

# Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors\*

Karin Strauss

Xiaowei Shen<sup>†</sup>

Josep Torrellas

University of Illinois at Urbana-Champaign  
<http://iacoma.cs.uiuc.edu>

<sup>†</sup>IBM Research  
[xwshen@us.ibm.com](mailto:xwshen@us.ibm.com)

## Abstract

Snoopy cache coherence can be implemented in any physical network topology by embedding a logical unidirectional ring in the network. Control messages are forwarded using the ring, while other messages can use any path. While the resulting coherence protocols are inexpensive to implement, they enable many ways of overlapping multiple transactions that access the same line — making it hard to reason about correctness. Moreover, snoop requests are required to traverse the ring, therefore lengthening coherence transaction latencies.

In this paper, we address these problems and make two main contributions. First, we introduce the *Ordering* invariant, which ensures the correct serialization of colliding transactions in embedded-ring protocols. Second, based on this invariant, we remove the requirement that snoop requests traverse the ring. Instead, they are delivered using any network path, as long as snoop responses — which are typically off the critical path — use the logical ring. This approach substantially reduces coherence transaction latency. We call the resulting protocol *Uncorq*.

We show that, on a 64-node Chip Multiprocessor (CMP), *Uncorq* improves the performance, on average, by 23% for SPLASH-2 applications and by 10% for commercial applications. With an additional simple prefetching optimization, the performance improvement is, on average, 26% for SPLASH-2 applications and 18% for commercial applications.

## 1. Introduction

As systems continue to integrate, it is becoming feasible to build medium-scale shared-memory multiprocessors with 32-128 processor cores at remarkably low cost. In such systems, a major challenge for designers is to implement a cache coherence solution that carefully balances performance, complexity, and cost.

Two traditional approaches to cache coherence are snoopy and directory-based schemes. Snoopy schemes that rely on one or more broadcast buses cannot scale beyond a small number of cores without significant increases in cost. On the other hand, directory schemes, while scalable, have the disadvantage of adding one level of indirection to coherence transactions, increasing latency. Moreover, directories can be expensive and complex to design.

A cost-effective approach for these machines is to support snoopy cache coherence on a point-to-point network rather than on

a broadcast bus. This approach is popularly referred to as *network-based* snoopy cache coherence. While this approach is not as scalable as directory schemes, it is inexpensive and may represent the best design approach for medium machine sizes. This general approach is used by IBM Power systems [15].

Interestingly, while the broadcast bus in bus-based snoopy schemes ensures that coherence messages are delivered in the same order to all the nodes, this is not the case in network-based snoopy schemes. Indeed, messages from two different concurrent transactions to the same address can be received by different nodes in different orders. This lack of ordering makes the design and verification of efficient network-based snoopy schemes challenging.

To address this lack of ordering, we proposed embedding a logical unidirectional ring in the physical network of the machine [14]. Snoop requests and responses use the logical ring, while other messages can use any path through the network. This approach is attractive because it is simple — it places no constraints on network topology or timing. However, there are still many ways in which multiple transactions that access the same line can overlap.

In [14], we did not list the invariants that need to be enforced to ensure the correct ordering of transactions in these protocols. Moreover, all protocols that use rings — either physical rings (e.g., [1, 7]) or logical rings [14] — require that snoop requests traverse the ring, as they visit all nodes. This requirement limits the parallelism of snoop operations and lengthens the latency of coherence transactions.

To address these problems, this paper makes two main contributions. First, it presents the *Ordering* invariant, which ensures the correct serialization of overlapping transactions in embedded-ring protocols. Second, based on this invariant, it introduces a novel protocol that allows snoop requests to be delivered to multiple nodes in parallel, using any path in the network. Snoop responses still use the ring, but they are typically off the critical path. The new protocol is called *Unconstrained Snoop Request Delivery*, or *Uncorq*. *Uncorq* greatly reduces miss latency, while still preserving protocol simplicity.

Our results show that *Uncorq* is very effective. On a 64-node Chip Multiprocessor (CMP), *Uncorq* improves the performance, on average, by 23% for SPLASH-2 applications and by 10% for commercial applications. Moreover, with an additional simple prefetching optimization, the performance improvement is, on average, 26% for SPLASH-2 applications and 18% for commercial applications.

This paper is organized as follows. Section 2 presents some background; Section 3 presents our invariant; Section 4 introduces *Uncorq*; Section 5 describes implementation issues; Sections 6 and 7 present an evaluation; Section 8 discusses related work; and Section 9 concludes.

\*This work was supported in part by the National Science Foundation under grant CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Karin Strauss was supported by an IBM PhD Fellowship. Karin Strauss is now with Advanced Micro Devices (AMD). Her e-mail address is [karin.strauss@amd.com](mailto:karin.strauss@amd.com).

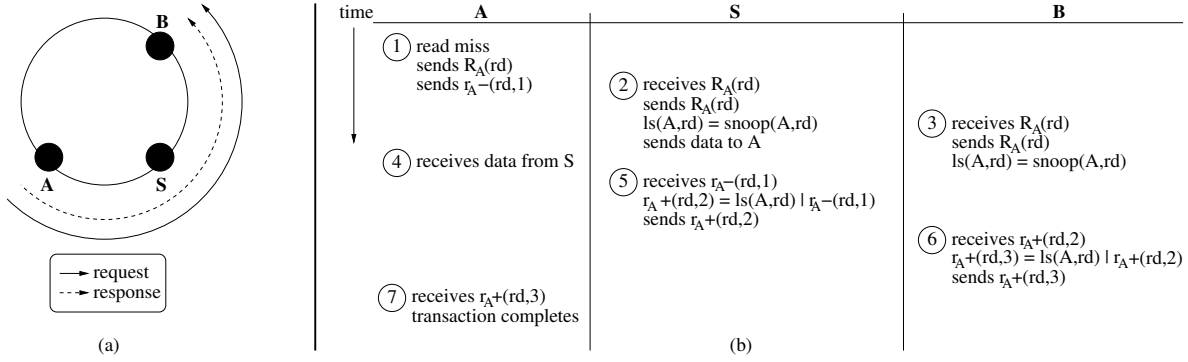


Figure 1. Read miss transaction in the Eager protocol.

## 2. Background

### 2.1. Basics of Embedded-Ring Protocols

In embedded-ring snoopy protocols, a unidirectional ring is locally embedded in whatever network topology a machine uses [14]. When a processor needs to initiate a coherence operation, it places a request on the ring. The request traverses the ring, visiting all nodes in order. In a naive implementation, as the request visits a node, it induces a cache snoop. If the request is a miss and the data is found, the node — which we call *Supplier* — sends it to the requester using the shortest path provided by the network topology. Irrespective of whether the data is found, after the snoop is completed, the request is forwarded to the next node in the ring. Unless the request is a read and the data was already found, the next node repeats the process. As soon as the data arrives at the requester, the latter can use it; as soon as the request message returns to the requester node, the coherence transaction is completed. For load balancing, messages to different line addresses can use different logical rings, or the same ring with different directions.

Since the supplier node may be far from the requester, the latency of a coherence transaction in this naive implementation may be long. To reduce the latency, the Eager Forwarding algorithm (*Eager*) is used. This algorithm breaks down the message in the transaction into a *request* ( $R$ ) and a *response* ( $r$ ) message. The requester node places both messages in the ring. The  $R$  races ahead, initiating snoops in all the nodes without waiting for their completion. As usual, as soon as the data is found in a cache, it is sent to the requester. The  $r$  travels behind, leaving a node only after the snoop has completed there. It also carries information as to whether the data has already been found in one of the visited nodes (positive combined response, or  $r+$ ) or not (negative combined response, or  $r-$ ). As usual, when  $r$  reaches the requester, the transaction is completed. While Eager was described in [14], the separation of  $R$  and  $r$  was already used in physical-ring protocols [1].

As an illustration, Figure 1 shows a read miss transaction under Eager. Part (a) shows the ring with nodes A, S (the supplier node) and B, and the ring traversal direction of both  $R$  and  $r$  messages. Part (b) shows a timeline of events in the three nodes. The notation used is as follows: requests are denoted by  $R_X(Y)$ , snoop operations by  $\text{snoop}(X,Y)$ , and local snoop outcomes by  $ls(X,Y)$ , where  $X$  is the requester node ID and  $Y$  is the type of request. A combined response is denoted by  $r_{X+}(Y,N)$  if it is positive or  $r_{X-}(Y,N)$  if it is so far negative, where  $N$  is the number of collected snoop outcomes so

far. Circled numbers are events referenced in the text — typically, but not necessarily, in chronological order.

When requester node A suffers a read miss (event 1), it places a read request  $R_A$  and a negative response  $r_{A-}$  on the ring. The latter indicates that no node has yet been found that can supply the requested data. Node S receives  $R_A$  (2), forwards it to B and then initiates a local snoop operation, which generates a local snoop outcome indicating whether S can supply the requested data. In our example, since S can supply the data, it generates a positive snoop outcome and immediately sends the data to A, using the shortest path. When B receives  $R_A$  from S (3), B forwards it and initiates a local snoop operation, which generates a negative snoop outcome. As soon as A receives the data from S (4), A is free to use it. When S receives  $r_{A-}$  from A (5), S combines it with its positive outcome, resulting in a positive combined response  $r_{A+}$ , that S then forwards to B. When B receives  $r_{A+}$  from S (6), it further combines it with its negative outcome into a new positive combined response  $r_{A+}$ , and sends it to A. When  $r_{A+}$  reaches A (7), the transaction completes.

