Protocol Buffer Deserialization DPU Offloading in the RPC Datapath

Raphaël Frantz
Eindhoven University of Technology
Eindhoven, The Netherlands
r.r.a.frantz@tue.nl

Jerónimo Sánchez García
Aalborg University
Copenhagen, Denmark
jsg@es.aau.dk

Marcin Copik
ETH Zürich
Zürich, Switzerland
marcin.copik@inf.ethz.ch

Idelfonso Tafur Monroy
Eindhoven University of Technology
Eindhoven, The Netherlands
i.tafur.monroy@tue.nl

Juan José Vegas Olmos
NVIDIA Corporation
Copenhagen, Denmark
juanj@nvidia.com

Gil Bloch
NVIDIA Corporation
Yokne'am Illit, Israel
gil@nvidia.com

Salvatore Di Girolamo
NVIDIA Corporation
Zürich, Switzerland
sdigirolamo@nvidia.com

***Abstract*—In the microservice paradigm, monolithic applications are decomposed into finer-grained modules invoked independently in a data-flow fashion. The different modules communicate through remote procedure calls (RPCs), which constitute a critical component of the infrastructure. To ensure portable passage of RPC metadata, arguments, and return values between different microservices, RPCs involve serialization/deserialization activities, part of the RPC data center tax. We demonstrate how RPC server logic, including serialization/deserialization, can be offloaded to Data Processing Units (DPUs). This effectively reduces the RPC data center tax on the host, where applications' business logic runs. While we focus on offloading Protocol Buffers deserialization used by the popular gRPC framework, our findings can be applied to other RPC infrastructures. Our experimental results demonstrate that RPC offloading performs similarly to traditional methods while significantly reducing CPU usage.**

***Index Terms*—SmartNIC, DPU, deserialization, offload, microservices, RPC, RDMA**

I. Introduction

The development of web services has shifted towards microservice architecture, which provides several key benefits, such as improved scalability, flexibility, and fault isolation. This approach involves running multiple specialized, independent applications—known as *microservices*—rather than relying on a single, monolithic application [1]. These applications are often deployed in containers within virtual machines, running across multiple nodes in data center servers.

To communicate, these microservices use RPCs, a programming paradigm that allows transparently calling a function on an application running on a remote host (RPC server) by wrapping the remote call in a single local function [1]. Networking complexities are hidden from the programmer, speeding up application development. RPC arguments are transmitted from the RPC client to the RPC server, and RPC return value and potential error information are transmitted from the RPC server back to the RPC client, closing the loop. RPCs are always initiated by the client; the server's work is to answer the requests.

As the RPC client and the RPC server can run on different machines, we must serialize this data in a platform-independent way. Local objects are serialized into platform-independent binary objects, or *messages*, transmitted over the network, and the receiver side deserializes the message back to a local object. Multiple serialization formats exist. Google's Protocol Buffer (protobuf) is a popular format used by gRPC [2], a popular RPC framework.

Serialization/deserialization imposes a significant computational burden on the RPC client and the RPC server. Like compression or memory allocation, serialization/deserialization is a frequent operation on the network stack. Together, these operations form the *data center tax* [3]. Studies have shown that deserialization can account for as much as 5% of the total data center tax in the case of Google warehouses scale computers [3]. In another case, serialization/deserialization operations account for up to 50% of the RPC stack [4].

Therefore, optimizing serialization/deserialization is critical to lower the data center tax. Novel methods are used to improve the efficiency of the systems, while traditional CPUs' frequency and efficiency increase slows down [5]. Offloading the computations to specialized hardware is a popular solution for freeing the CPU cycles to run application logic. The main issue with specialized hardware is its lack of flexibility. It cannot be updated to follow the evolution of serialization protocols, which are not standardized or stable over time and can evolve quickly.

In this work, we use Data Processing Units (DPUs) to offload serialization/deserialization, allowing us to retain this flexibility without deploying new hardware with a protocol update. DPUs are network cards that include programmable accelerators. DPUs are designed for networking tasks, and their lightweight, power-efficient, and limited-in-count cores are suitable for data-movement tasks that require a low amount of computing, like serialization/deserialization. In this paper, we use the term CPU to refer to the host's CPU specifically. Even if the DPU technically has a CPU, we use the term DPU to refer to the DPU's central processing unit.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00126

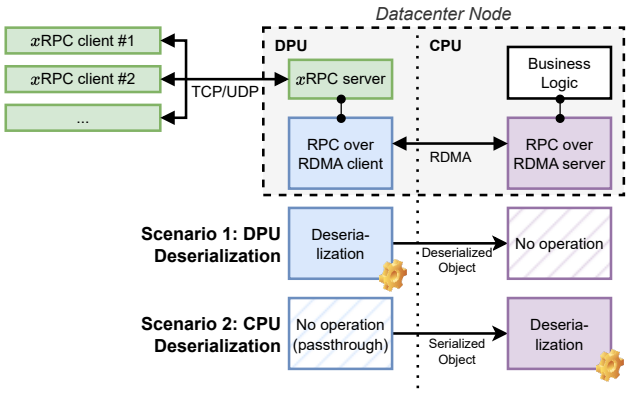


Fig. 1: Architecture for offloading RPCs. We focus on the host/DPU connection in this work. *xRPC* denotes any RPC protocol to offload, which can be, for example, gRPC.

Solutions to accelerate serialization/deserialization can focus on either serialization [6], deserialization, or both [7], [8]. Designing an efficient solution is challenging. The remote function (also called *business logic*) execution should stay on the host to integrate easily into existing applications while offloading serialization/deserialization [7]. Critical points for a qualitative solution, outside of performance, are: how much effort is needed to integrate the solution; if any specific technology is required; and if the solution can apply to any, or a particular serialization format [9] and programming language.

