Zerializer: Towards Zero-Copy Serialization

Adam Wolnikowski
Yale University
USA

Stephen Ibanez
Stanford University
USA

Jonathan Stone
Intel, Barefoot Switch Division
USA

Changhoon Kim*
Stanford University
USA

Rajit Manohar
Yale University
USA

Robert Soulé
Intel, Barefoot Switch Division and
Yale University
USA

ABSTRACT

Achieving zero-copy I/O has long been an important goal in the networking community. However, data serialization obviates the benefits of zero-copy I/O, because it requires the CPU to read, transform, and write message data, resulting in additional memory copies between the real object instances and the contiguous socket buffer. Therefore, we argue for offloading serialization logic to the DMA path via specialized hardware. We propose an initial hardware design for such an accelerator, and give preliminary evidence of its feasibility and expected benefits.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; **Application layer protocols**; • **Hardware** → **Communication hardware, interfaces and storage**.

KEYWORDS

serialization, network interface card, direct memory access

ACM Reference Format:
Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3458336.3465283

*This work was done when the author was at Intel, Barefoot Switch Division.

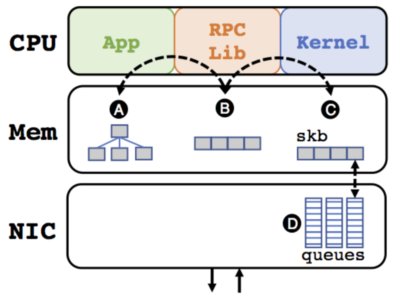

Figure 1: Memory copies in a traditional network stack.

1 INTRODUCTION

Achieving zero-copy I/O has long been an important goal in the networking community. Today, some operating systems and NICs have been able to achieve this goal with a variety of techniques [34], including direct memory access (DMA) and memory-mapping through an memory management unit. [1]

However, application-level serialization creates a challenge for zero copy I/O. By application-level serialization, we mean the process by which the in-memory representation of data structures is converted into architecture and language independent formats that are transmitted across the network. Distributed applications commonly rely on serialization libraries, such as Google's Protocol Buffers (Protobuf) [29] or Apache Thrift [2], when sending data from one machine to another.

The problem is that application-level serialization encoding requires the CPU to read, transform, and write message data, resulting in additional memory copies between the real object instances and a contiguous buffer. Figure 1 illustrates the numerous memory copies required for application-level serialization, starting from in-memory RPC objects (Ⓐ), to encoded RPC messages (Ⓑ), to kernel socket buffers (Ⓒ), and to NIC queues (Ⓓ).

[1] These techniques—and this proposal—do not truly ensure zero copy. At minimum, they require one memory copy to access main memory, read the object instances, and copy them to the NIC memory. Our use of the term *zero copy* is purely from the CPU's perspective: CPU cores are not involved any memory copy operations.

These additional memory copies are a huge source of overhead, and consequently, application-level serialization is one of the most widely recognized bottlenecks to system performance. As a prime example, Kanev et al. [16] report that at Google, serialization and RPCs are responsible for 12% of all fleet cycles across all applications. Indeed, they are referred to as the "data center tax", i.e., the cycles that an application must pay simply for running in a data center.

This bottleneck is not just due to inefficient operations (although we will argue in § 2 that implementing serialization on a CPU is inefficient), but also due to the frequency of invocation. The emerging trend in software engineering towards microservices and serverless/functionless computing, which advocates for the fine-grained decomposition of software modules, has exacerbated the problem. Invoking an innocuous-sounding method, `getX`, may result in hundreds or even thousands of RPC calls, each requiring data serialization. The result is the proverbial death by a thousand cuts.

To truly achieve high-performance message transfer, we must address this bottleneck, and eliminate the extra memory copies. Standard DMA techniques are not sufficient, as serialization requires not just memory copies, but also data transformation. We therefore argue that application-layer message processing logic should be added to the DMA path.

Of course, adding a serialization accelerator to NIC hardware assumes that the network transport protocol is also offloaded. Transport protocol offload itself has been the subject of a long-standing debate [24]. The question is whether the potential performance benefits of offloading to a NIC outweigh the costs of the added hardware complexity. However, the combination of the end of Moore's law, the increase in network speeds, and the wide-spread adoption of RDMA has recently led to revived support for transport protocol offload. There are several contemporary proposals for offloading network stacks and basic network functions [3, 8, 21].

