Implementing MPI_Barrier with the NetFPGA

O. Arap[1], **G. Brown**[2], **B. Himebaugh**[2], **and M. Swany**[1]
[1]Center for Research in Extreme Scale Technologies, Indiana University, Bloomington, IN, USA
[2]School of Informatics and Computing, Indiana University, Bloomington, IN, USA

Abstract— *Parallel programs written using the standard Message Passing Interface (MPI) frequently depend upon the ability to synchronize execution using a barrier. Barrier synchronization operations can be very time consuming. As a consequence, there have been investigations of custom interconnects and protocols for accelerating this operation and other collective operations in parallel MPI programs.*

In this paper, we explore the use of hardware programmable network interface cards utilizing standard media access protocols as an alternative to fully custom synchronization networks. Our work is based upon the NetFPGA – a programmable network interface with an on-board Virtex FPGA and four Ethernet interfaces. We have implemented a network-level barrier operation using the NetFPGA for use in MPI environments. This paper compares the performance of this implementation with MPI over Ethernet for a small configuration.

Keywords: NetFPGA, MPI, MPI_Barrier, Synchronization, Collective Operations

1. Introduction

Barrier synchronization can have a significant performance impact on programs running on large parallel processors. A barrier is a logical delimiter for participating processes to ensure that all the processes are at the barrier point in their execution sequence [6]. A participating process may continue with its execution after it receives a release notification either from one of its peers or after an appropriate set of peer message exchanges indicates that all the participating processes have called the barrier. Regardless of the task parallelization, the barrier is a sequential blocking call for all the processes. It introduces a latency completely depending on the execution sequences of other processes, underlying communication infrastructure and the logic of the barrier implementation.

In the past years, many proposals have been presented to reduce the latency of barrier synchronizations. They are classified as software solutions, hardware solutions and hybrid solutions that involve both hardware and software aspects. Software barrier proposals are largely independent of underlying hardware technology [12] [19]. They tend to be implemented using generic solutions that can be applied to different platforms by just changing the calls in the user level library implementation. Software solutions lack the performance of hardware and hybrid solutions due to the fact that software solutions are inherently limited by the hardware, which is not necessarily optimized to implement barrier logic.

Hardware based and hybrid solutions are typically proposed for specific target platforms such as parallel machines with custom interconnects, clusters of FPGAs with a specific communication medium, parallel machines with specific target topologies, etc [11] [14] [5] [20] [15] [7] [18] [1] [8]. However, not all researchers have access to special purpose parallel machines. As a result, many researchers build their own cluster using Commodity Off-the-Shelf (COTS) hardware. This is an active area of research focused on clusters of workstations, which can be constructed using Commodity Off-the-Shelf (COTS) processors and hardware to achieve high performance parallel execution.

This work is focused on investigating how programmable hardware platforms such as the NetFPGA [13] can be utilized to implement barriers. The NetFPGA has become a standard platform for learning and implementing networking hardware in academic research. To the best of our knowledge, this is the first attempt to utilize the NetFPGA in the implementation of barrier synchronization. The NetFPGA platform has been widely used to prototype networking hardware with the goal of reducing the performance costs by offloading some specific tasks to the hardware level. It has standardized interfaces between hardware modules and software level access to the hardware modules.

It is difficult to claim that our hardware barrier implementation using the NetFPGA bests all the other hardware barrier solutions in terms of performance since we do not have access to all the competitive technologies, and it is not our goal with this work. However, lowering the barrier logic into the hardware provides significant performance benefits compared to software based implementations, and we will show that our implementation using the NetFPGA does not conflict with this assumption. There are several proposals that target different FPGA platforms, which either implement the entire system on chip, or utilize single FPGA as a separate networking device such as NIC or switch. However, our design is different since the NetFPGA provides implementation standards with a specific development suite. We completely utilize the NetFPGA development environment, and thus leverage its extensibility for future functionality.

The remainder of this paper is organized as follows: Section 2 summarizes the design goals. Section 3 outlines the implementation details and architectural design. Section

4 presents performance evaluation of our design. Section 5 provides some background and discusses related work. Section 6 offers discussion about our work and how it could be extended in the future, and finally Section 7 concludes the paper.

