Software and Hardware Co-design for Low-Power HPC Platforms

Manolis Ploumidis(✉), Nikolaos D. Kallimanis, Marios Asiminakis, Nikos Chrysos, Pantelis Xirouchakis, Michalis Gianoudis, Leandros Tzanakis, Nikolaos Dimou, Antonis Psistakis, Panagiotis Peristerakis, Giorgos Kalokairinos, Vassilis Papaefstathiou, and Manolis Katevenis

Foundation for Research and Technology – Hellas (FORTH), Heraklion, Crete, Greece
{ploumid,nkallima,marios4,nchrysos,pxirouch,yanoudis,ndimou,psistakis, perister,george,papaef,kateveni}@ics.forth.gr

Abstract. In order to keep an HPC cluster viable in terms of economy, serious cost limitations on the hardware and software deployment should be considered, prompting researchers to reconsider the design of modern HPC platforms. In this paper we present a cross-layer communication architecture suitable for emerging HPC platforms based on heterogeneous multiprocessors. We propose simple hardware primitives that enable protected, reliable and virtualized, user-level communication that can easily be integrate in the same package with the processing unit. Using an efficient user-space software stack the proposed architecture provides efficient, low-latency communication mechanisms to HPC applications. Our implementation of the MPI standard that exploits the aforementioned capabilities delivers point-to-point and collective primitives with low overheads, including an eager protocol with end-to-end latency of 1.4 µs. We port and evaluate our communication stack using real HPC applications in a cluster of 128 ARMv8 processors that are tightly coupled with FPGA logic. The network interface primitives occupy less than 25% of the FPGA logic and only 3 Mbits of SRAM while they can easily saturate the 16 Gb/s links in our platform.

1 Introduction

With cluster computation power moving towards exascale, cost both in terms of installation and operation will play a significant role in future data centers and HPC clusters. This may impose a full system reconsideration from the ground up. More specifically, the processor, the memory hierarchy, the system interconnects and the system software may require fundamental changes to meet expectations for applications' performance.

With the end of Dennard's scaling, high-end computing chips turn to architectures that mix many, simple, RISC-like low-power processors with power-efficient accelerator units [6,7,17]. These heterogeneous chip multiprocessors are expected to use 3D-stacked DRAMs in order to improve their bytes-per-flop ratios. Along this direction, we need efficient interconnects to move data across system's distributed memories. These interconnects should be efficient not only

© Springer Nature Switzerland AG 2019
M. Weiland et al. (Eds.): ISC 2019 Workshops, LNCS 11887, pp. 88–100, 2019.
https://doi.org/10.1007/978-3-030-34356-9_9

among the chips of a blade, but also among the blades of a rack or across racks. An efficient interconnect should offer low latency and high throughput, and also issue multiple outstanding transactions in order to hide the memory latency and overlap the communication with computation.

However, efficient communication does not come for free. Traditional communication protocols deplete precious processor and memory cycles. More specifically, a common rule-of-thumb states that 1 GHz of CPU power is consumed per Gbit/s of (unidirectional) Ethernet-based traffic [12]. Additionally, by copying the message payload to intermediate buffers at network injection or reception time, the memory bandwidth is consumed needlessly and the caching subsystem is stressed. In order to offload the CPU from the overheads that the network protocols induce, two options are available. The first one is to deploy smart network interface cards (NICs), while the second one is to resort interconnects with memory semantics, e.g., InfiniBand [15] Aries [3]. These solutions rely on expensive hardware, since they have to support complex and continuously evolving operations. Moreover, such network interfaces have relatively big physical dimensions, e.g., a medium-sized PCIe network card. On the other side of the spectrum, cache-coherent memory interconnects implement intricate protocols in order to maintain consistency among caches [11]. These protocols are typically implemented in hardware, because they operate in the critical path of load and store instructions; nevertheless, typically they are very inefficient on simple copy operations due to protocol-induced overheads.