A node can only generate and forward a new combined response when both (1) the local snoop operation has completed and generated an outcome and (2) the node has received a combined response from the previous node in the ring.

When the requester receives the data (or just the data ownership in case of a store to locally cached but unowned data), the load or store is bound — it cannot be undone. If a load originated the transaction, the load is now complete and can retire. If a store originated it, there may still be copies of the line in other nodes to be invalidated. Only when the requester receives the  $r$  message it is guaranteed that all copies have been invalidated. At this point, the store is complete. To simplify the coherence protocol implementation, in all cases, the requester only changes the state of the line in its cache to a stable state and is allowed to supply the line to other nodes when it receives the  $r$  message.

Compared to the naive implementation, Eager reduces transaction latency but increases the number of snoop operations and ring messages — increasing power consumption and, in some cases, network contention. These effects are mitigated with the *Flexible Snooping* family of snooping algorithms [14]. The idea is to augment each node with a predictor of how likely the requested line is to be found in the node in the appropriate state. Then, when a  $R$  is received at a node, depending on the predictor’s information, it can either (i) proceed as in Eager, (ii) initiate the local snoop while stalling  $R$  until  $r$  is received and the snoop is complete, or (iii) sim-

ply forward  $R$  without snooping. Each algorithm is a different point in the energy and performance space.

The type of transaction illustrated in the example, where a second node satisfies the transaction, is called a *cache-to-cache* transaction. If, instead, no other node is able to satisfy the transaction, the requester receives a  $r$ -. In this case, the requester must get the line from memory. When the memory responds with the line, the transaction completes. This is a *memory-to-cache* transaction.

In this paper, we focus primarily on optimizing cache-to-cache transactions. These include the misses that, traditionally, have been the most expensive and that are common in many applications [2, 6]. Such misses are likely to continue to be frequent in the future, as CMPs continue to increase the size of their on-chip caches, and workloads become more sharing-intensive to leverage the presence of multiple on-chip cores.

## 2.2. Protocol Used

In this paper, we use a coherence protocol with a *single supplier*. This means that the protocol is such that, given a request, at most one node in the machine will be in a state capable of supplying the line to the requester. This condition substantially simplifies supporting embedded-ring coherence.

Our protocol is an invalidation-based one similar to the one for IBM Power4 [15]. The details are presented in [14] although, in this paper, we evaluate a single-CMP machine. Therefore, the  $S_L$  state mentioned in [14] does not exist. The supplier cache can have the requested line in one of the following *Supplier* states. If the line has the same value as in memory, the supplier states are Exclusive (no other node has a copy) or Master Shared (other nodes may have a cached copy but this one is designated to provide the line). If the line does not have the same value as in memory, the supplier states are Dirty (no other cache has a valid copy of the line) or Tagged (other nodes may share this dirty line but this one is designated to provide the line and write it back on eviction).

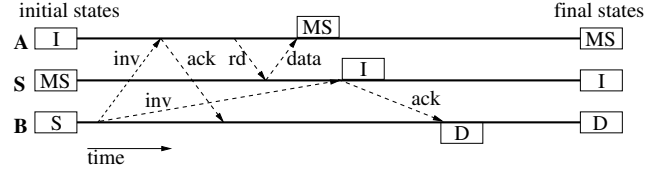
When a processor issues a write, the line ends up in its cache in a supplier state. When it issues a read miss, this is not always desirable. However, to simplify the description of our contributions, unless otherwise indicated, we will assume that read misses also lead to a supplier state.

## 2.3. Transaction Collision and Serialization

A *collision* occurs when two or more coherence transactions directed to the same line use the ring at the same time. Although collisions may be rare, the coherence protocol needs to handle them properly to ensure correctness. Specifically, given two colliding transactions, the coherence protocol has to ensure that (i) they get serialized — namely, that they appear as if one transaction completed before the other one started — and (ii) all nodes observe the same order of the transactions.

If such requirements are not guaranteed, different nodes observing two colliding transactions may transition into incompatible states, causing incoherence. As an example, consider Figure 2 and assume a generic network topology.

Initially, nodes  $B$  and  $S$  cache a line in state Shared (S) and Master Shared (MS), while node  $A$  does not have the line (I).  $B$  wants to write the line, so it broadcasts invalidations. An invalidation reaches  $A$ , which acknowledges it to  $B$ . Later,  $A$  wants to read the line and, because it misses in its cache, it sends a read request. Assume that



**Figure 2.** Two colliding transactions that lead to incompatible coherence states.

$A$ 's read request reaches  $S$  before  $B$ 's invalidation does.  $S$  supplies the data to  $A$  and  $A$  caches the line in state Master Shared (MS). Later,  $B$ 's invalidation reaches  $S$ , which invalidates its copy and acknowledges the invalidation to  $B$ . The latter, having received acknowledgments from  $A$  and  $S$ , transitions to state Dirty (D). The system is now incoherent:  $A$  caches the line in state Master Shared while  $B$  caches the same line in state Dirty. This problem is caused by the inability of the system to ensure that all nodes see the same transaction order:  $A$  sees the write transaction before the read one, while  $S$  sees them in the opposite order.

## 3. Serialization in Embedded-Ring Protocols

### 3.1. Invariant for Correct Serialization

Correctly serializing colliding transactions in an embedded-ring protocol is challenging because there is no central transaction arbitration point such as a broadcast bus or a directory. Consequently, we want to identify an invariant such that, if enforced, ensures the correct serialization of colliding transactions. Without loss of generality, our discussion assumes collisions of only two transactions.

Note that request ( $R$ ) and response ( $r$ ) messages have different roles.  $R$  messages determine the *order* of transactions: when two transactions collide, the first  $R$  that reaches the supplier node orders its transaction ahead of the other one. The supplier node sends a message to the initiator of that  $R$  with the supplier status and, if the transaction is a miss, with the data as well. In the rest of this paper, we call this message the *Suppliership* message.

On the other hand, the role of the  $r$  messages is to *communicate* the transaction order to the other nodes, so that they enforce correct serialization. Specifically, when the node that initiated one of the two colliding transactions sees the two  $r$  messages (one positive and one negative), it can enforce correct serialization. It can do so either by squashing (or marking as squashed) the loser transaction so that it is forced to retry, or by completing the winning transaction and then servicing the loser transaction.

Consequently, we propose the following *Ordering* invariant to guarantee the correct serialization of colliding transactions. The invariant relies on intuition and is proven by exhaustively testing all possible transaction combinations [13]. Section 3.3 gives an overview of the possible cases.

**Ordering Invariant:** Given two colliding transactions, consider the order in which their  $r$  messages (one positive and one negative) arrive at the first of the two requesting nodes found in ring order after the supplier node  $S$ . That order must be the same as the order in which their  $R$  messages arrived at  $S$ .

Knowing that the order of  $r$  messages reflects that of  $R$  ones at  $S$  gives this first requesting node enough information to enforce the correct serialization of the two transactions — either by squashing

the loser transaction or by completing the winner transaction and then servicing the loser one.

This invariant applies when there is a supplier node in the ring; we will see later the case when there is none.

### 3.2. Supporting the Ordering Invariant

Supporting the Ordering invariant in an embedded ring running the Eager or Flexible Snooping algorithms requires only three safeguards. First, both  $R$  and  $r$  messages should traverse the ring in the same direction. Second, a node that observes in the ring a request  $R_i$  to a certain line is not allowed to issue its own  $R$  to the same line until after it observes in the ring the corresponding response  $r_i$  to that request. We call this requirement the *In-Progress Transaction Restriction*. Limiting this type of transaction overlap is easy to enforce. Finally, messages to the same line should be transferred in FIFO mode in any ring link and should be processed in FIFO mode by any node. The only exception to this rule is that, to speed up transactions,  $R$  messages are allowed to overtake  $r$  messages from other transactions.

With these three safeguards, the ring order in which two requests  $R_i$  and  $R_j$  were initially placed in the ring determines the ring order in which the corresponding  $r_i$  and  $r_j$  will follow. This is sufficient to enforce the Ordering invariant. Indeed, if a supplier node receives requests  $R_i$  and  $R_j$  in this order, it will process them in this order, and then receive and forward responses  $r_i$  and  $r_j$  in the same order. Such response order will be preserved until the first of the two colliding requester nodes in ring order.

### 3.3. Protocol Operation

We now overview an implementation of Eager that relies on the Ordering invariant to correctly serialize colliding transactions.

#### 3.3.1. Types of Collisions

To understand the protocol, we classify collisions in two dimensions: (i) whether they induce *Natural* or *Forced Serialization*, and (ii) whether or not there is a supplier node in the ring for them. These two dimensions result in 4 cases, which are shown in Table 1. Unless otherwise noted, our description applies equally to collisions of two reads, a read and a write, or two writes.

A *Natural Serialization* collision occurs when one of the two requesting nodes (e.g.,  $A$ ) receives its own response  $r_A$  before it sees any messages from the other requesting node. In this case, both transactions usually complete without squashes. If there is a supplier in the ring, the response is positive ( $r_{A+}$ ), and when  $A$  receives it, the transaction completes. When  $A$  later sees  $R_B$ , it services  $B$ 's request. Note that there is the uncommon situation where, by the time  $A$  receives  $R_B$ ,  $A$  has received  $r_{A+}$  but not the suppliership. In this case,  $A$ 's transaction is not complete and a squash will be required. Specifically, when  $A$  receives  $R_B$ , it ignores it, and when later  $A$  receives  $r_{B-}$ , it marks it as squashed. When  $B$  sees that  $r_{B-}$  is marked as squashed, it retries its transaction.

