Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather

Deepti Raghavan[♠], Philip Levis[♠], Matei Zaharia[♠], Irene Zhang[♦]
[♠]Stanford, [♦]Microsoft Research

Abstract

The era of microsecond I/O will make data serialization a major bottleneck for datacenter applications. Serialization is fundamentally about data movement: serialization libraries coalesce and flatten in-memory data structures into a single transmittable buffer. CPUs are not optimized for data movement, so software serialization approaches will hit a performance limit and be unable to keep up with modern networks.

We observe that widely deployed NICs possess scatter-gather capabilities that can be re-purposed to accelerate serialization's core task: coalescing and flattening in-memory data structures. These capabilities make it possible to build a completely *zero-copy, zero-allocation* serialization library, which introduces many research challenges of its own. These challenges include using the hardware capabilities efficiently for a wide variety of non-uniform data structures, making application memory available for zero-copy I/O, and ensuring memory safety.

ACM Reference Format:
Deepti Raghavan, Philip Levis, Matei Zaharia, Irene Zhang. 2021. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotOS '21), May 31–June 2, 2021, Cyberspace, People's Couches, and Most Likely Zoom.* ACM, New York, NY, USA, 7 pages. https://doi.org/XX.XXX/XXX_X

1 Introduction

The microsecond era is here [5]. As shown in Figure 1, datacenter applications today can achieve microsecond packet round-trip times, reaching single digit RTTs with kernel-bypass. At these latencies, everyday systems services, like data serialization, become unaffordable bottlenecks.

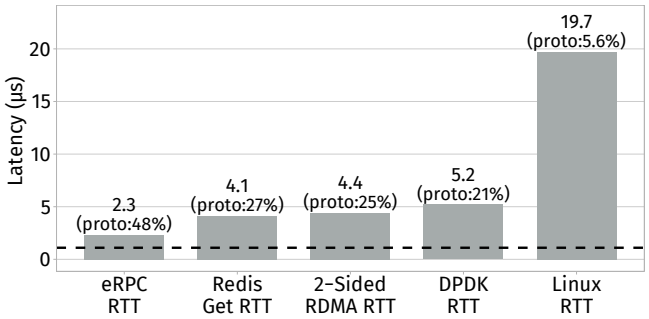

Figure 1: **Reported RTTs of recent microsecond-scale systems, annotated with the percentage overhead that Protobuf serialization and deserialization of a single 1024 byte string (1.1 µs) would add (shown in the dotted line). Redis RTT comes from Arrakis [27], eRPC from eRPC [16], while the RDMA, DPDK and Linux RTTs are measured on the Demikernel. [40].**

Data serialization [2, 3, 35–37] is important for datacenter applications. Many distributed applications [1, 32, 39], RPC libraries [12], and microservice deployments [9] rely on serialization as a communication primitive, but serialization already causes a big performance penalty. Google reported that Protobuf [36] accounts for 5% of its datacenter cycles [17] 2015, and we expect the problem to worsen.

As a concrete example, we find that Protobuf takes 1.1 µs to serialize and deserialize a simple data structure with a single 1024 byte-sized string. We overlay this overhead on Figure 1. Protobuf serialization for this data structure adds a staggering *48% overhead* to eRPC [16]. Each extra microsecond of serialization overhead significantly affects the throughput a server can achieve and the number of cores necessary to saturate the network.

The main problem is that general-purpose CPUs cannot perform serialization's core task efficiently enough. Serialization must move data, because there is fundamental tension between the application's optimal in-memory layout and the network's optimal on-the-wire layout for a data structure. Data structures often contain pointers (e.g., trees and graphs), so applications can easily modify data structures without having to re-allocate all the memory contiguously. Serialization coalesces these scattered pointers into a contiguous buffer for transmission. Performing this data movement in software will limit throughput in modern

HotOS '21, May 31–June 2, 2021, Cyberspace, People's Couches, and Most Likely Zoom

ACM ISBN XXX-XXXX-XX-XXX/XX/XX.
https://doi.org/XX.XXX/XXX_X

networks, because it requires copying each field at least once and providing a buffer to store the final result.

