A universal ordered NoC design platform for shared-memory MPSoC

Woo-Cheol Kwon* Li-Shiuan Peh†
Department of EECS, MIT
Cambridge, MA 02139
Email:*wckwon@csail.mit.edu, †peh@csail.mit.edu

***Abstract*—Shared memory is the predominant programming model in today's MPSoCs. However, existing SoC on-chip communication standards like AMBA relies on the interconnect for ordering. This is a problem as the number of actors increases, as traditional simple interconnects like buses and crossbars do not scale, yet scalable distributed NoCs are inherently unordered. Without built-in ordering capability from NoC, cache coherence protocols have to rely on external ordering points which can forward the requests so that every cache observes the requests in the same order. Such ordering points incur significant scalability issues though, such as indirection latency or communication hotspots in the network.**

In this paper, we propose a universal ordered NoC platform for shared-memory MPSoC designs to provide coherence request ordering in addition to communication. The proposed solution is based on a separate light-weight ordering network to establish the global request order which the receiving NIC leverages for delivering requests. The proposed solution provides a comprehensive support for general network topologies and various levels of memory consistency, while adhering to existing cache coherence protocol standards. The full-system simulation with heterogeneous MPSoC Rodinia benchmarks shows that it reduces the request latency by 37.6% and 35.7% over ordering points in 2D-mesh and butterfly fat tree topologies, respectively. This translates to overall runtime improvements of 17.8% and 12.0% in each topology, for a 36-node and 32-node MPSoC respectively.

I. Introduction

In the past decade, as single-core scaling hits a wall with diminishing performance gains and rising power consumption, we have witnessed a major transition to multi-core chips to continue performance scaling. The trend towards multi-core is now mainstream in system-on-chips(SoCs), with multiprocessor systems-on-chips (MPSoCs) having emerged in widespread use in networking, signal processing, and multimedia chips, from mobile processors like Qualcomm Snapdragon series to Intel Xeon D SoC family and Freescale QorIQ network processors.

While shared memory is the dominant programming model for these MPSoCs due to ease of programming and legacy codebase, it also imposes a major challenge to the designers as MPSoCs scale in complexity. Existing SoC on-chip communication standards such as AMBA 4[5][1], OCP 3.0[1], and HyperTransport[4] require the interconnect to order coherence requests, which worked thus far, as on-chip interconnect solutions were built with simpler topology structures that inherently support ordering, such as the AMD opteron HT bus[10], Intel haswell ring[14], and ARM CCI500 crossbar[6]. This no longer holds as MPSoCs scale to large numbers of actors.

The most prominent trend in recent MPSoC design is increasingly large number of actors being integrated onto a single chip with continuing scaling of CMOS and vast expansion of SoC application domains. Although a shared bus or simple crossbar can provide adequate request ordering for cache coherence protocols, they are becoming performance bottlenecks due to inherent scalability problems. As a result, scalable packetized network-on-chip(NoC) is gradually superseding traditional interconnects to cope with ever-increasing bandwidth demands. However, such NoCs with distributed routers are inherently unordered.

Without built-in ordering capability from NoC, the cache coherence protocol has to rely on indirection to serialization or ordering points which can forward the requests to other processors in order. This indirection adversely affects performance. It increases the network latency for delivery of coherence requests to other processors. Furthermore, it concentrates broadcasting traffic at the ordering points, which may become communication bottlenecks in the network. Alternatively, we can switch from the snoopy coherence prevalent in today's MPSoCs to directory-based protocols which can work atop unordered NoCs, like the designs in [28]. In a directory-based protocol, coherence requests are first sent to serialization points called *directory*, which track sharing status of each cache line, and forward intervention or invalidation requests accordingly. Since the directory handles coherence requests ordering, it only requires the interconnect to provide point-to-point ordering for each source-destination pair. However, it cannot avoid the performance degradation due to indirection latency, not to mention the area overhead imposed by keeping cache line information in the directory. Most critically, it requires redesign of existing SoC cores and cache controllers, which adds complexity in both design and verification, impacting the already tight design-to-market time.

In this paper, we propose that the scalable NoC should support coherence request *ordering*, in addition to communications. We propose an ordered NoC design platform (Ordered-NoC) for shared-memory MPSoC to provide efficient request ordering inside the NoC, and to support a wide range of cache coherence and memory consistency models. It universally

[1]Recently new cache coherence protocol AMBA 5 CHI has been proposed, targeted towards scalable many-actors MPSoCs. However, publicly-available information is very limited, and coherence request ordering model seems similar to AMBA 4

applies to irregular MPSoC topologies, while adhering to myriad existing cache coherence protocol standards so that existing cores and IPs can be readily plugged into the platform.

Experimental results show that the proposed solution can be applied to a variety of MPSoCs, ranging across different network topologies and request ordering semantics. The full-system simulation with heterogeneous MPSoC Rodinia benchmarks show that it reduces the request latency by 37.6% and 35.7% over ordering points in 2D-mesh and butterfly fat tree topologies, respectively. This translates to overall runtime improvements of 17.8% and 12.0% in each topology, for a 36-node and 32-node MPSoC respectively.

The rest of this paper is organized as follows. Section 2 presents relevant background and motivation, and Section 3 reviews related work. Section 4 presents the proposed design. Section 5 reports experimental results, and Section 6 concludes.

