Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking

Deepti Raghavan[*]
Stanford University

Shreya Ravi
Stanford University

Gina Yuan
Stanford University

Pratiksha Thaker
Carnegie Mellon University

Sanjari Srivastava
Stanford University

Micah Murray
UC Berkeley

Pedro Henrique Penna
Microsoft Research

Amy Ousterhout
UC San Diego

Philip Levis
Stanford University and Google

Matei Zaharia
UC Berkeley

Irene Zhang
Microsoft Research

Abstract

Data serialization is critical for many datacenter applications, but the memory copies required to move application data into packets are costly. Recent zero-copy APIs expose NIC scatter-gather capabilities, raising the possibility of offloading this data movement to the NIC. However, as the memory coordination required for scatter-gather adds bookkeeping overhead, scatter-gather is not always useful. We describe Cornflakes, a hybrid serialization library stack that uses scatter-gather for serialization when it improves performance and falls back to memory copies otherwise. We have implemented Cornflakes within a UDP and TCP networking stack, across Mellanox and Intel NICs. On a Twitter cache trace, Cornflakes achieves 15.4% higher throughput than prior software approaches on a custom key-value store and 8.8% higher throughput than Redis serialization within Redis.

CCS Concepts: • **Networks → Programming interfaces**; **In-network processing**.

Keywords: data serialization, zero-copy, hardware offload

ACM Reference Format:
Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613137

1 Introduction

Data serialization [1, 2, 52–54] is on the critical path for nearly all datacenter networking. As a result, it consumes much datacenter processing time: Google reports that Protobuf consumes 9.6% of its fleetwide cycles [23][1] and Meta reports that serialization consumes 6.7% of the CPU cycles in seven important microservices [47]. As NICs increase in throughput to 400 Gbps [33] and even 1.6 Tbps [17] in the next decade, data serialization will be an increasingly expensive tax on datacenter applications. This tax will be especially challenging for ultra-low-latency datacenter stacks that process packets in a few microseconds [3, 21, 36, 43, 50] or sub-microsecond [58].

At its core, serialization requires *data movement*: software serialization libraries use the CPU to copy application data values scattered in memory into a single, contiguous buffer for I/O.[2] Experiments we present in Section 2 find that eliminating data movement in serialization could improve throughput for low-latency applications by up to 2×; these results agree with past work showing that data movement is a large part of serialization's overhead [42, 55]. Novel NIC designs [55] and accelerators [19, 23, 38] have shown benefits to offloading serialization to hardware [22, 23, 47], but custom hardware is expensive to develop and deploy.

This paper takes another approach: accelerating network serialization *with existing, commodity hardware*. Prior work has shown performance benefits from using DMA and zero-copy APIs to gather data into a contiguous region for transmission [8, 15, 21, 42, 58]. However, none have designed or demonsrated a complete serialization library for microsecond-scale applications on modern datacenter NICs.

To be easy to use and incorporate into applications, such a library must have a similar API to existing copy-based approaches, despite the fact that the library may asynchronously

[*] Work partially completed during internships at MSR.

[1] This includes serialization both for networking and storage.

[2] Some serialization libraries also encode fields (e.g., Protobuf `varint` compression); libraries like Cap'n Proto and FlatBuffers do not as long as the host uses Little Endian. All perform data movement.

SOSP '23, October 23–26, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613137

transfer application data into NIC memory. Providing a friendly API on top of asynchronous I/O of application memory introduces overheads that can easily negate the benefits of hardware offload; at timescales of hundreds of nanoseconds even small factors, such as individual cache misses, impact performance. To have a similar API to existing approaches, the library must provide two properties whose overheads must be carefully managed: memory safety and memory transparency.

Memory safety is necessary to prevent data races between the application and NIC [48, 58]. When a NIC sends data, the send is asynchronous. A race can occur if the application frees data while the NIC is accessing it. Zero-copy systems prevent these safety violations through techniques such as reference counting of message buffers, while copying-based approaches automatically provide memory safety by returning data to the application immediately.

Memory transparency refers to the library accepting data regardless of the data's location in the address space. Typically, NICs can only DMA from pages pinned physically by the kernel (so they are not swapped out). Thus a library can only zero-copy application data in pinned pages (DMA-safe memory).[3] A scatter-gather serialization stack must accept objects with pointers to arbitrary application addresses, automatically determine which fields are not in pinned memory, and copy those fields instead.

Both memory safety and transparency have software penalties that can negate the benefits of hardware offload in microsecond environments. Memory safety and transparency require the stack to access bookkeeping data structures (e.g., reference counts, pinned memory address ranges) on every I/O. These accesses can cause cache misses, adding hundreds of nanoseconds to processing times. These overheads are critical; Section 2.4 shows that while raw scatter-gather outperforms copying even for small 64-byte objects, cache misses in the scatter-gather code path cause copying to be faster.

