be removed by the destination because the resources allocate upon issuing the request message. In RING-ORDER, before forwarding a token message, the interface checks the MSHR table to determine if tokens for the address should be removed from the ring. Accessing the MSHR table should have minimal impact on the latency of token messages because it is typically small.

In contrast to response messages, a request message may reach a node with insufficient resources to handle the snoop. This is especially problematic in GREEDY-ORDER because of its synchronous snoop response requirement. But for ORDERING-POINT and RING-ORDER, most contention can be handled by using deeper buffers because no synchronous snoop response is needed. In the rare case that a request reaches a processor with no available buffer space, we assume the ring interface will flip some bits in the header message to cause the message to traverse the ring again (ignored by subsequent ring nodes), essentially using the ring itself as a buffer. Thus there is never any back-pressure on the ring. In this way, the ring is analogous to a “traffic roundabout”. In a traffic roundabout, once a car is able to enter the roundabout (ring), it continues to circulate until it is able to exit (acquire a buffer).

4. Evaluation Methodology

We evaluate our protocols with full-system simulation using Virtutech Simics [33] extended with the Wisconsin GEMS toolset [34]. GEMS provides a detailed memory system timing model which accounts for all protocol messages and state transitions. Each processing core has private 64KB L1 I&D caches and a private 1MB L2 cache. The two shared L3 caches are 8MB each for a total on-chip L2/L3 capacity of 24MB. Each L2 cache is generously interleaved into sixteen banks to increase the amount of snoop bandwidth. We idealize the L3 caches such that a snoop can always be performed. The two memory interface caches (MIC) are both 128KB and each tag entry holds 256 owner bits. All other memory system parameters are specified in Table 1. For RING-ORDER, the logical number of tokens for each block is 16 to allow all caches to hold a shared copy of data. Thus response messages and cache tags encode the token count with 4 bits, plus an additional bit to denote the priority token.

To better understand and isolate memory system performance and its impact on overall performance, we model both in-order and out-of-order Sparc cores attached to the same memory system described above. The in-order cores are inspired by Niagara [26], but are 2-way superscalar, single-threaded, and do not share a floating-point unit. ALU operations are idealized and always take one cycle. The out-of-order cores use GEMS’ TFsim timing model

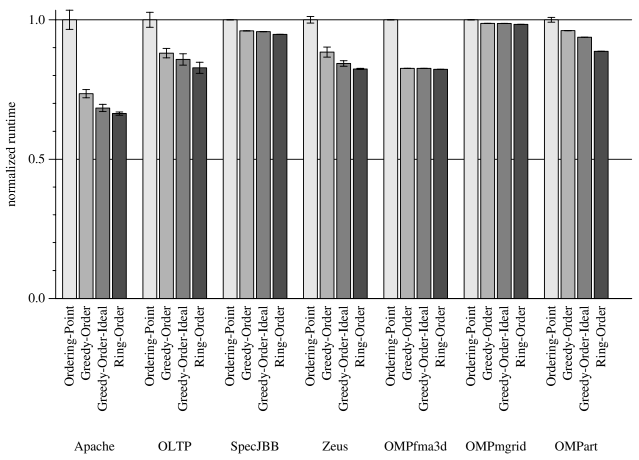


Figure 6: Normalized runtime with in-order cores

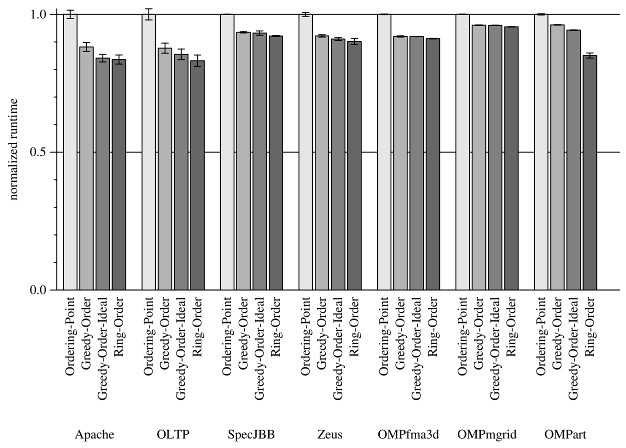


Figure 7: Normalized runtime with out-of-order cores

[38] with the parameters specified in Table 2. We model core frequencies of 12-FO4 delays, consistent with prior work [10]. All configurations implement sequential consistency.