In this work, we propose simple but generic network interface primitives that can be integrated in the same chip with the main processor. In order to achieve low latency and low CPU overhead, the network interface supports *virtualized, user-level initiated, protected and reliable* bulk and synchronization-oriented transfers that completely bypass the kernel. Integrating the network interface in the same chip (or package) with the processors and the memory interconnect offers the possibility to use the same block both for on-chip and system level communication, thus saving cost and reducing the silicon area footprint. In our scheme, the network interface exploits he IOMMU unit of the processor to translate process-level virtual addresses to physical memory pages, thus avoiding the need for a separate, synchronized TLB inside the network interface card, as well as the need to pin the pages involved in communication, We handle the occasional page faults that may occur in RDMA transfers by retransmitting the failing packets in hardware.

In addition, we implement a library that allows user-level accesses to network interface primitives, and an MPI runtime that supports real HPC applications. The main characteristics of our communication stack are the following:

– The required hardware is simple enough to be integrated on the processor chip, but offers all the features needed for protected low latency communication and adequately supports complex communication mechanisms (e.g., MPI).
– Our communication protocols completely offload the CPU and bypass the kernel on the communication path, thus being suitable for low-power (RISC-like) processors.

– The proposed communication architecture is demonstrated through an efficient port of the MPI standard that implements point-to-point and collective primitives that exploit the hardware capabilities.

We have implemented the network interface primitives and the full network stack in a large HPC prototype consisting of ARM processors tightly coupled with the network interface (and other accelerators) implemented in the Zynq Ultrascale+ FPGAs. We run real HPC applications (e.g., LAMMPS [1]) on a 128-core cluster. Our results show that:

– Our implementation of the network interface blocks in Xilinx Ultrascale+ FPGA utilizes only 25% (70 K LUTS) of the available LUTS and 10% of the available BRAMS (3 Mbits).
– Our communication architecture is able to provide efficient communication mechanisms to HPC applications.
– The MPI implementation introduces low overheads, while the exploitation of the eager protocol gives us an end-to-end latency of just a bit more than 1 μs (see Sect. 5).
– MPI applications show almost linear scaling in a many real-world workloads.

The remainder of this paper is organized as follows. In Sect. 2, we describe our network interface primitives. In Sect. 3, we present the user-level library and the MPI port. Next, in Sect. 4 we describe our evaluation platform and our performance evaluation results. Finally, we conclude in Sect. 5 with discussions and future work items.

2 Network Interface Primitives

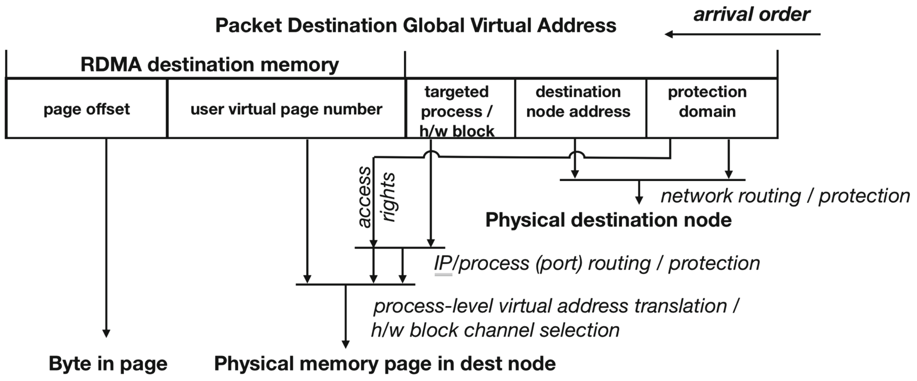

Packet Destination Global Virtual Address
arrival order
RDMA destination memory
page offset
user virtual page number
targeted process / h/w block
destination node address
protection domain
access rights
network routing / protection
Physical destination node
IP/process (port) routing / protection
process-level virtual address translation / h/w block channel selection
Byte in page
Physical memory page in dest node

Fig. 1. The global virtual address space assumed in this work as carried on a network packet.

We consider an environment in which all memory locations belong to a Global Virtual Address Space (GVAS) and can be addressed by network packets. In our prototype, a GVAS address is 80 bits. The network interface provides mechanisms that allow to multiple software and hardware processes to initiate multiple concurrent transfers to mailboxes and processes address space (both uniformly addressable within our GVAS) without kernel involvment.

