In-Network Snoop Ordering (INSO): Snoopy Coherence on Unordered Interconnects

Niket Agarwal, Li-Shiuan Peh and Niraj K. Jha
Department of Electrical Engineering
Princeton University, Princeton, NJ, 08544
niketa@princeton.edu, peh@princeton.edu, jha@princeton.edu

Abstract

Realizing scalable cache coherence in the many-core era comes with a whole new set of constraints and opportunities. It is widely believed that multi-hop, unordered on-chip networks would be needed in many-core chip multiprocessors (CMPs) to provide scalable on-chip communication. However, providing ordering among coherence transactions on unordered interconnects is a challenge. Traditional approaches for tackling coherence either have to use ordered interconnects (snoop-based protocols) which lead to scalability problems, or rely on an ordering point (directory-based protocols) which adds indirection latency.

In this paper, we propose In-Network Snoop Ordering (INSO), in which coherence requests from a snoop-based protocol are inserted into the interconnect fabric and the network orders the requests in a distributed manner, creating a global ordering among requests. Essentially, when coherence requests enter the network, they grab snoop-orders at the injection router before being broadcasted. A snoop-order specifies the global ordering of the particular request with respect to other requests. Before requests reach their destinations, they get ordered along the way, at intermediate routers and destination network interfaces. Our logical ordering scheme can be mapped onto any unordered interconnect. This enables a cache coherence protocol which exploits the low-latency nature of unordered interconnects without adding indirection to coherence transactions. Our full-system evaluations compare INSO against a directory protocol and a broadcast based Token Coherence protocol. INSO outperforms these protocols by up to 30% and 8.5%, respectively, on a wide range of scientific and emerging applications.

1 Introduction

With continued transistor scaling providing chip designers with billions of transistors, architects have embraced many-core architectures to deal with increasing design complexity and power consumption [1, 2, 17, 33]. With memory being shared by an increasing number of cores, a scalable cache coherence mechanism is imperative for these systems. Traditional approaches to cache coherence are broadcast-based snoopy protocols[1] and directory-based protocols. Broadcast-based snoopy protocols have been the most commonly used approach to building symmetric multiprocessors (SMPs) [9, 16, 26]. These protocols rely on ordered interconnects like a bus or tree to ensure total ordering of transactions. The primary advantage of snoop-based protocols is that they have direct cache-to-cache transfers and do not require a directory structure. However, the main limitation of these protocols is that they rely on ordered interconnects, which do not scale beyond a moderate number of cores. They also have the bandwidth overhead of broadcasts.

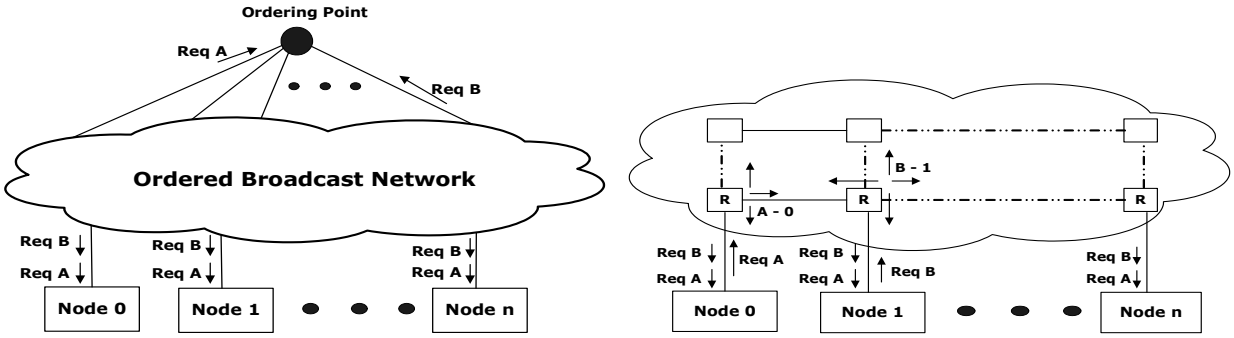
Directory protocols do not require ordered interconnects, as they rely on distributed ordering points and explicit message acknowledgments to achieve request ordering. This enables highly scalable interconnects, such as packetized meshes. Directory protocols are also not broadcast in nature. This imposes lower bandwidth requirements on the interconnect fabric. However, there is an added latency penalty introduced by directory indirection, along with an additional cost associated with the storage and manipulation of directory state.

In the past, there has been considerable effort to retain and scale snoopy protocols by adapting them for split transaction buses [16], hierarchical buses [26], and address broadcast trees [9] that provide a "logical bus" ordering. Expanding further, existing products, like the IBM Power4 and Power5, retain and scale snoopy coherence protocols onto a ring interconnect [27]. One of the reasons why so much effort has been devoted towards continuously scaling and supporting snoopy protocols, which were initially designed for bus-based systems, is that they enable direct cache-to-cache transfers and thus do not incur directory indirection for cache misses. For workloads that have fine-grain sharing, direct cache-to-cache transfers provide a huge advantage over going to an ordering point and suffering indirection. If the directory access misses on-chip and has to go off-chip, this problem is exacerbated.

The interconnect fabrics for future CMPs face different constraints than prior SMP systems. Future CMPs will most likely employ packet-switched on-chip interconnects. Implementing coherence on such unordered fabrics not only poses significant challenges, but the new on-chip substrate also offers new opportunities. With the abundance of on-chip bandwidth on point-to-point interconnects, meeting the broadcast bandwidth requirement of snoopy protocols may no longer be impossible, especially in the face of recent proposals [15] that reduce the cost of multicast traffic in on-chip networks, making broadcasting more scalable. However, providing a total order to snoop requests on unordered interconnects still remains a challenge (Section 5 surveys prior work that tackles this problem).

In this paper, we target the above problem and propose

[1]There are many different interpretations of snoopy protocols. We use snoopy to imply a broadcast protocol in which requests are sent directly to other nodes in the system, without having to go to an ordering point. Other nodes in the system "snoop" to determine whether the request is meant for them and act accordingly.



(a) Ordering-point protocol

(b) INSO

Figure 1. Global ordering for snoopy coherence

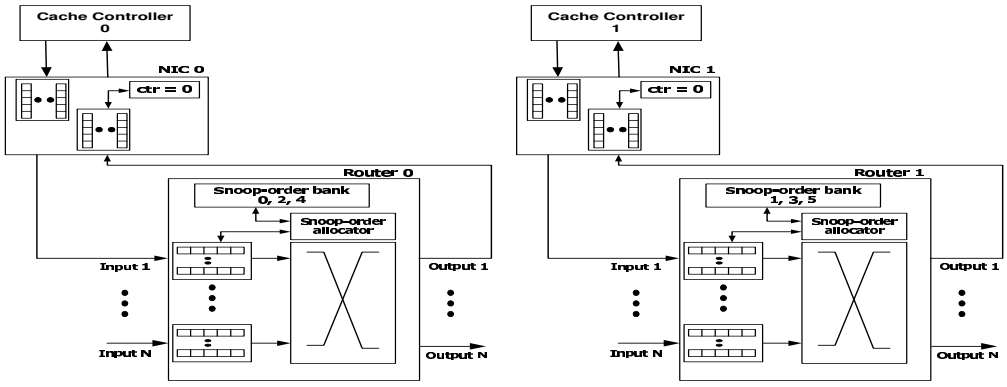
In-Network Snoop Ordering (INSO), which enables *scalable snoopy* coherence on *unordered* on-chip interconnects. INSO alleviates the burden on the coherence substrate (processors, caches, memory, coherence controllers) for ordering requests. Each coherence controller has the illusion that it is attached to a "logical bus" address network that delivers snoop requests in a totally ordered fashion. For every request message from a cache controller, the injection router assigns it a *snoop-order*, which is a globally-ordered *id*, before broadcasting the request to all nodes in the system. The snoop requests are ordered partially in the network and partially at the destination network interface, and delivered to the cache and memory controllers in an ordered fashion. INSO is essentially a distributed mechanism of ordering snoop requests. It enables direct cache-to-cache transfers because cache requests do not incur directory indirection. It also does not require the expensive directory storage structure. Full-system evaluations show that INSO outperforms the baseline directory protocol by up to 30% (average 19%) on a broad range of benchmarks. In short, INSO offers a scalable way to achieve cache coherence on future many-core systems.