The impact of memory coordination on performance motivates a *hybrid* serialization stack that dynamically chooses between scatter-gather and copying. Prior work observes that copying sometimes outperforms zero-copy for small buffers [6, 8, 16, 48, 58], and has varying recommendations for a tradeoff point, including 16 kB [48], 10 kB [8], and 1 kB [58]. At the microsecond timescale, designing a hybrid stack requires understanding the tradeoffs between scatter-gather and copy, including their potential software overheads, and an algorithm that efficiently chooses between the two.

In summary, a scatter-gather serialization stack should provide four properties: a standard API similar to existing approaches, zero-copy safety when directly accessing application memory, a hybrid API that transparently chooses between scatter-gather and copy, and efficient implementations of these mechanisms that enable it to match or surpass the performance of existing libraries.

This paper presents *Cornflakes*, a new, hybrid, co-designed zero-copy serialization library and networking stack that provides these four properties. In doing so, it makes two main intellectual contributions. First, a measurement study conducted on modern Mellanox NICs finds that scatter-gather I/O can improve serialization performance for buffers as small as 512 bytes (§5). Importantly, this result incorporates the overheads of memory safety and transparency. Second, the paper contributes a transparent, hybrid serialization API, which guarantees use-after-free protection for zero-copy I/O, while ensuring both code paths (copy and zero-copy) execute efficiently (§3). Cornflakes leverages both of these contributions, transparently zero-copying fields in DMA-safe memory when the fields are at least 512 bytes large.

Our evaluation of Cornflakes explores zero-copy serialization within UDP and TCP stacks, on Intel and Mellanox NICs. Cornflakes achieves within 3% of the throughput of and up to 128% better throughput than general-purpose serialization libraries in a custom key-value store, and 8.8% higher throughput than Redis's serialization when used in Redis, on a variety of realistic workloads that cover cases where scatter-gather can and cannot provide performance gains. The hybrid approach can provide between 1.4% and 14.0% improvement against an only scatter-gather approach. One particular API optimization the stack supports, serialize-and-send (§3.2.3), provides between 7.7% and 17.4% throughput gain.

2 Revisiting Serialization in the Datacenter

This section introduces our target system model (§2.1) and discusses the capabilities and limitations of scatter-gather in serialization. It reports two key findings. First, scatter-gather serialization can in some cases *significantly* outperform current copy-based libraries (§2.2). Second, given the library must provide memory safety and transparency guarantees (§2.3), scatter-gather's improvements are contingent upon the size of objects and software overheads. For smaller objects, copying costs less than safely managing an additional DMA operation (§2.4). This tradeoff motivates a *hybrid* approach that combines scatter-gather and copy.

We focus on serialization only, rather than deserialization, because existing libraries (Cap'n Proto, FlatBuffers) already provide zero-copy deserialization. They can turn serialized payloads back into in-memory data structures without data copies. We do not study accelerating encoding: while some older serialization libraries, like Protobuf, encode integers to reduce space, more recent wire formats like Cap'n Proto and FlatBuffers forgo this because the tradeoff between CPU cycles and network bandwidth has changed. Furthermore, encoding can be accelerated in hardware if needed [23, 55].

2.1 Target System Model

We assume a standard datacenter server using TCP or UDP with a 100 Gbps+ commodity NIC. We assume a low-latency

[3] Kernel bypass stacks provide pinned memory allocators for this reason.

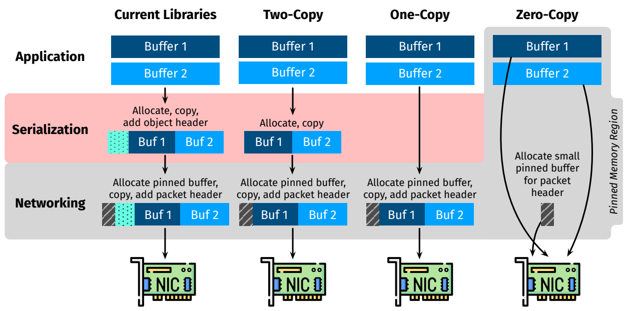


Figure 1: Four approaches to transmit two non-contiguous fields. Serialization libraries like Cap’n Proto or FlatBuffers calculate an object header and do two copies, one into a contiguous buffer and one into pinned memory. “Two-Copy” eliminates adding the header, “One-Copy” directly copies into pinned memory and “Zero-Copy” tells the NIC to make three PCIe requests to coalesce the buffers while constructing the packet, assuming the buffers live in DMA-safe memory.

kernel-bypass network stack [13, 24, 30, 34, 36, 58], which supports zero-copy I/O, such as Google’s widely deployed Snap stack [30]. We target applications benefiting from end-to-end latencies in the tens of microseconds, such as in-memory caches (e.g., Redis [44], memcached [11]), and assume that they use a low-latency stack rather than a POSIX API through the Linux kernel. Because these latencies are only relevant in the datacenter, we assume that stacks either use no encryption or a NIC encryption offload [37]. We focus on data structures that fit in a jumbo frame of about 9000 bytes to study scatter-gather for sub-MTU sized objects; our prototype implementation could be extended to support segmentation.

