Towards Efficient Datacenter Serialization

Deepti Raghavan*, Philip Levis*, Matei Zaharia*, Irene Zhang+
* Stanford University  ♦ Microsoft Research
Platform Lab Retreat
Thursday, February 11, 2021
Datacenter servers have less cycles to process packets to achieve line rate

- Networks have reached 400 Gbps speeds
- Kernel bypass (e.g., DPDK, RDMA) significantly reduces OS level packet processing overheads
Datacenter servers have less cycles to process packets to achieve line rate

- Networks have reached 400 Gbps speeds
- Kernel bypass (e.g., DPDK, RDMA) significantly reduces OS level packet processing overheads
- Datacenter applications today achieve microsecond packet round trip times

---


Microsecond Applications Can’t Afford Software Data Serialization

• Serialization is a core primitive for modern distributed systems

• Serialization transforms data structures from an app’s memory layout into a network-optimal representation


Microsecond Applications Can’t Afford Software Data Serialization

message Object {
  optional string msg = 1;
}

- Protobuf serialization + deserialization of simple data structure, 1024-byte sized, takes 1.1 us

![Graph showing latency comparison between various RTTs](image)


Research has proposed custom accelerators for serialization.

Optimus Prime: Accelerating Data Transformation in Servers

A Specialized Architecture for Object Serialization with Applications to Big Data Analytics

jaey86@skku.edu (miquel92, sunnyday8208, j.heo, zhnp, taejunham, jaewlee}@snu.ac.kr
Why is serialization a bottleneck and can we fix it in software?

Yes! By re-purposing existing capabilities in NICs to accelerate serialization’s core tasks!

- Kernel bypass exposes these capabilities to userspace applications
- Need to change serialization API to access these capabilities
Talk Agenda

• Introduction

• Motivation: How inefficient is serialization and why?
  • Only focusing on compilation-based approaches

• How do we fix serialization?

• Open Research Questions
How Inefficient is Serialization?

```java
message Object {
  optional string msg = 1;
}
```

- Echo server running over UDP DPDK stack
- 1024 byte-sized message
- Three serialization libraries, two baseline
How Inefficient is Serialization?

message Object {
  optional string msg = 1;
}

- **Protobytes**: uses bytes payload instead of a string
  - Removes utf8-validation
How Inefficient is Serialization?

```message Object {
    optional string msg = 1;
}
```

- **No Serialization:**
  - Removes serialization + deserialization

- **DPDK Single Core:**
  - Buffers aren’t copied in and out of networking stack
  - Peak zero-copy throughput of DPDK
How Inefficient is Serialization?

- Software serialization approaches achieve only up to **50% of DPDK’s peak single core throughput**

```java
message Object {
  optional string msg = 1;
}
```
message Object {
  optional string msg = 1;
}

- Software serialization approaches achieve only up to 50% of DPDK’s peak single core throughput.
Gap 1: CPU-Based Serialization is Bottlenecked By Data Movement

• Tension between application memory layout and optimal wire layout:
  • Programmers like scattered memory
    • Can modify data structures *without re-allocating* all the memory contiguously
  • Serialization *moves* scattered pointers into some *contiguous format*
Gap 1: CPU-Based Serialization is Bottlenecked By Data Movement

• Some request patterns involve bringing together memory values from different locations
  • Responding with multiple values for multiple keys in a key value store
  • Redis “mget” command does this

```
mget([key1, key3, key5])
```

Client

```
val 1  val 3  val 5
```

KV Server

```
key 1 | val 1
key 2 | val 2
key 3 | val 3
key 4 | val 4
key 5 | val 5
```
Gap 1: CPU-Based Serialization is Bottlenecked By Data Movement

- Moving memory in software requires copies and allocations:

  Allocate size of data structure fields + header

  ```
  "foo": 0xfoo
  "bar": 0xbar
  ```

  Copy foo

  Copy bar
Gap 1: CPU-Based Serialization is Bottleneck By Data Movement

```
message Object {
  optional string msg = 1;
}
```