II. Background and motivation

A. Cache coherence and memory consistency

The intuitive view of shared memory expects that read requests will always observe the latest write value to the memory. However, implementation of this intuitive shared memory is not straightforward in the presence of multiple caches. While caches are an indispensable part of processors due to high off-chip DRAM access delay, they create a serious design challenge for MPSoCs: *cache coherence problem*. If a variable is replicated into multiple local caches, processors can observe different values for the same variable. Further, if two or more processors attempt to write the same memory location simultaneously, processors might observe write values in different orders from each other. Thus, we need well-defined rules specifying correct shared memory behavior so as to provide a basis in writing parallel programs. These rules are often described in two separate concepts: *cache coherence* and *memory consistency*.

Cache coherence defines memory access ordering for the *same memory location* by two constraints: (1) write must be eventually seen by other processors; (2) writes to the same location must be seen in the same order by all processors. In contrast, memory consistency specifies memory access ordering across *different memory locations*. Table I summarizes various memory consistency models classified according to their ordering requirements. The most straightforward memory consistency model is *sequential consistency*, in which memory operations are observed in the same total order by all processors as all four types of program order are enforced. While sequential consistency corresponds to intuitive understanding of shared memory behavior, it hinders compiler and hardware optimizations from exploiting out-of-order instruction execution, and thus leads to severe performance degradation.

B. Motivating case study

The 36-core MIT SCORPIO chip recently fabricated on 45nm SOI process [13] forms the motivation of our proposed Ordered-NoC platform for MPSoC. SCORPIO demonstrated that ordering can be embedded within a scalable mesh NoC, delivering 24% better performance than directory-based ordering at low power and area overheads, and allowing for plug-and-play with Freescale PowerPC cores and Cadence memory controller through AMBA AXI and ACE communication standard.

In SCORPIO, for every memory request, the network interface controller(NIC) broadcasts notification of the request on a separate ordering network called *notification network*. The receiving NIC then reorders and delivers the requests in the notification order. The notification is represented as a bit-vector in which each bit indicates if corresponding processor has sent a memory request into the main network. The notification broadcast from each node has a fixed route along XY-routing path in 2D-mesh, and the routing can be done in bufferless manner, since two incoming notifications can be merged if the broadcasting paths are overlapping. This bufferless routing can guarantee a fixed bound for broadcast latency from each node, which enables synchronization of notification broadcasts. NIC issues notifications only at the start of each time window, which is set greater than the maximum broadcast latency. Consequently, all nodes receive the same set of notifications at the end of each time window. By applying a common ordering rule to received notifications, each NIC can locally constitute the same ordered list of source processor ID(PID)s which are associated with the source nodes of each notification. The main network is designed to provide point-to-point ordering, which guarantees in-order delivery of requests from the same source. Accordingly, each coherence request can be identified by its source PID. Although the coherence requests from different sources can arrive in any order, they can be reordered at each NIC according to the global order settled by notifications.

While SCORPIO pointed at the promise of embedding ordering within the NoC, it was heavily tailored for a specific target chip and has several limitations that make it not applicable universally to most MPSoCs. Firstly, its bufferless notification network is designed only for 2D-mesh topology; Heterogeneous processors in MPSoC platforms have non-uniform block sizes and bandwidth requirements, which renders 2D-mesh suboptimal in many target applications. Instead, the platform-specific optimal topology should be generated according to communication analysis at the design phase[24]. Secondly, SCORPIO indiscriminately reorders every coherence request in a strict total order, which often results in unnecessary ordering latency and performance loss. For example, if only read requests with different cache line addresses arrive, they could have been served immediately without waiting to be ordered. Specifically, the SCORPIO notification network was designed to provide a total order, aiming for sequential consistency. While sequential consistency is functionally sufficient for any memory consistency model, it imposes superfluous ordering and increased latency for the relaxed consistency models that are prevalent among existing commercial cores.

In short, as MPSoCs scale to many IP blocks, a universal NoC platform that can support ordering across a wide spectrum of shared memory models at high performance is needed. In this paper, we propose such a generalized NoC platform that encapsulates ordering, and efficiently supports a wide range of memory models, so as to interface readily with existing cores,

[2]Although cache coherence protocols usually require a total ordering for the same memory location, Read-After-Read ordering can be possibly relaxed since reordering does not make any semantic difference.

Memory Consistency Model	Total Ordering for the same address[2]	Program Order	Write Atomicity	Memory Fence
Cache Coherence	✓			
Sequential Consistency	✓	✓	✓	
TSO, Intel x86/x64, Sun Sparc v8	✓	RAR,WAR,WAW	✓	✓
Processor Consistency	✓	RAR, WAR, WAW		✓
Sun PSO	✓	RAR, WAR		✓
Weak Ordering, Release Consistency, IBM 370/Power, ARM	✓			✓

TABLE I: Ordering requirements according to memory consistency models

cache and memory controllers, and other IP blocks.