2.2 The Overhead of Copies in Serialization

To quantify the potential gains from scatter-gather serialization, we measure a minimal application: a simple echo server that reads and echoes a serialized message. The server has almost no application-level processing and the message has a simple format (two 2048-byte fields). We measure the performance of three commonly used serialization libraries: Protobuf [53], FlatBuffers [52], and Cap’n Proto [54]. To decompose the costs of serialization, we also measure three manual serialization approaches, shown in Figure 1.

The echo application has 16 concurrent clients sending to a single-core server, which deserializes, reserializes, and transmits the data back. All libraries use a minimal UDP networking stack built on the Mellanox OFED interface [32]. Section 6.1 provides more detail on the machines.

Figure 2 shows the results. With a <50 µs latency, the throughput without serialization is 77 Gbps, the zero-copy stack achieves 48 Gbps, and existing libraries achieve 13-15 Gbps. Data copies are the significant cost. A single copy reduces throughput to 28 Gbps, while a second copy reduces it to 23 Gbps; the second copy is less expensive because its source data is cached.

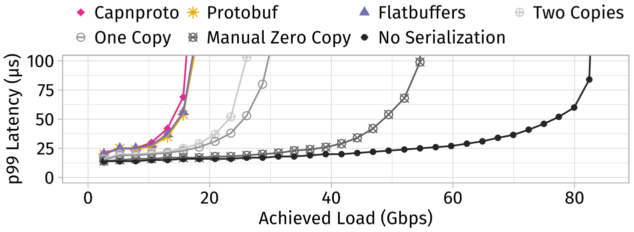


Figure 2: 99th percentile (p99) latency as achieved load increases of serialization libraries, 2-copy, 1-copy, zero-copy baselines, and no serialization. The echo server deserializes and reserializes a list with two 2048 byte elements. Even a single copy significantly reduces the achieved load for a particular p99 latency.

2.3 Safely Transmitting Application Memory

Figure 2 shows that scatter-gather can improve the performance of a serialization library. However, these results represent an upper bound on the potential performance gains. A scatter-gather serialization library that maintains a similar API as existing libraries requires *memory safety* and *memory transparency*, which both introduce software overheads.

Memory safety means that application data sent with the library must remain alive until the networking stack is done with it. The library must ensure application objects are not freed until all pending DMAs on them are completed. Systems such as Demikernel [58] and DPDK provide this “use-after-free” protection with per-application-buffer reference counts. Therefore, for each I/O and completion, the scatter-gather stack needs to access and update a reference count.

Memory transparency means that the library can handle memory in any location correctly, even though the NIC can only access pinned memory. The library should transparently support copying data that cannot be safely DMA’d, without extra programmer effort. As a result, the library needs to check each I/O address against a metadata structure containing the ranges of currently pinned memory pages.

Extra memory accesses per scatter-gather I/O causes cache misses, which add significant overhead at the packet processing timescales in Figure 2. 77 Gbps is 2.35 million packets per second from a single core, or 426 ns per packet. At 48 Gbps, a core has 683 ns per packet. A typical main memory access (an L3 cache miss) takes 100 ns; each access to uncached metadata would consume 15-23% of packet processing time.

2.4 Case for a Hybrid Approach

To understand how cache misses (which memory transparency and memory safety cause) affect the performance tradeoff between scatter-gather and copying, we ran a scatter-gather microbenchmark. Two load generators query a server containing a large array of memory, about 5× larger than L3 cache (see §6.1 for the hardware cluster). The requests contain IDs that map to non-contiguous physical buffers within the array; the server sends back the buffers concatenated

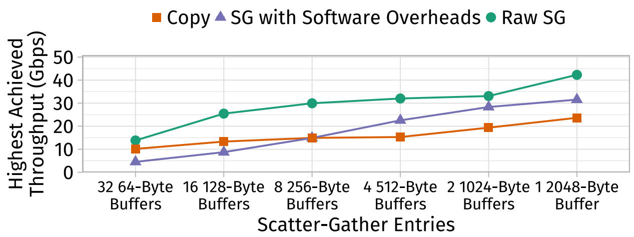


Figure 3: Highest achieved throughput for querying a 2048-byte packet assembled from 32 to 1 non-contiguous physical buffers, comparing copying, scatter-gather with software overheads, and raw scatter-gather. Raw scatter-gather strictly outperforms copying even for 64-byte buffers, but with software overheads, scatter-gather only outperforms copying for 512-byte buffers and above.

together (either using scatter-gather or copying). We compare the throughput of copying, scatter-gather with software overheads, and raw scatter-gather. In the software overheads baseline, each physical buffer has multiple virtual buffer IDs associated with it; client requests contain virtual IDs and the server accesses a reference count per virtual ID. To induce cache misses on reference count accesses, we increase the size of the reference count array to about 5× larger than the L3 cache, by increasing the number of virtual IDs per physical buffer. Real applications contain data (e.g., keys) that compete for cache space with zero-copy reference counts, causing cache misses on reference count accesses. This application has no competing data, and there are not enough physical buffers for 1 reference count per buffer to cause cache misses, so we artificially increase the size of the reference count array.