Still under *Natural Serialization*, if there is no supplier node in the ring, node  $A$  receives a negative response ( $r_{A-}$ ). At this point, node  $A$  gets the data from memory. As before, if node  $A$  sees the other transaction's request ( $R_B$ ) after it obtains the memory value, it services the request. Otherwise,  $A$  ignores  $R_B$  and, when it sees the transaction's response ( $r_{B-}$ ), it marks it as squashed. When node  $B$  sees that  $r_{B-}$  is marked as squashed, it retries its transaction.

Supplier Node	Serializ.	Process
Present	Natural	$S$ receives $R_A$ $S$ sends suppliership to $A$ $A$ receives $r_{A+}$ $A$ receives $R_B$ $A$ sends suppliership to $B$
Present	Forced	$A$ receives $R_B, r_{B-}$ ( $A$ is in transient) $S$ receives $R_A$ $S$ sends suppliership to $A$ $S$ receives $R_B$ $B$ receives $r_{A+}$ and records own transaction is loser $A$ receives $r_{A+}$ and completes its transaction $B$ receives $r_{B-}$ and retries its transaction
Not Present	Natural	$A$ receives $r_{A-}$ $A$ requests data from memory $A$ receives $r_{B-}$ and marks $r_{B-}$ as squashed $B$ receives $r_{B-}$ and retries its transaction
Not Present	Forced	$A$ receives $R_B; B$ receives $R_A$ ( $A$ and $B$ are in transient) $A$ receives $r_{B-}; B$ receives $r_{A-}$ $A$ and $B$ separately run winner-selection algo. $A$ receives $r_{A-}; B$ receives $r_{B-}$ if winner, node gets data from memory if loser, node retries its transaction

**Table 1.** Handling collisions in Eager.

A *Forced Serialization* collision occurs when there is no *Natural Serialization* of the transactions. For Eager, this means that each of the two requesting nodes sees the other transaction's  $R$  before receiving its own  $r$ . Under *Forced Serialization*, one of the transactions will be squashed. If there is a supplier in the ring, one of the responses is positive and the other negative. In this case, the requesting node that is closest to the supplier in ring order sees the two responses and marks the  $r$ - one as squashed. When the originator of the squashed transaction sees its squashed  $r$ -, it retries the transaction.

Still under *Forced Serialization*, if there is no supplier node in the ring, each of the requesting nodes sees: (i) the other node's  $R$ , (ii) the other node's  $r$ -, and (iii) its own  $r$ -. Since all responses are negative and not marked as squashed, the nodes cannot decide which transaction should be retried based only on the messages observed. An algorithm is separately run in each requesting node to decide which node is the winner and, therefore, can access memory, and which one is the loser and, therefore, needs to retry.

#### 3.3.2. Overall Operation

We now consider each of the four cases in detail. Without loss of generality, we assume that the relative position of nodes in ring order is as in Figure 1(a):  $A$  (one requesting node), then  $S$  (supplier, if there is one), then  $B$  (the other requesting node). In our protocol, a node with an outstanding  $R$  on a line keeps the entry in its cache in a transient state. This means that it takes no action on reception of another  $R$  on the line. However, if this other  $R$  is for a write, the cache records that, if its own transaction loses, it has to invalidate the line from its cache.

**Supplier Present, Natural Serialization** Without loss of generality, we describe the case when  $B$  wins. In this case,  $r_B$  traverses the entire ring before  $B$  observes  $R_A$ .

Figure 3 shows an example where  $A$  issues a read miss and  $B$  an invalidation, as in Figure 2.  $B$  sends the invalidation request  $R_B$  followed by a negative response  $r_{B-}$  (1).  $A$  receives  $R_B$ , forwards it, and performs a local snoop operation that generates a negative outcome (2).  $A$  combines the negative outcome with  $r_{B-}$  and forwards it (3). After that,  $A$  suffers a read miss on the same line and

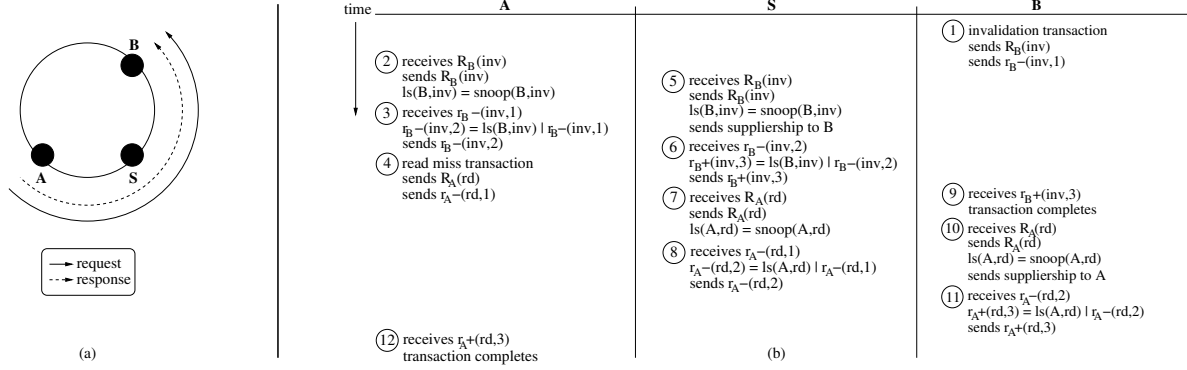


Figure 3. Example of Natural Serialization when the supplier is present in the ring.

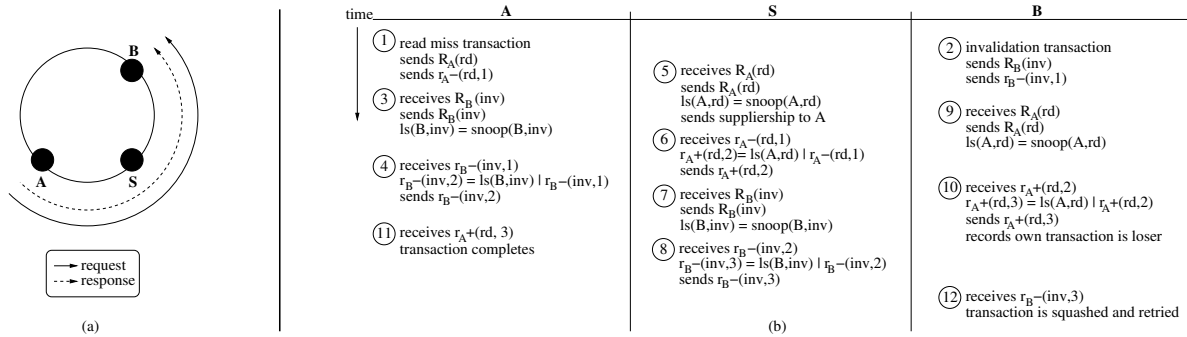


Figure 4. Example of Forced Serialization when the supplier is present in the ring.

sends a read request  $R_A$  and a  $r_{A-}$  (4). Our same-address FIFO policy guarantees that  $S$  observes and processes  $R_B$  first, sending the suppliership to  $B$  (5). Later,  $S$  receives and forwards the following three messages in order:  $r_{B-}$ , which is transformed into  $r_{B+}$  (6);  $R_A$  (7); and  $r_{A-}$ , which remains negative (8). Meanwhile,  $B$  receives the suppliership (not shown) and  $r_{B+}$  (9), which completes the transaction. Later,  $B$  receives  $R_A$  (10), which causes the suppliership to be sent to  $A$ , and  $r_{A-}$  (11), which is transformed into  $r_{A+}$ . Finally,  $A$  receives the suppliership (not shown),  $r_{A+}$  (12), and completes. The two transactions have correctly serialized.

Table 1 lists the main events in the dual case when  $A$  wins.

**Supplier Present, Forced Serialization** In this case,  $A$  and  $B$  observe each other's  $R$  before observing their own  $r$ . Due to the position of nodes on the ring,  $A$ 's transaction has to win. The case when  $R_B$  reaches  $S$  first, given the In-Progress Transaction Restriction (Section 3.2), is an example of Natural Serialization.

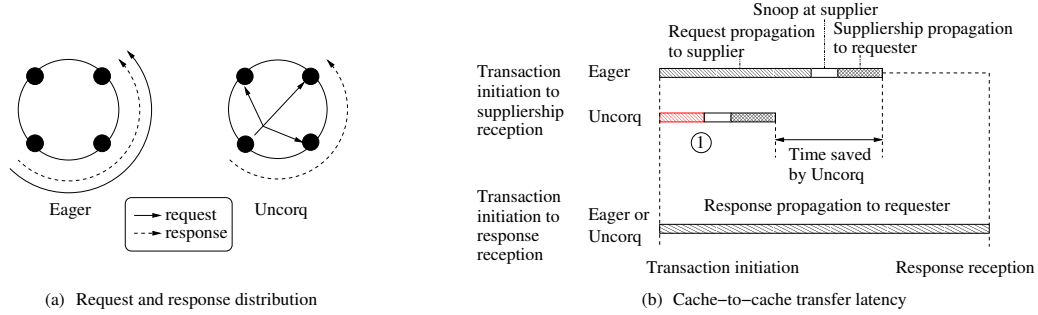
Figure 4 shows an example where  $A$  issues a read miss and  $B$  an invalidation, as in the previous case. Initially,  $A$  sends a read request  $R_A$  followed by a  $r_{A-}$  (1), while  $B$  sends an invalidation request  $R_B$  followed by a  $r_{B-}$  (2). When  $A$  receives  $R_B$ , it forwards it and performs a snoop operation to record whether the cache has the line (3). If present, the line would not be invalidated because of its transient state; it would be invalidated only if  $A$ 's transaction lost. When  $A$  receives  $r_{B-}$ , it combines it with its snoop outcome (which is negative because of the transient state) and forwards it (4). Then,  $S$  receives  $R_A$  and sends the suppliership to  $A$  (5). Later,  $S$  receives two messages in any order:  $r_{A-}$ , which is transformed into  $r_{A+}$  and forwarded (6), and  $R_B$ , which is forwarded (7). Note that  $R_B$  also invalidates the line from  $S$ 's cache. After that,  $S$  receives and forwards  $r_{B-}$  (8).

