Accelerating Data Serialization/Deserialization Protocols with In-Network Compute

Shiyi Cao
Department of Computer Science
ETH Zurich
Switzerland
shicao@student.ethz.ch

Salvatore Di Girolamo
Department of Computer Science
ETH Zurich
Switzerland
salvatore.digirolamo@inf.ethz.ch

Torsten Hoefler
Department of Computer Science
ETH Zurich
Switzerland
htor@inf.ethz.ch

***Abstract*—Efficient data communication is a major goal for scalable and cost-effective use of datacenter and HPC system resources. To let applications communicate efficiently, exchanged data must be serialized at the source and deserialized at the destination. The serialization/deserialization process enables exchanging data in a language- and machine-independent format. However, serialization/deserialization overheads can negatively impact application performance. For example, a server within a microservice framework must deserialize all incoming requests before invoking the respective microservices. We show how data deserialization can be offloaded to fully programmable SmartNICs and performed on the data path, on a per-packet basis. This solution avoids intermediate memory copies, enabling on-the-fly deserialization. We showcase our approach by offloading Google Protocol Buffers, a widely used framework to serialize/deserialize data. Our evaluation demonstrates that, by offloading data deserialization to the NIC, we can achieve up to 4.8x higher throughput than a single AMD Ryzen 7 CPU. We then show through microservice throughput modeling how we can improve the overall throughput by pipelining the deserialization and actual application activities with PsPIN.**

***Index Terms*—SmartNICs, sPIN, deserialization, offload, RPC, microservices**

I. Introduction

Data exchange between different applications or between different processes of the same application is a fundamental activity in any data center and HPC system. Different applications or processes can store data into different layouts that depend on the application itself (e.g., class definition), the language (e.g., row-major vs column-major), and the architecture (e.g., little vs big endianness). Data serialization/deserialization (SerDes) protocols abstract these details, allowing applications to easily exchange data in an architecture- and language-independent manner.

These protocols are becoming a central part of the software stack in data center and HPC systems. For example, the emerging microservice paradigm, where services are decomposed into multiple fine-grain tasks (i.e., microservices), heavily relies on remote procedure calls (RPC). RPC requests are sent by heterogeneous clients over the Internet or from other actors within the microservice framework. These requests are encoded at the source and decoded at the target microservice, to which they are dispatched, by using SerDes protocols.

Persistent data storage is another SerDes use case. Space-efficient schemes for long-term data storage, e.g., Apache Parquet [1], require serializing the data in a specific format before storing it. Data is then deserialized whenever it is retrieved from the storage.

However, SerDes activities come at a cost, the time overhead required to serialize and deserialize data. The continued advancement of interconnects (e.g., 400 Gbit/s networks are available on the market [2]) exacerbates these overheads, which exposes software bottlenecks that were before hidden behind network overheads. It has been shown that the SerDes overheads can be up to 50% of microservice runtime [3] and up to 5% of the total *datacenter tax* [4].

To accelerate SerDes activities, different hardware accelerators have been proposed [5]–[7]. They can be either co-processor of the main host CPU [5] or deployed directly on the data path [3] and encode specific SerDes protocols, such as Google Protocol Buffer (protobuf) [8] and Apache Thrift [9], in hardware. However, these approaches are not flexible and require additional hardware deployment iterations for extending protocols or implementing new ones.

In this work, we investigate how in-network compute can accelerate SerDes activities by processing data on a per-packet basis. Our approach targets fully-programmable SmartNICs, where applications can express processing tasks to be executed on the data-path. In particular, we focus on data deserialization as this activity can easily become a bottleneck on the server side (e.g., deserializing RPC requests in a microservice framework) and hinder non-overlappable read operations (e.g., deserializing data read from the storage). *Our solution, not only frees CPU resources by delegating data deserialization to the NIC, but also accelerates it by exploiting the multiple processing elements on the NIC and avoiding intermediate memory copies.*

Overall, our contributions are the following:

- We demonstrate how data deserialization can be offloaded to fully-programmable SmartNICs, addressing the challenge of transferring and manipulating complex pointer-based data structures between different memory spaces.
- We show how this approach can free up CPU cycles by pipelining the deserialization and the execution of the main application activities (e.g., microservices).

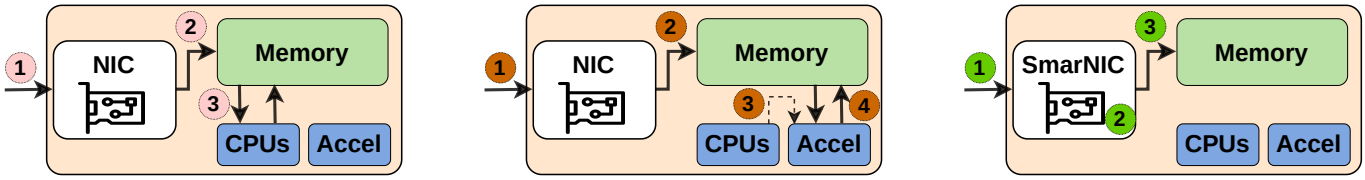


Fig. 1. Data deserialization strategies. (a) NIC writes serialized data into main memory and the host CPU performs the deserialization; (b) NIC writes serialized data into main memory and the host CPU instructs an hardware accelerator to perform the deserialization; (c) data deserialization is performed on-the-fly, as data flows in from the network and written directly to the right memory locations.