Figure 1 depicts an example of the GVAS, as used to specify the destination address of the first payload byte in an RDMA network packet. In order to access a location in the GVAS, a packet needs to specify a *protection domain id (PDID)*, 16 bits in our prototype, which is used by the hardware to safely check the initiator's access rights on particular GVAS locations. The PDIDs allow the administrator to create virtual groups of collaborating processes that share their virtual memory space, while protecting against unwanted accesses when we consolidate multiple such groups of processes on the platform. In order to shield against attacks, the PDID carried in network packets can only be set by network interface hardware, using registers set by systems software based on the process that requests a network channel (virtual interface).

The global virtual address of a packet additionally specifies a node ID, i.e. the physical location of the node in which the virtual location is contained. In our prototype, we have 22 bits node IDs, allowing 4M nodes, and 42 address bits for 4 Terabytes within each node. With 16-bits PDID, we can have up to 64K parallel instances of this deployment sharing the cluster. In principle, the node ID should also be virtual in order to allow process migration [10]. However, in our current prototype we assume a static mapping of GVAS node IDs to physical nodes – i.e. endpoints of the interconnect. Within the node, the GVAS expands further to identify a specific local port (3 bits in our prototype), such as a peripheral/accelerator or a (CPU) process with private memory space. Finally, the virtual address field can specify a channel (or virtual interface) of a peripheral or a byte in the virtual memory of a CPU or accelerator process (39 bits). Accesses to virtual locations from the network interface may be cacheable, reading memory locations that have updates present in caches or invalidating cache entries in case of writes.

Our network interface provides separate hardware primitives (i) for bulk data (RDMA) transfers and (ii) for latency-sensitive control messages.

Packetizers and Mailboxes: For fast notification (control) messages, we have built a *virtualized packetizer* and a *virtualized mailbox*. These blocks have been designed for latency-critical control messages. At the sending node, the virtualized packetizer offers 64 virtual interfaces (pages) that can be allocated to different threads and processes. Each page provides four (4) channels, where each channel can be used by the owner process to transfer a packet and monitor its execution by polling on specific bits of the page (i.e. load command). Each channel can be in one of the following states: ongoing, acknowledged, negatively acknowledged, timed out. A process acquires a page of the packetizer from a kernel driver. The driver writes into a special hardware register for this page the PDID of the requesting process, and returns to the process a virtual address which is mapped to the physical address of the packetizer page.

A transfer starts with the process filling up the payload of the transfer into the channel address using (posted) store commands, and is commenced when it writes the payload size and the destination address on a designated address of the channel. The hardware is responsible for creating a network packet that carries the user-defined payload and destination address, as well as the appropriate PDID. All hardware transactions additionally contain a unique transaction id that is filled up by the hardware and is used to match the end-to-end acknowledgements. A packetizer may target any location of the GVAS, such as the process virtual address or a virtual mailbox.

The virtualized mailbox is hardware block that consists of 64 virtual interfaces. The mailboxes keep their data in DRAM and in the L2 caches, but their tail pointers are maintained and updated by the hardware while their read pointers by the user-level libraries that read the data. Processes can acquire mailboxes from a kernel driver which associates each virtual interface with the PDID of the corresponding process group. When a packetizer sends a message to a virtualized mailbox, the receiving hardware checks the packet's PDID and tries to match it against that of the virtual mailbox, generating a NACK when these do not match. The mailbox may also drop a packet when it is full. Otherwise, the packet is enqueued into the virtual mailbox and an ACK is generated and routed to the issuing packetizer node. User processes can poll for new arrivals in their mailboxes by reading from a virtual addresses that has been memory-mapped to the physical address of their virtualized mailbox.

Simple RDMA: For large data transfers, we have built a simple virtualized RDMA engine, with coordinated units running at the sending (TX) and receiving (RX) endpoints. The RDMA engine implements an effective, hardware-level multi-path transport that provides reliability guarantees, allowing to completely bypass the kernel stack on I/O operations.