The rest of the paper is organized as follows. Section 2 provides an overview of INSO. Section 3 discusses various techniques employed to make INSO practical, and delves into the microarchitectural implementation details. Section 4 discusses the evaluation methodology and presents quantitative results. Section 5 contrasts INSO with prior related work and Section 6 concludes.

2 In-Network Snoop Ordering: Overview

It has been previously shown that snoopy protocols depend on the logical order and not the physical time at which requests are processed [7, 20], i.e., the physical time at which a snoop request arrives at nodes is not important, as long as the global order in which all nodes in the system observe a particular request remains the same. There is a class of cache coherence protocols in which global order of requests is created through the use of an ordering point. Figure 1(a) illustrates how an ordering-point protocol works. All messages first go to an ordering point, which then broadcasts the messages to all nodes in the system. The interconnection network has to ensure that the order in which requests leave the ordering point is the same as the order in which nodes observe the requests. A straightforward example of such an ordered interconnect is a bus. In a simple bus, all requests first arbitrate for the bus and on winning the arbitration, the request is broadcasted on the bus. The order in which requests win arbitration is the order in which nodes see the requests. Thus, a bus provides an implicit global ordering among requests. Global ordering has also been achieved previously by employing an ordered broadcast tree [10]. The tree interconnect uses a two-level hierarchy of switches to form a pipelined broadcast tree. The single root switch orders requests in the order in which they leave the switch and in-order delivery between switches at different hierarchy levels of the tree interconnect ensures all nodes observe messages in a fixed global order. A ring interconnect has also been previously proposed [22] as an ordering-point interconnect. All requests first go to a single ordering point in the ring and are then forwarded to other nodes. The requests then travel across the entire ring one behind the other and are thus seen by all nodes in the same order. As the node count increases, the latency cost of indirection to the ordering point increases; requests have to travel more and more hops to reach the ordering point and then get forwarded to other nodes. Going to a single ordering point also creates congestion at the ordering point and degrades network performance.

In an ideal scenario, requests should get ordered at the source itself and go directly to nodes, without having to go to any ordering point. This is the central intuition behind our INSO proposal: if requests can be ordered right at the network router attached to the requesting node, the ordering indirection can be significantly reduced. To achieve this aim, we need to arbitrate requests in a distributed manner, yet ensure a global order. Figure 1(b) presents a high-level overview of how such an ordering is obtained. On a cache miss, snoop requests are inserted into the network by cache controllers. The interconnect assigns distinct, but globally-ordered numbers (e.g., 0, 1, 2, ..) to the snoop requests. This number is called a *snoop-order*. Snoop-orders are initially distributed to each router in the interconnect in a round-robin fashion, such that no two routers have the same snoop-order. In a 64-node network, for instance, Router 0 gets snoop-orders 0, 64, 128, .., while Router 1 gets snoop-orders 1, 65, 129, .., and so on. The router attached to the requesting node then assigns the lowest snoop-order to each request injected into the network, before broadcasting them to all nodes in the network. Periodically, if a snoop-order is not consumed (as no requests are injected at this router), the router will let this snoop-order expire through a broadcast. At every node, a request is only released from the network interface (NIC) to the cache/memory controllers if its snoop-order is one higher than the last snoop-order received at that controller, initialized to 0 at OS bootup time. Essentially, this ensures that all nodes observe requests in a global order. Here, the order is dictated by how snoop-orders are initially



(a) Core 0 - Router 0

(b) Core 1 - Router 1

Figure 2. INSO modules

distributed across routers and how they are allowed to expire.

2.1 Walkthrough example

We will next walk through INSO in detail. The baseline system for our evaluations is a 64-tile CMP architecture with each tile consisting of a processor core, a private L1 cache, and a private L2 cache attached to a router. Eight memory controllers are placed along the edges of the chip, and connected to routers as well[2], which are part of a 64-node packet-switched mesh network. Cache coherence is maintained between the private L2 caches and the memory. The major modules involved in INSO are shown in Figure 2. All memory and L2 cache controllers attach to an interconnect router via an NIC. We next walk through how two requests A and B (shown in Figure 1(b)) are handled and ordered by INSO. INSO behaves similarly for read and write requests.

1. Core 0's L2 cache miss triggers a request, Req A, to be sent to its cache coherence controller, which leads to Req A being injected into the network via the NIC. All snoop-requests are broadcast in nature and need to be delivered to all nodes in a global order. The NIC takes every snoop request from the cache controller, encapsulates it into a single-flit packet and injects the packet into the attached router.
2. Each router has a *snoop-order bank* that contains a set of snoop-orders that the router can assign to requests. The snoop-orders present in a particular router's bank are distinct from any other snoop-order in the system (In the example in Figure 2, Router 0 has even snoop-orders while Router 1 has those that are odd.) The snoop-order bank is a RAM containing a set of numbers that are initialized at OS bootup time. Each router has a *snoop-order allocator* that picks the lowest non-assigned snoop-order from the snoop-order bank and assigns it to every injected request. For instance, here, the allocator looks up its snoop-order bank and assigns snoop-order 0 to Req A. Req A is then broadcasted to all nodes in the system.
3. Now, Core 1's L2 cache miss triggers a request, Req B, that is injected into the NIC and next into Router 1. Since, the lowest snoop-order in Router 1's bank is 1, it assigns Req B a snoop-order 1. Req B is then broadcasted into the system.
4. As Req A and Req B make their way through the network routers towards various nodes, there might be common paths where the requests meet. The routers along these shared paths prioritize lower snoop-orders over higher ones and order Req A ahead of Req B whenever they collide. In this way, the network partially orders the snoop-requests for various destinations.
5. When Req A reaches the destination (since snoop requests are broadcasts, the destinations are all nodes), the router forwards the request to the NIC. All NICs in the system have a *snoop-order counter* which is initialized to 0 at OS bootup time. On receiving a snoop-request, it compares the snoop-order of the request to the counter and if a match occurs, it forwards the request to the cache controller. The counter is also incremented in the process. Since Req A's snoop-order is 0, it results in a match with the snoop-order counter and is immediately forwarded to the cache controller. The value of the snoop-order counter is now incremented to 1.
6. When Req B reaches the destination, the NIC compares its snoop-order (i.e., 1) to its snoop-order counter. If the NIC is one which has already seen Req A, the counter value would be 1 and Req B is immediately forwarded to the cache controller, while incrementing the value of the counter to 2. If the NIC has not seen Req A so far, Req B is buffered at the NIC buffers, until this NIC receives Req A, at which time the snoop-order counter matches that of Req A (i.e., 0) and Req A is forwarded to the cache controller. The value of the counter will then be 1 which matches that of Req B's snoop-order, allowing Req B to be released to the cache controller and value of the counter to be incremented to 2.

The basic job of the network routers and NICs is to ensure that Req A is delivered to all cache and memory controllers before Req B, because Req A has a lower snoop-order than Req B. This ensures that requests are processed by destination nodes while respecting the snoop-order, enabling a total ordering of requests in the system, very much similar to that in ordering-point implementations. INSO does not demand changes to the asynchronous legacy snoopy cache coherence controllers, as it provides an illusion of an ordered network to them. It is also independent of network topology and adds little overhead to state-of-the-art on-chip routers (see Section 3.5).

Since INSO provides a global order of all requests, it supports the strictest consistency model – sequential consistency.

[2]The routers that are connected to both cache and memory controllers have two injection and ejection ports with associated NICs.

For more relaxed consistency models that do not require such ordering of requests, INSO can be tweaked for better performance. Without loss of generality, for the rest of the paper, we will assume sequential consistency, since it poses the hardest ordering requirements to the cache coherence protocol. We will discuss in Section 3.6 how INSO can be altered to incorporate performance enhancement techniques for relaxed consistency models.