Meanwhile,  $B$  receives  $R_A$  and realizes there is a collision (9). Later, when  $B$  receives  $r_{A+}$  (10), it realizes that its transaction has lost, records it, and forwards  $r_{A+}$ . Eventually,  $A$  receives the suppliership (not shown) and  $r_{A+}$  (11), which completes its transaction.  $B$  receives  $r_{B-}$  and, knowing that its transaction has been squashed, retries it (12).

The Ordering invariant has enabled the correct handling of the collision. It ensured that the first of the two requesting nodes after  $S$  in ring order ( $B$ ) sees the two  $r$  in the same sequence as  $S$  saw the two  $R$ . Table 1 summarizes the main events in the example.

**Supplier Not Present, Natural Serialization** In this case,  $A$  receives  $r_{A-}$ , which indicates that no other node can supply the data, before  $A$  observes any other message.  $A$  then gets the data from memory. If  $A$  has received the data by the time it observes  $R_B$ ,  $A$  services  $B$ 's request. Otherwise,  $A$  ignores  $R_B$  because  $A$ 's line is in transient state and, when  $A$  later receives  $r_{B-}$ , it marks it as squashed. In this case, when  $B$  receives  $r_{B-}$ , it retries the transaction. Table 1 summarizes the main events in this specific case.

**Supplier Not Present, Forced Serialization** In this case, after a node initiates a transaction, it sees the  $R$  and  $r$  of the other transaction before receiving its own, squash-free,  $r$ -. Both requesting nodes see the symmetric sequence of messages. In this case, the two nodes cannot decide which transaction is the winner based only on the messages observed. Instead, when each node receives the other's  $r$ -, the node runs by itself an algorithm to decide which node is the winner. Later, when the node finally receives its own  $r$ -, it acts based on the outcome of the algorithm: if the node was the winner, it proceeds to get the data from memory; if the node was the loser, it invalidates the line from its cache (if the winner was a write) and retries the transaction.



**Figure 5.** Comparing a transaction in Eager and Uncorq: message distribution (a) and cache-to-cache transfer latency (b).

There are a few possible algorithms to decide which transaction has priority and, therefore, wins. One is to use node ID numbers and select the highest — this is unfair, but it never ties. Another is to use random numbers attached to transactions and select the highest — this has a very small probability of a tie, but it is fair. Another is to use the transaction type. Specifically, a write hit that sends invalidations takes priority over any other transaction type, while a write miss takes priority over a read miss. Indeed, selecting a write hit minimizes accesses to memory, since the node issuing it already caches the data. Moreover, selecting a write miss over a read miss may speed up transferring a lock variable from one node to another. In our design, we use a hierarchy of the three algorithms. We start with the transaction type, then fall back to the random number, and then to node ID, if necessary.

Table 1 summarizes the main events in this case.

## 4. Unconstrained Snoop Request Delivery

### 4.1. Rationale and Benefits

A shortcoming of the existing embedded-ring protocols such as Eager and Flexible Snooping is that the latency of a ring transaction can be relatively large. This is because the  $R$  messages have to traverse the ring. If the supplier node is far from the requesting one in ring order, the suppliership message may take a relatively long time to be sent.

Intuitively, an embedded-ring protocol should be able to work without requiring  $R$  messages to traverse the ring. This is because, as indicated in Section 3.1, the role of  $R$  messages in a collision is only to order the transactions at the supplier node — while communicating this order to other nodes is left to the  $r$  messages. This idea inspired the Ordering invariant.

Based on this insight, we propose a new protocol called *Unconstrained Snoop Request Delivery (Uncorq)*. The idea is for the  $R$  message to be delivered to all nodes in the fastest possible manner — using any network links and leveraging efficient multicast. Individual  $R$  messages arrive to nodes quickly and, as soon as they arrive, initiate a snoop. When the supplier completes its snoop, it sends the suppliership to the requester. Meanwhile, the  $r$  message traverses the ring as usual. If the  $r$  arrives at a node that has not seen the  $R$  yet, it waits until  $R$  is received and the snoop is completed; then,  $r$  is combined with the snoop outcome and forwarded to the next node in the ring. Figure 5(a) shows the different message distribution in Eager and Uncorq.

The advantage of Uncorq over the Eager and Flexible Snooping algorithms is that, in a cache-to-cache transfer, it reduces the time

between when a node starts a transaction and when it receives the suppliership message. This has an obvious benefit in read misses, since such time is in the critical path of using the requested data. Depending on the memory consistency model, reducing such time also benefits write transactions (Section 5.3).

On the other hand, the time to reception of the  $r$  message is largely the same in all the algorithms. Such time is important for writes because it marks the point when all cached copies of the line are guaranteed to be invalidated. It is also the point when the requester realizes that there is no supplier on chip and, therefore, it has to use data from memory. We address the case of memory accesses in Section 5.4.

Figure 5(b) compares the latencies in a cache-to-cache transfer transaction for Eager and Uncorq. The top diagram shows the time to suppliership reception. It includes the time of  $R$  propagation to the supplier, snoop at the supplier, and suppliership propagation back to the requester. Because Uncorq does not deliver  $R$  messages using the ring, an  $R$  can reach the supplier node sooner ((1) in Figure 5(b)), reducing the time to suppliership reception.

The bottom diagram in Figure 5(b) shows the time to response reception, which is the time it takes for the  $r$  message to traverse the ring. Such time is the same in both algorithms.

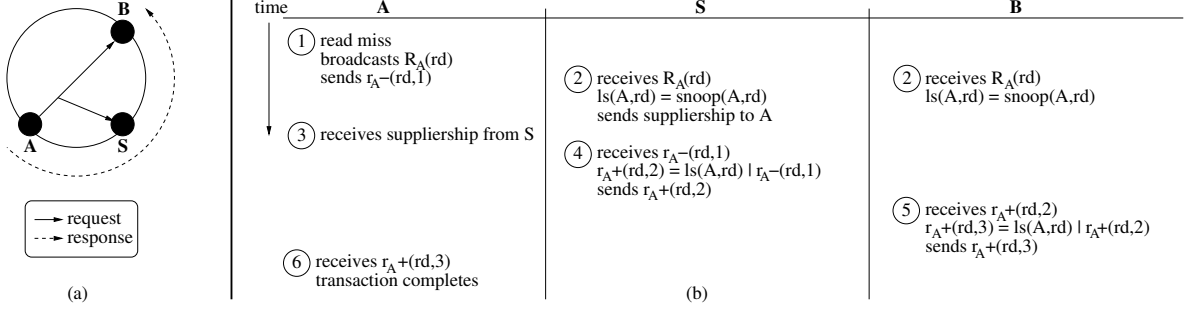
### 4.2. Uncorq in Action

Figure 6 shows how Uncorq services a read miss when there are no collisions. Figure 6(a) shows that  $R$  messages are directly delivered to all nodes, while  $r$  messages proceed as usual.

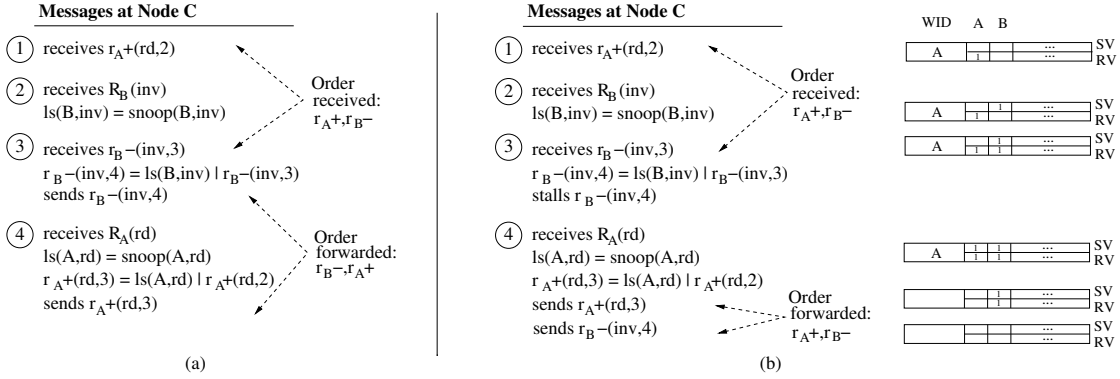
Figure 6(b) shows a timeline of events at each node. When  $A$  suffers a read miss, it broadcasts a read request  $R_A$  to  $S$  and  $B$ , and then forwards  $r_{A-}$  using the ring (1).  $S$  and  $B$  receive  $R_A$  and both initiate snoop operations (2). After  $S$  completes the snoop, it sends the suppliership message to  $A$ .  $A$  receives the message in (3). When  $S$  receives  $r_{A-}$ ,  $S$  combines it with its positive outcome and forwards it as  $r_{A+}$  (4). When  $B$  receives  $r_{A+}$ , it combines it with its local snoop outcome and forwards it (5). Note that, if a  $r$  message arrives at a node such as  $S$  or  $B$  before the corresponding  $R$  has arrived, the  $r$  message is buffered until  $R$  is received and the snoop operation is performed. Finally, when  $A$  receives  $r_{A+}$ , the transaction completes (6).

