StRoM: Smart Remote Memory

David Sidler*
Microsoft Corporation
Redmond, WA

Zeke Wang*
Collaborative Innovation Center of Artificial Intelligence
Zhejiang University, China

Monica Chiosa
Systems Group
Department of Computer Science
ETH Zurich, Switzerland

Amit Kulkarni
Systems Group
Department of Computer Science
ETH Zurich, Switzerland

Gustavo Alonso
Systems Group
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Big data applications often incur large costs in I/O, data transfer and copying overhead, especially when operating in cloud environments. Since most such computations are distributed, data processing operations offloaded to the network card (NIC) could potentially reduce the data movement overhead by enabling near-data processing at several points of a distributed system. Following this idea, in this paper we present StRoM, a programmable, FPGA-based RoCE v2 NIC supporting the offloading of application level kernels. These kernels can be used to perform memory access operations directly from the NIC such as traversal of remote data structures as well as filtering or aggregation over RDMA data streams on both the sending or receiving sides. StRoM bypasses the CPU entirely and extends the semantics of RDMA to enable multi-step data access operations and in-network processing of RDMA streams. We demonstrate the versatility and potential of StRoM with four different kernels extending one-sided RDMA commands: 1) Traversal of remote data structures through pointer chasing, 2) Consistent retrieval of remote data blocks, 3) Data shuffling on the NIC by partitioning incoming data to different memory regions or CPU cores, and 4) Cardinality estimation on data streams.

ACM Reference Format:
David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387519

*Work done while the authors were at ETH Zurich.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387519

1 Introduction

Large scale data processing, especially in cloud environments, is commonly implemented in a distributed manner. Such systems require, on the one hand, low latency communication to coordinate across nodes and, on the other hand, high bandwidth interconnects to transfer data between nodes. For this reason, recent years have seen an increasing adoption of Remote Direct Memory Access (RDMA) for distributed applications to take advantage of its low latency and ability to bypass the CPU [7, 13, 25, 36, 49, 56]. Capitalizing on the features provided by RDMA, in this paper we explore the implementation of a form of near-data processing by offloading access and data processing operations to the NIC as an extension of RDMA verbs. We do this through a first prototype of a network based, near-data processing system, StRoM, implemented as an FPGA-based RoCE v2 NIC. StRoM provides a mechanism to deploy arbitrary processing kernels on the NIC that, through RDMA, have direct access to the memory resident buffers and can be invoked remotely over the network. These kernels can be used to accelerate data access operations as well as to implement in-network processing of data streams as they are being sent or received. StRoM can then be used in a variety of contexts: disaggregated memory, remote memory, network attached storage, etc.

As an example of the advantages of StRoM, consider a GET operation in a distributed key-value store, a data access operation that currently either requires multiple network round trips when implemented with one-sided RDMA verbs [13, 36] or involves the remote CPU when using two-sided semantics [25]. In StRoM, the GET is implemented as a kernel on the remote NIC invoked in a single network round trip to lookup the key in the hash table and retrieve the requested value from host memory. It combines the benefit of two-sided and one-sided communication by executing the request in a single round trip (like two-sided RDMA operations) without involving the remote CPU (like one-sided RDMA operations). When operating on data streams, the StRoM kernel acts as a bump-in-the-wire and can execute operations such as filtering, aggregation [55], partitioning [53], and gathering of statistics while data is transmitted [20]. Since many of these operations are more data- than compute-intensive, they can

be efficiently mapped to the data-flow architecture of the NIC. Offloading these operations reduces data movement and saves CPU cycles that can be re-assigned to other operations.

StRoM includes: (1) an open source RoCE v2 implementation on an FPGA-based NIC working at 10 Gbit/s and 100 Gbit/s; (2) StRoM specific verbs for invoking kernels over RDMA; (3) a programmable fabric to deploy application-specific StRoM kernels supporting line-rate processing up to 100 Gbit/s while adding negligible latency; (4) a well-defined interface for kernel development, facilitating portability; and (5) a way for user functions to directly operate on the data payload (i.e., data streams) instead of being exposed to raw network packets, easing the development and integration process. To further remove obstacles to using StRoM, its kernels can be programmed in high-level synthesis (C/C++).

We demonstrate the capabilities of StRoM through detailed discussion and experimental analysis of four use cases: 1) traversal of remote data structures, 2) consistency verification when accessing remote data blocks, 3) on-the-fly data shuffling when writing to remote memory, and 4) cardinality estimation on streaming data. Given the increasingly wide deployment of custom and specialized hardware in data centers, we see our design as an efficient way to offload computation to the network card. The architectural design we propose is not hypothetical. Our design fits well in recent deployments in commercial clouds. In Microsoft's Catapult, the FPGA sits on the data path in front of the network card acting as a SmartNIC [14] which can be extended with the features discussed in this paper. The same applies to IBM's cloudFPGA [54], where the FPGA is an independent network-attached node providing accelerator services. To our knowledge, this is the first description and open source[1] implementation of RDMA on an FPGA and its extension for use as an in-network data processing platform.

2 Background

2.1 RDMA & RoCE

Remote Direct Memory Access (RDMA) is a mechanism to directly access data in the main memory of a remote machine. When RDMA operations involve the CPU on both the sender and the receiver, they are called two-sided (`SEND` and `RECEIVE`). Two-sided operations can be used to implement Remote Procedure Call (RPC) style of communication and enforce synchronization of sender and receiver. One-sided operations, `WRITE` and `READ`, access memory directly from the NIC, by performing DMA to a remote virtual memory address. RDMA is available over network fabrics such as Infiniband or Ethernet (RoCE). In RoCE, the physical layer (Infiniband link layer) is changed so that Infiniband (IB) packets are transmitted as Ethernet frames. RoCE is routable since version 2 by encapsulating IB packets into IP/UDP packets. In StRoM we use RoCE v2 over IPv4 and UDP on 10 G and 100 G Ethernet networks. We focus primarily on one-sided operations since the processing kernels we implement provide the equivalent of two-sided semantics without involving the remote CPU, hence enabling the best of the two options: single round trip interaction as in two-sided operations and not involving the CPU as in single-sided operations.

[1] https://github.com/fpgasystems/fpga-network-stack.git

2.2 FPGA

Field-programmable gate arrays (FPGAs) are programmable chips that can be used to implement arbitrary digital circuits [50]. Once programmed, they behave like an integrated circuit (ASIC) but operating at a lower clock frequency and being less power efficient. FPGA logic is traditionally implemented in hardware description languages, for instance Verilog and VHDL. Recently, compilers and tools for higher level languages such as OpenCL or C/C++ have started to emerge [2, 57]. Internally, FPGAs mainly consist of registers, lookup tables (LUTs), on-chip memory (BRAM), and digital signal processing units (DSPs). These components are connected over an internal reconfigurable network.

2.3 StRoM vs Programmable Switches

There is a wide range of research [10, 12, 23, 47] and products [37] around programmable switches in the context of software defined networks (SDN). Since SDN focuses on packet routing, operations are typically limited to the granularity of a packet, are triggered based on fields in the packet headers, and applied to the packet or its header, e.g., dropping the packet or rewriting the header. P4 [9] is a high-level language to program the packet forwarding plane and specify what actions to take for a given packet header. A fundamental difference between StRoM and programmable switches is that StRoM builds on RDMA's ability to access the host memory directly from the local NIC, something not possible from a switch. Thus, the in-network processing functionality provided by StRoM is complementary and different than that of programmable switches.

There are numerous efforts to explore the use of programmable switches beyond the scope of SDN for instance to push data processing into the network [22, 30, 32] or accelerate coordination among nodes [12, 21]. In case of data processing, a main drawback of programmable switches is the limited memory that can be used to maintain state. This restricts statefull operations, imposes assumptions on the data set, or requires fall-back mechanisms in case the available memory is exceeded. Additionally, state migration or recovery might be necessary in case flows are re-routed or a switch fails. Similarly, reliable network protocols require state for each flow to track packets making data reduction operations, such as aggregation, compression, or filtering at the switch highly complex or unfeasible. By implementing StRoM on the NIC, we avoid these limitations since the processing kernels are not exposed to the notion of network packets. Additionally,

StRoM kernels can access the host memory to store partial results or state.

2.4 StRoM vs Existing SmartNICs

SmartNICs are starting to appear in a number of applications. Liu et al. [34] provide an extensive analysis of their characteristics and behavior. In all cases, these SmartNICs are based on arrays of wimpy ARM cores (up to 16 at the time of writing) plus additional dedicated hardware acceleration modules for a variety of tasks (e.g., NVM controllers). Compared to such SmartNICs, StRoM is an *on-path* SmartNIC, processing packets directly on their communication path and, thus, capable of processing all incoming/outgoing packets. Off-path SmartNICs such as Mellanox BlueField I/O Processing Unit [35], need to filter the packets to re-route them to the processing cores, before any "smart" functionality can be applied. The filtering takes place on a NIC switch and reduces the load on the ARM cores which have limited capacity, but introduces additional latency.

