Offloading Collective Operations to Programmable Logic

In this article, the authors present a framework for offloading collective operations to programmable logic for use in applications using the Message Passing Interface (MPI). They evaluate their approach on the Xilinx Zynq system on a chip and the NetFPGA, a network interface card based on a field-programmable gate array. Results are presented from microbenchmarks and a benchmark scientific application.

Omer Arap,
Lucas R.B. Brasilino,
Ezra Kissel,
Alexander Shroyer,
Martin Swany
Indiana University

Network-based data sharing and coordination are critical to both high-performance computing environments and cloud computing applications. Indeed, for network-based computing of any sort, improving the network's efficiency and efficacy can yield performance dividends. In particular, collective operations play a key role in the performance of many of these applications. Collective operations are those in which some set of processes needs to exchange data, compute a summary function over some data, or simply agree to continue. The coordinated aspect of these operations brings complications beyond just exchanging data.

The use of programmable logic (PL) offered by field-programmable gate arrays (FPGAs) is becoming more prevalent across computing. Xilinx's Zynq product line marries PL with ARM cores in a single system on a chip (SoC). Intel recently released a Xeon processor with Altera PL in the package. FPGAs have been demonstrated to assist in large-scale datacenters, accelerating Microsoft's Bing search engine and implementing key-value stores.[1] These new systems clearly have a great deal of potential. There is widespread agreement that FPGAs have moved beyond simply being a means for prototyping and into a realm where dedicated logic, with the potential for gradual evolution over time, is the goal.

With this promise, it still remains to be seen how FPGAs can best be used in general reconfigurable environments. Indeed, the key challenge facing the widespread use of FPGAs is the difficulty in programming them. Our approach has been to consider PL on network interfaces or in the network fabric. If effective network performance can be improved, many applications will benefit with little or no modification, and the question of getting data to and from the FPGA will be addressed. Finally, thinking of this functionality as a "bump in the wire" can ease development overhead by reusing the packet processing infrastructure.

Background

The architecture presented here implements collective operations implemented in programmable logic in different platforms. This section provides some background on collective operations and describes the platforms utilized in this work.

Collective Operations

Collective operations are a powerful mechanism for distributed memory programming and often play a large role in these applications' performance. These operations are an important part of the message passing interface (MPI) programming model. A collective operation is used by processes to exchange and compute over their locally held data. The set of involved processes is called a *communicator*, in which each process has an ID known as a *rank*. The performance challenges are to minimize CPU interrupts for network packet arrivals and to alleviate CPU usage due to computation. In this regard, the network substrate is a promising place for acceleration, because it can embody computation over packets during their movement.

In this article, we present a generic framework for offloading various collective operations to PL on FPGA. In our bump-in-the-wire model, the customized logic could reside essentially anywhere in the network datapath; here, it is logically part of the network interface controller (NIC).

Hardware Support for Collectives

Researchers have proposed a tremendous number of hardware-level optimizations in the past. Adam Moody and colleagues studied NIC-level reduction operations on large-scale clusters and utilized Quadrics's NIC programmability.[2] Sameer Kumar and colleagues discussed optimizations applied on a newer-generation Blue Gene/Q supercomputer.[3] Brian W. Barrett and colleagues explored the benefits of using counting events and triggered operations in Portals MPI.[4]

The CORE-Direct technology in Mellanox's ConnectX InfiniBand NICs provides primitive operations such as send, receive, and wait to generate task lists to offload collective operations. Richard Graham and colleagues initially demonstrated how task lists are generated to offload collective operations.[5] They also offloaded barrier synchronization to the NIC employing the standard recursive doubling algorithm with CORE-Direct.[6] In another work, they focused on hierarchical collective operations using CORE-Direct technology.[7] Krishna Kandalla and colleagues also used CORE-Direct to study nonblocking `MPI_Allreduce` on an InfiniBand cluster.[8]

ZedBoard with Zynq SoC

The Xilinx Zynq all-programmable SoC comprises a dual-core ARM Cortex-A9 MPCore subsystem called the *Programmable System* (PS), a complete FPGA fabric called the *PL*, a high-performance DRAM memory controller, and various peripherals. The PS and PL subsystems are connected through ARM's Advanced Extensible Interface (AXI). Such interfaces enable direct access of the DRAM by the PL and coherent accesses to the CPU caches.