### 4.3. Ordering Invariant with Uncorq

Supporting the Ordering invariant with the Uncorq algorithm requires a different set of mechanisms than the ones described in Section 3.2 for the Eager and Flexible Snooping algorithms. To understand them, recall why we required FIFO handling of same-line



**Figure 6.** Servicing a read miss with Uncorq when there are no collisions.



**Figure 7.** Breaking the Ordering invariant (a) and enforcing it with the LTT (b) in Uncorq.

messages in Section 3.2. It ensured that, in a collision, the  $r$  order seen by the nodes beyond the supplier node (in ring order) is the same as the  $R$  order seen by the supplier.

Unfortunately, in Uncorq, we can only talk about the ring order of  $r$  messages — not of  $R$  messages. Since  $R$  messages travel along any network links, two  $R$  messages may arrive at a node in a given order and at a second one in the opposite order.

What matters is the order in which the two  $R$  messages are processed at the supplier node. To support the Ordering invariant, such order should determine the order in which the nodes following the supplier in ring order — up until the first of the two requesting nodes — receive the  $r$  messages. This cannot be ensured only by FIFO transfer and FIFO processing.

To see why not, consider the example in Figure 7(a), which shows the messages at a node  $C$  that is between the supplier node  $S$  and  $B$  in Figure 6(a). Assume that  $A$  starts a read transaction at the same time as  $B$  starts an invalidation one. Suppose that  $R_A$  arrived at  $S$  first and, therefore, the  $A$  transaction was the winner. In the figure, we see that  $C$  receives  $r_{A+}$  first (1), before it receives  $R_A$  — which was delayed in the network. Since  $R_A$  is missing,  $C$  cannot process  $r_{A+}$ . Then,  $C$  receives  $R_B$  and performs a snoop operation (2). After that,  $C$  receives  $r_{B-}$  and, since the snoop is done,  $r_{B-}$  is combined with the local snoop outcome and forwarded (3). Later,  $C$  receives  $R_A$ , performs the snoop and forwards  $r_{A+}$  (4). This breaks the invariant.

To support the Ordering invariant with Uncorq, we argue that the hardware must enforce the following two mechanisms, which refer to messages requesting the same line:

1. After a supplier node processes a winning request message  $R_i$ , the node cannot forward any response  $r_{j-}$  (where  $j \neq i$ ) before it forwards  $r_{i+}$ .
2. If a node receives a positive response  $r_{i+}$ , the node cannot forward any response  $r_{j-}$  received after  $r_{i+}$  until it receives  $R_i$  and forwards response  $r_{i+}$ .

The first mechanism ensures that the positive response  $r_{i+}$  leaves the supplier before the negative ones. The second mechanism ensures that  $r_{i+}$  is not overtaken by any negative response  $r_{j-}$  in any of the nodes that follow the supplier in ring order. Therefore, one of the two requesting nodes will see  $r_{i+}$  before  $r_{j-}$ . Note that two negative responses can always overtake each other.

These two mechanisms, combined, require that a node that has either (i) performed a snoop that triggered a suppliership transfer or (ii) received a positive  $r$ , keeps buffering incoming  $r$  messages to the same line until the node can forward the positive  $r$ .

This adds a third condition to the conditions to forward responses presented in Section 2.1: (3) if the node has performed a snoop that triggered a suppliership transfer or has received a positive response for the same memory line, a combined response cannot be forwarded until the positive response has been forwarded.

#### 4.4. Protocol Operation

With Uncorq, the types of collisions and how they are handled are largely like in Eager (Section 3.3). The exception is that Uncorq introduces three new collision instances: one under Forced Serialization with supplier present and two under Forced Serialization with no supplier present. We discuss them next. Table 2 shows all the collision types, but focuses on the new instances.

Supplier Node	Serializ.	Process
Present	Natural	Same as in Eager
Present	Forced	Case $S$ receives $R_A$ first: same as in Eager Case $S$ receives $R_B$ first: $B$ receives $R_A$ ( $B$ is in transient) $S$ receives $R_B$ $S$ sends suppliership to $B$ $S$ receives $R_A$ $B$ receives $r_{B+}$ and records $A$ 's trans. is loser $B$ receives $r_{A-}$ and marks $r_{A-}$ as squashed $A$ receives $r_{A-}$ and retries its transaction
Not Present	Natural	Same as in Eager
Not Present	Forced	$A$ receives $R_B$ ( $A$ is in transient) Case $A$ receives $r_{B-}$ and then $r_{A-}$ : $r_{B-}$ arrives: $A$ runs winner-selection algorithm $r_{A-}$ arrives: $A$ gets data from mem. or retries Case $A$ receives $r_{A-}$ and then $r_{B-}$ : $r_{A-}$ arrives $r_{B-}$ arrives: $A$ runs winner-selection algorithm $A$ gets data from mem. or retries

**Table 2.** Handling collisions in Uncorq.

**Supplier Present, Forced Serialization** With Eager, since  $R$  messages use the ring,  $R_A$  reaches  $S$  before  $R_B$  does. With Uncorq,  $R_B$  may reach  $S$  before  $R_A$ . This is a new collision instance. In this case,  $B$  receives the suppliership message and, after that,  $r_{B+}$ . At the time  $B$  receives  $r_{B+}$ , since  $B$  has already seen  $R_A$ , it records that  $A$ 's transaction is the loser. Later, when  $B$  sees  $r_{A-}$ ,  $B$  marks it as squashed. Finally, when  $A$  sees  $r_{A-}$ , it retries.

**Supplier Not Present, Forced Serialization** In this case, with Eager, since  $r$  messages cannot get reordered, each requesting node sees the other node's  $r$ - before seeing its own  $r$ -. With Uncorq, two negative  $r$  messages can get reordered — i.e., a  $r$ - that arrives first at a node waits for its  $R$  to arrive, while a  $r$ - that arrives second is combined with its  $R$ 's snoop outcome and forwarded. This effect causes two new collision instances.

To describe these collisions, consider node  $A$ , which has already seen  $R_B$ . The first new collision appears when  $r_{A-}$  overtakes  $r_{B-}$  and arrives at  $A$  first. In this case,  $A$  sees  $r_{B-}$  and  $r_{A-}$  out of order. The second new collision appears when  $B$  issues  $R_B$  and  $r_{B-}$  after having forwarded  $r_{A-}$ , but  $r_{B-}$  overtakes  $r_{A-}$  and arrives at  $A$  first. In this case, while  $A$  sees what appears to be the usual message order,  $B$  is unaware of any collision. We need to ensure that, if necessary,  $B$  is made aware of the collision — so that it does not appear as a Natural Serialization instance to  $B$ .

These two reorderings, together with the case of no reordering, are handled in the same way — with only a small extension over Eager's way. Specifically: (i) when  $A$  receives  $r_{B-}$ , it runs the winner-selection algorithm; (ii) when  $A$  receives the latest of  $r_{B-}$  and  $r_{A-}$ , it acts on the algorithm's outcome (get data from memory or retry); and (iii) as part of running the algorithm, if  $A$  is the winner, augment  $r_{B-}$  with a *Loser Hint* bit. When a node receives its own  $r$ - with such bit set, it knows it has to retry. Table 2 summarizes the actions.

## 5. Implementation Issues in Uncorq

### 5.1. Supporting the Ordering Invariant

To support the Ordering invariant, each node has a set-associative table called *Local Transaction Table* (LTT). The LTT records information on transactions that are *in flight* at the node

— namely, any transaction for which the node has received the  $R$  and/or the  $r$  message, but for which it has not yet forwarded the  $r$  combined with the snoop outcome.

The LTT stalls  $r$  messages when necessary to enforce the Ordering invariant. Specifically, when a node has either (1) received a  $R$  message for a line and produced a positive snoop outcome — i.e., the node was the supplier for the line — or (2) received a  $r+$  message for a line, then the LTT stalls and buffers all subsequent  $r$  messages to the same line. Only after the  $r$  and snoop outcome for the line are combined into a  $r+$  message and forwarded, can the buffered  $r$  messages be forwarded as well.

All simultaneous in-flight transactions at a node for a given memory line are mapped to the same LTT entry. An entry consists of: (i) the line address, (ii) the ID of the node that initiated the in-flight transaction with the positive snoop outcome or the  $r+$  message — the *Winning Node ID* (WID) field — and (iii) two bit vectors with as many bits as nodes. The bit vectors are the *Snoop Vector* (SV) and the *Response Vector* (RV). They record which nodes initiated the in-flight transactions for which the local node has performed a snoop operation or received a  $r$  message, respectively.

When a node receives a  $R$  or a  $r$ , the LTT is accessed. If an entry is not already allocated for the line, one is allocated with the line address. If this is a  $R$ , the node performs a snoop operation and, after that, the corresponding bit in SV is set. Moreover, if the snoop outcome is positive, when the suppliership is sent to the requester node, the WID is set to the requester's ID. Instead, if this is a  $r$ , the corresponding bit in RV is set. Moreover, if the  $r$  is positive, the WID is set to the requester node's ID.