Overall, StRoM is a simpler SmartNIC in the sense that it just involves an FPGA on the communication path, mirroring Microsoft's Catapult design [14]. Commercial SmartNICs are complete Systems-on-Chip, involving far more hardware (not only arrays of cores but also dedicated ASICs for network processing, packet switching, memory controllers, potentially storage controllers, etc.). As such, they can do more but they also cost more, require more energy, and are more complex to deploy, program, and use.

2.5 Smart Memory

The idea of moving data processing closer to the data has been explored before in different domains. For instance, Active Pages [38] proposed to move data manipulation operations to the memory subsystem to address the *processor-memory* gap resulting from the rapid increase in processor performance at the time. Similar approaches have been explored for storage [29, 42, 51]. All these approaches take advantage of the general purpose compute capabilities of the storage device to offload different types of data processing operations with the goal of reducing data movement. StRoM addresses the increasing gap between network and CPU due to stagnating CPU frequencies. In particular, it reduces the number of CPU cycles that need to be allocated towards network processing and data movement with the overall goal of reducing network traffic by pushing operations closer to the data. StRoM can be used to make disaggregated memory and network attached storage more efficient, thereby providing a sound basis for implementing the decades old idea of smart/active memory.

3 Design Methodology in StRoM

StRoM implements RPC on top of RDMA by extending the semantics of one-sided operations while offloading the execution of the RPC logic to the NIC instead of using the remote CPU [24, 26, 48]. In designing StRoM, we kept the following five goals in mind.

3.1 Minimize Modifications to RDMA Verbs

With StRoM, we accelerate a variety of applications in an FPGA-based NIC. A naive approach to do so would be to introduce a new verb for each new function to offload. This does not scale and would increase the complexity of the Infiniband (IB) protocol manifold. Further, each new verb would have to address the trade-off between wide-applicability and specialization to maximize the potential performance gains. Finally, in contrast to the current, basic but robust, IB verbs, the programmer would have to program against a more complex and potentially evolving interface. Instead, in StRoM we minimize the changes to the IB protocol to avoid an increase in complexity: StRoM introduces only two new IB verbs and five new Reliable Transport Header (RTH) op-codes (Table 1) to support a broad range of applications.

3.2 Integration on the Data Path

Current Approaches. Recently NIC manufacturers have started to integrate co-processors on their NICs [34, 39], such as 64-bit ARM processors (e.g., Mellanox BlueField and Broadcom PS225) or FPGAs (e.g. Mellanox Innova-2 Flex). ARM cores provide a familiar development environment to many users but, in current solutions, the latency is significantly affected when the ARM cores are involved. A simple ping-pong microbenchmark on a Broadcom PS225 NIC has shown a latency increase of 3 μs when the ARM cores are on the data path [34]. The reason for the increase is that the ARM cores are not actually on the data path, instead they are implemented as an additional endpoint over an on-board switch. Involving the ARM cores for processing adds an additional hop to the transmission. Programmable logic in the form of FPGAs is already widely used for high bandwidth stream processing, such as packet inspection [41] and, as a result, also deployed on some commercial SmartNICs as a co-processor. In these solutions, the interface between the FPGA and the network stack and the DMA engine is barely documented and fairly restricted, limiting the offloading of complex functionality.

Our Approach. To implement StRoM, we have developed an FPGA-based RDMA NIC including a RoCE v2 stack, supporting one-sided read and write verbs, and a DMA engine accessing the host memory. The platform allows us to integrate StRoM on the data path between the RoCE v2 stack and the DMA engine such that StRoM acts as a bump-in-the-wire: adding negligible latency while not impacting throughput.

3.3 High Programmability

Current Approach. Existing in-network processing solutions, e.g., P4 [9], ARM-based SmartNIC [35] and sPIN [17], operate at the packet level, so the user needs to reason about

the exact role of each packet in the context of the data stream to be processed rather than focusing on the application logic.
Our Approach. StRoM's philosophy is to expose a higher-level abstraction (i.e., focusing on the data inside the packet) to programmers (Subsection 5.2) such that the programmers do not need to work at the stateless level, i.e., raw RoCE packets, making the system much easier to program. Furthermore, in StRoM we opt to use high-level synthesis [57], raising the programming abstraction from cycle-sensitive hardware description languages (HDL) to C/C++. To guarantee that the offloaded functionality is portable and interchangeable at run-time (e.g., through partial-reprogramming), we define and restrict the hardware interface of StRoM that exists between the DMA and the NIC (Subsection 5.2).[2] This approach is similar to OpenCL kernels for FPGAs which also adhere to a pre-defined hardware interface but give the programmer the freedom to implement any functionality within (hence our use of the term kernel for the programmable logic of StRoM).

3.4 Easy Extension to a Broad Range of Applications

Currently, StRoM kernels support offloading of RPCs over RDMA functionality and stream processing. For stream processing, StRoM is able to directly operate on a data stream read/written from/to remote memory. StRoM allows us to offload a compute-heavy application to the FPGA-based NIC under two conditions. First, the FPGA needs to have enough resources to accommodate the compute-heavy application, considering the resources pre-allocated to the NIC functionality which includes the RoCE v2 stack and the DMA engine. This condition always holds as the NIC functionality only occupies a minor amount of the total available resources (Subsection 6.1). Second, the application's hardware implementation needs to consume the data stream at line rate.[3] Otherwise, StRoM might affect the functionality of the original RDMA operation. For RPCs over RDMA, kernels can implement arbitrary functionality. In this paper we focus on pointer chasing over remote data structures as commonly used in RDMA based key-value stores and consistency checks when reading remote data objects as examples of what can be done.

3.5 Easy Design Space Exploration

StRoM is intended as a fully operational NIC but also as a platform for research.
Scale-up. Our prototype implementation of StRoM runs at 10 Gbit/s on a low-end Xilinx Virtex 7 FPGA. The RoCE stack architecture, introduced in Section 4.1, has a parametrizable data path. For 10 Gbit/s the data path is 8 B wide and the stack runs at 156.25 MHz. In Section 7, we show that on a latest generation FPGA supporting higher clock frequencies, with more logic resources, and higher on-chip memory capacity, we can scale the data path width up to 64 B and increase the clock frequency to 322 MHz, thereby allowing StRoM to operate at 100 Gbit/s.

[2] StRoM can, nevertheless, also be programmed in HDL.

[3] In the context of FPGA programming, we only need to achieve initiation interval (II) = 1, where II is the number of cycles before the hardware module can consume new input data.

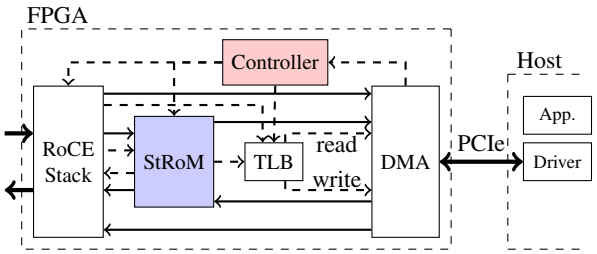


Figure 1. Hardware modules deployed on the FPGA, connected over PCIe to the host machine and over 10 G Ethernet to the network, the figure indicates commands issued (dashed lines) and the data flow.

Migration to FPGA-based Commercial SmartNICs. The kernels on StRoM can be easily ported to a commercial SmartNIC with an FPGA co-processor or to a design such as the one of Microsoft's Catapult [14] with an ASIC NIC and a packet processor on the FPGA.
Local StRoM Invocation. StRoM kernels are not limited to invocation of RPCs on a remote NIC. They can also be used to process data before being sent (send kernel), to process data or be called on arrival (receive kernel), and combinations thereof (send-receive kernels) to implement complex protocols and interactions between the two involved NICs.

4 StRoM RoCE NIC

As a first step in describing StRoM, we present the FPGA-based RoCE NIC dealing with network communication. The StRoM NIC consists of two main components: the RoCE network stack and the DMA engine accessing the host memory over PCIe (Figure 1). The RoCE network stack can process and generate packets at a rate of 10 Gbit/s and is directly connected to the 10 G Ethernet interface on the FPGA board. Through the DMA engine, it can directly access the host memory. Address translation is provided by the Translation Lookaside Buffer (TLB) on the NIC. The *Controller* module allows the host to control and monitor the NIC.

4.1 RoCE Network Stack

The current version of StRoM implements a subset of the RoCE v2 protocol supporting the two one-sided IB verbs: `RDMA WRITE` and `RDMA READ`. We have not implemented the two-sided operations because our intention is to replace two-sided operations with one-sided StRoM kernels.

