Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking

SOSP 2023

Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, Irene Zhang

NIC speeds increase, CPUs are stagnant

20 Gbit/s throughput

2008

~100µs end-host RTT

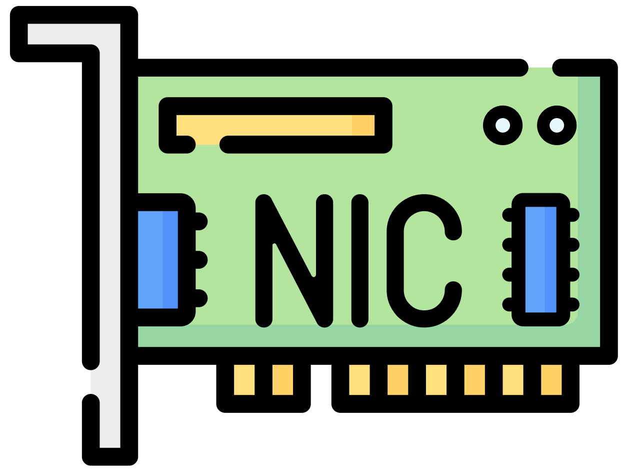
20x

20x

400 Gbit/s throughput

2023

~5µs end-host RTT

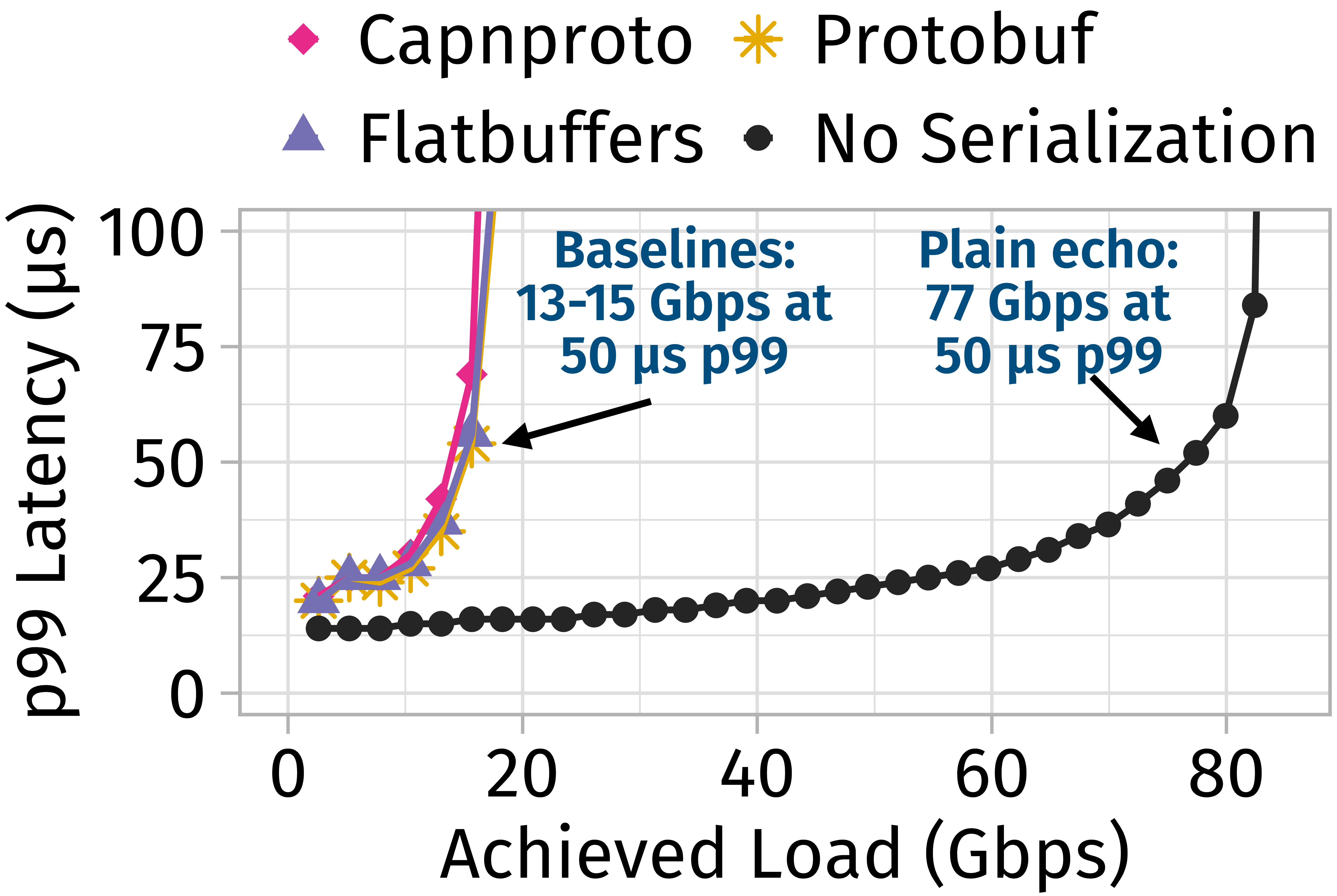


- ***Shaving down CPU time*** is important for modern microsecond-scale apps
  - CPU frequencies stagnated at ***3-4 GHz***
  - Multicore helps, but not always

Serialization is a bottleneck for (10s of)-µs-scale apps

- Serialization: converting in-memory data structures to on-the-wire formats for communication

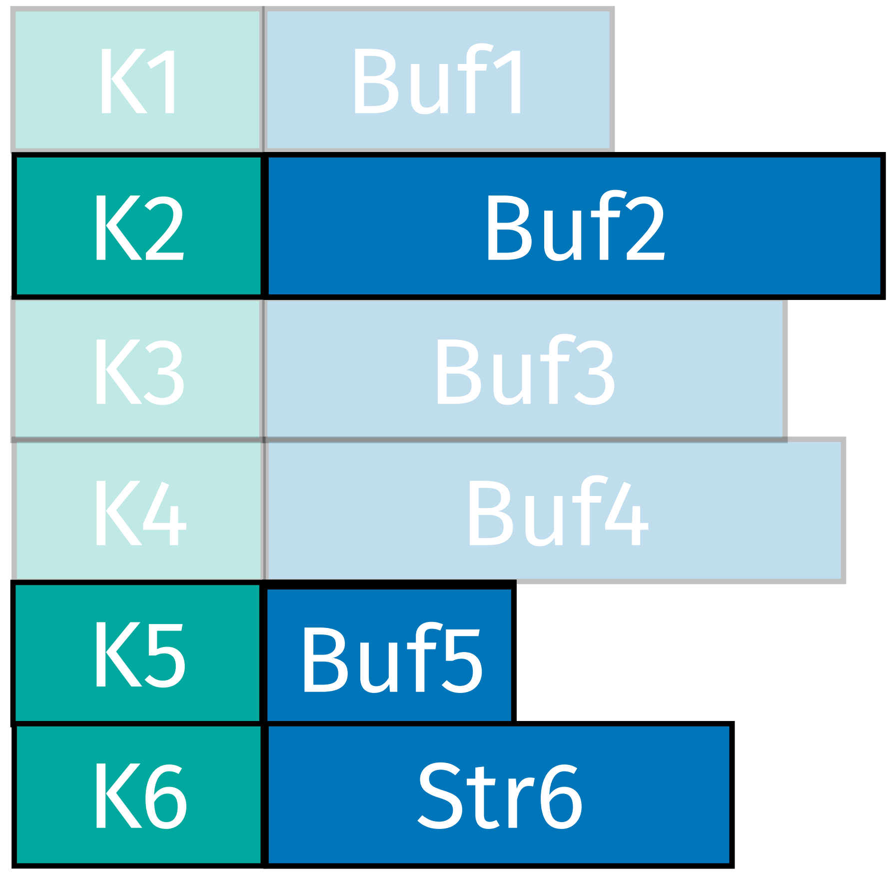
- Experimental takeaway: serialization ***reduces achievable throughput by 80%***



- ***Message size***: 4096 bytes
- ***Serialized struct***: 2 equal-sized bytes fields
- ***Networking stack:*** minimal UDP stack (100 Gb NIC)

Networked serialization’s core task is data movement

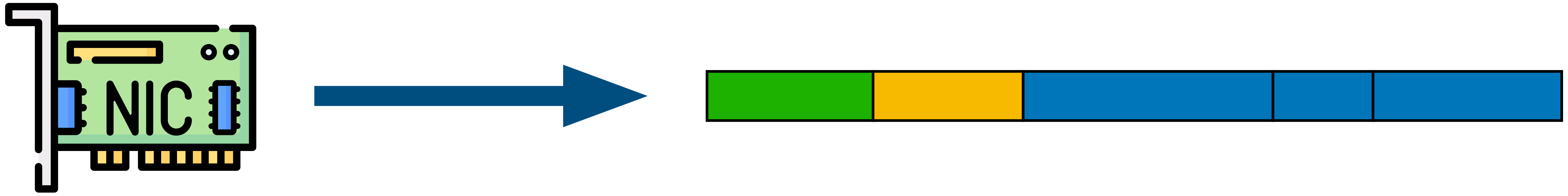
Application Layer




```
let getm = GetM::new();
getm.init_vals(3);
getm.get_mut_vals().append(&buf2);
getm.get_mut_vals().append(&buf5);
getm.get_mut_vals().append(&str6);
netstack.send(getm.serialize_to_bytes());
```