We approximate a well-engineered slotted ring with wide, 80-byte unidirectional links. We do so by using GEMS' packet-switched interconnect infrastructure with a ring topology and sufficient buffering at the endpoints. Links are wide enough to interleave a single control message with a data message, but we prevent routing a control and data message for the same block address in the same cycle to prevent reordering. Our technology assumptions model a link delay of 300 picoseconds per millimeter, and each of the ring links in the target CMP measure 5 mm. We clock the ring at half the core frequency, consistent with the IBM Cell [23]. Thus the modeled delay per rink link is six processor cycles (assuming 4 GHz cores) plus two cycles for the switch, making the total round-trip latency 80 processor cycles.

In GREEDY-ORDER, synchronous snoop responses follow requests by 25 cycles (6.25 ns). With our modeled L2 tag latency of 8 cycles, this allows the L2 to process three tag accesses within the architected time. This was an engineering tradeoff based on empirical evaluation. With the L2 interleaved sixteen ways, we found that an aggressively timed snoop response, unable to handle bank conflicts, resulted in occasional starvation due to pathological behavior.

Nonetheless, system designers may go to great lengths in order to engineer GREEDY-ORDER's snoop response to be aggressive, yet avoid excessive Nacks due to bank contention. Therefore, we also simulate GREEDY-ORDER-IDEAL in which all snoop responses magically generate immediately even if the bank is busy with other queued requests.

For RING-ORDER and ORDERING-POINT, each of the interleaved L2 banks is backed by a snoop queue sized to eight entries (exceeding the maximum size we observe in simulation).

The workloads used are the following: an online transaction processing workload (OLTP), static web-serving workloads (Apache and Zeus), a Java middleware workload (SpecJBB), and scientific workloads from the SpecOMP suite [4] (OMPfma3d, OMPart, OMPmgrid). For the commercial workloads, we measured transactions completed. The SpecOMP workloads were split by main loop completion because of the prohibitive simulation time required to finish the entire execution using the reference input set. Alameldeen et al. [3] further describe all workload configurations including the lengths of cache warmup and simulation. To account for non-determinism in multithreaded workloads, we pseudo-randomly perturb simulations and calculate error bars to 95% confidence [2].

5. Evaluation

5.1 Runtime

Figure 6 shows the normalized runtime for all four protocols running with the in-order cores. Runtime is normalized to ORDERING-POINT. We find:

- RING-ORDER performs the best as it is 6-52% faster than ORDERING-POINT for all workloads except OMPmgrid, which performs comparable because nearly all misses are to read-only data satisfied by memory.
- RING-ORDER outperforms GREEDY-ORDER by 8-12% for Apache, OLTP, Zeus and OMPart. Compared to GREEDY-ORDER-IDEAL, RING-ORDER performs comparable or slightly better.

Figure 7 shows the normalized runtime with out-of-order cores. RING-ORDER still performs the best as it is 5-21% faster than ORDERING-POINT (for all workloads) and 6-13% faster than GREEDY-ORDER for Apache, OLTP, and OMPart. The quantitative performance differences between in-order and out-of-order processors change for two primary reasons: out-of-order cores tolerate some of the coherence latency, and protocol-independent stalls contribute more to the execution time (e.g., pipeline flushes).

To gain further insight into the performance differences, Table 3 shows L2 misses-per-1000 instructions, the percentage of sharing misses, and the average latency of sharing misses. The protocols exhibit similar L2 misses-per-1000-instructions for most workloads. But for OMPfma3d and OMPmgrid, ORDERING-POINT incurs twice as many L2 misses due to its lack of an exclusive cache state. This noticeably penalizes OMPfma3d because of its high miss rate. All other performance differences are mostly due to sharing behavior. For example, 48.6% of Apache's L2 misses are sharing and the protocols behave differently for these misses. For Apache, a sharing read miss averages 128.8 cycles with ORDERING-POINT, 106.8 cycles with GREEDY-ORDER, and 96.4

