Supporting Sequential Consistency through Ordered Network in Many-Core Systems

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Hariharasudhan Venkataraman

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Professor Antonia Zhai

December, 2017

Acknowledgements

First and foremost I would like to thank my advisor, Professor Antonia Zhai for giving me an interesting idea to work on and her guidance on doing research. She has been very supportive throughout my research experience and enabled me to think in the right direction and also helped me construct the problem concretely and in writing quality paper. I have gained immense experience in working independently at the same time collaborating and brainstorming ideas with her students as well.

I am grateful to Jieming Yin for all the invaluable assistance with the infrastructure set up and some useful insights through emails without which it would have been extremely difficult for me to complete my project. I thank my lab members and friends: Minjun Wu, Wenwen Wang, Kartik Ramkrishnan, Vinoth Selvan, Vignesh Balaji and Shashank Hegde for all the advices while working on my research. I also thank my final thesis committee members, Professor Pen Chung Yew and Professor Gerald Sobelman for their feedback and guidance.

Last but not the least, I offer my sincere gratitude to my parents and my sister for the constant encouragement throughout my graduate studies supporting me both morally and financially.

Abstract

Recently, there are two trends in parallel computing. On one hand, emerging workloads have exhibited significant data-level parallelism; on the other hand, modern processors are increasing in core count to satisfy the increasing demand of processing power under stringent power and thermal constraints. Hence, multi-core and many-core systems have become ubiquitous. To facilitate software development on such processors, it is desirable to efficiently support an intuitive memory consistency model, such as the sequential consistency model.

In this work, we demonstrate the feasibility of supporting the sequential memory consistency model on many-core systems. Our experiments show that in many-core systems where in-order cores with no private caches and shared memory modules are connected with a 2D-mesh network that supports circuit-switching, we are able to efficiently support sequential memory consistency by ordering memory requests in the network. In this work, memory requests are ordered by time-stamping each memory request and circulating a token among the memory modules. Furthermore, we extended the mechanism for ordering memory traffic in network to speed-up the performance of critical sections. We evaluated the proposed techniques on three different many-core systems that contain 8, 20 and 32 cores respectively. Compared to conventional systems where sequential consistency is supported by serializing memory requests at the cores through fences, the proposed systems are able to outperform the conventional systems by 4.95% , 5.74% and 9.70% respectively on the three different many-core systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In todays world of computing, there is a need for increasing the processing power under stringent power and thermal constraints to satisfy the demand of computing. Transitioning from a uniprocessor system, the exploitation of data and thread level parallelism has led to the advent of increasing the core count on a single die. Hence, multi-core and many core systems have become ubiquitous and programmers must efficiently adapt to writing programs on parallel systems. Shared memory models with a unified address space is an ideal option to adopt for programming these architectural models since it reduces concerns of data partitioning, load distribution, etc. for the programmers. The programmer needs to know the precise behavior of the memory, in particular what to expect when reads and writes happen concurrently on different processors. Typically, programmers expect a sequentially consistent memory behaviors even on a parallel system. However, implementing sequentially consistent memory model impacts performance since its implementation requires to forgo many hardware related optimizations. Thus, weaker memory consistency models are predominantly implemented at the hardware level [8]. Such memory models require that the programmers imbibe a very cautious approach in writing programs to avoid any counter intuitive results that can be caused due to weak behavior of these models.

As core count increases, the complexity and performance impact involved in supporting sequentially consistent models increases significantly. A large class of many-core accelerators [11, 14, 22, 3] and in-memory computing systems [23] consists of many cores and assume a shared memory model but their memory consistency models are not well defined especially with architectural heterogeneity present in these systems. These issues have been addressed in recent research [10] and we are aware of the difficulties in bringing about solutions and solving memory consistency problems.

Existing multi-core systems support memory consistency behaviors by enforcing a memory order on consecutive memory operations within the same core. Unfortunately, this approach can significantly limit memory-level parallelism. Thus we propose to build a sequentially consistent memory model at the hardware level which provides a global memory order by implementing an ordered network in a multi-core system with a shared cache existing as multiple memory modules.

When there are many cores and we have one shared cache but present as multiple memory banks at different nodes in a network topology the cores can issue memory operations simultaneously and multiple writes from a core can be serviced by different memory modules simultaneously. Eliminating private caches can simplify coherence but it does not guarantee a sequentially consistent memory model. In this scenario the interconnect that is responsible for the routing of the data between cores and memory does not guarantee a memory order even if the cores process instructions in order, considering we provide no room for caching any shared data by elimination of private caches. If we could provide some mechanism of ordering at the NoC to ensure all the memory accesses obey a global memory order amongst the multiple memory nodes and the cores we can guarantee sequential consistency.

In current multicore and manycore systems on a mesh based NoC, packet switching network is the de-facto choice because they are scalable and flexible [17, 18]. If the NoC does not provide an order of packet transmission between the cores and memory, there is no guarantee of write atomicity which means writes can get overlapped and thus,

not guaranteeing sequential consistent memory model by the hardware. We eliminate the usage of fences and guarantee sequential consistency at the hardware level with in-order cores and thus providing an ordered network. The following example illustrates a situation that can occur due to an unordered network.

1.1 Sample Illustration

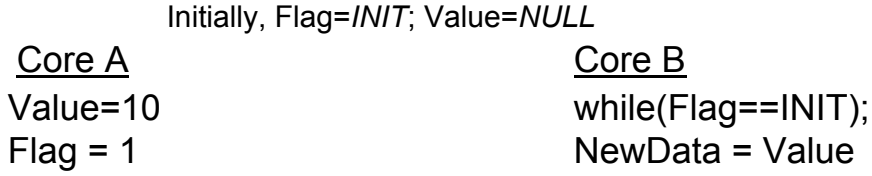

Figure 1.1: Scenario depicting writes getting overlapped

Consider the code snippet shown in the Figure 1.1. We show two cores accessing shared variables namely `Flag` and `Value`. Initially it is assumed `Flag` is a `NULL` and `Value` has already been assigned a value called `INIT`. Now Core A does two writes where it updates `Value` to `10` and `Flag` to `1`. While Core B reads the value `Flag` and spins till `Flag` gets updated to `1` and then assigns the value, `Value` to a variable `NewData` which is essentially a read of `Value` which is written to the variable `NewData`. Logically we expect `NewData` to get assigned value, `Value` once updated as `10` by Core A. A sequentially consistent system guarantees this but there exist some indeterminism due to the interconnect and this can violate sequential consistency. Weaker memory models cannot guarantee a sequentially consistent behavior. We need to enforce fences between the two writes to avoid sequential consistency violation. We shall demonstrate how a wrong outcome can result due to the unordered interconnect.

Let us assume `Flag` and `Value` are stored in different memory modules and the location of `Value` is far away from Core A compared to the memory location of `Flag` by

which we mean that the network latency to access `Value` is longer than accessing `Flag`. So, even if the writes are issued in order, even before `Value` from Core A gets written to the location, `Flag` could be updated from Core A since it's memory update is serviced before the update of `Value` due to its closer proximity to Core A. This implies that if Core B reads `Flag` then there is a possibility it can read the wrong value of `Value` and assign it to `NewData`. This clearly violates the semantics of sequential consistency. This is because the two writes from Core A is overlapped and the order is not maintained by the interconnect and in fact the violation is caused due to the unpredictability in terms of delay in the network when data is being routed.

One way to solve this is using a hardware-sequentially-consistent model where there is ordering guaranteed at the cores. That means, in this case till Core A sees that the write of `Value` is completed, it does not write `Flag`. However, this is inefficient since we can clearly see that we are virtually stalling Core A's execution. However, if we ensure that they are ordered at the network, which means Core A can proceed with its execution but Core B will execute as expected because at the network we guarantee that `Value` gets written first before `Flag`. This is the scenario we tackle by ensuring writes do not overlap and do not result in non-intuitive outcomes. This exist when operations from the same core may be serviced by different memory modules and there is no control in the network to ensure ordering of packets even if memory requests are issued in program order by the cores.

The following figure 1.2 graphically depicts the advantages of ordering at the network rather than stalling the core explained earlier. We see that the Core A is allowed to proceed with its execution and also ensures no sequential consistency violation. So by ensuring the writes complete in program order, we can avoid any non-intuitive results. In the Figure 1.2 we demonstrate the impact we believe to gain when the memory accesses are ordered at the network level by our proposed technique.

Here, we develop an idea of implementing sequentially consistent hardware in a relatively efficient manner so that programmers can have a better confidence of writing

parallel programs. We try to avoid any usage of hardware fences and propose a method of ordering memory access that would enable this.