We argue that these first steps towards offload do not go far enough, and advocate for the inclusion of hardware for application-layer message processing in a SmartNIC (§2). We have developed an initial hardware design for such an accelerator, which uses hard-coded building blocks for performing data transformations, and a programmable configuration that allows them to support application specific message formats (§3). We refer to our design as Zerializer, which allows for zero-copy I/O augmented with serialization in the DMA path, or what we call *zero-copy serialization* directly between Ⓐ and Ⓓ.

2 WHY SERIALIZATION OFFLOAD?

Stagnating CPU performance has led inevitably to the adoption of domain-specific processors. This trend raises a pair of important questions. First, *what functionality can and should benefit from hardware acceleration or offload?* Second, *how should this functionality be incorporated into the overall architecture?* We consider these questions from two perspectives: the hardware designer and the software developer. From the hardware perspective, we want to build custom hardware for functionality or logic that is unlikely to change in the near future, i.e., standards or other components upon which a community has converged. From the software perspective, we want to accelerate functionality that is a bottleneck. But, somewhat more subtly, we want to focus on offloading computation where the freed-up CPU cycles could be productively used on something else.

From both the hardware and software perspective, serialization is a natural candidate for hardware acceleration. Serialization formats are standardized and do not change frequently. They are also a performance bottleneck. And, just as with zero-copy I/O, if serialization were offloaded, the CPU could focus exclusively on application logic. Importantly, though, we argue not only should serialization be offloaded, but it should be on the DMA path.

What is serialization? Serialization is the process that translates the in-memory representation of data structures into architecture and language independent formats that are sent across the network or stored to disk. There are a wide range of serialization formats, differing along several dimensions: self-describing vs. schema based, text-based vs. binary encoding, and support for collections or other types. In this paper, we focus on serialization libraries like Protocol Buffers [29] and Thrift [2], because they are widely adopted. When using such libraries, developers define the data structures they will use for messages using a domain-specific language. A compiler translates the message specification into sender and receiver code that can be invoked from the application. Both of these serialization formats are schema based, binary encodings with support for various complex data types, such as lists and maps. These structures may be nested, and the in-memory representations may contain pointers to different memory locations. The serialization process must convert the pointers to references based on name or position.

Why is serialization a bottleneck? When an application uses serialization, it is typically accessing fresh data recently received over the network. This takes a number of steps. First, accessing encoded data for the first time results in cold misses through memory hierarchy—either to main memory, or to the last level cache in the event the network interface places incoming data in the last level cache. Second, the CPU must perform the computations to encode/decode the data. Third, the native data has to be written back to memory.

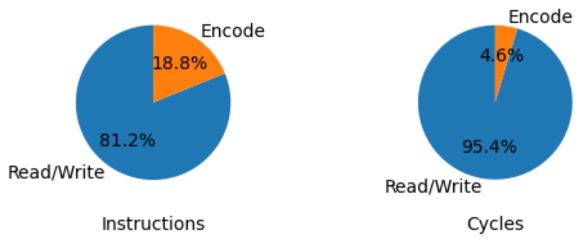

Figure 2: The makeup of Protobuf serialization for a message of three, 64-bit unsigned integers.

Of course, the cost of each of these operations is workload and message dependent. However, to give a basic understanding of the cost of these first and third steps, reading and writing, relative to the second step, encoding, we profiled the serialization method, `SerializeToOStream`, of Protobuf's Message class (v3.14.0) for a simple message containing three, 64-bit unsigned integers. This message could represent a point in 3D space or a small set of numerical IDs. To perform the measurement, we used the Performance Application Programming Interface [25].

Figure 2 shows the results, in terms of instructions and cycles, averaged over 1000 runs. We see that memory accesses dominate the overhead. So, simply accelerating or offloading the data transformation alone [28] will not sufficiently address the bottleneck. However, because some computation must be performed to do the data encoding, DMA alone is also not a viable solution.

Could we just make software changes? A great variety of serialization libraries and formats have been proposed over the years, making different sets of trade-offs (e.g., human readability vs. space efficiency, performance vs. expressiveness, etc.). Some implementations are known to have poor performance. For example, Java Serialization [13] due to its dependence on reflection. Some performance improvements could be made through careful design decisions. Flatbuffers [9] is a less expressive, more efficient version of Protobuf. However, there is a limit to what software-only changes can achieve. They do not address the fundamental problem of the extra memory copy.