The ZedBoard is a low-cost Zynq development board with several physical peripherals that the Zynq PS can control. In particular, we use the FPGA Mezzanine Card (FMC) interface to incorporate an Ethernet FMC daughter card that consists of Marvell Gigabit Ethernet PHYs, terminated in four RJ45 1G Ethernet ports.

NetFPGA SUME

The NetFPGA SUME is the latest entry in the NetFPGA family of open-source network FPGA devices.[9] The SUME is a PCIe x8 Gen. 3 card with a Xilinx Virtex-7 XC7VX690T FPGA with four 10 Gbits per second (Gbps) Ethernet SFP+ interfaces, 27 Mbyte 36-bit QDR++ SRAM running at 500 MHz, and 8 Gbyte 64-bit DRAM running at 933 MHz. The nodes hosting these cards are based on a 12-core Intel Xeon E5-2620 v3 CPU operating at 2.40 GHz. These devices typically have been used to prototype and evaluate network devices. The current implementation of PL modules and driver software is not optimized to behave as a high-performance network interface for a host. Thus, our hosts contain a Mellanox MT27500 ConnectX-3 interface with two 10 Gbps Ethernet ports, one of which is directly connected to the NetFPGA. While the NetFPGA device is physically in the host, all communication with it is via the Ethernet. This is a rather different realization of a "bump in the wire."

Implementation

Our approach is built on PL in the network path. State is shared through a common datapath between the host-side direct memory access (DMA) engine and physical interfaces. We modified modules from the NetFPGA package to support this configuration and allow compatibility with both the ZedBoard's Zynq architecture and with the Ethernet FMC.

We developed two new modules, the collective forwarding engine (CFE) and the collective instruction processing engine (CIPE); in these, the collective operations and algorithms are defined, implemented, and processed.

Host Interface

The host interface is simplified through offloading. It sends request datagrams to a predefined address and later receives the response. The Zynq's host employs a generic NIC driver customized for the Ethernet FMC ports, whereas the NetFPGA's host uses the standard Ethernet driver for the Mellanox NIC. The PL in the datapath detects and intercepts the necessary datagrams, performs the requested collective operation, and generates the response. The collective operation request to the PL is at least one User Datagram Protocol (UDP) datagram containing an offload request header and the data to be processed. The PL tracks outstanding requests by storing the various media access control (MAC) and IP addresses and UDP header fields, all of which are later used to generate the result messages for the host processes.

Our design uses a UDP-based protocol with fields providing an operation ID, operation type, message type, rank information, and information about the group ("communicator"). This is followed by the data type and count, and the data itself. Not all fields are necessary for every type of collective operation, but the same packet format is used.

Hardware Design

Figure 1 displays the PL design architecture. This section describes each entity's role and provides implementation details about the CFE and the CIPE, the core modules in the collective offload framework.

On the Zynq, we use the *AXI_Ethernet* IP core from the Xilinx Vivado catalog to interface with the four-port Ethernet extension board, the EthernetFMC. Each *AXI_Ethernet* core is connected to an *AXI_DMA* core to make each interface visible by the host, providing necessary I/O queues. By default, all the Ethernet ports are simply separate ports with no physical or logical connection existing between them in the PL. We modified the direct links between the queues to introduce our datapath and collective offload logic. From the host perspective, the host can still send standard Ethernet traffic on these as if they were separate NICs. With the datapath and CFE, a packet can be forwarded from any port to another as well as to the host itself without the host's intervention.