Table 3: Breakdown of L2 Misses

	L2 misses / 1000 instructions			% sharing L2 misses (Load, Store)	average cycles of L2 sharing misses (load, store)			
	ORDERING-POINT	GREEDY-ORDER and GREEDY-ORDER-IDEAL	RING-ORDER		ORDERING-POINT	GREEDY-ORDER	GREEDY-ORDER-IDEAL	RING-ORDER
Apache	28.7	26.4	26.0	33.1, 15.5	128.8, 153.6	106.8, 101.6	96.8, 93.1	96.4, 90.4
OLTP	13.1	12.5	12.6	47.6, 24.2	131.3, 154.8	106.2, 102.7	96.2, 93.9	96.1, 91.4
SpecJBB	3.9	3.0	3.0	21.6, 2.5	133.4, 153.7	108.3, 104.3	98.8, 95.3	97.3, 93.7
Zeus	19.8	18.7	18.0	29.4, 15.1	133.4, 153.8	108.2, 103.4	97.3, 94.6	96.9, 90.3
OMPmgrid	3.6	1.5	1.5	3.9, 0.9	156.9, 154.7	142.4, 119.3	136.9, 112.6	124.3, 94.4
OMPfma3d	22.6	12.6	12.6	0.3, 0.9	174.9, 158.0	157.0, 125.7	148.9, 117.4	139.9, 94.0
OMPart	62.5	62.5	62.5	57.8, 0.01	128.8, 161.3	117.9, 128.1	111.7, 128.1	103.5, 94.9

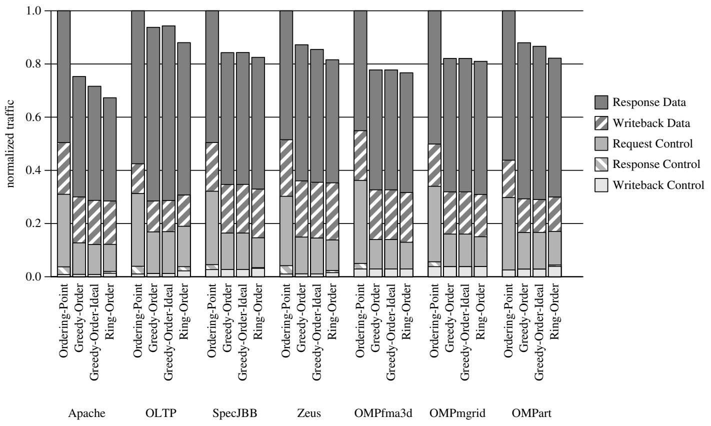


Figure 8: Normalized ring traffic (in-order cores)

cycles with RING-ORDER. These average sharing miss latencies match the expected behavior of the protocols. ORDERING-POINT must traverse half the ring, on average, to activate a request and GREEDY-ORDER incurs a penalty for the timing of the synchronous snoop response. Likewise the OLTP, Zeus, and OMPart workloads exhibit significant sharing misses and see similar latencies. OMPmgrid and OMPfma3d show higher sharing miss latencies because of barrier contention, but these do not impact performance much because the contention is infrequent.

All of our protocols work with different configurations of unidirectional rings, including multiple rings interleaved by address. We did not simulate this plethora of options, however, we did perform sensitivity analysis to the latency and width of ring links. If the ring runs at the same frequency as the in-order processor cores, the control overhead penalty of ORDERING-POINT's cache-to-cache transfers diminishes. In this case, RING-ORDER still performs 4-30% faster for all workloads except OMPmgrid. A narrower ring also diminishes control overhead because more cycles are spent on actual data transfer rather than the smaller control messages. When simulating 40-byte links, RING-ORDER outperforms ORDERING-POINT by 5-36% (except OMPmgrid). Furthermore, narrower links negatively impact GREEDY-ORDER and GREEDY-ORDER-IDEAL because increased data traversal latency can actually exacerbate the retry problem as there is more opportunity for a request message to miss the in-flight data. With 40-byte links, the increase in retries affects performance, especially for Apache, where GREEDY-ORDER-IDEAL incurs 30% more race-induced retries and causes RING-ORDER to outperform it by 11%. For all the configurations we simulated, RING-ORDER performed the best.

5.2 Bandwidth

Figure 8 shows normalized bytes transferred on the CMP ring. Reducing bandwidth can lead to lower latency (if constrained), less resources devoted to the interconnect, and reduced power consumption. We assume control messages use 8 bytes and data messages use 72 bytes. For GREEDY-ORDER and GREEDY-ORDER-IDEAL, bandwidth used by the combined snoop response is ignored. We find:

- RING-ORDER consumes the least amount of ring bandwidth for all workloads. It uses 15-34% less bandwidth than ORDERING-POINT, and 2-12% less bandwidth than GREEDY-ORDER.