2. Design Goals

In this paper, we propose a barrier synchronization framework utilizing the standard infrastructure from the NetFPGA platform and using standard protocols such as UDP, IP and Ethernet. The unique contributions of our work are as follows:

- The design relies on the standard NetFPGA driver and there is no need to change anything in the OS. We incorporate some simple changes in the user-level code, utilizing the Open MPI [2] library to generate the packets that the NetFPGA recognizes and processes.
- All of our additional hardware modules live in the user-data-path [4], as recommended by the NetFPGA user community. Therefore, it is self-describing and could be extended by someone who is familiar with the NetFPGA environment.
- We enable a flexible topology that could be created by connecting different ports of the NetFPGA directly to each other. The current implementation supports four distinct physical topologies.
- We are providing a framework that can be easily extended to other types of MPI collective operations. We began with the barrier implementation as our base.
- Our work does not require a separate control network for barrier synchronization as it can perform the synchronization on the network where the data also flows.

3. Implementation

Our FPGA node design is derived from the *reference NIC* implementation distributed with the NetFPGA package. The host communicates with our synchronization engine through a UDP socket – operating system support for such sockets is part of the standard package. The NF_Barrier implementation consists of sending a specially crafted UDP message, and then blocking until a barrier release message is received. An added feature of building our implementation upon the NetFPGA reference NIC is that our node maintains the ability to forward standard IP packets.

The simplicity of the host interface belies the complex task that the barrier node must perform. The barrier tracks outstanding requests by storing the various MAC, IP addresses, checksum and UDP header fields. These are later used to generate a message to release the host from the barrier. The generated release packet must arrive user-space travelling up to the protocol stack. Therefore, it must be properly formed, so that none of the layers prevent packet to be processed by the application layer.

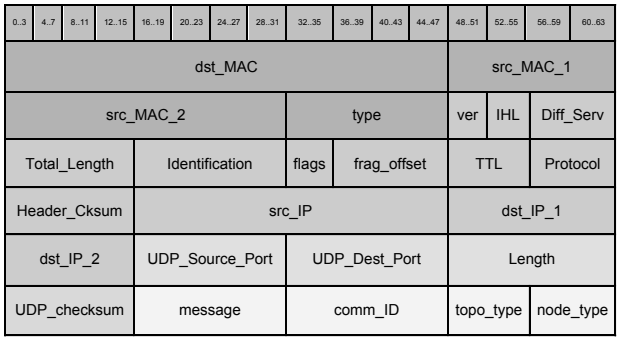


Fig. 1: Fields and structure of an actual NF_Barrier packet

3.1 Packet Format

Our design is intended to support a variety of topologies. We use the packet format presented in Figure 1 to inform the underlying synchronization hardware about the current topology. The synchronization hardware state-machine is customized to support each specific topology.

The *message* field denotes the packet type. Host processes handle only two types of message – a barrier start and a barrier release message. The NetFPGA updates the message type based on its current state in the state machine. It may handle other message types based upon the current topology. For example, with the tree topology, there is an additional message to indicate that there are children at a barrier, and to notify the parent NetFPGA that all of its children have called the barrier. The *topo_type* field is to specify the network topology. Currently, we support ring, binary tree, butterfly and star topologies. The *node_type* field denotes the node type in a specified topology.

3.1.1 Life of Barrier Packet in the NetFPGA

Once a packet arrives at the NetFPGA, it is placed in the appropriate receive queue and is passed to the user data path. The receive queues attach a module header to inform subsequent modules about the packet source and length. From the input queue, the packet arrives at the *output_port_lookup* module which examines whether it is a barrier packet. If it is a barrier packet, based on the state it is in, the *output_port_lookup* module determines which ports the packet is going to be injected. If the packet is going to be forwarded to multiple ports, the packet is duplicated to the multiple transmit queues at the same time. If the packet is not a barrier packet, the packet is handled as a regular Ethernet traffic, and is forwarded based on the receive queue number it is received in.

3.2 Supported Topologies

We explain in detail how the organization of NetFPGA nodes in specific topologies and the communication required to achieve synchronization.