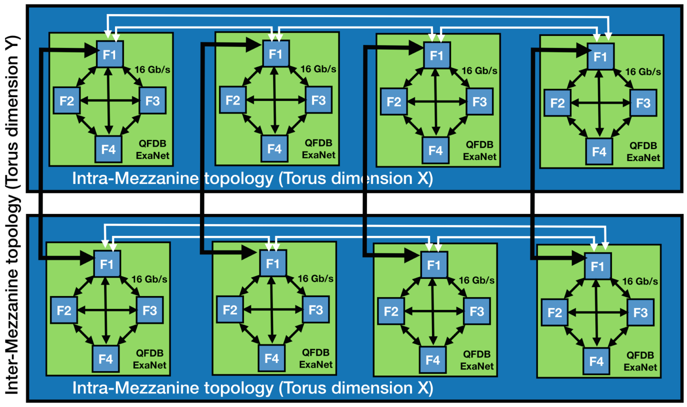
Every (Write or Read) RDMA operation transfers a message between two GVAS locations: the source, which in our implementation is always local to the TX engine that will realize the transfer, and the destination, which is local to the RX engine of the transfer. A detailed description of the hardware RDMA engine is subject of a future work item.

When running MPI applications, we can think that every process has its own virtual space, which is a subspace of the system-level GVAS. All memory accesses issued by the network interface go through a multi-channel I/O-MMU[1] to translate the virtual to physical addresses and to verify the access rights of the initiating processes using the PDID associated with a hardware channel or carried along a network packet. The I/O-MMU keeps a local translation lookaside buffer (TLB) and a page walker engine that runs through the page table of the targeted process to handle TLB misses. Page faults generate NACKs that are propagated to the source in order to retry the transaction.

[1] System MMU in the case of ARM processors.

3 HPC Prototype

This section provides a brief overview of the HPC prototype where the proposed software-hardware codesign was ported and evaluated. This prototype has been developed within the ExaNeSt project – for a more detailed description and the full potential of the project please refer to [9].



Inter-Mezzanine topology (Torus dimension Y)
Intra-Mezzanine topology (Torus dimension X)
F1
F2
F3
F4
16 Gb/s
QFDB ExaNet

Fig. 2. HPC platform prototype used for porting and evaluating the proposed software-hardware codesign.

A high-level view of the prototype used in this study is depicted in Fig. 2. This currently consists of two mezzanines each one carrying four (4) *Quad FPGA Daughter Boards (QFDBs)* for a total of 32 FPGAs or 128 ARMv8 cores. The QFDBs, depicted through green boxes, are connected in a 2D Torus topology. The prototype is still growing in size – at the time of writing, it uses a 3D Torus topology to connect six (6) mezzanines, and, after the next planned update, it will reach 12 mezzanines, (48 QFDBs, 192 FPGAs, 512 ARM cores).

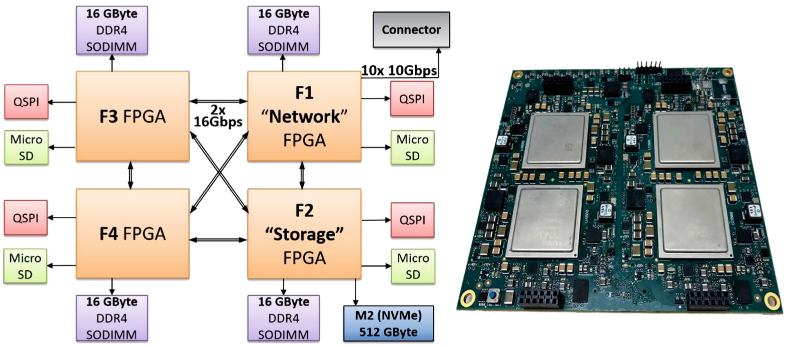


Fig. 3. QFDB block diagram and actual board.

As shown in Fig. 3, each QFDB provides four (4) interconnected FPGAs, a large amount of memory, and one SSD, within a small footprint (120 mm x 130 mm). The FPGAs are Xilinx Zynq Ultrascale+ devices (ZCU9EG), featuring four (4) ARM-A53, 16-GByte DDR4, along with a rich set of hard IPs and reconfigurable logic. Effectively, having two mezzanines, the prototype used for evaluation purposes in this study consists of 128 low-power ARMv8 cores.