ORDERING-POINT's activation and acknowledgment of requests, and GREEDY-ORDER's retries consume more bandwidth than RING-ORDER's non-data token responses (Response Control), and RING-ORDER's coalescing and non-silent replacement of tokens (Writeback Control). RING-ORDER further optimizes bandwidth by completing all simultaneous outstanding requests, for the same block, with a single data message traversal.

5.3 Performance Stability

We now consider performance stability by examining the worst-case behavior observed in simulation. For GREEDY-ORDER, the worst-case is pathological starvation that can theoretically occur due to its lack of total ordering. We find:

- Starvation situations did occasionally arise in our simulations for both GREEDY-ORDER and GREEDY-ORDER-IDEAL.

Figure 9 shows an excerpt from a trace of GREEDY-ORDER-IDEAL running the OMPmgrid workload. Processor 6 continually issued a retry for the block because, due to the timing conditions encountered, its request continually missed the owner in flight to a

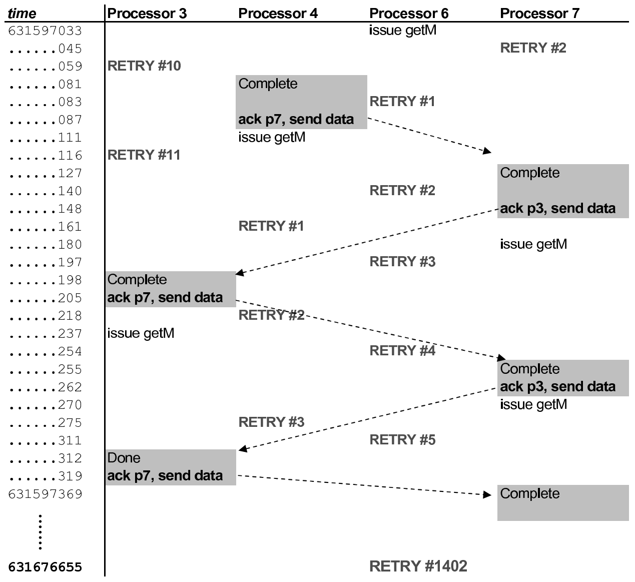


Figure 9: Excerpt of a GREEDY-ORDER-IDEAL trace, running OMPmgrid, for a single cache block. Processor 6's request is pathological starved for over 75,000 cycles.

Table 4: Overall Miss Latencies in Cycles (MAX, AVG)

	ORDERING-POINT	GREEDY-ORDER-IDEAL	GREEDY-ORDER	RING-ORDER
Apache	490, 86.9	750, 73.6	**5206**, 73.6	**308**, 70.7
OLTP	516, 55.4	833, 44.4	765, 46.3	297, 43.6
SpecJBB	476, 66.6	800, 55.1	**8316**, 55.4	**316**, 53.8
Zeus	524, 83.4	1516, 72.6	**6611**, 72.6	**336**, 67.7
OMPmgrid	517, 92.2	**large**[1], 51.9	**14791**, 51.9	**422**, 50.0
OMPfma3d	589, 222.6	2945, 152.2	**large**[1], 152.2	**302**, 150.3
OMPart	517, 118.5	851, 110.4	973, 112.9	422, 104.3

Table 5: Retries per Miss Request (MAX, AVG)

	ORDERING-POINT	GREEDY-ORDER-IDEAL	GREEDY-ORDER	RING-ORDER
Apache	0,0	10, 0.24	**53**, 0.24	0,0
OLTP	0,0	8, 0.12	11, 0.12	0,0
SpecJBB	0,0	11, 0.37	92, 0.37	0,0
Zeus	0,0	14, 0.31	68, 0.31	0,0
OMPmgrid	0,0	**large**[1], 0.07	**259**, 0.07	0,0
OMPfma3d	0,0	29, 0.12	**large**[1], 0.12	0,0
OMPart	0,0	10, 0.11	13, 0.12	0,0

[1]At least one simulation run encountered a request that exceeded our per-request watchdog timer of 80,000 cycles.

different requestor. We could further engineer GREEDY-ORDER and GREEDY-ORDER-IDEAL to complete all our simulations by reducing the chances of starvation through techniques previously discussed. However, we would not be convinced that our efforts would result in starvation-free execution for months and years on a real system, given that our simulation target runs for only a few seconds.