3.2.1 Ring Topology

The ring topology has two types of nodes: head and regular nodes. Head nodes wait for their host to call the barrier, and then for the rest of the ring to call the barrier. After the head node learns that all previous nodes are at the barrier, it initiates a release message to the next node and to the host. Non-head nodes wait for the nodes preceding them and their hosts to call the barrier before sending the *MSG_PREV_AT_BR* message; they then wait to receive a release message, initiated by the head node, and subsequently forward the release message to their host and successor in the ring.

In the ring topology, $port0$ is used to connect to the successor in the ring and $port1$ is used to connect to the predecessor node in the ring. On the wire, the synchronization packet flow runs in only one direction, which is from $port0$ of the current node to the $port1$ of the next node.

3.2.2 Tree Topology

The tree topology is implemented with three node types: root, internal, and leaf. As expected, a leaf node has no children and a root node has no parent. Internal nodes have both children and a parent. A leaf node waits for the host to call the barrier, sends a *MSG_CHILD_AT_BR* message, waits to receive a release message from its parent, and finally releases its host. The root node waits for its children and host to call the barrier (the order is irrelevant), then it sends a release message to its children and the host. An internal node is a combination of leaf and root nodes. It initially waits for its children and host to call the barrier. Then, it notifies its parent that its host and children have called the barrier. Finally, it waits for a release message from its parent. When an internal node receives a release message from its parent, it forwards the message to its children and host.

In the tree topology, $port0$ and 1 are used to connect to children (if any) and $port2$ is used to connect to the parent node (if any). In this topology, the packet flow is on both directions on the link.

3.2.3 Star Topology

The star topology is implemented with 2 node types: center and regular nodes. Center node will wait for all the other nodes connected to it to call the barrier. After all regular nodes send the *MSG_AT_BR* messages; they wait for center node to send the release message. The center node will craft the release message when all the other nodes and its host call the barrier. It sends the release message to the regular nodes and its host. In the star topology, if the node is a regular node, only $port0$ is used to connect to the center node. All of the ports of center node could be used to establish the star topology which is up to 5 nodes because of the NetFPGA port limitation.

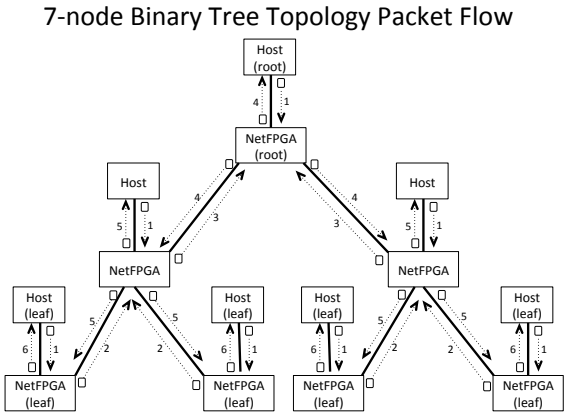


Fig. 2: NF_Barrier packet flow for 7 nodes in a binary tree

3.3 Sample Packet Flow

Figure 2 depicts a sample packet flow scenario for a 7-node complete binary tree.

1) All the hosts invoke the barrier and the NetFPGAs receive the *MSG_AT_BR* messages. The NetFPGA stores necessary header fields for constructing a release message when the time comes.
2) Leaf NetFPGAs update the message received from the host and tell their parents that the node and its children are all at the barrier even when they have no children.
3) The NetFPGAs that are in between root NetFPGA and leaf NetFPGAs receive the messages from their children and since their hosts are also at the barrier, they forward the message to their parent which is the root NetFPGA.
4) Since all of its children and the host itself are at the barrier, the root NetFPGA crafts a release message with the remembered header fields and sends it to its children and the host at the same time.
5) Internal NetFPGAs also perform necessary header field updates, and forward the release message both to the host and children.
6) The leaf NetFPGAs receive the release message and, after updating the header fields, they release their host processes from the barrier.

The preceding scenario demonstrates the packet flow in our tree design. We are going to present a sample packet flow for our ring topology in the next section, while we describe our performance measurement model.

3.4 State Machines

To describe the designs, we provide the protocol state machines for the various nodes in the tree topology. Figure 3 presents the state machines employed in a full binary tree topology. Figure 3.d details the meaning of the state names and transitions between them.