Why not just use another CPU? Standard microprocessors are not optimized for the bit-level operations that are needed for encoding/decoding [28]. Additionally, data encodings also tend to be sequential. As an example, consider the `varint` encoding used in Google Protobuf. The encoding uses the most-significant bit in each byte to indicate if the next byte is part of the same integer. This creates sequential dependency chains with conditional branches, which are challenging to execute efficiently on a CPU. Hence, a standard CPU instruction set is not well-matched for serialization.

Why not add extra instructions or a CPU accelerator/co-processor? Instead of a complex CPU, we could instead augment an existing one with special instructions that are tailored to serialization to reduce the computational overhead. In an extreme case, the CPU could be augmented with a dedicated accelerator/co-processor tailored to serialization tasks. While this addresses the computational needs, we are still left with the significant memory system performance issues. The transformation must be on the DMA path.

How does this interact with DMA? Consider an incoming data message where the NIC can directly transfer the arriving data into buffers accessible in user space. The NIC can use a DMA engine to transfer the data directly to the main memory, and/or the last level cache.

Serialization effectively creates a second copy of incoming data. The encoded data travels up the memory hierarchy, is processed, and then travels back down in decoded form—the second copy. This uses the cache inefficiently, especially since the data is only touched once for it to be decoded. Hence, having an application-driven CPU/co-processor approach effectively undoes all the engineering effort that went into eliminating extra copies and achieving "zero copy" network interfaces.

Why now? As network speeds reach or exceed 100 Gbps, and common operations complete in microsecond-scale, serialization is a big source of overhead. Given that the plateauing performance of general-purpose processors has led to renewed interest in transport protocol offload on to NICs, there is an opportunity to significantly reduce this overhead by additionally offloading serialization/deserialization into the NIC, where it sits on the DMA path. This is feasible because serialization is a stable, straightforward transformation, not much more complicated than typical packet processing. Moreover, if we focus on intra-data-center RPCs, serialization offload would not require excessive NIC memory, as the NIC need fetch only a bandwidth-delay product (BDP)-full of data from the main memory in a streaming fashion. The BDP of a modern data center is only about a few hundred KBs (e.g. 100Gbps $\times$ 10μsec = 125KB).

3 PROPOSED DESIGN

Figure 3 depicts how we propose to integrate Zerializer on the NIC. At a high level, the interface exposed to software is very similar to the one provided by modern DMA-based NICs. However, rather than passing packet descriptors, applications directly pass RPC message descriptors. RPC messages are automatically serialized on transmission as they are DMA'ed from host memory, and they are automatically deserialized on

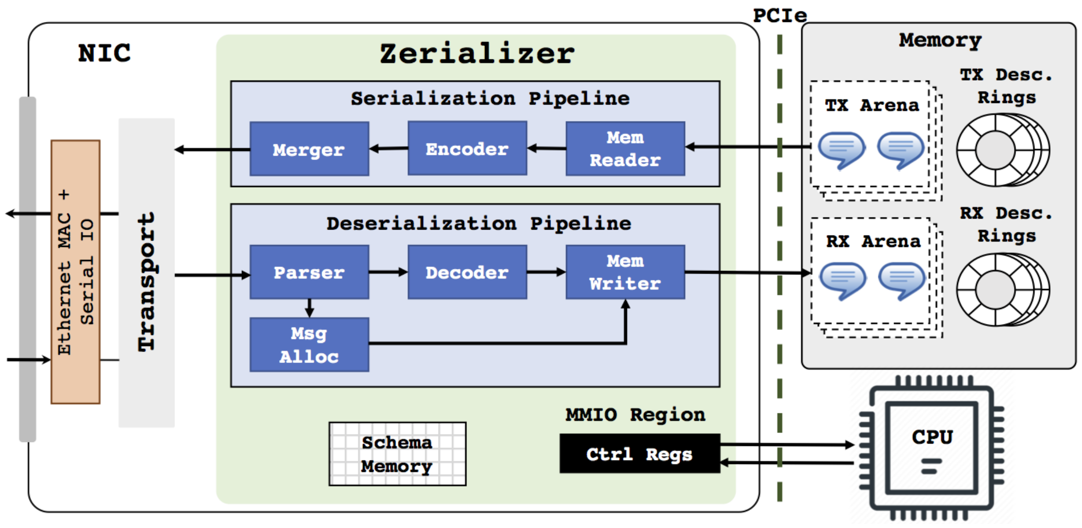

Figure 3: A diagram of the Zerializer architecture and how it is integrated into the NIC.

reception as they are DMA'ed into host memory. This section explains our proposed software interface and the Zerializer architecture in more detail.