III. Related Work

Snoopy cache coherence on unordered networks Commercial cache coherence protocol standards such as AXI Coherence Extensions(ACE) [5], OCP Coherence Extension [1], AMD Hypertransport [9], and home snoop in Intel Quickpath Interconnect(QPI)[15] require the interconnect to support request ordering for correct operation. While an ordering point can provide such request ordering support over unordered point-to-point interconnects, they add unnecessary latency. Hence, there have been several proposals to enable direct coherence request snooping. Timestamp snooping[21] and INSO[3] assign a global logical order to each coherence request, and the interconnects process them in the logical order. Recently, SCORPIO[13] introduced request ordering based on notifications by decoupling ordering function from the main network. These prior works can provide only total request ordering, whereas Ordered-NoC aims to selectively relax ordering requirements according to target memory consistency model. Source snoop such as that in QPI[15] also allows out-of-order snooping for lower latency, but it requires a separate home agent for conflict resolution, which requires each transaction to wait for the acknowledgment from the home agent with increased latency. In contrast, Ordered-NoC can resolve ordering conflicts without the intervention of separate agents thanks to a chip-wide global request order. Token Coherence protocol[20] is an alternative solution for request ordering on unordered network. To access a particular cache line, the requesting processor should collect at least one token for read, and all tokens for write. Similar to our approach, it can avoid both indirection latency and total ordering overhead. However, Token Coherence requires reimplementation of cache controllers to bookkeep the number of tokens for each cache line, whereas Ordered-NoC can be readily applicable to standard cache coherence protocols.

NoC for heterogeneous SoCs There has been a plethora of NoC studies for a heterogeneous multi-core system or MPSoC. Application-specific communication requirements and non-uniform IP block sizes render homogeneous NoC routers suboptimal, and naturally lead to heterogeneous architectures. Substantial research has proposed design flows for optimized topology generation based on bandwidth and latency characterization[24]. In topology design, floorplan issues have been considered to facilitate timing closure[25], [26]. Design methodologies for optimized router architectures for application-specific communication patterns have also seen substantial research [23], [12]. Ordered-NoC provides a general framework to build request ordering functionality into the NoC for heterogeneous NoC. It is orthogonal to the optimization of the main network for communications, whether it be topology, router micro architecture, etc. Ordered-NoC can embed the ordering network and network interface controllers in any conventional NoC, interfacing seamlessly existing IP blocks and supporting any shared memory semantics.

As offchip memory traffic is dominant in today's MPSoC, researches have proposed optimized topology and router architectures for higher DRAM utilization[16], [27], [11]. There have also been prior NoC studies on ordering to guarantee in-order packet delivery for multi-path routing [22], [18], or for concurrent split memory transactions[17], [11]. Heterogeneous NoC designs to facilitate cache coherence protocols have been proposed as well, such as reconfigurable NoC for localized snooping[29] and efficient broadcast support for acknowledgments[19]. Ordered-NoC goes beyond optimizing a NoC for shared memory communications; It advocates instead the embedding of request ordering in the NoC for supporting shared memory coherence and consistency semantics.

IV. Ordered-NoC Design

Our proposed Ordered-NoC platform supports ordering within the NoC, in a general manner so that existing IP cores, cache and memory controllers that run a variety of memory semantics can be plugged unchanged. In addition, it supports heterogeneous, irregular MPSoC layouts by enabling ordering to be embedded within any NoC topology.

The tenet behind our Ordered-NoC platform lies in a separate ordering network that maintains *ordering* amongst requests, while the main network handles the traditional *communication* function of NoCs. This split enables the main network to deliver coherence requests in any order. The NIC issues notifications on the ordering network in synchronization with a time window, which is the maximum notification broadcast latency of the ordering network. By the end of each time window, all nodes receive the same set of notifications. Each NIC applies the same ordering rule to the received notifications, and thus shares a common global request order. Ordered-NoC selectively reorders the requests from the main network according to the memory semantics. If any inconsistent request ordering is detected at each NIC, it performs necessary recovery actions. By sharing a common request order, each NIC can resolve ordering conflicts in a distributed manner without the intervention of separate home nodes or agents functioning as ordering points. We will next dive into how our Ordered-NoC platform enables the selective enforcement of ordering between specific memory transactions, thus supporting diverse memory semantics, then go into how it functions atop any NoC topology, including irregular ones.

A. Reordering requests

Figure 1 shows the microarchitecture of the NIC for reordering coherence requests. Outgoing coherence request from

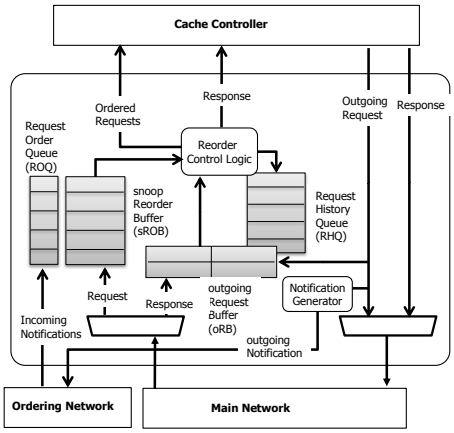


Fig. 1: Microarchitecture of Network Interface Controller(NIC)

the local processor is kept at the outgoing request buffer (oRB) until it receives the data response and completes coherence operation. The response is also stored into oRB until it is ready for the delivery after ordering conflict is resolved. The global request order by the ordering network is maintained by storing the corresponding source PID for each notification in the request order queue(ROQ). Snoop reorder buffer (sROB) is a circular buffer holding incoming requests until the delivery of the request to the processor. Each entry of sROB is paired to each PID in the ROQ in order from the top to the bottom (from the oldest to the most recent).When the oldest entry from sROB retires, the corresponding top (oldest) PID from the ROQ is also removed. If an incoming request arrives at the NIC on the main network, its source PID is matched against the ROQ and moved into a corresponding entry of sROB. Once stored in the sROB, a request can be delivered for snooping in any order, subject to the ordering requirements. Requests are retired in-order from the sROB as a snooped request can only release its entry when it reaches the head of the sROB.

