In-Network Memory Access Ordering for Heterogeneous Multicore Systems

Jieming Yin
Lehigh University
Bethlehem, PA, USA
yin@lehigh.edu

Antonia Zhai
University of Minnesota, Twin Cities
Minneapolis, MN, USA
zhai@umn.edu

***Abstract*—In heterogeneous multicore systems, implementing a programmer-friendly memory consistency model while maximizing memory-level parallelism is challenging. Ideally, memory accesses can be performed out of order as long as program order is not violated. But enforcing memory access order at the end-point (e.g., a core) prohibits a number of architecture optimizations and limits memory-level parallelism. In this work, we explore the opportunity of preserving memory access order inside the on-chip interconnection network. We propose a hybrid switching networks-on-chip (NoC) attached with a light-weight token ring network to guarantee global memory access order. The hybrid switching NoC that supports both packet and circuit switching serves as the underlying communication infrastructure, while the token ring network is used to preserve memory order among multiple ordering points. Our proposed design enables strong memory consistency models and deterministic program execution, with negligible performance overhead compared to an un-ordered packet switching network.**

I. Introduction

Memory consistency models specify how reads and writes from one thread are visible to other threads. A key aspect in implementing a memory consistency model is to maintain certain memory access order. A strong memory model such as sequential consistency (SC) does not allow any memory reordering [1]. SC is intuitive for programmers to reason about. Debugging programs under SC is straightforward due to its good reproducibility and predictability. However, strong memory models prevent many optimizations and degrade performance. As a result, both CPUs and GPUs adopt weaker memory consistency models, which enable certain degrees of memory access reordering for better memory-level parallelism. Many modern CPUs adopt total store ordering (TSO), which guarantees stores from the same thread to occur in program order while allowing younger reads to bypass pending stores [2]. Many commercial GPUs adopt release consistency (RC) models that rely on programmers to explicitly manage communication and synchronization operations [3]. While weak memory models might provide better performance, debugging becomes extremely difficult. Because memory reorderings occur in an unpredictable manner due to the relaxed semantics of weak consistency models, programmers are required to understand the program to ensure the correct memory access order.

Implementing a programmer-friendly memory consistency model while maximizing memory-level parallelism is challenging, especially in heterogeneous systems where data-parallel cores generate tremendous amount of outstanding memory requests. End-point ordering at cores is expensive, because it prohibits a number of architecture optimizations. However, if the correct memory operation order is provided in the interconnection network during message transmission, we can potentially improve the performance, as memory instructions can still be reordered inside each core. Another advantage of in-network memory ordering is that the implementation can be independent of the core architecture and the underlying cache coherence protocol, which is particularly attractive for future heterogeneous systems where a variety of computation resources are integrated together in the form of chiplets [4].

In this work, we explore the opportunity of preserving memory access order inside the NoC. Packet switching NoCs are widely used in multicore and many-core systems due to its flexibility and scalability [5]–[7]. However, packet switching NoCs do not inherently guarantee message order and could result in uncertainties during message transmission. Circuit switching NoCs, on the other hand, are predictable and can preserve message order inside the network [8], [9]. The combination of packet and circuit switching NoCs provides us a communication infrastructure that can effectively enforce memory access order within the network. Based on the above observations, we propose a NoC design that enables in-network memory access ordering for CPU-GPU heterogeneous multicore systems. The proposed NoC consists of a hybrid switching main network and a light-weight token ring network. The hybrid switching network that supports both packet and circuit switching handles heterogeneous traffic efficiently. The light-weight token ring network maintains the global order of memory requests. The proposed NoC architecture ensures deterministic program execution, which simplifies multi-threaded program debugging in heterogeneous systems. Compared to an un-ordered packet switching NoC, our design causes only 0.7% and 1.0% performance degradation in 16-node and 36-node heterogeneous systems, respectively.

II. Background and Motivation

Consider the example shown in Figure 1 involving two cores ($C0$ and $C1$) and two shared variables (x and y) stored at different memory locations ($M3$ and $M1$). In the code snippet presented in Figure 1a), $C0$ first writes to x, and then writes to y; while $C1$ first reads from y, and then from x. Program order (PO) is indicated by the arrow. Now assume both x and y are initialized to 0 at the beginning of program execution, as suggested by Figure 1b). Due to the uncertainty of message traversal latency in packet switching NoCs, these read and

978-1-4673-9030-9/20/$31.00

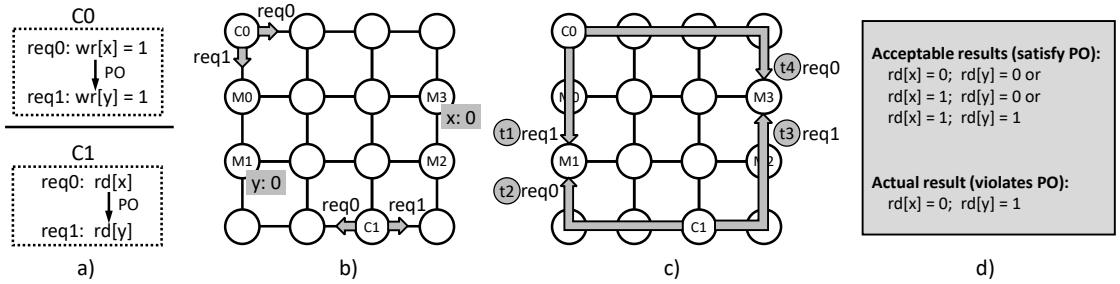


Fig. 1. Message transmission in an unordered packet switching network.

write requests might arrive at the memory controllers in any order, even though they are issued from the cores in program order. Figure 1c) suggests one possible order, in which *req*1 from $C0$ arrives at $M1$ first and writes to y, followed by *req*0 from $C1$ that reads the value of y; then *req*1 from $C1$ reads the value of x, followed by *req*0 from $C0$ writing to x. Following the suggested order, the value of (x,y) read by $C1$ is $(0,1)$. Depending on the memory consistency model the underlying machine assumes, this result can be illegal as it violates the program order (Figure 1d)).