The *AXI_Ethernet* and *AXI_DMA* cores communicate using two AXI buses: one for data and one for status. The status bus simply triggers different streaming behavior between these modules. However, the datapath architecture that we adopted from the NetFPGA project runs only on a single data AXI stream bus. In order to make the behavior as simple as that of the NetFPGA datapath, we implemented I/O queues that first aggregate the status and datastreams and then separate them when the stream reaches the end of the datapath that interfaces with the internal input queues of either the *AXI_Ethernet* or *AXI_DMA* cores.

The *AXIS_width_converter* is responsible for converting the stream width (32 bits) in the space of *AXI_Ethernet* and *AXI_DMA* to the datapath width of 64 bits. The input arbiter and output queues modules are directly adopted from the NetFPGA design. The input arbiter module runs in a round-robin fashion and tests if there are any packets available in any of the incoming ports. If any exist, they are simply forwarded to the CFE for further processing. Conversely, the output queues module handles delivery of the packet to the output queues specified by the CFE. If the packet is destined for more than one output port, the output queues module generates a copy of the packet and then places it in the intended output queues.

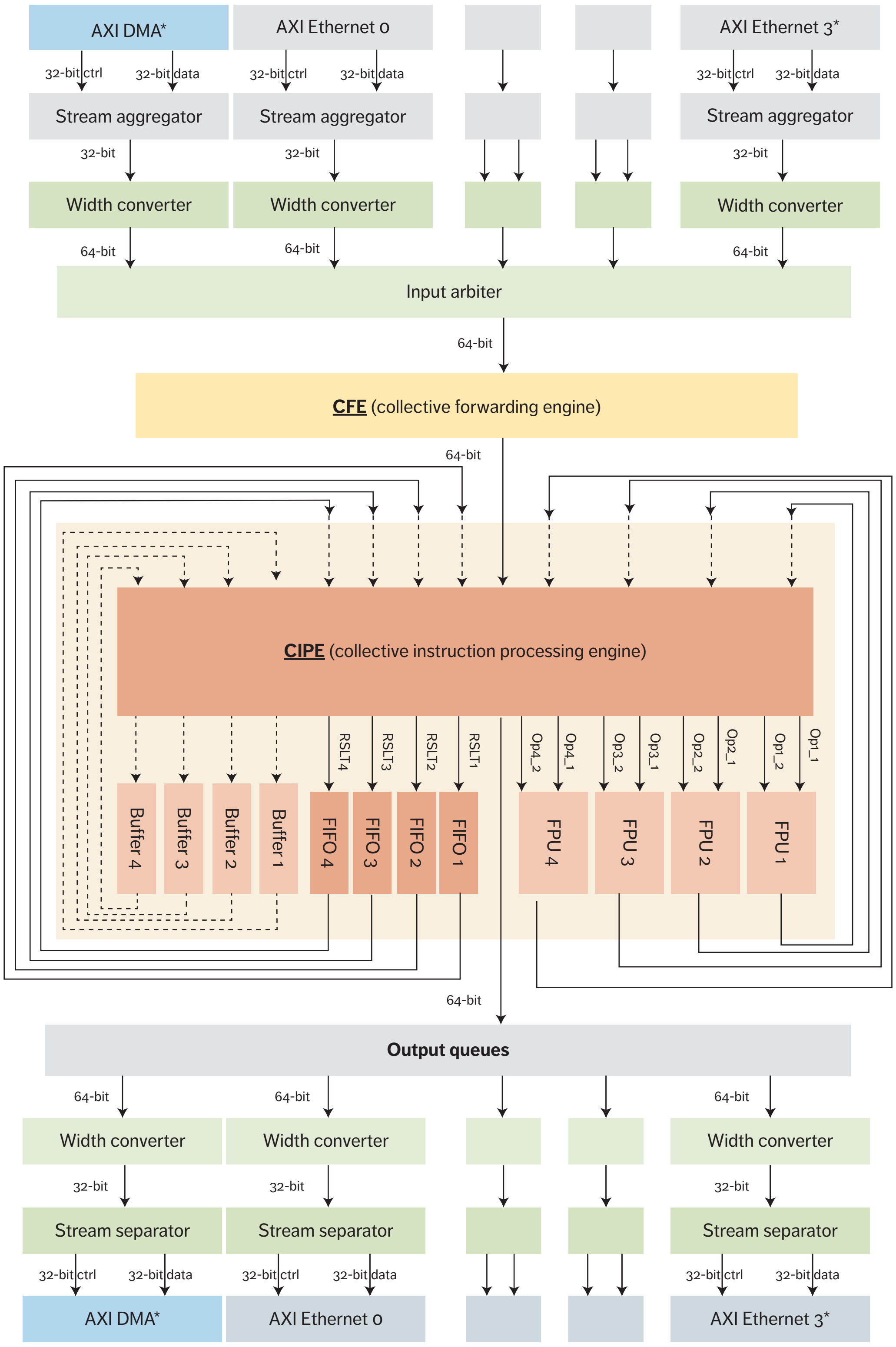

Figure 1. High-level architecture of collective operation offload design in programmable logic (PL). The inputs and outputs include a direct memory access (DMA) interface toward the host and three Ethernet interfaces. All inputs are routed through the collective forwarding engine (CFE) and the collective instruction processing engine (CIPE), and the logic can select any output.

On the NetFPGA, the design functions identically. However, the datapath uses a single 256-bit wide AXI stream; thus, neither the *AXIS_stream_aggregator* nor the *AXIS_width_converter* were employed. Moreover, the DMA function is performed by the host's Ethernet interface connected to *AXI_Ethernet_3* (both are marked with asterisks in Figure 1).

Collective forwarding engine. This module is the heart of the design and is responsible for all the algorithmic decisions for the collective operation. It has three main tasks:

- Detect which collective operation the host offloaded and which algorithm needs to be employed by assessing the offload request header fields.
- On the basis of the state of the algorithm, decide to which neighbor to forward the offload packet.
- Determine the instruction that will be performed on the received offload packet by the CIPE in the pipeline.