- We identify and discuss SmartNIC specialties that are needed to optimize SerDes activities.

As our approach is fully expressed in software, it can be easily extended to support new protocol versions or variations. While we focus our analysis on protobuf, we envision how the outlined approach can be used as a blueprint to offload and accelerate other SerDes protocols.

II. SerDes and In-network Compute

By abstracting language- and machine-dependent details away, SerDes protocols ease data exchange between different environments and architectures. While there exist a large number of SerDes protocols [8]–[12], in this work we focus on Google Protocol Buffers because of their widespread usage in datacenters [4] and HPC systems. Moreover, we focus on the deserialization task as this can easily become a communication bottleneck in, e.g., incast-like scenarios, where a server node needs to deserialize requests issued by its clients.

Fig 1 sketches an overview of possible data deserialization strategies: classical CPU-based ones, strategies deploying specialized hardware accelerators, and the one presented in this work, where data is deserialized on the NIC.

Fig 1(a) shows the case of CPU-based deserialization. Serialized requests are received by the NIC (1), and written to host memory (2), either passing packets to the operating system network stack or using remote direct memory access (RDMA). The host CPU will read the serialized data, deserialize it, and write the result to the main memory (3).

Fig 1(b) shows the case where deserialization is offloaded to specialized co-processor of the host CPU. Incoming data (1) is written to the host memory (2). After that, the host CPU instructs a specialized hardware accelerator to perform the deserialization (3). In this case, the hardware accelerator reads data from the host memory, deserializes it, and writes the result back (4).

Finally, Fig. 1(c) shows our approach. Incoming data (1) is processed directly on the NIC, where the data deserialization process is executed (2). The deserialized data is written directly to host memory (3), without intermediate copies or host CPU intervention.

A. Google Protocol Buffers

In Protocol Buffers, messages are described by *message descriptors*. These descriptors, which define the message layout, identify the fields composing a message, together with their type and cardinality. A *message* is an instance of a *message descriptor*. A message descriptor can mark a field as *repeated* and *embed* other message descriptors.

A message descriptor does not fully define a message but allows one to deserialize it. For example, while a field can be marked as *repeated*, the actual number of repetitions is not specified and is directly encoded in the data stream. This makes Protocol Buffers different from MPI Derived Datatypes [13], where a datatype fully specifies a message.

Listing 1 shows an example of message descriptors. The message descriptor `Student` defines a message carrying student information, such as the name, the id, and one or more advisors (embedded message). Each field is identified by an optional field rule (i.e., *repeated*, *required*, or *optional*), a type (i.e., `wire_type`), a name, and a field number.

```
message Student {
  uint64_t id = 1;
  string name = 2;
  repeated Advisor advisors = 3;
}

message Advisor {
  uint64_t id = 1;
  string name = 2;
}
```

Listing 1. Protobuf Message Definition

During serialization, messages are encoded into a byte stream, with each field consisting of a key/value pair. The key encodes field number and type (i.e., `key` = `field_number` $\ll 3 \mid$ `wire_type`). Fig. 2 shows an example of a variable-length integer type (*varint*) and a length-delimited one (e.g., a string). The field type determines the format of the following bytes. For example, length-delimited fields encode the length immediately after the key.

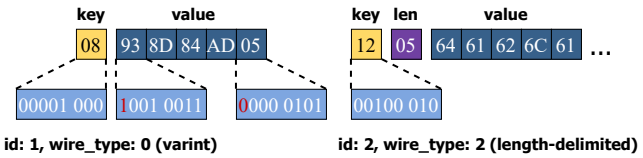


Fig. 2. An Example Encoded Data Stream.

1) Varint encoding: Variable-length integers are not only used for encoding integer values but also the field metadata (e.g., field numbers, length of length-delimited fields). *Varints* cannot be directly used as integers but need to be deserialized. The most significant bit of each field of a *varint* indicates whether it is the last byte of the integer or not. The remaining bits are part of the actual value of the integer. Therefore, to decode an `varint`, we need to check for every byte whether this is the last byte and do bit-wise operations to calculate the original integer value. Listing 2 shows the pseudo-code for decoding a `uint32_t`. Such chain-style encoding/decoding results in data-dependent branch per byte that can limit the transformation throughput [7].

```
unsigned max_rv = len > 5 ? 5 : len;
for (rv = 1; rv < max_rv; rv++) {
  if (data[rv] & 0x80) {
    tag |= (data[rv] & 0x7f) << shift;
    shift += 7;
  } else {
    tag |= data[rv] << shift;
    *tag_out = tag;
    return rv + 1;
  }
}
```

Listing 2. Pseudo-code for Varint Decoding

2) Sequential processing: Decoding protobuf data is a sequential process. In fact, as fields metadata and data are encoded together, it is not possible to know the starting position of the next field without processing the current one (e.g., scanning all bytes of a *varint* or decoding the length of a length-delimited one), making parallelization challenging.