In order to guarantee the correct result, the core has to maintain the order of read/write requests. One possible way is to stall the later write operations until the previous writes are completed. Such end-point ordering can be very expensive and cause performance degradation. Trade-offs can be made between performance and programmability–relaxed memory consistency models allow reordering of read/write operations while programmers are responsible for adding fence instructions for correct program execution. However, if a program is not properly written, the memory operations may be performed in an order different from what the programmer expects, resulting in unpredictable program behaviors. In the era of heterogeneous multicore architecture, this will lead to significant debugging effort given the mass number of threads supported by the systems. Having realized that end-point ordering is expensive and relaxed memory model is not programmer-friendly, the goal of this work is to seek answers to the question: *How can we exploit memory level parallelism in heterogeneous systems without imposing heavy burdens to programmers?*

III. Memory Access Ordering Network Design

In this section, we first give an overview of the proposed ordering network, and then describe the main network, token network, and network interface designs in detail.

A. Network Overview

At a high level, the ordering network consists of: 1) a hybrid switching main network that supports both packet and circuit switching; 2) a light-weight token ring network passing around a token to ensure global memory access order; and 3) network interface (NI) controllers that examine and update the token, as well as store and manage memory requests.

Hybrid switching main network. A hybrid switching NoC, built upon a packet switching NoC, allows traffic to be sent in both packet and circuit switching manners. Dedicated circuit switching paths are established between source-destination communication pairs. Once a connection is set up, messages from the same source to the same destination can repeatedly reuse the path without destroying the connection. Circuit switching have two major advantages compared to packet switching. First, communication latency is predictable because the length of each circuit switching path and message traversal latency are deterministic. Second, buffering and routing are not required, therefore circuit switching has lower per-hop delay. Time-division multiplexed (TDM) hybrid switching NoC allows fine-grained sharing between packet and circuit switching data paths [8], which is potentially a good candidate for ordering network messages. In our proposed NoC, we enforce memory access order by routing them through circuit switching, while the rest of the traffic is routed through packet switching without preserving any order (more details in Section III-B).

Token ring network. This light-weight token network connects ordering points into a ring. Ordering points are nodes in which global memory access order is maintained, so they must be in the level of memory hierarchy that is visible to all cores (e.g., shared last-level caches, directories, memory controllers). A token is used for maintaining access order, and it is passed through the token network from one ordering point to another. In the proposed design, each memory request that needs to be ordered is assigned an ID. The token records the IDs of the requests that are being served by each ordering point. Each core generates its own request ID independently. Therefore, the token contains n IDs for an n-core system. The network interface controllers at the ordering points modify the token when all requests with the same ID from the same core are completed.

Network interface. The NI is responsible for receiving and examining the token at each ordering point, making sure that the connected memory component (e.g., last-level cache, directory, memory controller) only processes requests whose IDs are smaller than or equal to the ID indicated by the token. Once the memory component finishes processing the requests, the NI will set the corresponding bit inside the token. The NI will also increment the token's request ID when all memory components have finished processing the old requests. The updated token is then injected back to the token network. Notice that a single memory instruction could result in multiple memory requests sending to multiple destinations, it is possible that more than one memory component receive requests from the same core with identical request ID. To guarantee in-order memory access, the NIs are also responsible for storing and managing requests

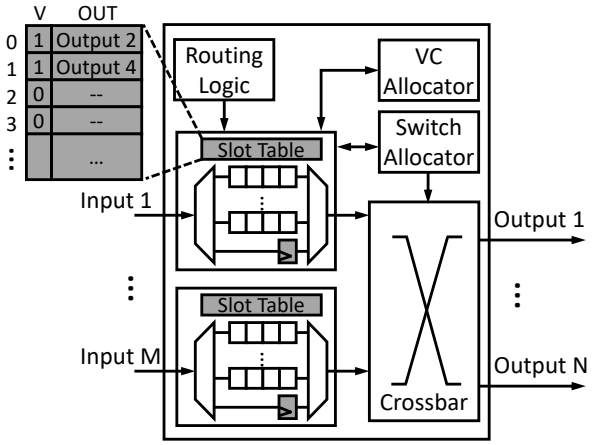


Fig. 2. Hybrid switching router microarchitecture.

when necessary. Requests from the same core are grouped and stored in an array, from which requests with smaller IDs are issued prior to those with larger IDs. Details of the NI microarchitecture are presented in Section III-D.

B. Main Network Microarchitecture

The beauty of a hybrid switching NoC is that, its circuit switching capability ensures deterministic message transmission latency, which is critical for ordering memory accesses. We use a hybrid switching NoC as our baseline network, and establish circuit switching connections from cores to ordering points.