There are two GTH transceivers (16 Gb/s each) for each FPGA pair, offering a total bandwidth of up to 32 Gbps. The top right FPGA, referred to as the Network FPGA, provides connectivity to the external world through ten (10) GTH links. The bottom right FPGA, named the Storage FPGA, provides connectivity to the NVMe memory through PS-GTR transceivers implementing a 4xPCIe Gen 2.0 channel. Finally, each FPGA can boot from an attached NOR flash, accessible through QSPI.

Our platform supports two networks, namely, Exanet and 10G Ethernet. Exanet is a custom packet-based hierarchical interconnect realized over high speed serial links, developed by FORTH and INFN (Istituto Nazionale di Fisica Nucleare) within the ExaNeSt project, using as baseline APENet [4]. Within each QFDB, there is an all-to-all connectivity, both for Exanet and Ethernet traffic, shown using black arrows among $F1$, $F2$, $F3$, and $F4$ in Fig. 2. As discussed above, in ExaNet we use a multi-dimensional Torus topology to connect the QFDBs, whereas for Ethernet, we employ external commercial switches with one 10G Ethernet interface per QFDB.

4 User-Level Communication Library

Part of the proposed architecture is a user-space API that allows user-level access to the hardware blocks described in Sect. 2, that is, a simple RDMA engine, virtualized packetizer and virtualized mailbox. The simplicity of the hardware blocks provided by the network interface allows for a minimal but powerfull user-space API.

The virtualized mailbox/packetizer hardware blocks that are exposed through the user space API allow the realization of user-level low-latency atomic message delivery. More precisely, this API gives to application threads the functionality for attaching a virtual interface of the virtualized mailbox and of the virtualized packetizer that reside on the local compute node. Notice that only the functionality for attaching/detaching the virtual interfaces involves the kernel of the operating system. Specifically, a kernel driver that is responsible for the mailbox hardware block exposes a set of hardware registers to threads and processes. With this set of hardware registers, the users can send and/or receive small messages to/from remote processes in a user-level manner. The provided API allows applications to atomically send messages of up to 64-bytes to any virtual interface (or process address) of any remote node. Furthermore, in case that a thread has acquired an interface of the virtualized mailbox, it is also able to receive messages sent by remote nodes in the same protection domain. The messages generated by the packetizer wait for an end-to-end acknowledgment, and can be retriggered in case of time-out.

The user-space API also allows applications or runtimes (such as the MPI implementation) to perform RDMA protected transfers with virtual local and remote addresses, without involving the kernel of the operating system. The API is quite minimal, providing calls for contructing a remote virtual address, inserting a descriptor, and for polling for completion. Both RDMA read and RDMA write operations are supported.

5 MPI Implementation over the Proposed Architecture

This section presents a partial implementation of the MPI standard, tunned to take advantage of the hardware design described in Sect. 2. This MPI implementation is realized on the HPC prototype described in Sect. 3, which has been developed in the ExaNeSt project [2]. From now on, we will refer to the proposed MPI implementation as *Exanest-MPI*. This MPI implementation is characterized as partial since it provides support for point-to-point and collective primitives while it does not support one-sided and MPI-IO communications. These primitives are delegated to an MPICH library (going over the Ethernet network) that is slightly modified to expose a communicators context ID – this is a 16-bit field needed to distinguish messages on different communicators.

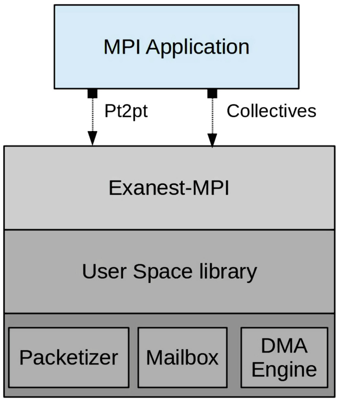


Fig. 4. Proposed co-design components.