As shown in Figure 1, we focus on offloading the RPC server deserialization from the CPU to the DPU. The entire RPC server, including deserialization, is offloaded to the DPU, while the business logic execution stays on the host. We focus on deserialization only, but serialization can be offloaded with similar techniques. To offload deserialization, we first implement an RPC protocol between the DPU and the host that allows objects to be transmitted in the same address space, removing the need for deserialization on the receiver side, specifically on the RPC server, which we are interested for offloading.

As seen in Figure 1, two different RPC protocols coexist with this design: the original RPC protocol we want to offload and the host-DPU protocol. The DPU receives the RPC calls and converts the original RPC requests to our custom protocol's requests. A compatibility layer can be written to allow the host application to continue using the offloaded RPC library's standard programming interface (API), which we demonstrate by writing one for gRPC.

Then, we demonstrate the functionality of the system by offloading protobuf deserialization. We also implement a simple gRPC server with minimal code modifications thanks to the automatic code generators we write. We then measure the performance of offloading deserialization on NVIDIA's BlueField-3 DPU, which achieves the same requests per second (RPS) rate as without offloading, while freeing up to 7 host cores.

To summarize, the original contributions of the works presented in this paper are: ① Design and implementation of a format-agnostic RDMA-based protocol to offload deserialization; ② Implementation of a layer to offload protobuf deserialization; ③ Benchmarking of the proposed solution on BlueField-3 DPUs. The code for the library and benchmarks is open-sourced on GitLab[1], which includes our custom RDMA-based protocol implementation, and the protobuf deserialization layer.

II. Background

A. Remote Direct Memory Access

Remote direct memory access enables data to be transferred directly to remote memory without involving the processor and bypassing the host's operating system. These features make this protocol ideal for offloading. Programmers can leverage the `libibverbs` library to use RDMA. We leverage RDMA for the host-DPU communication, as shown in Figure 1.

The different operations of RDMA are read, write, write with immediate, send, and receive [10]. We mainly use the write with immediate operation, which permits writing to a specific address and carrying 4 bytes of immediate data. This operation is called two-sided [10] because the remote host is actively notified when the operation is completed.

Users post work requests to *receive queues* to specify where to receive data or to *send queues* to send data to a remote host. They are notified via *completion queues* and *completion channels*. Data are stored in pinned *memory regions*. All RDMA resources are grouped in *protection domains* to help them to work together.

B. Zero-copy Deserialization

As the trade-off between network bandwidth cost and CPU cycle cost over the last years have been reversed in favor of the bandwidth, new high-performance serialization formats (Cap'n Proto, FlatBuffers) are all focusing on a key feature known as *zero-copy* deserialization. This means that the in-memory object and the wire format are the same: *there is no deserialization*. Sending a zero-copy object increases the count of transmitted bytes but reduces CPU cycles used on the receiver side.

A zero-copy object is contained in one (or multiple) position-independent, contiguous memory slices. This format is well-suited for read-only objects, like a server-side received RPC argument. Modifying a zero-copy object has higher memory move costs than standard objects because fields are allocated from a stack (also known as *arena buffer*); thus, freeing or resizing a previously allocated field is difficult or impossible.

While newer formats that offer zero-copy deserialization are increasingly popular, many microservices and applications are still centered around older libraries, like protobuf, which follow the traditional serialization-deserialization workflow. This large legacy codebase makes optimizing deserialization essential for data center performance.

[1]https://gitlab.tue.nl/20233461/sc24_ixpug_paper

C. Data Processing Units Deserialization Offloading

There is no way to bypass one copy if the serialization format does not offer this feature natively. Offloading deserialization to DPUs is still a *one-copy* deserialization because the external hardware transforms data once, with a decoding step. However, from the application's point of view, the message can be directly processed without any CPU intervention. The DPU writes a deserialized message that exactly matches the native hardware architecture of the host, which emulates zero-copy deserialization for any serialization format that does not natively support it. As a use case for DPUs, we test the offloading of deserialization to DPUs and design an architecture suitable for it.

BlueField-3 is a DPU manufactured by NVIDIA. This is a system-on-a-chip SmartNIC with 16× ARMv8.2 A78 Hercules cores and 32 GB onboard DDR5 memory [11]. This DPU can run in one of multiple modes, for example, to mirror the traffic from the host to the DPU cores. We are interested in the "embedded CPU" mode, which allows the installation of a complete operating system (OS) independent from the host. BlueField-3 supports RDMA with the host in one of two modes, Infiniband or RDMA over Converged Ethernet. In practice, the driver will leverage the host's DMA hardware.

III. Offloading RPC with RDMA

In this section, we explain the critical differences between traditional RPCs and offloaded RPCs. RDMA is ideal for offloading thanks to its CPU bypass feature and high throughput. To avoid confusion, we use the term *xRPC* to refer to the original RPC protocol we aim to offload. Then, we use the term *RPC over RDMA* for the new, custom protocol proposed in this paper. Even though we focus on the DPU as the RPC over RDMA client and the host as the RPC over RDMA server and use these terms interchangeably, our work can be run on commodity hardware and other types of DPUs, which permits high compatibility.

The only hardware requirement for using our RPC over RDMA library is an RDMA connection between the RPC over RDMA client and the RPC over RDMA server. Even though our RPC over RDMA implementation can be seen as another RPC protocol and used as such, the central point is to offload as many computations as possible from the RPC over RDMA server to the RPC over RDMA client, which is primarily serialization/deserialization.

A. Requests Lifecycle Overview

With traditional RPCs, the xRPC clients send requests to the xRPC server, which runs on the host. In the case of offloaded RPCs, the DPU acts now as the xRPC server. From the xRPC client's point of view, there is no difference, and no code needs to be changed. The only configuration change is to modify the xRPC server address to reflect the DPU's address instead of the host's. The DPU is a SmartNIC but has a distinct IP address to the host.