1) TDM-based Hybrid Switching Router: A hybrid switching router supports both packet and circuit switching. Figure 2 shows the microarchitecture of a hybrid switching router, in which the components in grey (i.e., slot tables and latches) are for circuit switching. With the support of *slot tables*, the hybrid switching router allows both packet and circuit switching traffic to share the same communication fabric through time-division multiplexing [8]. A slot table has limited number of entries; and it is looked up by the allocation logic every cycle in modulo-fashion. A slot table entry indicates whether a particular cycle is reserved for a circuit switching data path (when the V bit is 1), or packet switching (when the V bit is 0). It also keeps track of the input-output port mapping for a circuit switching path (indicated by the OUT field). A flit is injected to either the packet switching data path (i.e., FIFOs) or the circuit switching data path (i.e., latch) from the source NI. Once the flit is on circuit switching data path, it is forwarded without buffering and routing at each router.

For example, in Figure 2, slot table entry 1 for input port 1 is valid, indicating that a circuit switching connection is established for this cycle. According to the OUT field, output port 4 is reserved for this connection. Therefore, if a circuit switching flit arrives in cycle $1+n\times S$ (S is the slot table size and n is an arbitrary number), it is then directly forwarded to output port 4 without performing route computation and arbitration. Assume link traversal takes 1 cycle, the flit will arrive at the downstream router in 2 cycles. Since buffering is not required for circuit switching flits, slot table entry 3 in the downstream router must be configured properly beforehand to accept and forward this flit. In a TDM-based hybrid switching router, if a time slot is reserved for circuit switching but a circuit switching flit does not present in that cycle, a packet switching flit is allowed to use this time slot. Such design greatly improves router utilization.

2) Slot Table Configuration: Slot tables are critical in circuit switching NoCs, because they ensure that flits are forwarded to the correct destinations without any contention during transmission. Slot tables can be configured either dynamically during run-time or statically at design-time. Dynamic configuration captures run-time traffic patterns and adapts well to traffic behavior changes. However, a dynamic approach usually requires a separate setup network and therefore adds additional hardware overhead to the NoC. Furthermore, the lack of global information makes dynamic slot table configuration less efficient–conflict happens when two circuit switching paths try to reserve the same slot table entry, which results in a setup failure and possibly infinite retries. As a result, circuit switching path reservation in a dynamic approach is not guaranteed to be successful. In contrast, a static approach programs the slot tables at design-time without introducing additional hardware overhead. However, slot allocations are usually fixed hence lacks flexibility. In our proposed design, we use the static approach for three main reasons. Firstly, the main purpose of using a circuit switching network is to guarantee deterministic message latency and message ordering, while flexibility and adaptiveness are less important. Secondly, our proposed NoC requires circuit switching connections for ALL source to destination communication pairs, which might not be satisfied by dynamic approaches. Thirdly, a static approach avoids setup latency, hence eliminates unnecessary communication delay.

We present a slot allocation algorithm in Algorithm 1. The first input parameter *comm_pairs* is the number of communication pairs for which circuit switching connections are required. For each communication pair, the path (essentially a list of links that messages traverse through) from the source to the destination is stored in the second input parameter $path[comm\_pairs]$. The output $starting\_slot\_id[comm\_pairs]$ is the starting time slot entry assigned to each communication pair at the source router.

Line 1 initializes a global slot table, keeping track of time slot assignment for all communication pairs. The g_slot_table is a 2D array, which has as many rows as there are in a slot table. The key idea behind this algorithm is the following. For each $path$, place its $link_id$s one after another to the corresponding rows inside g_slot_table. Globally, placing link id m in row n means that link m is assigned to a circuit switching connection in cycle n. Slot n should never be assigned to the same link m for another path, because having the same link ID appear in the same cycle indicates a conflict in circuit switching setup. If a conflict occurs between two paths, the latter path has to revert all its slot allocations and choose a new row from g_slot_table to start again, until all its links are placed without conflict. Every path starts placing its first link from the first row, as shown in Line 4. Line 8-19 attempt to place all links from a path into g_slot_table. When a conflict occurs, the failed path

Algorithm 1 Slot allocation algorithm

Input: comm_pairs, path[comm_pairs]
Output: starting_slot_id[comm_pairs]

```
1:  g_slot_table[ ][ ] ← {}
2:  starting_slot_id[comm_pairs] ← {0}
3:  for i = 1 to comm_pairs do
4:      starting_slot = 0;
5:      succeed = false;
6:      while (!succeed) do
7:          slot_id = starting_slot;
8:          for each link_id in path[i] do
9:              if (g_slot_table[slot_id].find(link_id)) then
10:                 roll_back(); //undo all changes for this path
11:                 succeed = false;
12:                 starting_slot++;
13:                 break;
14:             else
15:                 slot_table[slot_id].push_back(link_id);
16:                 succeed = true;
17:                 slot_id++;
18:             end if
19:         end for
20:     end while
21:     starting_slot_id[i] = starting_slot;
22: end for
```

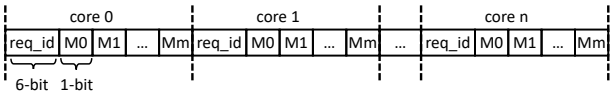
rolls back its previous slot allocations and starts a re-try from a new *starting_slot*, as shown in Line 10-13. If there is no conflict, the *link_id* is then pushed back to the corresponding row in *g_slot_table*, as stated in Line 15-17. In Line 21, the *starting_slot* is stored when slot allocation is completed for a communication pair.