2.2 Properties of INSO that ensure correct global ordering

The following invariant properties of INSO allow such a distributed scheme to ensure correct global ordering:

1. *All requests have distinct snoop-orders:* Each snoop-order only resides at one router in the entire chip and can only be assigned to a single request, thus ensuring a monotonic order across all requests.
2. *A snoop-order counter is incremented only when the snoop-order it is waiting for is broadcasted and received at the node:* This ensures that the cache and memory controllers only see requests in the order of snoop-orders.
3. *Unused snoop-orders expire:* This ensures forward progress.

3 Making INSO Practical

Although the scheme described in the previous section is semantically correct, there are some implementation issues that need to be addressed to make it practically viable. We tackle these issues one-by-one in the following subsections.

3.1 Finite snoop-orders

Snoop-orders are appended to request flits at source routers. A flit in an on-chip environment is of the order of 8-16 bytes [25, 31, 33]. A request message contains the address of the requested block along with some information to describe the kind of request. This is usually not large and leaves bytes of space for the encoding of snoop-orders. This, however, is not infinite. On a system's forward progress, snoop-orders need to be wrapped around. For instance, in a system with 32 snoop-orders, and 16 routers, if a particular router's snoop-order bank contains snoop-orders 0, 16, when the router has serviced two requests, and assigned them snoop-orders 0 and 16, the third request that comes along is assigned snoop-order 0, thus reusing 0. Having two requests with the same snoop-order can violate the global ordering ensured by INSO. To avoid this, the second use of snoop-order 0 has to be ordered after the first instance. We achieve this by designing the network to guarantee point-to-point ordering for the message class in which the requests travel, i.e., messages sent from node A to B are always received at B in the order they were sent from A. Figure 3(a) illustrates how we use point-to-point ordering to ensure that snoop-order 0s that are assigned later do not overtake earlier ones in the network[3]. Req 0_1 stands for the request that was assigned the first snoop-order 0, 0_2 for the second, and so on. Router X has requests 0_2 and 0_3 competing for the output link. It always ensures that 0_2 leaves the router first and then 0_3. 0_1 is already at Router Y. Since 0_1, 0_2 and 0_3 reach Router X in that order, it can simply use a virtual channel (VC) and switch allocator (e.g., queuing arbiters [11]) in which requests that arrive first at a particular input port always leave first. Figure 3(b) shows the router microarchitecture that handles the proposed point-to-point ordering. Since Req 0_2 arrives first, it places its VC request first. In case Req 0_3 arrives before Req 0_3 has been granted an output VC, Req 0_3's VC request is queued behind Req 0_2's request. The VC allocator services the requests in the order in which they are queued. This guarantees that Req 0_2 always wins the VC first and hence places the switch request first. The switch allocator similarly services requests in a queued fashion and hence guarantees that the requests leave the router in the order in which they arrived. This ensures point-to-point ordering of messages. This property is only applied to the message class in which requests travel, so that other message classes can have an allocation policy of their choice and are not limited to this policy. Since this is done at every router, the second use of snoop-order 0 is always received at a node later than its first use. Each router thus distinguishes, and orders, snoop-orders based on the arrival order. By not stalling on the wrapping around of snoop-orders, the finiteness of snoop-orders does not affect performance.

Determining the number of snoop-orders, N, in the system. With reuse of snoop-orders as detailed above, having just one snoop-order per router ensures semantic correctness. However, this creates a static priority in the system where priority is always given to requests originating from nodes with lower snoop-orders. On the other hand, all requests from the same node are ordered in the order in which they are injected into the network. Even when requests with higher snoop-orders arrive at the destinations earlier, they would have to always wait for lower snoop-orders from the same node. Hence, to ensure fairness in the system (say, with 64 routers), we distribute the snoop-orders such that *Router* 0 has snoop-orders 0, 127, .., and *Router* 1 has snoop-orders 1, 126, .., and *Router* 63 has 63, 64, .., and so on. What this does is that for 64 rounds of snoop-orders, every router has the same average priority. Thus, we require no more than R snoop-orders in every router, i.e., total number of snoop-orders in the entire network, $N = R^2$, where R is the number of routers in the interconnect.

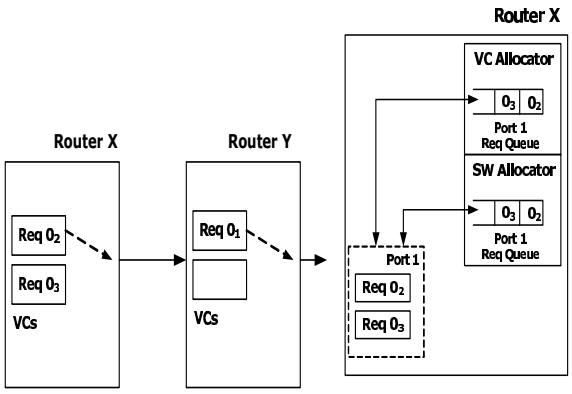
INSO and adaptive routing. In order to enable finite snoop-orders in the system, INSO relies on employing point-to-point ordering for the message class in which requests travel. Apart from point-to-point ordering, deterministic routing in the network is also required to guarantee that requests with the same snoop-order do not take different routes. Thus, in its current form, INSO does not allow adaptive routing in the message class in which requests travel. Adaptive routing can be employed for other message classes and INSO does not restrict that.

3.2 Deadlock avoidance

Prioritizing allocation for lower snoop-ordered packets is not enough to guarantee that the lowest snoop-ordered request always makes forward progress towards the destination. In classic on-chip networks and NICs, packets can be stalled when they are unable to obtain a VC or buffer. Figure 4(a) shows a scenario in which packets with higher numbered snoop-orders hold onto VCs/buffers at the NIC as well as intermediate routers. The NIC, however, waits for the request with snoop-order 0, which is blocked at an intermediate router as it is unable to get hold of a buffer or VC. In such a scenario, the system is deadlocked.

To address the above problem, we reserve a VC and buffer at every router and NIC for the lowest snoop-order not seen so

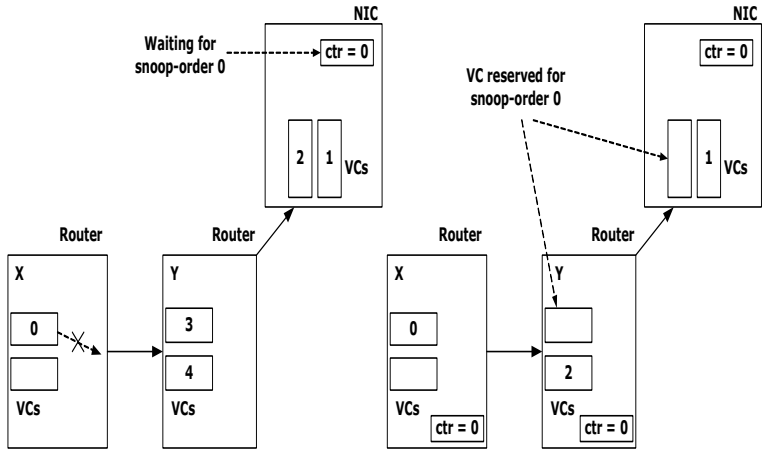
[3]Note that point-to-point ordering is also required for certain message classes in Token Coherence [19] and Uncorq [30]



(a) Example

(b) Router microarchitecture

Figure 3. Point-to-point ordering



(a) INSO deadlock

(b) INSO deadlock avoidance

Figure 4. Deadlock scenario and avoidance scheme

far. Each router also maintains a *snoop-order counter*, similar to that of NICs. This ensures that the lowest snoop-ordered requests always proceed towards the target without starvation. Figure 4(b) shows how reserving a VC for the lowest snoop-order guarantees deadlock freedom. One thing to note about the router's snoop-order counter is that it cannot be simply incremented once the packet holding a matching snoop-order passes through the router. This is because higher snoop-orders may have already traversed and left the router. This is unlike the snoop-order counter of an NIC which can be simply incremented when it sees a matching request. Ideally, the router could keep a record of the snoop-orders it has seen so far and then increment the counter accordingly. This bookkeeping would incur high overheads. We get around this problem by exploiting the broadcast nature of the protocol. Every message, which a router receives, goes to the NIC attached to the router, while going out to next-hop routers. Once the request with the lowest snoop-order is ejected from the router, it triggers the NIC to determine the next lowest snoop-order the router has yet to see and update the router's snoop-order counter. We will discuss the microarchitectural details associated with this scheme in Section 3.5. We will also present quantitative results in Section 4.5 to show the performance impact of reserving a VC for deadlock avoidance.