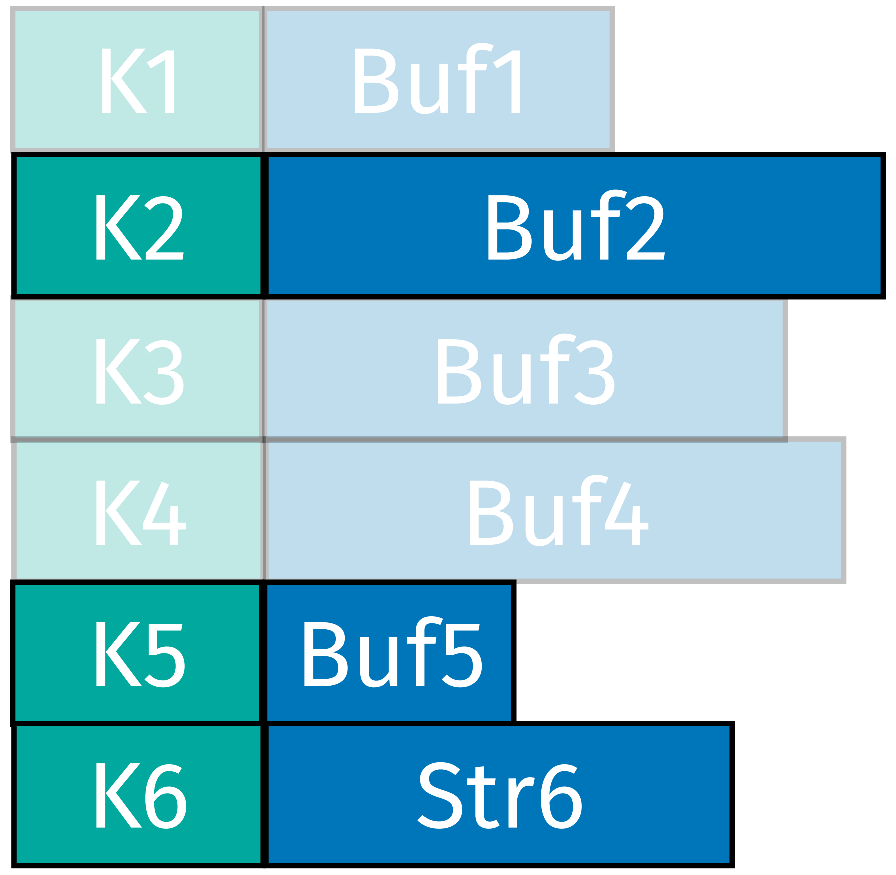
Serialization Library

Networking Stack



Networked serialization’s core task is data movement

Application Layer



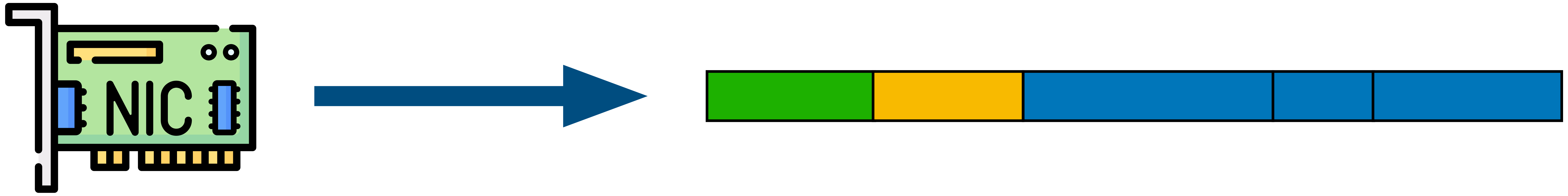

```
let getm = GetM::new();
getm.init_vals(3);
getm.get_mut_vals().append(&buf2);
getm.get_mut_vals().append(&buf5);
getm.get_mut_vals().append(&str6);
netstack.send(getm.serialize_to_bytes());
```

Copy 1

Serialization Library

Networking Stack

Pinned (unswappable) Memory



Networked serialization's core task is data movement

Application Layer

K1	Buf1
K2	Buf2
K3	Buf3
K4	Buf4
K5	Buf5
K6	Str6

```
let getm = GetM::new();
getm.init_vals(3);
getm.get_mut_vals().append(&buf2);
getm.get_mut_vals().append(&buf5);
getm.get_mut_vals().append(&str6);
netstack.send(getm.serialize_to_bytes());
```

Copy 1

Serialization Library

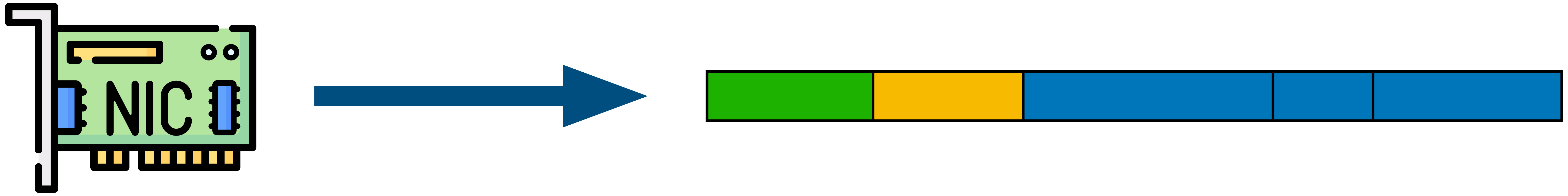
Obj Hdr	Buf2	Buf5	Str6

Copy 2

Networking Stack

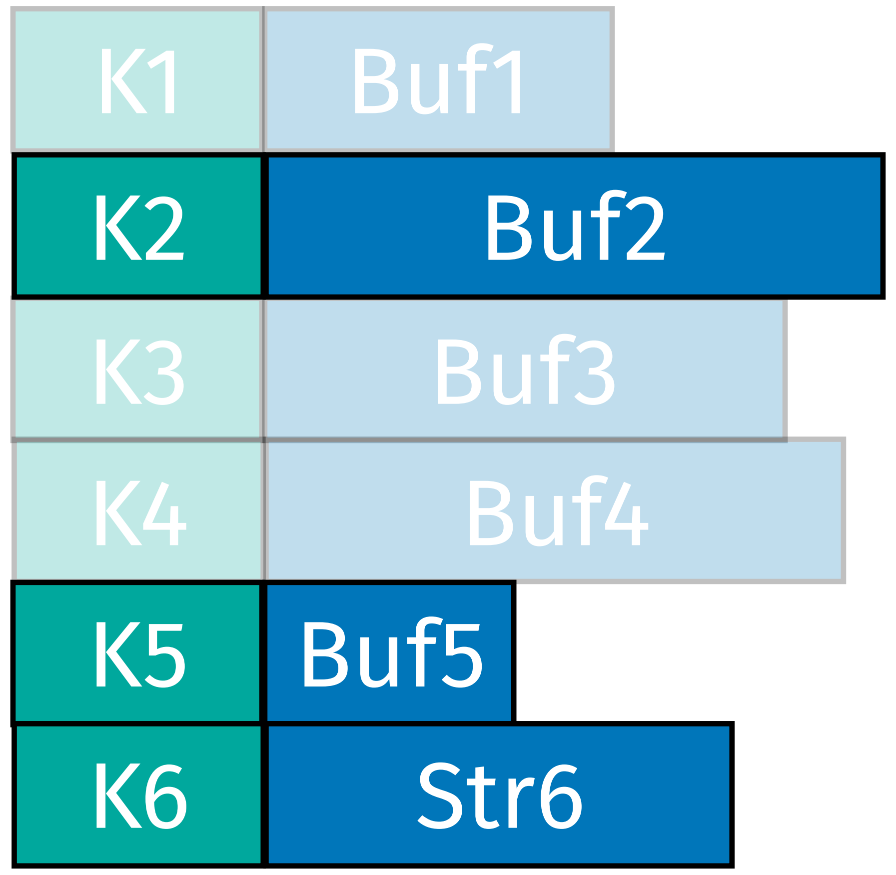
Pkt Hdr	Obj Hdr	Buf2	Buf5	Str6

Pinned (unswappable) Memory



Networked serialization’s core task is data movement

Application Layer



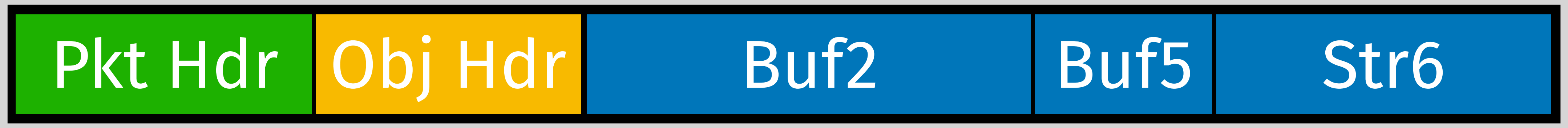

```
let getm = GetM::new();
getm.init_vals(3);
getm.get_mut_vals().append(&buf2);
getm.get_mut_vals().append(&buf5);
getm.get_mut_vals().append(&str6);
netstack.send(getm.serialize_to_bytes());
```