B. Network acceleration

Our goal is to investigate the benefits of offloading data deserialization to the NIC. By offloading this task, we expect to alleviate CPU load, which can be re-invested in running, e.g., microservices functional code, and to eliminate intermediate memory copies, which can improve system performance due to reduced congestion.

To make deserialization-offloading effective, our solution should: be available to user-level applications; and be flexible in terms of expressing the offloaded tasks.

User-level. User-level applications should be able to offload tasks without requiring elevated privileges. For example, a user-level application should be able to define tasks to be offloaded to the network without violating isolation principles (e.g., getting access to network data targeting other applications or users). This requirement also enables applications to express different SerDes protocols, without the need to stick to a limited set of them for which, e.g., system administrators made offloading-support available. This requirement is not satisfied by in-network compute solutions like DPDK [14] and Portals 4 [15]), where tasks are defined at the system-level and not at an application-level.

Flexibility and High Programmability. Algorithms for deserializing data can be complex and they are dependent on the specific SerDes protocol. In-network compute solutions like eBPF/XDP [16] and Portals 4 [15] impose constraints on the code describing the offloaded task (e.g., bounded loops) or allow the offloading of specific operations, making them not flexible.

An in-network compute solution that satisfies these requirements is sPIN (streaming Processing in the network) [17]. With sPIN, user-level applications can express per-packet tasks to be offloaded to the NIC. These tasks can be expressed in C/C++ or any other languages that the specific sPIN implementation has compiler support for. We now provide a brief overview of sPIN.

1) sPIN: streaming Processing In the Network: In sPIN, applications define packet handlers to be executed on the NIC. These handlers are associated with classes messages and are triggered and executed as packets arrive from the network. An application defines an execution context that includes information such as: pointers to the packet handler functions in NIC memory; NIC memory previously allocated by the application and used by the handler to share state). This execution context is then associated with a message (e.g., an MPI message or a Portals 4 [15] match list entry) or packet flow (e.g., socket). The NIC is in charge of matching packets to execution contexts, and then triggering the respective packet handlers.

In this paper, we use PsPIN [18], an implementation of sPIN based on the PULP framework [19]. PsPIN provides 32 RISC-V cores (32 bit, in-order) where sPIN packet handlers are scheduled and executed. The PsPIN architecture is modular and composed by four computing clusters (each with 8 cores). Each cluster is equipped with a 1 MiB fast scratchpad memory (L1). An off-cluster memory, which is slower (i.e., 20 ns average access latency) but bigger (4 MiB), is available.

III. Network Accelerated SerDes

A. Overview

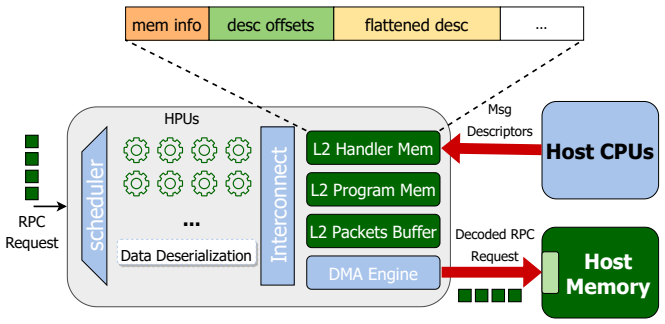


Fig. 3. Overview of Offloading Data Deserialization to sPIN.

To enable efficient distributed communication frameworks, we want to offload data deserialization to sPIN. Data (de)serialization frameworks such as Protobuf are not self-describing, which means to decode a message we will also need the corresponding message and field descriptors. The NIC driver will first flatten the needed *Protobuf Message*

Descriptors for supported RPC methods and write them to the NIC's L2 Handler Memory with an array indicating the offset of each descriptor. Packets are then received and deserialized by the HPUs in the NIC using the descriptor indicated by the RPC method type. The decoded contents of the message will then be written directly to the host memory through the DMA engine. Fig. 3 illustrates an overview of the workflow.

B. Offloading Message Descriptors

The first thing for the handler to deserialize the coming messages is that it should have the corresponding *Protobuf Message Descriptors* for the messages it wants to decode. Typically the *Protobuf Message Descriptors* contains pointers to the *Protobuf Field Descriptors* which may point to other *Protobuf Message Descriptors* for embedded messages, resulting in hierarchical referencing structures, as shown in Listing 3 and Listing. 4. Therefore, to use these descriptors correctly in the NIC, we need to flatten these descriptors recursively and change all the pointers to point to the NIC's memory address.

```
struct ProtobufCMessageDescriptor {
  uint32_t        magic;
  const char      *name;
  const char      *short_name;
  const char      *c_name;
  const char      *package_name;
  uint32_t        sizeof_message;
  unsigned        n_fields;
  /* Field descriptors sorted by tag number.
    */
  const ProtobufCFieldDescriptor  *fields;
  /* Used for looking up fields by name. */
  const unsigned      *fields_sorted_by_name;
  /* Number of elements in 'field_ranges'. */
  unsigned      n_field_ranges;
  /* Used for looking up fields by id. */
  const ProtobufCIntRange   *field_ranges;
    ...
}__attribute__((packed));
```