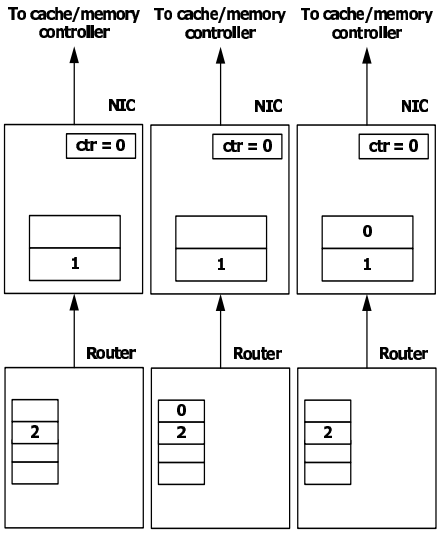
3.3 Finite destination buffering

The scheme described so far requires a huge amount of buffering at the destination NICs, as the request with the lowest snoop-order may not arrive at all destination NICs before the requests holding higher snoop-orders. In this case, the NICs need to buffer the waiting requests, till the lowest snoop-order arrives. Although the network does partial ordering of snoop-orders, in the worst case, the snoop-orders can arrive in a strictly decreasing order of their snoop-orders. If the memory system allows M messages to be outstanding at a time, none of these messages might grab lower snoop-orders and this demands M request buffers at each destination interface. The expiration logic will take care of the system's forward progress by expiring lower snoop-orders. However, providing so much buffering is impractical. We thus need a mechanism to handle finite and practical number of buffers at every destination NIC.

Our solution is as follows. Suppose we are permitted to have B buffers at the destination NICs. The destination routers eject a request from router buffers (typically the input VC buffers of a router) into the NIC buffers, only if the snoop-order of the request is less than snoop-order counter $+B$. Otherwise, the incoming requests remain in the router. The NIC scans these B buffers for the request with the lowest snoop-order so as to forward that onto the cache/memory controller. Once the lowest snoop-order request is forwarded, its corresponding NIC buffer is released, and the NIC snoop-order counter is incremented. A simple example to illustrate this is shown in Figure 5. Suppose the number of NIC buffers, B, is 2 and the snoop-order counter at the NIC is 0. The NIC thus would allow only snoop-orders 0 and 1 to be ejected from the router into the NIC buffers. The figure shows how a request with snoop-order 2 is buffered in the router to allow a request with snoop-order 0 to proceed. Thus, INSO handles finite destination buffering by buffering higher snoop-ordered packets in the network itself, which triggers backpressure through the network to throttle injection nodes. As the network routers prioritize lower snoop-ordered packets, this allows such packets to overtake higher snoop-order packets and help alleviate buffer pressure at the destination interfaces, easing impact on network throughput.

3.4 Snoop-order expiration

Our INSO scheme is a simple and straightforward policy of generating and assigning snoop-orders in a round-robin fashion to the routers' snoop-order banks. It assigns snoop-orders in a fair manner to all nodes and if the request stream from all nodes is at approximately the same rate, INSO works well. A scenario may, however, arise where snoop-orders with lower numbers are not assigned frequently to requests, because the routers that have lower numbered snoop-orders do not receive requests. This would create a situation in which requests with higher numbered snoop-orders keep on waiting for the lower snoop-orders that get assigned very infrequently. To tackle this problem, we employ a snoop-order expiration scheme. All routers check at regular intervals (called *expiration window*, W) how many snoop-orders they have assigned in the last interval (say, C). If this is below a threshold (T), the routers broadcast the lowest $T - C$ snoop-orders as *expiration messages*, so that nodes waiting on them can proceed. All NICs



(a) 2 at router (b) 0 arrives at router (c) 0 goes to NIC

Figure 5. Finite buffering

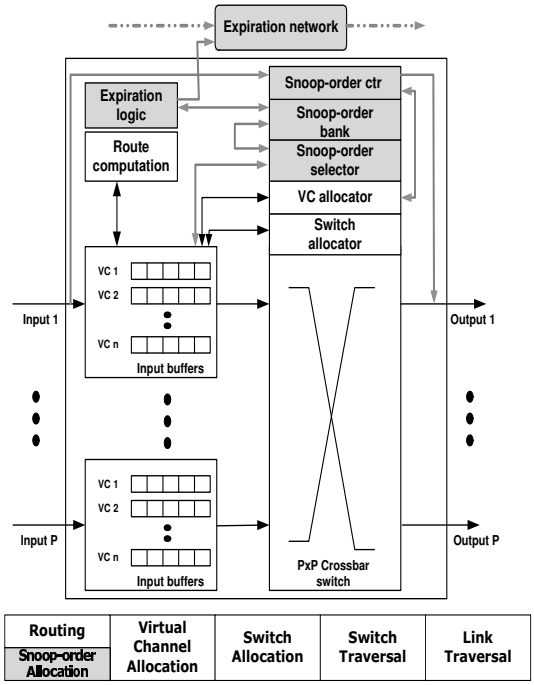


Figure 6. Router microarchitecture and pipeline

Pseudo code: Expiration logic at routers

Initialize: *current time* = 1, *timeout window* = *W*
Initialize: *Threshold* = *T*
1: AT EVERY CLOCK CYCLE:
2: *current time* = *current time* + 1
3: **if** *current time* == *W*
4: *C* = *snoop-orders assigned in last window*
5: *Expire* (*T* – *C*) *snoop-orders*
6: *current time* = 1
7: **end if**

Figure 7. Expiration logic

increment their snoop-order counters upon receiving the expiration messages. If expiration snoop-orders are sent as separate flits, a higher T would consume higher interconnect bandwidth. However, since the snoop-order distribution is static, just encoding the lowest snoop-order and the number of snoop-orders into a single flit would be sufficient to indicate the expired snoop-orders.

The values of W and T determine how frequently and how many times expirations take place. Let us consider a scenario in which, in the last expiration window, the maximum number of requests that can be injected into any router (say, Router A) in the system from its local NIC was X and there was a router (say, Router B) that did not receive any request from its NIC. Let us assume that all the snoop-orders that Router A assigned to its requests were higher in number than the snoop-orders present in Router B. In such a case, Router A's requests reach their destinations and wait for Router B's snoop-orders to expire. Ideally, Router B should expire all the lower X snoop-orders at once which requires $T = X$. It would also be ideal to have expiration window W as small as possible so that the waiting time for unused snoop-orders is not large. However, having a smaller W and higher T would lead to more frequent expirations and might consume extra interconnect bandwidth. Thus, an intermediate scenario is desirable where the performance benefits of expiring many snoop-orders does not lead to huge bandwidth overheads. We describe the snoop-order expiration logic further in Section 3.5.

3.5 Router microarchitecture

Figure 6 shows the proposed INSO router microarchitecture and pipeline (newly added portions are highlighted in grey). The difference in the router microarchitecture that implements INSO versus a typical interconnection network router is the snoop-order bank, snoop-order selector, snoop-order counter and expiration handling logic. The VC and switch allocators are modified to provide point-to-point ordering for the request message class.

Router pipeline and snoop-order bank. In a state-of-the-art on-chip router [11], the header flit of a packet goes through the *Routing* stage to determine the output port for the packet. The header then arbitrates for a output VC in the *VC Allocation* stage. Upon successful allocation of a VC, the header proceeds to the *Switch Allocation* stage where it arbitrates for the switch input and output port. On winning the switch, the header moves onto the *Switch Traversal* stage in which it traverses the crossbar. This is followed by *Link Traversal* in which the header traverses the link to the next router. Subsequent body and tail flits simply follow the route and VC that are reserved by the head flit.