B. Resolving ordering conflicts

Since out-of-order snooping is allowed independently at each NIC when not constrained by the memory semantics, each cache controller may end up with inconsistent cache data from each other. To address this, Ordered-NoC makes a slight modification to coherence protocol that does not interfere with internal states or behaviors of cache controllers; when coherence request is delivered from sROB to the cache controller, NIC extends the request with a bit-vector(called the snoop status vector (SSV)) indicating the snoop status of coherence requests for the same cache line. The length of the SSV is $2 \cdot (sROB_d - 1)$ bits where $sROB_d$ is the depth of sROB; a half for preceding requests and the other half for subsequent ones to cover all requests within the maximum reordering boundary. The SSV can be obtained from the current snoop status of sROB. For the requests already retired from sROB, it can search for the same cache line address the request history queue(RHQ), which stores up to $sROB_d - 1$ recently retired requests for future look-up. When the cache controller returns a data response, the SSV received from the request is carried over to the response. When other NICs receive the response, they extract the SSV, and compare it against the snoop status of local sROBs to resolve any inconsistency. Once those inconsistencies found, NIC is required to take relevant recovery actions. We present two different schemes.

Recover-Total-Order The first scheme is to restore correct behavior according to the global request order so that memory operations take the same effect as the request snooping at each NIC is performed in the total order. In this scheme, we allow an out-of-order snooping only for read requests, which means that only read requests can be snooped ahead of other preceding requests. Under this scheme, out-of-order snooping for read requests can be viewed as prefetching for reduced latency. If prefetched value is known to be outdated, then NIC should wait for updated value while dropping the obsolete one. When receiving the response, NIC inspects the SSV of the response to check if the response is sent from the latest data owner. In other words, the response is confirmed to be valid if there is no preceding write request for the cache line which is not snooped in the SSV from the response(Accordingly, the SSV only needs to provide the snooping status for preceding requests.). Otherwise NIC discards the response since the data will be overwritten by new data owner who issued that write request.

Since we do not allow out-of-order snooping of write request, new data owner cannot serve any preceding request with newer value. Further, we can guarantee that all subsequent requests after a write request (until a next write request) are served by new data owner, by prohibiting snooping incoming requests for the same cache line before completing all preceding outgoing requests. If new response arrives while there is already another response stored in the oRB, then NIC compares SSVs of two responses from the earliest bits to later. Because the later owner always has longer consecutive snooped requests, we can distinguish the newer response. Additionally, since NIC is required to deliver the response to the processor only when the local request reaches the head of the sROB, the global request order provides a logical point at which each memory operation takes effect. Thus, we can guarantee sequential consistency as all memory operations appear in the global request order.

Figure 2 gives a walk-through example, which features 3 processors with MOESI cache coherence protocol[3] with $sROB_d = 3$. For convenience of explanation, the RHQ is omitted, and the requests remain in the sROB even after the retirement. In the example, we denote read(write) request from processor i by $R_i(W_i)$. The figure also shows cache states of each processor for two cache line addresses. The request for the second address is shown as shaded in the sROB. (R_3 is for the second cache line, and the others are for the first). (a) Each processor sends a coherence request to the main network, and NIC issues a corresponding notification at the start of notification time window to the ordering network. All notifications arrive until the end of time window, and each NIC settles on the same request order $2 \rightarrow 1 \rightarrow 3$ as shown in the figure. (b) R_3 is the first request arriving at processor 1 and 2, while R_1 is the first for processor 3. They are snooped without waiting for preceding requests. The SSVs are $(0, 0)$ (an upper half for preceding requests) since there were no earlier snooped requests. Processor 2 is at M(Modified) state

[3]In our example, it is assumed that processor at O(Owner) state takes responsibility for generating data response when receiving read request from other processors.