Copy 1

Serialization Library

Copy 2

Networking Stack



Pinned (unswappable) Memory

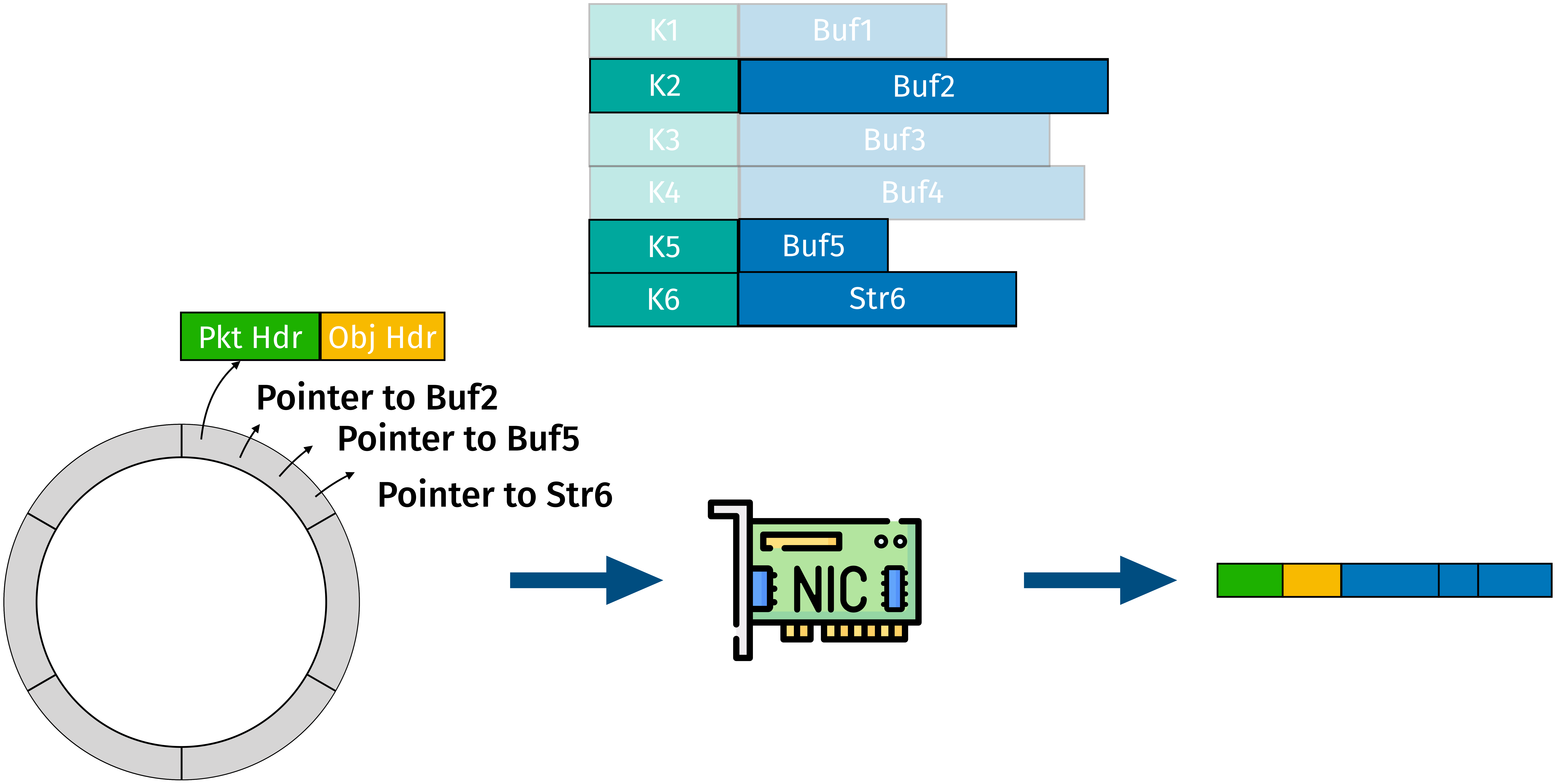
Networked serialization today involves up to two data copies. Can existing zero-CPU-copy APIs remove copies and reduce overhead?

Yes, but it's complicated….

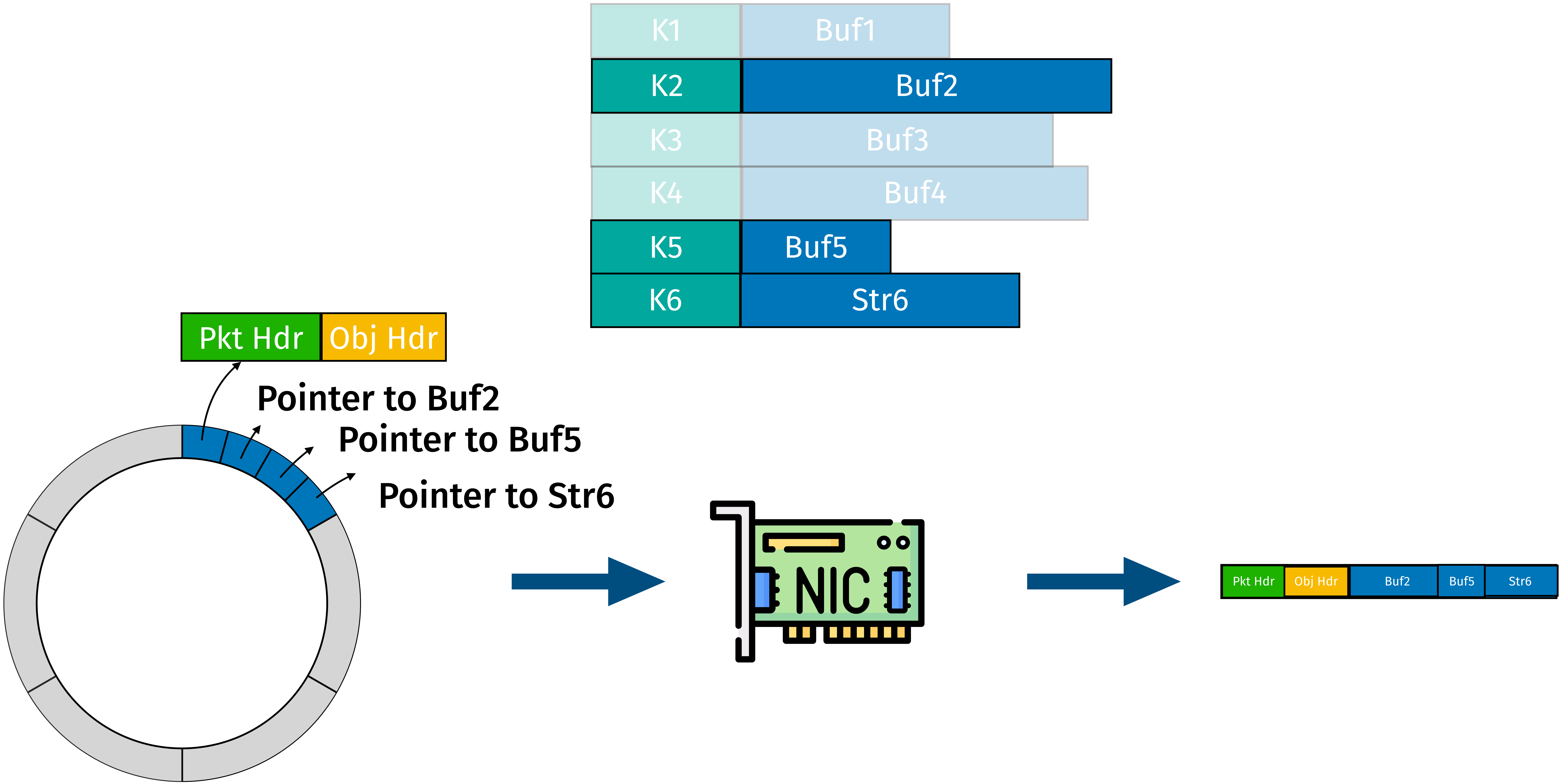
Talk Agenda

- Introduction and background
- Motivation for using zero-copy APIs in serialization and challenges
- Design of ***Cornflakes***: a new, general-purpose, zero-copy, serialization library
- Implementation
- Evaluation