Figure 3 shows the results. Raw scatter-gather always outperforms copying, even for 32 64-byte buffers, given scatter-gather never brings the physical buffer data into the cache. With software overheads, however, the cost of the cache misses outweigh the benefits from scatter-gather for buffers smaller than 512 bytes. There are two important implications of these results. First, the stack must be *hybrid* and support both copy and zero-copy, because of zero-copy software overheads (also to support serializing data residing in non-DMA safe regions of memory). Second, software overheads are critical. The entire library must be designed around minimizing the number of cache misses it can suffer for each transmission.

3 Cornflakes Design

Cornflakes is a hybrid serialization library that is co-designed with an integrated networking stack. It provides a general-purpose serialization API that safely and transparently handles scatter-gather's accesses to application memory. It achieves this on commodity NICs in modern datacenter servers.

Cornflakes consists of three pieces: a collection of common data types, a compiler to generate code that builds custom data structures on top of these types, and a co-designed runtime library and networking stack that serializes, sends, receives and deserializes messages. Cornflakes has four design goals:

Standard, simple API. Cornflakes's programming model should be as similar as possible to programming models provided by libraries such as Protobuf [53], such that it can be incorporated into existing applications with minimal effort. Due to the NIC's asynchronous access of application memory, the programming model cannot exactly match that of Protobuf.

Transparently hybrid. Cornflakes should copy application data when it resides in non-DMA-safe memory or when zero-copy would add too much overhead, without programmer effort or knowledge.

Zero-copy safety. Cornflakes should ensure that if a program frees memory after sending it, Cornflakes can continue to use the memory. The memory will not be fully released until the transmission (and potential re-transmission) completes. Even though Cornflakes does not provide protection against concurrent sends and writes, Cornflakes's *use-after-free* guarantee applies to a wide range of applications such as Redis, versioned key-value stores, applications written in managed languages or applications written for asynchronous networking APIs; these applications naturally avoid updating data in place during sends.

Fast. Every cycle matters at timescales of microseconds, so Cornflakes must avoid creating intermediate data structures and minimize the amount of metadata it must access to guarantee memory safety and transparency.

To meet these goals, Cornflakes deviates from existing serialization libraries in three ways. First, Cornflakes *transparently supports both zero-copy and copy data* (§3.2.2). A data structure definition specifies a `bytes` or `string` field as usual, and this field can be represented as either a memory safe zero-copyable reference, or data that has been copied. Therefore, Cornflakes automatically supports non-DMA-safe memory. Second, Cornflakes *dynamically applies a per-field heuristic at assignment time* to decide which representation to try to use, based on the field size (§3.2.1). Cornflakes uses this heuristic because more complex heuristics cause additional cache misses, which are a significant overhead at microsecond timescales. Third, Cornflakes introduces a novel interface between the serialization library and networking stack: *serialize-and-send* (§3.2.3). In order to minimize intermediate representations of data that incur extra overhead, the Cornflakes networking stack directly accepts custom objects. We first discuss Cornflakes's overall programming model (§3.1) before diving deeper into these features (§3.2) and the wire format (3.3).

3.1 Cornflakes's Programming Model

At a high level, Cornflakes's programming model is nearly identical to existing serialization libraries. To use Cornflakes, a developer defines a data structure schema such as the `GetM` message shown in Listing 1, using Protobuf's existing schema language. Next, the developer invokes the Cornflakes compiler to generate objects and their implementations for the messages in the schema file (`impl GetM` in Listing 1). Finally, the developer constructs and fills in a Cornflakes object in

```
message GetM {
   optional uint32 id = 1;
   optional repeated string keys = 2;
   optional repeated bytes vals = 3;
}
impl GetM { // some functions omitted
   fn new() -> Self; // constructor
   fn init_vals(&mut self, cap: usize); // list initializer
   fn get_mut_vals(&self) -> &mut List<CFPtr>; // getter
   fn deserialize(pkt: RcBuf) -> Self; // deserializer
}

impl CornflakesObj for GetM {
   fn object_len(&self) -> usize;
   fn write_object_header(&self, &mut [u8]);
   fn iterate_over_copy_entries(&self, &mut [u8]);
   fn iterate_over_zero_copy_entries(&self, callback: Fn(RcBuf) );
}
```