Given the start cycle of a slot entry at the source, slot table allocation in each individual router for a circuit switching path can be easily calculated. The slot allocation algorithm has a time complexity of $O(N^3)$, where N is the number of nodes in the system. To connect all communication pairs through circuit switching, a larger system might need larger slot tables. In general, the minimum slot table size required to establish all circuit switching paths is proportional to the size of the multicore system.

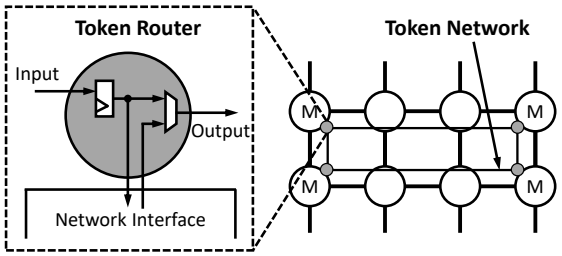
C. Token Network Microarchitecture

Token network is used to preserve memory access order among multiple ordering points. Ordering points are shared but distributed structures where memory accesses are being handled. For example, directories or memory controllers are considered as ordering points. Only one token is passed around all ordering points through this light-weight token network.

Token format. The token format is shown in Figure 3(a). For a system with *n* cores and *m* ordering points, the token contains *n* fields, and each field corresponds to an individual core. Inside each field, *req_id* indicates the ID of the memory request that is currently being processed by the ordering points. Followed by *m* valid bits (i.e., M_0, M_1, ..., M_m), each represents an



(a) Token format



(b) Token network and token router

Fig. 3. Token format and token network architecture.

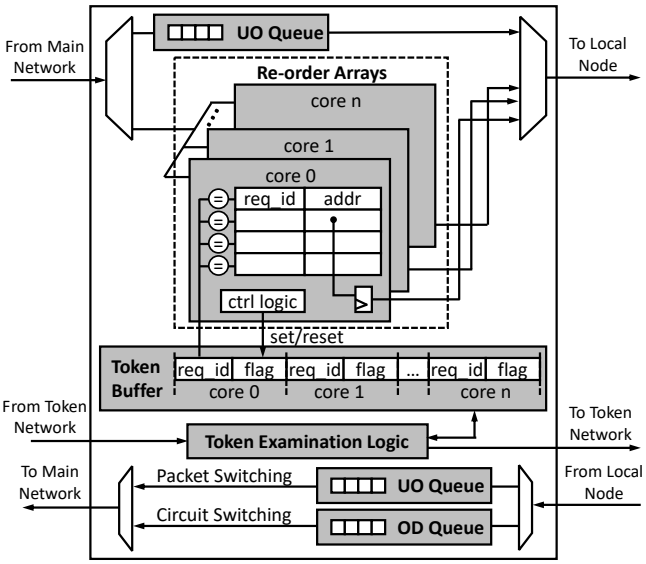


Fig. 4. Network interface microarchitecture.

ordering point. A valid bit is set when an ordering point finishes processing the memory request with the exact *req_id* indicated by the token. When a core executes a load/store instruction and generates new memory request(s), it increments its local request ID by 1 and embeds this ID with the request packet(s), which is(are) then sent to the ordering point(s). All memory requests will be buffered at the destination network interfaces. Then, the network interface at each ordering point examines the token and forwards requests to memory components when certain constraints are satisfied (details in Section III-D).

Token network. The token network is a unidirectional ring with extremely simple token routers. As shown in Figure 3(b), a token router receives the token from the upstream token router, forwards the token either to the NI or to the downstream token router, depending on whether the NI needs to examine the token. If the NI decides to accept the token, the token is removed from the token network. After processing, the NI updates the token and injects it back into the token network. In any given time, there is at most one token in the token network.

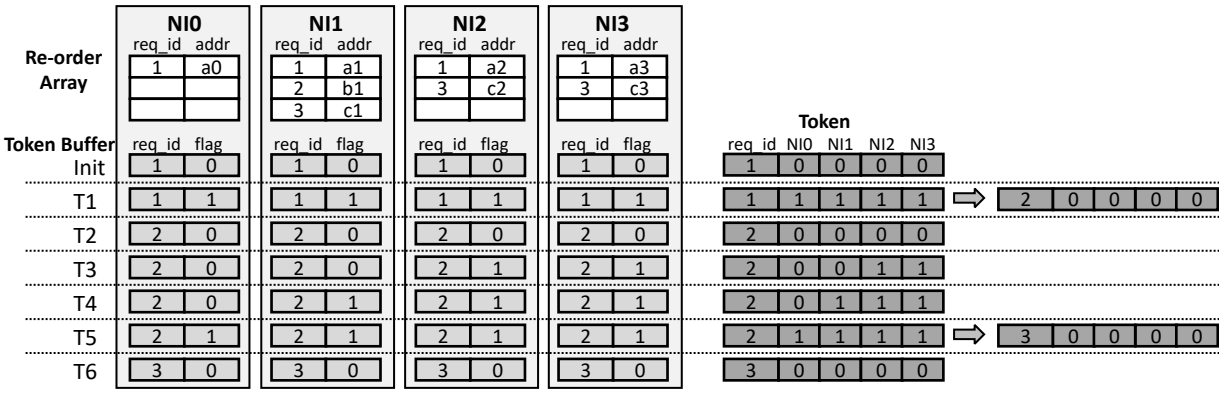


Fig. 5. Token network walk-through example.