The architecture of the stack (Figure 2) consists of two data paths, one for incoming and one for outgoing packets and data structures that keep track of the state for each queue pair. Our design implements a clear separation of the two data paths and the state-keeping data structures, allowing independent

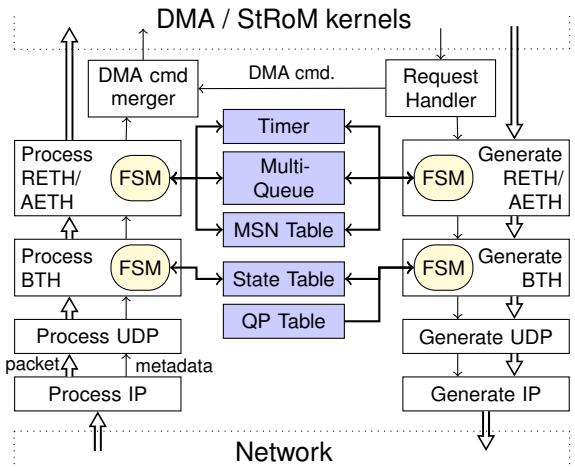


Figure 2. Architecture of the RoCE v2 network stack, with clear separation between data paths and state-keeping data structures. Protocol processing is fully pipelined.

processing on the two paths which is essential to achieve line-rate bidirectional bandwidth. The queue pair state is stored in the *State Table*, the *MSN Table* and the *Retransmission Timer*. The *State Table* stores all packet sequence numbers (PSNs) to define the valid, invalid, and duplicate PSN regions. This information is stored for two cases when the NIC acts as a responder and when it acts as a requester. The *MSN Table* stores the message sequence number (MSN) and the current DMA address. This is necessary since for write operations with payload spanning multiple packets the address is only part of the first packet. The *Retransmission Timer* implements one timer per queue pair to detect packet loss. The timers are implemented as an array of time intervals stored in on-chip memory. The *Retransmission Timer* module is continuously iterating over this array and decreasing the time intervals of all active timers. If any timer reaches zero an event is triggered and forwarded to the transmitting data path to retransmit the lost packet(s).

Protocol processing in the two data paths is pipelined to achieve line-rate bandwidth. The following protocol headers are processed: IP, UDP, BTH (Base Transport Header), RETH (RDMA Extended Transport Header) and AETH (ACK Extended Transport Header), respectively. At each stage on the receiving data path, the current protocol header is parsed and all relevant metadata extracted. Then the packet header is removed and the packet is re-aligned to the width of the data path (8 B). After checking the IP checksum and UDP port, the *Process IP* and *Process UDP* module extract metadata, e.g., IP addresses, UDP ports, and packet length, and forward it on a separate bus to the *Process BTH* module. This module extracts the RDMA op-code, the packet sequence number (PSN) and the queue pair number (QPN) from the header. Next a finite state machine (FSM) checks the extracted metadata

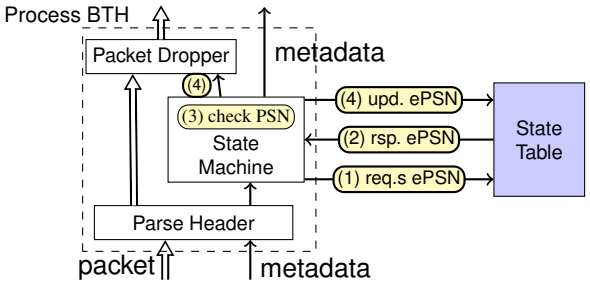


Figure 3. Interaction between the *Process BTH* module and the *State Table*.

against the state stored in the *State Table* and possibly updates it. Figure 3 illustrates how the extracted PSN is checked against the expected PSN: (1) requesting the *State Table* entry using the QPN as a key, (2) the *State Table* returns the corresponding entry, (3) checking if the expected PSN in the entry matches the extracted PSN, and (4) instructing the *Packet Dropper* module to either drop the packet or forward it to the next stage. In case of a match the state machine concurrently writes the updated, expected PSN back to the *State Table*. These steps take around 5 cycles per packet, given that the smallest possible Ethernet frame is 64 B corresponding to 8 cycles, we can guarantee that the hardware pipeline can sustain line-rate processing at 10 Gbit/s. At 5 cycles, the update step is a potential bottleneck for small packets at higher bandwidths. However, in Section 7 we show that the message rate at higher bandwidths is limited by the host issuing commands and not by the packet processing. The final stage processes the RETH and AETH headers and implements an FSM that takes decisions based on the RDMA op-code and if required issues DMA commands and requests to generate response packets.

On the transmitting data path, the *Request Handler* module receives requests issued by the host through the *Controller*. Depending on the request it will fetch payload from the DMA engine. The request is then forwarded to the *Generate RETH/AETH* module which generates the corresponding headers and appends payload if applicable. Similar to the receiving data path, before forwarding the packet to the next stage, it is re-aligned such that the next packet header can be prepended. The *Generate RETH/AETH* and *Generate BTH* module both deploy an FSM to retrieve and update metadata stored in the data structures. They also forward metadata to the *Generate UDP* and *Generate IP* modules.

To support multiple outstanding RDMA read operations per queue pair we implement a *Multi-Queue* data structure which logically implements one linked-list per queue pair. Each linked list has a variable length defined at runtime, but the combined length of all linked lists is fixed. The actual hardware implementation consists of two fixed-size arrays stored in on-chip memory. The first one stores the list metadata pointing to the head and tail of the list. The second array contains all list elements where each element consists of a

local host memory pointer (the target of the read operation), a pointer to the next element in the list, and a flag indicating if this is the tail.

The architecture and implementation of the stack allows scalability in two dimensions. First, thanks to the clear separation of the state-keeping data structures from the packet processing, the latter can be scaled independently to support the desired number of queue pairs. The number of supported queue pairs is a compile-time parameter and has a linear impact on the required on-chip memory usage, see Section 6.1. Second, the width of the data path is parametrizable in power of two steps. The width can be varied from 8 B to 64 B resulting in a bandwidth of 10-80 Gbit/s at 156.25 MHz. Similarly, the *Multi-Queue* data structure can be parametrized by the number of queue pairs as well as the total number of outstanding RDMA read operations

While we deploy StRoM on a traditional Ethernet network, the used Ethernet IP core[4] supports Priority-based Flow Control (PFC) necessary for Converged Ethernet. For a seamless integration into the network infrastructure, we use an open source module [44–46] to handle the Address Resolution Protocol.

4.2 DMA Engine and Driver

For Direct Memory Access (DMA) over PCIe, we deploy the Xilinx *DMA/Bridge Subsystem for PCI Express*[5] IP core, running at 8 GT/s and clocked at 250 MHz. The accompanied driver[6] assumes synchronous interaction between the software and hardware where the software thread has to actively push and pull data through a stream-based interface to/from the FPGA. To support RDMA capabilities, the card must be able to independently access the host memory without explicit synchronization with the CPU and be able to address large DMA buffers. To achieve this functionality, we have implemented our own linux kernel driver.

Our driver exposes the PCIe bar that maps to control and status registers on the FPGA as a device */dev/roce*. By mapping this device into the user space of the application through *mmap*, the software application can directly interact with the FPGA at low latency without involving the operating system. On the hardware a *Controller* module converts the register accesses into commands that are issued to the RoCE stack, the StRoM kernels, or to populate the TLB (Figure 1). Apart from issuing commands, the host can also retrieve status and performance metrics.

For data transfers, the DMA IP core is configured with two 32 B streaming interfaces. One stream for writing data to host memory and the other to retrieve data from host memory. Further, we enable the *Descriptor Bypass* on the DMA IP core. This interface allows the FPGA to issue requests directly to the DMA engine without synchronization with the CPU. Each descriptor describes a data transfer from the host memory to the card or vice versa. To enable direct access to the host memory from the FPGA, memory has to be pinned in advance. To do so the application passes a memory region to the driver which pins every page and also returns its physical addresses. The current version of the driver does not support interrupts, as such applications use polling for low latency communication.

[4]http://www.xilinx.com/products/intellectual-property/ef-di-25gemac.html

[5]http://www.xilinx.com/support/documentation/ip_documentation/xdma/v3_1/pg195-pcie-dma.pdf

[6]http://www.xilinx.com/support/answers/65444.html

Table 1. Reliable Extended Transport Header op-codes to support StRoM kernels.

verb	op-code	Description
`RPC`	11000	RDMA RPC Params
`RPC WRITE`	11001	RDMA RPC WRITE First
	11010	RDMA RPC WRITE Middle
	11011	RDMA RPC WRITE Last
	11100	RDMA RPC WRITE Only
	11101 - 11111	reserved

4.3 Memory Management

RDMA operations specify virtual memory addresses, but to access host memory over PCIe physical addresses are used. To translate from virtual to physical addresses a Translation Lookaside Buffer (TLB) is deployed on the NIC. Each entry in the TLB stores one 48 bit physical address corresponding to a 2 MB huge page which is contiguous in the physical address space. We use 2 MB huge pages to reduce the number of entries in the TLB which can hold up to 16,384 entries. This allows the FPGA to directly address up to 32 GB of host memory which seems to be enough for most uses cases, but can be further increased by allocating more on-chip memory to the TLB if necessary. The TLB module is populated once and does not support page misses which requires that the physical pages are pinned by the kernel driver (such a requirement also holds for RDMA buffers). Even though all the huge pages combined build a single contiguous virtual address space, physically they might not be contiguous. This means the TLB has to check if a read or write operation is crossing a 2 MB page boundary. If this is the case the TLB resolves those accesses by splitting the command into multiple commands, none of them crossing page boundaries.