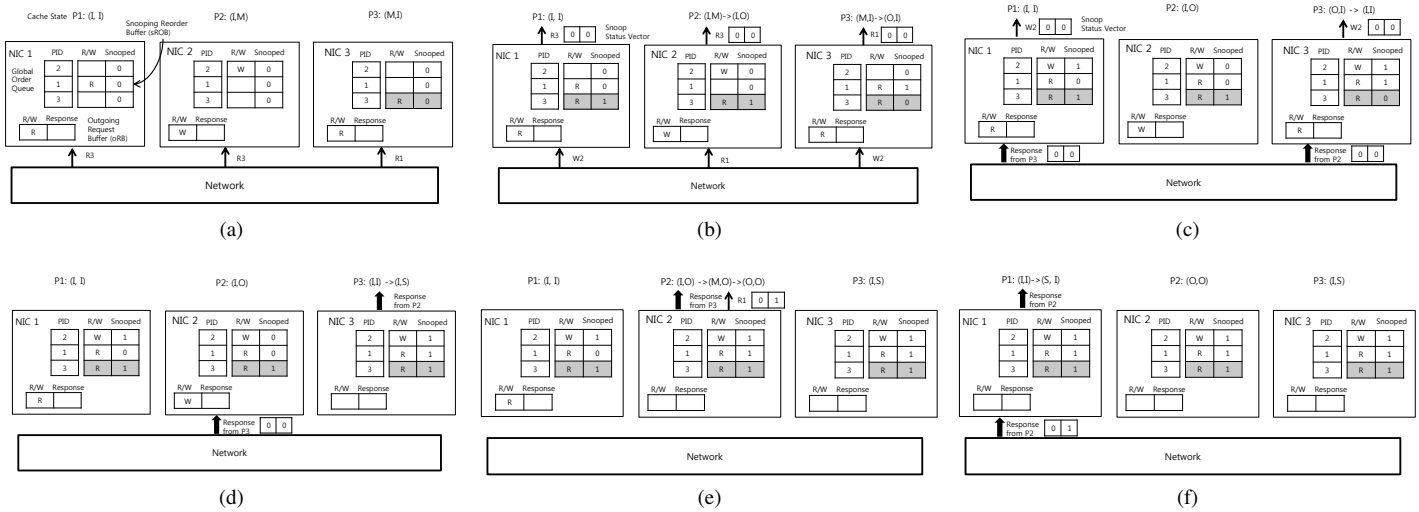


Fig. 2: Walk-through Example for Recover-Total-Order

for the second cache line, and changes the state to O(Owner) as it snoops R_3. Similarly, processor 3 changes M(Modified) state for the first cache line to O(Owner) as it snoops R_1. (c) Note that R_1 is received by NIC 2, but it is kept at sROB until local request W_2 is completed since W_2 has smaller request order and two requests have the same cache line. Processor 3 sends the data response to processor 1. NIC 1 receives the response but discards it since there was a preceding write W_2 with the first cache line address, which is not snooped in the SSV, $(0, 0)$ at the response. On the other hand, NIC 3 can accept the response from processor 2, since there is no previous requests with the second cache line address. (d)(e) The response for W_2 arrives at NIC 2 and is delivered to processor 2. After the retirement of W_2, NIC 2 can deliver R_1 for snooping. (f) The response from processor 2 arrives at NIC 1. As the SSV shows that the last write W_2 is serviced, the response is confirmed to be valid and delivered (two steps are shown in the same figure). Note that R_3 was immediately served at NIC 2 without reordering latency. Still, final memory operations have the same effect as all coherence requests are snooped in the global request order, $2 \rightarrow 1 \rightarrow 3$.

Reorder-On-the-fly We have seen how Recover-Total-Order scheme can guarantee strong memory consistency models like sequential consistency and total store order(TSO) without reordering of every coherence request in a total order. However, it still imposes considerable ordering restrictions. Although it allows out-of-order snooping, the response must wait until all preceding requests are received. It also enforces in-order snooping of write requests and generates redundant responses. We can further enhance the performance for the relaxed memory consistency by addressing these issues.

The second scheme aims to provide utmost flexibility in the request ordering as the actual snoop order is determined on the fly by the data owner at the time. When receiving the response, local snooping status is readjusted by the SSV from the received response as follows. There are two types of requests which need readjustment in the snoop order. First, NIC identifies unsnooped requests by the local processor which are known to be already snooped by the previous owners. NIC marks those requests in the sROB as snooped, and skips the delivery. On the other hand, there can be the requests that are already snooped by the local processors, but not snooped by the previous owner. To maintain the same snoop order, NIC changes them unsnooped, and resends those requests to the processor after returning the response.

Figure 3 revisits the previous walk-through example. Now it features only the first cache line address. (a)(b) The global request order is established as $2 \rightarrow 1$ as before. Processor 1 and 3 snoop W_2 and R_1 respectively. (c) The response from processor 3 arrives at NIC 1. Processor 3 next snoops W_2. (d) NIC 1 can deliver the response from processor 3 immediately. Recall that in the previous walk-through, NIC 1 discarded the first response expecting that a valid response would arrive later. On the contrary, every response is valid now, and NIC is responsible for readjusting its local snoop order accordingly. In the example, NIC 1 changes the snoop status of W_2 to unsnooped , and resends it to the processor, since the response from processor 3 demands to reorder W_2 behind R_1. (e) NIC 2 delivers the response for W_2. This time, the SSV indicates that R_1 is already processed by the previous data owner. Accordingly, R_1 is marked as snooped without actual snooping. Note that the final snoop order is $1 \rightarrow 2$, which is set by the first data owner, processor 3.

C. Ordering network construction

The role of the ordering network is to broadcast notifications to all nodes within a guaranteed latency, which provides synchronized time window for global request ordering. Figure 4 illustrates the ordering network construction process atop the main network. We assume the design phase to produce any irregular NoC topology which is optimized according to various factors such as proximity and communication traffic analysis. Given the underlying main NoC topology(Figure 4a), we independently construct a bufferless broadcast tree to deliver notification bits for each node. The broadcast tree is carrying only a 1-bit notification signal (with an additional flow-control signal to stop further notification injections). The tree is constructed by running a shortest-path tree algorithm for each node as a root, as shown in Figure 4b for two nodes J and K, to obtain the minimum broadcast latency. The final