Without high performance serialization libraries, applications are forced to hand-roll their own serialization or integrate custom hardware accelerators. Redis [31] and RAMCloud [26] improve CPU-based serialization by making it less general-purpose, but cannot avoid the overhead required to move memory. The most complicated object Redis can serialize is a list. On the other hand, deploying and integrating custom hardware accelerators that do serialization [14, 28] can be difficult in today's datacenters as it requires extra coordination between network administrators, offload developers and application developers [21].

Our key observation is that while CPUs coalesce scattered memory regions inefficiently, widely deployed NICs already possess hardware capabilities that perform a similar function: scatter-gather functionality. These capabilities were designed for high-performance computing, where applications frequently move large, statically-sized chunks of memory between servers. Kernel bypass exposes this NIC capability to the serialization library, but it is not obvious how to directly use it for serialization. Thus, this paper asks: *How can we leverage NIC scatter-gather capabilities to build serialization libraries that keep up with modern networks?*

The remainder of the paper describes why existing software serialization is inefficient (§2) and a simple use of NIC scatter-gather capabilities for serialization (§3). We finally discuss open research questions around building general-purpose serialization libraries with scatter-gather (§4) and related work (§5).

2 The Limits of Software Serialization

This section shows that CPU-based serialization cannot keep up with the peak packet processing throughput of kernel bypass I/O (§2.1), because CPU-based serialization cannot avoid certain data movement overheads (§2.2).

2.1 Software Serialization is Unaffordable

To demonstrate the overhead of serialization, we benchmarked three software serialization libraries [35–37] on DPDK and found that they only achieve up to 52% of DPDK's peak single core throughput. We only consider compilation-based serialization [2, 3, 35, 36] because dynamic type inference at runtime [19, 22] (e.g., Java serialization of arbitrary Java classes) imposes additional unaffordable overheads. We use a data structure with a single 1024-byte string field. Although the data structure is so simple that serialization is theoretically unnecessary, it captures the minimal overhead for serialization today.

The experiment runs on 11 20-core dual socket Xeon Silver 4114 2.2 GHz servers, connected by Mellanox ConnectX-5 100 Gbps NICs and an Arista 7060CX 100 Gbps switch, with

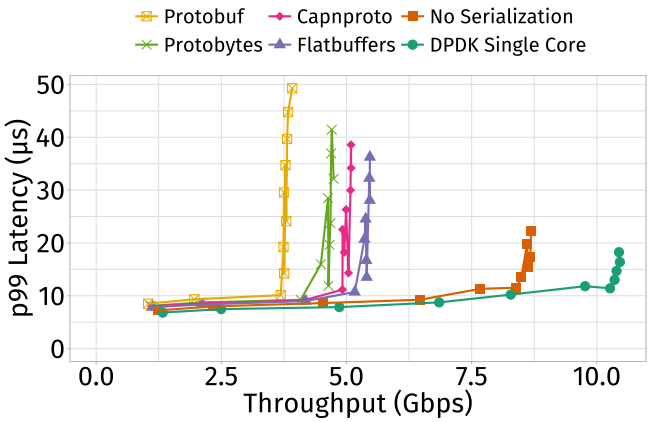

Figure 2: **Measured achieved throughput and p99 latency for sets of 1 to 20 concurrent clients (across up to 10 separate machines) pinging a single-core serialization echo server with a message containing a single 1024 byte string. No software serialization library can keep up with the peak zero copy throughput without serialization, which is about 10.4 Gbps.**

a minimum 450 ns of switching latency. We use concurrent, closed-loop clients to send a serialized message to the server, which deserializes, then re-serializes the same payload and returns it to the client. We use a minimal UDP networking stack for DPDK based on LWIP [7].

We show the results in Figure 2. The "No Serialization" line removes serialization and gives the raw network stack performance. Kernel bypass requires that packet memory lives in pinned, non-swappable pages, so the networking stack still copies application payloads into registered packet memory on transmission and copies packets into general memory on receive. The "DPDK Single Core" line removes these copies and represents the peak, zero-copy processing throughput possible with DPDK. We include another version of Protobuf, "Protobytes", where the payload is `bytes`, not a string, as Protobuf spends a significant amount of time in utf8-validation.

Experiment Results. FlatBuffers, the fastest serialization baseline, achieves only 5.4 Gbps, about 52% of DPDK's peak throughput, 10.4 Gbps (at 15 µs of tail latency), due to two performance gaps. Serialization itself contributes the first 3 Gbps gap between FlatBuffers and No Serialization. Having the networking stack and serialization manage memory separately contributes the 2 Gbps gap between No Serialization and DPDK Single Core. Section 2.2 closely breaks down these gaps.