We now examine the performance stability of our protocols by considering the maximum latencies and retries encountered for all misses (not just sharing misses). Table 4 shows the average and maximum latency of any L1 miss. RING-ORDER has the lowest maximum observed request latency of 422 cycles. Some requests in GREEDY-ORDER and GREEDY-ORDER-IDEAL take *thousands of cycles* and even exceed the per-request watchdog timer of 80,000 cycles we use in the simulator. Table 5 shows the average number of retries used for each coherence request and the maximum observed. Misses to the MIC account for most of the actual number of retries (not shown) even though the hit rate of each 128KB MIC is 89-91% for all workloads. However for those requests that required several retries, the reasons were mostly due to bank conflict and coherence races. Generally SpecOMP workloads, with their use of a barrier for fine-grained loop synchronization, encounter the most severe situations requiring numerous retries due to coherence races.

6. Related Work

Barroso et al. [5, 6] developed a snooping protocol for SMP systems using a slotted ring, which served as the basis of our greedily ordered protocol. They compared their snooping implementation against a directory-based ring protocol and a split-transaction bus, finding the snooping-on-rings approach preferable. We build upon the work of Barroso et al. by extending and applying the protocol to a CMP, classifying snooping ring protocols based on ordering, and comparing it to a new type of ring protocol as well as one using a directory-less ordering point.

IBM's Power4 [48] and Power5 [44] both use a protocol similar to GREEDY-ORDER [29]. One difference between GREEDY-ORDER and the IBM protocols is that memory does not contain owner bits and does not participate in the combined response. Instead, the requestor will resend the combined response on the ring. If no other cache acknowledged the request in the combined response, the memory controller will send the data (which it prefetches when observing the initial request). If the combined response indicates a coherence conflict, the processor instead issues a retry. To explicitly detect a conflict, whenever a processor acknowledges a request and sends data, it remembers the address in a table until cleared by the combined response that the winning requestor resends. In contrast, our GREEDY-ORDER protocol uses owner bits and a memory interface cache to reduce memory latency and bandwidth in a CMP. In doing so, memory participates in the combined response such that resending it, and explicitly detecting coherence conflict with an extra table, is unnecessary.

Strauss et al. [47] present flexible snooping for optimizing performance and power in a system using an embedded ring *between* bus-based CMPs. The protocol is similar to the IBM Power5 and GREEDY-ORDER except that they selectively and predictively change when request messages are forwarded to the next CMP. All of our protocols *immediately* forward a request message on the ring before performing the snoop (eager forwarding) for maximum performance because snoops parallelize. In contrast, Strauss et al. selectively and predictively do the opposite by first performing a snoop and then forwarding the request to the next node on the ring (lazy forwarding), thereby potentially saving power. One consequence of this approach in a greedy ordered protocol is that a separate response message trails the request message whenever eagerly forwarded. The GREEDY-ORDER protocol we model instead uses bandwidth-efficient response *bits* that follow the request by a fixed number of cycles. Our work focuses on the ordering of requests on a ring, eliminating the retries used by Strauss et al., and targets a CMP system.

Nonetheless, their forwarding strategy applies especially well to RING-ORDER because our protocol does not need an expensive combined response message for every eagerly forwarded request.

The IBM Cell processor [23] uses a ring-based interconnect for transferring data between the main processor and the eight "synergistic processing elements" (SPEs). A tree-based centralized arbiter determines when the processors access the ring to transfer data. We do not assume centralized arbitration for accessing the ring, although if one were present, it could be used for coherence ordering. Since each individual SPE has its own private memory with separate addressing, the Cell interconnect is optimized for DMA-like operations rather than cache coherence at the line level.

The Scalable Coherence Interface (SCI) [20] is based on a register-insertion ring and used a distributed directory-based protocol. Several systems were built with the SCI including the Sequent STiNG system [32]. The SCI protocol does *not* exploit the ordering properties of a ring and requires many messages to manipulate the doubly linked list of sharers. For example, obtaining a shared copy of data and updating the list of sharers requires four ring traversals. In contrast, for all of our protocols, getting a shared copy requires only a single request message and a single response message.