The CFE internally employs another submodule to recognize offload packets. If the destination UDP port number matches the predetermined port number, then the submodule extracts the offload header fields and feeds them to the CFE as input to the appropriate network-level state machine.

When the CFE receives an offload packet and it is in the idle state, meaning there have not been any collective offload operations initiated either by its host or from its neighbor PLs, the CFE registers the collective operation type, algorithm type, and collective ID in order to recognize the future packets that are necessary to run the collective algorithm to completion. If the CFE receives a packet associated with an offload operation that has already been completed, it simply ignores it.[10]

Our design supports multiple processes per host and provides options for each local rank to offload its request to the PL instead of running the local collective operation on the CPU. In this case, the CFE does not switch to the offloaded state until each rank that is running on the host issues the offload request to the PL. The CFE introduces extra states for handling the case of multiple ranks per host.

The CFE is also responsible for making routing decisions on the basis of the algorithm's state. This involves simple multicasting that lets the CFE send the packet to multiple destinations at the same time when the protocol allows it. The routing decisions are static in some algorithms, such as ring and reduce-broadcast tree-based algorithms. However, recursive doubling as well as the 2D torus algorithm are nondeterministic from the hardware perspective. We have studied the number of necessary multicast rules for algorithms in which the hosts are connected to each other via an OpenFlow switch.[11] In this work, we connect the PLs to each other directly, but multicasting is still used by having the CFE forward multiple copies of a packet.

Collective instruction processing engine. After the CFE makes a forwarding decision, it introduces a 64-bit instruction header for the entire packet and passes it to the CIPE for processing, as shown in Figure 1. The CIPE defines packet buffers for storing instruction results as well as storing data from the packet that will pass as operands to the instruction. It includes FIFOs for streaming intermediate results by creating a loop for operations we implement in the core and in the floating-point units (FPUs) for collectives that require floating-point arithmetic operations. The number of these entities is not larger than for the recursive doubling algorithm—two for the 2D torus algorithm, which can be generalized as the dimension number, and just one for the rest of the algorithms supported in our architecture.