2.2 Why is Software Serialization So Expensive?

Today's software serialization libraries are limited by moving data on CPUs. In-memory data structures often contain pointers, so serialization must flatten the data into a contiguous representation. Additionally, sometimes applications use serialization libraries to construct and transmit data structures

Step	Protobuf	Cap'n Proto
Initialize Data Structure	34 ns	352 ns*
Copy String Payload	172 ns*	85 ns*
Encode to Wire Format	544 ns*	52 ns
Decode from Wire Format	411 ns*	80 ns
Total Overhead	**1161 ns**	**569 ns**

Table 1: **Breakdown of steps to serialize and deserialize a message with a single 1024-byte-sized `string` field. Cap'n Proto's encode and decode are zero-copy because the in-memory buffer layout matches the eventual wire format, while Protobuf requires an expensive transformation to the wire format. Both libraries' copy and allocation-based overheads, marked by stars, scale with message size.**

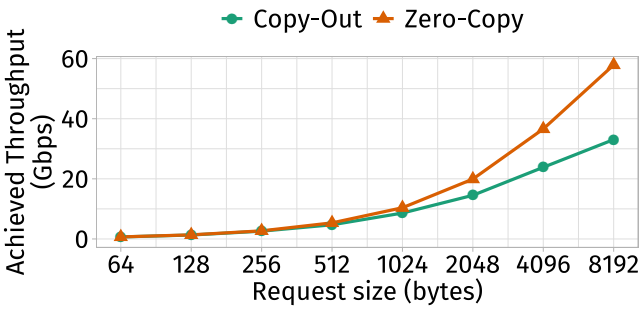

Figure 3: **Achieved throughput for 16 clients pinging a single-core echo server with different message sizes (transmitted as a single chunk, without scatter-gather). The server either copies the payload out to a transmit buffer or uses zero-copy transmission. The difference between zero-copy and copy-out becomes visible at payloads of at least 512 bytes. Note the log scale in the x-axis.**

on-demand to respond to application requests (e.g., returning the value of a range of specified keys in a key value store).

All serialization libraries, no matter their final wire-format, must pay the cost of the copies and allocations required for this data movement. Table 1 breaks down the serialization latencies from Figure 2 with Protobuf and Cap'n Proto (FlatBuffers behaves similarly to Cap'n Proto). After copying the field in ("Copy String Payload"), Protobuf performs an expensive transformation to the on-the-wire format. This transformation causes additional copies, allocations, and utf8-validation during "Encode", and corresponding costs during "Decode". Cap'n Proto's "Encode" and "Decode" are cheaper because the in-memory format matches the wire-format exactly, but even Cap'n Proto must allocate space for the serialized buffer ("Initialize Data Structure") and copy the payload in ("Copy String Payload") during transmission. For data structures with large payloads, data movement dominates serialization costs, while converting integers to network ordering, which few wire formats require, adds minimal cost.

The second performance gap in Figure 2 comes from the firm separation between the serialization library and networking stack. Modern kernel bypass stacks require that packets live in non-swappable, pinned memory, so they typically use their own buffers for I/O. Serialization libraries are unaware of the networking stack altogether, so there are inherently copies between the two. Completely eliminating the performance gap in Figure 2 would require tight integration between the serialization library, application and networking stack. This integration would involve agreeing on an interface, making pinned memory available, and coordinating ownership and memory safety of buffers. Fortunately, with kernel bypass, the networking stack, serialization library, and application are all in the same address space, so coordinating memory management may be possible (§4.3).

3 Leveraging the NIC for Serialization

To lower the cost of serialization, we must reduce CPU data movement. Our key insight is that datacenter servers already have a hardware accelerator for coalescing non-contiguous I/O regions: the NIC itself. Modern NICs have scatter-gather engines for high-performance computing, e.g., to coalesce the same non-zero elements of sparse matrices into packets millions of times. Networking stacks [8,33] have re-purposed scatter-gather to manage sending packets that are larger than the maximum packet buffer size. Serialization is more challenging because it needs to move many small fields whose size and placement dynamically depend on external data or user requests. This section describes the design of a prototype serialization library for the popular Mellanox CX-5 [24] NIC.