Then, the DPU forwards the xRPC request to the host. The xRPC request is transformed to an RPC over RDMA request, which consists of the deserialized binary object. This costly transformation, which essentially consists of allocating the memory for the RPC over the RDMA request and running the deserialization, is entirely run on the DPU.

The host receives the RPC over RDMA request, which can be processed directly, as all the arguments are already deserialized. A compatibility layer mocks the xRPC server on the host and interprets the RPC over RDMA requests as xRPC requests. This layer enables RPC offloading without rewriting the host application. For the response, the host sends the RPC over RDMA response to the DPU. The DPU transforms the RPC over RDMA response to an xRPC response and forwards it to the xRPC client transparently.

As we mainly focus on offloading deserialization, our implementation for protobuf only offloads the request's deserialization and not the response's serialization, but this can be implemented similarly in our design. As shown in Figure 1, the DPU sits in between the host and the xRPC client as a middle-man. Since the DPU now handles all the xRPC client connections and multiplexes them to the host, it can alleviate the burden of managing multiple xRPC sessions and network connections, often TCP/IP.

B. Shared Address Space Advantages

Traditional RPC libraries do not allow sender to learn the address of receive buffer for the message [2], [12]. If the protocol is extended with a shared address space, the serialization could be eliminated. Memory management is seen as an implementation detail, and the location of the messages that arrive in memory is often unpredictable. Knowing the destination address permits deserializing objects without modifying the serialization library. The sender can reconstruct the objects and manually adjust the pointers to reflect the receiver address space.

In addition, sharing this same address space between the client and the server permits us first to remove the need to adjust the memory pointers, accelerating deserialization. That means a request's pointer on the client side `x` will have the value `x` on the server side, and as well a response's pointer on the server side `y` will have the same value `y` on the client side.

If the serialization library allows to deserialize messages into contiguous slices, the standard deserializer can also be used without writing a custom one for offloading. In the case of protobuf, certain limitations of the official implementation force us to write a custom deserializer, which is still greatly simplified due to the shared address space.

Our implementation offers the shared address space as shown in Figure 2. The RPC over RDMA client and the RPC over RDMA server possess receiving buffers (RBufs) and sending buffers (SBufs). Receiving buffers are written from the remote side without using the local CPU by leveraging the RDMA write with immediate operation. Receiving buffers **mirror** the remote sending buffer. All the buffers are stored in contiguous slices for better cache efficiency. Multiple RDMA

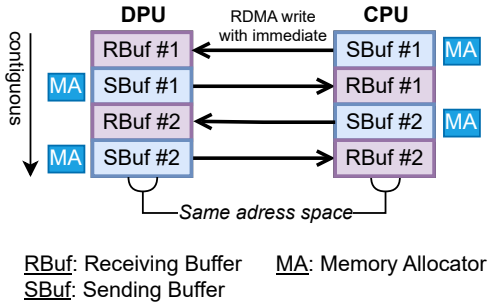


Fig. 2: Shared address space overview.

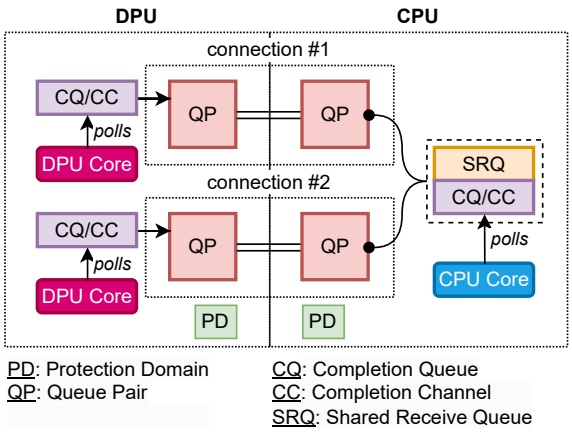


Fig. 3: Threading architecture overview.

connections are run concurrently, each independent, to allow the best performance by preventing data race conditions.

C. Network Architecture and Design

Figure 3 shows the ownership semantics of the RDMA objects and the threading model. Since our goal is to run the RPC over RDMA server on a powerful host and the RPC over RDMA client on a multi-core DPU, there is an imbalance between both sides: the RPC over RDMA client should dedicate more resources per connection. Therefore, a poller is dedicated to a single connection on the client side. Still, a single poller can share multiple connections on the server side using a single received queue and a single completion queue shared between connections.

Busy polling improves the performance up to 10%, at the cost of an unacceptable 100% CPU utilization. Therefore, we use the `poll()` system call to allow the process to sleep under a low-workload scenario. Since the file descriptors count is low, we don't use `epoll()`, adapted in such a scenario.

We use the *many to one to one* model: the DPU multiplexes the xRPC client connections. Even though multiple queue pair connections coexist between the client and the server, they all share the same RDMA device, and the number of connections stays low. These two elements permit the sharing of RDMA resources between the queue pairs, which saves memory and processing cycles on both sides [12].

The RPC over RDMA library does not handle directly xRPC requests dispatching. Existing RPC libraries already distribute requests by providing an interface that can be directly polled in each RPC over RDMA poller thread.

D. Application Programming Interface

We base our API similarly to previous work [12]. Our API lets the threading model to the user's discretion by providing an event loop function that should be called continuously to update the network events. On the RPC over RDMA server side, the user can register RPCs by providing a *callback* function. On the RPC over RDMA client side, the user enqueues requests that trigger a *continuation* function when the response is received.

On the RPC over RDMA server side, RPCs can be executed in one of two ways: foreground or background. Foreground RPCs are directly executed in the polling thread, while background RPCs are executed in background threads. Background RPCs are well-used for long-running RPCs, while foreground RPCs are best used for lightweight procedures or ones that require low latency.

Memory management and tracking information about background RPCs is challenging. Our implementation only supports foreground RPCs, but the protocol complexities are designed to allow background RPCs with little modifications in our code by adding a thread pool. Background RPCs are heavier as they need more information on bookkeeping to be transmitted.