We utilize a fully pipelined design by instantiating multiple cores of FPUs and FIFOs depending on the algorithm we employ. If the instruction involves more than two operands for the arithmetic operations, we directly feed the first operation's result as an operand to the next core while the previous core is still streaming and producing the results of the previous two operands. Therefore, we do not lose any cycles other than the initial cost of streaming.

The 64-bit instruction is divided into four parts. The last 32 bits are used for the instruction that will be processed by CIPE. The initial 32 bits are used for passing metadata from the CFE for rewriting some fields of the packet. First, we use 8 bits to pass the message type generated by the collective algorithm that will be written into the message type field of the offload header. Second, 8 bits report the packet size that is to be processed by the CFE. The remaining 16 bits define the destination UDP port that is used when we generate a response packet.

Word count and UDP port fields are necessary when we generate separate offload response packets for each rank that is running in the same host. When we reach the final state of the algorithm and are ready to send response packets to the host, the CFE applies all the header manipulations and constructs headers for Layers 2, 3, and 4 that can be sent to the first rank that resides in the host, as well as the instruction that the CIPE will perform for the final stage of the running algorithm. If necessary, this packet will also be multicast to other neighbor PLs. After the CIPE finishes executing the instruction, it buffers the result in its main on-chip buffer.

The CIPE distinguishes instructions by reading the last 32 bits of the 64-bit instruction word. The instruction space depends on the data types, reduction operations, and algorithms we support. As stated earlier, recursive doubling algorithms employ more instances of FIFOs, buffers, and FPUs, and this results in an increase in the instruction space. We currently support *MPI_INT*, *MPI_UNSIGNED_INT*, and *MPI_DOUBLE* data types and operations such as *MPI_SUM*, *MPI_MIN*, *MPI_MAX*, *MPI_BAND*, *MPI_LAND*, *MPI_BOR*, *MPI_LOR*, *MPI_BXOR*, and *MPI_LXOR* on applicable data types. Each instruction and data type combination requires a new set of instructions, and the CIPE can decode the instructions. Base instructions are defined by the algorithm's state, which is passed to the CIPE by the CFE. Based on the data type and the operation, the CIPE understands what actual instruction it must run on the received packet.

Supported collective operations. We currently support offloading *MPI_Allreduce*, *MPI_Reduce*, *MPI_Barrier*, and *MPI_Scan* collectives. *MPI_Reduce* and *MPI_Barrier* are derivatives of *MPI_Allreduce*, and we do not actually employ separate state machines for these operations. *MPI_Scan* requires specific handling that *MPI_Allreduce* algorithms cannot directly apply, because every rank participating in the collective operation receives a different result from the collective operation. *MPI_Reduce* should only deliver a response for a single target rank, but in our implementation the other ranks receive the same response as the target rank, and the software simply ignores the received packet.

Evaluation

One of our experimental systems consists of eight ZedBoards, each with an EthernetFMC card providing direct connection to up to four neighbors, depending on the test setup. The other system consists of NetFPGAs in Xeon systems. Our microbenchmarks test variations in collective operation type, algorithm, data types, reduction, number of iterations, cluster size, and communicator size. We present only a representative subset of those results here, but these are consistent across other collective types. We also present results of different flavors of MIMD Lattice Computation (MILC) quantum chromodynamics (QCD) applications, which utilize *MPI_Allreduce* extensively, and see that they do impact the application performance.

Microbenchmark

An MPI application often benefits from lower latency more than bandwidth. First, a collective operation's network traffic is unlikely to saturate a 10 Gbps network with small messages. Second, and more importantly, latency can significantly impact the time to solution, which is the critical metric for application performance. Figure 2 presents the results from the microbenchmarks running on the ZedBoard cluster. Figure 3 shows the results on the NetFPGA/Xeon system. We configured the benchmark to run 100,000 back-to-back *MPI_Allreduce* operations on *MPI_DOUBLE* data applying the *MPI_SUM* reduction on various data sizes. The figures present results for messages of various numbers of 4-byte *MPI_DOUBLE* elements. Our approach provides a clear improvement for each message size.