3.1 NIC Scatter-Gather Capabilities

Whether NIC scatter-gather can be used for high-performance serialization depends on its performance properties and restrictions. The section focuses on the CX-5; other modern scatter-gather NICs with PCIe interconnects likely behave similarly (§4.1).

Given a list of I/O addresses, a CX-5 makes multiple PCIe requests to coalesce the memory into a single packet. The NIC supports up to 60 scattered memory chunks, but each chunk requires another NIC-to-PCIe round trip. The number of these round trips that can make progress concurrently depends on hardware implementation details of the PCIe endpoint at the NIC and the CPU, which we currently do not have knowledge of. To understand this penalty, we ran an experiment where the DPDK echo server described in Section 2.1 transmits a pre-initialized payload of size 1024 bytes (no copies) equally divided into different numbers of chunks to a single client. The RTT increases from 6 us to a 10.5 us RTT when the message is sent as a single buffer, versus 60 scatter-gather chunks.

```
struct ScatterGatherArray {
    size_t num_entries;
    void * ptrs[MAX_ENTRIES];
    size_t length[MAX_ENTRIES];
};
```

Listing 1: **The scatter-gather array, the core abstraction for scatter-gather based serialization.**

Sending back the 1024-byte message as 16 chunks results in higher latency than using FlatBuffers to deserialize, reserialize and transmit the request (which requires copying the payload twice). These results suggest that, for a 1024-byte message, the "maximum" number of chunks should be less than 16.

There is also a tradeoff between the cost of an additional PCIe request and simply copying the memory. Figure 3 shows an experiment that measures the difference in achieved throughput for 16 clients pinging the single-core DPDK echo server with messages of varying size consisting of a single buffer. The payload is either pre-initialized ("Zero-Copy") or copied into the packet ("Copy-Out"). The only discernible difference between copy-out and zero-copy starts at about 512 bytes. Additionally, entries much smaller than 256 bytes could hurt performance. When the NIC reads memory regions over PCIe, the PCIe controller sends back 256-byte-sized memory chunks (the chunk size is a hardware setting). Each chunk contains a header, so the header could dominate in the case of small payloads.

These characteristics indicate that maximum performance on a CX-5 requires passing in I/O lists with entries that are at least 512 bytes large. The "maximum" number of entries in the I/O list depends on the size of each entry as well as how many concurrent DMAs can run. These tradeoffs preclude simple solutions, such as one scatter-gather operation per data structure field.

3.2 Integrating Networking and Serialization

Core Abstraction: Scatter-Gather Array. Our serialization library's core abstraction is the *scatter-gather array* abstraction, shown in Listing 1. Scatter-gather arrays point to application data in their original memory location. When applications call `serialize`, the library produces a scatter-gather array that can be passed to the networking stack instead of a single contiguous buffer. Transmitting scatter-gather arrays is conceptually similar to calling the `writev` system call [11] in Linux with an `iovec` data structure, except the Linux kernel still copies the `iovec` into a contiguous buffer before transmission. Section 4.3 discusses research challenges around ensuring application memory can be used for I/O directly.

Serialization API. Our prototype serialization library requires a zero-copy application interface. The generated setter functions store pointers to application memory directly,

```
message Object { optional string msg = 1; }
class ObjectGenerated {
    std::pair<char *, size_t> get_msg();
    void set_msg(const char *addr, size_t len);
    ScatterGatherArray serialize(size_t num_entries);
    void deserialize(const char *payload);
};

ObjectGenerated obj_recv, obj_send;
obj_recv.deserialize(connection.recv());
recved = obj_recv.get_msg();
obj_send.set_msg(recved.0, recved.1);
ScatterGatherArray sga_send = obj_send.serialize();
connection.send(sga_send);
```

Listing 2: **Interface produced by our serialization library in C++, for the listed object schema (in Protobuf syntax), along with example code for an echo server. Unlike prior serialization interfaces, this interface uses *zero-copy* writes and reads. The serialization library avoids copying fields into a pre-allocated buffer and passes a scatter-gather array to the networking stack for transmission.**

rather than moving the memory. Listing 2 shows the interface our library would produce for the simple data structure benchmarked in Section 2.1 and how an echo server could use the interface. However, the library only stores pointers for variable-sized values, such as strings, bytes or nested objects. Maintaining pointers to integer fields would not improve performance (storing the pointer to an integer takes about the same space as storing the integer itself), so the serialize function copies integers into the object header. The header also contains a bitmap to index which fields are present, followed by metadata for each field that is present. For the data structure in Listing 2, the corresponding scatter-gather array points to the object header in the first entry and to the string field in the second entry. The object header contains a bitmap that indexes whether the single field is present or not, and an offset which points to the string field if it is present. The resulting wireformat is similar to Cap'n Proto's wireformat.