- Initialize data structure:
  - *Init proto or Capnproto object*
- Copy string payload:
  - *Set the string field*

<table>
<thead>
<tr>
<th>STEP</th>
<th>Protobuf</th>
<th>Cap’n Proto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Data Structure</td>
<td>34 ns</td>
<td>352 ns*</td>
</tr>
<tr>
<td>Copy String Payload</td>
<td>172 ns*</td>
<td>85 ns*</td>
</tr>
<tr>
<td>Encode to Wire Format</td>
<td>544 ns*</td>
<td>52 ns</td>
</tr>
<tr>
<td>Decode from Wire Format</td>
<td>411 ns*</td>
<td>80 ns</td>
</tr>
<tr>
<td>Total Overhead</td>
<td>1161 ns</td>
<td>569 ns</td>
</tr>
</tbody>
</table>
Gap 1: CPU-Based Serialization is Bottleneck By Data Movement

```plaintext
message Object {
    optional string msg = 1;
}
```

- Encode to wire format:
  - *Turn in-memory format into contiguous representation*

- Decode from wire format:
  - *Turn wire representation into memory format*

<table>
<thead>
<tr>
<th>Step</th>
<th>Protobuf</th>
<th>Cap’n Proto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Data Structure</td>
<td>34 ns</td>
<td>352 ns*</td>
</tr>
<tr>
<td>Copy String Payload</td>
<td>172 ns*</td>
<td>85 ns*</td>
</tr>
<tr>
<td>Encode to Wire Format</td>
<td>544 ns*</td>
<td>52 ns</td>
</tr>
<tr>
<td>Decode from Wire Format</td>
<td>411 ns*</td>
<td>80 ns</td>
</tr>
<tr>
<td>Total Overhead</td>
<td>1161 ns</td>
<td>569 ns</td>
</tr>
</tbody>
</table>
Gap 1: CPU-Based Serialization is Bottleneck By Data Movement

```plaintext
message Object {
  optional string msg = 1;
}
```

- Cap’nProto < Protobuf:
  - Cap’nProto wire format = in-memory format
- But even Cap’nProto allocates space for fields and copies in payloads

<table>
<thead>
<tr>
<th>STEP</th>
<th>Protobuf</th>
<th>Cap’n Proto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize Data Structure</td>
<td>34 ns</td>
<td>352 ns*</td>
</tr>
<tr>
<td>Copy String Payload</td>
<td>172 ns*</td>
<td>85 ns*</td>
</tr>
<tr>
<td>Encode to Wire Format</td>
<td>544 ns*</td>
<td>52 ns</td>
</tr>
<tr>
<td>Decode from Wire Format</td>
<td>411 ns*</td>
<td>80 ns</td>
</tr>
<tr>
<td>Total Overhead</td>
<td>1161 ns</td>
<td>569 ns</td>
</tr>
</tbody>
</table>
message Object {
  optional string msg = 1;
}

- 3 Gbps gap between flatbuffers and no serialization comes from data movement cost
  - Penalty for data structures with large (> ~256 bytes) payloads
How Inefficient is Serialization?

```cpp
message Object {
  optional string msg = 1;
}
```

- 3 Gbps gap between flatbuffers and no serialization comes from *data movement cost*
  - Penalty for data structures with large (> ~256 bytes) payloads

![Graph showing p99 latency vs. throughput for different serialization methods](image)

- Networking Stack <-> Application Boundary gap
- Serialization gap
- Protobuf, Capnproto, No Serialization, Protobuffer, Flatbuffers, DPDK Single Core
Gap # 2: Boundary Between Serialization Library and Networking Stack

• Kernel bypass stacks require that packets live in pinned, registered memory
• Serialization Library agnostic to any networking stack memory requirements
• Causes a copy on send and receive
message Object {
    optional string msg = 1;
}

- 3 Gbps gap between flatbuffers and no serialization comes from *data movement cost*
- 2 Gbps gap between no serialization and dpdk single core comes from *differing memory requirements*
Talk Agenda

• Introduction
• Motivation: How inefficient is serialization and why?
  • How do we fix serialization?
• Open Research Questions
How do we make serialization more efficient?