5 StRoM – Programmable Kernels

5.1 Protocol

One main objective when integrating StRoM on the RDMA NIC is to minimize the changes to the RoCE stack as well as the impact on existing RDMA verbs. To send an RPC and its parameters to the remote NIC, we introduce the `RDMA RPC` verb. This verb maps to a single Base Transport Header (BTH) op-code `RDMA RPC Params`, see Table 1. Packets with this op-code are treated similarly to an `RDMA WRITE`

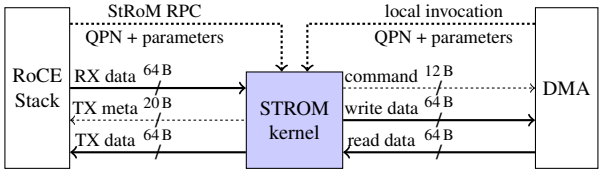


Figure 4. StRoM kernel and its interfaces.

Listing 1. Function interface of STROM kernels.

```
void strom_kernel(stream<ap_uint<24> >&      qpnIn,
                  stream<ap_uint<256> >&     paramIn,
                  stream<net_axis<512> >&    roceDataIn,
                  stream<memCmd>&            dmaCmdOut,
                  stream<net_axis<512> >&    dmaDataOut,
                  stream<net_axis<512> >&    dmaDataIn,
                  stream<roceMeta>&          roceMetaOut,
                  stream<net_axis<512> >&    roceDataOut);
```

`Only` which means the payload size is at most one Maximum Transmission Unit (MTU). The payload contains the parameters to the StRoM kernel. Instead of replacing the Reliable Extended Transport Header (RETH) on top of the BTH, we reuse the address and length field. In case of an RPC, the address field encodes an RPC op-code that is used to match the request against the deployed StRoM kernels on the remote NIC. This mechanism resembles the matching used in Portals [4] and enables multi-kernel deployments on the remote NIC. If the RPC op-code does not match any of the deployed kernels, either a fallback implementation on the remote CPU is triggered (if configured a priori by the remote CPU) or an error code is written back to the requesting node.

RPCs implemented over StRoM not only allow to access the remote memory and execute complex operations on it, it is also possible to attach payload to an RPC call. To support this, the RoCE stack is extended with 4 BTH op-codes (Table 1) which are exposed through the `RDMA RPC WRITE` verb. The packets using these op-codes are processed in the same way as packets with the corresponding `RDMA WRITE` op-codes, the only difference is that on the remote NIC the payload is not written to the host memory but forwarded to the StRoM kernel using the address field in the RETH as an RPC op-code.

Both new IB verbs are semantically similar to `RDMA WRITE` which is used to transmit data from the remote NIC back to the requesting node. Using write semantics for the new IB verbs, instead of read semantics, has the advantage that the size of the response does not have to be known in advance. In contrast, an `RDMA READ` operation requires the length of the response in advance to pre-calculate the number of expected packets and their sequence numbers. In the context of StRoM, this constraint would inhibit many operations, e.g. (data reduction), where the response size is determined at run-time and not known a priori.

In summary, we extend the RoCE stack with 5 new op-codes in the BTH and two new Infiniband verbs resulting in a code change of less than 50 lines in the RoCE stack implementation. Additionally, the existing one-sided verbs are only affected by a negligible latency increase of a few clock cycles due to the arbitration of data streams.

5.2 StRoM Kernel

A critical factor in the integration of the programming kernels with the NIC is the placement of the kernels. We place the kernels on the data path between the RoCE stack and the DMA engine which benefits especially stream-oriented operations that can operate at a high bandwidth while incurring minimal latency. Further, the existing direct data path between the RoCE stack and the DMA engine remains and is only extended with arbitration logic that adds negligible latency.

To simplify the deployment of StRoM kernels and make them run-time interchangeable, we strictly define the hardware interface (Figure 4). The data paths between the kernel and the RoCE stack as well as the DMA engine are 64 B wide. Further, the kernel is connected to a 32 B bus to receive the RPC parameters, a 20 B bus to issue RDMA write operations, and a 12 B bus to issue local DMA commands.

The well-defined interface also translates to the software implementation. Listing 1 shows the kernel interface in C++, the stream type in Vivado HLS [57] maps to FIFOs in hardware. The interface consists of four metadata and four data streams. The first two metadata streams (`qpnIn` and `paramIn`) provide the queue pair number (QPN) and the parameters to the kernel, optional payload is received through the `roceDataIn` stream. To access the host memory the kernel can issue DMA commands consisting of a virtual address and length over the `dmaCmdOut` interface, data from and to the DMA engine is sent over the `dmaDataOut` and `dmaDataIn` streams. To transmit data over the network the kernel can issue metadata and data over the `roceMetaOut` and `roceDataOut` streams. The metadata consists of the QPN, the target virtual address, and the length. While not the focus of this work, StRoM kernels can also be invoked by the local host by posting an RPC to the local network card and providing the QPN together with the required parameters.

Example Kernel

Listing 2. Main function of the GET kernel.

```
void get(...) {
   #pragma HLS DATAFLOW
   static stream<readOp>         readSrcFifo;
   static stream<memCmd>         htCmdFifo;
   static stream<memCmd>         valueCmdFifo;
   static stream<internalMeta>   metaFifo;
   static stream<ap_uint<512> > htEntryFifo;

   fetch_ht_entry(qpnIn, paramIn, htCmdFifo, metaFifo);
   parse_ht_entry(metaFifo, htEntryFifo,  valueCmdFifo,
         roceMetaOut);
   merge_read_cmds(htCmdFifo, valueCmdFifo,  readSrcFifo,
         dmaCmdOut);
   split_read_data(readSrcFifo, dmaDataIn, htEntryFifo,
         roceDataOut);
}
```

Existing work showed one-sided [13, 36] and two-sided [25] implementations of the GET operation in a key-value store. We use the same operation to illustrate how a StRoM kernel can be implemented with the given hardware interface and high-level synthesis. The GET operation consists of two read operations, one to fetch the hash table entry and another one to retrieve the data value. For simplicity, in this example we assume that there is always exactly one matching key in the hash table entry, therefore our example omits handling of misses and corresponding mechanisms such as linear probing or chaining. A more sophisticated kernel supporting additional data structures is shown and evaluated in Section 6.2. Listing 2 shows the main function *get* of the *GET kernel*. The `stream` data structure, which maps to FIFOs in hardware, is used to stream data in and out of the kernel as well as between functions within the kernel. The kernel consists of 4 functions: 1) `fetch_ht_entry` reads the hash table entry from the host memory, 2) `parse_ht_entry` parses the retrieved hash table entry and requests the value data from the host memory, 3) `merge_read_cmds` merges the DMA read commands from the previous two functions, and 4) `split_read_data` distributes the data read from host memory to the requesting function. The `HLS DATAFLOW` pragma means that each of these 4 functions will map to a hardware module that operates independently and concurrently. Combined with the FIFOs connecting the 4 modules the whole kernel is mapped to a pipelined data-flow on the FPGA that can operate at a high data rate.

Listing 3. Function in GET kernel to fetch the hash table entry from the host memory.

```
void fetch_ht_entry(stream<ap_uint<24> >& qpnIn,
                    stream<getParams>&    paramIn,
                    stream<memCmd>&       htCmdFifoOut,
                    stream<internalMeta>& metaFifoOut) {
  #pragma HLS PIPELINE II=1
  if (!qpnIn.empty() && !paramIn.empty()) {
    ap_uint<24> qpn = qpnIn.read();
    getParams params = paramIn.read();
    htCmdFifoOut.write(memCmd(params.getAddress(), 64));
    metaFifoOut.write(internalMeta(qpn, params.getKey(), params.
        getTargetAddr()));
  } }
```

The `fetch_ht_entry` function in Listing 3 implements the fetching of the hash table entry. It checks the two input streams `qpnIn` and `parameterIn` for new metadata. Once new metadata is available, it is consumed and used to issue a DMA request to fetch the hash table entry. Simultaneously, part of the metadata is stored into an internal FIFO that is later consumed by the `parse_ht_entry` function. The `PIPELINE` pragma instructs the compiler to pipeline this function and fulfill an initiation interval (II) of 1 meaning that the resulting hardware module can consume and process data from its input streams every clock cycle (II=1).

The `parse_ht_entry` function (Listing 4) reads the metadata and the hash table entry containing 3 buckets. It then concurrently compares the lookup key in the metadata against the key in each bucket and determines the index of the matching bucket. The `UNROLL` pragma specifies that the loop is unrolled in hardware, meaning that all iterations are performed concurrently. As a last step, the matching index is used to extract the value pointer and length to generate a DMA read command and the RoCE metadata for transmission of the data value. This function is also pipelined with an initiation interval of 1 using the `PIPELINE` pragma.