Listing 3. Protobuf Message Descriptors

```
struct ProtobufCFieldDescriptor {
  const char     *name;
  uint32_t    id;
  ProtobufCLabel    label;
  ProtobufCType   type;
  unsigned    quantifier_offset;
  unsigned offset;
  /* for MESSAGE and ENUM types */
  const void     *descriptor;
  ...

}__attribute__((packed));
```

Listing 4. Protobuf Field Descriptors

Fig. 4 demonstrates the simplified buffer layout after depth-first recursive flattening of the hierarchical referencing descriptor structures in the driver. Note that after flattening, all the pointers are now referencing the NIC's memory space. Then the driver can write the flattened descriptors to the NIC's L2 handler memory with the corresponding message type offset.

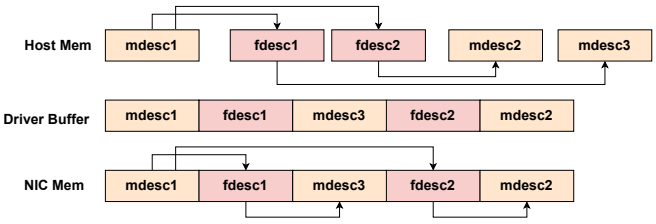


Fig. 4. Descriptors Flattening.

C. Message Handling Workflow

Profiling results from previous paper [5], [20], [21] show that most of the RPC messages are 512 bytes or less, which makes the per-packet processing nature of sPIN even more beneficial as now we can always process many messages in parallel on the NIC. Below we use RPC requests for all the examples as they are representative workloads. Here we assume the RPC message packet format as shown in Fig. 5. Now when an RPC message arrives, the handler will start deserializing the encoded payload using the message descriptors indicated by the RPC type of the message.

To decode, we follow the original protobuf decoding logic while replacing the memory allocation part for storing some of the decoded data (e.g., strings) by allocating some memory on the pinned host memory and writing the decoded data directly to the host. On the NIC, we only store the pointers referencing these data on the host memory (i.e., the top-level ProtobufCMessage structures). These top-level ProtobufCMessage structures will be written to the host memory after all its fields' data have been decoded. The high-level idea is shown in Listing 5. The detailed decoding process and related memory management on the NIC and host are illustrated in Sec. IV.

Fig. 5. RPC message packet format.

```
__handler__ void protobuf_ph(handler_args_t *
    args){
   /* Get the packet and descriptor info
      ...
   */
   ProtobufCMessage* msg = (ProtobufCMessage*)
     (task->scratchpad[args->cluster_id] +
    MEM_RESERVED + args->hpu_id*MEM_PER_HPU);
   ScannedMember sm;
   protobuf_c_message_unpack_all(gdesc, msg,
    len, pkt_pld_ptr, &sm);

   uint32_t host_low_addr = do_alloc_host_amo(
    gdesc->sizeof_message);
   uint64_t host_address = HOST_HIGH;
   host_address = (host_address << 32) | (
    host_low_addr);

   spin_cmd_t dma;
   spin_dma_to_host(host_address, (uint32_t)
    msg, gdesc->sizeof_message, 1, &dma);
```

```
    spin_dma_wait(dma);
}
```

Listing 5. Pseudo-code for the payload handler.

D. Large Size Request

As profiled in many previous works, most of the messages for microservices are within 2KB [5], [20], [21], thus can be fit into one single packet. At the same time, these small messages always come in large numbers, polluting the cache and distracting the host CPUs from handling the core application logic, which will significantly impair microservices' overall serving ability. These large numbers of single packets, however, can be efficiently processed with sPIN in parallel without the involvement of the host CPU. Therefore, we mostly focus on processing small messages on the NIC. For large size requests that cannot be fit into one single packet, we dispatch them directly to the host, since these large requests are quite rare, as profiled in previous works.

E. Functional Offloading

Since the entire offloading workload is just about writing a handler in sPIN, for certain stateless services that have simple processing logic, they can be directly processed in the NIC with easy implementation (e.g., authentication), without the need to write the packets to the host, which further eases the CPU for more performance-critical tasks.

IV. Decoding Strategies on the NIC

A. Host Memory Management

The host allocates a buffer in host memory and transfers its ownership to the NIC. This pinned memory region is further split into disjoint regions that are assigned to each cluster. We have at the very beginning of each cluster's L1 scratchpad memory some bookkeeping information for managing its own partition of the pinned memory. Whenever an RPC request is decoded in the NIC and written to the host memory, the NIC will generate an event to notify the host by writing the address of the request to a waiting queue.