D. Network Interface Microarchitecture

Network interface plays an important role in ordering memory accesses. As shown in Figure 4, the proposed NI contains an un-ordered (UO) queue, an ordered (OD) queue, re-order arrays, a token buffer, and a token examination logic, usages of which are explained below.

1) Sending and Receiving Process: At the source NI, messages that require to be ordered are buffered in *OD Queue* and sent through circuit switching, while the rest are buffered in *UO Queue* and send through packet switching. Ordered requests are injected into the main network when the reserved time slots are ready. Un-ordered requests are sent when the output port is free. Upon receiving, un-ordered requests are buffered in a first-in-first-out manner at the destination NI. Ordered requests are stored in different *Re-order Arrays* based on their source nodes, that is, ordered requests from core n are stored in the re-order array reserved for core n. An ordered request is retrieved from the re-order array by referring to its *req_id*, indicated by the *Token Buffer*. Token buffer is a structure that interacts with the token. Like the token, in an n-core system, a token buffer has n fields. Inside each field, *req_id* indicates the memory request that is being processed, which is just a copy from the token. When a request with a particular *req_id* is being served, the corresponding $flag$ bit in the token buffer will be set to 1.

2) Token Update Process: The *Token Examination Logic* is responsible for updating the token. When all valid bits (M_0-M_n) of a particular core inside the token are set, the *req_id* for that core will be incremented by 1, meanwhile all valid bits will be reset to 0. The token examination logic is also responsible for updating the corresponding fields in the token buffer, that is, when the logic detects that the token's *req_id* is larger than that of the token buffer, it updates the token buffer's *req_id* and resets the flag. Since the ordered requests are sent through circuit switching from the source, they are guaranteed to arrive in-order at the same destination. Therefore, a re-order array should see requests coming with ascending *req_id*s. This feature is important because it allows an NI to time-out when no further request is arriving (more details below). If the *req_id* in the token buffer is smaller than the smallest *req_id* observed from the re-order array, the NI will simply skip processing this request by setting the flag to 1.

3) Time-out Process: Time-out happens if the token's *req_id* is larger than the largest *req_id* observed from the re-order array. For example, the token's *req_id* is x, and the largest *req_id* in the re-order array of *NIk* is $x-1$. This means at least one ordering point other than *NIk* has received and started processing request x. While *NIk* is uncertain whether it will receive request x (x can be a unicast), it needs to start the time-out process to avoid starvation. The time-out window in our design is twice the number of slot table entries. Time-out is aborted if an NI receives a request whose ID is larger than the token's *req_id*. In the above example, *NIk* will stop the time-out process if it receives request $x+1$. This is because memory requests from the same core is guaranteed to arrive in-order. When *NIk* receives request $x+1$, it is ensured that request x will not appear in *NIk*.

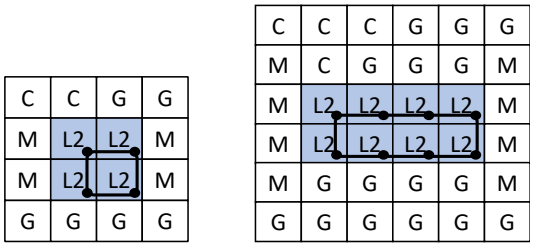
4) Walk-through Example: Figure 5 demonstrates how the proposed mechanism maintains memory access order. In this example, four ordering points are connected to the token network through NIs ($NI0-3$). For simplicity, we only show one re-order array in each NI, assuming only one core is sending requests. This core executes three memory instructions and generates three memory requests. The first request ($req1$) is a broadcast that requires accessing all four ordering points. The second request ($req2$) is a unicast that accesses only one ordering point. And the third request ($req3$) is a multicast that accesses three ordering points. The re-order array in $NI0$ stores only $req1$. In $NI1$, 3 requests $req1-3$ are waiting to be served. $NI2$ and $NI3$ both have 2 pending requests.

i) Initially, *req_id* is set to 1 and $flag$ is set to 0 in all token buffers. The token has *req_id* set to 1 and 4 valid bits corresponding to 4 ordering points set to 0.

ii) In T1, all ordering points finish serving $req1$, so $flag$ bits are set to 1 in all NIs. Meanwhile, the NIs examine and update the token, until all the valid bits inside the token become 1. Then one of the four NIs manipulates the token by setting *req_id* to 2 and all valid bits to 0.

iii) In T2, after the token has been updated, it continues circling in the token network and notifies the ordering points to process $req2$. Now all the token buffers have the latest information.

iv) Note that $req2$ only presents in $NI1$, therefore the other NIs can skip this request. Both $NI2$ and $NI3$ see requests with larger *req_id*s inside their re-order arrays. Since requests



(a) 16-node System (b) 36-node System

Fig. 6. Evaluated multicore systems.

are guaranteed to arrive in-order through circuit switching (Section III-D2), *NI*2 and *NI*3 can skip *req*2 by setting $flag$ to 1, as shown in T3. When the token reaches *NI*2 and *NI*3, its valid bits will be set to 1 by the token examination logic.

v) In T4, *NI*1 sets its flag to 1 after completing *req*2. However, until now, $flag$ bit in *NI*0 is still 0. This is because *NI*0 is still waiting for *req*2 or a request with a larger *req_id*. Remember the arrival of circuit switching messages are deterministic, the NI knows when a message from a particular node is to arrive by referring to the slot table. Therefore, after noticing some of the valid bits in the token are set, *NI*0 starts to time-out .

vi) In T5, *NI*0 sets $flag$ to 1 after the time-out window and manipulates the token. All the ordering points should start processing *req*3.

vii) In T6, all token buffers are updated and the above described process repeats.