Listing 4. Function in GET kernel to match the key against the buckets and fetch the value from host memory.

```
void parse_ht_entry(stream<internalMeta>&  metaFifoIn,
                    stream<ap_uint<512> >& htEntryFifoIn,
                    stream<memCmd>&        valueCmdFifoOut,
                    stream<roceMeta>&      roceMetaOut) {
  #pragma HLS PIPELINE II=1
  if (!metaFifoIn.empty() && !htEntryFifoIn.empty()) {
    internalMeta meta = metaFifoIn.read();
    htEntry entry = (htEntry) htEntryFifoIn.read();
    bool match[3];
    for (int i = 0; i < 3; ++i) {
      #pragma HLS UNROLL
      match[i] = (entry.getKey(i) == meta.lookupKey);
    }
    //Check which key matches
    int matchIdx = (match[1]) ? 1 : ((match[2]) ? 2 : 0);
    //Write DMA command
    valueCmdFifoOut.write(memCmd(entry.getValuePtr(matchIdx),
        entry.getValueLen(matchIdx)));
    //Write RoCE Metadata
    roceMetaOut.write(roceMeta(meta.qpn,meta.targetAddress, entry.
        getValueLen(matchIdx)));
   } }
```

5.3 Software Integration/API

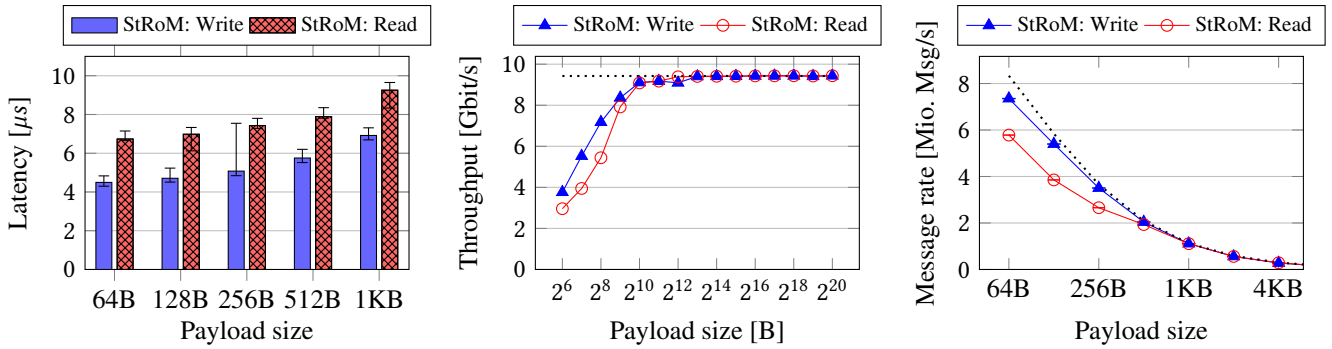
On the software side we expose the RDMA RPC functionality of StRoM through two simple function calls (Listing 5). The user can issue an RDMA RPC by calling the `postRpc` function that takes as arguments: an RPC op-code, the queue pair, the pointer to the parameters and their size. The RPC op-code specifies the StRoM kernel type and is required for matching the RPC request to the kernel on the remote NIC. To attach payload to the RPC, the user can call the `postRpcWrite` function which takes the RPC op-code, the virtual address of the data, and its size as an argument.

Listing 5. Application interface to issue an RDMA RPC.

```
void postRpc(rpcOpCode op, QueuePair* pair, const void* parameters
     , uint32_t size);
void postRpcWrite(rpcOpCode op, QueuePair* pair, const void*
     originAddr, uint32_t size);
```

6 Experimental Evaluation of StRoM

In the following, we present the hardware details of our StRoM NIC in Subsection 6.1, and evaluate the performance of StRoM for a broad range of applications, including data traversal (Subsection 6.2), key-value stores (Subsection 6.3), databases (Subsection 6.4), and stream processing (Subsection 7.2). These example applications show the benefit of offloading RPCs to the remote NIC as well as the efficiency of in-NIC processing while transferring data using RDMA.



(a) Median latency of RDMA read and write operations.

(b) Throughput of RDMA read and write operations.

(c) Message rate of RDMA read and write operations.

Figure 5. Performance metrics of the StRoM RoCE NIC. Error bars indicate the 1st and 99th percentile. The dotted lines indicate the ideal throughput and message rate, respectively, for 10 G RoCE v2 (MTU 1500).

6.1 StRoM RoCE NIC

We have implemented StRoM on an Alpha Data ADM-PCIE-7V3 FPGA board equipped with a Xilinx Virtex 7 XC7VX690T FPGA. The board is connected to the host machine over PCIe Gen 3 x8. We directly connected two StRoM NICs to each other to remove the potential noise introduced by a switch.

The RoCE implementation including the DMA engine, TLB, and 10 G Ethernet interface uses only 24% of the available logic resources on the device, allowing the deployment of multiple StRoM kernels. For 500 queue pairs (QPs) 9% of the on-chip memory is occupied. Most of it is allocated to the TLB and the state-keeping data structures in the RoCE stack. Since the data structures scale linearly with the number of supported QPs, the logic resource usage stays within 1% when going from 500 to 16,000 QPs, the on-chip memory usage on the other hand increases to 20%. The RoCE stack is clocked at 156.25 MHz and the DMA at 250 MHz.

Latency. The latency of the write operation is determined through a ping-pong benchmark involving two machines. The initiator writes data to the remote machine at a predefined address. The remote machine polls on this address until the data is received. Upon reception, it immediately writes the data back to the initiator machine which likewise polls on the corresponding memory address. This round trip time is measured on the initiator machine and the corresponding latency ($\frac{RTT}{2}$) is reported. Figure 5a shows the median latency of RDMA read and write operations on the StRoM NIC when varying the payload size from 64B to 1KB.

Throughput. To evaluate the throughput, we vary the payload size from 64 B to 1 MB (Figure 5b). For large payloads the NIC reaches the theoretical peak bandwidth of 9.4 Gbit/s. For small messages the throughput is bound by the message rate, as shown in Figure 5c. The hardware pipelines on the NIC to process and generate packets are able to operate at 10 G line-rate even for small packets, as explained in Section 4.1.

6.2 Traversing Remote Data Structures

Traversing data structures is a memory-intensive but low compute operation. In a single node system, data structure traversals are commonly limited by the memory latency. When moving to distributed systems that expose data structures over RDMA, the latency of a single memory lookup significantly increases due to the network round trip. As a result, applications using one-sided operations are optimized to minimize the number of round trips required per operation. For instance, a hash table lookup in Pilaf [36] requires on average 1.6 RDMA reads at 75% occupancy. In the FaRM key-value store [13], the same operation takes 1.04 RDMA reads at 90% occupancy. Thus, to complete a GET operation, an additional round trip is required to get the value, leading to at least two round trips for the whole operation. More complex data structures, such as B-trees or graphs, would require even more round trips per operation and are therefore commonly implemented with an RPC over two-sided RDMA [43, 56].

To reduce the number of network round-trips when accessing remote data structures, we implement a StRoM kernel on the remote FPGA-based NIC, called *traversal kernel* that allows the traversal of a remote data structure. The key idea of StRoM is to replace high-latency network round-trips with PCIe round trips of relatively low latency. The kernel starts from a root element and then extracts one or multiple keys in that element and compares them against a given key. In case of a match, the data value associated to that key is read out. Otherwise the *next* element in the data structure is fetched (or the traversal terminates if it is the leaf/tail element). The *traversal kernel* accepts the parameters listed in Table 2. This makes it quite flexible as it allows the traversal of linked lists, hash tables, trees, graphs, skip lists, and other data structures. The current implementation assumes that each data structure element cannot exceed 64 B, that the key has a fixed size of 8 B, and that the fields within the element are 4 B aligned. These are all aspects of the design that can be easily changed.

Table 2. Parameters of the StRoM *traversal kernel*.

Parameter	Description
remoteAddress	The address of the initial element in the remote data structure.
valueSize	The size of the final value to be read.
key	The lookup key.
keyMask	This masks specifies where the key(s) is/are located in the data structure element.
predicateOpCode	Operation applied to compare the key in the command and in the data structure, has to be one of the following: `EQUAL`, `LESS_THAN`, `GREATER_THAN`, `NOT_EQUAL`.
valuePtrPosition	The position of the value pointer within the data structure element which can be absolute or relative to the key that matched.
isRelativePosition	Indicates if the *valuePtrPosition* is relative to the key or absolute.
nextElementPtrPos.	The position of the pointer to the next element in the data structure. The next element is read in case none of the keys in the current element matched.
nextElementPtrValid	This boolean indicates if the data structure element contains a pointer to a next element.

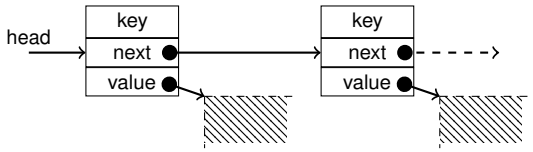


Figure 6. Linked list in remote memory

Example: Linked List

As a first example we show how the *traversal kernel* can be used to traverse a linked list such as that used in some key-value stores to store entries for keys hashing to the same position. Accordingly, we consider a short linked list (Figure 6) where the key in each element is unique. To lookup a given key, the list is traversed starting from the head until the lookup key matches the key in the list element. Once the matching element is found, the data value pointed to by the value pointer is read. Given the layout of the list element, we set the *keyMask* to 1, the *valuePtrPosition* to 4, and the *nextElementPtrPosition* to 2.