According to the conditions presented in Sections 2.1 and 4.3, a node is ready to forward the  $r$  of a transaction when: (1) the corresponding SV bit is set — i.e., the node has completed the snoop; (2) the corresponding RV bit is set — i.e., the node has received the  $r$ ; and (3) either WID is clear — i.e., no node has obtained the suppliership of the line yet — or WID is equal to the node that initiated this transaction — i.e., the initiator of this transaction has received the suppliership of the line. When a  $r$  is forwarded, the corresponding SV and RV bits are cleared and, if applicable, the WID field is cleared. Finally, when the SV and RV bit vectors are clear, the LTT entry is deallocated.

Figure 7(b) shows how the mechanism works. When  $C$  receives  $r_{A+}$ , an LTT entry is allocated (1). It has the line address, the RV bit for  $A$  set to one, and WID set to  $A$ . When  $C$  receives  $R_B$ , it performs a snoop operation (2). When the snoop operation completes with a negative snoop outcome, the SV bit for  $B$  gets set. When  $C$  receives  $r_{B-}$ , the RV bit for  $B$  gets set, and the hardware verifies that the SV bit for  $B$  is already set (3). At this point, the hardware checks whether WID is  $B$ . Since it is not, the combined response  $r_{B-}$  is stalled. When  $C$  receives  $R_A$ , it initiates a snoop operation (4). When this operation completes with a negative snoop outcome, the hardware sets the SV bit for  $A$ , verifies that the RV bit for  $A$  is already set, and checks whether the ID in WID is  $A$ . Since it is, the hardware forwards  $r_{A+}$  and resets both the SV and RV bits for  $A$ , as well as WID. After that, the hardware checks if there are any stalled  $r$  messages ready to go. Since  $r_{B-}$  is,  $r_{B-}$  is immediately forwarded, resetting the SV and RV bits for  $B$ . As a result,  $r_{A+}$  and  $r_{B-}$  have not been reordered, preserving correct transaction serialization.

LTT-induced stalling of negative responses cannot lead to a deadlock. The reason is that, given multiple colliding transactions,

only messages from the single, already-identified winner transaction can stall messages from the other, loser transactions. Moreover, no message from the winner transaction can be stalled.

Finally, to size the LTT, a designer may choose one of two different approaches. One is to structure it so that it can support the maximum possible number of in-flight transactions at a node. This requires approximately as many entries ( $E$ ) as number of nodes ( $N$ ) times the maximum number of outstanding transactions per node ( $T$ ), namely  $E=N \times T$ . Moreover, the LTT associativity has to be equal to the maximum possible number of in-flight transactions that can map to an LTT set. To reduce hardware requirements, we envision such number ( $e$ ) to be much smaller than  $E$ . This can be done by placing restrictions on the addresses of the  $T$  outstanding transactions of a node. For example, a banked MSHR may allow only  $t$  outstanding transactions that map to the same LTT set. Consequently, the LTT associativity only needs to be  $e=N \times t$ . Overall, even with a sizable number of entries and associativity, accessing the LTT takes only a handful of cycles.

The second approach is to reduce the size and associativity of the LTT to save real estate, while risking running out of entries in rare cases. When the latter happens, the LTT would send a negative acknowledgment to the requester of the dropped message, which would then retry. Since this approach requires changes to the protocol to avoid inducing livelocks, we do not consider it here.

## 5.2. Forward Progress

There are two aspects to forward progress, namely system and individual-node forward progress. System forward progress is guaranteed in Eager, Flexible Snooping and Uncorq because, in any collision, at least one of the requests succeeds.

While the collision resolution algorithms described in Sections 3.3 and 4.4 are expected to provide, in practice, individual-node forward progress, they do not guarantee it. Consequently, after a node has unsuccessfully retried the same transaction a certain number of times, an algorithm needs to kick in to ensure that the transaction eventually succeeds. Such algorithm is different in Eager (or Flexible Snooping) and in Uncorq. In both cases, however, it tries to ensure that the starving node's  $R$  message gets to the supplier before the other nodes's  $R$  messages.

### 5.2.1. Node Forward Progress in Eager

In Eager,  $R$  messages use the ring. Consequently, a starving node  $N$  can intercept all the conflicting  $R$  messages that it sees, stall them, and insert its own  $R$  ahead of them. This ensures that  $N$  will succeed before the nodes that are between the supplier node and  $N$  in ring order. However, the nodes that are between  $N$  and the supplier in ring order can potentially complete their transactions before  $N$ 's. Fortunately, as the transactions of these nodes complete, the supplier status moves from node to node counter-ring wise, closer to  $N$ . If there are many starving nodes, in the worse case, after as many transactions as there are nodes between  $N$  and the supplier in ring order,  $N$ 's transaction will succeed.

### 5.2.2. Node Forward Progress in Uncorq

In Uncorq, since  $r$  messages use the ring, they can be used to guarantee node forward progress. Specifically, we add a new field to each  $r$  message, called starving node ID (SNID). A starving node  $N$  intercepts all the conflicting  $r$  messages that it sees, and places its

ID in their SNID before letting them go. When a  $r$  message reaches its destination node, if it is a positive  $r$ , it means that the destination has become the new supplier. In this case, the hardware checks the SNID. If it is not null, it allocates an entry in the new supplier's LTT and saves the SNID in the WID field. It is, therefore, reserving the next winner transaction for  $N$ . The hardware will stall any new transactions from other nodes to that line — for a limited period of time — until it processes the transaction from  $N$ . If there are several starving nodes, they all update the SNID field of  $r$  messages, and the node closest to the supplier node in counter-ring order wins. As in Eager, the supplier status moves counter-ring wise, closer to  $N$ . In the worse case, after as many transactions as nodes between  $N$  and the supplier in ring order,  $N$ 's transaction will succeed.

## 5.3. Fences and Memory Consistency Issues

When the requester node receives the suppliership message, the transaction is bound and cannot be undone. If the transaction originated from a load, the load is then complete. If it originated from a store, there may still be copies of the line in other nodes to be invalidated. Only when the requester node receives the  $r$  message it is guaranteed that all copies have been invalidated. At that point, the store is complete.

From this, it is clear that, compared to Eager or Flexible Snooping, Uncorq speeds up loads. However, to understand the full impact of Uncorq on loads and stores, consider how accesses are ordered within and across threads.

Within a thread, a fence instruction stalls the issue of requests after the fence until those requests that precede the fence are complete. In relaxed memory consistency models such as that of IBM's PowerPC [12], fences are largely associated only with synchronization operations — not with all accesses as in strict models such as Sequential Consistency. In the absence of fences, loads and stores can overlap and reorder. Even in the presence of fences, there are optimizations to enable request overlap and reorder (e.g., [5]). Still, in this environment, Uncorq largely speeds up only loads.

Consider now the interaction between threads — a transaction from node  $A$  followed by one from node  $B$  to the same line. In Sections 3 and 4, we have assumed that node  $A$  only changes the state of the line in its cache to a stable state after it receives the  $r+$  message. Moreover, it can send suppliership to  $B$  only after that.

In reality, we can optimize this case:  $A$  can potentially provide suppliership to  $B$  right after  $A$  gets suppliership, even before  $A$  receives the  $r+$  message. As a result, a load or a store in  $B$  would get the data faster. This is clearly safe when  $A$ 's transaction is a load because, by this time,  $A$ 's load is complete. Importantly, it is also correct when  $A$ 's transaction is a store — if we use a relaxed memory consistency model such as PowerPC's. This is because this kind of model does not require write atomicity; the data can be provided to another processor even though not all other processors have been invalidated. Consequently, this is a case where Uncorq speeds up stores. While we do not evaluate this optimization in this paper, it can speed up the situation where a processor releases a lock and another acquires it.

## 5.4. Optimization: Hardware Data Prefetching

Uncorq focuses on reducing the latency of cache-to-cache transfers. If we also want to reduce the latency of memory-to-cache transfers, we must augment Uncorq with further optimizations.

One such optimization is hardware data prefetching. In this case, the hardware at the requesting node sends a prefetching request to the on-chip memory controller at the same time as it issues a  $R$  to the ring. The controller gets the data from memory. If the ring transaction returns a  $r$ - message, the requesting node’s hardware sends a request to the memory controller, which provides the line. If an on-chip node provides the data, the data in the controller is discarded.

Issuing a prefetch at every cache miss wastes power and bandwidth. Consequently, this optimization requires a mechanism that predicts, at the time of the miss, whether or not the line will be provided by a node in the ring. Depending on the prediction, a request is sent to memory controller in addition to the ring.

As an example, we describe a simple prefetch predictor that we evaluate in Section 7. Each node has a Node Prefetch Predictor (NPP), while the memory controller has a Controller Prefetch Predictor (CPP). The NPP is a table with the line addresses of cache miss and invalidation transactions recently seen in the ring. When a node’s hardware sends a  $R$  to the ring, it checks the NPP. If the requested address is not found there, a prefetch request is sent to the memory controller.

The CPP is a table where each entry corresponds to one page. Each entry has as many bits as lines in a page. Whenever the memory controller brings in a line from memory due to a cache miss or a prefetch, the corresponding bit gets set; whenever a dirty line is written back, the bit gets cleared. When the memory controller receives a prefetch request from a node, it checks the CPP. Only if the corresponding bit is clear, is the line fetched from memory. When the line is received, it is kept in a small buffer for a certain number of cycles in case the requesting node wants it.

There are other possible optimizations for memory-to-cache transfers. Their analysis is beyond this paper’s scope.

## 5.5. Optimization: Supplier Status Transfer

In our description of Uncorq, every successful transaction provides the requester with the supplier status. While this makes the protocol simpler to describe, it implies that when two cache-to-cache *read* transactions collide, unless they are naturally serialized, one of them needs to be squashed. Squashing one transaction in such a two-read collision is unintuitive and limits concurrency.