In our proposed INSO router, a new pipeline stage is added – for assigning snoop-orders to snoop requests. Snoop requests that have not been assigned snoop-orders go through *Snoop-order Allocation* to obtain a snoop-order. The snoop-order bank is implemented as a register that contains the lowest snoop-order to be assigned next, filled from a RAM that contains the snoop-orders allocated to the router. During the Snoop-order Allocation stage, a request is sent to the snoop-order selector, which simply looks up the snoop-order bank register and returns the value. Since this Snoop-order Allocation stage is just a simple register lookup, it can occur in parallel with the Routing stage. The scanning for the lowest snoop-order in the bank and updating of the register can be done off the critical path. The header flit goes through snoop-order allocation in parallel with the routing stage. Body and tail flits experience no change to the router pipeline. After successful snoop-order allocation, the requests become normal broadcast packets and do not go through this stage again. Thus, snoop-order allocation occurs only once per snoop request in the entire network. The snoop-order bank contains R snoop-orders, where R is the number of routers in the on-chip network. For a 64-node network, each snoop-order bank contains only 64

snoop-orders, each requiring log(4096) = 12 bits. Since, the number is so low, the hardware overhead of the snoop-order bank is not significant.

Expiration logic. The pseudo-code for the expiration logic is shown in Figure 7. Every router has a timeout window (W), at which the expiration logic checks whether the router has used up T snoop-orders in the last expiration interval. If the router has used up C snoop-orders which is smaller than T, the router generates a broadcast expiration flit that encodes the lowest snoop-order present in its bank and $T - C$. In our simulations, we assume that the expirations go through a separate network consisting of a contention-less single-cycle router pipeline. We next justify our assumption of a single-cycle router for the expiration network. For a 64-node system, an expiration flit consists of a snoop-order, which is 12 bits, and $T - C$, which is $\log(T - C)$ bits. We will show later that having a T of 1-4 is sufficient for INSO to perform well. Thus, $T - C$ can be encoded in two bits implying an expiration flit of 14 bits. The maximum traffic injection load for expiration network is known (one 14-bit flit every W cycles) and thus the buffers of the routers can be sized to avoid running at saturation. A 14-bit crossbar will have a timing path that is much faster than the regular 128-bit crossbar (because of quadratically shorter interconnects and drivers). Thus the *Crossbar* and *Link Traversal* stages can potentially fit within a single cycle. Since every packet is a broadcast, *VC Allocation* and *Switch Allocation* can be done off the critical path every cycle. Apart from assuming a single-cycle router, we also do not model contention in the expiration network. The expiration network can have up to N expirations every W cycles. Since all expirations are broadcast in nature, each NIC can potentially receive N flits in W cycles. We did not model this contention and are currently working on realistic modeling of the expiration network and investigating the sensitivity of INSO's performance to W.

Snoop-order counter update. As mentioned earlier, the snoop-order counter inside a router cannot simply increment its value on the departure of the lowest snoop-order. This is because other higher snoop-orders might have already traveled through the router. Routers are also not aware of snoop-order expirations and thus need to somehow know what is the minimum snoop-order for which they should reserve a VC. On the departure of the snoop-order for which the router was reserving resources, it obtains the next lowest outstanding snoop-order, which the router has not seen so far, from the NIC. Due to the broadcast nature of the system, all messages that traverse a particular router always reach its local NIC. The NIC thus has up-to-date information about what the router has seen. It responds to the router's request by informing it about the next snoop-order the router should reserve resources for. This information can be piggybacked with the usual credit signals between NICs and routers.

Ideal multicasting. A multicast message contains a destination set that comprises multiple nodes. Tree-based multicast messages move along a common path and branch off into multiple multicasts when different destinations require traversal through different paths. We assume ideal hardware tree-based multicast support in our routers, i.e., that each multicast message loads at most one flit across every link in the network. Implementing practical hardware multicast support in on-chip routers has been discussed in VCTM [15]. VCTM achieves the ideal hardware tree-based multicast network we assume with 64 trees, one per node, each tree with all 64 destinations. As shown in that paper, this leads to feasible overheads.

3.6 Interaction of INSO with the memory consistency model

Throughout the paper, we have assumed sequential consistency as the memory consistency model. This is the strictest consistency model and provides the toughest constraints on the cache coherence protocol. It requires a global ordering of all requests in the system and INSO achieves that. However, a cache coherence protocol is not always required to provide such support. In [3], Adve et al. discussed various memory consistency models and their requirements. A cache coherence protocol is essentially a mechanism that propagates a newly written value to the cached copies. A memory consistency model can be interpreted as the policy that places an early and late bound on when a value can be propagated to a processor. The two main aims of a coherence protocol are: (1) to ensure that writes are eventually seen by all processors, and (2) writes to the same location are serialized. If the memory consistency model poses just these requirements on INSO, then it can simply maintain *per-cache-line* snoop-orders. Requests from different cache lines would then not impact the ordering wait time of each other. For consistency models, such as the IBM PowerPC [23], that do not require atomicity of writes (a write request does not need to wait for all sharers in the system to be invalidated before it can assume the write is complete), INSO can be further relaxed. For a sequentially consistent memory model, on a write request, the cache controllers in INSO wait for the request to return to itself to ensure global order. Note that the request waits for its global order in the NIC and the cache controller sees a request only after it is ordered. Even if the cache controller receives the data before that, it waits for the ordering of the request. This is because the cache controller needs to be sure that all cache copies have been invalidated. Receipt of its own request in a global order guarantees that. If atomicity of writes is not a requirement, the cache controller can be relieved from this wait. In short, we believe that INSO is a generic technique that can support all memory consistency models and is not limited to sequential consistency.

4 Evaluation

For all our evaluations, we do full-system simulation using Virtutech Simics [32] extended with the GEMS [21] tool set. The GARNET [4] network model was used to capture the detailed aspects of the interconnection network. GARNET is a cycle-accurate interconnect model that models a detail packet-switched [13] router pipeline including VCs [12], buffers, switches and allocators. GEMS, along with GARNET, provides a detailed memory system timing model.

4.1 Target system

We simulate a tiled 64-core CMP system with parameters given in Table 1. Each tile consists of a two-issue in-order SPARC processor with 32 KB L1 I&D caches. It also includes a 1 MB private L2 cache. DRAM was attached to the CMP via eight memory controllers along the edges. The DRAM access latency is modeled as 275 cycles. The on-chip network was chosen to be an 8×8 mesh consisting of 16-byte links with deterministic dimension-ordered XY routing. Each input port contains eight VCs and four buffers per VC. We simulate a 4-cycle router pipeline with a 1-cycle link. The on-chip routers are assumed to have perfect hardware multicast support. The INSO specific parameters are also given in Table 1.

We did some state-space exploration and then chose an expiration window (W) of 20 cycles and threshold (T) of 3 to strike a balance between performance improvement and bandwidth penalty. As described earlier, we assume the expiration flits go through a separate network consisting of a contention-less single-cycle router pipeline.

Table 1. Simulation parameters

Processors	64 in-order 2-way SPARC cores
L1 Caches	Split I&D, 32 KB 4-way set associative, 2 cycle access time, 64-byte line
L2 Caches	1 MB per core, 10 cycle access time, 64-byte line
Directory Caches	1 MB per memory controller, 8 MB total on chip
Memory	8 memory controllers, 275-cycle DRAM access + on-chip delay
On-chip Network	8×8 2D Mesh, 16-byte links, 4 cycle router pipeline, 8 virtual channels, 4 buffers per virtual channel
INSO Parameters	
Number of Snoop-orders (N)	64^2 = 4096
Expiration Timeout Window (W)	20 cycles
Threshold Snoop-orders (T)	3

4.2 Workloads

We ran SPLASH-2 [35] and PARSEC [6] application suites on the above-mentioned configuration. SPLASH-2 is a suite of scientific multithreaded applications that has been used in academic evaluations for the past two decades. PARSEC is a recent benchmark suite that focuses on emerging parallel workloads. Thus, we chose to evaluate INSO on both SPLASH-2 and PARSEC to explore the impact of their diverse characteristics on INSO. We ran the parallel portion of each workload to completion for each configuration. All benchmarks were warmed up and checkpointed to avoid cold-start effects, and we ensured that caches were warm by restoring the cache contents captured as part of our checkpoint creation process. To address the variability in parallel workloads, we simulated each design point multiple times with small, pseudo-random perturbations of request latencies to cause alternative paths to be taken in each run [5]. We averaged the results of the runs.