We evaluate the latency of retrieving a value in the linked list by randomly picking a key and then retrieving its corresponding value by traversing the remote linked list. We vary the length of the list and thereby the expected number of traversals to find the key (Figure 7).

Relying on `RDMA READ` as Pilaf or FaRM do, each traversal involves a network round trip resulting in a linear increase of the latency with the length of the list. In case of the *traversal kernel*, each traversal requires a read over PCIe which takes around 1.5μs. Since the network round trip is the dominating cost, reducing the lookup to a single round trip leads to a sublinear increase of the latency with increasing list length. We further varied the value size, however this has a marginal impact on the latency given that it is dominated by the traversal of the linked list for these value sizes. As an additional baseline, we use the *rpcgen* compiler [11] to generate RPCs that can be invoked over TCP on the remote machine. In the case of an RPC the remote CPU is traversing the linked list.

Figure 7 illustrates that the latency of the TCP-based RPC implementation does not vary when increasing the length of

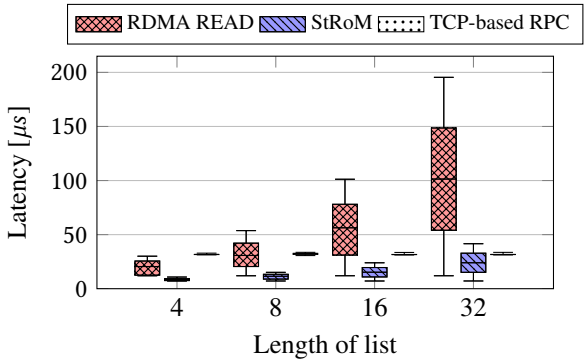


Figure 7. Traversing remote linked list using three approaches: conventional `RDMA READ`, *STROM kernel* and TCP-based RPC. Whiskers indicate the 1st and 99th percentile. Value size is 64 B.

list, as the remote function invocation dominates the overall cost while the actual list traversal on the CPU is faster than that over the PCIe link.[7] Given that the STROM kernel bypasses the CPU cores, it offers a better solution than conventional one-sided RDMA or dedicating one core to the task (which actually does not scale for larger value sizes as shown below).

Example: Hash Table

In our second example we look at a hash table. Our implementation mimics the implementation and data layout used in Pilaf [36]. The hash table consists of two memory regions, the first one contains fix-sized hash table entries which point to the corresponding data value and the second one contains all the values. A *GET* operation requires in the best case two `RDMA READ` operations, one to read the hash table entry and another one to fetch the data value. By using the *traversal kernel* the *GET* operation can be executed with a single network round-trip. The *traversal kernel* deployed on the remote NIC

[7] A modern CPU's memory latency is roughly 80 ns, while the PCIe's memory access latency is roughly 1.5 μs. Upcoming interconnects for the FPGA (Intel's CXL, IBM's CAPI, or CCIX all promise better latency).

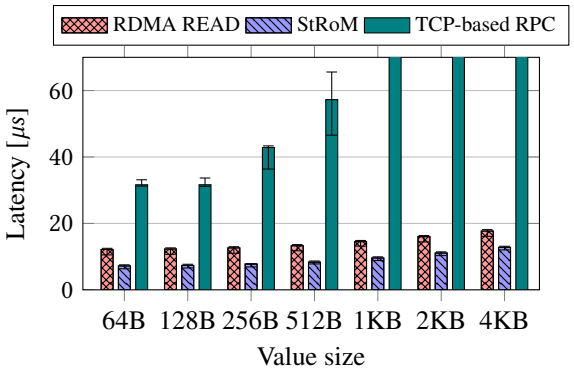


Figure 8. Median latency of remote hash table lookup using three approaches: conventional `RDMA READ`, *STROM kernel* and TCP-based RPC, while varying the value size. Error bars indicate the 1st and 99th percentile.

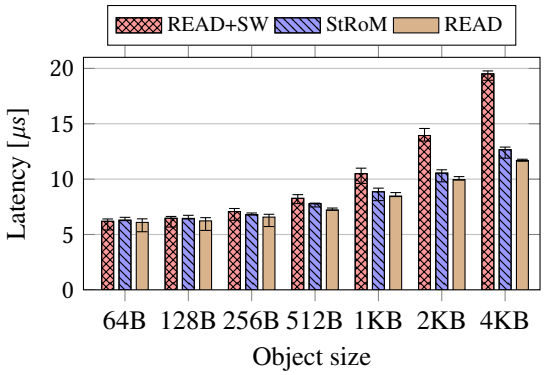


Figure 9. Median latency of reading a remote value without a consistency check ("READ"), with a local CRC64 check on a CPU core ("READ+SW"), and with the CRC64 check offloaded to the *consistency kernel* on the remote NIC ("StRoM"). Error bars indicate the 1st and 99th percentile.

will fetch the hash table entry specified by the *remoteAddress* parameter. It will extract the keys and match them against the given key. If successful, the data value of size *valueSize* will be read and transmitted to the client. Otherwise, the remote NIC could either return an error code or fetch the next hash table entry in case the implementation uses chaining for collision resolution.

Figure 8 illustrates the latency of retrieving a value by using two `RDMA READ` operations, StRoM, or TCP-based RPC.We assume that the hash table entry always matches the given key resulting in the best case of two RDMA read operations to retrieve the value. Using StRoM the latency can be reduced by around 5μs per lookup due to saving one network round trip. The TCP-based RPC also requires only one round trip, but suffers from long message passing latency for value sizes larger than 256 B.

6.3 Key-value Store (Data Consistency Check)

The x86 memory system only provides atomic operations at the granularity of cache lines. However, many data objects exposed through remote memory are larger than a single cache line. Additionally, atomic operations over RDMA incur a high latency and are avoided whenever possible. In fact, optimistic execution with a way to recover in case of inconsistency is often a faster option. When accessing a data object through a one-sided read operation, the retrieved object can be inconsistent if the object was modified at the same time by the remote host. In FaRM [13], data objects have a version number stored in every cache line of the object. A client issues a one-sided read operation to retrieve the object, it then checks if the version number in the cache lines is consistent. If such is not the case, the object must be read again resulting in an additional round trip. Similarly, Pilaf [36] calculates a checksum for each object and stores it in the object. When a client reads an object over RDMA, it will check on the local CPU if the checksum is correct and must read the object again otherwise.

We can use StRoM to implement the data consistency check as a kernel on the remote NIC. This kernel, *consistency kernel*, reads a data object from the remote host memory, calculates the CRC64 checksum over the object, and verifies its correctness on the remote NIC. In case of inconsistency, the kernel re-reads the data object, otherwise it issues an RDMA write to place the object in the requester's memory.

Effect of Object Size. Figure 9 illustrates the latency of offloading the CRC64 check to the remote NIC ("StRoM") in comparison to an RDMA read operation without verifying the checksum ("READ") and with verifying the checksum in software by the requester ("READ+SW"). For small object sizes, the overhead of checking the consistency in either software or hardware is marginal. With increasing object size, the CRC64 calculation in software introduces up to 40% overhead.[8] For the same object size, StRoM only introduces an overhead of 1μs (< 8%) due to the efficient hardware implementation, demonstrating the great potential of StRoM; that is, StRoM does not introduce a notable latency overhead, while bypassing CPU cores. Given the negligible overhead, this particular use case of StRoM can also be applied to systems implementing key-value stores on network attached FPGAs [18, 19].

Effect of Failure Rate. Figure 10 illustrates the effect of inconsistent reads on the overall latency to retrieve an object. The failure rate is the probability that the consistency check fails when an object is read; note that in this evaluation it does not affect consecutive retries, which always succeed. For a failure rate of 1% or less the average latency is barely affected. With a 10% failure rate the required retries in case of `RDMA`

[8] To our knowledge, it is impossible to use SIMD instructions to accelerate CRC64, as CRC64 is inherently sequential. To be precise, we are not able to explore parallelism among bytes in the same data object. Another potential way to accelerate CRC is to use custom CRC instructions provided by CPUs. However, no CRC64 instruction is currently available, only a CRC32 instruction (e.g., *_mm_crc32_u8*) that cannot be used in our CRC64 implementation.

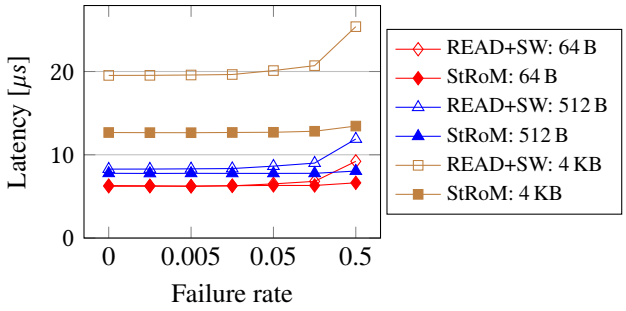


Figure 10. Average latency of reading a remote data object with varying failure rate and data size, in case of a failure the following re-read will always succeed.

`READ` incur a measurable overhead while the overhead from StRoM is minimal up to a failure rate of 50%. Up to this point the re-reads over PCIe on the remote NIC have a minor impact on the overall latency, demonstrating the efficiency of StRoM to handle checksum failures.