It can be shown that the Uncorq protocol can be implemented so that cache-to-cache read misses do not transfer supplier status. Under this implementation, colliding cache-to-cache read transactions are always serviced without squashes. We do not evaluate this implementation in this paper.

## 6. Evaluation Methodology

We use detailed simulations using the SESC [10] cycle-accurate simulator to evaluate Uncorq, Uncorq plus the prefetching optimization of Section 5.4 (Uncorq+Pref), Eager, and two Flexible Snooping algorithms from [14], namely SupersetCon and SupersetAgg. In our evaluation, the Uncorq and Uncorq+Pref improvements only apply to read transactions — in write transactions,  $R$  messages still traverse the ring and do not use prefetching.

The architecture modeled is a CMP with 64 cores connected in a 2D torus with xy routing. Each core is a 4-issue out-of-order processor with private L1 and L2 caches. We estimate that this design

will fit in a large chip in a near-future technology. The system uses release memory consistency. All algorithms use exactly the same network and, therefore, see the same available network bandwidth. Table 3 lists the main parameters.

System	CMP with 64 cores
Core	4-issue, out-of-order, and 4GHz
D-L1 (and I-L1)	32KB, 4-way, 64B line, 2 cyc round trip
Unified L2	512KB, 8-way, 64B line, 7 cyc round trip
On-chip network	8x8 2D torus, 8 cyc/hop, 2GHz, 64GB/s, xy routing
Memory	DDR2-800, 224 cyc round trip, 4KB pages
Node prefetch pred.	8K line addresses
Cont. prefetch pred.	16K entries, 64 bits/entry
LTT	512 entries, 64-way

**Table 3.** Parameters of the architecture simulated. All latencies are given in processor cycles.

We run 11 SPLASH-2 applications [17] (all but *volrend*, which does not run on our infrastructure), SPECjbb 2000 and SPECweb 2005. SPLASH-2 applications run to completion after skipping initialization. They execute with 64 threads. The SPEC applications run for over 750M instructions after skipping initialization. To capture system-level references, we run them through a Simics [16] front-end that interfaces to SESC. They run with 8 threads. In our experiments, we run 8 concurrent instances of a given SPEC application, for a total of 64 threads. However, there is no sharing between the 8 instances. SPECjbb uses an 8 warehouse configuration, while SPECweb uses the e-commerce workload.

## 7. Evaluation

In this evaluation, we examine read miss latency (Section 7.1), overall performance (Section 7.2), impact of the prefetching optimization (Section 7.3), and a comparison to cache-coherent Hypertransport (Section 7.4).

### 7.1. Read Miss Latency

Figure 8 compares the read miss latency with Uncorq and Eager. This latency is the number of processor cycles from the time the read miss is declared in the requester’s L2 cache until the data arrives at the requester’s L1 cache.

Figures 8(a) and (b) show a histogram of the read miss latency for *cache-to-cache transfers only*, for Eager and Uncorq, respectively, in *fmm* — one of the SPLASH-2 applications. We limit the data to cache-to-cache transfers because these are the misses that Uncorq optimizes. The figures also show the cumulative distribution. Other applications behave similarly.

Comparing Figures 8(a) and (b), we see that the latencies of cache-to-cache transfers in Eager are typically much longer than in Uncorq. This is because requests reach the supplier nodes much faster in Uncorq. Moreover, the fact that both figures exhibit sharp spikes rather than smooth curves is an indication that there is not much network contention. Figure 8(a) has many spikes, which are due to the many distinct latencies between a requester node and a supplier. The variability of the spike heights is due to the network topology rather than to any uneven distribution of the data across nodes.

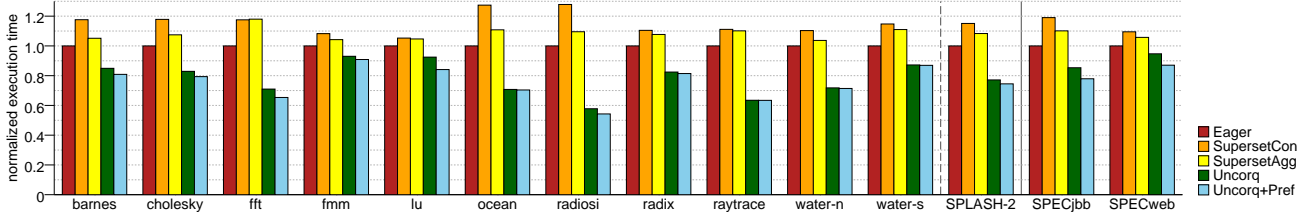
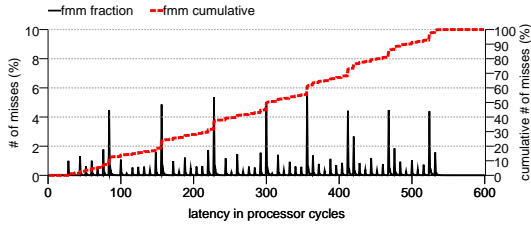
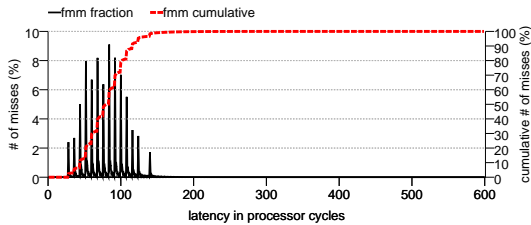


Figure 9. Execution time normalized to Eager.



(a) Cache-to-cache read miss latency in *fmm* with Eager



(b) Cache-to-cache read miss latency in *fmm* with Uncorq

Application	Read Miss Lat. (Proc. Cycles)		(Eager-Uncorq)/Eager (%)	c2c (%)
	Eager	Uncorq		
barnes	319	107	66	97
cholesky	354	145	59	90
fft	517	391	24	54
fmm	345	144	58	90
lu	385	195	49	82
ocean	454	330	27	99
radiosity	301	80	74	99
radix	316	95	70	99
raytrace	320	106	67	95
water-n squared	365	158	57	90
water-spatial	312	92	70	98
SPLASH-2 avg.	363	168	56	90
SPECjbb	416	252	39	72
SPECweb	598	522	13	32

(c) Read miss latency characteristics

Figure 8. Read miss latency with Uncorq and Eager.

Figure 8(c) shows data on *all* read misses for all applications. Columns 2-3 list the average read miss latency in Eager and Uncorq, respectively, while Column 4 shows the reduction as we go from Eager to Uncorq. We see that, for SPLASH-2 applications, Uncorq reduces the latency by an average of 56%. This is a substantial impact. For SPECjbb and SPECweb, the reduction is a more modest 39% and 13%, respectively. To understand these numbers, the last column of Figure 8(c) shows the fraction read misses serviced with cache-to-cache transfers. We can see that SPLASH-2 applications have a large fraction of cache-to-cache transfer misses

— on average, 90%. On the other hand, this fraction is a lower 72% and 32% for SPECjbb and SPECweb, respectively. Therefore, these applications cannot benefit as much from Uncorq.

## 7.2. Overall Performance

Figure 9 shows the execution time of the applications in Eager, SupersetCon, SupersetAgg, Uncorq, and Uncorq+Pref. In each application, the execution time is normalized to that of Eager.

The figure shows that Uncorq is consistently faster than Eager. Uncorq reduces the execution time of the applications by an average of 23% for SPLASH-2, 15% for SPECjbb, and 5% for SPECweb. This reduction is largely correlated with the reduction in read miss latency shown in Figure 8(c).

In addition, Uncorq+Pref reduces the execution time even further. Compared to Eager, Uncorq+Pref reduces the execution time of the applications by an average of 26% for SPLASH-2, 22% for SPECjbb, and 13% for SPECweb. The impact of the prefetching optimization is higher for the commercial workloads because they have a lower fraction of cache-to-cache transfers (Figure 8(c)).

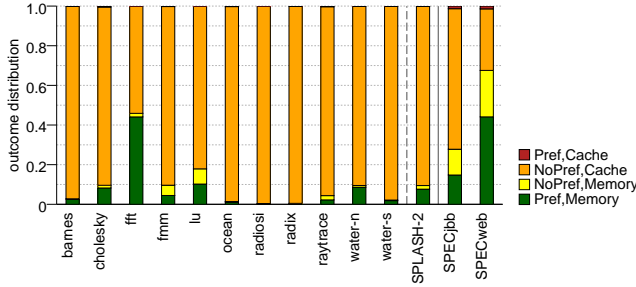
Comparing SupersetCon and SupersetAgg to Eager, we see that both perform worse. This is in contrast to the results in [14], which showed that both algorithms have a performance similar to Eager. The reason for the difference is that the two papers evaluate different architectures: while [14] evaluates a multi-CMP multiprocessor, this paper models a single CMP. Such architecture has a much lower inter-node latency. As a result, the latency of the snoop filters in SupersetCon and SupersetAgg adds significantly to the request delivery latency, rendering these algorithms less attractive in our setting. A more detailed analysis is presented in Strauss’s thesis [13].

## 7.3. Impact of the Prefetching Optimization

Figure 10 provides insight into the prefetching optimization. Figure 10(a) classifies read misses into four categories: (1) a prefetch request is issued but the miss is serviced from a cache (*Pref,Cache*); (2) no prefetch request is issued and the miss is serviced from a cache (*NoPref,Cache*); (3) no prefetch request is issued but the miss is serviced from memory (*NoPref,Memory*); and (4) a prefetch request is issued and the miss is indeed serviced from memory (*Pref,Memory*).