B. NIC Memory Management

On the NIC, each cluster has its own L1 scratchpad memory. We reserve some regions at the beginning of each cluster's scratchpad memory to cache the message descriptors. Then we further assign each HPU in the cluster a disjoint partition of the remaining scratchpad memory. Each HPU utilizes its own part of the L1 scratchpad memory as a stack.

C. Deserialization Methodology

The overall deserialization logic is similar to the original Protobuf. Typically, the most challenging thing for offloading deserialization to accelerators that do not share the same memory space with the host is *rewriting the pointers* to reference host memory space [3]. For this reason, most of the current works try to avoid this challenge either by using on-chip NICs [3] or putting accelerators near core [5], [22]. Instead, our approach directly addresses it, thanks to the fully-programmability of sPIN. In general, we replace the typical `malloc` and `memcpy` logic with `alloc_host` and `dma_to_host` in the deserialization logic. We will show with some examples how we decode messages on the NIC.

1) String Field: For example, to decode a field of `string` type as shown in Fig. 6, we first decode the field length varint, and allocate some memory on the pinned host memory according to the decoded length. Then we can use the DMA engine to write the string data directly from the L1 Packet Memory to the host memory ①, and the pointer `f_ptr` is then naturally referencing the host memory without the need for rewriting. Finally, we write the top-level message M to the host ②.

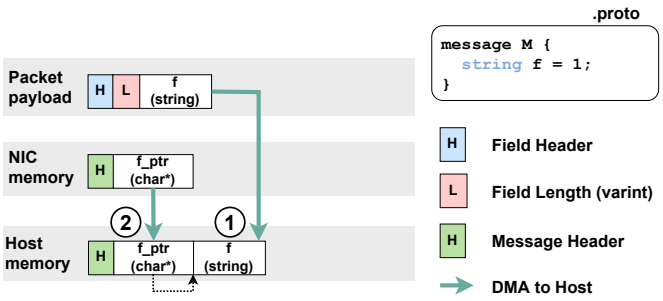
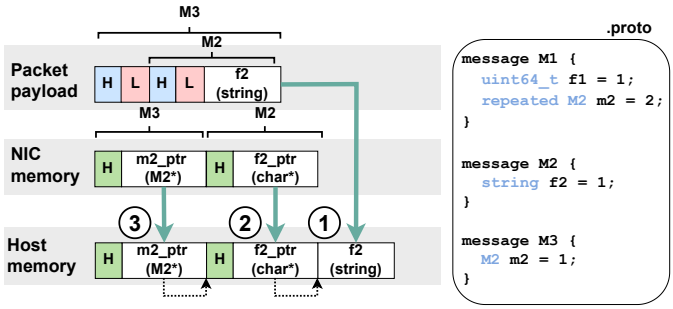


Fig. 6. Deserialization for Length-delimited field on one HPU.

2) Embedded Fields: There are also complex nested messages (e.g., messages with embedded and repeated fields) that need to store some intermediate data structures on the NIC before all of its child messages are decoded and sent to the host. Below, we give some examples of decoding these nested messages to show how we can properly deal with them. For the following examples, we use the message definitions as shown in Fig. 7(b).

As shown in Fig. 7(a), to decode an embedded message we just need to call the decoding function recursively. We regard our L1 scratchpad memory as a kind of "stack". To decode M3, we first decode field m1's header and length (remember that they are varint), and then we can know that m1 is an embedded field of type M2. The decoding of the inner M2 (①, ②) is similar to what we did in the previous example, and the host address to which the sub-message M2 is written is stored in `m_ptr` of M3. After all the fields of M3 are decoded, we write M3 to the host ③.

3) Repeated Fields: In Fig. 7(c), we have a much more complex message which has repeated fields of type M2. To start, we first decode the `uint64_t` field and keep the value in the corresponding field ①. We go on scanning the packet payload and record how many times M2 is repeated (i.e., the quantifier for repeated fields). Knowing the quantifier of the repeated field is 2 ②. In ③ we allocate an array of 2 `M2*` accordingly, and record start address of the array in `m2_vec_ptr`. For the decoding of the first M2, we follow similar procedure described in the embedded message case: decode M2 ④, allocate host memory (`hostaddr1`), write M2 to the host's `hostaddr1` ⑤, and set `m2_ptr[0]` to