IV. RPC over RDMA Protocol

This section explains in detail the RPC over RDMA wire protocol. We optimize for the most common case: small RPCs. In a previous study, nearly 90% of analyzed messages are 512 bytes or less [8], [13]. Batching is necessary, as a small size is not optimal for an RDMA two-sided operation, as each side generates a packet on the physical layer.

We implement a simple buffering method similar to the Nagle algorithm to aggregate messages. Messages are buffered into blocks, acting as an arena buffer, and the RDMA write with immediate is issued on a complete block.

The user is responsible for queueing enough requests to fill a block before calling the event loop update function. Blocks that contain fewer requests than the limit are still sent when calling the event loop. This prevents high latency when only a few outstanding requests are sent at a time and a deadlock if the block is partially filled. This non-critical case can happen under a low workload scenario. Messages can be larger than the minimum block size; in this case, the block is composed of a single message.

A. Write with Immediate Wire Protocol

Each block starts with a fixed-size *preamble* and contains multiple requests. Each request is formed by a *header* and a

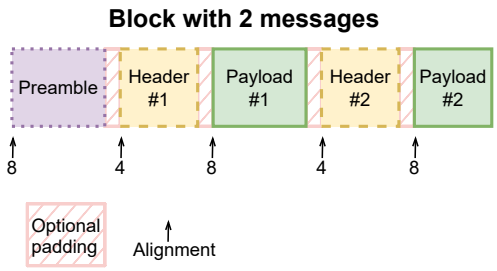


Fig. 4: Layout of the data, called *block*, written to remote memory by the RDMA write with immediate operation.

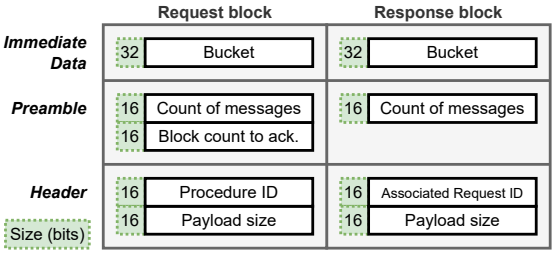


Fig. 5: Protocol metadata: immediate data, preamble, and header.

payload. The byte cost of the preamble is greatly amortized with the large block size.

The RPC over RDMA client/server communication follows this block format in both directions. Only the content of the preamble and the header differ, with different purposes on each side. Unlike the preamble, the header is small because a header will precede each message. The payload contains the data sent by the application.

Blocks are allocated from the sending buffer. Dynamic allocation is needed since RPCs can be completed out-of-order on the server side: a future request can outlive a past one, making dynamic allocation a better solution than standard ring buffers.

Our implementation uses Vulkan® Memory Allocator [14], which permits the allocation of memory by working on a virtual address space and working purely on offsets instead of pointers. Unlike standard allocators that store bookkeeping information before the allocated data, the allocator state is entirely stored externally. This is adapted to manage remote memory.

To allow zero-copy processing, data is properly aligned. For payloads, we set the alignment to 8 bytes, which should be enough for any reasonable class (no 16-byte data type like `long double`, no SSE data type, which are unlikely to be fields of a message class). The preamble and headers are also aligned to allow zero-copy processing of the blocks on the receiving side. The data is stored in little-endian; we assume such architecture is the most common nowadays.

B. Recycling Messages' Memory

The client and the server acknowledge the blocks they receive from the remote side. This permits the recycling of block memory and the release of bookkeeping information. The process is different for the client and the server.

The server implicitly acknowledges the received blocks by simply sending responses. On the client side, when the first response associated with a block is received, this block is implicitly acknowledged.

Since the client's block count does not necessarily equal the server's, the client must also acknowledge the response blocks. We use the RDMA reliable connection to send implicit acknowledgments in a single counter, incremented on each received block. A reliable connection sends and receives data in order. Therefore, the client sends this counter indicating the count of response blocks processed and resets it to zero. This acknowledgment is implicit in the preamble of the next block sent.

C. Managing Congestion

We implement a credit-based system that permits managing congestion. This limits the number of concurrent blocks in processing. Managing congestion is essential, as overflowing the RDMA completion queue or the RDMA receive queue on the receiving side causes data retransmission and massively reduces performance.

Since the count of blocks sent by the client does not necessarily equal the count of blocks sent by the server, the client and the server have separate credits. We use the already-built acknowledgment system for this purpose. Each block sent consumes one credit, and each block acknowledged replenishes one credit. When the credit count reaches zero, no more data is sent until it increases again.

D. Tracking Remote Procedure Calls

A unique ID is associated with each request to retrieve associated metadata for background RPCs. When the client receives the response, the request ID can be recycled. To limit the header size, these IDs are stored on 2 bytes, which allows up to 2^{16} concurrent requests. In addition, the request ID is not sent explicitly to the server. We again take advantage of the reliable connection to keep the IDs synchronized:

1) The client sends a block and flushes all the pending acknowledgments: it first frees the associated request IDs and then allocates the new request IDs.
2) The server receives a block: it does the same process as the client, in the same order.

As a result, a request header does not need to transmit the associated request ID explicitly. The IDs are deterministically allocated from a pool.

E. Messages' Metadata Overview

A bucket contained in the immediate data permits to locate the block in the receiving buffer by knowing its offset, with the simple formula $\texttt{offset} = (\texttt{rbuf} + \texttt{bucket} * \texttt{block\_alignment})$. Blocks are aligned on 1024 bytes to keep

a high range of addressable memory while saving bits in the immediate data. In addition, the optimal block size for high throughput is higher than this alignment, and therefore, the cache performance due to the data locality is not reduced.