6.4 Database Application (Data Shuffling)

In database systems data partitioning is a common approach to accelerate high level operations such as joins [3, 8], aggregations, and sorting [40]. Data partitioning splits the data into cache-sized pieces to improve cache-locality for further operations. In distributed database systems [5, 6], data is also partitioned across nodes to increase parallelism further. Liu et al. [33] evaluate 6 different data shuffling operator designs. In their one-sided approach, the receiver issues read operations to retrieve data from the sender and then shuffles data locally. A different approach for data shuffling was taken by Barthels et al. [6] using one-sided write operations. In their implementation, the sender first shuffles the data locally and then writes each data partition to its corresponding remote memory location. In this approach the receiver is not involved in the data transfer. Recent work has shown how to speed up data partitioning for join operators on a multi-core server by offloading the partitioning to an FPGA [28].

We implement a *shuffling kernel* that supports data shuffling on the remote NIC. When data is transmitted, the kernel on the remote NIC partitions the incoming data on-the-fly and writes the partitioned data values to the corresponding location in its host memory. The kernel treats the payload as 8 B values and partitions them using a radix hash function that simply takes the N least significant bits of the value as its hash value. The kernel creates on-chip buffers for up to 1024 partitions, each of which accommodates up to 16 values (128 B). Such buffering is required to keep up with line-rate processing throughput over PCIe. The kernel is parametrized through an `RDMA RPC` message containing a histogram indicating the size and memory location of each partition.[9]

[9]The *shuffling kernel* can also be invoked on the local network card such that data is partitioned among different queue pairs and correspondingly different remote machines. However, data shuffling before transmission requires more buffering, up to MTU size, to achieve high bandwidth over the network. This limits the number of partitions, further increases the latency introduced by buffering, and requires more on-chip memory per partition.

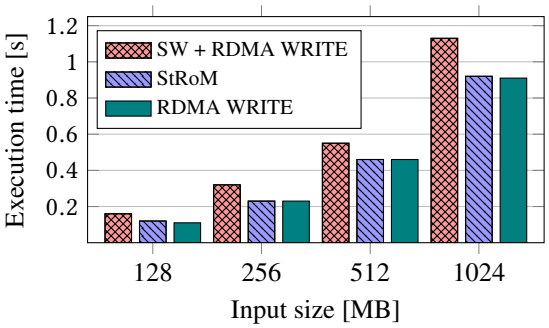


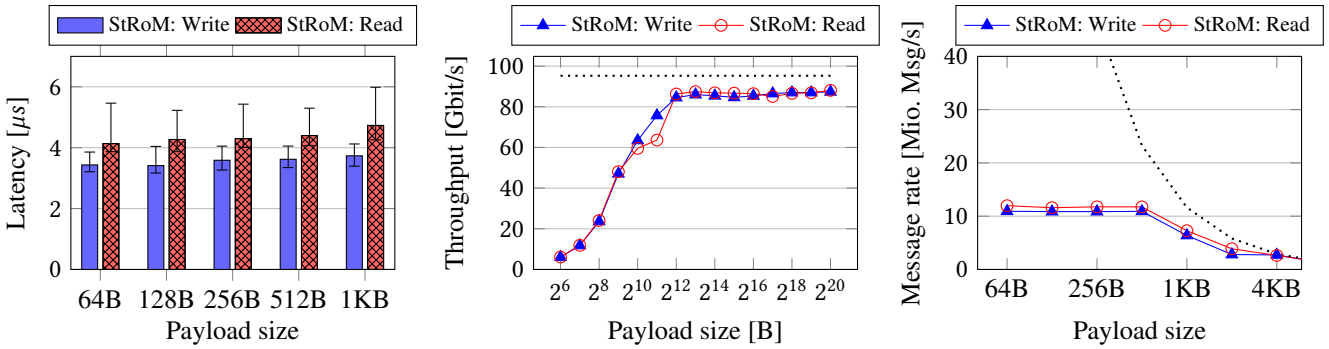
Figure 11. Average execution time for partitioning and transmitting data consisting of 8 B tuples.

Figure 11 illustrates the efficiency of StRoM for data shuffling. The baseline ("SW + RDMA WRITE") is the implementation by Barthels et al. [6] that first partitions the data locally and then writes each partition separately to the remote memory. Doing the partitioning on the CPU requires a pass over the data and each tuple has to be copied to its partition buffer. Once the size of a partition buffer reaches a threshold, i.e., 16 values, the partition buffer is written to the remote memory. Since the radix hash function is inexpensive, the overhead of partitioning stems from the additional data pass and copy. StRoM partitions the data on-the-fly upon reception, thus avoiding data copies. Note that the use of an inexpensive hash function benefits the CPU, since more robust hash functions would require more CPU cycles potentially reducing the throughput. In contrast, the FPGA can sustain the same performance even when more robust hash functions are deployed as shown by Kara et al. [27]. In conclusion, first, StRoM significantly reduces the overall execution time due to the elimination of costly data shuffling on the CPU; second, StRoM achieves close to the same performance as an "RDMA WRITE" just transmitting the data without partitioning, as data partitioning acts as a bump in the wire.

7 Scalability

By design, StRoM's network stack architecture allows scaling the data bus width, which directly affects network bandwidth. Additionally, on modern FPGAs the same circuit can often be clocked at a higher frequency. Based on this, we increase the data bus width from 8 B, used for 10 G, to 64 B and increase the clock frequency from 156.25 MHz to 322 MHz. All the HLS modules in the network stack undergo the same clock transformation. The modules are not redesigned, but additional register stages are inserted by the compiler if required to meet the target frequency. This results in the capability to process network packets at a line-rate of 100 Gbit/s.

The 100 G implementation of StRoM uses a Xilinx VCU118 FPGA board equipped with a Xilinx UltraScale+



(a) Median latency of RDMA read and write operations.

(b) Throughput of RDMA write and read operations.

(c) Message rate of RDMA read and write operations.

Figure 12. Performance metrics of the StRoM RoCE NIC at 100 G. Error bars indicate the 1st and 99th percentile. The dotted lines indicate the ideal throughput and message rate, respectively, for an MTU size of 1500 B.

FPGA, a 100 G CMAC interface, and host-connectivity over PCIe Gen 3 x16. While this increases the overall bandwidth of the NIC, it also shifts the ratio between the PCIe bandwidth and network bandwidth from around 6:1 on the Alpha Data card used for the 10 G version close to 1:1 on the VCU118. This means there is no spare PCIe bandwidth and, therefore, the PCIe link has to be utilized efficiently to maintain line-rate. This change has to be considered when designing a StRoM kernel and reduces the number of potential use cases. As an example, the *shuffling kernel* previously introduced requires random access to the host memory. This reduces the effective PCIe bandwidth sufficiently such that it can no longer keep up with the network bandwidth. However, kernels operating on data streams retain the sequential memory access pattern and can thus benefit from the increased bandwidth and operate at 100 G. We illustrate this through a *Hyperloglog (HLL) kernel*, which does cardinality estimation over data streams (Section 7.2).

7.1 Evaluation at 100 G

To evaluate StRoM at 100 G we use a Xilinx VCU118 FPGA board equipped with a Xilinx UltraScale+ XCVU9P FPGA. Figure 12 shows the evaluation of the 100 G StRoM NIC using the same microbenchmarks as in Section 6.1.

Latency: Two factors lead to a reduction in latency at 100 G in comparison to 10 G (Figure 12a). 1) The increased clock frequency from 156.25 MHz to 322 MHz. If we consider that many operations require a fixed number of cycles, the absolute time is reduced by more than 2x. 2) The wider data path reduces the amount of data words per packet (e.g., 176 vs 22 for a full MTU). The number of words determines the time taken by the store-and-forward when calculating/validating the ICRC of the RoCE packet. This can be observed by the smaller difference between 64 B and 1 KB payload at 100 G.

Throughput: Figure 12b shows that the 100G version of StRoM can saturate the available bandwidth once the payload is large enough. For messages smaller than 2 KB, the message rate is the limiting factor (Figure 12c). Messages are issued to the NIC through a single memory mapped AVX2 store operation containing all relevant parameters. Therefore the message rate is limited by the rate at which the application can issue these AVX2 stores and at which the I/O subsystem can serve them to the NIC over PCIe. Batching of application commands will eliminate this limitation of the current implementation.