Zero-CPU-copy avoids copies through DMA

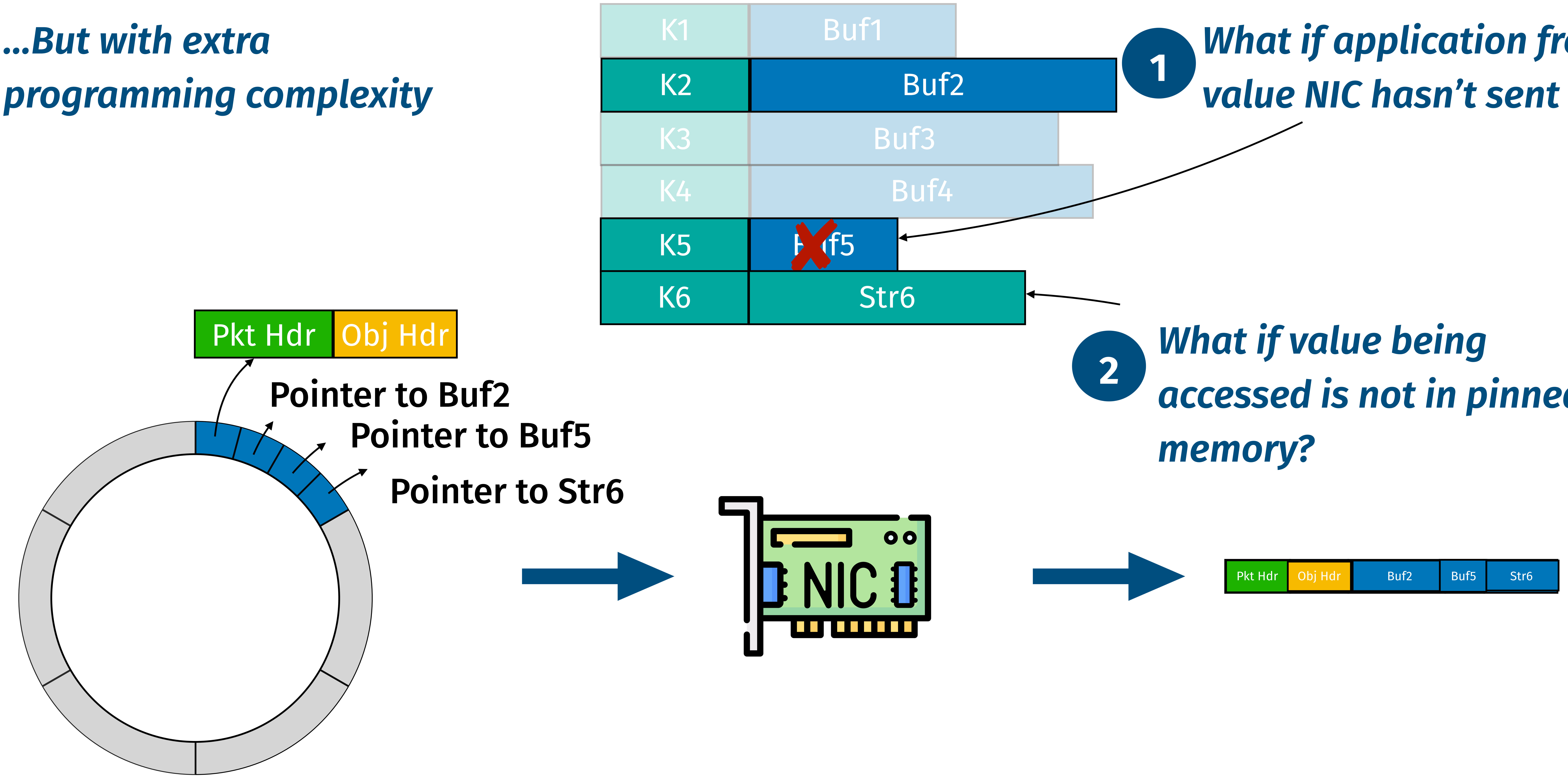


Zero-CPU-copy avoids copies through DMA



Zero-CPU-copy avoids copies through DMA

…But with extra programming complexity



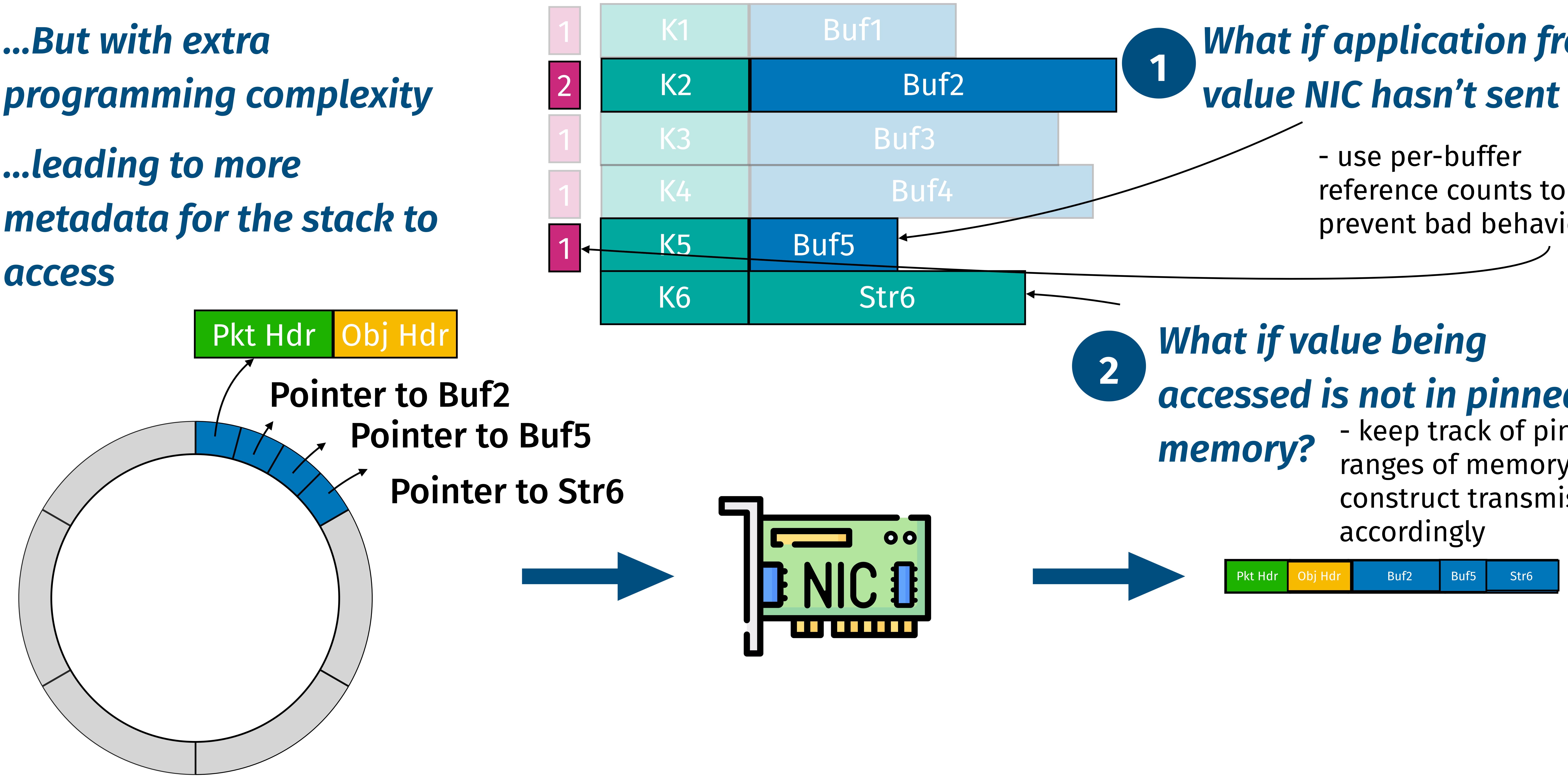
1 ***What if application frees value NIC hasn't sent yet?***

2 ***What if value being accessed is not in pinned memory?***

Zero-CPU-copy avoids copies through DMA

...But with extra programming complexity

...leading to more metadata for the stack to access



1 *What if application frees value NIC hasn't sent yet?*

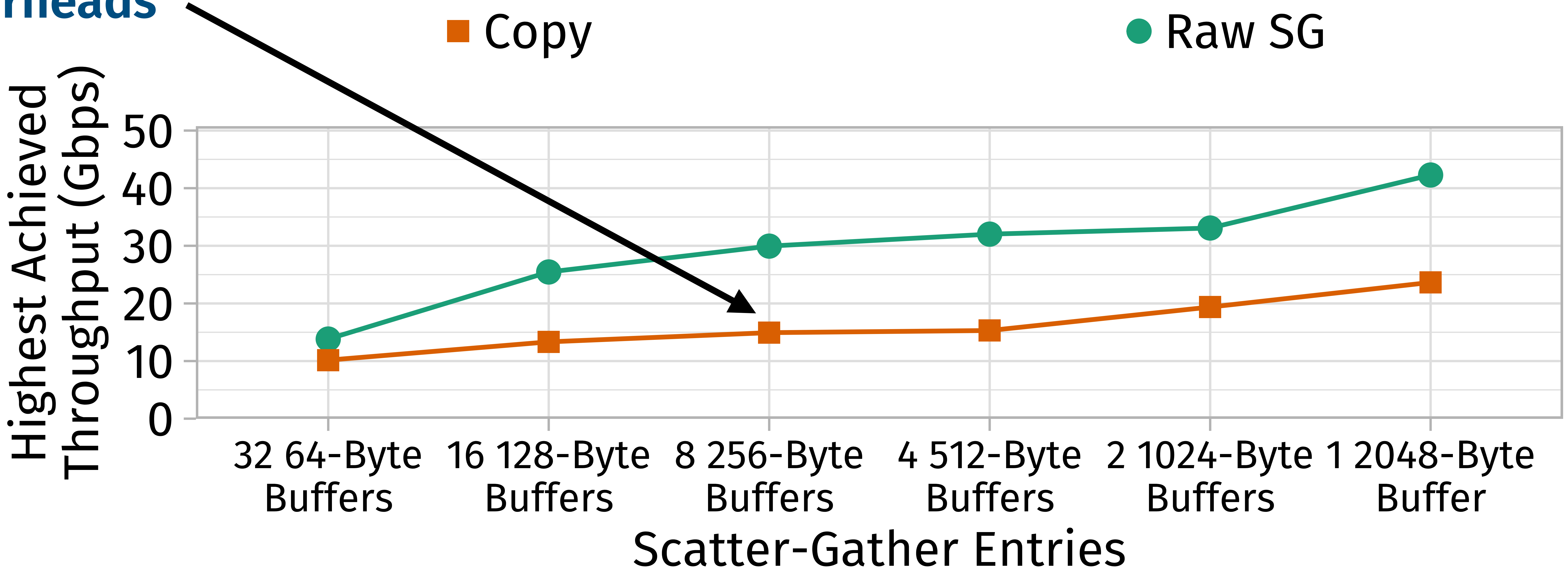
- use per-buffer reference counts to prevent bad behavior

2 *What if value being accessed is not in pinned memory?*

- keep track of pinned ranges of memory and construct transmission accordingly

Reference counting imposes cache misses, which add overhead

For 256 bytes and below, copy better than SG w/ overheads



- Workload: Server serves client requests for individual segments from large array of physical segments using copy or scatter-gather

When considering cache misses, copy beats zero-copy for small objects

Challenges of applying zero-copy APIs to serialization:

- ***Not always good to zero-copy***: for small objects, reference count access is not worth it
- Programmer ***must manually choose*** between copy/zero-copy (based on memory location and performance)
- ***Separating serialization from networking*** adds overhead

Cornflakes applies zero-copy to serialization and contributes:

- ***Measurement study*** that shows when zero-copy is useful
- ***Efficient, hybrid copy/zero-copy API*** that hides complexity from programmer
- Co-designed API between serialization and networking which allows for ***combined serialize-and-send***

Cornflakes library design tackles three challenges

Which values should be zero-copied?