The running time linearly increases with the number of ranks in the software-only runs. However, offloading to the PL generates significantly faster results, relatively stable across numbers of ranks, and allows the host to handle more processes. The bar charts show that offloading for various communicator sizes

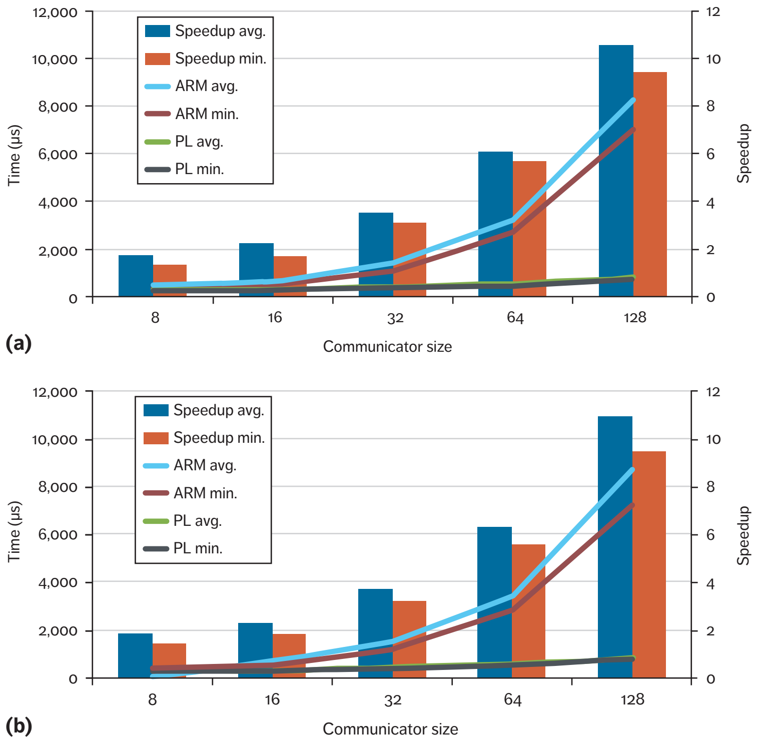

Figure 2. Microbenchmark comparison of *MPI_Allreduce* on *MPI_DOUBLE* with *MPI_SUM* operation running on the ARM host with the PL offloaded version. Zedboard (a) 8-byte and (b) 128-byte message.

(different numbers of ranks) speeds up both average and minimum latencies. The results are due to a reduction in overhead, with processing taking only 125 ns (25 clock cycles on the FPGA) for a packet to traverse both the CFE and CIPE. This eliminates all other system overheads, eliminating interrupts and data copying for all but the initiating and completion packets.

To further analyze the framework's stability, we compared average and minimum latencies on different numbers of collective sizes. The difference in the minimum and average latencies varies from around 10 percent to as much as 25 percent in the software implementation of collectives, as compared to 3 to 15 percent in our design. Overall, our implementation exhibits significantly less variability.

MIMD Lattice Computation QCD

In addition to the microbenchmark result, we evaluated our framework on high-performance computing applications that rely on collective operations in many parts of the code. MILC is a software package that studies QCD with large-scale numerical simulations. It relies on conjugate gradient solvers, which employ many collectives in sequence, summing data from each process, and send them back to all processes. Here, we tested the *clover_dynamical* and *schroed_cl_inv* applications. Figure 4 shows the results for running these applications.

Both of the applications use *MPI_Allreduce* on the *MPI_DOUBLE* datatype in various operations such as *MPI_MIN* and *MPI_MAX*, but primarily *MPI_SUM*, on a single instance of data. There are calls to the *MPI_Allreduce* with larger data sizes, and we handle these cases by spitting the data into MTU packet sizes to fully utilize our offload framework.