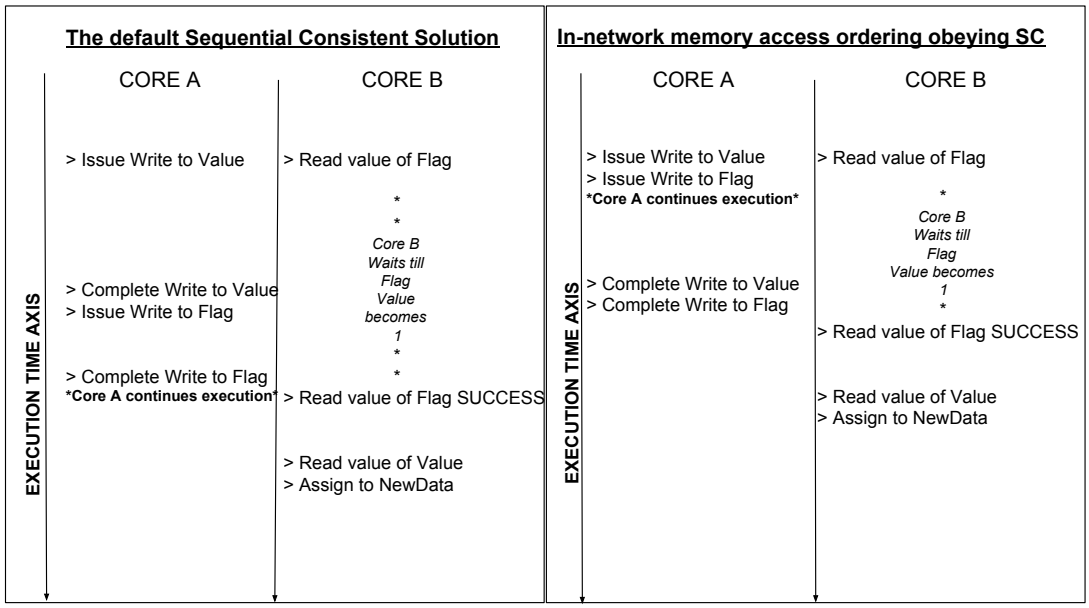

Figure 1.2: The flow of execution is depicted and both the figures ensure write atomicity. While the first is a default method of enforcing sequential consistency (SC), the second shows the benefit of ordering at the network over the default implementation

1.2 Dissertation Outline

The organization of the rest of the thesis is as follows:

- Chapter 2 briefly describes implementing sequential consistency (SC) with an ordered network along with the necessary infrastructure in order to build the same. We briefly discuss the architectural system on top of which we implement SC.

- In Chapter 3 we give a detailed description on how we set up circuit switched paths and how we implement the ordering of memory requests using the circuit

switched paths. We also explain the mechanism of ordering memory requests for critical section and a non-critical sections of a program.

- In Chapter 4 we show our evaluation methods by initially showing the benefits of ordering in network using micro benchmarks where we distinguish for critical sections and non-critical sections. Then we analyze the results and use benchmarks to test our proposed idea and discuss the results in detail.

- Chapter 5 mentions related work and distinguishes these works with the proposed idea of memory ordering

- Chapter 6 concludes the dissertation talking about the scope and further improvements possible on this type of architectural design.

Chapter 2

Sequential Consistency with In-order Network

Our main focus is to establish a way to order memory accesses at the interconnect level to make the expected order of memory accesses to follow a global scheme of memory order so that it does not violate program order the cores. Instead of using fences that can ensure that the writes and the reads are visible to all the processors, we provide a method by which the memory accesses are ordered by the network interfaces of the memory nodes which guarantees that the order of memory requests (a load/store) issued by the processor is satisfied in the same order. This in-network memory access ordering will engender some amount of determinism so that the programmers can expect the same effect of fences by actually not worrying about the usage of fences to avoid sequential consistent violations.

We show how to implement a sequentially consistent hardware with the use of in-order cores and an ordered network. Firstly, we discuss in detail about the architecture and how we implement an ordered network using circuit switching on a packet switched fabric. We then propose a mechanism of ordering memory access at the interconnect level using this ordered network. We essentially compare two sequentially consistent

models where one involves in-network memory access ordering and the other ensures write atomicity by applying fences when needed.

2.1 Base Architecture

We are very much aware that the caches manage coherence between each other to ensure a single writer multiple reader invariant criterion. However, the hardware complexities make it necessary to define a memory model so that we can expect behavior of a program's execution, this behavior comes from the existence of: write buffers, multiple memory nodes, out of order execution, hardware pre-fetching and other speculative execution in a shared memory model. We take a simpler approach focusing on using simple cores, simpler memory hierarchy in a multi-core system that provides SC by providing the memory ordering implemented at a network level.

To simplify things, just as in relaxed memory models fences are used to implement data race free programs. This makes these models look like sequentially consistent, we try to implement SC at hardware by not using any private L1 caches and use just one shared cache that is present as multiple memory nodes at the network and for this we ensured by having no L1 caches modifying the coherence protocol to just have simple coherence states present only in the shared cache. Our goal is to establish SC at a hardware level by establishing a memory order at the network. Essentially we eliminate any coherence states that need to be maintained due to the presence of private caches and hence we just have one level of cache that is shared by the processors. This can benefit the programmer who need not think of inserting fences as this design provides a more efficient SC implementation. We do not use any pre-fetching and speculation in our design and we also, do not have store buffers separately. However, there are load-store queues that are responsible for queuing memory packets to the network. This approach aids establishing SC and depicting the performance of our proposed design of ordering memory packets at a network level.

We intend to develop a scalable design so that we generate scope for future architectural design space to use the idea we propose in this paper. We consider multiple cores that can exploit parallel processing and intend to implement memory ordering at a interconnect level. We consider an interleaving topology that basically consist of multiple cores, shared cache memory modules and the memory controllers communicating with the main memory. We address the problem of serialization by not using a bus-based design but, using multiple memory modules in our architectures promotes the ability of servicing multiple memory accesses simultaneously.

2.2 Circuit Switched Network

Packet switching network networks do not guarantee any ordering of messages at the network and as a result they may result in uncertainties while transmitting messages when there are multiple messages coming from different cores that is being services my different memory nodes. In the Figure 2.1 we can see that circuit switched paths are providing dedicated channels from a source to destinations while packet switching is done by constant communication with the routers present in the network. Packet switching involves buffering and routing of these packets which can cause significant delays depending on the traffic at a particular router and does not guarantee any order at which memory requests arrive at the memory nodes while circuit switched routing enables the message transmission ordering[6]. Due to this we use circuit switching to preserve an order of memory accesses requested by the cores. So we use an infrastructure that provides circuit switching on a packet switching fabric like the outline shown in Figure 2.1.

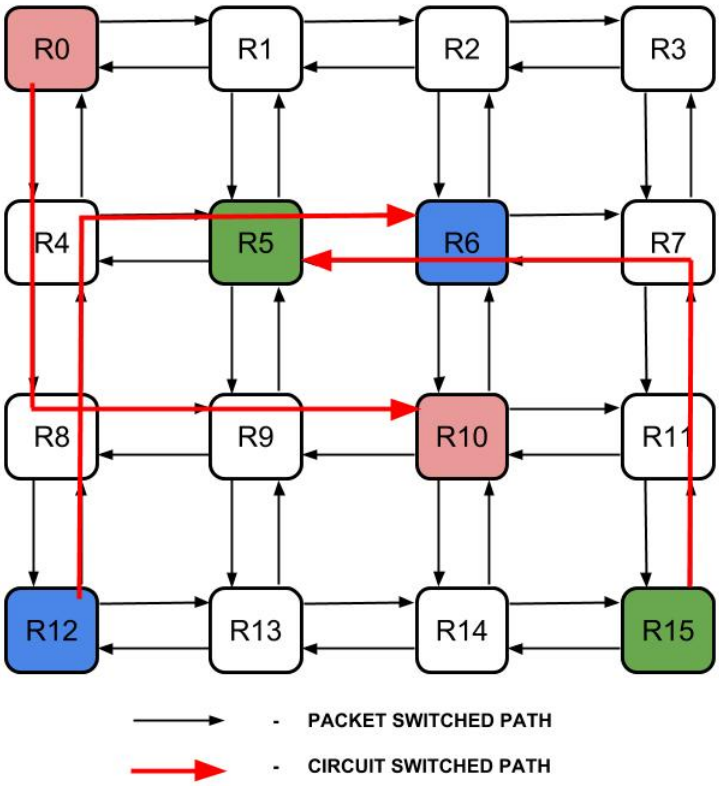

Figure 2.1: Providing circuit switching and packet switching on one framework

We provide a sequentially consistent model by using circuit switching where we set up paths from a source (cores) to the destinations (memory nodes) thereby we establish dedicated paths to service memory requests. We set up circuit switching paths over a packet switching mesh based NoC. We use the infrastructure of the Hybrid-switched NoC[21] to enable ordering at the network. We use a simple technique by using light-weight tokens to establish the ordering of memory accesses at the network which we briefly describe in the next section. We shall elaborate on the setting up circuit switching paths in the next section.

2.3 Ordering Messages using tokens

This mechanism of using a token ring network has been utilized before [20] but we use CPU cores instead of deploying this mechanism for GPU cores as done before. Moreover, we modified this technique to use for the CPU cores where we also do the ordering for critical sections which is different from doing it for non critical sections.

We use tokens which has a metadata from the every memory requests from every core being serviced. These essentially consists of IDs of memory request that the core generates. So every core generates this ID locally and sends to the network. These are circuit switched along with the memory request specific to it. We use tokens to route into a ring like network that are transmitted to all the memory nodes which we call ordering points where the ordering of memory request is maintained. Though the requests from the cores arrive in order, without the above mentioned infrastructure ordering cannot be guaranteed by the network. The network interface controllers present at these ordering points are responsible for managing these token data and ensuring order. They do this by grouping token data from one core by storing them in a cache like array structure by which they can queue the messages in order at the ordering points.

Essentially with the metadata from the core and the token management done by the network interface controllers, we maintain an order which ensures that smaller IDs are serviced before larger IDs always for every core. Thus guaranteeing that the global memory order follows the program order. Figure 2.2 shows how token ring network exist over the memory system. We explain in detail in the next section how the ordering mechanism is implemented.