Efficient heuristic to ***choose mechanism*** at runtime, based on ***measurement study***

How can the hybrid API be transparent?

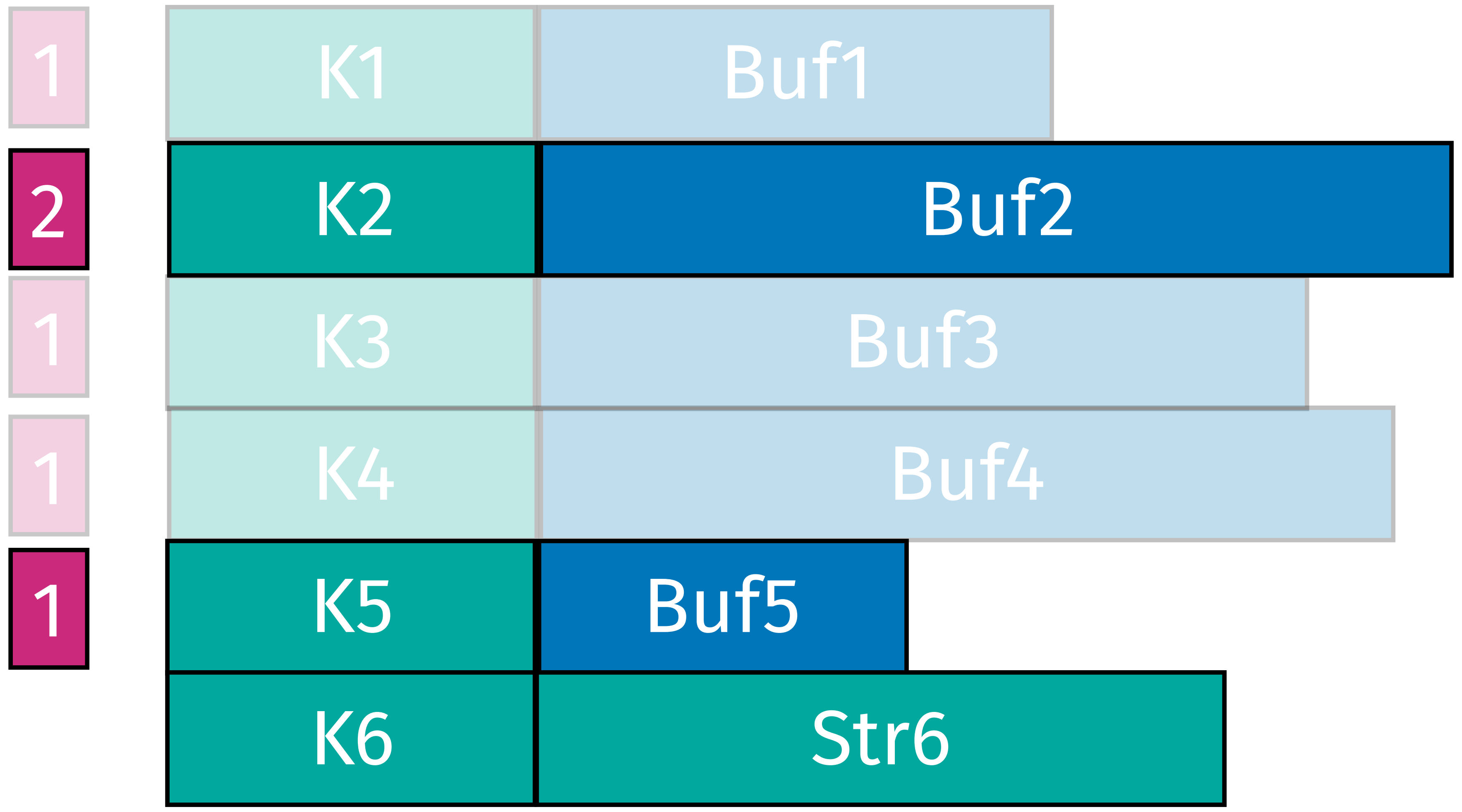
Represents data as ***either copy or zero-copy,*** without extra programmer effort

What API should exist between serialization and networking?

Novel ***serialize-and-send*** optimization

Cornflakes uses a greedy heuristic based on a threshold

Which values should be zero-copied?




```
getm.get_mut_vals().append(&buf2);
```

Use zero-copy (incr reference count)

Use copy (access data)

How can the hybrid API be transparent?

Decide on copy / zero-copy greedily

```
append(&buf2) => zero-copy (R2++)
```

Buf2 >= Threshold

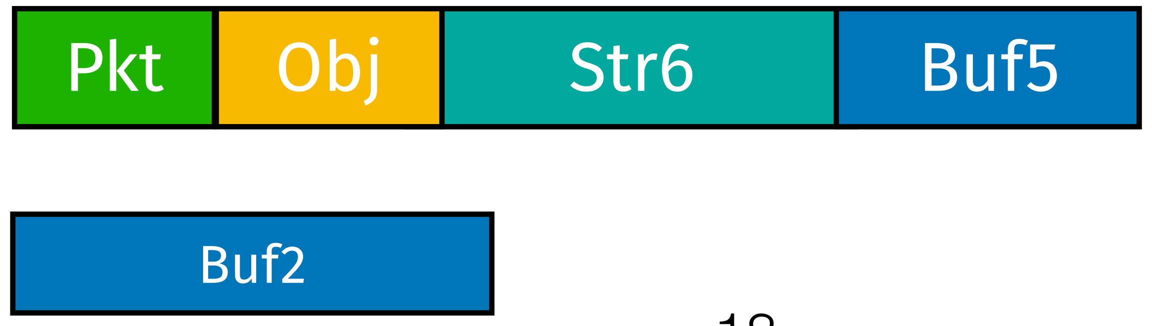
```
append(&buf5) => use copy repr
```

Buf5 < Threshold

```
append(&str6) => use copy repr
```

We use a ***measurement study to determine*** the threshold value for ***our hardware: 512 bytes***

What API should exist between serialization and networking?



Serialize-and-send eliminates intermediate representations

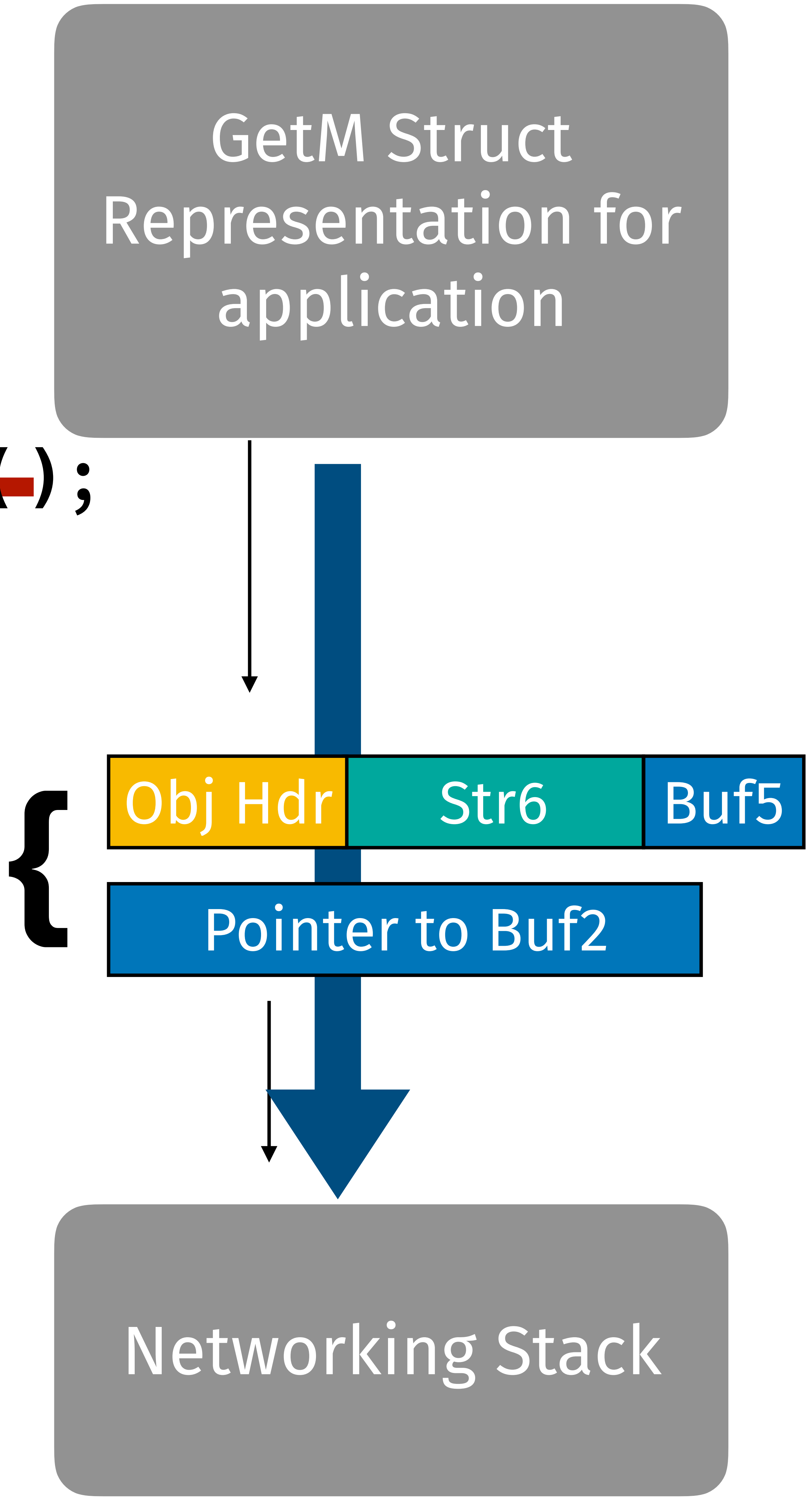
Which values should be zero-copied?