We did not provide figures for the state machines for ring and star topologies because they are very simple. On

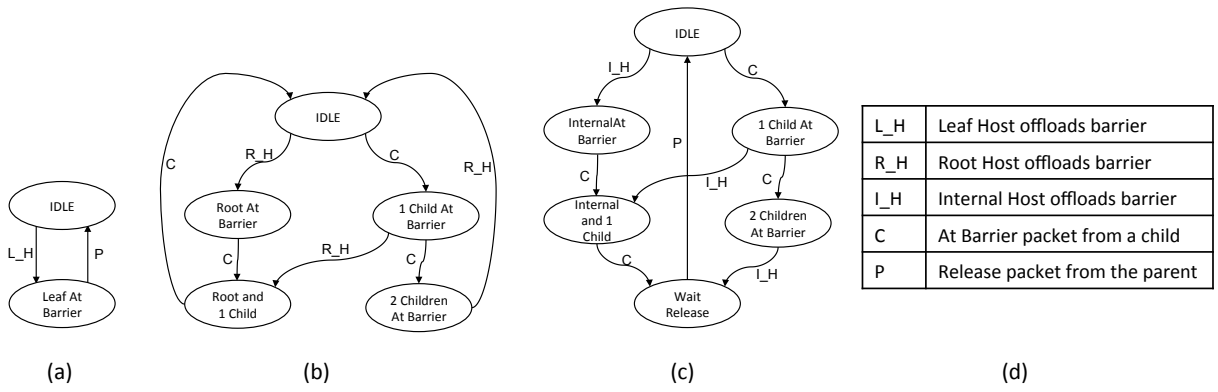
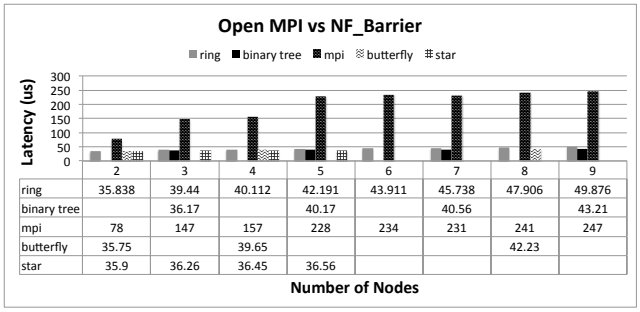


Fig. 3: State Machines for the binary tree topology (a) leaf nodes (b) root node (c) internal nodes (c) Legend for state transitions

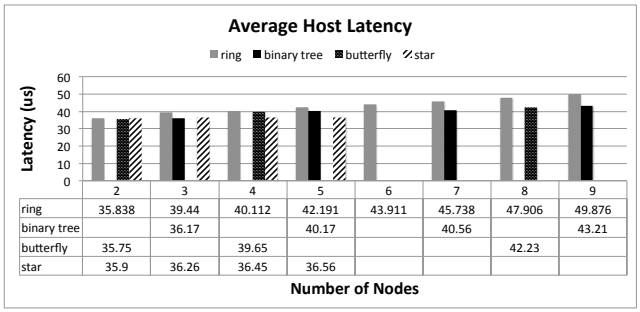
the other hand, the butterfly topology state machine is not presented but briefly discussed because it is so complex compared to the other cases. The non-central node in the star topology should employ a state machine like a leaf node in a tree case because its role is to wait for the host to call the barrier, notify the central node about the barrier call and wait for the central node to broadcast the release message. The central node is like the root node in the tree topology and it waits for regular nodes notifications to generate the release message. The order of message arrival from the neighbor hosts does not matter.

The most complex topology is the butterfly. The number of states is a lot more than other cases because the order of packet arrival matters in this implementation. There is no specific node role in this implementation and every node employs the same state machine unlike other cases discussed so far. In an 8-node butterfly topology, each NetFPGA must connect three other NetFPGAs. In addition, it will also interface to the host. Therefore, there are total of four ways to receive barrier packets. Since the order of the packet arrival matters, there are 4!=24 different sequences these packets can arrive. The order is important, and the ports, which packets are received from, represent different states. For example, if it is received from the $port1$, it means 2 nodes in the whole topology called the barrier. The states somewhat employ a logic to keep track of who have called the barrier until then in the whole topology from a single node's perspective. Since the butterfly algorithm is a 1-phase barrier algorithm, there is no release message circulating between the NetFPGAs and the only release message is sent from NetFPGA to the host.