The Kendall Square Research KSR-1 [9, 17] used a hierarchy of slotted unidirectional rings for cache coherence. In the KSR, a request is always lazily forwarded—a message visits a node, performs the snoop operation, and only then is forwarded to the next node. Our protocols parallelize the snoops by eagerly forwarding requests immediately to the next node before performing the snoop. We fail to find the specific strategy the KSR used for handling coherence races. We suspect that by not performing snoops in parallel, it is able to construct a linked chain of requests because the searches are slow and can carry information about previous snoops.

Chung et al. [12] also proposed a snooping protocol for SMP systems based on register-insertion rings. It too is greedily ordered and uses retries to handle conflict. Oi et al. [41] developed a cache coherence protocol that operates on bidirectional rings for SMPs. Because bidirectional rings have even less order than unidirectional rings, they use both retries and an ordering point. The Hector SMP system used a hierarchy of rings and a write-update protocol with filters [14]. We only considered write-invalidate protocols in this work.

Marty et al. [37] also apply token coherence in a CMP system. However they used unordered interconnection networks, retries, and persistent requests. The focus of this work is the study of ring protocols for CMPs rather than token coherence. We leverage token coherence to create a better ring protocol, but simplify it by exploiting the ordering of a ring to avoid retries and persistent requests. Furthermore, we innovate by requiring only a single bit per memory block rather than an entire token count.

7. Conclusions

Rings offer an interconnect with short point-to-point links, distributed control, and ordering properties exploitable by a coherence protocol. However, a ring does not provide the same ordering as a logical bus. In this paper, we classify snooping ring protocols based on how coherence requests are ordered. The ORDERING-POINT protocol establishes an ordering point to recreate the total order provided by a bus, but it is inefficient with latency and bandwidth. The GREEDY-ORDER protocol offers lower latency, but an unbounded number of retries are used to resolve conflicts. Our new type of ring protocol, RING-ORDER, offers the best of ORDERING-POINT (good performance stability) and GREEDY-ORDER (good average performance). Furthermore, RING-ORDER does not require a synchronous snoop response, potentially easing design complexity and verification, and potentially improving performance with its relaxed timing.

Acknowledgements

We thank Steve Kunkel, Jim Rose, Ching-Tsun Chou, Xiaowei Shen, David Wood, the anonymous reviewers, and the Wisconsin Computer Architecture Affiliates for feedback on this work. We also thank Dan Gibson, Natalie Enright, Jason Cantin, Luke Yen, Kevin Moore, and UW Multifacet students for comments and proofreading. Finally we thank Virtutech, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their support.

References

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

[2] 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*, pages 7–18, Feb. 2003.

[3] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, Jul/Aug 2006.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[5] L. A. Barroso and M. Dubois. Cache Coherence on a Slotted Ring. In *Proceedings of the International Conference on Parallel Processing*, pages 230–237, Aug. 1991.

[6] L. A. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-based Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 268–277, May 1993.

[7] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[8] C. Bazeghi, F. J. Mesa-Martinez, B. Greskamp, J. Torrellas, and J. Renau. uComplexity: Estimating Processor Design Effort. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[9] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR 1 computer system. *Technical Report KSR-TR-9202001, Kendall Square Research*, 1992.

[10] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

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

[12] S. W. Chung, S. T. Jhang, and C. S. Jhon. PANDA: ring-based multiprocessor system using new snooping protocol. In *International Conference on Parallel and Distributed Systems*, pages 10–17, 1998.

[13] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Design Automation Conference*, pages 684–689, 2001.

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

[15] I. T. R. for Semiconductors. ITRS 2002 Update. Semiconductor Industry Association, 2002. http://public.itrs.net/Files/2002Update/2002Update.pdf.

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

[17] S. J. Frank, H. Burkhardt, L. O. Lee, N. Goodman, B. I. Margulies, and F. D. Weber. Multiprocessor Digital Data Processing System, Oct. 1991. U.S. Patent 5,055,999.

[18] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, Nov. 2000.

[19] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[20] D. Gustavson. The Scalable Coherent Interface and related standards projects. *IEEE Micro*, 12(1):10–22, Feb. 1992.

[21] M. Horowitz, R. Ho, and K. Mai. The future of wires. In *Semiconductor Research Corporation Workshop on Interconnects for Systems on a Chip*, May 1999.

[22] J. Huh, S. W. Keckler, and D. Burger. Exploring the Design Space of Future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, 2001.