How can the hybrid API be transparent?

What API should exist between serialization and networking?

```
let getm = GetM::new();
getm.init_vals(3);
getm.get_mut_vals().append(&buf2);
getm.get_mut_vals().append(&buf5);
getm.get_mut_vals().append(&str6);
let sg_array = getm.serialize();
netstack.send(sg_array);
netstack.send(getm);
```

APIs like DPDK's require materializing intermediate lists of pointers



Cornflakes implementation

- ***Application library***
  - Rust code + C bindings
- ***Networking stack support***
  - Mellanox OFED, Intel ICE direct drivers
  - Partial demikernel integration
- ***Applications***
  - Custom key value store, echo server
  - Redis

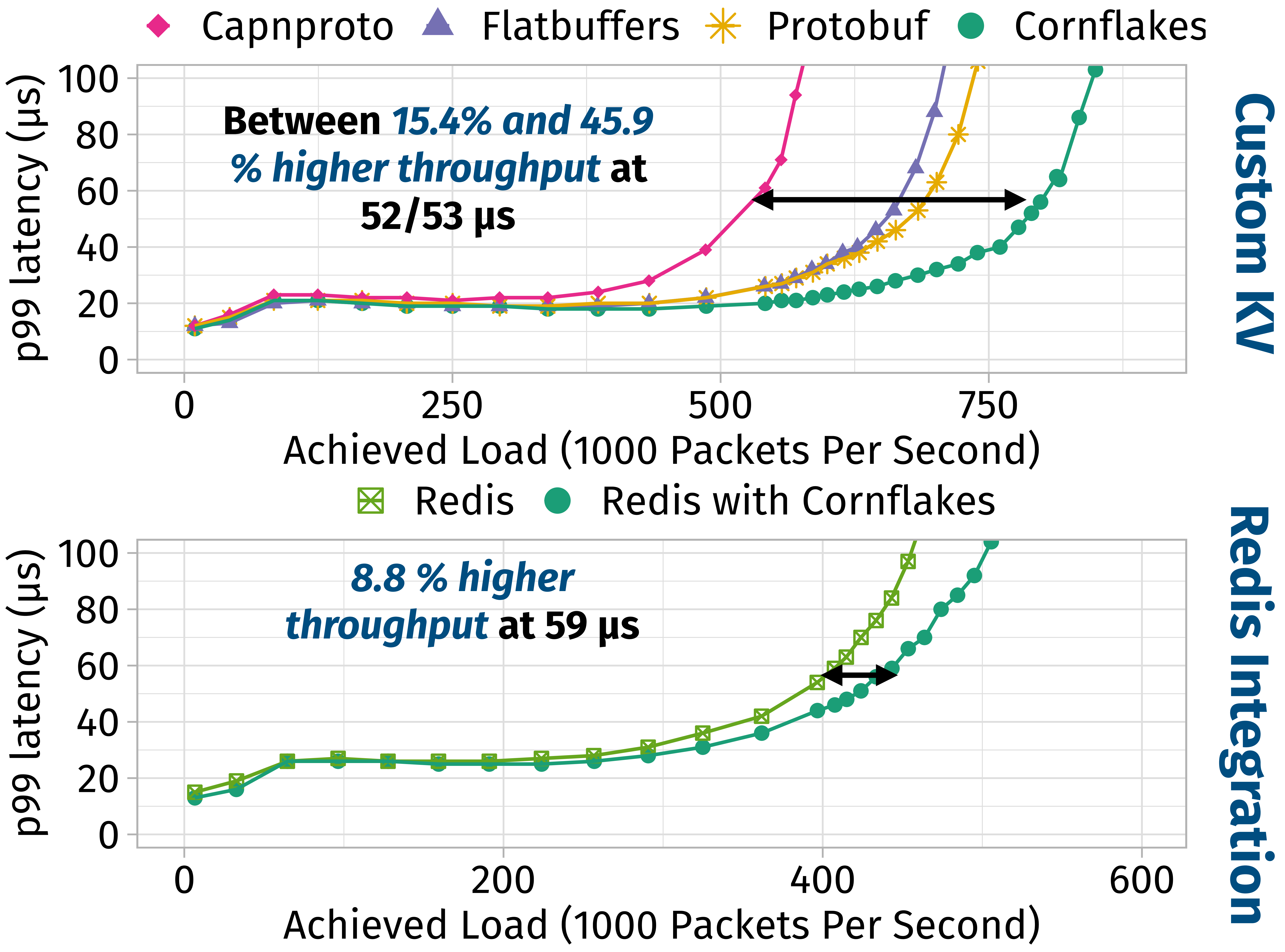
Evaluation setup

- Cloudlab c6525-100g nodes: 2-3 24-core AMD 7402P servers with Mellanox ConnectX-5 EX 100 Gb NICs, and a Dell S5296F switch
  - Jumbo frames turned on (9000 bytes)
- Cornflakes ***sustain a higher throughput***, for the ***same p99 latency***, compared to existing software serialization approaches?

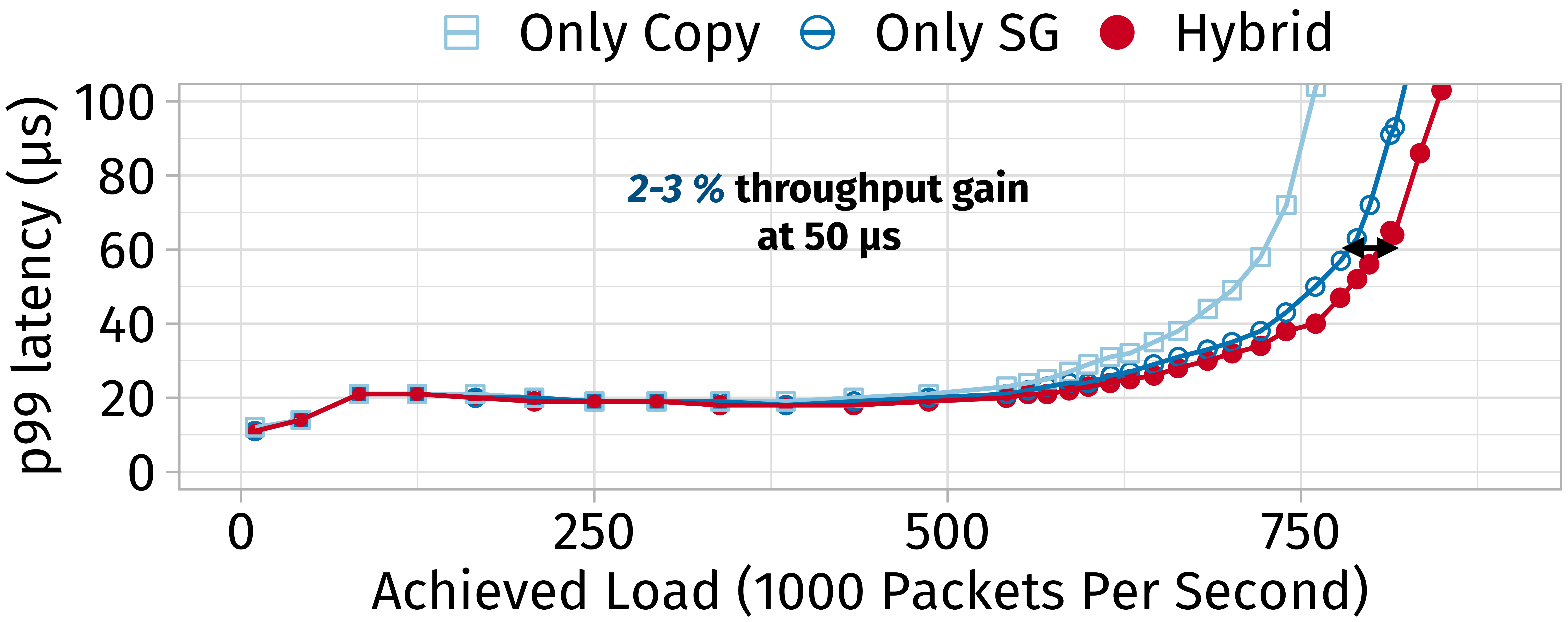
Benefits over existing approaches for Twitter cache trace

- Workload:
  - Twitter Cache Trace # 4*
- Size distribution:
  - 32 % of values queried are larger than 512 bytes

*A large scale analysis of hundreds of in-memory cache clusters at Twitter (OSDI 2020), Yang et. al.