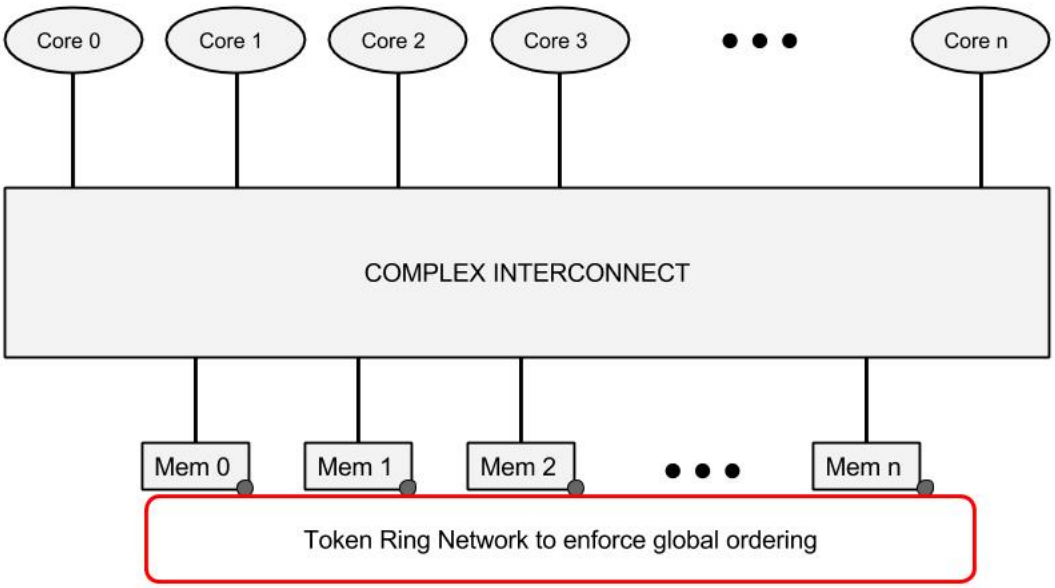

Figure 2.2: Token ring framework

Chapter 3

Design Details

In the section we provide a detailed description of the infrastructure built for enabling in-network memory access order. To demonstrate this idea we assume just one cache for the entire network which is shared and multiple CPU cores can access the cache in parallel and this shared cache is having different banks sitting on different nodes. So the upshot is that there are multiple cache nodes in the network which are shared between all the CPU cores and there are no private data caches for the CPU cores. Also, we use circuit switching to implement the ordering. We elaborate how we set up circuit switching paths on an existing packet switching network. We then explain how the ordering is done by passing the `ReqID` and `CoreID` from the CPUs to the network interfaces.

3.1 Setting up Circuit Switching Paths

Circuit switching network is necessary to do the ordering between packets efficiently since the links between the source and destination is determined and hence the path is set up deterministically. Here the idea is to deploy circuit switching paths for the ordering purposes. Circuit switching is cost efficient if the traffic is more throughput intensive otherwise it leads to under utilization of resources on the network. The cost

of setting up these paths are high however, once set up, these can reduce considerable network delays over packet switching especially, considering workloads in which each cores do similar work because the paths once set up to destinations from the cores will be utilized at similar instants when cores do similar parallel work.

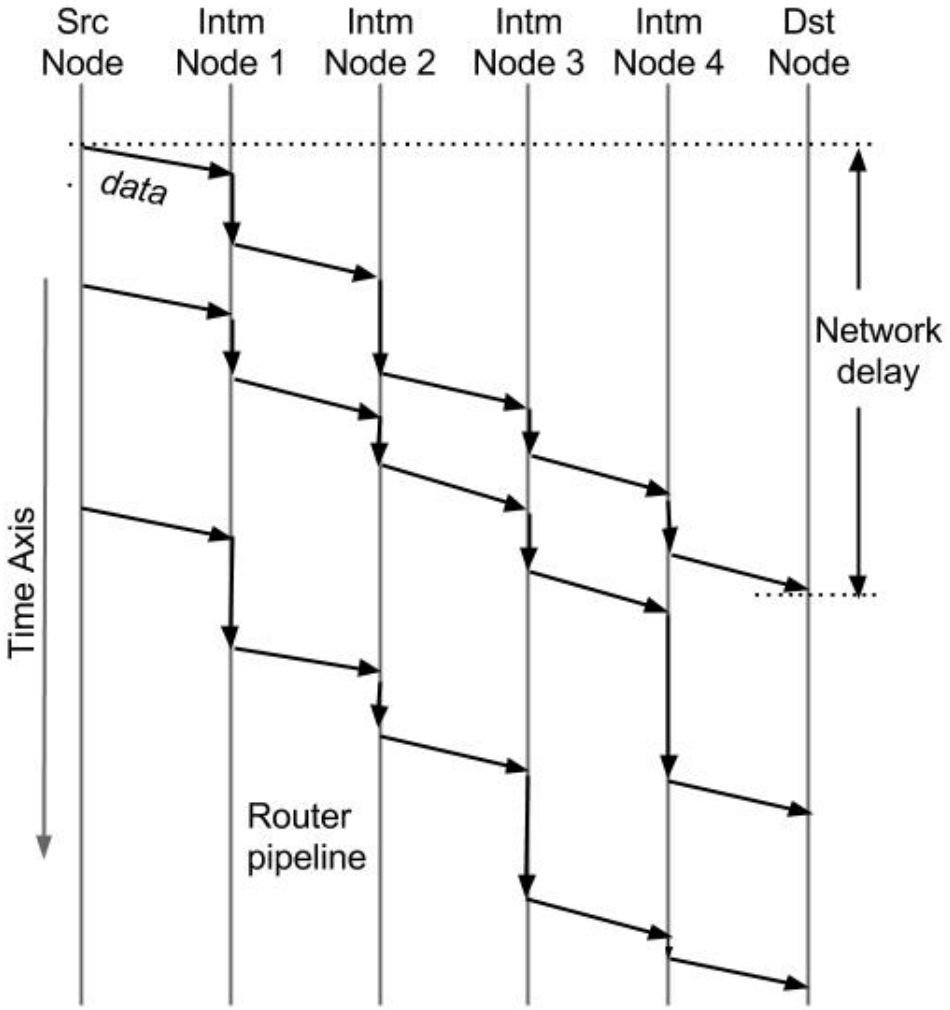

Figure 3.1: Flow of a packet switching network

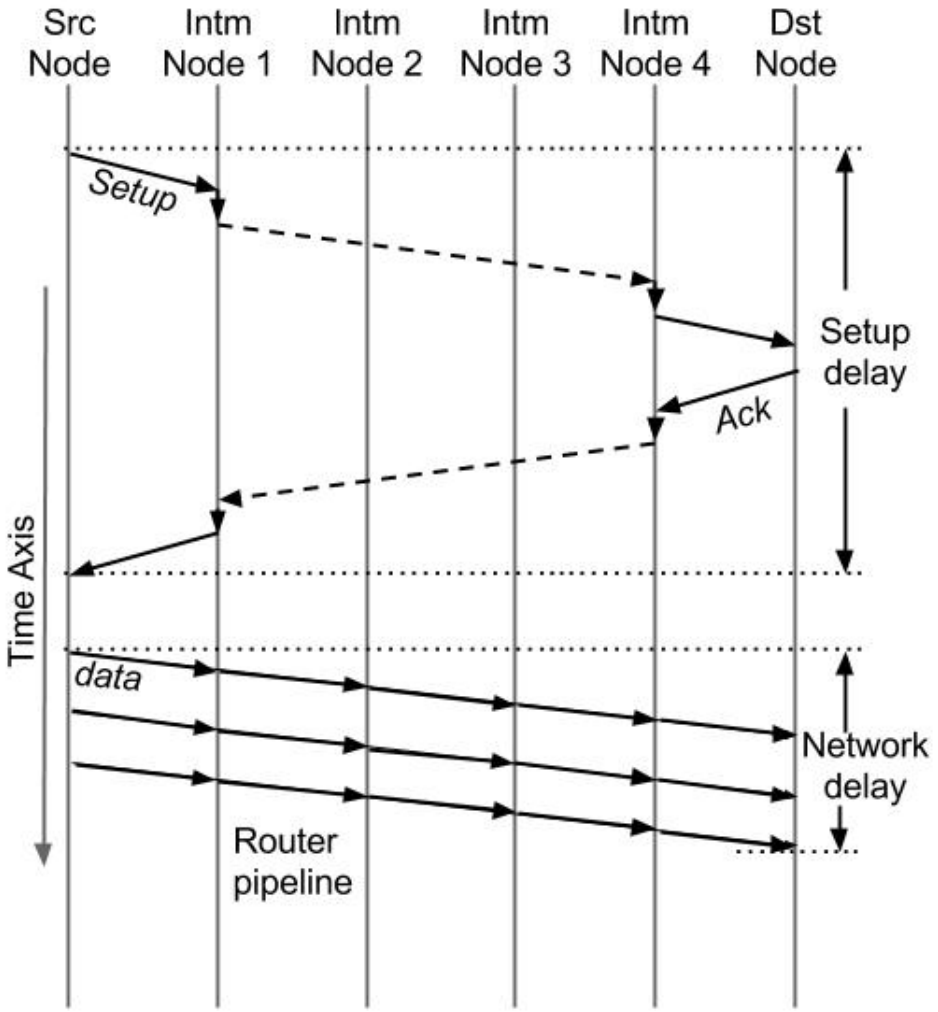

Figure 3.2: Flow of a circuit switching network

It is hard to order requests through packet switching which involves a lot of router to router communication and may also cause severe delay due to the network when considering the arrival of packets from cores in sequence which we identify as requests called `ReqID`. The implementation of this kind of network that has the communication