Preambles contain the count of messages (maximum 2^{16}) in the block. Headers contain the user-defined payload's size (maximum 2^{16}). This limit can be removed with minor modifications using variable-length encoding to store the payload size. Larger messages are more likely to be computationally expensive, making this cost negligible.

V. Offloading Deserialization

We demonstrate the design by implementing a layer to offload protobuf deserialization. The RPC over RDMA client deserializes the C++ object in place, and the RPC over RDMA server receives an already-built protobuf object. We implement a version compatible with `libstc++`, like in previous work [8], and we support `proto3` domain-specific language. We test specifically on protobuf v5.26, but other versions are supported if the ABI of message classes does not change.

We write a custom deserialization routine. The costly operation in CPU cycles is the varint decoding, the UTF-8 validation for strings, and the recursion for deeply nested messages. The string deserialization is much faster without offloading since x86 SIMD instructions permit processing the Unicode validation very quickly.

A. Binary Compatibility Guarantees

Offloading deserialization on the client from the server poses the problem of binary compatibility, which is a well-known problem for RPCs [15]. Let `T` be a C++ type, and if `T` is a class, let `f` be any field of `T`. `T` is said to be *binary-compatible* between two programs if, for two instances representing the same logical state, the raw byte values of the instances are the same. This statement is true if the ABIs are the same for layouts, sizes, and alignments, which is equivalent to saying that, for any field `f` (recursively if `f` is a class), these expressions should evaluate to the same value on both programs: `sizeof(T)`, `alignof(T)`, and `offsetof(T, f)`.

We base the design of the offloading architecture on the assumption that all fields of the deserialized message are binary-compatible between the client and the server. This assumption holds in practice for our scenario, which is the client on an ARM64 [16] (DPU) and the server on an x86-64 (host), and both ABIs are based on Itanium [17].

We assume a message type to be a simple class containing mostly primitive types, strings, nested messages, and limited bookkeeping information. This makes the client and server instances very likely to be binary-compatible. We also assume the floating point values are represented in the widely used IEEE 754 format. We assume one of the two most widely used compilers, `clang` or `gcc`. In general, they have compatible ABIs [18]. Compiler flags that affect the ABI should, however, be the same.

B. Building an Accelerator Description Table

Similarly to previous work [8], we build an *Accelerator Description Table* (ADT) on the host. The ADT contains all the necessary information to deserialize any protobuf message directly into a C++ object. The ADT consists of a list of metadata for each message type. The metadata of each class includes the default instance, each field offset, and field type, including a pointer to the child table if the field is also an object.

This information is per class rather than per instance, alleviating any per-instance bookkeeping metadata to be transmitted to zero bytes. Therefore, the ADT needs to be transmitted only once. The DPU application does not need to be recompiled and can work with any protobuf object. The ADT is transmitted from the host to the DPU at the start of the application.

When using C++ inheritance like protobuf does, the first bytes of an instance store a pointer to a *vptr*, necessary for proper polymorphism. Setting only the fields is not sufficient since the *vptr* should also be filled for the application not to crash. Storing directly the bytes of the default instances also stores the *vptr*, which can therefore have the correct value.

The ADT is automatically generated by a custom protobuf plugin in `.adt.pb.{h,cc}` files, which contains the ADT to the corresponding `.pb.{h,cc}` files, without any further user intervention. The ADT files are generated when protobuf message definitions are transpiled to C++ files with the `protoc` compiler. Each ADT contains a set of all message classes in a given protobuf definition file, recursively including all nested field message types.

C. Arena-Based Deserialization

The object must be constructed as a contiguous memory slice to achieve high deserializing performance. Studies have been conducted to explore copying fields scattered in memory [9], [19], but this approach has limitations. If the object does not reside in pinned memory—which is most likely the case if the application is not designed from the ground up for offloading—the NIC cannot offload the gathering of the fields, and an additional copy operation is required.

To achieve optimal performance, serialization APIs provide ways to store objects in contiguous slices, also known as arenas. Protobuf offers this functionality, albeit with some limitations. First, strings are stored outside of the arena. This limitation exists because constructing an `std::string` by taking ownership of an existing character array is impossible in portable, safe code. Secondly, the arena stores metadata related to the allocation, increasing the arena's size.

Therefore, we write a custom protobuf deserializer to address these two issues. The first issue can be resolved if we forgo portability and focus on a specific standard library implementation. Fortunately, most Linux programs are based on libstdc++, allowing us to maintain high portability within this operating system. Still, the same method can support libc++, which has a comparable implementation for `std::string`. The source code can be portable, but knowing which standard library is used at runtime by the host from the DPU cannot be

```
class std::string {
    char* data;
    size_t size;
    union {
        char sso[16];
        size_t capacity;
    };
};
```

Fig. 6: `std::string` layout of libstdc++.

done unless this information is explicitly transferred from the host to the DPU, which can then choose the `std::string` layout to use for deserialization.

Strings are byte containers composed of a pointer to the data, a capacity, and a size. If strings are small enough, they are stored directly in the instance without memory allocation, a technique known as small-string optimization (SSO). Both standard libraries feature this optimization but have differences in the implementation. We show the layout for libstdc++ in Figure 6. If the pointer to the data is equal to the SSO buffer, no dynamic allocation is performed, storing at most 15 characters. The implementations is slightly more complicated for libc++, storing an SSO flag in the first bit of the capacity field. Once the SSO subtlety is addressed, crafting zero-copy `std::string` instances becomes straightforward.

D. Compatibility Layer for gRPC

We write a compatibility layer for gRPC C++ on top of the RDMA over RPC protocol for unary calls, using our arena-based protobuf deserialization algorithm. The DPU executes the gRPC server, and the business logic is kept inside the gRPC services on the host side. Existing compiled services, such as those in dynamic libraries, do not need recompilation, and application code needs minimal modifications to be adapted for offloading.