Gains of hybrid approach for Twitter cache trace (Custom KV)



- While gains are limited here, ***hybrid allows us to use one library / deployment to serve workloads with different characteristics***

Limitations + future work

- So far, we only explored tradeoffs for certain hardware + implementations of system
  - Goal: explore measurement study on more hardware, alternate implementations of ideas

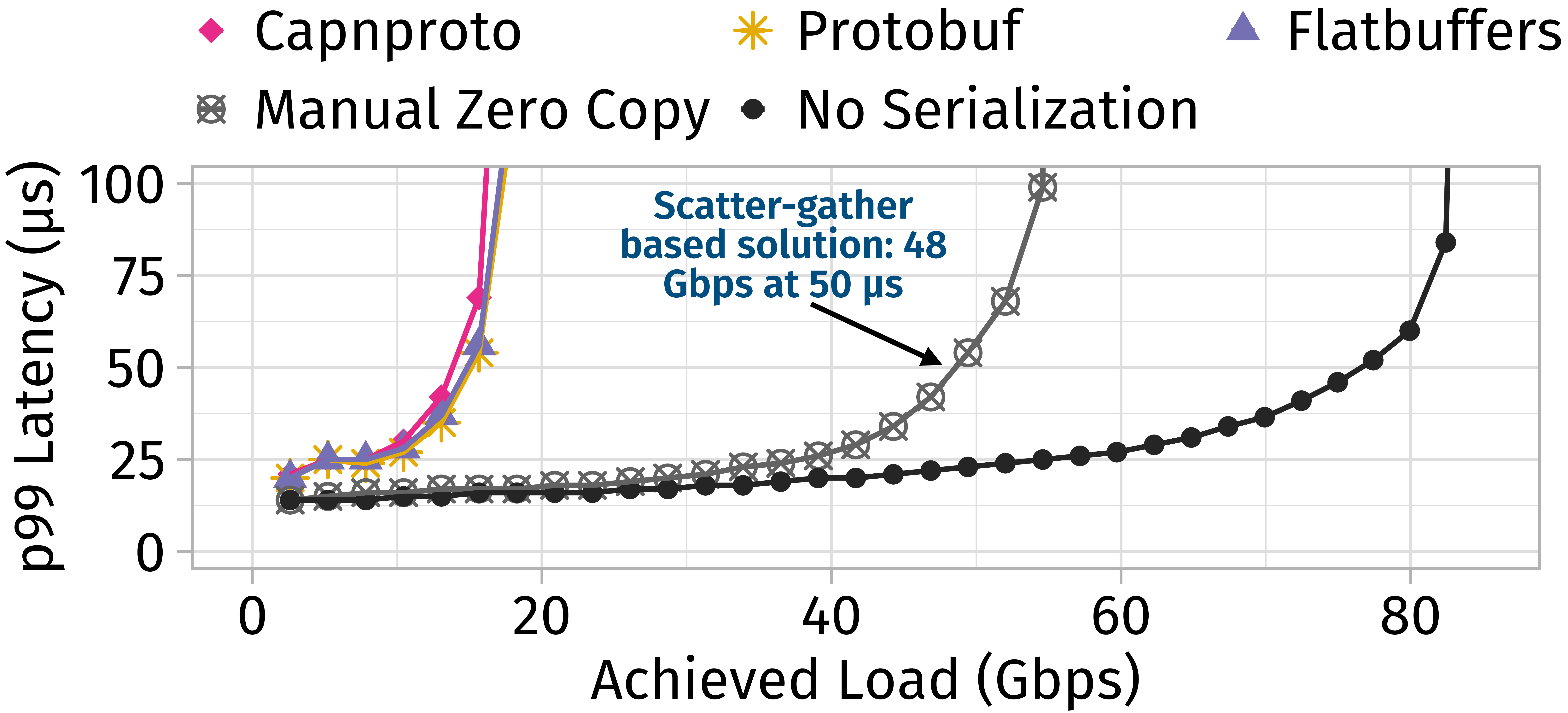
Cornflakes accelerates serialization without new hardware with NIC scatter-gather

- We learned:
  - Serialization benefits from coalescing data in NIC, but ***copying outperforms hardware offload for small buffers, due to reference count accesses***
  - Performance benefits from co-designed serialization / networking interface that ***removes extra intermediate representations of data***

- Getting started with Cornflakes:
  - Implementation: https://github.com/deeptir18/cornflakes
  - Reproducibility instructions: https://github.com/deeptir18/cornflakes-scripts

deeptir@cs.stanford.edu

Removing copies improves throughput in microsecond networks



Cornflakes compiles serialization code from protobuf schemas

1 Developer-provided schema

```
message GetM {
  optional uint32_t id;
  optional repeated bytes list;
  optional repeated string msg2;
}
```

2 Struct definition

```
struct GetM {
  id: uint32_t,
  list: Vec<CfPtr>,
  msg2: Vec<CfPtr>,
}
```

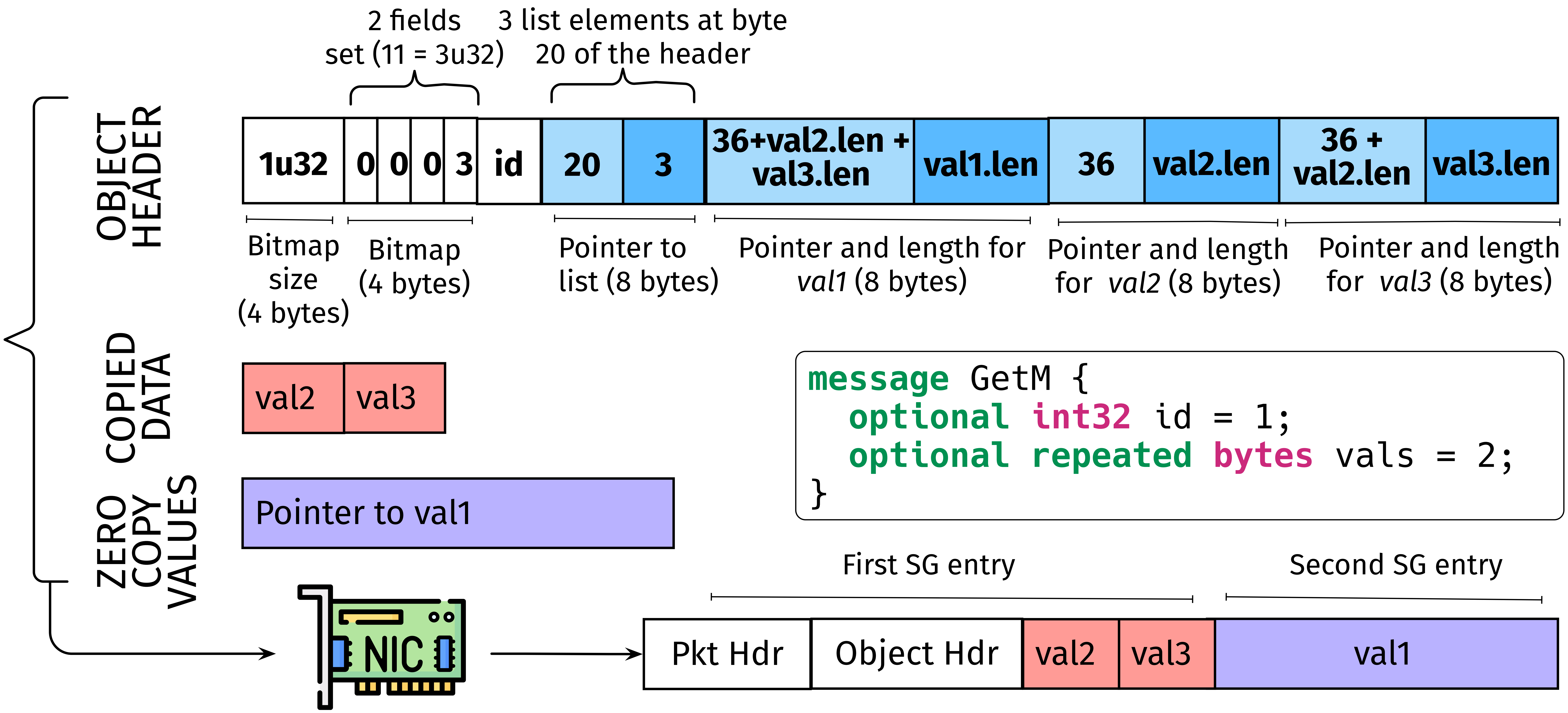
3 Struct implementation (getters/setters)

```
impl GetM {
  fn new() -> Self;
  fn init_vals(&mut self, cap: usize);
  fn get_mut_vals(&mut self) -> &mut List<CFPtr>;
}
```