Listing 1: The schema for a GetM object with a list of keys and values, with the Rust interface produced by Cornflakes's compiler. Like other libraries, the interface includes setters, getters, and deserialize functions. The interface exposes iterators that the Cornflakes networking stack uses to finish serialization, rather than an explicit serialize function.

```
pub struct RcBuf {  // Reference counted buffer
  data_pointer: *mut c_void; // pointer to beginning of allocation
  offset: usize; // offset into buffer
  len: usize; // length of reference
  refcnt: *mut Atomic<u8>; // reference count
}

impl Network {
   fn alloc(&self, size: usize) -> RcBuf;
   fn recv_packet(&self) -> RcBuf;
   fn recover_ptr(&self, ptr: &[u8]) -> Option<RcBuf>;
   fn send_object(&self, obj: impl CornflakesObj);
}
```

Listing 2: Cornflakes networking stack API. The networking stack allocates reference counted buffers (RcBuf) for applications to use, receives network data into RcBufs, and recovers RcBufs from raw pointers. The networking stack directly sends Cornflakes generated objects, rather than scatter-gather arrays.

```
pub enum CFPtr {
   Copy(Vec<u8>),
   ZeroCopy(RcBuf)
}

impl CFPtr {
   fn new(ptr: &[u8], conn: &Network) -> Self;
}
```

Listing 3: The CFPtr abstraction in Cornflakes represents either a reference counted buffer, or data directly copied into a vector. The constructor takes in raw bytes, so developers need not worry about whether the data passed in resides in a DMA-safe RcBuf or not.

```
fn handle_get(&self, pkt: RcBuf, conn: &Network) {
   let getm = GetM::deserialize(pkt);
   getm.init_vals(getm.get_keys().len());
   for key in getm.get_keys().iter() {
      let val = self.map.get(key);
      let ptr = CFPtr::new(val, conn);
      getm.get_mut_vals().append(ptr);
   }
   conn.send_object(getm);
}
```

Listing 4: Example code for an application using Cornflakes to handle a request for multiple values from a key-value store. If the values queried reside in DMA-safe memory and are larger than the size threshold, Cornflakes will send the data structure without any memory copies.

their code with the generated functions. For list fields, the compiler generates additional initialization functions that reserve space for the list (e.g., init_vals in Listing 1) and append functions (not shown). We briefly introduce two base programming primitives that Cornflakes uses to provide memory safety, memory transparency, and a hybrid API: reference counted buffers (RcBuf) and hybrid smart pointers (CFPtr), before going over an example application.

Ref-counted buffers (RcBuf). When the application, serialization library and networking stack are accessing the same memory asynchronously, higher layers must not free and reallocate memory lower layers are still accessing. Cornflakes's networking stack provides an allocator that returns buffers of reference-counted, DMA-safe memory, shown in Listing 2: RcBuf. Allocating or cloning an RcBuf increments its reference count and dropping an RcBuf decrements its reference count; Cornflakes automatically frees the underlying memory when the last outstanding reference is dropped. Existing kernel bypass systems such as Demikernel [58] and DPDK also provide reference counted, DMA-safe buffers to applications, but Cornflakes directly integrates the abstraction into a serialization library.

Hybrid smart pointers (CFPtr). To enable a hybrid stack, Cornflakes provides smart pointers that encapsulate zero-copy references to data or copied data. Cornflakes generated code represents bytes or string fields as CFPtr objects (shown in Listing 3). A CFPtr either contains a vector of bytes (copied data that will later be copied into a DMA buffer) or an RcBuf (that will be sent with an extra scatter-gather entry). The constructor of CFPtr is agnostic to the location of the underlying memory (e.g., on the stack, unpinned heap memory, middle of an RcBuf allocation); Cornflakes transparently recovers the RcBuf if the pointer passed in is part of an RcBuf allocation.

3.1.1 Example Application.

Listing 4 shows a simple example application that implements a key-value store with Cornflakes, using the GetM message defined in Listing 1, which is used both to receive data and send outgoing data. The application first deserializes an incoming GetM request and reserves space for the outgoing values in the list inside the GetM object, depending on the number of keys that will be queried (init_vals). For each key, the application queries the value, constructs a CFPtr object with the raw pointer to the corresponding value, and appends the CFPtr to the vals field within the GetM (append). Finally, the application sends the GetM directly to the networking stack (send_object); the application does not need to separately call "serialize" before sending.

3.2 Serialization Library Features

This section describes three aspects of Cornflakes designed for efficient use of scatter-gather: the scatter-gather heuristic (§3.2.1), the construction of `CFPtr` objects with this heuristic (§3.2.2), and the combined serialize-and-send API (§3.2.3).

3.2.1 Scatter-gather Heuristic. Cornflakes's dynamic decision whether to copy or zero-copy a field must be efficient. Cornflakes uses a fast-to-compute size threshold and only uses zero-copy for `bytes` and `string` fields larger than or equal to the threshold. Section 5 shows experiments that derive a threshold for our hardware platforms, 512 bytes. Cornflakes performs the threshold check when `CFPtr` data structures are constructed, to be appended into the data structure.