TABLE I
SYSTEM CONFIGURATION AND NETWORK PARAMETERS

CPU Configuration	
Processor	Single-issue in-order
L1 Cache	Split private I/D caches, each 64KB, 4-way set associative, 64B block size, 1-cycle latency
GPU Accelerator Configuration	
Accelerator	32-wide SIMD pipeline, 1024 threads, 32KB shared memory
L1 Cache	Split private I/D caches, each 16KB, 4-way set associative, 128B block size, 2-cycle latency
Memory Configuration	
L2 Directory Cache	Shared banked, 2MB/bank, 8-way set associative, 128B block size, 8-cycle latency
Memory	4GB DRAM, 200 cycle access latency
Main Network	
Topology	16/36-node, 2D-Mesh
Router	2 stage, 4 VCs/port, 5 buffers/VC, Minimal Adaptive Routing
Channel Width	32 Bytes
Packet Size	4 flits (circuit switching packet) 5 flits (packet switching packet)
Slot Tables	16-node: 16-entry slot table 36-node: 32-entry slot table
Token Network	
Token Width	16-node: 80 bits 36-node: 336 bits
Link Width	16 Bytes

IV. EVALUATION

In this section, we describe the evaluation infrastructure and present the experimental results, followed by a discussion on the flexibility and limitations of the proposed network.

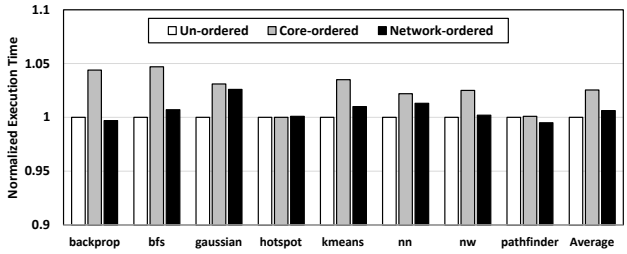
A. Evaluation Infrastructure

We use gem5-gpu [10] as our evaluation infrastructure. Simulator parameters can be found in Table I. Figure 6 shows the topology of the evaluated systems. The 16-node (4-by-4 mesh) system contains 2 CPU cores, 6 GPU cores, 4 L2 cache banks shared between CPU and GPU, and 4 memory controllers connected to off-chip memories. CPU and GPU share the same memory address space. MESI protocol is used to keep both CPU and GPU caches coherent. Directories are located at L2 caches (shaded in figure), whose NIs are connected to the token network. Similarly, the 36-node (6-by-6 mesh) system has 8 directory nodes connected to the token network. We augment the baseline Garnet [11] network with a TDM-based circuit switching network as well as a token ring network. Token width is dependent on the number of cores and directories. In the 36-node system, limited by the token network's link width, the token must be sent in multiple cycles. Therefore, the NI is responsible of breaking the token into multiple flits when sending and reassemble the token when receiving. The main network and token network configuration parameters are also listed in Table I.

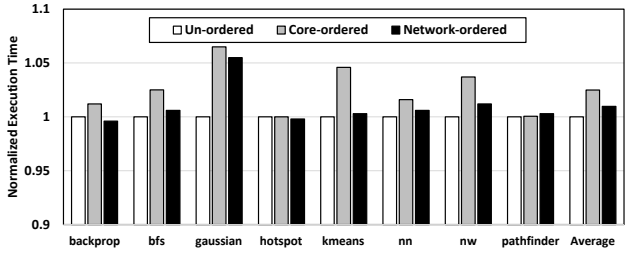
We use CUDA applications from Rodinia benchmark suites [12] as accelerator workloads. Explicit memory copies between CPU and GPU are removed from the code and pointers are used instead. The evaluated workloads include: *backprop*, *bfs*, *gaussian*, *hotspot*, *kmeans*, *nn*, *nw*, and *pathfinder*. We execute only one application each time on one CPU, which then launches kernels across all GPU cores.

B. Circuit Switching Decision

Circuit switching all memory requests is unnecessary and can cause performance penalty. For read/write requests accessing private data, or read requests accessing read-only shared data, sending them with un-ordered packet switching network is safe. However, write requests that access shared data must be ordered properly. Prior work have proposed memory management unit and OS page protection mechanisms to classify memory blocks into private and shared [13]–[15], so that access safeness can be determined. In this work, we use a similar mechanism as proposed by Cuesta *et al.* [14] to decide whether to circuit switching a write request or not. In particular, we extend the page table entries with a core identifier (CID) field, a read-only (RO) bit, and a shared (SH) bit. Memory accesses to a page with RO bit unset and SH bit set need to be ordered. TLB entry is extended with a lock (LK) bit. When LK bit is set, all write accesses must be sent through circuit switching to maintain the order. Upon a TLB miss, the TLB miss handler refers to the page table and checks the ownership (CID) of the page. If the page is owned by another core with RO bit unset, the TLB miss handler allocates a TLB entry and sets its LK bit. Meanwhile, if the SH bit is also unset, a notification is generated and sent to the owner of this page, which will then set the LK bit in the owner's TLB entry. Such notification is



(a) 16-node system



(b) 36-node system

Fig. 7. Program execution time (normalized to the Un-ordered system).

not required when TLB miss handler finds the SH bit set in the page table entry, since the combination of RO bit unset and SH bit set means all the sharers already have their LK bits set in TLB entries. Notice that in GPGPU applications, CPU initializes shared data before a kernel launch, and usually it does not access the data until kernel completion. Therefore, CPU should not update the CID field in page table entries during initialization. Once initialization is finished, a CPU TLB shootdown is required, such that later accesses to the shared data can be captured properly.