fabric using both circuit switching and packet switching is through time division multiplexing (TDM) as done in [21]. We have discrete time slots in which the available link bandwidth is allocated to a packet switched network or to a circuit switched network. These time slots are assigned at each router by having slot tables in which the entries consist of the link assignment with output port IDs. These reservations in the slot table occurs at every router and if a source node wants to set up circuit switched path to a destination then it sends `setup` messages which configures the path. These messages have information of the source node and the destination node along with the slot ID that it uses to reserve in the slot table. If the slots in the slot table are not free then the circuit switching paths are not set up. Since we need to guarantee circuit switching paths to ensure we can order the memory accesses we pre-configure the slot table size for a specific mesh network size consisting of CPU, shared cache nodes and memory controller nodes and also employ a static slot table allocation algorithm [20] to ensure we always succeed in providing circuit switching paths. So the setting up of paths is done by routing these messages to reserve a slot at every router from the source node to destination. Once this process is done, an acknowledgement message is sent back to the source node and the source node records a source-destination information so that the packets can be sent in circuit-switched fashion. We also ensure we do not tear down any paths after the paths have been set up. Removing the circuit-switched paths are done by `teardown` messages which works in a similar way compared to the `setup` messages but removes the path instead of sending an acknowledgement to the source to do path setup. We take due care to ensure this kind of messages are not transmitted to ensure the guarantee of circuit switched paths. Figure 3.2 shows the flow of a circuit switching network and we can see the impact of the set up time while Figure 3.1 shows the packet switching network's flow and we can see though there is more network delay though there is no set up delay involved. The slot table size is an important parameter to prevent degradation in performance due to circuit switching. Since we have to guarantee the setting up the circuit switched paths there exist a minimum size of the slot

table for a particular mesh network size. If the slot table size is small, the setup would not happen and while trying to set up the paths the source node will repeatedly try to poll into the slot table to look for a reservation and will be unsuccessful is setting up a path. This re-sending of set up messages will also lead to high overhead to set up paths creating delays.

At the same time using larger slot tables cause messages to stall for a longer period before transmission since the setup requires to go through all the slot tables at every router in the mesh network. We also show how the slot table size is affected with the network size in Figure 3.3. We run one micro-benchmark which is a simple increment operation in a small sized loop done for a local variable of each thread on three different network sizes and varying the slot table size. For applications that has huge number of memory accesses the performance is going to degrade even more. Keeping in mind there is a minimum limit for the slot table size for networks of different sizes, there is going to be a definite impact increasing the slot table sizes respective to a network size.

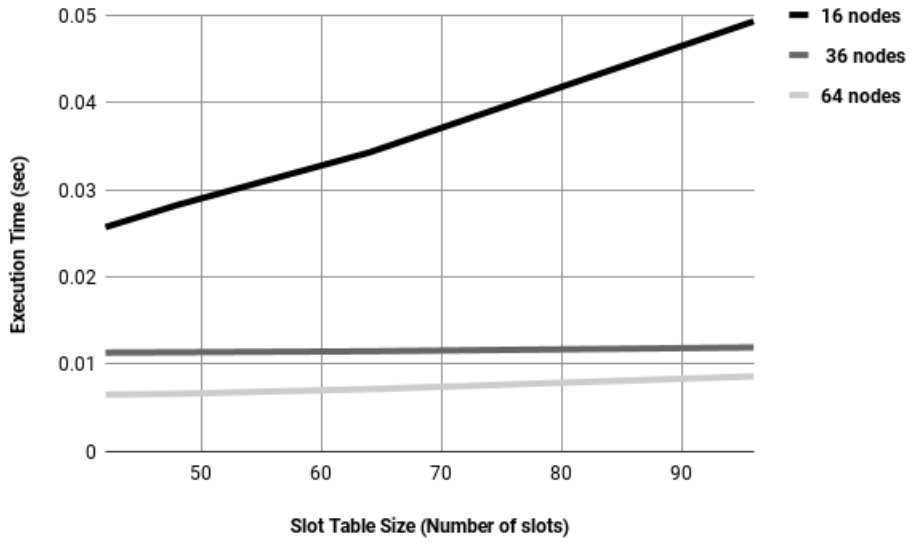

Figure 3.3: Impact of performance with increasing slot table size

As we can see there is a prominent impact on performance and hence we decided to have a fixed slot table size for a particular network size when testing our design with benchmarks. Having said this, the slot table size is statically determined that ensures all the paths can be allocated from source to destination and at the same time not have a large slot table that causes performance loss. By configuring circuit switching paths using the slot table reservations and sending packets through circuit switching means that, from a source to destination we can determine how a packet flows through the network. This is exploited to enforce ordering at the network.

3.2 Ordering at the network

After setting up the circuit switched paths the latency and the hops are determined from a source to a destination. Here the sources are essentially the CPU cores and the destinations are the shared cache memory nodes. Now since we use in-order cores, the issues of memory requests are in order to the network. The baseline and the proposed architectures are compared equivalently and our focus is to see the benefits of ordering at the network. We demonstrate the method for non-lock operations and lock operations. We have two mechanisms to handle the method of ordering for accesses in a critical section over accesses that do not involve critical section.

3.2.1 For non-lock operations

At a particular instant, we have memory access from different cores and every access can reach one of the shared cache memory nodes. For a particular cache node it cannot recognize which packet should be serviced first to ensure a global memory order since it can process any memory packet that comes first. When this happens there is a possibility that "correctness" need not be maintained. But if we pass a `ReqID` which is essentially a counter that incremented by each core locally based on its memory requests after its addresses have been translated into physical address values, we see that the

(ReqID,CoreID) pair is unique and this information is used to implement the ordering at these shared cache nodes. This information of all these tuples (ReqID, CoreID) is grouped at a memory node as tokens. They are stored by the network interface controllers, in a structure similar to a cache called reorder array which is analogous to the indexing of the cache with the "data" being the messages. So essentially the ordering is done based on this (reqID,coreID) tuple information from the core and the tokens update which is managed by the network interface controllers which are present along with the routers. At a particular time stamp we ensure queuing of the message with the lowest reqID first for every core before servicing the ones that come later. By this we ensure that at any memory node requests with lower request IDs are satisfied first from every core. Since the reqIDs are treated in incremental fashion whenever there are reqID say i and j, where i< j, from same core say A and at different nodes, the lower reqID, i will be serviced first as the other cache node would be servicing a request with a reqID, say k where k <= i from a different core request other than core A. This is how we ensure the earliest memory request from each processor is processed first irrespective of the distributed shared memory cache nodes.

For lock operations

For lock operations the methodology to order packets is fundamentally different. When core A acquires a lock it executes its critical sections and the other cores wait to acquire the lock till core A releases the lock. Essentially at a fine grained level these operations involve memory fences and a series of writes which is the updates to the critical section and de-fence. In this if we set up the paths for each cores to the destination nodes we can nonetheless ensure that the stores reach the destination shared cache nodes in order which essentially eliminates the effect of fences as the writes will be done sequentially where the writes to the critical section update is pipelined by queuing into circuit switched buffers present at the network interface. In this case, since the stores are guaranteed to finish at the same time we can eliminate the wait time due to write

acknowledgement back to the processor.

At this time the order should be such that for a particular core A, the `reqID`s must be processed in order completely and then next same set of sequence must be expected from the other cores looking from every shared cache nodes when they acquire the lock. Here since the `reqID`s are incremental by one, we simply ensure that the IDs are processed in order maintaining a global order. So if three `reqID`s from one core `i`, `j` and `k` where `i < j < k` is to be serviced. If `i` and `k` are accessing one shared node and `j` is accessing the other, our ordering ensures that the sequence of messages will be always `i` , `j` and `k` while in a non-ordered scenario it could be `i`,`j` `k` or `j`, `i`, `k` in which the former sequence is not always guaranteed. This is guaranteed using a token ring network

However, in a critical section where the shared variables are guaranteed mutual exclusion, we need not do the ordering, but by implementing we get some benefit since we use the circuit switching paths and try to squeeze all the writes to the network there by pipelining the writes and gaining some performance. This is an optimization and we will see in the next section how much benefit we benefit from the critical section ordering. We need to realize that parallel applications' bottleneck in extracting parallelism is the serialization parts of the application.

Here, since we know the set of `reqID`s being serviced for a core when executing the critical section while the other cores wait, we implicitly know the destination nodes. Based on this we overlap the setting up of circuit switching paths for the other cores while the one core executes the critical section. In the case where every core has equal priority we can ensure every core updates the same critical section and we do not have to worry about giving access to a higher priority core over a lower one. This is very specific to locking at a fine grained level for applications in which each cores are doing similar work just like in an accelerator workload.

- We provide a brief illustration of ordering done at the network. We show this in figure 3.4. Let us assume there are two shared memory nodes P and Q. And there

are two cores A and B. Each core services 4 ReqID each accesses one of P or Q.

- We show the sequence of how memory accesses are processed at the memory node P and Q. The request IDs are 0, 1, 2 and 3 for Core A and Core B. We can see that the tuple (CoreID, ReqID) is unique.

- We show how the ordering is done for both non-atomic operations and atomic operations involving critical section as shown in figure 3.4.

- For non-atomic operations we can see that multiple memory requests are services at a time at different memory module and we ensure there is a global memory order obeying the program order. The mechanism is depicted in figure 3.4a. We can see that it is enforced that the ReqIDs are in increasing order at each node.