4.3 Coherence protocols

We now discuss the implementation details of the cache coherence protocols simulated. We compared our INSO proposal to a directory protocol and a broadcast based Token Coherence protocol. All protocols were part of the GEMS distribution.

INSO. We evaluated our INSO network with a broadcast snoopy protocol. The protocol supports the MOSI states. On a private L2 cache miss, the snoop request is broadcasted into the network assuming the network orders the requests like an ordering-point protocol. The memory determines if it should respond by maintaining per-block state. Sorin et al. [28] presented a detailed specification of the broadcast protocol that we evaluated. An ideal INSO scheme would be one in which each snoop request, on entering the network, would be assigned the lowest available snoop-order in the system, so that the request's wait time for getting ordered upon arriving at the destination would be minimal. There would be no expirations in such a scheme as all snoop-orders would always get used. We call this scheme **INSO-Oracle**, and model it as a single, global snoop-order bank that all routers can magically look into at no overhead. This is the ideal that INSO can reach and we evaluate this scheme to estimate the potential of INSO.

Directory protocol. We evaluated a directory protocol that supports the MOESI coherence states. In the directory protocol, when a memory access is not satisfied within a tile (private L1/L2), it seeks the directory for the sharing information of the cache block. We evaluated an implementation in which the directory information is stored in DRAM. For a private L2 configuration like ours, the entire directory state cannot be fully accommodated within the on-chip private L2 tags. A bit-vector is stored for every memory block to indicate the sharers. We implemented on-chip directory caches distributed along with the memory controllers. On-chip directory cache misses lead to off-chip directory access with high delays. To minimize this and study the actual effect of only on-chip indirection, we evaluated a 1 MB directory cache at each memory controller, totaling a generous 8 MB of on-chip capacity.

Token Coherence. We evaluated a broadcast based Token Coherence protocol (TokenB) [19] that tries to achieve higher performance by directly broadcasting requests to all nodes in the system[4]. Token Coherence is a protocol that resolves protocol races without an ordering-point indirection by decoupling coherence into a correctness substrate and a performance protocol. The correctness substrate guarantees correct transfer and access to blocks by tracking tokens, and prevents starvation using persistent requests. In the absence of a data race, the requester gets all tokens it desires along with the data, and direct cache-to-cache transfers are made possible. Token Coherence represents a protocol that does not incur an ordering-point indirection in the common case and thus is a strong competitor to our INSO proposal. The TokenB protocol that was evaluated supported the MOESI coherence states. To ensure a fair comparison, we evaluated TokenB on the same ideal broadcast interconnect as used in INSO.

4.4 Evaluation results

We next present the evaluation results for comparing INSO against the above-mentioned protocols.

Normalized runtimes. Figure 8 gives the benchmark runtimes (smaller is better) for all protocols normalized to the directory protocol. INSO consistently performs better than the directory protocol over the range of workloads. INSO improves the runtime over the directory protocol by 19% on average (maximum of 30%). INSO-Oracle improves the runtime over the directory protocol by 25% on average (maximum of 32%). This speedup in runtime is primarily due to the avoidance of ordering-point indirection in INSO. We measured the average snoop-order expiration wait time for INSO, which is the average number of cycles a request has to wait at the NIC to get ordered. In INSO-Oracle, a request magically gets the lowest outstanding snoop-order in the system and is broadcasted. While active requests are being broadcasted, they might reach different destinations in different order. Thus, the wait time for requests would be the time they wait in the NIC for requests that have grabbed lower numbered snoop-orders and hence is

[4]We could not get TokenB to run PARSEC for our 64-core configuration. A situation was arising in which sharers were replacing their shared lines and passing their tokens to the memory. This led to a situation in which memory eventually had all the tokens except the owner. Since memory is not the owner, it could not supply tokens to requests. This should have led to persistent requests eventually resolving the starvation. Our runs were, however, deadlocking.

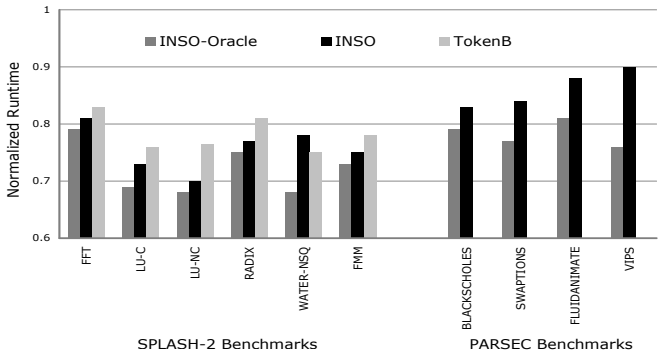


Figure 8. Runtime for various protocols normalized against directory protocol

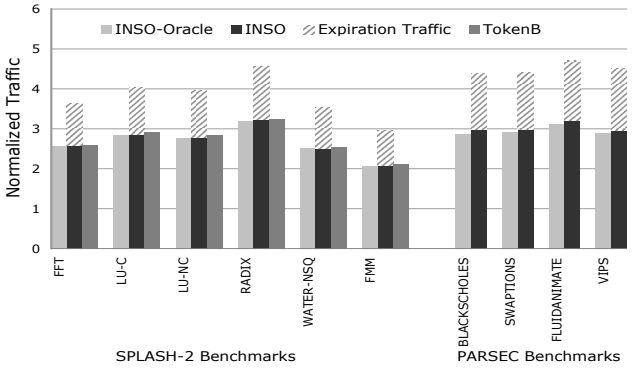


Figure 9. Interconnect traffic for various protocols normalized against directory protocol

a true ordering wait. For INSO, requests would also have to wait for lower numbered snoop-orders that were not grabbed and expire. For the benchmarks evaluated, the average ordering wait time for INSO-Oracle was 0.89 cycles. For INSO, this number was 11.53 cycles for SPLASH-2 benchmarks and 23.45 cycles for PARSEC benchmarks. This explains why INSO performs better than the directory protocol by a smaller margin on PARSEC as compared to SPLASH-2.

INSO also performs on an average 3.5% faster (maximum of 8.5%) than TokenB for the SPLASH-2 benchmarks. INSO-Oracle outperforms TokenB by 8% on average (maximum of 11.2%). INSO wins over TokenB in cases when there are data races, when TokenB relies on retries and expensive persistent requests to guarantee that requests are not starved. This includes going to an ordering point that orders competing requests one by one. INSO does not incur any such indirection. The reason why INSO does not outperform TokenB by much is because the SPLASH-2 benchmarks do not incur significant data races. The percentage of persistent requests (percentage of requests that had to finally rely on persistent requests to get tokens) for our experiments was 1.02% on average. This relatively low percentage of persistent requests was expected for SPLASH-2 benchmarks as they do not have fine-grained parallelism and thus competing writes (that cause data races) among same cache lines are infrequent. In one specific case (WATER-NSQ), INSO performs 4% worse than TokenB. The average ordering wait time for INSO for this benchmark was 25.2 cycles. Also, the percentage of persistent requests with TokenB for this benchmark was 0.97%. The higher ordering wait time for INSO and the lower percentage of persistent requests for TokenB seem to be the reason for INSO's inferior performance on this benchmark.

Network traffic. Figure 9 shows the normalized traffic (smaller is better) per interconnect link for all protocols normalized to the directory protocol. While INSO improves system performance, it puts additional bandwidth requirement on the interconnect. This is because of the broadcast nature of INSO. Directory protocols have the least interconnect traffic since most of the messages are unicast in nature. In a directory protocol, the requester sends a request to the directory which responds if it has valid permissions. Otherwise, it forwards the request to the exact set of sharers. Thus, no message that is traveling in the network is redundant. INSO-Oracle presents the least amount of interconnect bandwidth that a broadcast protocol would require. This is because it involves a source request broadcast followed by the data response. TokenB's bandwidth requirement is a little more than INSO-Oracle because of retries, explicit acknowledgments and persistent requests. INSO has additional snoop-order expirations that contribute to interconnect traffic. The shaded bar in Figure 9 shows the bandwidth on the expiration network in INSO. Since this additional traffic travels on the separate expiration network, it does not affect the normal request and data traffic. However, contention in the expiration network might lead to additional ordering wait time for INSO. We are currently working on evaluating the impact of this additional delay, by modeling the detailed expiration network.