	2	3	4	5	6	7	8	9
ring	35.838	39.44	40.112	42.191	43.911	45.738	47.906	49.876
binary tree		36.17		40.17		40.56		43.21
mpi	78	147	157	228	234	231	241	247
butterfly	35.75		39.65				42.23	
star	35.9	36.26	36.45	36.56				

Fig. 4: Performance comparison of NF_Barrier for implemented topologies to generic MPI_Barrier for Open MPI



	2	3	4	5	6	7	8	9
ring	35.838	39.44	40.112	42.191	43.911	45.738	47.906	49.876
binary tree		36.17		40.17		40.56		43.21
butterfly	35.75		39.65				42.23	
star	35.9	36.26	36.45	36.56				

Fig. 5: Performance comparison of different topologies which are currently implemented for NF_Barrier

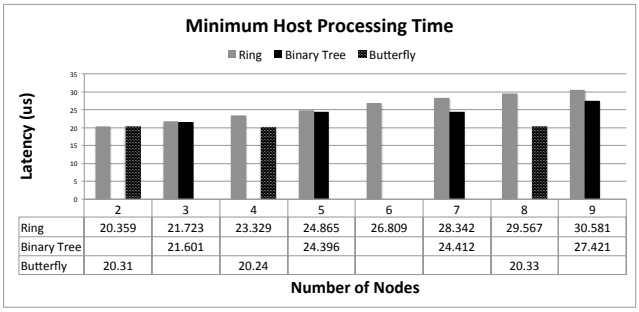
4. Evaluation

4.1 Experimental setup and results

Our experimental setup consists of 8 NetFPGAs in hosts with Intel(R) Core i5-2400 at 3.10GHz CPUs, 4GB RAM, and a dual Gigabit Ethernet NIC. The NetFPGA ports were directly connected to the each other establishing a tested topology. In this paper, we present micro-benchmark results obtained running OSU Micro-Benchmark Suite [3] for MPI_Barrier. In addition, we are going to describe how we can precisely time the NetFPGA operations after we offload the collective to the NetFPGA network.

The benchmark is configured to run 10 million barrier calls and averaged latency results are recorded. Figure 4 shows the latency of a single barrier operation for different numbers of hosts and various topologies.

Even though averaged results give us significantly bet-



	2	3	4	5	6	7	8	9
Ring	20.359	21.723	23.329	24.865	26.809	28.342	29.567	30.581
Binary Tree		21.601		24.396		24.412		27.421
Butterfly	20.31		20.24				20.33	

Fig. 6: Minimum latency experienced by different topologies which are currently implemented for NF_Barrier

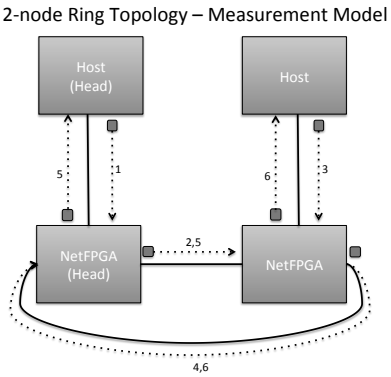


Fig. 7: Example packet-flow scenario that describes our precise performance measurement model

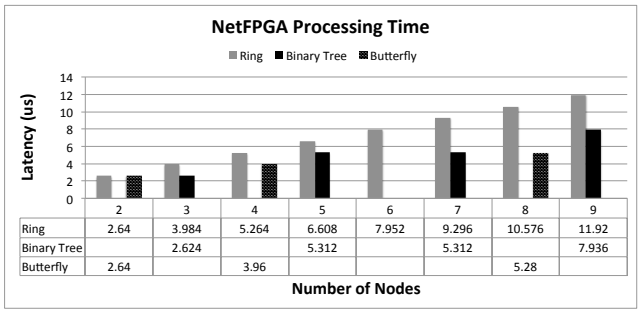
ter performance compared to the point-to-point Open MPI implementation, it does not precisely demonstrate how our design contributes to the overall barrier latency. According to the results presented in Figure 5, if the number of nodes is increased by one in ring topology, it introduces approximately 2μ latency. If the height of the tree increased by one, it introduces additional 3.5μ. It is also the same for the butterfly topology, if we increase the number of nodes by the power of 2. However, because of the node parallelism, these numbers are expected to be close to each other since we are introducing a single parallel NetFPGA processing to the overall processing time. We run our benchmarks to find out what the minimum latency of a barrier would be for various hosts in various topologies. The minimum latencies experienced are presented in Figure 6. The purpose of presenting the minimum results are to show that the host itself introduces a huge variance to the overall performance of our implementation. Therefore, it is not fair to evaluate our design based on average results unlike some other previous work [8]. As observed in Figure 6, when the host involvement in barrier latency is minimal, it provides more precise data for understanding how our design really contributes to the overall performance. Hence, we can extrapolate valuable information about processing time of the NetFPGAs. According to these results, an additional node to the ring introduces an average of 1.46μs latency, an increase in the height of the tree introduces 2.95μs of latency to the leaf nodes, and there is no latency introduced for the case of butterfly implementation.