Initialization. Prior to sending and receiving RPC messages, an application must perform several initialization operations. First, it must allocate both TX/RX descriptor rings as well as TX/RX arenas and register them with Zerializer by configuring the appropriate control registers with MMIO operations. The TX and RX arenas are used to store all RPC message objects that are sent and received, respectively. In order to ensure that Zerializer is able to quickly access these message objects, the host must populate the IO-TLB with the necessary virtual-to-physical address translations. Note that modern systems which use traditional NICs perform a similar step to ensure that the DMA engine is able to quickly access all the necessary packet buffers. The descriptor rings store pointers to RPC message objects, which are allocated within the TX and RX arenas. Each application uses a separate pair of descriptor rings and arenas.

Next, the application describes the format of the RPC messages it wishes to use in a domain specific language (DSL) such as Protobuf [29]. The DSL definition of the message format is then compiled into what we call a message schema, which must be loaded into the Zerializer module. This schema is used to properly serialize and deserialize RPC messages.

Message Transmission (Serialization). Before an RPC message can be transmitted, it must be constructed in memory. The application will first allocate a message object from the TX arena and initialize its fields. To transmit, the application creates a TX descriptor, adds it to the TX descriptor ring, and notifies Zerializer using an MMIO write operation. The descriptor includes a pointer to the message object, the message type, as well as a status bit.

Upon receiving the transmission request, the memory reader first fetches the descriptor from the TX descriptor ring. Then, using knowledge of the message schema, it fetches each field of the message. After every field is acquired, Zerializer marks the message as complete by updating the status bit in the message descriptor. This signal indicates to the application that it is now allowed to reclaim the message buffer space.

Each field is transformed independently in the encoder. The data transformation logic to use for each field is dictated by the field type, which is specified in the message schema. The encoder contains dedicated hardware to transform each field type; and it contains a sufficient amount of parallel hardware resources to ensure that the encoding logic is not a throughput bottleneck of the system. The merge module at the end of the serialization pipeline accumulates the transformed fields and combines them into the final encoded message. The encoded message is then passed to the NIC's transport layer so that it can be delivered over the network to the destination.

This architecture significantly improves serialization performance because it explicitly reads and transforms each field in parallel. In traditional software-based serialization, message fields are only operated on in parallel if the CPU can speculate far enough ahead in the instruction stream, which is unlikely on modern processors [28].

Message Reception (Deserialization). The NIC transport layer reassembles packets into wire-format messages and passes them to Zerializer to be decoded. The parser module identifies the message type and informs the message allocator. A standard NIC will DMA packet data into one or more buffers that have been preallocated by host software. This approach works well for packet data which is simply treated as a sequence of raw bytes where multiple buffers can be linked together to store the whole packet. However, RPC message objects must be laid out in memory in a very specific format in order to be useful to applications. Unless the host knows exactly how many of each message type it will need to process at any given time, using preallocated message objects will likely lead to a very inefficient use of memory resources. If the host software is required to allocate a message object for each RPC as it arrives over the network, then the NIC would be unable to immediately deliver decoded messages into host memory. In order to avoid these issues, we choose to let Zerializer manage message object allocation in the RX arenas. Recent work has demonstrated that memory allocation in hardware is fast and inexpensive [17].

As the parser extracts each field from the encoded message, the decoder transforms them into the format expected by the application. Similar to the encoder, the decoder contains dedicated hardware to transform each field type. Using knowledge of the message format from the schema, the memory writer loads the decoded fields into the newly allocated message object. Once the entire message has been loaded into memory, Zerializer will create an RX descriptor and add it to the RX descriptor ring. The application becomes aware of the new message either by polling the RX descriptor ring, or Zerializer can be configured to generate an interrupt upon completing a RX operation. After the application finishes processing the message, it updates the RX descriptor ring, which in turn allows Zerializer to free the message object in the RX arena.