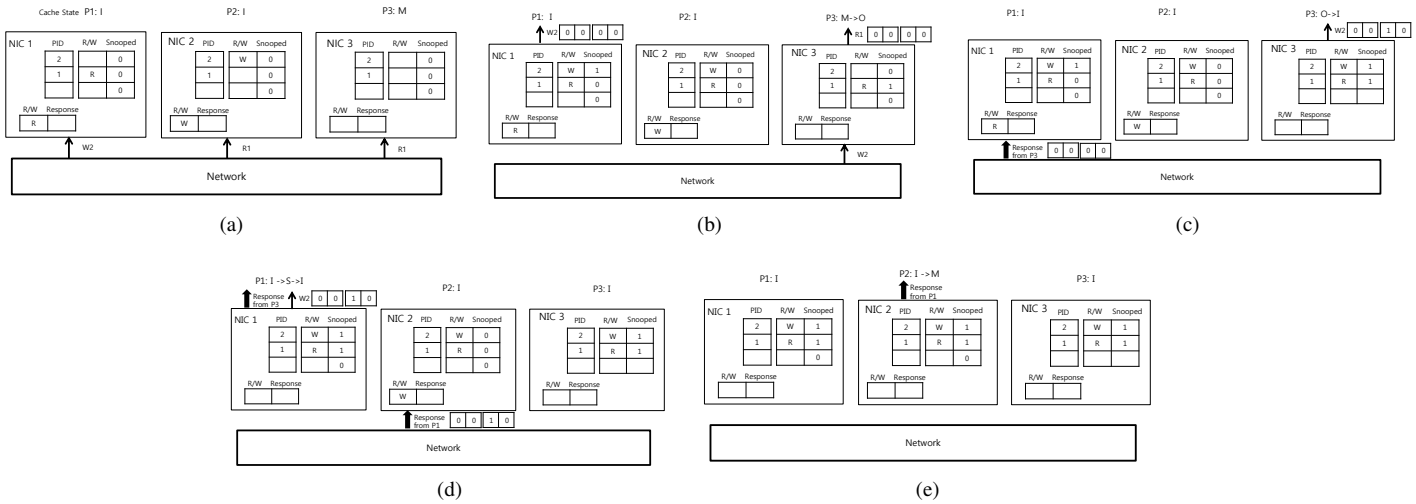
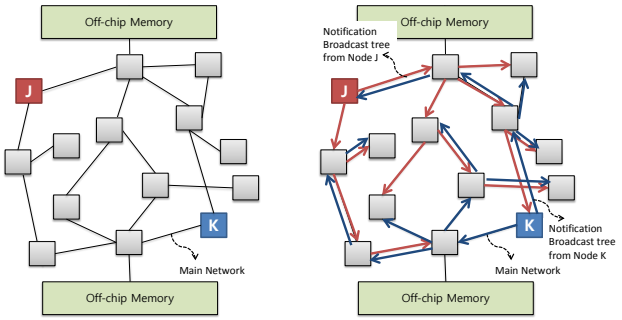


Fig. 3: Walk-through Example for Reorder-On-the-fly



(a) Given main network topology

(b) Notification broadcast trees from node J and K

Fig. 4: Construction of ordering network atop given main network topology. The ordering network is the sum of individual broadcast trees rooted at each node.

ordering network is the sum of all individual broadcast trees by iterating this process for all nodes. The latency bound is given as the maximum latency of all broadcast trees.

In the main network, the requests from different sources are allowed to pass through the routers in an out-of-order manner. Without a proper measure, out-of-order requests may occupy all the virtual channels in the router, while the NIC waits for the request with smaller request order, and thus blocks all virtual channels, which leads to a deadlock. To address this, we add an escape virtual channel(EVC) to ensure the request with the smallest request order at the time can always proceed to the next router. For this purpose, the main network router should track the global request order from the ordering network with the PID order queue (the same structure as the request order queue inside NIC in Figure 1), while observing PIDs of coherence requests that have already passed through the router. It matches the PID of passing requests against PIDs in the PID order queue and remove it from the queue, and vice versa. EVC is reserved for the request having the PID that sits on the top of the PID order queue. Accordingly, NIC sets aside a separate register for EVC in the request buffer.

V. Experimental Results

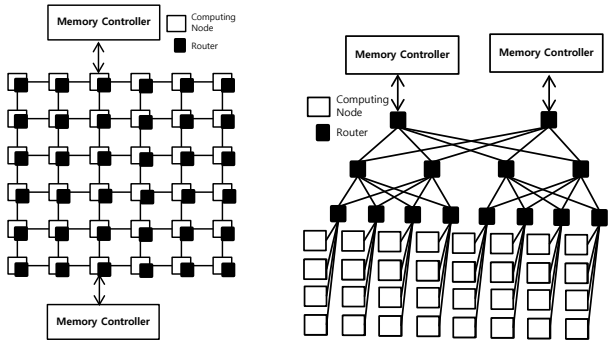
We illustrate the generality of the Ordered-NoC platform by applying it to a diverse range of MPSoC designs and memory models, then evaluate its impact on memory access latency and overall application performance against the state-of-the-art.

A. Methodology

We used the C++-based multi-processor simulation tool GEM5 [7] along with the cycle-accurate network model Garnet [2] modified to model the proposed Ordered-NoC. For workloads, we use the heterogeneous computing benchmark suite, Rodinia [8]. Simulations are run to completion for each application, with statistics gathered at the end of the parallel portions.

Our baseline models include both directory and ordering points-based protocols. In all system configurations, each processor has local private L1/L2 caches with MOESI cache coherence protocols. Different system configurations then differ only in how the coherence requests are ordered and delivered to other processors. In the directory-based baseline protocol, the sharer information of each cache line is stored in distributed directories at the static home nodes. When a cache miss occurs at the local L2 cache, the coherence request is delivered to the static home node, and the directory either forwards the request to the data owner (the processor having O(Owner) state in the local cache) or generates invalidations to all the sharers. Both directory and ordering points-based protocols use a directory for request ordering, but in ordering points-based protocols, the directory contains no storage for the sharer information, and thus just broadcasts forwarded coherence requests and invalidations. We use SCORPIO as another baseline to illustrate how Ordered-NoC reduces the performance overhead of total ordering by applying request ordering selectively.