On the DPU side, each thread listens asynchronously to the gRPC API calls. When intercepted, the request is deserialized and triggers the corresponding RPC over RDMA procedure. On the host side, received requests are forwarded to the user-defined service callback handlers. Our custom protobuf plugin automatically generates introspection code to allow the inspection of gRPC service classes, such as mapping procedure IDs to the service's callback function. For the gRPC context, we use a null pointer for simplicity, but metadata can also be passed along with the message in the payload.

VI. Experimental Results

In this section, we focus on the protobuf deserialization implementation. We compare how the CPU and DPU perform in deserialization and in the RPC over RDMA datapath. We compare different metrics in the RPC over RDMA dapath: the average requests per second, the PCIe bandwidth usage and host CPU utilization. We show that when offloading deserialization to the DPU, the host's CPU usage is substantially reduced at the cost of higher bandwidth usage while keeping similar performance regarding request throughput.

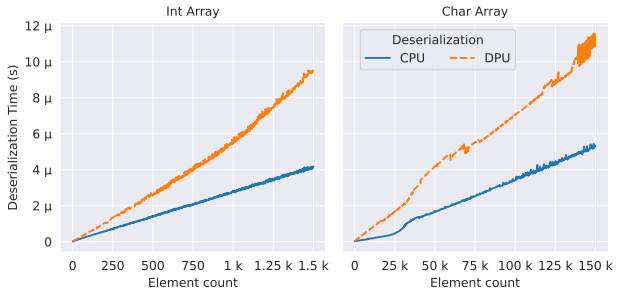


Fig. 7: Time to deserialize a single message composed of an array of elements, relative to the element count. We measure for an int array and a char array, run on the CPU and the DPU.

	Client	Server
Hardware	BlueField®-3	PowerEdge R760
CPU	Cortex-A78AE	x2 Intel® Xeon® Gold 6430
Cores	x16	x64
RAM	30 GiB	251 GiB
Cache sizes		
`L1d`	1 MiB	4 MiB
`L1i`	1 MiB	2 MiB
`L2`	8 MiB	128 MiB
`L3`	16 MiB	120 MiB
Compiler	`gcc -O3 -flto -march=native`	
OS	Ubuntu 22.04	
System Allocator	TCMalloc 4.2	
Configuration Parameters		
Threads	16	8
Credits	256	256
Block Size	8 KiB	8 KiB
Concurrency	1024	n/a
Buffer Sizes	3 MiB	16 MiB

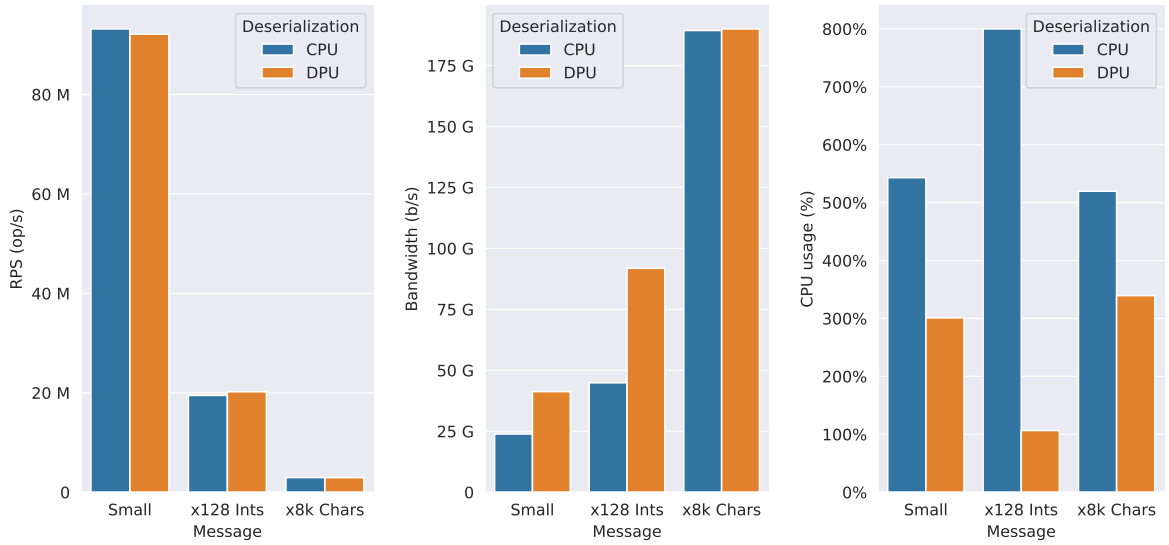
TABLE I: Environment and configuration parameters of the client and the server applications.

A. Environment

Table I shows the hardware and software environment. Our server is a PowerEdge R760 equipped with a BlueField-3 DPU. For the software environment, we run Ubuntu. Because of the parallelism, we use TCMalloc to minimize the thread contention when allocating memory. This has been shown to achieve a 15% increase in throughput in the tests. Additionally, using link-time optimization with `-flto` has provided a further 10% boost in performance, probably due to the aggressive inlining in the deserialization algorithm, consisting of numerous small specialized functions.

We fix the different configuration parameters as shown in Table I (unless expressly stated otherwise). Most configuration parameters are set equally on the DPU and CPU sides. We use the configuration parameters that give better performance after the first preliminary tests with small 15-byte messages to benchmark the RPC over RDMA architecture itself. We set the count of CPU threads to eight, assuming that two DPU cores can match one CPU core, with the DPU having sixteen cores.

The optimal minimal block size for the highest throughput



(a) Achieved requests per second. (b) Bandwidth usage. (c) CPU usage.

Fig. 8: RPC datapath metrics results, comparing the DPU and the CPU deserialization.

is around 8 KiB. The buffer size parameter corresponds to the size of each receiving and sending buffer for each connection. For the count of concurrent requests (concurrency), the initial credits should be high enough to have true concurrency, which means credits $> \frac{\text{concurrency} \times \text{blocksize}}{\text{msgsize}}$ (ignoring the small overhead of the preamble and message headers), but not too high to preserve data locality and caching efficiency. The credits should also never reach zero. This is always true for the experimentation presented here. The credits are per connection, and the concurrency is also per connection.