We define p as the NetFPGA's single packet processing time. In the ring case, if a node is the last one to arrive barrier call, it will wait for its packet to circulate through the ring once. Therefore, an increase in the number of nodes in the ring would introduce extra latency of p to the last node arrived at the barrier. In a full binary tree, if a leaf node is the last one to arrive barrier, its notification packet goes up to the root, and then it is sent back to all the children as a release packet. Therefore, an increase in the height of a tree would introduce latency of $2p$ to the leaf nodes. In butterfly topology, we expect latency to increase p amount when the number of nodes increase by the powers of two. However, we do not see consistant results for this case in Figure 6. The presented numbers are a lot more consistent than averaging overall latencies and give us an idea about how fast the NetFPGA processes the packet. So, based on these results, p is around 1.46μs. However this is still not a precise measurement.

To precisely measure the NetFPGA processing time, p, we developed the model pictured in Figure 7. NetFPGA has a 125Mhz clock and we created a 64-bit timer which increments on each clock cycle. The steps to measure the NetFPGA's single packet processing time in 2-node ring topology are listed below.

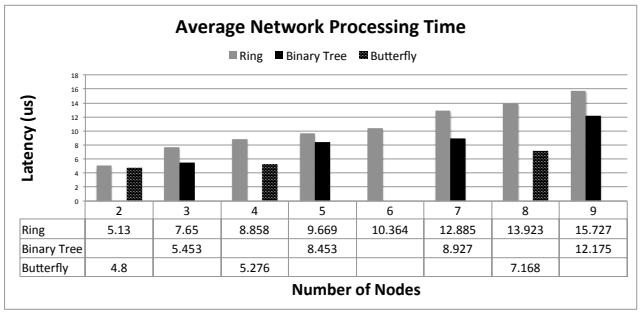
1) The host of the head node manually sends an MPI_Barrier message to the NetFPGA.
2) The NetFPGA forwards this packet to the second NetFPGA on the ring. Second NetFPGA then waits for its host to call MPI_Barrier.
3) The host of the second NetFPGA sends the MPI_Barrier message to its NetFPGA. The NetFPGA records the time at a certain place through the data-path.
4) The second NetFPGA now forwards the packet to the head NetFPGA, which is the head-node.
5) The head NetFPGA now knows that everyone has called MPI_Barrier. It generates a release message and forwards it to the host and the second NetFPGA at the same time.
6) When the second NetFPGA receives the release message from the head NetFPGA, it records the time again at the same place on the data-path. The difference between the two recorded timestamps provides the NetFPGA processing time for both nodes. This data is written into a packet which is sent to the host as a release message.

The time measured in this model includes the propagation delay. However, the propagation delay is negligible since we



	2	3	4	5	6	7	8	9
Ring	2.64	3.984	5.264	6.608	7.952	9.296	10.576	11.92
Binary Tree		2.624		5.312		5.312		7.936
Butterfly	2.64		3.96				5.28	

Fig. 8: Precise processing time of NetFPGA for different topologies and various number of nodes



	2	3	4	5	6	7	8	9
Ring	5.13	7.65	8.858	9.669	10.364	12.885	13.923	15.727
Binary Tree		5.453		8.453		8.927		12.175
Butterfly	4.8		5.276				7.168	

Fig. 9: Average latency introduced after MPI_Barrier is offloaded to the NetFPGA network for various topologies

used short cables to connect the hosts. Based on our precise measurements, p is 1.32μ. An increase in the number of nodes in the ring introduces 1.32μ delay, and the results are presented in Figure 8 that prove the consistency of our precise measurement. Similar model is used for the tree topology for the leaf nodes and an increase in the height of a tree introduces $2p$ latency. Based on the number of processing time, we put the estimate results for the butterfly algorithm, however they are not measured, since it is very hard to inject packets to the NetFPGAs at the same time because of the system noise of the different arrival times.