We explored a design that waited until the `send_object` call to choose which fields to use scatter-gather for, after all fields are set. This design allowed for heuristics that make decisions based on multiple fields or field size distributions. We found, however, that this strategy incurs both a metadata cache miss (reference count access) *and* data cache miss (data access) in certain cases. When constructing a `CFPtr` (e.g., `CFPtr::new` in Listing 3) to store inside a Cornflakes object, Cornflakes accesses and increments the reference count if the raw pointer is inside DMA-safe memory. If Cornflakes later decides to copy this field due to the heuristic, Cornflakes will also incur a data cache miss; both cache misses are a significant overhead. Therefore, Cornflakes only considers heuristics that can be calculated on a per-field basis as individual `CFPtr` objects are constructed, such as the size threshold. This way, for each field, Cornflakes incurs *either* a data cache miss (copying the data) *or* a metadata cache miss (incrementing the reference count).

3.2.2 Constructing `CFPtr`. Cornflakes uses `CFPtr` to provide its transparent, hybrid API. The `CFPtr` construction depends on whether the data should be transmitted with scatter-gather or not. The `CFPtr` constructor first executes the scatter-gather heuristic by comparing the data size to the threshold.
Copy. If the size is smaller than the threshold, the constructor copies the data into a vector and returns the `Copy` variant of `CFPtr`. The Cornflakes networking stack later copies the copy data *again* into a DMA-safe buffer. Because the data is already in cache from the first copy, the second copy is cheap. Cornflakes uses efficient *arena* allocation for the vectors inside `CFPtr` that offer fast allocation and mass deallocation in order to avoid more expensive heap allocations for copied data.
Zero-copy. For data larger than the threshold, Cornflakes first checks whether the data resides within DMA-safe (pinned) memory using the networking stack's `recover_ptr` function (Listing 2). If so, `recover_ptr` further recovers the reference count associated with the raw pointer, increments it, and constructs and returns an `RcBuf`. The reference count lookup is a map lookup and fast arithmetic operation. If memory is not within a DMA-safe region, Cornflakes cannot safely use zero-copy and instead copies the data.

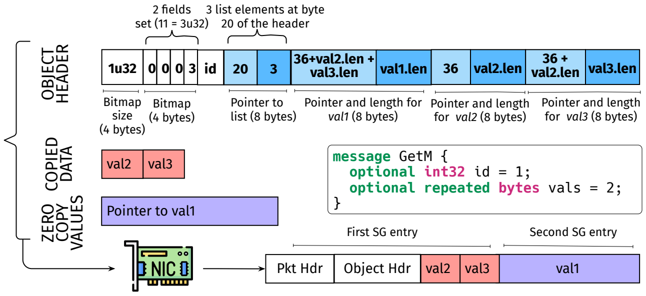
3.2.3 Combined Serialize-and-send. Cornflakes uses a combined serialize-and-send API to eliminate the cost of intermediate transformations of data that cause additional cache misses. Current scatter-gather networking stacks take scatter-gather arrays as input and internally iterate over the array to send the referenced buffers (e.g., Linux's `writev` takes as an argument a `struct iovec *iov` and an `int iovcnt`). If Cornflakes were an independent layer on top of of such a stack, it would have to transform its application object into a scatter-gather array. This transformation has a performance cost, in terms of computational cycles, memory accesses (potential cache misses), and an extra vector allocation. Cornflakes avoids this translation. The networking stack accepts generated serialization objects for transmission (`impl CornflakesObj` in Listing 1), a functionality similar to system upcalls [7]. While many Protobuf implementations expose ways to write objects directly into a socket (e.g., `SerializeToOstream` in C++), Cornflakes provides such an API without having to copy the data. This optimization is only possible in a co-designed networking stack; it is possible to implement the other aspects of the Cornflakes design (hybrid zero-copy) in a stack where serialization and networking are independent, by using scatter-gather arrays.

To implement this functionality, the networking stack uses the iterator functions in Listing 1. The networking stack first calculates the object size and number of copy and zero-copy entries, to decide how many scatter-gather entries are required. The stack calculates and writes the packet header and object header (`write_object_header`), followed by the copy entries (`iterate_over_copy_entries`). Cornflakes then calls `iterate_over_zero_copy_entries` with a callback that posts each pointer and length directly on the ring buffer. The stack concludes by marking the packet as queued onto the ring buffer. To support objects larger than a single jumbo frame, the copy and zero-copy iterators could take in start and end offsets so they only operate on entries within the specified range; the networking stack could call the iterators for each message frame until the entire object has been written.