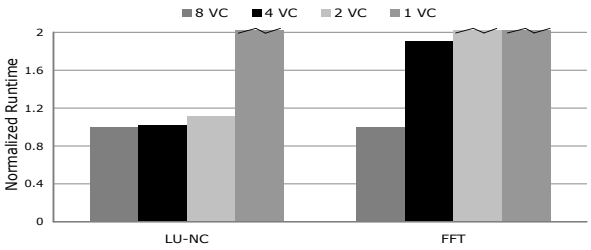


Figure 10. Runtime of INSO with different number of request VCs normalized against 8-VC

4.5 Effect of reserving a VC for the minimum snoop-order

As described in Section 3.2, INSO reserves a VC and a buffer at every router to avoid deadlocks that might occur due to the minimum snoop-order getting backlogged behind higher snoop-orders. To understand its effect, we conducted an experiment in which we retain the simulation configuration as above, gradually reducing the number of VCs from 8 to 1 in the request message class, and study the performance impact. We evaluated the FFT and the LU-NC benchmarks from the SPLASH-2 benchmark suite. Figure 10 shows the benchmark runtimes for the various configurations normalized to the 8-VC configuration. It is evident that as the number of request VCs is reduced, the performance of INSO degrades. For low VC configurations (1 VC for LU-NC and 1, 2 VC for FFT), the simulation seems to run indefinitely. Thus, INSO does not perform well with very low number of request VCs.

5 Related Work

INSO achieves a direct protocol (one with direct cache-to-cache transfers) on unordered interconnects. Our work is thus related to protocols which work towards direct cache-to-cache transfers as well as those that create ordering (partial/full) in unordered networks.

5.1 Protocols with direct cache-to-cache transfers

Token Coherence [19] is a protocol that achieves direct cache-to-cache transfers in the absence of a data race. It relieves the interconnect fabric from any ordering requirement and performs ordering at the protocol level. It decouples coherence into a correctness substrate and a performance protocol. The protocol associates a fixed number of tokens with a cache line. To access a particular line, the core broadcasts a request for tokens for the particular line. If there are no competing requests for the cache line in the system (which is the common case), the requesting core gets the required tokens and data to proceed with the request. In case of a data race (competing request for the same cache line), the correctness protocol is invoked by re-issuing requests, and an ordering point orders the requests and guarantees forward progress. Similar to Token Coherence, INSO also enables direct cache-to-cache transfers. However, unlike Token Coherence, direct transfers always occur, not only in the absence of a data race. INSO relies on distributed ordering to attain direct cache-to-cache transfers. Token Coherence requires per-cache-line tokens which incur substantial storage overheads. The number of tokens is at least the number of cache lines. In contrast, INSO requires R^2 snoop-orders, where R is the number of routers in the system. This is much fewer than the potential cache lines in the system. INSO also does not require changing the asynchronous legacy snoopy cache controllers whereas Token Coherence requires a re-implementation of the whole coherence substrate.

Similar to Token Coherence, Intel's new Quickpath Interconnect (QPI) [18] decouples the ordering of requests from the interconnect and still attains direct cache-to-cache transfers. QPI is a source broadcast protocol in which requesters broadcast requests for data, and in the absence of a data race, the data are sent to the requester while maintaining coherence. All broadcasts also go to per-node directories which detect data races and ask nodes to fall back to a coherent state. Although INSO enables direct cache-to-cache transfers, unlike QPI, it happens for all cache requests. INSO thus does not require a race detection and fall-back mechanism like QPI.

In-network cache coherence [14] also aims at cache-to-cache transfers, through the network routing requests towards nearby caches. However, again, it needs a fall-back mechanism when the network does not have directory information, regressing to the directory home node as the ordering point. It also requires significant per-cache-line storage within the network.

5.2 Ordering on unordered interconnects

Uncorq [30] is an embedded ring coherence protocol in which snoop requests are directly broadcasted, using any network path, to all nodes which reply with data directly to the requester. This achieves direct cache-to-cache transfer. However, along with a snoop request broadcast, a response message is initiated by the requester. This response message traverses the entire logical ring, collecting responses from all nodes. Read requests do not have to wait for the response message, but write requests have to. Unlike Uncorq, INSO does not enforce such a response wait for requests. INSO also does not require embedding of a unidirectional ring on top of the existing network.

Multicast snooping [7] uses totally ordered isotach-like [24] fat-tree networks to create a global ordering of requests. Delta coherence protocols [34] also implement sequential consistency by employing isotach networks to guarantee global ordering of requests. In contrast, we do not restrict our ordering technique to any particular topology. Our design proposes an in-network ordering scheme that can be mapped to any underlying physical network.

Gray Zone [8] implements weak ordering by timestamping requests over the network. The protocol assigns timestamps to every processor request on its creation. Processors communicate with each other to determine the oldest message currently present in the system. Ordinary reads and writes are serviced immediately, while synchronization operations are serviced only after the oldest message has been serviced. This achieves weak ordering by ensuring that all accesses issued before a synchronization operation complete first. A global clock is used to assign timestamps. In contrast, INSO is not restricted to weak consistency and does not rely on "physical" time to provide ordering.

Comparison with Timestamp Snooping. Timestamp Snooping (TS) [20] creates ordering of snoop requests on unordered interconnects by using logical timestamps and reordering requests at the end points. TS is the closest related work that tries to assign logical orders to requests, similar to INSO. We explain two terms from TS that help us compare the two techniques better. Ordering time (OT) is the logical ordering time of a request, similar to snoop-order (SO) in INSO. Guaranteed time (GT) is defined as the logical time that is guaranteed to be less than the OTs of any requests that may be received later by a router/NIC, similar to snoop-order counter in INSO. To clearly contrast the two approaches, we present their pseudo-codes in Figure 11. The differences between INSO and TS are as follows:

1. **Notion of logical order.** INSO and TS differ in their definition of logical order. A snoop-order X in INSO is assigned to a single request and upon receipt of all snoop-orders up to $X-1$ at the destination, the request can be processed. However, an OT of X in TS can be assigned to N requests, where N is the number of nodes in the system. The destination waits for all requests with OT = X and a token to advance GT to $X+1$. It is then that all requests with OT = X are ordered, breaking ties with a function of the source ID numbers. It should be noted that the token to advance GT from X to $X+1$ will always arrive after all requests with OT = X arrive. Thus, even after the request with OT = X from source 0 arrives, it has to wait for all valid requests with the same OT as well as the token to advance GT.
2. **Expiration orders.** INSO relies on explicit expiration of snoop-orders, i.e., an expiration message expires specific snoop-orders. TS employs tokens that are exchanged between routers as well as between routers and NICs. A token advances GTs at routers and NICs. A token that advances GT from X to $X+1$ is essentially an expiration signaling that no other request with OT = X will arrive. By not explicitly sending expirations, TS's expiration bandwidth requirement is lower than that of INSO if expirations are sent at the same frequency. However, the TS paper suggested that expirations be sent every cycle. This is equivalent to having an INSO timeout window (W) of 1, while INSO suggests a less aggressive timeout window. Conceptually, both TS and INSO can use either scheme and the trade-off is essentially between higher expiration bandwidth and shorter wait time at destinations for expired logical orders.