Wouldn’t it be nice if we could transmit app memory from its original location somehow, so no data movement is required?

CPUs are bad at data movement!

NIC photo: https://twitter.com/graziaprato
How do we make serialization more efficient?

• Datacenter servers *already have* a hardware accelerator for coalescing non-contiguous I/O regions: the NIC itself

• Modern NICs have *scatter-gather* functionality that coalesce and flatten scattered memory regions
How do we use scatter-gather to accelerate serialization?

- Turn data structures into scatter-gather arrays rather than single buffer
- Give scatter-gather array to the networking stack
  - Assumes app memory lives in DMA’able pages

```c
struct ScatterGatherArray {
    void * ptr;
    size_t length;
}
```

NIC photo: https://twitter.com/graziaprato
What would a scatter-gather based serialization API look like?

```cpp
message Object { optional string msg = 1; }

class ObjectGenerated {
  std::pair<char *, size_t> get_msg();
  void set_msg(const char *addr, size_t len);
  ScatterGatherArray serialize(size_t num_entries);
  void deserialize(const char *payload);
};
```
Simple Algorithm to Translate Data structures into Scatter-Gather Arrays

message Object {
    optional int id = 1;
    optional string msg = 2;
}

- Create a fixed size object header and set data structure fields:
  - Header indexes which fields are present
- First scatter-gather entry = packet header + object header copied in
- Create a scatter-gather entry for each remaining data structure field

```
message Object {
    optional int id = 1;
    optional string msg = 2;
}
```

```
object header

1

“1024-byte-string”
```

```
packet header  object header

sizeof(uint32_t)

1024

sizeof(pkt_hdr) + sizeof(obj_hdr)
```
For PCIe connected NICs, putting small buffers in separate entries adds overhead

• **Characteristic # 1**: Each scatter-gather entry adds extra latency penalty due to an extra PCIe round-trip

• **Characteristic # 2**: Zero-Copy is only worth it when memory regions are large enough

Can’t waste scatter-gather entries on super small fields!
Revised Algorithm to Translate Data Structures into Scatter-Gather Arrays

• Create a fixed size object header:
  • Indexes which fields are present
  • Copy in any integer sized field directly to the header

• First scatter-gather entry = packet header + object header copied in

• Create a scatter-gather entry for each remaining, variable sized field

```
sizeof(pkt_hdr) + sizeof(obj_hdr) + sizeof(uint32_t) = 1024
```

"1024-byte-string"
message Object {
  optional string msg = 1;
}

• Buffer initialized in DMA’able memory
• Prototype library performs slightly better than No Serialization baseline
• Overhead from an entire entry for just the packet and object header
Open Research Questions

• How does scatter-gather behave across NICs?
  • Can we come up with a model of scatter-gather behavior for different NICs?

• How do we use scatter-gather efficiently?
  • Given a performance model, how do we translate application data structures into scatter-gather arrays that work well?
    • Some fields will need to be copied -- which fields?
Open Research Questions

• What about deserialization?
  • Scatter-gather only on send
  • On receive -> what if memory needs to go to some specific location (and is written rather than read)?
    • Redis supports modifying in-memory data structures (indexed by some key)
      • e.g., appending to a list
Open Research Questions

• How do we access application memory for zero-copy transmission?
  • Kernel bypass aware memory allocators

• How do we ensure memory safety with zero-copy?
  • Need to ensure write protection and free protection
Conclusion: serialization will be a bottleneck for datacenter applications

• Software serialization can only achieve up to 50% of DPDK’s peak single core throughput
• NIC’s scatter-gather offload presents an opportunity to achieve zero-copy, zero-allocation serialization that can keep up with modern networks

deeptir@cs.stanford.edu
https://deeptir.me
github.com/deeptir18