Results show that we improve this application benchmark's runtime by up to 45 percent in some cases, with consistent improvements throughout.

We have presented a general design and implementation of a framework for offloading MPI collective operations to PL on the network interface. By implementing a bump in the wire model together with the ability to be placed anywhere in the network datapath, the design has proven to achieve great outcomes in disparate architectures, such as a Zynq-based cluster and a Xeon-based commodity cluster. Also, we have shown that this approach offers significant speedup of collective operations in synthetic benchmarks as well as improving the performance of the combinations of collectives in real scientific applications. This type of application notably benefits from offloading by exploring a better overlap of communication with computation. In future work, we will expand our study of this approach in real applications and in larger environments. With the increased availability of PL as accelerators for general-purpose platforms, we believe that an approach that accelerates common network functionality shows great promise.

Acknowledgments

The authors thank the National Science Foundation for award CNS-1446950, which partially supported this effort.

References

1. J.W. Lockwood and M. Monga, "Implementing Ultra Low Latency Data Center Services with Programmable Logic," *Proc. IEEE 23rd Ann. Symp. High-Performance Interconnects*, 2015, pp. 68–77.
2. A. Moody et al., "Scalable NIC-Based Reduction on Large-Scale Clusters," *Proc. ACM/IEEE Conf. Supercomputing*, 2003, p. 59.
3. S. Kumar et al., "PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer," *Proc. IEEE 26th Int'l Parallel & Distributed Processing Symp.*, 2012, pp. 763–773.
4. B.W. Barrett, K.S. Hemmert, and K.D. Underwood, *Using Triggered Operations to Offload Collective Communication Operations*, tech. report, Sandia National Laboratories, 2010.
5. R. Graham et al., "ConnectX-2 InfiniBand Management Queues: First Investigation of the New Support for Network Offloaded Collective Operations," *Proc. 10th IEEE/ACM Int'l Conf. Cluster, Cloud and Grid Computing* (CCGrid), 2010, pp. 53–62.
6. R.L. Graham et al., "Overlapping Computation and Communication: Barrier Algorithms and ConnectX-2 CORE-Direct Capabilities," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. Workshop* (IPDPSW), 2010, pp. 1–8.
7. R. Graham et al., "Cheetah: A Framework for Scalable Hierarchical Collective Operations," *Proc. 11th IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing* (CCGrid), 2011, pp. 73–83.
8. K. Kandalla et al., "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers," *Proc. IEEE 26th Int'l Parallel Distributed Processing Symp.* (IPDPS), 2012, pp. 1156–1167.
9. N. Zilberman et al., "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, no. 5, 2014, pp. 32–41.
10. O. Arap et al., "Adaptive Recursive Doubling Algorithm for Collective Communication," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. Workshop* (IPDPSW), 2015, pp. 121–128.
11. O. Arap et al., "Software Defined Multicasting for MPI Collective Operation Offloading with the NetFPGA," *Euro-Par 2014 Parallel Processing*, Springer, 2014, pp. 632–643.