To show the generality of our proposed Ordered-NoC platform, we applied it to MPSoC designs running atop two



(a) 2D-Mesh for 36-core (b) Butterfly Fat Tree for 32-core

Fig. 5: NoC Topologies for evaluation

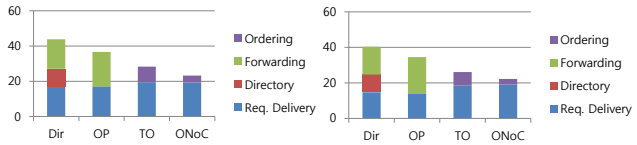
different topologies (2D-mesh and butterfly fat tree) and supporting two alternative memory models, total store order(TSO) and relaxed memory consistency. In TSO, each write request is required to complete after invalidating all the sharers to provide write atomicity, or the cache coherence protocol should operate on a strict total request order. Accordingly, Recover-Total-Order scheme is applied in Ordered-NoC. In relaxed memory consistency model, we assume basic cache coherence with the support of atomic writes and memory fences, while writes can complete before receiving the acknowledgments for invalidations, hence, Ordered-NoC's Reorder-On-the-fly scheme can be applied. Table II summarizes the detailed system configurations.

Processor Configuration	
Processing cores	in-order core with X86-64 ISA
Operating System	Linux Kernel v. 2.26.28.4
Cache line size	64 Bytes
L1 cache	Split 32KB I&D, 4-way, 1-cycle access latency
L2 cache	Exclusive unified 64KB per each core 4-way, 4-cycle access latency
Cache coherence	MOESI protocol with private L2 cache
Request Ordering	Directory(Dir) , Ordering Points(OP) Total ordering(TO) by SCORPIO, Ordered-NoC
On-Chip Network	
Topology	2D-mesh (6×6) for 36-core Butterfly fat tree for 32-core
Latency	1-cycle pipeline for router 1-cycle for link traversal
Virtual Networks	2
Virtual Channels	4 per Virtual Network
Link channel width	16 Bytes
Directory	10-cycle access latency
Memory Interface	
Memory	2 memory controllers (10-cycle latency) 100-cycle access latency for off-chip DRAM

TABLE II: Target System Configuration

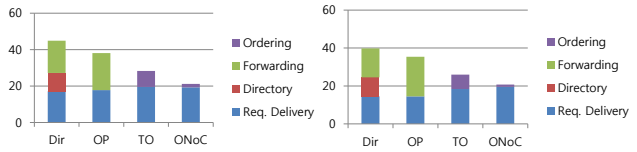
B. Results

Figures 6 and 7 compare average snooping latency of coherence requests of each system configuration. In a directory-based protocol(Dir), when a L2 cache miss occurs, the request is first sent to the directory(Req. Delivery). After a directory look-up(Directory), it is forwarded to the sharers to receive a data copy, or to invalidate upon writes (Forwarding). For the request ordering system based on ordering-point(OP), the request is forwarded to all nodes without incurring a directory look-up latency. In total ordering(TO) and Ordered-NoC(ONoC), the coherence request is directly delivered avoiding forwarding latency, but each NIC imposes additional ordering latency(Ordering). As shown in Figure 6a, Ordered-NoC leads to reduction in memory latency of 18.1% and 14.9% in each topology for TSO when compared against total ordering. With relaxed memory consistency, Ordered-NoC results in even higher latency reduction of 25.0 % and 21.7 % over total ordering[4]. This shows that Ordered-NoC can flexibly configure to the desired level of request ordering, and yet serve memory requests quickly. Especially, in a butterfly fat tree topology, where SCORPIO cannot provide request ordering for cache coherence, the latency reduction over ordering points reaches 12.3 and 14.7 cycles for each memory consistency model, which are 35.7% and 41.5% reduction. In 2D-mesh topology, the latency reduction is 13.4(37.6%) and 16.9 cycles(44.5%) over ordering points.



(a) 36-core(2D-Mesh) (b) 32-core(Butterfly Fat Tree)

Fig. 6: Snooping Latency Breakdown for TSO

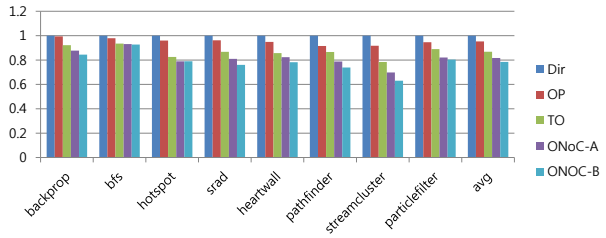


(a) 36-core(2D-Mesh) (b) 32-core(Butterfly Fat Tree)

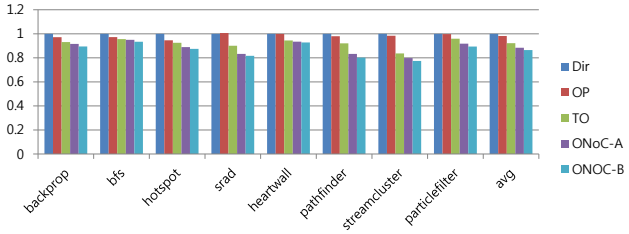
Fig. 7: Snooping Latency Breakdown for Relaxed Memory Consistency