- For operations like critical section it is seen that the reqID sequence is going to be same and processed sequence by sequence, so whenever we see a ReqID being serviced for one core other cores are going to follow suit. For this, we essentially try to pipeline the setting up of circuit switched paths thus overlapping some overhead latency. This is clearly depicted in figure 3.4b. So at P we can see that ReqID 0 and 2 from Core A is queued and the same sequence is queued from B onto the same Node P

This mechanism is done by the token ring network where the metadata is passed from the network interfaces of the routers connecting to the shared cache nodes. Whenever a message gets enqueued to the cache the token updates at the network interface to keep track of the ReqIDs and thus all messages are tracked this way. Initially the ReqID value is reset to zero at these interfaces. These interfaces know the highest ReqIDs being serviced at a particular instant among all the shared cache nodes. The ring network ensures the lower ReqIDs are serviced before the higher ones. We have buffers at these network interfaces of memory nodes which can communicate with the token and the metadata to check and make sure the correct ReqID is serviced.

Core A		Core B	
Req ID	Memory node	Req ID	Memory Node
0	P	0	P
1	P	1	Q
2	Q	2	Q
3	P	3	Q

Memory Node P	(A,0)	(B,0)	(A,1)			(A,3)
Memory Node Q			(B,1)	(A,2)	(B,2)	(B,3)

(a) Ordering for non-atomic operations

Core A		Core B	
Req ID	Memory node	Req ID	Memory Node
0	P	0	P
1	Q	1	Q
2	P	2	P
3	Q	3	Q

Memory Node P	(A,0)		(A,2)		(B,0)		(B,2)	
Memory Node Q		(A,1)		(A,3)		(B,1)		(B,3)

(b) Ordering for atomic operations

Figure 3.4: Sample illustration showing how the ordering is implemented at the network

3.2.2 Network Architecture and Target Applications

We target in-order cores since we want to emulate this architecture similar to accelerators. We assume cores processing similar tasks though we provide ordering if there are dissimilar work from cores. Our ordering mechanism works efficiently with this assumption as the ordering is at a fine grained message level in the network. As our proposal is best suited for parallel applications, we test the implementation against workloads that are multi-threaded and involve some synchronization. We prove that the correctness of the program execution and sequential consistency is guaranteed by the ordering the memory accesses at the network level. Each thread runs on one core.

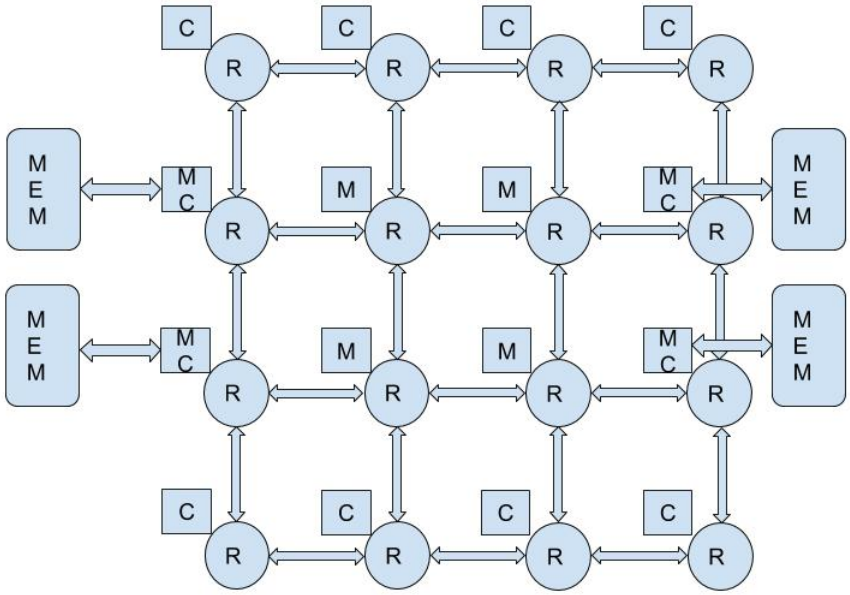

Figure 3.5: Interleaving topology of 16 nodes

As explained in the Introduction 1 section, we use an interleaving topology rather than clustered multi-core system like a Chip Multiprocessor (CMP) architecture. Based on a previous study [20], it is seen that a clustered system generates more traffic, and the network generally gets, more congested than an interleaving system that has just

CPU cores or memory controllers or shared cache nodes on a particular node. When the intention is to order memory accesses at the network, we do not want to have a clustered multi-core system that may cause long latencies in queuing packets at a specific memory node. This is especially a problem when the network size grows bigger. Thus to illustrate efficient ordering of memory accesses at the network we use an interleaving topology. We have shown a 4x4 Mesh topology that consists of 16 nodes in Fig 3.5. It is a 16 tiled system connected with routers to a 4-by-4 Mesh Network. **R** represents the routers and the tiles are connected by them. The tile that says **C** consist of a CPU core. The tile denoted by **M** is a bank of shared cache and the off-chip memories are connected through the memory controllers represented by **MC**.These routers are capable of both packet switching and circuit switching. They are designed and adopted from [21] that enables both types of routing using TDMA.

Chapter 4

Evaluation Methodology and Results

4.1 Evaluation Infrastructure

We use gem5 simulator [4] and the memory system is modeled using the Ruby-SLICC integrated with gem5. The interconnect is modeled in Garnet [1]. We use the system emulation mode of gem5 and analyze our workloads by linking pthread library to m5threads and carefully obtained the statistics by using the m5calls to extract the statistics for our Region-of-Interest.

We use the gem5 Ruby sequencer to distinguish the ordering mechanism implementations for a critical section and a non critical section and handle the memory accesses ordering differently by sending metadata information from the cores to the network. The CPU cores we use is the MinorCPU model of gem5 simulator. The memory has been remodeled by not using L1 cache and the protocol of Two Level MESI has been modified thoroughly to implement the architecture we desire. We do not have any complex protocols that keep tracks of the coherences in the cache.

We demonstrate three scenarios where each is based on a model described below

and we illustrate our idea using micro-benchmarks, written in assembly language, run on the three models and show results of each type. We do this for non-synchronization and synchronization involved situations. The three scenarios are as follows:

- Model 1 is a sequentially consistent memory model. It consist of a packet switching NoC in a Mesh topology connected through routers. As described in the 3, the topology consists of nodes that contain CPU cores, shared cache nodes along with the memory controller nodes which connects to the off-chip memories. As a packet switched unordered network we employ sequential consistency by applying memory fences whenever a memory access is taking place through a load or a store. We call this model Core-Ordered (CO). This is our baseline configuration.

- Model 2 is our proposed model that does in-network memory access ordering and we call it Network-Ordered (NO) model. This uses circuit switching to route the network traffic and provides sequential consistency by ordering the memory access packets at the network and ensures all the updates are seen by every other cores. This is our proposed architecture that does memory ordering

- Model 3 is a packet switching network which does not guarantee sequential consistency as write atomicity is not guaranteed in a packet switching network and does not employ any memory fences. We call this unordered (UO) model.

Each model has been tested for three network sizes consisting of 16, 36 and 64 nodes respectively as shown in table 4.1 below. All the cache nodes have memory controllers and the off chip memories are communicated through them. Also as the network size increases we double the shared cache nodes than the CPU core count since our aim is to order packets at the these cache nodes and hence doubling cache size along with the nodes for the three configurations is a good way to test scalability than doubling core count with increasing network size. Table 4.1 below describes the network configuration details.

Network Size	CPU Core Count	Shared Cache Nodes	Memory Controller Nodes
16 nodes	8	4	4
36 nodes	20	8	8
64 nodes	32	16	16

Table 4.1: Network configuration

We evaluate using the Core-Ordered (CO) network as our baseline which is a packet switched network that ensures sequential consistency through the usage of memory fences guaranteeing write atomicity. We compare a sequentially consistent Network-Ordered (NO) model and the unordered network which is not sequentially consistent with the baseline. Our evaluation metric is execution time of the micro-benchmarks normalized to the baseline. We choose this metric since we run parallel applications for which execution time gives the correct picture of evaluating performance.

4.2 Illustration for non-atomic accesses

In this section, we explain our proposed concept for applications without having any synchronization. We run an application that creates threads and every thread is run in one CPU core and each thread does a similar task that is independent without any requirement of mutual exclusion of variables in the task. The task is basically a simple set of operations written as instructions that loads five variables to the registers and does an increment operation on each of them and then updates the variable and this is set of operations is repeatedly done for 10 times. We ensure that all the variables in one thread are not dependent with the others for testing purposes, so that we do not need to use any synchronization primitives to maintain mutual exclusion of variables. To ensure the sample program is correctly executed to avoid any deliberate violation

in sequential consistency in model 3 which is bound to happen if same variables get updated by two different cores we employ independent variables for every thread for the purpose of demonstration to verify that the application executes completely and not fail any assertions.

4.2.1 Results and Discussion

The table 4.2 below depicts the performance gain values over the baseline.