In Fig. 4 we provide an overview of how Exanest-MPI exploits the proposed software-hardware co-design to provide an efficient MPI library. The hardware blocks that it relies on are the virtualized mailbox/packetizer and the RDMA engine. Access to these blocks is provided through the user-space library which gives user-level access to hardware blocks avoiding the kernel intervention. In short, the packetizers and mailboxes are exploited to relay control traffic, such as, message envelopes and MPI acknowledgments, in a low latency manner. For data transfers, we exploit the high throughput offered by the RDMA engine.

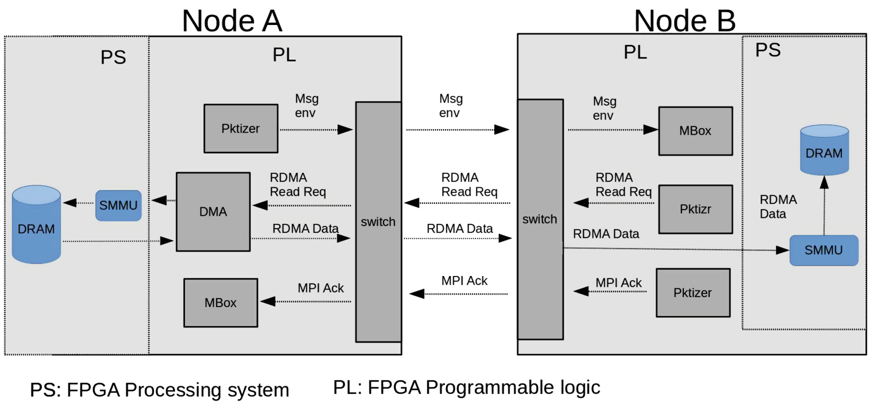


Fig. 5. Rendezvous protocol demonstrated through a pair of matching MPI_Send and MPI_Recv.

As far as point-to-point primitives are concerned, we have implemented both a rendezvous and an eager protocol. Figure 5 depicts the steps along with the hardware blocks involved in realizing the rendezvous protocol. Recall that multiple prototype nodes are arranged in a torus topology through a custom packet-based interconnect, where each node consists of four (4) Zynq FPGAs or 16 Armv8 cores. Inter-node traffic passes through the routers contributed by INFN within the ExaNeSt project [2]. The scenario depicted is as follows: a process running on node A issues an *MPI_Send* while a second process belonging to node B posts the matching receive. For control messages the low latency mechanism provided by the virtualized mailbox hardware block is used. In the above figure, process on node A uses the virtualized packetizer to send a message envelope to its peer process (arrows annotated with *Msg env* label in the above figure). Message envelopes for MPI messages typically contain the following information: *source*, *destination*, *tag*, and *communicator*. In our case, the message envelope also carries the buffer advertised by the process issuing the send operation. The message envelope is sent from a packetizer of node A to a mailbox in node B. As soon as the process on node B receives the message envelope in the virtualized mailbox used by the MPI library, it initializes an RDMA read from the remote virtual address extracted from the message envelope to the local virtual address described in the *MPI_Recv* call. The RDMA read requests is routed to node A, and is executed as an RDMA write operation by the RDMA TX engine. Note that reading (writing) data from (to) memory goes through the SMMU to translate virtual to physical addresses. When the RDMA read completes on node B, the process running on it will use a virtualized packetizer to send an MPI acknowledgment back to the peer process on node A[2].

[2] Note that, at the hardware level, the transfer has been separately acknowledged from the RX engine on node B to the TX engine on node A.

The low latency communication mechanism provided the packetizer and mailbox is also leveraged to implement an *eager protocol.* Messages of up to 32 bytes are sent through the packetizer without initializing an RDMA transfer.

In the current version of the MPI library discussed, almost all collective operations except for less frequently used ones are supported. The algorithms used to implement collective operations are the ones also used in MPICH and summarized in [16]. In the current MPI library version, for the case *MPI_Bcast*, *MPI_Reduce*, and *MPI_Allreduce*, the same algorithm is employed for small and large messages. Part of ongoing work is the validation of more efficient algorithms for the aforementioned primitives and large message sizes.