Similar to how Cap'n Proto and FlatBuffers are able to support nested objects and lists, our library can support complex data structures. To support a nested field, the object header contains an offset to the nested object's header (if present). To support a list, the header stores the length of the list and an offset to the actual list data. The final scatter-gather array contains the object header in the first entry (including any nested header data), and pointers to string or bytes fields in further entries from the top-level object as well as any nested objects or lists.

Deserialization API. Deserialization requires turning all of the fields in the received buffer back into pointers, to turn the received payload back into a pointer-based data structure.

This requires linearly scanning through all of the possible fields in the object schema, checking if they are present in the bitmap, and recasting all the field offsets into pointers. While linearly scanning through all the fields may add overhead for a data structure with a large number of fields, deserialization could be "lazily" evaluated if the library changed its wire format slightly. Instead of only including field metadata if a field is present in the object header, the header could include header space for *all fields*, so the location of the header information for each field is known ahead at compile time. This way, recasting the pointers for a particular field could be lazily evaluated only when the programmer calls `get_field`.

Zero-copy deserialization involves passing a pointer directly to the application and causes the application to take ownership of data in allocated packet buffers, which the networking stack might need to reclaim later. Additionally, unless the application uses in-place updates when writing data from received packets (e.g., a put request in Redis), the deserialized data might need to be "re-scattered" into specific in-memory data structures, which requires copies. A fully integrated serialization library and networking stack would need to deal with memory safety and reclamation on the deserialization path (§4.4).

3.3 Prototype Implementation

We implemented this approach for the echo server workload for the data structure in Listing 2 in `C++` on top of the Demikernel DPDK stack [40]. We modified the DPDK datapath to produce a linked list of `mbuf` packet data structures given the scatter-gather array. The first `mbuf` contains the packet header with the serialization header copied in. The further `mbuf`s point to the payloads referenced by the scatter-gather array using DPDK's `attach_extbuf` API. To comply with kernel bypass I/O memory requirements, the server directly initializes the data structure payload from pre-registered memory. However, Section 4.3 discusses strategies to ensure application memory addresses can be used for I/O.

The prototype implementation achieves about 8.8 Gbps at 15 µs tail latency. The prototype's performance improves on all the serialization libraries and the 1-copy ("No Serialization") baseline, but falls about 1.6 Gbps short of the optimal DPDK throughput. We speculate this gap comes from inefficient use of scatter-gather entries (allocating an entire `mbuf` for just the packet header and object header). Nonetheless, this prototype shows that leveraging NIC scatter-gather is a promising way to accelerate serialization.

4 Open Research Questions

Many challenges remain in building general-purpose and usable serialization libraries that leverage NIC scatter-gather. This section covers four areas of future work.

4.1 How Does Scatter-Gather Behave Across NICs?

Building a scatter-gather based serialization library requires developing a way to model the performance trade-offs of the scatter-gather functionality. Scatter-gather behavior can vary across NICs as well as device drivers. As discussed in Section 3.1, our NIC adds overhead for many scatter-gather entries and small entry sizes. Eliminating the PCIe interconnect [23] in the NIC, for example, could reduce the cost for additional scatter-gather entries or the penalty for small entries. Building an explicit way to model the trade-offs would let serialization libraries optimize data structure layouts before giving them to the NIC.

4.2 How Do We Use Scatter-Gather Efficiently?

Translating application data structures into scatter-gather arrays that work efficiently with a specific NIC requires solving an optimization problem about memory allocation. Data structures could vary in size (many fields or few fields), shape (different sized fields) and complexity (contain nested objects). To achieve good performance, the serialization library must optimize the memory layout of the scatter-gather array before handing it to the NIC, instead of naively creating one entry per data structure field. This optimization encompasses coalescing some fields into larger buffers and keeping some fields as separate entries, given a maximum number of entries and minimum size each entry should be.