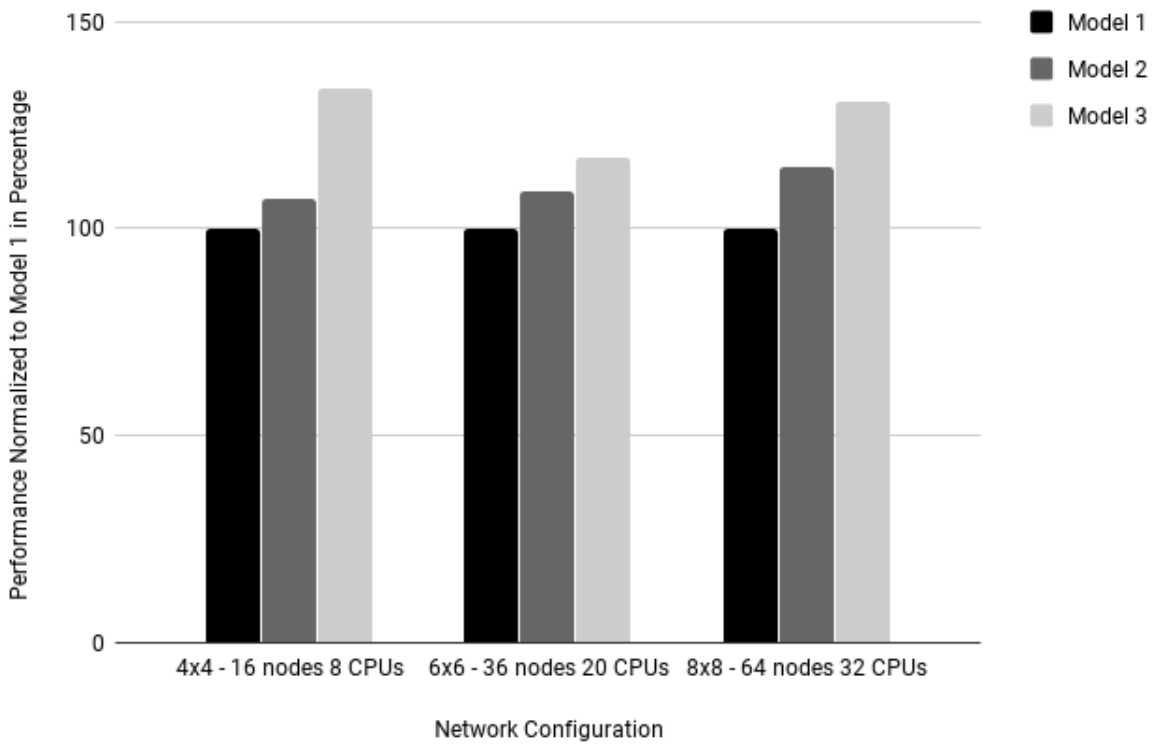

Figure 4.1: Performance of the three models for non-atomic accesses

Network Size	Performance gain of Model 2 over Model 1	Performance Gain of Model 3 over Model 1
16 nodes	7.18%	34.02%
36 nodes	9.03%	17.04%
64 nodes	14.9%	30.85%

Table 4.2: Comparison with a sequentially consistent packet switched Network for non-atomic accesses

From 4.1 we have shown the performance of the three models normalized to Model 1. We can see that compared to a packet switched unordered network that does not guarantee ordering, a circuit switched network that ensures ordering does not show benefit even though the latter is showing significant improvement compared to a packet switched network that is sequential consistent by ensuring every update is seen by every other processor.

Firstly, it is unfair to compare a sequential consistent model with any model that does not guarantee sequential consistency. But for our illustration purposes we show this comparison to depict the overhead to maintain sequential consistency at the network level. We need to set up circuit switched paths for implementing the ordering. This only can ensure a packet will take a pre-assigned path from a source to destination. The network interface of every router will have slot tables which essentially reserves the paths. Also this causes the packets traversing in the network to look into the slot table to choose the reserved path. This is essentially the overhead in setting up the paths and also based on an application these packets eventually get queued into the buffers at the memory nodes. At this part we ensure ordering by using the metadata (`CoreID`, `ReqID`) and ensure that the first one gets served and is queued before the next. This can lead to some more queuing delay at the memory nodes when we have to wait for a response from the main memory (when a miss in the cache is observed). Also, when

there are a large amount of memory accesses at the shared cache level, the ordering we employ can cause some delay since we might have to stall processing of one memory node over the other depending on the (`coreID`,`ReqID`) values. In spite of this, for our micro-benchmarks, we observe a noticeable benefit from ordering at the network over the cores.

Also, the packet switched unordered network is not showing a great improvement for 36 node network because as we had mentioned before we increase memory nodes proportional to the network size. Therefore, for a 36 node network we have 20 CPU threads while other two networks have half the number of cores as opposed to the total nodes, while in a 6x6 network we have more cores which means as each cores accesses memory we essentially have more memory accesses and if they are accessed at a particular cache bank node more frequently than others this leads to some contention and that is why we observe the unordered network not performing equivalently compared to the other two network sizes.

We see a good performance benefit over the packet switched network in which we enforce ordering to ensure write atomicity. This significant improvement of a circuit switched ordered network is because of the ordering of memory request at the network that at the core level. This essentially prevents any cores to stall and guarantees that every write is seen by all other cores. So as we can see the benefit is increasing with increase in the network size as well. We need to keep in mind that this ordering at the network will depend on the application, specifically, if there is going to be a lot of misses at the shared cache nodes, we do not expect any advantages. This micro-benchmark is a good example to demonstrate our idea as we ensure all the data is cached at the shared cache, since using a relatively small kernel we do not see many cache misses other than the compulsory ones.

Network Size	Network Overhead Cycles Lost	Benefit due to Ordering (Cycles Gained %)
16 nodes	4.27%	7.38%
36 nodes	5.05%	13.86%
64 nodes	7.21%	17.73%

Table 4.3: Network Overhead and Ordering Benefit Breakdown

In the table 4.3 we have shown the the network overhead due to setting up of circuit switching paths and also the benefit gained due to ordering using token ring. The network overhead has been calculated by comparing the packet switched network (Model 3) with circuit switched network without accounting for the token ring network's ordering scheme. While the benefit of ordering has been compared with circuit switched token ring network (Model 2) with circuit switching and using fences without using token ring network.

4.3 Illustration for Atomic accesses

For a lock type kernel that acquires a lock and executes critical section and releases the lock for other cores waiting to acquire, we see that the mechanism at the core level employs fences to maintain atomicity. However at the network level the ordering is not guaranteed. This can lead to problems when certain cores that accesses these shared variables and not waiting for the specific lock. To avoid this we use our idea of ordering for critical sections that guarantees sequential consistency.

4.3.1 Results and Discussion

We have depicted the performance normalized number over Model 1 showing for critical section accesses in Tables 4.4 and 4.6 for shorter and longer critical section accesses

and graphically showed the same in 4.2 and 4.3. We show performance benefit for a small and a large critical section. To quantify the larger critical section, it consists of instructions that are five times more than the smaller ones.

Network Size	Performance gain of Model 2 over Model 1	Performance Gain of Model 3 over Model 1
16 nodes	3.31%	4.47%
36 nodes	6.18%	7.18%
64 nodes	10.14%	10.98%

Table 4.4: Comparison with a sequentially consistent packet switched Network for atomic accesses with short critical section

We see a good improvement as the size of the network increases. Observing the performance with the unordered packet switched network, the networked ordered architecture performance almost similar to the unordered network. This is because we essentially set up the circuit switched paths during the lock acquire stage and by doing this we overlap the setup time with the lock overhead typically hiding the setup time with the lock spinning time. So the effect of the actual network overhead is minimized and effectively the overhead is less. At the same time during critical sections mutual exclusion criterion in general does indirectly ensures less traffic at the network so the network congestion is not significant compared to a non-atomic accesses thus analyzing on the traffic and bandwidth during critical section is not a reliable analysis.

The two tables 4.5 and 4.7, shows the breakdown in terms of the network overhead and the ordering benefit separately for short and long critical section execution of the Network Ordered model respectively. The network overhead compared the circuit switching overhead essentially. The benefit of ordering shows that token ring is a viable option for ordering memory requests.

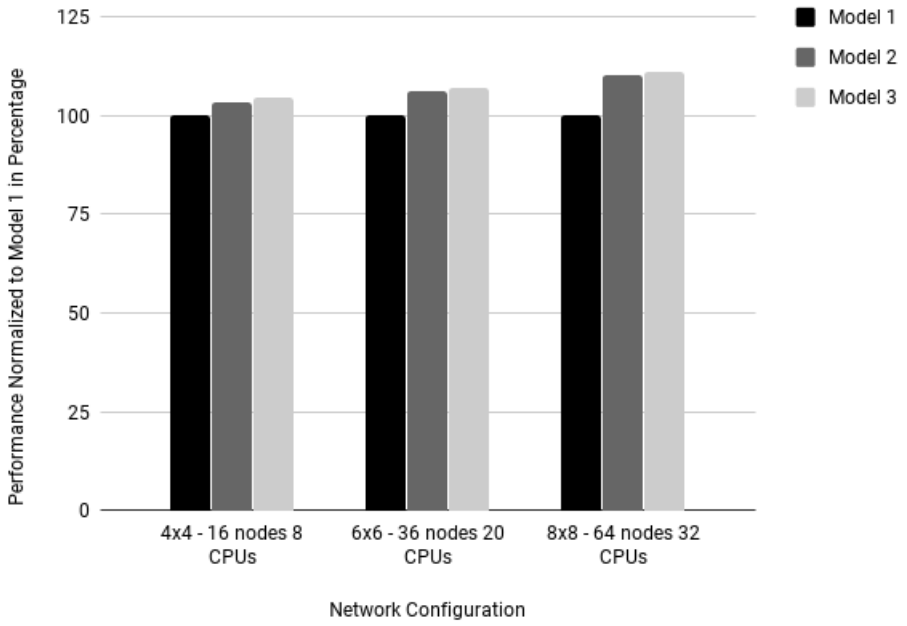

Figure 4.2: Performance of the three models for atomic Accesses involving a shorter critical section