[23] Unleashing the Cell Broadband Engine Processor. http://www-128.ibm.com/developerworks/power/library/pa-fpfeib/, Nov. 2005.

[24] Platform 2015: Intel Processor and Platform Evolution for the Next Decade. ftp://download.intel.com/technology/computing/archinnov/platform2015/download/platform_2015.pdf, June 2005.

[25] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March-April 2003.

[26] P. Kongetira. A 32-way Multithreaded SPARC Processor. In *Proceedings of the 16th HotChips Symposium*, Aug. 2004.

[27] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Symposium on Computer Architecture, Computer Architecture News*, volume 9, pages 81–87, May 1981.

[28] R. Kumar, V. Zyuban, and D. Tullsen. Interconnections in multi-core architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[29] S. Kunkel. IBM Future Processor Performance, Server Group. Personal Communication, 2006.

[30] 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.

[31] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[32] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, May 1996.

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

[34] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[35] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[36] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[37] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, Feb. 2005.

[38] 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*, pages 108–116, June 2002.

[39] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(5):395–404, July 1976.

[40] 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.

[41] H. Oi and N. Ranganathan. A Cache Coherence Protocol for the Bidirectional Ring Based Multiprocessor. In *International Conference on Parallel and Distributed Computing and Systems*, 1999.

[42] D. Shasha, A. Pnueli, and W. Ewald. Temporal Verification of Carrier-Sense Local Area Network Protocols. In *Proceedings of The 11th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 54–65, Jan. 1984.

[43] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yaun, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvery, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of the 4th Hot Interconnects Symposium*, pages 41–52, Aug. 1996.

[44] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. Power5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4), 2005.

[45] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[46] W. Stallings. Local Networks. *ACM Computing Surveys*, 16(1), 1984.

[47] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

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

[49] T. Villiger, H. Kaslin, F. K. Gurkaynak, S. Oetiker, and W. Fichter. Self-timed Ring for Globally-Asynchronous and Locally-Synchronous Systems. In *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems*, pages 141–151, May 2003.

[50] F. Weber. AMD's Next Generation Microprocessor Architecture, Oct. 2001.

Appendix

The following page shows detailed specifications of the GREEDY-ORDER and RING-ORDER cache controllers using a table-based technique [45]. We believe this representation provides clear, concise visual information yet includes sufficient detail (e.g., transient states) arguably lacking in the traditional, graphical form of state diagrams.

The rows of each table correspond to the *states* that the cache controller can enter, including both stable states (e.g. M, O, E, S, I) and transient states (e.g., IS, IM). The columns correspond to *events* that cause the cache to take actions and to potentially change the state. Events are usually the result of receiving a message from the interconnect or processor. The table entries themselves are the atomic *actions* taken and, if the state changes, the resulting state (denoted with a slash, e.g., /S indicates the new state is S).

Table 6 shows the specification of the GREEDY-ORDER cache controller, with stable states `{M, O, E, S, I}`. Consider an example: when a processor issues a `Load` to the cache controller in State I, it sends a `GETS` message and transitions to state `IS`. The cache controller in state M, O, or E will acknowledge the `GETS`, send data, and transition to state O. Various `Own GET` events indicate the result of the combined response that follows the request. For example, the `{Own GETS (acked, shared)}` event indicates that the request was acknowledged and that there also exists a sharer such that the requestor cannot enter the exclusive state when clean data is received. The shaded cells indicate *retries* which can occur an unbounded number of times.

Table 7 shows the specification of the RING-ORDER cache controller. The stable states, `{NONE, SOME,` $\texttt{SOME}^{\texttt{P}}$`, ALL}` represent the number of tokens held. $\texttt{SOME}^{\texttt{P}}$ also indicates if the cache holds a subset of tokens along with the priority token. $\texttt{IM}^{\texttt{P}}$ and $\texttt{IM}^{\texttt{SOME}}$ indicates an outstanding exclusive request where the requestor holds the priority token or some tokens, respectively. The `P` and `COA` states handle token coalescing during replacements of shared data. The `{FurthestDest GETS}` and `{Furthest-Dest GETM}` events locally generate when the processor finishes a request but needs to send tokens/data on the ring, as described in Section 2.3. We show an explicit *forward* action because token response messages can be handled by any requestor with an MSHR allocated, instead of being delivered to a particular processor.