4 Custom networking / serialization trait

```
impl CornflakesObj for GetM {
  fn object_len(&self) -> usize
  fn write_header(&mut self, buf: &mut [u8]);
  fn deserialize(pkt: RcBuf) -> Self
  …
}
```

Wire format for simple data structure

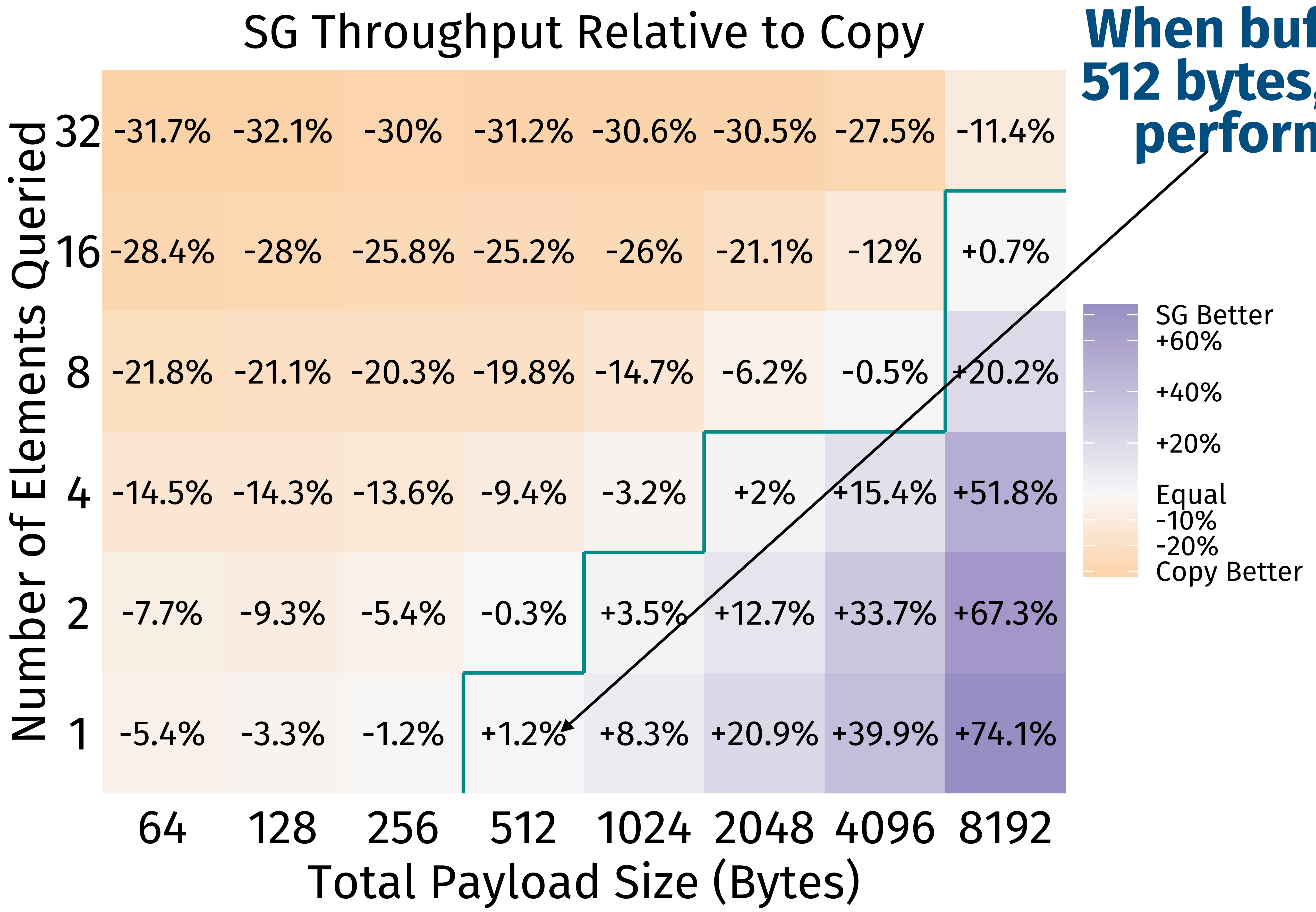


Per-hardware measurement study determines threshold

Which values should be zero-copied?

How can the hybrid API be transparent?

What API should exist between serialization and networking?



When buffers are >= 512 bytes, zero-copy performs better

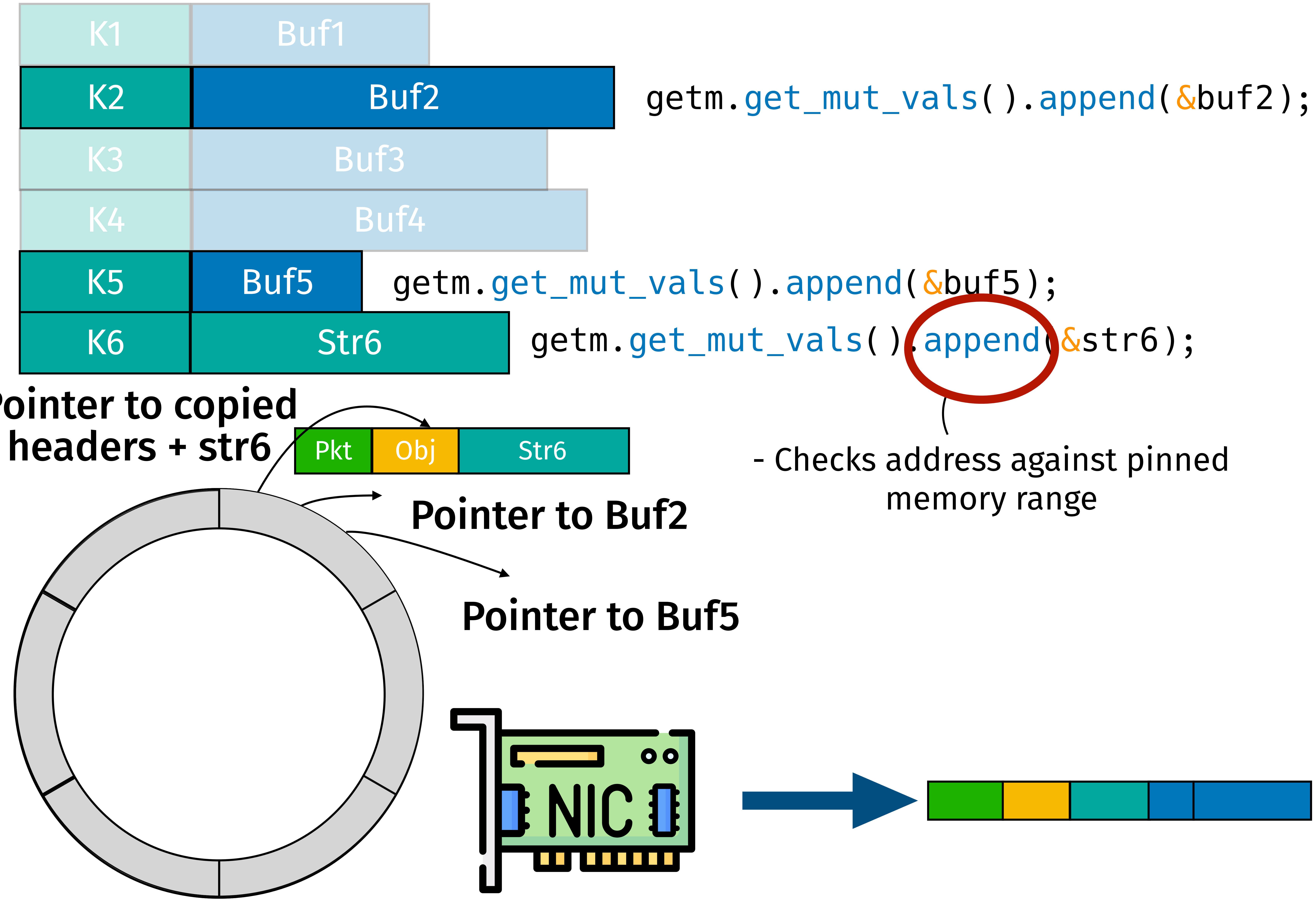
- Workload: YCSB-C workload, modified to have 1 million 30-31 byte keys mapped to lists of values of various sizes (zipf = .99)
- Metric: highest throughput achieved across all offered loads
- ***Hardware***: Mellanox CX-5 NIC, AMD EPYC 7402P Servers

Cornflakes transparently copies data when it is not NIC accessible

Which values should be zero-copied?

How can the hybrid API be transparent?

What API should exist between serialization and networking?



Serialize-and-send optimization improves throughput

	Twitter Trace	YCSB trace (1024x4), only zero-copy
W/o serialize-and-send	814 krps	16.1 Gbps
With serialize-and-send	***899 krps***	***18.9 Gbps***

Cornflakes performance falls back with predominantly small values

- Queried values contain lists of 1-8 elements (list length uniformly distributed over 1-8)
- Size of each item derived from Google Protobuf bytes size distribution measurement:
  - ***94 %*** of the elements are ***smaller than 512,*** about 33 % of the items are even less than 8 bytes