C. Experimental Results

For evaluation, we compare our proposed mechanism against 1) a baseline system, which uses only packet switching NoC and does not maintain any ordering, referred to as **Un-ordered**; and 2) a system using only packet switching NoC and maintains total store order (TSO) in each core, named as **Core-ordered**. TSO guarantees that the sequence in which store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor [16]. To support TSO, we provide a FIFO write buffer for each core. The write buffer must be drained before each atomic instruction and memory fence instruction [17]. Our proposed work is referred to as **Network-ordered** in the evaluation.

Figure 7 shows the normalized execution time for 16-node and 36-node systems. In 16-node system, Network-ordered system degrades the performance by 0.7% on average, in comparison to Core-ordered which incurs 2.6% performance penalty. In 36-node system, execution time is increased by 1.0% on average in Network-ordered system, in comparison to 2.5% for Core-ordered system. Overall, enforcing memory access order inside the network has insignificant performance impact. This is mainly because the parallelism in GPGPU applications is capable of hiding memory access latency so that long access delay is tolerable. The second reason is that SIMD accelerators are simple in-order cores, which leads to fewer optimization

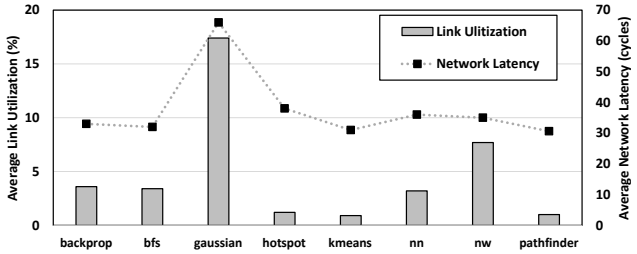


Fig. 8. Average link utilization and network latency for 36-node system.

opportunities in terms of re-ordering memory operations to begin with. Network-ordered system performs better than Core-ordered system because write requests from a single core are parallelized, in other words, the core does not need to stall and wait for previous write requests to complete.

Throughput intensive applications such as gaussian, nn, and nw suffer noticable performance degradation in Network-ordered system. Figure 8 shows the average link utilization and average network latency for 36-node configuration. The proposed network forces some write requests to be transferred in order through circuit switching, which under-utilizes the available on-chip bandwidth. When traffic injection rate is low, circuit switching can even reduce packet transmission latency, which in turn improves the overall performance for some applications (e.g., backprop, pathfinder). However, when cores begin to generate considerable amount of outstanding requests, circuit switching can impact the performance because it takes up network bandwidth from packet switching.

D. Area and Power Overhead

We use DSENT [18] to estimate the area and power consumption of the proposed design. We consider 22nm technology node and assume the NoC operates at 1.5GHz. As a reference point, for the 36-node system described in Section IV-A, the hybrid switching router is $0.27mm^2$, with a power consumption of $0.31W$. The area and power consumption of a token router are $0.00085mm^2$ and $0.00129W$, respectively. To support message ordering in the 36-node system, assuming 64-bit memory address, the area and power overhead (UO/OD queues, re-order arrays, and token buffer) added to an ordering point NI are $0.19mm^2$ and $0.23W$, respectively.

E. Discussions

While the proposed network can potentially provide a strong memory model with minimal performance overhead, it is based on several assumptions. First, we assume that the optimizations performed during compile time and kernel launch time do not violate program order. However, optimizations such as common subexpression elimination and dead code elimination might affect the access to shared data, which in turn violates the program order. Second, we assume the underlying processors are SIMD-like cores and applications exhibit good thread-level parallelism, and CPUs are less active during kernel execution. Enforcing memory access order in superscalar out-of-order CPU cores can be more expensive due to lacking parallelism to hide long access latencies [1], [19].

V. Related Work

Hybrid switching NoCs have been studied thoroughly in both homogeneous and heterogeneous multicore systems. Spacial-division-multiplexing-based (SDM-based) hybrid switching NoCs have been proposed to provide QoS support in SoCs [20]–[22]. Enright Jerger *et al.* [23] use SDM-based hybrid switching NoCs to improve the performance of coherence-based traffic in CMPs. TDM-based circuit switching provides bandwidth and latency guarantees in Æthereal [24] and NOSTRUM NoC [25]. More recently, TDM-based hybrid switching NoCs for heterogeneous multicore and accelerator rich architectures have been proposed [8], [9], [26], [27]. These works mainly focus on reducing NoC energy consumption and are only able to provide best-effort circuit switching communication. Our proposed slot allocation algorithm guarantees all computation cores are connected to ordering points through circuit switching.