Table 6. GREEDY-ORDER Cache Controller State Transitions
(shaded cells indicate retries)

	Load	Store	Replacement	Other GETM	Other GETS	Own GETM (acked)	Own GETM (unacked)	Own GETM (nacked)	Own GETS (acked)	Own GETS (acked, shared)	Own GETS (unacked)	Data (retry mismatch)	Data
I	send GETS / IS	send GETM / IM	replace / I	/ I	x	e	e	e	e	e	e	e	e
S	do Load	send GETM / IM	replace / I	/ I	ACK GETS (shared)	e	e	e	e	e	e	e	e
E	do Load	do Store / M	replace / I	ACK GETM, send data / I	ACK GETS, send data / O	e	e	e	e	e	e	e	e
O	do Load	send GETM / OM	send data, replace / I	ACK GETM, send data / I	ACK GETS, send data	e	e	e	e	e	e	e	e
M	do Load	do Store	send data, replace / I	ACK GETM, send data / I	ACK GETS, send data / O	e	e	e	e	e	e	e	e
IS	z	z	z	/ ISD	x	e	e	e	/ ISA^E	/ ISA	send GETS	discard data	e
ISA^E	z	z	z	x	x	e	e	e	e	e	e	discard data	save data, do Load / E
ISA	z	z	z	/ ISD	x	e	e	e	e	e	e	discard data	save data, do Load / S
ISD	z	z	z	x	x	e	e	e	send GETS / IS	send GETS / IS	send GETS / IS	discard data	discard data
IM	z	z	z	x	x	/ IMA	send GETM	send GETM	e	e	e	e	e
IMA	z	z	z	x	x	e	e	e	e	e	e	e	save data, do Store, / M
OM	z	z	z	x	x	e	do store / M	send GETM	e	e	e	e	e

Table 7: RING-ORDER Cache Controller State Transitions

	Load	Store	Replacement	Other GETM	Other GETS	Own GET	Other PUT	PUT-ACK	Tokens	Tokens (all)	Priority Token w/ Data	Priority Token w/ Data (all tokens)	FurthestDest (GETM)	FurthestDest (GETS)
NONE	send GETS / IS	send GETM / IM	replace	x	x	x	forward	forward	forward	forward	forward	forward	e	e
SOME	do Load	send GETM / IM^{SOME}	send Tokens, replace / NONE	Send Tokens / NONE	x	x	remove PUT, send PUT-ACK / P	forward	forward	forward	forward	forward	e	e
SOME^P	do Load	send GETM / IM^P	send PUT / COA	Send P-Data / NONE	Send P-Data / SOME	x	e	e	Remove Tokens (writeback)	Remove Tokens (writeback) / ALL	e	e	Send P-Data / NONE	Send P-Data / SOME
ALL	do Load	do Store, mark dirty	send P-Data[1], replace / NONE	Send P-Data / NONE	Send P-Data / SOME	x	e	e	e	e	e	e	Send P-Data / NONE	Send P-Data / SOME
IS	z	z	z	x	x	x	remove PUT, send PUT-ACK	forward	forward	e	Remove P-Data, do Load / SOME^P	Remove P-Data, do Load / ALL	e	e
IM	z	z	z	x	x	x	remove PUT, send PUT-ACK	forward	forward[2]	e	Remove P-Data / IM^P	Remove P-Data, do Store / ALL	e	e
IM^P	z	z	z	Update FD	Update FD	x	e	e	Remove Tokens	Remove Tokens, do Store / ALL	e	Remove P-Data, do Store / ALL	e	e
IM^{SOME}	z	z	z	Send Tokens / IM	x	x	remove PUT, send PUT-ACK	forward	forward[2]	e	Remove P-Data / IM^P	Remove P-Data, do Store / ALL	e	e
P	do Load	z	z	Update FD	Update FD	x	e	e	Remove Tokens (writeback)	e	Remove P-Data / SOME^P	Remove P-Data / ALL	e	e
COA	z	z	z	x	x	x	e	Send P-Data[1], replace / NONE	Remove Tokens (writeback)	Remove Tokens, send P-Data[1], replace / NONE	e	e	e	e

Key: z = stall, x = don't care, e = error, FD = Furthest Destination field, P-Data = Priority token with data

1. Data can be optionally omitted if replacing clean data
2. Tokens can be optionally removed before receiving priority token if only requestor