Network Size	Network Overhead Cycles Lost	Benefit due to Ordering (Cycles Gained %)
16 nodes	0.15%	3.45%
36 nodes	0.17%	6.28%
64 nodes	0.33%	10.51%

Table 4.5: Network Overhead and Ordering Benefit Breakdown for short critical section

Network Size	Performance gain of Model 2 over Model 1	Performance Gain of Model 3 over Model 1
16 nodes	6.1%	6.78%
36 nodes	10.57%	11.21%
64 nodes	11.17%	11.64%

Table 4.6: Comparison with a Sequentially consistent Packet Switched Network for Atomic Access with Longer Critical Section

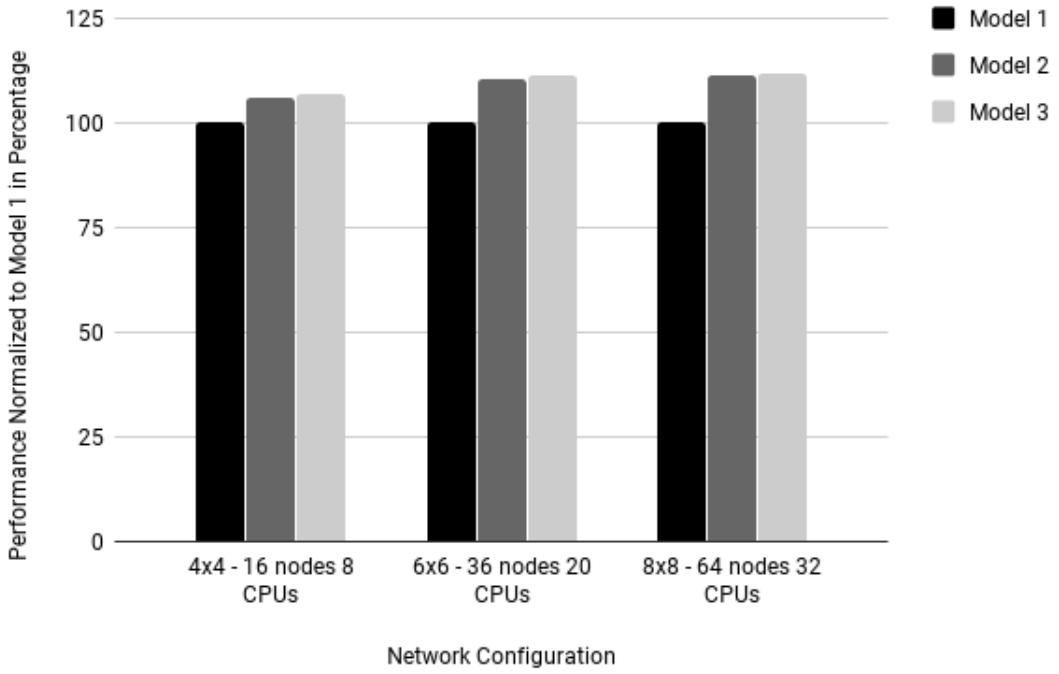

Figure 4.3: Performance of the three models for atomic accesses involving a long critical section

Network Size	Network Overhead Cycles Lost	Benefit due to Ordering (Cycles Gained %)
16 nodes	0.13%	7.45%
36 nodes	0.16%	11.28%
64 nodes	0.33%	11.91%

Table 4.7: Network Overhead and Ordering Benefit Breakdown for long critical section

We see a similar performance benefit in both cases of shorter and longer critical sections. However, we see that the network overhead is becoming less significant marginally as the critical section is becoming larger. This is a simple illustration to show that the network overhead gets overlapped with the lock overhead irrespective of the length of the critical section.

4.4 Analysis of Important workloads

4.4.1 Choice of Benchmarks

Since our architecture emphasizes on implementing a sequential consistent memory model in a multi-core framework, we have chosen benchmark that uses the state of the art parallelization techniques. We have simulated six benchmark applications mainly based on graph partitioning and search namely, Breadth First Search (BFS), Community, Page Rank, Triangle Counting (Triangle) , Depth First Search (DFS) and Connected Components (CC). We have used the source code from the CRONO Benchmark suite [2] and modified accordingly to test our proposed technique of memory access ordering. We employ these benchmarks mainly to keep focus on the parallel applications that involve parallelism as well locks that ensures mutual exclusion for the critical sections. For every benchmark, we have tested it for the three network sizes to study about the scalability of our Network-Ordered architecture. We also used input datasets,

synthetically generated and we analyzed for graph size in the order of ten thousand to hundred thousand nodes.

The way we did the analysis is that we used the source code and used its assembly form of the code and modified the necessary variables to be declared as volatile and compiled it accordingly so that we do not involve any compiler level reordering of instructions. Also for the baseline comparison which is a packet switched unordered network that guarantees sequential consistency by using memory fences for memory accesses. Our proposed model is to compare between two sequentially consistent models and for seeing the benefit of ordering at the network level and ensuring SC. To model the baseline we carefully insert memory fences to ensure the memory updates are made visible to every core.

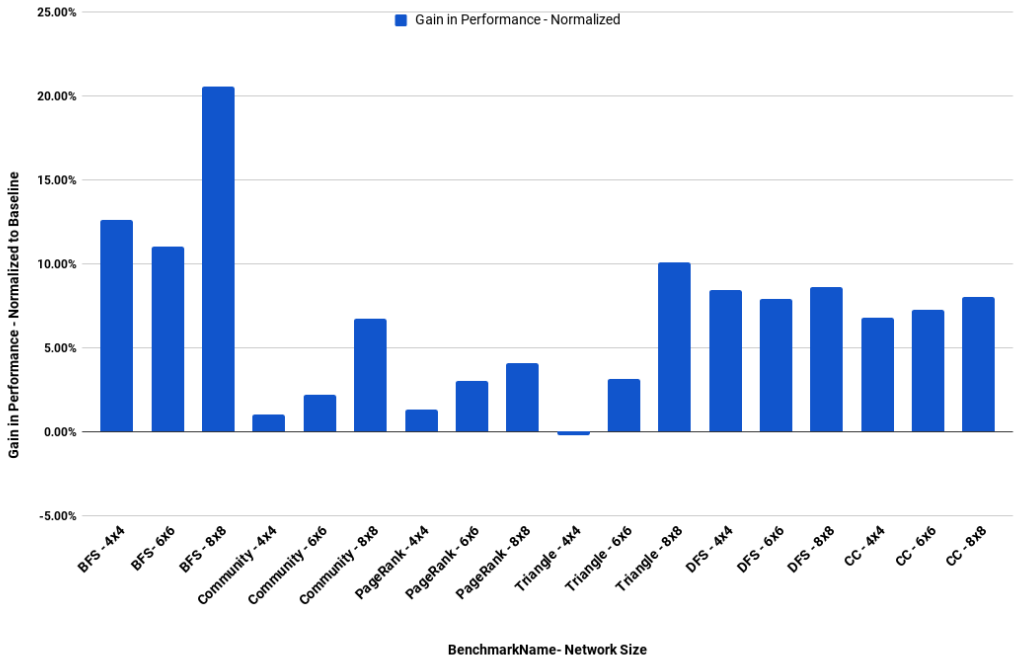

Figure 4.4: Normalized performance gain of the proposed model

Results and Analysis

We have depicted the performance gain of the Network-Ordered Architecture over the baseline which is a packet switched network guaranteeing SC by appropriate usage of fences. The metric we use to measure the performance is the execution time of the the region of interest of the application. We divide the execution time of the baseline with the proposed architecture and calculate the gain in the performance as shown in Fig 4.4. Since these workloads involve a critical section execution and based on the behavior of the workloads this can be critical to the performance. We show the effect of ordering the critical section over not ordering the critical section as well. By pushing the critical section's updates into the network and forcing the ordering done at the network, we try to pipeline the atomic updates at the same time ensure mutual exclusion by means of the ordering technique. We show why the critical section speed up is very crucial for parallel applications as we know the serialized sections do cause the limitations to exploiting all the speed available in a parallel application as projected in Amdahl's Law [9].

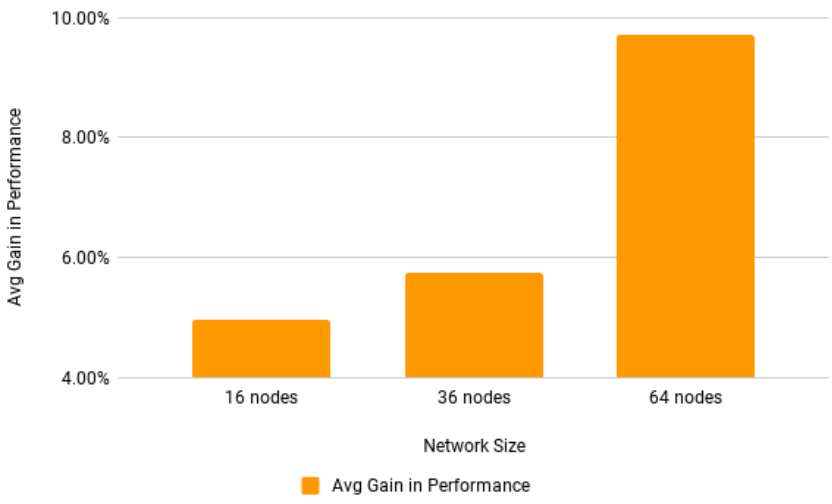

Figure 4.5: Average performance gain with Network Size

Benchmark-Network Size	Performance Gain without considering ordering of critical section (in percentage)	Performance Gain after considering ordering at critical section (in percentage)
BFS- 4x4	12.414	12.6
BFS- 6x6	10.72	11
BFS- 8x8	19.71	20.75
Community - 4x4	0.49	1
Community - 6x6	-.32	2.2
Community -8x8	3.98	6.7
PageRank -4x4	1.261	1.3
PageRank -6x6	2.766	3
PageRank -8x8	3.59	4.1
TriangleCounting-4x4	-.355	-.2
TriangleCounting-6x6	.62	3.1
TriangleCounting-8x8	-3.88	10.1
DFS - 4x4	8.27	8.44
DFS - 6x6	5.34	7.89
DFS - 8x8	5.37	8.58
Connected Components 4x4	-.02	6.775
Connected Components 6x6	-.8	7.23
Connected Components 8x8	1.02	7.998

Table 4.8: Tabulated results of performance gain with the baseline

As given in Table 4.8 we show how much of impact the critical section ordering each

benchmark can provide. This can vary depending on the work done during the lock acquire and release procedure compared to the work done by non atomic accesses.