Network's average performance results after the host offloads the barrier operation to the NetFPGA still present valuable information especially for the butterfly topology. For these measurements we used a similar approach as we did for the precise measurements. However, in this case we recorded the timestamp when NetFPGA receives offload request from the host. The second timestamp is recorded when the NetFPGA sends the release message to the host. The difference is attached to the release packet. Measurements for the ring and butterfly are averaged for each host. However, for the tree case, only the results for the leaf nodes are averaged since the upper nodes are released quite earlier than the leaf nodes. Non-leaf nodes can introduce a huge bias and do not offer how the network processing time is related with the height of the tree. The results are presented in Figure 9.

5. Related Work

Zotov [20] proposes a hardware mechanism to synchronize n-dimensional mesh-connected MIMD computers. This work is one of the most comprehensive works in the literature about barrier synchronization and maps out the limitations of different synchronization frameworks. The work itself proposes a separate control networks for mesh-connected MIMD computers, and it is different from our work in three key aspects. This work proposes a separate control network for barrier synchronization. Instead, our work implements synchronization on the data network. Almasi et al. [5] are also another example claiming that it is better to have separate barrier logic and build a separate network to handle the synchronization.

Even though they are not considered as clusters of workstations, FPGA based network on chip (NoC) architectures are also related to our work. Mahr at al. [14] implement an MPI library for multiprocessor systems on a single chip. They connect the processing elements on a single chip in different ways such as a ring topology, star topology and shared bus. [15] also similarly proposes a centralized synchronization solution for 8 cores on a single chip. [7] is another example to achieve barrier synchronization on a NoC environment. According to [7] the defining feature is that the barrier release messages are broadcasted to facilitate the job of storing the source node information. Our work differs in that sense since we store the source node protocol information until the end of barrier release message. [18] discusses scalability and effect of different barrier algorithms on a NoC based platform. The algorithms investigated in this paper are central counter, combining tree and dissemination algorithm. Huang et al. [11] also focus on optimizing MPI primitives on a NoC system.

Moreover, there are some other proposals for different platforms. TMD- MPI [16] focuses on MPI_Send and MPI_Recv implementation in multi-FGPA platforms and [17] is the extension of their work to unite their design with a specialized x86 platform. Our work provides both a distributed barrier implementation and has the potential to support a variety of network topologies. Previous work [8] is the most similar to our work, but with some caveats. It is applicable only for a specific FPGA cluster architecture and topology - a tree. In contrast, we support a variety of topologies - all ring, tree and star are discussed in this paper. In addition, the communication between the FPGA systems is not using standard protocols as we do in our work. In another FPGA implementation [10], a single FPGA is used to collect barrier messages from connected hosts and to distribute them a release message when all nodes call barrier. It implements a centralized barrier algorithm employing a simple state machine. Fabric Collective Accelerator (FCA) [1] offloads the collective communication burden to Mellanox InfiniBand adapters and switches. Along with that [9] describes the implementation of a non-blocking barrier call

with CORE-Direct hardware capabilities introduced in the InfiniBand NIC ConnectX-2. They provide a list of tasks that achieves the barrier utilizing recursive-doubling algorithm. However, unlike our work, this implementation does not totally implement the barrier collective in the hardware but defines the routine that employs the primitive tasks provided by the hardware.

6. Discussion and Future Work

Our design has obvious limitations, including manual configuration. We leave these to be addressed in future work. Moreover, even though we integrated our design into the Open MPI via simply replacing the included MPI_Barrier, a more significant integration effort is necessary to preserve the architecture and semantics of Open MPI.