Towards RPC Offload. RPC performance optimization has been widely studied [5] and has recently seen increased interest due to the trend towards microservices [7, 15, 33]. Zerializer also allows for offloading RPC. On the sender side, we would simply augment the message format with an additional tag for the function ID (i.e., the unique identifier for the application function). On the receive side, the RPC mechanism could be implemented as a function dispatch table to DMA. Note that with this design, an application would register its intent to receive a particular type of message, and pre-allocate a buffer on which to receive that message (or a set of messages, if we wanted queuing). This is slightly different from regular RPC interface. But, the results would be completely offloaded RPC calls.

Discussion. The data transformation component of Zerializer is similar to Optimus Prime [28]. However, Optimus Prime is integrated into the host as a co-processor, whereas we propose to offload this logic onto the NIC. We believe this approach provides a number of benefits. First, by offloading to the NIC, it means that the application does not need to waste cycles moving network data between DMA buffers and the ser/des accelerator. This is especially important for applications with μs-scale RPC service times, which are becoming increasingly prevalent. Second, this approach prevents the cache hierarchy from being polluted by encoded message data, which is useless to applications. This increases the cache hit rate for application data leading to higher performance. And finally, it makes applications easier to write. Separating the ser/des logic from the network interface means that applications need to manage them separately, which leads to more complicated code.

4 BENEFITS AND FEASIBILITY

Evidence of Potential Benefits. To evaluate the potential performance benefits of using Zerializer on real-world applications, we examined Intel's Deep Insight Network Analytics Software. The Deep Insight software collects network telemetry data from switches in the form of "reports" in the In-band Network Telemetry (INT) specification, similar to the work by Handigol et al. [11]. It then analyzes the data in those reports looking for network anomalies, and publishes the results of those analyses to a time-series database in Thrift-formatted records via the Apache Kafka [1] message broker using librdkafka [22].

To quantify the impact of serialization, we measured the throughput of Deep Insight running with a single thread under two configurations: one with Thrift serialization and publishing to Kafka, and one with a "null" serializer, that just counts incoming objects to be serialized and discards them. We believe it is reasonable to not measure the publishing time, since Zerializer implicitly assumes the transport protocol offload to the NIC. The experiment was run on a server with an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with 32 cores (non-hyper-threaded), 128GB of memory, and a 40GbE 5th-gen Mellanox NIC.

In both cases, an identical server used `tcpreplay` to replay a pcap file of telemetry reports, at a rate approximately 4x what the single-threaded Deep Insight can process (~200k/sec). With Thrift serialization and publishing enabled, a single worker thread saturates at processing 52,356 reports/sec. With the null serializer, the thread reached 147,991 reports/sec, or roughly a 3× increase. This demonstrates that Zerializer has the potential to tremendously improve application performance.

Feasibility. In order to evaluate the feasibility of offloading message serialization into NIC hardware, we designed and implemented one of the key building blocks that would be required. Specifically, we implemented a Verilog module for Protobuf's varint encoding logic [30]. The module consumes 8B unsigned integers and produces the corresponding variable length encoded version. We synthesized the design to a Xilinx Ultrascale+ FPGA. Table 1 shows the corresponding resource

	Utilization	Relative to IceNIC	Relative to Rocket core
LUTs	254	3.8%	4.8%
Regs	295	7.4%	14%

Table 1: Xilinx Ultrascale+ resource utilization of our varint encoding module. Also includes the resource utilization relative to a simple DMA-based NIC (IceNIC [18]) and simple 5-stage, in-order, RISC-V core (Rocket core [4]).

utilization. To provide some context, the table also shows the resource utilization relative to a simple DMA-based NIC, called IceNIC [18] and a simple 5-stage, in-order, RISC-V core called the Rocket core [4]. Adding our varint encoding module to the NIC would only increase the size by a very small fraction. Obviously, a varint encoder is only one of Zerializer's many building blocks, but this is a promising start.

Zerializer could be deployed on an FPGA-based SmartNIC or custom silicon could be added to the NIC ASIC. On an FPGA running at 300MHz, our encoder module would be able to process 300M unsigned integers / second, which corresponds to ~19Gbps. This means that to saturate 100Gbps, the design would require about 5 encoders in parallel. On the other hand, an ASIC running at 2GHz would be able to process 128Gbps with a single module.