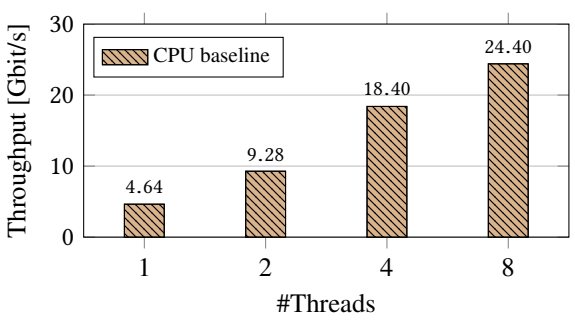
Table 3. Resource Usage of StRoM for 500 QPs on VCU118

	Logic [LUTs]		On-chip memory [BRAMs]		Register [FFs]	
10 G	92 K	7.8%	181	8.4%	115 K	4.8%
100 G	122 K	10.3%	402	18.6%	214 K	9.1%

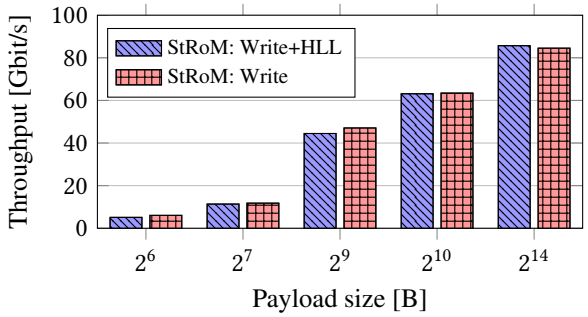
Resource Utilization: To have a fair resource comparison and to get a better understanding of the spatial size of the design, we compare the StRoM 100 G implementation on VCU118 with the StRoM 10 G implementation for the same FPGA. We observe that the numbers of used on-chip memory and registers have doubled, while the logic consumption has increased by 32%. In regards to on-chip memory, each BRAM corresponds to 36 Kb of FPGA on-chip memory. When increasing the data path by 8x and introducing additional registers to meet the higher target frequency, many parts such as data structures storing QP state, the TLB, or the DMA engine are untouched, resulting in an overall resource increase of 2x or less.

7.2 Cardinality Estimation Using Hyperloglog (HLL)

Statistics on a data set, such as cardinality, are valuable metrics in data processing applications. They can influence the choice of an optimal algorithm for a certain operation or determine the required memory footprint in advance. By implementing HLL as a StRoM kernel, we can gather this statistic as a by-product of data reception, e.g., when data is received using RDMA from a storage node by a compute node.



(a) Performance of HLL on the CPU receiving the data through StRoM



(b) Performance of HLL when processed on StRoM

Figure 13. HLL-throughput on (a) the CPU and (b) StRoM

Hyperloglog: HLL [15] is a sketch algorithm used to estimate cardinality, i.e., the number of distinct data items in a data set. HLL achieves sub-linear space complexity by approximating the cardinality calculation. HLL is widely used by companies such as Google to analyze large data collections [16]. We have implemented HLL as a StRoM kernel operating at 100 G and compare it against an optimized (AVX2), multi-threaded CPU only implementation that has performance comparable to that published in previous systems [16].

In a first experiment, we feed data to the server through RDMA using StRoM without processing the data on the NIC side. The CPU (Intel Core i7-7700 CPU @ 3.60GHz) runs HLL while data is received and placed into the memory by the NIC. The results shown in Figure 13a indicate that we need 8 threads to reach a throughput of about 25 Gbit/s. There are several factors that determine the observed performance. On the one hand, StRoM and the HLL code running on the CPU are competing for memory access. We observe a higher throughput for the HLL CPU version when the data is resident in memory instead of being transmitted (still well below 100 Gbit/s). On the other hand, HLL is memory bound as it uses a hash table to approximate how many times it sees an item, inducing many random memory accesses. These two factors combined result in a throughput that is far lower than the data arrival rate.

In a second experiment, we perform the HLL computation as a StRoM kernel and compare the performance of StRoM when performing a plain RDMA write and when computing HLL on the stream during an incoming RDMA write. The results are shown in Figure 13b and indicate that StRoM can perform HLL over the RDMA stream with no overhead and can reach line rate once the packet sizes are sufficiently large.

8 Related Work

Hoefler et al. [17] have addressed the lack of a programming interface to offload simple packet processing to the NIC. Li et al. [31] propose KV-Direct. It uses an FPGA-based programmable NIC to extend RDMA with a PUT and GET verb to support key-value store operations natively. The work focuses on the key-value store processors accessing the host memory over PCIe. How the Infiniband protocol is extended and how the new verbs are implemented is not described in detail. We have shown that StRoM kernels could also implement this type of functionality without the need to introduce specific PUT and GET verbs. In addition, StRoM is extensible and can support a wider range of applications. Microsoft [14] has deployed in their Azure cloud an FPGA-based SmartNIC to offload their SDN network stack into hardware. The architecture is very similar to that of StRoM and differs from commercial SmartNICs in using an FPGA rather than an array of ARM cores. NetFPGA [58] is a platform to facilitate research in the area of packet processing, switching and routing. StRoM could play a similar role for research on pushing data processing closer to the network while benefiting from the low latency of RDMA. Tsai et al. [52] introduce LITE a local indirection tier in kernel space to virtualize and manage RDMA for datacenter applications. The high-level abstractions provided by LITE could benefit from hardware offloading as provided by StRoM to reduce the CPU load and latency. In the area of databases, recent work exploring in-network data processing and RDMA [1, 30] discusses interesting possibilities for the functionality provided by StRoM.

9 Conclusion

In this paper we present StRoM a system supporting the deployment of processing kernels to the NIC. We have shown that StRoM can accelerate both remote RPCs and stream processing, thereby having the potential to improve latency and throughput in distributed systems by bringing processing closer to the data. We present four kernel examples and their use cases to illustrate that even simple operations on the NIC can have a significant impact on operations at the application level. StRoM kernels offer the ability to either accelerate or reduce the complexity of a wide range of applications. To our knowledge, StRoM is the first open source implementation of an RDMA NIC that we are making available with the goal of facilitating and encouraging further research in the area.

Acknowledgements

We would like to thank Xilinx for the generous donation of the equipment used in this work.

References

[1] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. Dpi: The data processing interface for modern networks. In *CIDR*, 2019.

[2] Altera. Programming FPGAs with OpenCL. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf, 2013.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[4] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson. The portals 4.0.2 network programming interface. http://www.cs.sandia.gov/Portals/portals402.pdf, 2017.

[5] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, 2015.

[6] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. *PVLDB*, 2017.

[7] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 2016.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, 2011.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 2014.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 2013.

[11] J. R. Corbin. Rpcgen. In *The Art of Distributed Applications*, pages 179–206. Springer, 1991.

[12] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 2016.

[13] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, 2014.

[14] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, et al. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*, 2018.

[15] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science*, 2007.

[16] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, 2013.

[17] T. Hoefler, S. D. Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance streaming Processing in the Network. In *SC*, 2017.

[18] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. *PVLDB*, 2017.

[19] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *NSDI*, 2016.

[20] Z. István, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *SIGMOD*, 2014.

[21] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. In *NSDI*, 2018.

[22] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*, 2017.

[23] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.

[24] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 2014.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.

[27] K. Kara and G. Alonso. Fast and robust hashing for database operators. In *FPL*, 2016.

[28] K. Kara, J. Giceva, and G. Alonso. Fpga-based data partitioning. In *SIGMOD*, 2017.

[29] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 1998.

[30] A. Lerner, R. Hussein, and P. Cudre-Mauroux. The case for network accelerated query processing. In *CIDR*, 2019.

[31] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *SOSP*, 2017.

[32] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with switchkv. In *NSDI*, 2016.

[33] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *EuroSys*, 2017.

[34] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *SIGCOMM*, 2019.

[35] Mellanox. Mellanox asap2 accelerated switching and packet processing. https://www.mellanox.com/related-docs/products/SB_asap2.pdf, 2019.

[36] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *ATC*, 2013.

[37] B. Networks. Tofino: World's fastest p4-programmable ethernet switch asics. https://barefootnetworks.com/products/brief-tofino/, 2019.

[38] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: a computation model for intelligent memory. In *ISCA*, 1998.

[39] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018.

[40] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, 2014.

[41] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna. Multi-dimensional packet classification on FPGA: 100 gbps and beyond. In *FPT*, 2010.

[42] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB*, 1998.

[43] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 2015.

[44] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo. Limago: An fpga-based open-source 100 gbe TCP/IP stack. In I. Sourdis, C. Bouganis, C. Álvarez, L. Toledo, P. Valero-Lara, and X. Martorell, editors, *FPL*, 2019.

[45] D. Sidler, G. Alonso, M. Blott, K. Karras, K. A. Vissers, and R. Carley. Scalable 10gbps TCP/IP stack architecture for reconfigurable hardware. In *FCCM*, 2015.

[46] D. Sidler, Z. István, and G. Alonso. Low-latency TCP/IP stack for data center applications. In *FPL*, 2016.

[47] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *HotSDN*, 2013.

[48] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is faster than server-bypass with RDMA. In *EuroSys*, 2017.

[49] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic RDMA-based replication. In *ATC*, 2018.

[50] J. Teubner and L. Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[51] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *FAST*, 2013.

[52] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *SOSP*, 2017.

[53] Z. Wang, B. He, and W. Zhang. A study of data partitioning on opencl-based fpgas. In *FPL*, 2015.

[54] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. Enabling FPGAs in hyperscale data centers. In *UIC-ATC-ScalCom*, 2015.

[55] L. Woods, Z. István, and G. Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 2014.

[56] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GraM: Scaling graph computation to the trillions. In *SoCC*, 2015.

[57] Xilinx. Vivado HLS. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2019.

[58] N. Zilberman, Y. Audzevich, et al. NetFPGA SUME: Toward 100 gbps as research commodity. *IEEE Micro*, 2014.