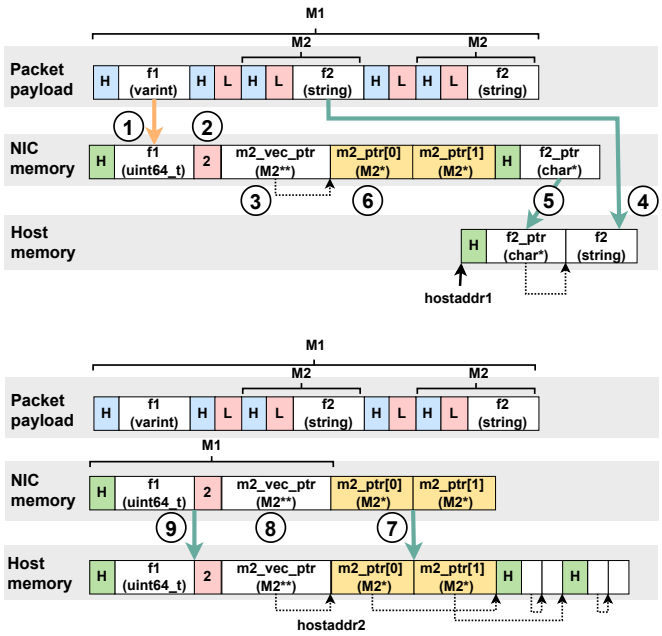
(a) Embedded Field

```
.proto
message M1 {
  uint64_t f1 = 1;
  repeated M2 m2 = 2;
}

message M2 {
  string f2 = 1;
}

message M3 {
  M2 m2 = 1;
}
```

(b) Proto Definition



(c) Repeated Field

Fig. 7. Deserialization for complex messages on one HPU.

be `hostaddr1` ⑥. We then just repeat these procedures for other M2s. At last, after all the fields are decoded, we first allocate host memory (`hostaddr2`) for the array of `M2*`, write them to the host ⑦, and reset `m2_vec_ptr` with `hostaddr2` ⑧. In the final step, we write the ProtobufCMessage structure for M1 to the host ⑨.

For general message decoding, we only need to repeat or recursively perform the fore-mentioned procedures.

V. Evaluation

We evaluated our approach using two sets of experiments: 1) Per-field deserialization throughput test. 2) Microservices benchmarks. For our approach, all the tests are performed on sPIN through cycle-accurate simulations with the PsPIN toolchain. PsPIN handlers are compiled with a PULP-custom version of GCC 7.1.1 (riscv32, `-O3 -flto`). For the per-field deserialization throughput test, we compare against a state-of-the-art near-core specialized hardware accelerator for protobufs, implemented in RTL and integrated into a RISC-V SoC. In the microservices benchmarking test, we compare our approach with decoding directly on AMD Ryzen 7 5700G CPUs (256KiB L1d and L1i cache, 4MiB L2 cache, and 16MiB L3 cache).

A. Per-field Deserialization Throughput

1) Workload: For each *field-bytes* pair, we inject into the network 256 messages of size "bytes" containing only the tested type of field.

2) Results: Fig. 8 shows the overall processing throughput of the per-field packet where each packet also includes $36B$ header information. Therefore, to get the real deserialization throughput related to the protobuf data, we do a recalculation: $Throughput_{real} = Throughput \times \frac{ProtoDataSize}{PacketSize}$. Fig. 9 shows the recalculated per-field deserialization throughput of PsPIN compared with ProtoACC [5]. Note that for ProtoACC, the data to be processed are already in the host memory, therefore the cost of moving data from the network stack to the host memory is not included. Still, PsPIN shows a higher per-field deserialization throughput thanks to the highly parallelized processing.

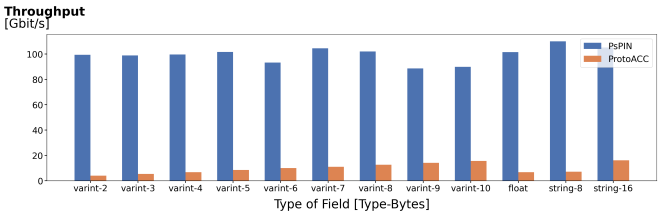


Fig. 8. Packet Processing Throughput Per Field.

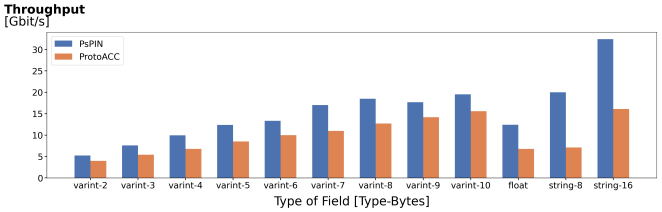


Fig. 9. Message Deserialization Throughput Per Field.

We can see from the simple type per-field results that the increase field size can result in higher deserialization throughput, which may due to better amortization on some fixed overhead per field (e.g., the overhead of processing the `key`).

B. Microservices Benchmarks

1) Evaluated Microservices: Similar as done in previous works [3], [7], we extract 5 most representative RPC requests from the DeathStartBench [23], composing three typical services in today's various kinds of microservices:

- User Service. This service contains 2 RPC methods: `RegisterUser` and `Login`. Methods in this class of service are composed of basic-type fields: integers and strings, and do not contain embedded and repeated

TABLE I
STATISTICS FOR REQUESTS

Type	Instructions	Loads	Stores	Branches	L2 Accesses	L1 Accesses	Avg. Msg. Size (B)
StorePost	3962	788	671	671	11	1279	427
ComposePost	1409	272	220	264	4	421	252
ComposeMedia	1615	315	248	295	5	497	76
RegisterUser	1008	182	167	196	1	281	103
Login	712	128	103	146	1	189	74

fields. However, as these requests normally do not carry long strings, they have relatively higher deserialization overhead per byte.