5 RELATED WORK

Heterogeneous and Programmable NICs. There are several proposals on heterogeneous architectures which do co-processing between a NIC and CPU, including Gallium [35], ClickNP [21], Floem [26], iPipe [23], and UNO [20]. A Zerializer-approach to serialization could be compatible with such designs. There has been an uptick in interest in programmable NICs in the past few years [10, 12, 19, 27, 32]. These systems have proposed using the programmability for transport protocol offload and some basic network functions, such as load-balancing of advanced congestion control. None of them have proposed serialization offload.

Optimizing Serialization. A wide range of memory-copy reduction techniques have been proposed which hope to reduce the overhead of sending data over the network. Varghese [34] provides a good overview of techniques explored in various end host stacks.

There have been several attempts to optimize serialization through software improvements, such as Flatbuffers [9] and Cap'n proto [6]. A contemporaneous proposal by Raghavan et al. [31] attempts to use existing support for scatter-gather memory operations in modern NICs, without making any hardware modifications. In contrast to all of these software-only approaches, Zerializer argues for the inclusion of new hardware in the NIC to perform message field encoding/decoding.

Serialization Accelerators. Recent work has proposed specialized hardware for serialization, including Optimus Prime [28], Cereal [14], and HGum [36]. Of these, Optimus Prime is closest in spirit to Zerializer, as the designs for data transformation in the two systems are similar. Zerializer differs in that it argues for adding data transformation logic to the DMA path, as opposed to a co-processor.

6 CONCLUSION

Data serialization is a significant bottleneck to application performance, and is likely to become the dominant contributing factor to end-to-end latency as applications increasingly rely on RPC calls to fine-grained components. To reduce this overhead, we argue that serialization offload will be essential and inevitable. We have proposed a design for DMA augmented with data transformation logic, allowing for zero-copy serialization. This design would dramatically improve application performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers who helped us improve this paper. This work was partially supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0107.

REFERENCES

[1] Apache Kafka 2021. Apache Kafka. https://kafka.apache.org.
[2] Apache Thrift 2021. Apache Thrift. https://thrift.apache.org.
[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.
[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1990. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems (TOCS)* 8, 1 (1990), 37–55.
[6] Cap'n proto 2021. Cap'n proto. https://capnproto.org.
[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[8] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[9] FlatBuffers 2021. FlatBuffers. https://google.github.io/flatbuffers/.

[10] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2020. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 38–46.

[11] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[12] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. 2019. The Case for a Network Fast Path to the CPU. In *17th Workshop on Hot Topics in Operating Systems (HotOS)*.

[13] Interface Serializable 2021. Interface Serializable. https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html.

[14] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *47th International Symposium on Computer Architecture (ISCA)*.

[15] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[16] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *42nd International Symposium on Computer Architecture (ISCA)*.

[17] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. *Notices of the ACM Special Interest Group on Programming Languages (SIGPLAN Notices)* 52, 4 (2017), 33–45.

[18] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *45th International Symposium on Computer Architecture (ISCA)*.

[19] Antoine Kaufmann, Simon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[20] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. 2017. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *8th ACM Symposium on Cloud Computing (SOCC)*.

[21] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *ACM Annual Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[22] librdkafka 2021. librdkafka. https://github.com/edenhill/librdkafka.

[23] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *ACM Annual Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[24] Jeffrey C. Mogul. 2003. TCP Offload is a Dumb Idea Whose Time Has Come. In *9th Workshop on Hot Topics in Operating Systems (HotOS)*.

[25] Performance Application Programming Interface 2021. Performance Application Programming Interface. https://icl.utk.edu/papi/.

[26] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. 2019. Flowblaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[28] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[29] Protocol Buffers 2021. Protocol Buffers. https://developers.google.com/protocol-buffers.

[30] Protocol Buffers Encoding 2021. Protocol Buffers Encoding. https://developers.google.com/protocol-buffers/docs/encoding.

[31] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of Champions: Efficient Datacenter Serialization. In *18th Workshop on Hot Topics in Operating Systems (HotOS)*.

[32] Brent Stephens, Aditya Akella, and Michael M Swift. 2018. Your Programmable NIC Should be a Programmable Switch. In *17th Workshop on Hot Topics in Networks (HotNets)*.

[33] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *12th European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3064176.3064189

[34] George Varghese. 2010. *Network Algorithmics* (2nd ed.). Chapman & Hall/CRC.

[35] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *ACM Annual Conference of the Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*.

[36] S. Zhang, H. Angepat, and D. Chiou. 2017. HGum: Messaging Framework for Hardware Accelerators. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.