Evaluation

In this section we present the evaluation of the proposed communication architecture. Results are derived through both micro-benchmarks and a scientific MPI application triggering all components from all layers of the proposed co-design. Recall that Fig. 4 shows that point-to-point and collective MPI primitives are handled by Exanest-MPI using the proposed user-space communication library to exercise the implemented hardware blocks of the Exanet network.

In order to assess the performance of the proposed hardware and software blocks, the well known osu latency and osu allreduce micro-benchmark were used [14]. Two different sets of runs were performed for each of them. The first one uses the standard rendezvous protocol (Table 1) and the second one exploits the eager protocol available in Exanest-MPI (Table 1). For each set of runs, message size was set to 8 bytes and each run involved nodes at different distances starting from nodes in the same QFDB up to nodes that are 3 hops away. Note also that distance in terms of hop is only meaningful for the case of the ExaNet network (2D torus described in Sect. 3. Nodes that are 2 hops away in terms of ExaNet are connected through the saem Ethernet switch). Table 1 shows that the latency between nodes that are 1 hop away is almost higher than 6 µs when no eager protocol is employed. However, eager protocol cuts down this latency to 1.5 µs. This is so because the eager protocol eliminates the overhead of the initialization of a DMA transaction whenever a packet of small size is transmitted. For processes that are placed in the same QFDB board, osu latency value becomes as low as 1.21 µs. As Table 1 also shows, ExaNest MPI outerperfors MPICH utilizing TCP/IP over Ethernet (referred to as *standard MPICH* hereafter). Even for the case of 3 hops travelled for the case of ExaNest MPI, osu latency achieved by ExaNeSt MPI with eager protocol enabled is almost 20 times lower than the one achieved by standard MPICH. As this table also shows, for the case of the osu allreduce microbenchmark, ExaNest MPICH achives alomst 10 times lower latency than standard MPICH. The first reason for this performance benefit is that ExaNest MPI relies on the user-space API described in Sect. 4 which offers user-level access to the hardware blocks. In contrast with TCP/IP over Ethernet, the overhead of a system call is avoided. Secondly, the hardware blocks exploited by ExaNeSt MPI are part of the reconfigurable logic of each node (also discussed in Sect. 2) which means that communication does not share the CPU with computation as in the case of standard MPICH. There is ongoing work in order to lower the latency under the barrier of 1 µs.

Table 1. Osu latency and osu allreduce (usecs) with Exanest-MPI

ExaNet distance	Osu latency (8bytes)			Osu all reduce (8bytes)		
	No eager ExaNeSt MPI	Eager ExaNeSt MPI	TCP MPI	No Eager ExaNeSt MPI	Eager ExaNeSt MPI	TCP MPI
Intra-qfdb	4.97	1.21	37.2	19.13	6.69	70.5
Inter-qfdb (1 hop)	6.06	1.50	39.7	20.28	7.21	70.8
Inter-qfdb (2 hop)	7.47	1.75	39.4	21.41	7.71	69.8
Inter-qfdb (3 hop)	8.43	1.99	40.1	22.34	8.12	70.6

The proposed software-hardware co-design approach along with Exanest-MPI was also evaluated using the LAMMPS [1] scientific benchmark. LAMMPS is a state-of-the- art molecular dynamics code [13]. From the LAMMPS benchmark suite, the *rhodopsin* problem was selected. Different runs are obtained by varying the number of nodes (N) and the number of timesteps which are expresses as $N * 100$. For every *rhodopsin* run, 3 OpenMP threads were used. As in the case of the osu latency micro-benchmark, two different set of runs were performed, one using the standard rendezvous protocol and another exploiting the eager protocol available in Exanest-MPI. For each run Table 2(a) reports the number of timesteps per second that were achieved with higher values indicating better performance. This table also shows that the throughput achieved by the *osu_bibw* microbenchmark increases with message size.