We see that the *Pref,Cache* category is very small. This means that the prefetching optimization is not wasteful: the great majority of prefetching requests bring useful data from memory. Moreover, from the two bottom categories, we observe that the prefetcher is able to prefetch a good percentage of all memory-to-cache transfers.

Figure 10(b) shows the impact of the prefetching optimization on the average read miss latency. Column 2 shows the read miss latency for Uncorq+Pref, while Column 3 shows the reduction in



(a) Breakdown of read misses

Application	Read Miss Lat. in Uncorq+Pref (Proc. Cycles)	(Uncorq - Uncorq+Pref)/Uncorq (%)
barnes	99	7
cholesky	126	13
fft	294	25
fmm	134	7
lu	174	11
ocean	236	28
radiosity	78	2
radix	94	1
raytrace	101	4
water-nsquared	148	6
water-spatial	88	5
SPLASH-2 avg.	143	10
SPECjbb	219	13
SPECweb	427	18

(b) Read miss latency characteristics

**Figure 10.** Impact of the prefetching optimization.

such latency as we go from Uncorq to Uncorq+Pref. From the data, we see that such a reduction is significant — 10% on average for the SPLASH-2 applications and 13-18% for the commercial applications. The reduction is largely correlated with the reduction in execution time. One notable exception is *ocean*, where prefetching reduces the read miss latency but not the execution time. This is because, when the prefetching optimization is used, the application changes its sharing behavior and suffers from thread imbalance.

## 7.4. Cache-Coherent Hypertransport

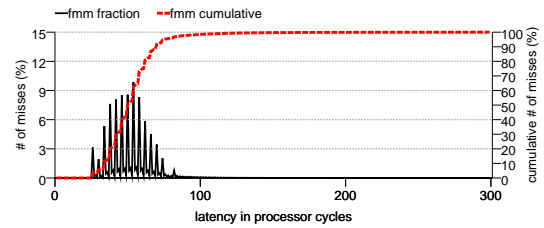
An alternative to Uncorq in a point-to-point network is a directory-based protocol. However, directory protocols require the addition of a relatively expensive directory structure. A simpler alternative that is also broadcast-based like Uncorq is AMD’s cache coherent Hypertransport (HT) [4]. In HT, like directories, each address is assigned a serialization point in the network. When a node misses on an address, it sends a request to the corresponding serialization point. That point then broadcasts the request to all other nodes, which in turn directly respond to the requester. The existence of a serialization point for each address simplifies transaction ordering.

Compared to Uncorq, HT (like directories) has longer cache-to-cache transaction latencies. This is because, due to the indirection of the serialization point, a transaction needs three “node hops” to complete instead of the two in Uncorq. In addition, since reply messages to the requester are not combined in HT, HT induces more network traffic than Uncorq. Finally, HT requires a new hardware

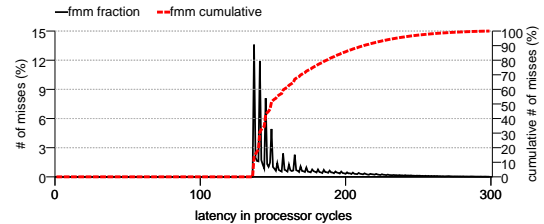
module in each serialization point in the network that collects and broadcasts messages.

On the other hand, the latency of memory-to-cache transactions is longer in Uncorq than in HT. This is because the *r* message that tells the requester in Uncorq that no other node can supply the data has to sequentially visit all nodes.

To assess some of these effects, Figure 11 compares the latency and traffic characteristics of read misses in Uncorq and in (an estimate of) HT. We use Uncorq rather than Uncorq+Pref because it is fairer — HT could also use data prefetching. Starting from the top, Figures 11(a) and (b) show a histogram of the read miss latency for *cache-to-cache transfers only*, for Uncorq and HT, respectively, in *fmm*. Figure 11(a) repeats Figure 8(b) with an expanded Y axis. Comparing Figures 11(a) and (b), we see that Uncorq reduces the latency of cache-to-cache transfer read misses substantially.



(a) Cache-to-cache read miss latency in *fmm* with Uncorq



(b) Cache-to-cache read miss latency in *fmm* with HT

Application	Read Miss Lat. in HT (Proc. Cycles)	(HT - Uncorq) / HT (%)	
		Latency	Traffic
barnes	172	38	56
cholesky	273	47	55
fft	431	9	52
fmm	190	24	55
lu	197	1	55
ocean	460	28	56
radiosity	144	44	56
radix	213	55	56
raytrace	153	31	56
water-nsquared	277	43	55
water-spatial	149	38	56
SPLASH-2 avg.	242	33	55
SPECjbb	205	-23	54
SPECweb	268	-95	48

(c) Latency and traffic characteristics of read misses

**Figure 11.** Comparing Uncorq and Hypertransport (HT).

Figure 11(c) shows data on *all* read misses for all applications. Column 2 lists the average read miss latency in HT. Moreover, Columns 3-4 compare Uncorq to HT, showing how much of the latency and traffic of read misses is saved by Uncorq. From the

data, we see that Uncorq reduces the read miss latency of SPLASH-2 applications by an average of 33%. On the other hand, it increases the latency of read misses relative to HT for SPECjbb and SPECweb. This is because these applications have a sizable fraction of memory-to-cache transfers, which are faster in HT. However, Uncorq generates much less read miss traffic than HT. Indeed, as shown in Column 4, Uncorq reduces the traffic (in bytes) by an average of 55% for SPLASH-2 applications, and 54% and 48% for SPECjbb and SPECweb, respectively. These are all large reductions, which can have a large impact on execution time when the system is limited by network bandwidth.

## 8. Related Work

Barroso and Dubois propose the use of a slotted ring to implement snoopy cache coherence [1]. Their design is different in that they propose using the ring as the *physical network topology*, while we embed a logical unidirectional ring in any network. In addition, they use time slots to send messages, while we impose no timing constraints.

In previous work, we propose embedding a ring in any network to provide ordering in a snoopy network-based cache coherence system [14]. We suggest adaptive forwarding and filtering algorithms called Flexible Snooping to improve performance and reduce energy consumption. However, these algorithms require requests to use the ring, which limits performance. In this paper, we take embedded-ring protocols one step further by allowing requests to use any path in the network.

Marty and Hill propose Ring-Order, a physical-ring protocol that avoids transaction retries by intercepting response-and-data messages on the ring and completing transactions in ring position order [7]. Although their proposal is suitable for physical rings, we feel it may not be profitable on an embedded-ring system. The reason is that Ring-Order would require data to follow the embedded ring, which would add long latencies to the data reception path. However, Uncorq and Ring-Order could be combined to reduce request delivery latency while avoiding retries.

Martin *et al.* propose Destination-Set Prediction, which uses the sharing behavior of applications to predict the set of nodes that need to receive a given request [6]. Our prefetch predictor is different in purpose and implementation. While their goal is to predict which nodes should receive a request, our goal is to decide whether any node can supply the requested line. Because of this, our predictor is simpler and only records addresses. However, their techniques and other techniques to improve snoopy protocols, such as JETTY [9] and coarse-grain coherence tracking [3, 8], can also be applied in our framework.

Prefetching predictors like ours are also proposed in [11]. Their proposal focuses on using the address of invalid or shared lines found in the caches to predict line presence in other caches.

## 9. Conclusions

While snoopy cache coherence based on embedded logical rings is relatively inexpensive to implement, all proposed designs required that snoop requests be delivered using the logical ring — lengthening miss latency and limiting performance.

To address this problem, this paper made two contributions. First, it introduced the Ordering invariant, which ensures the cor-

rect serialization of colliding transactions in these protocols. Second, using insights from this invariant, it presented Uncorq, a protocol that removes the requirement that snoop requests traverse the ring. Instead, they are delivered using any network path, as long as snoop responses — which are typically off the critical path — use the logical ring. This substantially reduces transaction latency.

Our results indicated that Uncorq is very effective. On a 64-node CMP, Uncorq improved the performance, on average, by 23% for SPLASH-2 applications and by 10% for commercial applications. Moreover, with an additional simple prefetching optimization, the performance improvement was, on average, 26% for SPLASH-2 applications and 18% for commercial applications.

## References

- [1] L. A. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-Based Multiprocessors. In *International Symposium on Computer Architecture*, May 1993.
- [2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *International Symposium on Computer Architecture*, June 1998.
- [3] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *International Symposium on Computer Architecture*, June 2005.
- [4] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture, Present and Future. *IEEE Micro Magazine*, March/April 2007.
- [5] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *International Conference on Parallel Processing*, August 1991.
- [6] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Trade-off in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, June 2003.
- [7] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-Based Chip Multiprocessors. In *International Symposium on Microarchitecture*, December 2006.
- [8] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *International Symposium on Computer Architecture*, June 2005.
- [9] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *International Symposium on High-Performance Computer Architecture*, January 2001.
- [10] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. Sarangi, P. Sack, and P. Montesinos. SESC Simulator, January 2005. <http://sesc.sourceforge.net>.
- [11] X. Shen, J. Huh, and B. Sinharoy. US Patent #7,266,642: Cache Residence Prediction, 2007.
- [12] E. Sikha, R. Simpson, C. May, and H. Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. 1994.
- [13] K. Strauss. *Cache Coherence in Embedded-Ring Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.
- [14] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *International Symposium on Computer Architecture*, June 2006.
- [15] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. In *IBM Journal of Research and Development*, January 2002.
- [16] Virtutech. Virtutech Simics. <http://www.virtutech.com/products>.
- [17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, June 1995.