3.3 Cornflakes Wire Format

Cornflakes's wire format is modeled off of the wire formats of Cap'n Proto and FlatBuffers, with pointers that encode the location and size of arbitrarily-sized data fields and nested objects. Overall, the Cornflakes wire format consists of a header that indexes information about the fields, followed by the actual field data. Figure 4 shows the format for a simple data structure with an integer field and a repeated `bytes` field. The header contains a `u32` indicating how large the bitmap is, a bitmap that indicates which schema fields are present, and then information for each present field (in the order specified by the schema).[4] Like Protobuf, Cornflakes assumes schema

[4] Given a `u32` represents the size of the bitmap, Cornflakes can only represent data structures with at most $2^{32} * 8$ fields.



```
message GetM {
  optional int32 id = 1;
  optional repeated bytes vals = 2;
}
```

Figure 4: Cornflakes wire format for a data structure with an integer and a repeated `bytes` field. The object header has a `u32` indicating the size of the bitmap in `u32`s (1), a bitmap indicating both fields are present (0003), the id field itself, and pointers to each list element. The final packet is constructed from two scatter-gather entries: one for the headers and copied data (`val2` and `val3`), and one for `val1`.

fields are only added and never removed or modified (so deserializers are automatically backwards compatible). For integer fields, the data is copied directly into the header (e.g., the field labeled "id" in Figure 4). For `bytes` or `string` fields, the header contains an offset (a forward pointer) and a size. For lists, the header contains a size (number of list elements), and an offset to where information about each list element is present. For nested objects, the header contains an offset to the sub-header (which contains its own bitmap and so on).

Zero-copy and copy entries. The data for the all top-level and nested ("leaf") `bytes` and `string` fields within the object follows the header; copied entries precede zero-copy entries. When transmitting a sub-MTU sized object, the PCIe request sent to the NIC will have $Z+1$ PCIe entries: Z is the number of zero-copy fields; the extra entry contains the networking header, object header, and any copied data. Figure 4 shows this in the context of a data structure with 3 list elements. Note that the order of the object pointers in the header does not necessarily match the resulting order of data fields or the order in which these fields were appended. In Figure 4, because val2 and val3 are smaller than the size threshold, they end up coming before val1 in the wire format even though val1 was appended first, because the copied data precedes the zero-copy entries. This does not preclude interoperability by two servers using different heuristic thresholds: the references to all leaf fields in the header are in a deterministic order based on the data structure schema. However, servers with different thresholds may serialize repeated elements in different orders.

Tradeoffs. The key difference from Protobuf is that all message framing (e.g., field size) is in a header at the front of the message, rather than embedded between fields. Cornflakes has similar space overheads to FlatBuffers and Cap'n Proto, which are larger than space overheads in Protobuf. Placing framing in the front requires storing a pointer and size for each field, rather than just the size. This interfaces well with the constraints of scatter-gather and PCIe. If Cornflakes embedded framing information between fields, each field would require 2 scatter-gather entries: an entry for the framing before a field, and one for the field itself. This takes up space on the ring buffer, and, depending on the NIC, could even require an extra PCIe request. By placing all of the framing upfront, there only needs to be one extra scatter-gather entry for each zero-copy field.

4 Implementation

Our prototype implementation of Cornflakes includes a code generation module that generates Rust serialization code (and C bindings) from data structure schemas (11272 LOC), a networking stack interface and base serialization primitive library in Rust (1829 LOC), and a UDP networking stack written in Rust over bindings to a custom Mellanox OFED datapath and a custom Intel ICE drivers datapath (13218 LOC combined). We additionally include a limited integration with the Demikernel [58] TCP stack (≈4300 LOC). The prototype supports serialization of base integer types, strings, bytes, nested objects, and lists of strings, bytes or nested objects; we confirmed this support with a replicated key value store application that serializes nested Protobuf objects. Our implementation uses a custom datapath for Mellanox and Intel NICs because DPDK's APIs require initializing linked lists of `mbufs` (too costly) and Cornflakes requires fast recovery of zero-copy metadata, which we found difficult to implement with the memory layout provided by DPDK's allocator. The implementation includes a pinned memory allocator as part of the Cornflakes networking stack API that allocate power-of-two-sized objects. The main UDP implementation is available at https://github.com/deeptir18/cornflakes.

4.1 Porting Existing Applications to Cornflakes

To port existing applications to Cornflakes, developers must ensure I/O data is allocated inside pinned memory, modify the serialization and I/O code to use the Cornflakes API and memory safety model, and determine an appropriate zero-copy threshold to configure Cornflakes with.

Allocation. Cornflakes expects developers to modify I/O allocation sites to explicitly allocate from pinned memory; if the data is not already directly accessible by the NIC when it is serialized, it must be copied into a DMA-safe buffer. However, the serialization library API is agnostic to the memory location and handles both cases (DMA-safe and non-DMA-safe).