Table 2. (a) LAMMPS performance (Timesteps/s) with Exanest-MPI (no eager protocol), (b) Osu bibw intra QFDB (Bandwidth (MB/s))

Num of FPGAs	no eager	eager
1	1.041	1.041
2	2.024	2.023
4	3.811	3.815
8	7.228	7.242
16	13.302	13.349
32	23.319	23.364

(a)

Msg size (bytes)	Bandwidth (MB/s)
8	7.29
64	24.90
256	88.88
1K	341.8
4K	863.8
16K	1100.3
1M	2221.0

(b)

6 Conclusions and Future Work

In this work, we described a hardware-software co-design for future low power processors. We have designed and implemented in hardware simple network interface primitives that are integrated close to the CPU, occupy less than 25% of a

Xilinx Ultrascale+ FPGA, and are suitable for bulk memory-to-memory transfers and for fast control messages. The hardware blocks are virtualized, and exploit the IO MMU to allow zero-copy, user-level initiated, reliable transfers, thus minimizing latency and the CPU overhead. On top of the NI primitives we derived an efficient MPI implementation that achieves a nearly 1 µs OSU microbenchmark latency in a cluster of 128 ARM cores. Ongoing work includes deriving performance results on a liquid cooled version of the HPC platform described populated with 512 ARM cores.

Acknowledgments. This work is supported by the European Commission under the Horizon 2020 Framework Programme [8] for Research and Innovation through the EuroEXA project [5] (g.a. 754337), the EU H2020 FETHPC project Exanode (g.a. 671578) and the ExaNeSt project (g.a. 671553) [2].

References

1. LAMMPS Molecular Dynamics Simulator. Sandia National Laboratories. https://lammps.sandia.gov
2. The ExaNest project. European Exascale System Interconnect and Storage. GA-671553. www.exanest.eu
3. Alverson, B., Froese, E., Kaplan, L., Roweth, D.: Cray xc series network. Cray Inc., White Paper WP-Aries01-1112 (2012)
4. Ammendola, R., et al.: Apenet: a high speed, low latency 3d interconnect network. In: cluster, p. 481. Citeseer (2004)
5. EuroEXA: European Exascale System Interconnect and Storage. https://euroexa.eu/
6. Feldman, M.: Fujitsu switches horses for post-k supercomputer, will ride arm into exascale. Recuperado de (2016). https://www.top500.org/news/fujitsu-switcheshorses-for-post-k-supercomputer-will-ride-arm-intoexascale
7. Fu, H., et al.: The sunway taihulight supercomputer: system and applications. Sci. China Inf. Sci. **59**(7), 072001 (2016)
8. HORIZON 2020: The EU Framework Programme for Research and Innovation. https://ec.europa.eu/programmes/horizon2020/
9. Katevenis, M., et al., N.C.: The exanest project: Interconnects, storage, and packaging for exascale systems. In: 2016 Euromicro Conference on Digital System Design (DSD), pp. 60–67, August 2016. https://doi.org/10.1109/DSD.2016.106
10. Katevenis, M.G.: Interprocessor communication seen as load-store instruction generalization. In: The Future of Computing, essays in memory of Stamatis Vassiliadis. In: Bertels, K., et al. (eds.) Delft, The Netherlands. Citeseer (2007)
11. Katz, R.H., Eggers, S.J., Wood, D.A., Perkins, C., Sheldon, R.G.: Implementing a cache consistency protocol, vol. 13. IEEE Computer Society Press (1985)
12. Leitao, B.H.: Tuning 10gb network cards on linux. In: Proceedings of the 2009 Linux Symposium, pp. 169–185. Citeseer (2009)
13. LAMMPS Benchmark suite. http://lammps.sandia.gov/bench.html
14. OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
15. Pfister, G.F.: An introduction to the infiniband architecture. High Perform. Mass Storage Parallel I/O **42**, 617–632 (2001)

16. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005). https://doi.org/10.1177/1094342005051521. http://dx.doi.org/10.1177/1094342005051521
17. Yokokawa, M., Shoji, F., Uno, A., Kurokawa, M., Watanabe, T.: The k computer: Japanese next-generation supercomputer development project. In: IEEE/ACM International Symposium on Low Power Electronics and Design, pp. 371–372. IEEE (2011)