INSO Pseudo-code

1: GTs initialized to 0 at NICs
2: Routers assigned OTs in a round-robin manner
3: Request assigned lowest OT from router
4: Request waits for GT at destination
5: GT updated at destination based on OTs received through normal and expiration messages

(a) INSO

TS Pseudo-code

1: GTs initialized to 0 at NICs
2: Request assigned OT = GT + max-hop logical delay + slack
3: Slack updated at intermediate routers
4: GT updated at routers and destinations via tokens and messages passing by
5: Request waits for GT at destination

(b) TS

Figure 11. Comparison of INSO and TS

3. **Logical order assignment.** INSO assigns logical orders to a request from the local router's snoop-order bank. The snoop-order banks of routers are populated in a round-robin manner. In contrast, a request in TS is assigned a logical order that is at least the source's GT plus the logical time to get from the source to the furthest destination.
4. **Implementation challenges.** INSO addresses implementation challenges, such as finite end-point NIC buffering and router complexity, which TS does not. For a 64-processor configuration that allows eight outstanding requests from each processor, TS requires 512 address buffers at every end-point, which is not practical. INSO can work with any amount of finite end-point buffering. TS also requires updating of slack in messages buffered in the router. This updating requires logic that can read and write all messages queued inside the router. This would require additional ports to the buffer queues so that normal router operations are not interrupted. INSO's routers do not have this complexity.

Quantitative comparison. Since INSO and TS use different schemes for logical ordering, destination processing and expiration, we present a quantitative comparison of the two techniques on a set of scientific benchmarks as well as the LMBENCH [29] microbenchmarks. LMBENCH is a suite of microbenchmarks that stresses the system's bandwidth and thus we chose to evaluate TS and INSO on it. We ran the memory bandwidth benchmarks in LMBENCH [CP (copy), WR (write), RDWR (read-write)] with the data set being 80 KB in size. The simulation setup was the same as in Section 4 except where mentioned here. We implemented TS's detailed network inside the current GEMS framework. We modeled slack processing and token exchange exactly the way described in the TS paper. For a fair comparison, we assumed infinite end-point buffering in both TS and INSO. For INSO, we selected $W = 10$, $T = 1$, and for TS, we selected the initial Slack = 4. We selected a lower W to match the aggressive expiration of TS.

Table 2. Ordering wait time in INSO and TS

	Avg cycles		Max cycles	
	INSO	TS	INSO	TS
FFT	42.10	41.31	2893	279
LU-NC	22.73	47.59	389	242
WATER-NSQ	46.82	42.51	721	258
LM-WR	11.62	41.83	372	206
LM-RDWR	10.95	41.95	375	184
LM-CP	17.39	41.45	413	185

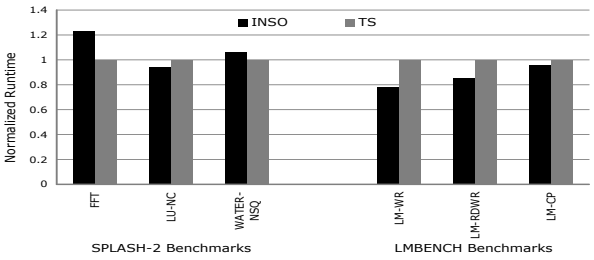


Figure 12. Runtime of TS and INSO normalized against TS

Figure 12 shows the benchmark runtimes (smaller is better) for TS and INSO normalized to TS. The results show that INSO outperforms TS on some occasions while TS is better in some cases. Table 2 shows the average and maximum ordering wait time for the benchmarks. Although the maximum ordering wait time in INSO is always higher, due to the skew in SO assignment and higher W, the average ordering wait time in INSO is less than that in TS for LU-NC and the LMBENCH benchmarks. This explains why INSO outperforms TS for those benchmarks. For FFT and WATER-NSQ, the average and maximum ordering wait time in INSO is higher than that of TS; thus, TS performs better than INSO for these benchmarks. The above experiment shows that the different approaches to providing logical ordering in TS and INSO lead to different behaviors across benchmarks. It should be noted that the expirations in INSO go through a contentionless single-cycle router. Realistic modeling of the expiration network will adversely affect the ordering wait time in INSO. We are currently working on modeling the detailed expiration network.

6 Conclusion

In this paper, we presented INSO, an *in-network snoop ordering* technique that re-engineers the interconnect fabric (without touching the coherence substrate) to enable ordering of snoop requests on underlying packetized on-chip networks that do not provide any inherent ordering properties. Our logical proposal can be overlaid on any unordered interconnect. INSO tags globally-ordered ids, called *snoop-orders*, which determine the global order of the requests. The network orders the request based on snoop-orders and ensures the delivery of requests to nodes in a globally-ordered fashion. Our evaluations show that by saving directory indirection and ordering requests at the source router, INSO improves performance by up to 30% over a directory based protocol. By not having to incur an expensive fallback mechanism, INSO also outperforms TokenB in our experiments. In summary, INSO provides an in-network ordering technique that enables snoopy coherence protocols to scale on unordered interconnects.

Acknowledgments

The authors would like to thank David Wood for his insightful comments and feedback on the work. The authors would also like to thank Milo Martin for the original Timestamp Snooping code. This work was supported in part by NSF (grant no. CNS-0613074), MARCO Gigascale Research Center and SRC (contract no. 2008-HJ-1793).

References

[1] IBM Power6. http://www-128.ibm.com/developerworks/power/library/pa-expert1.html.

[2] Sun Niagara. http://www.sun.com/processors/throughput/.

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

[4] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A Detailed On-chip Network Model inside a Full-system Simulator. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, Apr. 2009.

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

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

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

[8] R. Bisisani, A. Nowatzyk, and M. Ravishankar. Coherent Shared Memory on a Message Passing Machine. In *Proceedings of International Conference on Parallel Processing*, Aug. 1989.

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

[10] A. Charlesworth. The Sun Fireplane System Interconnect. In *Proceedings of ACM/IEEE Conference on Supercomputing*, Nov. 2001.

[11] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Pub., 2003.

[12] W. J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, Mar. 1992.

[13] W. J. Dally and B. Towles. Route Packets, not Wires: On-chip Interconnection Networks. In *Proceedings of Design Automation Conference*, Jun. 2001.

[14] N. Eisley, L.-S. Peh, and L. Shang. In-Network Cache Coherence. In *Proceedings of International Symposium on Microarchitecture*, Dec. 2006.

[15] N. Enright Jerger, L.-S. Peh, and M. Lipasti. Virtual Circuit Tree Multicasting: A Case for On-chip Hardware Multicast Support. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2008.

[16] M. Galles and E. Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In *Proceedings of Annual Hawaii International Conference on System Sciences*, Jan. 1994.

[17] Intel. From a Few Cores to Many: A Tera-scale Computing Research Overview. http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[18] D. Kanter. The Common System Interface: Intel's Future Interconnect. http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032, 2007.

[19] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of International Symposium on Computer Architecture*, Jun. 2003.

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

[21] M. M. K. 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. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[22] M. R. Marty and M. D. Hill. Coherence Ordering for Ring-based Chip Multiprocessors. In *Proceedings of International Symposium on Microarchitecture*, Dec. 2006.

[23] C. May, E. Silha, R. Simpson, H. Warren, and CORPORATE International Business Machines, Inc., editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers Inc., 1994.

[24] P. F. Reynolds, Jr., C. Williams, and R. R. Wagner, Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):337–348, Apr. 1997.

[25] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. *ACM Transactions on Architecture and Code Optimization*, 1(1):62–93, Apr. 2004.

[26] D. J. Schanin. The Design and Development of a Very High Speed System Bus – The Encore Multimax Nanobus. In *Proceedings of ACM Fall Joint Computer Conference*, Nov. 1986.

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

[28] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, 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, Jun. 2002.

[29] C. Staelin and H. P. Laboratories. lmbench: Portable Tools for Performance Analysis. In *Proceedings of USENIX Annual Technical Conference*, Jan. 1996.

[30] K. Strauss, X. Shen, and J. Torellas. Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors. In *Proceedings of International Symposium on Microarchitecture*, Dec. 2007.

[31] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of IEEE International Solid State Circuit Conference*, Feb. 2007.

[32] Virtutech AB. Simics Full System Simulator. http://www.virtutech.com/.

[33] D. Wentzlaff, P. Griffin, H. Hoffman, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, pages 15–31, 2007.

[34] C. Williams, P. Reynolds, Jr., and B. de Supinski. Delta Coherence Protocols. *IEEE Concurrency*, 8(3):23–29, Jul.-Sep. 2000.

[35] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of International Symposium on Computer Architecture*, Jun. 1995.