4.3 How Do We Access Application Memory?

A completely zero-copy serialization solution requires using arbitrary application memory for I/O, which raises issues of programming effort and memory fragmentation. Kernel bypass requires that any memory used for I/O lives in pinned and backed pages, because the virtual to physical mappings of this must remain the same during the program lifetime. As a result, pinning an entire application's memory for kernel bypass I/O could lead the OS to allocate large amounts of memory that the application will never use. For memory-intensive datacenter workloads, this could impact the performance of other processes or even the ability for other applications to share infrastructure. Thus, the networking stack or serialization library must have a good sense of what application memory will be used for I/O and must be pinned.

Pinning memory on demand in the networking stack seems promising but would hinder performance on the packet-processing fast path. On-demand pinning would tell the networking stack which data needs to be pinned, but would add the overhead of a system call to packet transmission. Some NICs have additional penalties to consider. Mellanox NICs requires registering memory pages, so the device can do address translation. However, the NIC can only hold a fixed number of address mappings. Fetching a mapping, done when the first address in a newly mapped region is transmitted,

adds a 1 µs latency penalty. If the networking stack registers too many regions, some mappings might fall out of the NIC memory, causing an effect similar to a cache miss.

A new class of *kernel bypass-aware memory allocators* [38, 40] could enable zero-copy dataflows, but raises research challenges related to application integration and memory fragmentation. They could pin large regions of memory beforehand, and allocate "dataplane" memory directly into these regions, while allocating "control" memory into a normal heap. To do this transparently, allocators would need to understand which data needs to be registered with minimal programming effort, perhaps with some sort of compiler-based control flow analysis [4]. To enable multitenancy and minimize interference with other processes, the allocators need to to minimize memory fragmentation and understand how to give up unused memory back to the OS.

4.4 How Do We Ensure Memory Safety?

To make application programming easier, the serialization and networking stack must provide memory safety, in the form of write and free protection. As the Demikernel paper [40] suggests, the memory allocator could provide free protection by adding a reference count to any buffers that are transmitted. However, write protection is more difficult: relying on Linux write protection would add potentially add the overhead of a page fault to kernel bypass applications [10]. The networking stack could adopt techniques from recent work [6] to use cache invalidation to detect when addresses are being overwritten, and accordingly respond. While writing the entire serialization library and networking stack in a memory-safe language like Rust seems like a promising way to enforce write protection, language safety cannot protect against concurrent memory accesses between the NIC hardware and networking stack software. On the deserialization path, the kernel bypass networking stack may need to eventually reclaim application buffers (e.g., buffers where put request values are stored). If the application does not free received buffers in time, the networking stack could run out of memory.

5 Related Work

Serialization Acceleration. Many libraries attempt to improve CPU-based serialization by optimizing their wire format [35, 37], employing SIMD parallelism for decoding [20], or reducing the overhead of type inference in dynamic serialization [19, 22]. These approaches do not remove the fundamental cost required to move memory in software. As a result, recent research proposes offloading serialization to custom accelerators [14, 28] or directly within SSDs for storage [34]. Unlike these accelerators, the scatter-gather functionality already exists in widely used NICs.

Kernel Bypass Systems. Our work is enabled by recent kernel bypass I/O frameworks that expose NIC interfaces directly to applications in userspace [13, 30, 33] to eliminate OS level packet processing overheads. Many recent kernel bypass networking stacks [25, 27, 29, 40] build on top of these interfaces to provide APIs to applications while offering low latency, optimized thread scheduling, or zero-copy I/O. eRPC [16] offers general-purpose RPC for commodity networking hardware, and zero-copy networking. None of these systems directly offer general-purpose, zero-copy, data structure serialization as a programming primitive, which requires scatter-gather.

Scatter-Gather Capabilities. Kesavan, et al. [18] uses scatter-gather to measure when zero-copy helps an in-memory database, but does not consider serialization of arbitrary data structures. Derecho [15], a recent SMR system, uses the scatter-gather offload to provide zero-copy I/O for scattered data structures, but relies on specific layouts of data structures provided by their memory allocator. We propose designing general-purpose serialization for application data in arbitrary memory layouts.