In our packet format we defined a field called *comm_ID*. However, it is not used in this design; the goal is to distinguish active barrier operations, which may run on simultaneously for different MPI communicators. Each of the simultaneous barrier operation will require a separate state machine. Therefore, in order to distinguish the states of active barrier synchronizations, we are planning to investigate the best way to store the *comm_ID* with their associated barrier states. We are currently investigating approaches to store the (*comm_ID*, *barrier_state*) tuples since the read and write operations for those tuples are going to be almost equal.

Moreover, we are planning to put hardware logic into the NetFPGA to learn the topology of the NetFPGA collective network and configure node roles as appropriate. This information will be propagated to the MPI environment, eliminating the hardcoding that comes with the current design and making it portable to other NetFPGA network configurations. We are also planning to achieve the self-configurability without changing any system level driver, and implementing the logic at the hardware and user-level.

7. Conclusion

In this paper, we have presented preliminary results using NetFPGAs to implement MPI_Barrier synchronization. While the hardware designs presented have some limitations, the results provide strong evidence that this is likely to be a fruitful research domain. Limitations in our initial design include lack of mechanisms for failure recovery and the need for a pre-assigned root node. Our plans include better and more robust implementations of barriers and other synchronization mechanisms, performance evaluation on real parallel code, and integration with MPI libraries.

References

[1] Fabric collective accelerator. `www.mellanox.com/products/fca/`.

[2] Open mpi: Open source high performance computing. `http://www.open-mpi.org`.

[3] Osu micro-benchmarks 4.0. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[4] Reference router walkthrough. `http://wiki.netfpga.org/foswiki/bin/view/NetFPGA/OneGig/ReferenceRouterWalkthrough`.

[5] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 1–10. ACM Request Permissions, June 2005.

[6] T. S. Axelrod. Effects of synchronization barriers on multiprocessor performance. *Parallel Comput.*, 3(2):129–140, May 1986.

[7] X. Chen, S. Chen, Z. Lu, A. Jantsch, B. Xu, and H. Luo. Multi-FPGA implementation of a Network-on-Chip based many-core architecture with fast barrier synchronization mechanism. *NORCHIP, 2010*, 2010.

[8] S. Gao, A. G. Schmidt, and R. Sass. Hardware Implementation Of MPI_Barrier On An FPGA Cluster. In *International Conference on Field Programmable Logic and Applications (FPL 2009)*, pages 12–17. IEEE, 2009.

[9] R. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. Overlapping computation and communication: Barrier algorithms and connectx-2 core-direct capabilities. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, 2010.

[10] T. Hoefler, J. M. Squyres, T. Mehlan, F. Mietke, and W. Rehm. Implementing a Hardware-Based Barrier in Open MPI. *Proceedings of KiCC*, 2005.

[11] L. Huang, Z. Wang, and N. Xiao. Accelerating NoC-Based MPI Primitives via Communication Architecture Customization. In *2012 IEEE 23rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 141–148. IEEE, 2012.

[12] I. Jung, J. Hyun, J. Lee, and J. Ma. Two-phase barrier: A synchronization primitive for improving the processor utilization. *Int. J. Parallel Program.*, 29(6):607–627, Dec. 2001.

[13] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*, pages 160–161. IEEE, 2007.

[14] P. Mahr, C. Lörchner, H. Ishebabi, and C. Bobda. SoC-MPI: A Flexible Message Passing Library for Multiprocessor Systems-on-Chips. In *2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 187–192. IEEE, 2008.

[15] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Efficient Synchronization for Embedded On-Chip Multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(10):1049–1062, 2006.

[16] M. Saldana and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 1–6, 2006.

[17] M. Saldana, A. Patel, C. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam. MPI as a programming model for high-performance reconfigurable computers. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4):22, 2010.

[18] V. F. Silva, C. de Oliveira Fontes, F. R. V. Wagner, and S. on-Chip VLSI-SoC 2012 IEEE IFIP 20th International Conference on. The impact of synchronization in message passing while scaling multicore MPSoC systems. In *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference on*, 2012.

[19] D. Tsafrir and D. Feitelson. Barrier synchronization on a loaded smp using two-phase waiting algorithms. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 8 pp–, 2002.

[20] I. Zotov. Distributed virtual bit-slice synchronizer: A scalable hardware barrier mechanism for n-dimensional meshes. *Computers, IEEE Transactions on*, 59(9):1187–1199, 2010.