- Media Service. This service contains 1 RPC method: `ComposeMedia`. Method in this class of service has a deeper nested structure (i.e. depth $\geq$ 2) and some repeated fields, which will further increase the deserialization overhead per byte.
- Post Service. This service also contains 2 RPC methods: `ComposePost` and `StorePost`. Methods in this class of service have the deepest nested structures (i.e. depth $\geq$ 3) and many repeated fields used to represent the "Post" contents. However, on the other hand, these messages normally contain many long strings as "Post Text" that can better amortize the deserialization cost, resulting in lower per-byte overhead and higher throughput.

These five RPC requests vary in several aspects that can significantly influence the deserialization throughput:

- Number and composition of fields.
- Depth of the nested message.
- Whether containing long strings/bytes.

RPC requests in other microservices provided by DeathStarBench are similar to the requests we have extracted.

2) Workload Generation: For experiments, we have three modes: 1) single request, 2) single services, and 3) mixed services. In single services mode, for each service, the requests it contains will occur with the same probability. In mixed service mode, all the requests will occur with the same probability. Tab. I demonstrates the average instruction-related statistics for each request in the single request mode generated on the NIC.

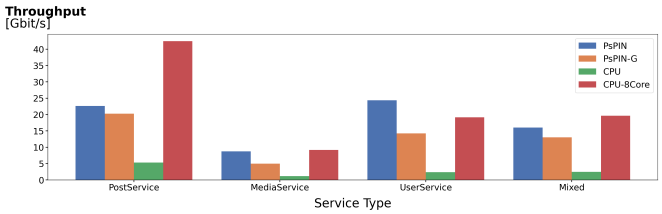


Fig. 10. Throughput for single/mixed services.

3) Deserialization throughput: Fig. 10 shows the throughput achieved by PsPIN and the CPU for deserializing requests belonging to different services. Fig. 11 shows the throughput for each request type involved in the considered services.

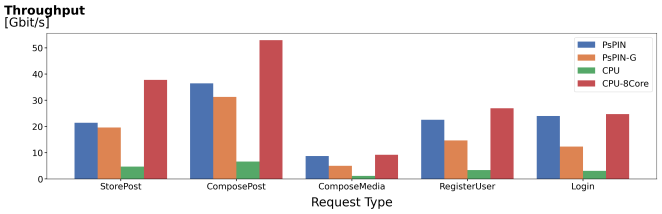


Fig. 11. Throughput for single request.

The CPU case models an ideal CPU-based deserialization scenario, where all requests are available in host memory and no contention or other overheads are experienced. Each service can generate different types of requests. For each service, we report: the PsPIN throughput (*PsPIN*); the throughput of PsPIN without considering packet header overheads (i.e., goodput), labeled as *PsPIN-G*; the single-core CPU throughput (*CPU*); and an extrapolation of the CPU performance when 8 cores are used for deserialization, labeled as *CPU-8Core*. PsPIN-based deserialization is consistently faster than the single-core CPU-based one (i.e., representing the case where requests are deserialized on one core before being dispatched to other worker threads): on average, PsPIN can achieve 4.3x and 4.8x higher throughput than a single CPU in request mode and service mode respectively. If we consider multi-core deserialization, then we see how the CPU outperforms PsPIN. While this is expected due to the compute-intensive nature of deserialization, it is worth noting that the CPU-based results represent a speed-of-light scenario, where all cores are dedicated to data deserialization and none to the execution of the actual microservices.

4) Microservice throughput estimation: To gain a better understanding of the impact of deserialization offloading to the NIC, we show in Fig. 12 an analysis of the overall throughput (y-axis) as a function of the microservice time (x-axis). As in Sec. V-B3, dedicating an 8-core CPU to the deserialization process results in better overall throughput than PsPIN. However, as the microservice running time increases, the deserialization overhead impacts the microservice throughput: i.e., deserialization and microservice are serially executed. Instead, with PsPIN these two activities are pipelined, resulting in better performance (i.e., the CPU can dedicate more cycles to the microservice execution). Finally, as the microservice time increases, the microservice execution itself becomes a bottleneck, causing the performance convergence of the two approaches.

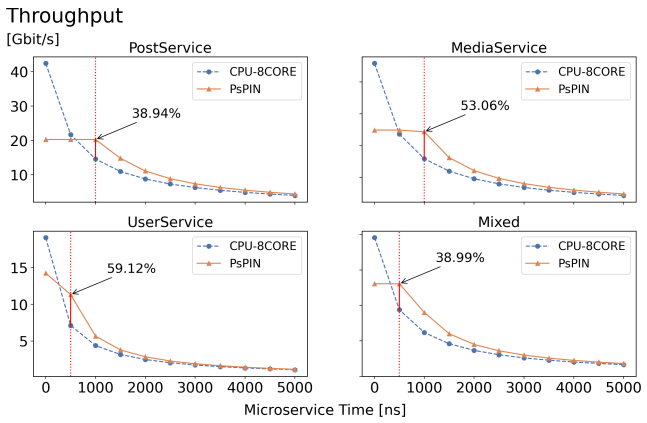


Fig. 12. Throughput modeling regarding microservice time.