Figure 8 shows overall performance improvement by Ordered-NoC in terms of normalized runtimes (against the Directory baseline). In general, Ordered-NoC tends to have relatively smaller performance impact compared to the snooping latency reduction, because it is relevant to only L2 cache misses. The figure graphs normalized runtimes for relaxed memory models, where ONoC-A represents Recover-Total-Order scheme, and ONoC-B Reorder-On-the-fly. The overall runtime reduction is 17.8% and 12.0% compared to ordering points in each topology. The reduction over total ordering is 10.3% and 7.2%, respectively. As expected, the system performance improvement is highly sensitive to the cache miss rate. For example, **streamcluster** has 37.3 L2-cache misses per thousand instructions, whereas **bfs** merely has 1.2. Therefore, in **streamcluster**, Ordered-NoC gives 31.7% and 21.4% runtime reduction over ordering points in each topology, which are considerably higher than 6.4% and 4.1% in **bfs**. Considering that the input and working set sizes in the experiments are markedly limited due to the limitation of simulation times, we believe that Ordered-NoC will show more significant performance improvement in realistic workloads that will have much larger cache footprint.

[4]As SCORPIO can be applied only for 2D-mesh, for butterfly fat tree topology, total ordering(TO) is implemented by Ordered-NoC by only allowing strict in-order snooping.



(a) 36-core with 2D-Mesh



(b) 32-core with Butterfly Fat Tree

Fig. 8: Normalized runtime

VI. CONCLUSION

In summary, we motivated the need for an ordering layer in NoCs that can allow actors in shared-memory MPSoCs to plug-and-play readily. We proposed such a universal Ordered-NoC that works with any memory coherence and consistency model and NoC topology. We demonstrated how Ordered-NoC delivers better performance than other alternatives.

ACKNOWLEDGMENT

This work was supported by the Center for Future Architectures Research (C-FAR), one of six SRC STARnet Centers sponsored by MARCO and DARPA.

REFERENCES

[1] Accellera. Open core protocol 3.0 specification. *avaiable at http://www.accellera.org/*, 2009.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *ISPASS*, pages 33–42, 2009.

[3] N. Agarwal, L. Peh, and N. K. Jha. In-network snoop ordering(inso): Snoopy coherence on unordered interconnects. In *HPCA*, 2009.

[4] D. Anderson. *HyperTransport Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] ARM. Amba axi and ace protocol specification axi3, axi4, and axi4-lite, ace and ace-lite. *avaiable at http://infocenter.arm.com*, 2013.

[6] ARM. Arm corelink cci-500 cachecoherent interconnect. *avaiable at http://infocenter.arm.com*, 2015.

[7] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Washington, DC, USA, 2009. IEEE Computer Society.

[9] P. Conway and B. Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, Mar 2007.

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, Mar. 2010.

[11] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen. Memory-efficient on-chip network with adaptive interfaces. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):146–159, Jan 2012.

[12] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–291, New York, NY, USA, 2009. ACM.

[13] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh. Scorpio: A 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, pages 25–36, 2014.

[14] P. Hammarlund et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[15] Intel. An introduction to the intel quickpath interconnect. *Intel White Paper*, 2009.

[16] W. Jang and D. Z. Pan. An sdram-aware router for networks-on-chip. In *Proceedings of the 46th Annual Design Automation Conference*, 2009.

[17] W.-C. Kwon, S. Yoo, J. Um, and S.-W. Jeong. In-network reorder buffer to improve overall noc performance while resolving the in-order requirement problem. In *Design, Automation Test in Europe Conference Exhibition, 2009*, pages 1058–1063, April 2009.

[18] M. Lis, M. H. Cho, K. S. Shim, and S. Devadas. Path-diverse in-order routing. In *Green Circuits and Systems (ICGCS), 2010 International Conference on*, pages 311–316, June 2010.

[19] M. Lodde, J. Flich, and M. Acacio. Heterogeneous noc design for efficient broadcast-based coherence protocol support. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 59–66, May 2012.

[20] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceeding of the Annual International Symposium on Computer Architecture*, pages 182–193, 2003.

[21] 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 *ASPLOS*, pages 25–36, 2000.

[22] S. Murali, D. Atienza, L. Benini, and G. De Michel. A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. In *Proceedings of the 43rd Annual Design Automation Conference*, 2006.

[23] S. Murali, L. Benini, and G. De Micheli. An application-specific design methodology for on-chip crossbar generation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1283–1296, July 2007.

[24] S. Murali and G. De Micheli. Sunmap: A tool for automatic topology selection and generation for nocs. In *Proceedings of the 41st Annual Design Automation Conference*, pages 914–919, 2004.

[25] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo. Designing application-specific networks on chips with floorplan information. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 355–362, Nov 2006.

[26] S. Pasricha, N. Dutt, E. Bozorgzadeh, and M. Ben-Romdhane. Floorplan-aware automated synthesis of bus-based communication architectures. In *Proceedings of the 42nd Annual Design Automation Conference*, pages 565–570, 2005.

[27] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. A dram centric noc architecture and topology design approach. In *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*, pages 54–59, July 2011.

[28] D. Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[29] H. Zhao et al. A hybrid noc design for cache coherence optimization for chip multiprocessors. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.