SCORPIO [28] supports global ordering of requests on a mesh network for snoopy coherence. SCORPIO consists of an un-ordered packet switching main network for broadcasting coherence messages, and a fixed-latency bufferless notification network to achieve distributed global ordering. SCORPIO is demonstrated to be efficient in CMP systems that contain only CPU cores. However, in heterogeneous multicore systems, snoopy coherence is less efficient than directory-based coherence. Moreover, maintaining global ordering for wide GPU data requires more buffering resource at the NIs.

One approach to preserve the memory access order is to rely on a recorded sequence of requests and use network information theory concepts to broadcast network coded messages. Xue *et al.* [29] propose a network coding-based NoC architecture to improve the NoC performance for multicasting. Duraisamy *et al.* [30] propose a wireless NoC architecture to handle high volumes of multicast injections. These techniques can be applied on top of our proposal to improve system performance.

Hechtman and Sorin [17] revisit hardware memory models in massively-threaded throughput-oriented processors like GPGPUs and observe that sequential consistency achieves the same performance compared to weak consistency models on a variety of applications. While similar observation is made in our work, we focus on heterogeneous multicore systems that contain both CPU and GPU cores. Moreover, we propose to push the responsibility of enforcing memory consistency to the network so that performance critical hardware modifications at cores can be avoided.

VI. Conclusion

Future computer systems can be extremely heterogeneous, with various processors, accelerators, programmable logic, and even heterogeneous memories such as DRAM and non-volatile memory integrated into the same package [31]. Programming and debugging in such a heterogeneous environment can be challenging. In this work, we proposed an in-network memory access ordering mechanism, which supports a stronger memory consistency model than relaxed consistency and allows programmers to easily reason about their program execution. Evaluation results show that the proposed design brings negligible impact to overall performance. We believe the proposed NoC is an important step towards exploiting memory-level parallelism while maintaining programmability for heterogeneous architectures.

References

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, Dec. 1996.

[2] P. Sewell *et al.*, "X86-TSO: A rigorous and usable programmers model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, Jul. 2010.

[3] "Hsa platform system architecture specification 1.1," http://www.hsafoundation.com/?ddownload=5114.

[4] J. Yin *et al.*, "Modular routing design for chiplet-based systems," in *ISCA 2018*.

[5] M. Taylor *et al.*, "Scalar operand networks: on-chip interconnect for ilp in partitioned architectures," in *HPCA 2003*.

[6] S. Vangal *et al.*, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, Jan 2008.

[7] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, Sep. 2007.

[8] J. Yin *et al.*, "Energy-efficient time-division multiplexed hybrid-switched noc for heterogeneous multicore systems," in *IPDPS 2014*.

[9] J. Cong *et al.*, "On-chip interconnection network for accelerator-rich architectures," in *DAC 2015*.

[10] J. Power *et al.*, "gem5-gpu: A heterogeneous cpu-gpu simulator," *Computer Architecture Letters*, vol. 13, no. 1, Jan 2014.

[11] N. Agarwal *et al.*, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS 2009*.

[12] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC 2009*.

[13] N. Hardavellas *et al.*, "Reactive nuca: Near-optimal block placement and replication in distributed caches," in *ISCA 2009*.

[14] B. A. Cuesta *et al.*, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *ISCA 2011*.

[15] A. Singh *et al.*, "End-to-end sequential consistency," in *ISCA 2012*.

[16] S. Owens *et al.*, "A better x86 memory model: X86-tso," in *TPHOLs 2009*.

[17] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," in *ISCA 2013*.

[18] C. Sun *et al.*, "DSENT - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *NOCS 2012*.

[19] S. Sarkar *et al.*, "Understanding power multiprocessors," in *PLDI 2011*.

[20] F. Palumbo *et al.*, "Concurrent hybrid switching for massively parallel systems-on-chip: The cyber architecture," in *CF 2012*.

[21] M. Modarressi *et al.*, "A hybrid packet-circuit switched on-chip network based on sdm," in *DATE 2009*.

[22] A. Lusala and J. Legat, "Combining sdm-based circuit switching with packet switching in a noc for real-time applications," in *ISCAS 2011*.

[23] N. Enright Jerger *et al.*, "Circuit-switched coherence," in *NoCS 2008*.

[24] K. Goossens *et al.*, "Aethereal network on chip: concepts, architectures, and implementations," *IEEE Design Test of Computers*, vol. 22, no. 5, pp. 414–421, 2005.

[25] M. Millberg *et al.*, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *DATE 2004*.

[26] A. Biswas, "Efficient timing channel protection for hybrid (packet/circuit-switched) network-on-chip," *TPDS*, vol. 29, no. 05, may 2018.

[27] S. Hesham *et al.*, "HPPT-NoC: A Dark-Silicon Inspired Hierarchical TDM NoC with Efficient Power-Performance Trading," *TPDS*, vol. 31, no. 3, 2020.

[28] B. Daya *et al.*, "Scorpio: A 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering," in *ISCA 2014*.

[29] Y. Xue and P. Bogdan, "User cooperation network coding approach for noc performance improvement," in *NOCS 2015*.

[30] K. Duraisamy and other, "Multicast-aware high-performance wireless network-on-chip architectures," *VLSI*, vol. 25, no. 3, 2017.

[31] S. Che and J. Yin, "Northup: Divide-and-conquer programming in systems with heterogeneous memories and processors," in *IPDPS 2019*.