VI. Related Work

Most of the existing works co-design the software and hardware to accelerate the SerDes, either near-core or attached to the NIC. *Cereal* [22] is a specialized hardware accelerator for SerDes fo Java objects, which co-designs the serialization format with hardware architecture. *Optimus Prime* [7] presents an SerDes accelerator that can exploit the field-level parallelism for deserialization, which requires the application to construct and pass to the accelerator a per-message *schema*, holding the type and address of each field. Such method not only introduces additional memory/compute overhead for the applications but also needs to change all Protobuf setters and clear methods. *ProtoACC* [5] proposed a novel near-core specialized hardware accelerator for Protobuf. *Zerializer* [24] proposes (but does not implement) a hardware design for accelerating SerDes. *Cerebros* [3] and *Dagger* [20], offload the entire RPC stack to NIC-attached specialized hardware and near-core NIC respectively. However, *Dagger* cannot support deserializing complex messages that contain references to other objects. In general, accelerators such as *Cerebros* and *Dagger* are even more specialized and require laborious hardware/software co-design. Other works such as [25] build new SerDes libraries leveraging NIC's scatter-gather capabilities instead of offloading existing popular SerDes protocols.

Notably, our proposed method is purely on the software level and is thus with much higher flexibility. Moreover, the methodology adopted in our deserialization process well addresses the challenge of transferring and manipulating complex pointer-based data structures between different memory spaces and can be applied to other similar tasks.

VII. Conclusion

We propose in this work a purely software-level data SerDes acceleration approach with in-network compute. Targeting general-purpose fully-programmable SmartNICs, our approach in general deals with the problem of transferring and manipulating complex pointer-based data structures between different memory spaces, which has been a major challenge of offloading SerDes to the NIC [3]. We discuss and identify SmartNIC specialties that are needed to offload SerDes operations. We then show through microservices benchmark experiments that our approach can achieve on average 4.3x and 4.8x higher throughput than a single AMD Ryzen 7 CPU in request and service mode respectively. We finally perform microservice throughput modeling to demonstrate how we can improve the overall throughput by pipelining the deserialization and actual application activities with PsPIN. We believe it is both beneficial and flexible to offload data SerDes to SmartNICs.

Acknowledgment

This work has been partially funded by the European Projects RED-SEA (grant no. 955776).

References

[1] D. Vohra, "Apache parquet," in *Practical Hadoop Ecosystem*. Springer, 2016, pp. 325–335.
[2] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–20.
[3] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the rpc tax in datacenters," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, p. 407–420.
[4] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer," in *International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.
[5] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan, "A hardware accelerator for protocol buffers," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 462–478.
[6] J. Jang, S. 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 *International Symposium on Computer Architecture (ISCA)*, 2020, pp. 322–334.
[7] A. P. Zarandi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1203–1216.
[8] K. Varda, "Google protocol buffers: Google's data interchange format," Technical report, Tech. Rep., 2008.
[9] W. Abernethy, *Programmer's Guide to Apache Thrift*. Simon and Schuster, 2019.
[10] Apache Software Foundation, "Apache avro," https://avro.apache.org/, 2012.
[11] W. van Oortmerssen, "Flatbuffers: a memory efficient serialization library," https://opensource.googleblog.com/2014/06/flatbuffers-memory-efficient.html, 2014.
[12] K. Varda, "Cap'nproto," https://capnproto.org/, 2020.
[13] S. D. Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler, "Network-accelerated non-contiguous memory transfers," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2019, pp. 56:1–56:14.
[14] R. Rajesh, K. B. Ramia, and M. Kulkarni, "Integration of lwip stack over intel(r) DPDK for high throughput packet delivery to applications," in *International Symposium on Electronic System Design*. IEEE Computer Society, 2014, pp. 130–134.
[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[16] J. Kicinski and N. Viljoen, "Hardware offload to smartnics : cls bpf and xdp," 2016.
[17] T. Hoefler, S. D. Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "spin: high-performance streaming processing in the network," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 59:1–59:16.

[18] S. D. Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler, "A RISC-V in-network accelerator for flexible high-performance low-power packet processing," in *International Symposium on Computer Architecture (ISCA)*, 2021, pp. 958–971.

[19] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "PULP: A parallel ultra low power platform for next generation iot applications," in *IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–39.

[20] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 36–51.

[21] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 733–750.

[22] J. Jang, S. 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 *International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 322–334.

[23] 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 *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 3–18.

[24] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: towards zero-copy serialization," in *Workshop on Hot Topics in Operating Systems (HotOS)*. ACM, 2021, pp. 206–212.

[25] D. Raghavan, P. A. Levis, M. Zaharia, and I. Zhang, "Breakfast of champions: towards zero-copy serialization with NIC scatter-gather," in *Workshop on Hot Topics in Operating Systems (HotOS)*, 2021, pp. 199–205.