To ensure a consistent comparison, both the offloaded and the non-offloaded deserialization scenarios use our custom stack-based protobuf deserialization algorithm. Google's deserialization algorithm is highly optimized and generally performs slightly better, but it depends on the message type.

B. CPU vs. DPU Deserialization Time

In Figure 7, we show the time to deserialize a single message on a single core, with *nanobench* [20], a benchmarking framework, without considering the RPC datapath. In all cases, median absolute percentage errors are less than 0.01%, showing that the benchmarks are very stable.

We benchmark two types of messages, an *int array* and a *char array* (or string), comparing the deserialization time to the number of elements in the array. The int array elements are random-generated, unsigned 32-bit integers stored between 1 and 5 bytes. The pseudorandom number generator is a Mersenne twister with a constant seed for reproducibility. The integer distribution we choose is not uniform: integers are more likely to be smaller, thus being stored on fewer bytes. Data access is then unaligned, and different instruction paths are run. The char array is not compressed; each element always takes one byte. Given that both element counts are equal, the deserialization time is significantly faster for the char array than the int array.

When the element count is high, the time to deserialize is linear to the time for both messages, with approximately 2.75 ns per element for the int array and 42.5 ns per 1024 elements for the char array on the CPU. The results presented in Figure 7 fluctuate more, especially in the case of the DPU string deserialization, because we show a more realistic low count of elements. The CPU is faster to deserialize, with the DPU taking, on average, 1.89× more time to deserialize for the int array and 2.51× more time to deserialize for the char array. This is expected, and these preliminary results show that around two DPU cores can replace one CPU core in deserialization to keep similar performance, specifically showing a better affinity in the varint decoding scenario.

C. RPC Datapath

We compare the performance of the offloaded DPU deserialization scenario to the traditional CPU deserialization scenario, as shown in Figure 1. The business logic is left empty to measure the impact of deserialization offloading. For the DPU deserialization scenario, the CPU workload is minimal. It only manages the RDMA connection, and the server responds with an empty message. We measure the performance of four metrics in the RPC datapath, aggregated over all cores. Per-core results show an even workload distribution between the cores, and maximum performance is reached on sixteen DPU threads. The different metrics are:

- Average requests per second;
- Average bandwidth consumed by RDMA via the PCIe bus;
- Average CPU usage, regarding cores used by the RPC over RDMA server application.

The RPC over RDMA library is directly instrumentalized at the library level with a Prometheus [21] client that gathers the various metrics for a small fraction of the performance cost (around 5%). This permits the gathering of statistics independently of the scenario or application. The metrics are sent to a monitoring server. A monitoring process gathers the result after a fixed amount of time (around 10s) and will wait until the RPS rate is stable (within 1%), which takes around 20 seconds, before collecting the final results. We look at the last two data points of each metric to obtain the per-second increase rate, also known as the instant rate of increase.

1) Message: Crafting messages that represent an actual workload is challenging because of the heterogeneity of applications. Google has built a suite to benchmark protobuf serialization and deserialization by providing synthetic messages that reflect their specific workloads [8], which are huge messages with deeply nested structures.

The bottleneck of deserialization can be either memory I/O or CPU core speed. In the context of RPCs, network I/O can also be a limiting factor. Some messages, like hierarchical and compressed data, have a high computational cost, whereas other message types, like non-compressed byte arrays, have a high copy cost. In the latter case, most of the deserialization work consists of a single memory copy from the serialized message, like a network packet, to the deserialized object residing in the heap.

Therefore, we decide to build and benchmark three synthetic messages, each reflecting a different aspect of RPCs:

- **Small**: A small 15-byte message of various fields representing the most common message type. Here, the bottleneck is the capacity of the NIC and the network infrastructure to handle a large amount of messages. This type of message is best for measuring the efficiency of the RPC over RDMA implementation.
- **x512 Ints**: A 32-bit unsigned integer array of 512 elements representing a high computational cost since varint elements should be decompressed.
- **x8000 Chars**: A string of 8000 random characters representing a high copy cost. This message represents data such as requested text files for web services.

The x512 Ints and x8000 Chars messages are the same as shown in Figure 7 for the given count of elements.

2) Requests per Second Rate: Figure 8a shows average requests per second rates. The performance ratio of 1:2 between a DPU core and a CPU core measured in the deserialization benchmarks stays in the RPC datapath. The DPU can match the host's performance when allocating twice as many cores for deserialization as the host. The small message scenario reaches 9×10^7 processed requests per second.

3) Bandwidth: Figure 8b shows average bandwidth utilization. The cost of offloading deserialization is the increased quantity of data sent via PCIe, because deserialized objects take up more space. The serialized x512 Ints message is compressed by the varint encoding by a 2.06× factor by the protobuf format, with a serialized size of only 276 bytes. Small messages are also highly compressed due to the fixed-size C++ instance storing all fields (including unset fields) plus a minimal internal state, which is, in the case of protobuf, a bitfield storing field presence. As a high-compression example, the serialized small message takes 15 bytes on the wire, while the deserialized object size is 40 bytes. It is worth noting that the bandwidth usage in the RPC over RDMA datapath does not precisely reflect this ratio due to the payload's header and protocol alignments, which are non-negligible for small messages. Contrarily, the serialized x8000 Chars message is only compressed by a 1.01× factor, with a serialized size of 8003 bytes. In this case, the bandwidth usage is very similar between deserialization offloading and no offloading and goes up to 180 Gbps.