- For Community, Triangle Counting and Connected Components, the program spends more time on critical section compared to BFS and Triangle counting. So the impact of ordering at the critical section will be more. For eg., for 8x8 network: for Community and Triangle Counting the improvement in ensuring the order at network is a lot more beneficial that BFS, Connected Components and Pagerank since the time spent in critical section is less.

- When the CPU count increases and for a benchmark that has more accesses with mutual exclusion, there is essentially more time in serialization, and here, queuing the packets in order at network is beneficial. So the impact is found to be more for larger networks.

- We can infer from the table that the time spent is an approximate way to see a benchmark behaving to the ordering of critical section. There could be mild variations depending on the input data as well. We also performed this experiment for varying input data size and found similar trends, thus showing the impact that serialized execution can have on parallel application with increasing network size.

- Figure 4.5 is the average of the performance gain of all the benchmarks with network size. This is just an evidence that critical sections are of concern when there are more cores complying with Amdahl's law and hence benefit of ordering on the critical sections seems to have a significant impact. We see that the average performance increase as we increase the network size.

4.5 Summary

We guarantee ordering of memory access in a multi-core set up by deploying a circuit switched network. This provides a sequential consistent behavior for this type of architecture and the Network ordered model performs better than the Core-ordered model.

Chapter 5

Related Work

Bulk SC [5] is one idea of implementing SC at the hardware. However, the design is based on transaction-like method where the design is done at the core level and we provide a solution by doing memory ordering at the network. Also we show scalability as the network size increases. SCORPIO [7] is similar to the idea of engendering a global ordering of request on a mesh topology by ordering in the network. However they have proposed this for a snoopy coherence and compare with directory coherence and we try to use just one cache that is shared among all cores with simple coherence states that exists on separate nodes in the network. We employ circuit-switching instead of their packet-switched network, where they implement the global order. For our work circuit switching enables ordering using token ring network. In our mechanism of ordering we also show the importance of pipelining writes by employing ordering for critical sections also for graph based workloads.

Uncorq [15] talks about implementing snoop based request based on ordering invariant on a ring network. Though they ensure requests are serialized to cachelines, they do not implement sequential consistency. Although, they try to speed up the read latency by not waiting for response messages they talk about speeding up stores for certain memory models like the PowerPCs but not for a stronger memory model. Also,

they have not evaluated speed-up of stores which they have mentioned in their paper but they do claim will happen for the weaker models based on their idea. We have a different method of ordering using tokens and use circuit switching to do the same.

Ring-based interconnects are widely adopted by commercial vendors like the ones in the IBM Cell [12], Intel Larrabee [13], etc. and these interconnects are implemented to connect multi-processors and data parallel accelerators. Efficient routing in ring based NoCs is due to its low complexity in control logic and datapath implementation. We use the token-ring network to implement the ordering mechanism by just passing tokens which are meta data and is lightweight and does not contain the complete message of address or data.

There has been some previous work related to accelerating critical sections in a program [16, 19]. They talk about using asymmetric cores in which larger cores can execute critical sections instead of smaller ones while we focus on implementing SC and try to speed up critical sections by ensuring requests are pushed into the network and ordering is guaranteed. Also, researchers have worked on reducing overhead on the locks at a more coarser level granularity while we try to indirectly overlap the lock overhead with circuit switching set up when executing critical sections at a fine-grained ordering of memory access at the network. Also, we focus on implementing SC efficiently by employing memory ordering in the network.

Chapter 6

Conclusion and Future Work

Seeing along the future of multi-core systems we believe that memory consistency problems will be prevalent and challenging to address. From this work we demonstrated one way to support sequentially consistent memory model in many-core systems. We proposed to order memory requests in the network by time-stamping each memory request and circulating a token among the memory modules. Our experiments show that in many-core systems where in-order cores with no private caches, the proposed mechanisms can efficiently support sequential memory consistency by utilizing circuit-switching in the network. We have also show that the proposed mechanisms can scale considerably with the network size.

There are many opportunities for improving the proposed system so it can adapt to more diverse architectures. The following are some of the limitations to this design:

- Though we have showed in this work that ordering memory accesses in network scales with the network size, there is a need for using buffers to queue the packets. So there could be some queuing delay if there are a lot of packets pushed into one memory node by many cores at one time instant. So if this situation arises there could be a degradation in performance as the circuit switched paths are not efficiently utilized.

- The slot table sizes used to reserve circuit switched paths are statically determined so we try to ensure the paths can be set up at the same time not make the slot table too large as this increases interconnect delays. However, if we do not set the table size optimally, we could see excess delays as overhead and there is a minimal overhead that is required to ensue all the paths are set up and this increases as the memory nodes go up.

- As the memory nodes go up and there could be a situation where we need to stall a request being serviced to ensure a global order which involves (`CoreID`,`ReqID`) information passed to all memory nodes. If this happens more frequently, there is a possibility that the ordering of the memory packets by the token ring network can increase the execution time.

Since this implementation is aimed at making the hardware reliable in terms of memory behaviors so that programmers can write parallel programs without worrying about counter intuitive results, we do not involve compiler level optimizations. However, compilers are significant in making a program efficient while running on a hardware. Future work could be to enable compiler optimizations selectively so that we enjoy the benefit of having a sequentially consistent memory model and allowing certain optimizations form the compiler too. Also, using this technique on heterogeneous architecture will be a challenging as the network traffic may vary which is a problem to be tackled. Using out of order cores and having private caches in a system and working with a coherence protocol will be the next step towards making this design an attractive option to adopt over relaxed memory models. In this work, we created some opportunity for researchers to think about employing a global memory order by the network that provides performance benefits over a system that provides sequential consistency by using fences in hardware.

References

[1] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42. IEEE, 2009.

[2] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 44–55. IEEE, 2015.

[3] Ali Bakhoda, John Kim, and Tor M Aamodt. Throughput-effective on-chip networks for manycore accelerators. In *Proceedings of the 2010 43rd annual IEEE/ACM international symposium on microarchitecture*, pages 421–432. IEEE Computer Society, 2010.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 278–289. ACM, 2007.

[6] Jason Cong, Michael Gill, Yuchen Hao, Glenn Reinman, and Bo Yuan. On-chip interconnection network for accelerator-rich architectures. In *Proceedings of the 52nd Annual Design Automation Conference*, page 8. ACM, 2015.

[7] Bhavya K Daya, Chia-Hsin Owen Chen, Suvinay Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P Chandrakasan, and Li-Shiuan Peh. Scorpio: a 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering. *ACM SIGARCH Computer Architecture News*, 42(3):25–36, 2014.

[8] Mark D Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, 1998.

[9] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[10] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. Armor: Defending against memory consistency model mismatches in heterogeneous architectures. *ACM SIGARCH Computer Architecture News*, 43(3):388–400, 2016.

[11] Sanjay Patel and Wen-mei W. Hwu. Accelerator architectures. *IEEE Micro*, 28(4):4–12, July 2008.

[12] Dac C Pham, Tony Aipperspach, David Boerstler, Mark Bolliger, Rajat Chaudhry, Dennis Cox, Paul Harvey, Paul M Harvey, H Peter Hofstee, Charles Johns, et al. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, 2006.

[13] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al.

Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[14] Konstantin S Solnushkin and Yuichi Tsujita. Marrying many-core accelerators and infiniband for a new commodity processor. *arXiv preprint arXiv:1307.0100*, 2013.

[15] Karin Strauss, Xiaowei Shen, and Josep Torrellas. Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–342. IEEE Computer Society, 2007.

[16] M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 253–264. ACM, 2009.

[17] M Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 341–353. IEEE, 2003.

[18] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE micro*, 27(5):15–31, 2007.

[19] Yuan Yao and Zhonghai Lu. Opportunistic competition overhead reduction for expediting critical section in noc based cmps. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 279–290. IEEE Press, 2016.

[20] Jieming Yin. *Time-Division-Multiplexing Based Hybrid-Switched NoC for Heterogeneous Multicore Systems*. PhD thesis, UNIVERSITY OF MINNESOTA, 2015.

[21] Jieming Yin, Pingqiang Zhou, Sachin S Sapatnekar, and Antonia Zhai. Energy-efficient time-division multiplexed hybrid-switched noc for heterogeneous multicore systems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 293–303. IEEE, 2014.

[22] Kazumi Yoshinaga, Yuichi Tsujita, Atsushi Hori, Mikiko Sato, Mitaro Namiki, and Yutaka Ishikawa. Delegation-based mpi communications for a hybrid parallel computer with many-core architecture. *Recent Advances in the Message Passing Interface*, pages 47–56, 2012.

[23] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.