6 Conclusion

As linkspeeds have increased, servers have less cycles to process packets. Object serialization is a core component of datacenter systems, but cannot keep up with modern networks. We identify that CPU-based software serialization is inherently inefficient, as it relies the CPU to perform data movement. We propose using a hardware capability already present in widely deployed NICs to accelerate serialization: the NIC scatter-gather functionality. Our prototype shows that serialization can leverage NIC scatter-gather to offload data movement from the CPU to the NIC for zero-copy and zero-allocation serialization libraries. We identify several areas of future work: using the offload efficiently, providing transparent memory registration, and ensuring memory safety with zero-copy.

7 Acknowledgements

We thank the anonymous HotOS reviewers for their invaluable feedback. We are grateful to Akshay Narayan, Amy Ousterhout, Anirudh Sivaraman, Anuj Kalia, Jacob Nelson, Kostis Kaffes, Qian Li, Shoumik Palkar, and the members of the Stanford Future Data and SING Research groups for their comments on various versions of this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project — Ant Financial, Facebook, Google, and VMware, as well as the NSF under CAREER grant CNS-1651570 and Graduate Research Fellowship grant DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Apache Software Foundation. Hadoop. https://hadoop.apache.org.

[2] Apache Software Foundation. Apache avro. https://avro.apache.org/, 2012.

[3] Apache Sofware Foundation. Apache thrift. https://thrift.apache.org/download, 2017.

[4] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, 2002.

[5] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communicatons of the ACM*, 2017.

[6] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. In *HotOS*, 2019.

[7] lwIP - A Lightweight TCP/IP stack - Summary. https://savannah.nongnu.org/projects/lwip/.

[8] A. Gallatin, J. Chase, and K. Yocum. Trapeze/ip: Tcp/ip at near-gigabit speeds. In *ATC*, 1999.

[9] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.

[10] mprotect(2) - linux manual page. https://man7.org/linux/man-pages/man2/mprotect.2.html.

[11] writev(2) - linux man page. https://linux.die.net/man/2/writev.

[12] gRPC Authors. grpc: A high-performance, open source universal rpc framework. https://grpc.io/.

[13] Storage performance development kit. https://spdk.io/.

[14] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *ISCA*, 2020.

[15] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems*, 2019.

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

[17] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ISCA*, 2015.

[18] A. Kesavan, R. Ricci, and R. Stuntsman. To copy or not to copy: Making in-memory databases fast on modern nics. https://rstutsman.github.io/papers/copy-not-to-copy.pdf.

[19] Kyro. https://github.com/EsotericSoftware/kryo, Accessed January 23, 2021.

[20] G. Langdale and D. Lemire. Parsing gigabytes of json per second. *The VLDB Journal*, 2019.

[21] A. Narayan, A. Panda, M. Alizadeh, H. Balakrishnan, A. Krishnamurthy, and S. Shenker. Bertha: Tunneling through the network api. In *HotNets*, 2020.

[22] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ASPLOS*, 2018.

[23] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out numa. In *ASPLOS*, 2014.

[24] Nvidia. Connectx-5. advanced offload capabilities for the most demanding applications. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.

[25] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[26] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Transactions on Computer Systems*, 2015.

[27] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *OSDI*, 2014.

[28] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch. Optimus prime: Accelerating data transformation in servers. In *ASPLOS*, 2020.

[29] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[30] A rdma protocol specification. http://rdmaconsortium.org/, 2009.

[31] redis labs. Redis. https://redis.io/.

[32] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed sql database. In *SIGMOD*, 2020.

[33] Dpdk: Data plane development kit. https://www.dpdk.org/.

[34] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *ISCA*, 2016.

[35] W. Van Oortmerssen. Flatbuffers: a memory efficient serialization library. https://opensource.googleblog.com/2014/06/flatbuffers-memory-efficient.html, 2014.

[36] K. Varda. Protocol buffers: Google's data interchange form. https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html, 2008.

[37] K. Varda. Cap'n proto. https://capnproto.org/, 2020 (Accessed October 22, 2020).

[38] B. Yi, J. Xia, L. Chen, and K. Chen. Towards zero copy dataflows using rdma. In *SIGCOMM Posters and Demos*, 2017.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[40] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *HotOS*, 2019.