4) CPU Usage: Host CPU usages are shown are Figure 8c. The CPU usage is significantly reduced with DPU deserialization offloading by a factor of 1.8× for small messages, by a factor of 8.0× for x128 int messages, where the DPU shows the best affinity. Seven host cores are freed. The Unicode validation and data movement offloading permits to reduce the CPU usage by a factor of 1.53× in the case of the x8000 Chars message.

5) Last-level Cache Misses: There are almost zero last-level cache (L3) misses in all cases. This can be explained by the fact that practically all memory writes happen in the pinned memory buffers, with no use of the system allocator in the RPC datapath. We still use dynamic allocation in the user space by working exclusively in our preallocated address space. Additionally, the message types are always the same. This would also be the case in a real-world scenario: like in an object-oriented program, the count of classes is bounded and low, in contrast to the count of instances, which is unbounded.

VII. Related Work

eRPC [12] is a high-performance RPC library that supports Infiniband and UDP by leveraging DPDK. The remote receiving address is not exposed to the user. CPU cycle counter `RDTSC` instruction is used to have low overhead and precise timings to manage traffic congestion. While it is possible to replace it with `PMCCNTR_EL0` register read on ARM, a minimum frequency of 500 Mhz is required by eRPC, which the ARM performance counter does not reach.

Not all hardware is suitable for offloading deserialization. Graphics Processing Units (GPUs) are accelerators with low frequency but high parallelism that now drive the industry for machine learning but are challenging to use for serialization/deserialization [7]. GPUs work on *Same Instruction Multiple Data* (SIMD) parallel model. Still, messages are of various types, sizes, and field orders, which makes it hard to execute the same instructions for different messages unless the messages are the same, resulting in not utilizing the parallel cores to the fullest extent. Field-Programmable Gate Arrays (FPGAs) can be used to run specialized algorithms. Some effort has been made to provide hardware acceleration for deserialization on FPGAs [8] or specialized accelerators [22] at the expense of flexibility by not making the solution available to commodity hardware.

VIII. Conclusion

Hardware accelerators are challenging to use in deserialization due to the heterogeneity of systems, serialization formats, and their rapid evolution. Thanks to their programmability, DPUs are more flexible than accelerators and thus we can adapt them for offloading CPU-intensive parts of RPC stack. We design a format-agnostic RPC protocol that allows deserialization to be offloaded to the DPU, and we implement a thin layer for protobuf. We demonstrate that BlueField-3 DPUs are powerful enough to support protobuf deserialization in the RPC datapath and match the host's performance. By replacing one x86 CPU core with two ARM DPU cores, we match the performance of protobuf deserialization. In the case of the varint decoding scenario, we free up to seven x86 CPU cores to run application logic. Thus, we show that deserialization can be offloaded while keeping software flexibility and without sacrificing performance.

Acknowledgment

This work was partly funded by the QUARC project by the European Union Horizon Europe research and innovation program within the framework of Marie Skłodowska-Curie Actions with grant number 101073355, and by the grant PID2021-123041OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe".

References

[1] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[2] Google, "grpc: A high performance, open source universal rpc framework," https://grpc.io/, 2024, accessed: 2024-05-28.

[3] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 158–169. [Online]. Available: https://doi.org/10.1145/2749469.2750392

[4] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the rpc tax in datacenters," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 407–420. [Online]. Available: https://doi.org/10.1145/3466752.3480055

[5] R. S. Williams, "What's next? [the end of moore's law]," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 7–13, 2017.

[6] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: towards zero-copy serialization," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 206–212. [Online]. Available: https://doi.org/10.1145/3458336.3465283

[7] S. Cao, S. Di Girolamo, and T. Hoefler, "Accelerating data serialization/deserialization protocols with in-network compute," in *2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI)*, 2022, pp. 22–30.

[8] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for protocol buffers," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 462–478. [Online]. Available: https://doi.org/10.1145/3466752.3480051

[9] D. Raghavan, S. Ravi, G. Yuan, P. Thaker, S. Srivastava, M. Murray, P. H. Penna, A. Ousterhout, P. Levis, M. Zaharia, and I. Zhang, "Cornflakes: Zero-copy serialization for microsecond-scale networking," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 200–215. [Online]. Available: https://doi.org/10.1145/3600006.3613137

[10] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 437–450. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[11] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20.

[12] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be general and fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/kalia

[13] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2020, p. 733–750.

[14] A. GPUOpen, "Vulkan® memory allocator," https://gpuopen.com/vulkan-memory-allocator/, 2024, accessed: 2024-05-28.

[15] A. S. Tanenbaum and R. van Renesse, "A critique of the remote procedure call paradigm," 1988. [Online]. Available: https://api.semanticscholar.org/CorpusID:16132591

[16] ARM, "C++ application binary interface standard for the arm® 64-bit architecture," https://github.com/ARM-software/abi-aa/blob/2982a9f3b512a5bfdc9e3fea5d3b298f9165c36b/cppabi64/cppabi64.rst, 2023, accessed: 2024-07-15.

[17] I. C. ABI, "Itanium c++ abi," http://itanium-cxx-abi.github.io/cxx-abi/abi.html, 2024, accessed: 2024-07-15.

[18] LLVM, "clang: Language compatibility," https://clang.llvm.org/compatibility.html, 2023, accessed: 2024-07-15.

[19] F. Lu, X. Wei, Z. Huang, R. Chen, M. Wu, and H. Chen, "Serialization/deserialization-free state transfer in serverless workflows," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 132–147. [Online]. Available: https://doi.org/10.1145/3627703.3629568

[20] M. Leitner-Ankerl, "ankerl::nanobench is a platform independent microbenchmarking library for c++11/14/17/20." https://nanobench.ankerl.com/, 2024, accessed: 2024-05-28.

[21] C. N. C. Foundation, "Prometheus - monitoring system & time series database," https://prometheus.io/, 2024, accessed: 2024-07-15.

[22] 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1203–1216. [Online]. Available: https://doi.org/10.1145/3373376.3378501