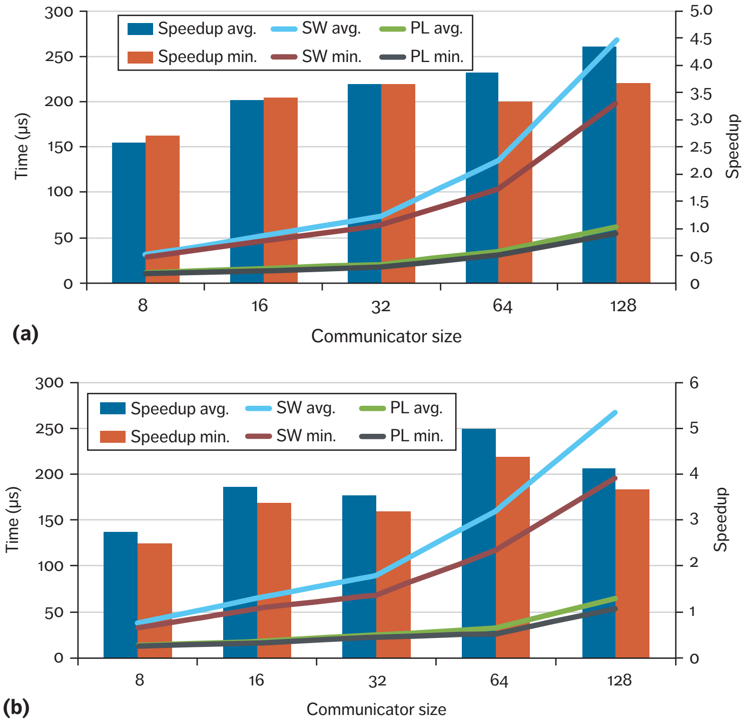

Figure 3. Microbenchmark comparison of *MPI_Allreduce* on *MPI_DOUBLE* with *MPI_SUM* operation running on the Xeon host with the PL offloaded version. NetFPGA (a) 32-byte and (b) 256-byte message.

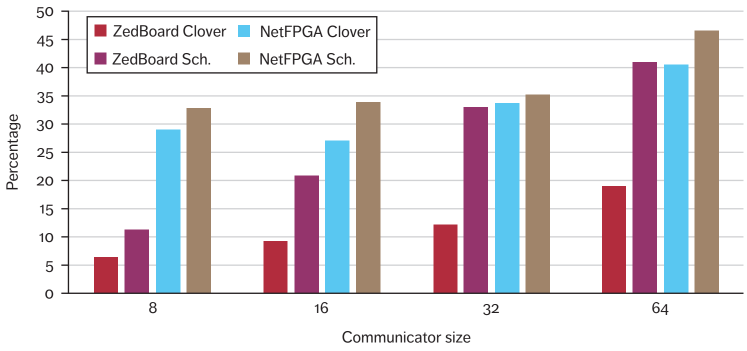

Figure 4. PL offloaded collective operation speedup compared with software implementation of message passing interface (MPI) collective operations on applications from the MIMD Lattice Computation (MILC) software suite on the ZedBoard and NetFPGA/Xeon platforms.

Omer Arap is a senior data R&D engineer at Pivotal. His research interests include optimization of collective operations in parallel execution frameworks and query optimization in massively parallel processing databases. Arap received a PhD in computer science, particularly high-performance computing, from Indiana University, where he performed the work for this article. Contact him at omerarap@indiana.edu.

Lucas R.B. Brasilino is a PhD student in the Intelligent Systems Engineering Department at Indiana University. His research interests include FPGAs, heterogeneous SoCs, network acceleration and programmability, and embedded systems. Brasilino received an MSc in computer science from the Federal University of Pernambuco. Contact him at lbrasili@indiana.edu.

Ezra Kissel is an assistant scientist in the Intelligent Systems Engineering Department at Indiana University. His research interests include embedded systems, the IoT, HPC interconnects, and network protocol design. Kissel received a PhD in computer science from the University of Delaware. Contact him at ezkissel@indiana.edu.

Alexander Shroyer is an embedded systems engineer with experience in microcontroller data acquisition devices and robotics at Indiana University. His research interests include thermal management for high-altitude electronics, fMRI-safe electronics, and pro audio electronics. Shroyer received a BS in electrical engineering from the Purdue School of Engineering and Technology. Contact him at ashroyer@indiana.edu.

Martin Swany is an associate chair and professor in the Intelligent Systems Engineering Department in the School of Informatics and Computing at Indiana University. His research interests include embedded systems, reconfigurable computing, and high-performance parallel and distributed computing and networking. Swany has a PhD in computer science from the University of California, Santa Barbara. He is a member of IEEE and ACM. Contact him at swany@indiana.edu.