Serialization. Applications must generate Cornflakes serialization code as a part of compilation. The application code must invoke the Cornflakes generated code for getting and setting fields within Cornflakes data structures, which is similar to the Protobuf API for getters and setters (other than the memory safety semantics). Finally, the code must use the asynchronous Cornflakes I/O datapath API to send objects and receive packets that can be deserialized into Cornflakes objects.

Memory safety model. Cornflakes's memory safety guarantees only support applications that avoid updating data in place during outgoing I/Os, such as Redis. Therefore, to be compatible with Cornflakes, applications must ensure that data within Cornflakes objects are not modified in-place, if they are ever sent. To do so, applications could replace in-place updates with allocations and pointer swaps (e.g., in the case of an object-store put), reducing write protection to the case of free protection. Applications that use asynchronous networking APIs like Linux zero-copy send [8] or Demikernel [58] are already compatible with this memory safety model.

Configuring Cornflakes. Finally, practitioners must configure Cornflakes with a zero-copy threshold for their hardware platform. Section 5 presents an empirical method to determine the threshold that practitioners can re-run on their own hardware to determine an appropriate threshold.

5 To Copy or Not to Copy?

This section presents a measurement study that explores the tradeoff between scatter-gather and copy in a serialization library and shows when the software overheads associated with scatter-gather outweigh the benefits. The measurement study compares two configurations of Cornflakes: one with the threshold configured to 0 (scatter-gather every `bytes` or `string` field) and one with the threshold configured to infinity (copy every `bytes` or `string` field).[5] The results indicate that scatter-gather has performance benefits when buffers are at least 512 bytes; we configure Cornflakes's zero-copy threshold to 512. Section 6.3 validates this threshold across other NICs; the current experiments only explore AMD CPUs, but future work could include more CPUs.

5.1 Experimental Setup

We conduct this measurement study using the c6525-100g CloudLab servers [10] described in §6.1.1.

Application. The measurement study uses a single-core custom key-value store implemented in Rust, where keys are `strings` and values are linked lists of DMA-safe buffers. Individual values are allocated non-contiguously and each client `get` query fetches an entire linked list. See §6.1.2 for details.

Workload. We run a workload based on the YCSB-C trace [4] (Zipf coefficient 0.99) (§6.1.4). This workload serves constant-size values, so it supports modifying the number of scatter-gather entries and entry size; any constant-size workload would work. To study how data structure shape affects the tradeoff between scatter-gather and copy, we vary the size of the buffers in the list and the length of the linked list.

Metrics. We use highest achieved throughput across all offered loads to compare scatter-gather and copy; we found this to be a good proxy to summarize trends in the corresponding throughput-latency curves.

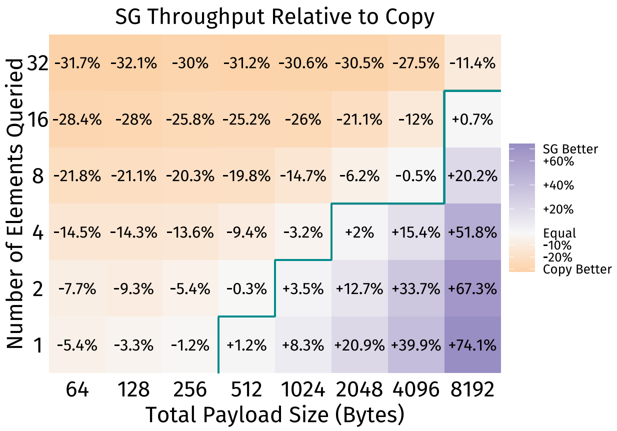


Figure 5: Throughput percent difference between only scatter-gather and only copy serialization (difference divided by copy) for a sweep of number of buffers queried and total payload size, on the YCSB key-value store workload, where client threads query linked lists from the server. On this hardware, scatter-gather only provides benefits when the individual field size is at least 512 bytes, motivating that Cornflakes should use 512 as its per-field size threshold.

5.2 Results and Implications for Cornflakes

Figure 5 shows the percent difference in maximum throughput between an all scatter-gather approach to serialization and an all copy approach, for different total response payload sizes and numbers of scatter-gather entries, on the YCSB workload. We observe that scatter-gather becomes more advantageous with larger payload sizes (x-axis) and when those payloads are comprised of fewer elements (y-axis). The green line indicates the crossover point where scatter gather starts to outperform copying. This occurs when individual fields are about 512 bytes or larger. Intuitively, this is the point at which the CPU time for a zero-copy transmission equals the CPU time for a copy transmission for a particular data size, in this implementation of Cornflakes. This does not include any time spent in the NIC, because none of the experiments so far hit the throughput or rate limitations of the NIC.[6]

Based on these results, the version of Cornflakes evaluated in Section 6 only uses scatter-gather for a field when it is at least 512 bytes large. Note that this is a lower threshold than the 1 KB, 10 KB, or 16 KB thresholds used by existing systems [8, 48, 58].

5.3 Factors Affecting the Threshold

The threshold is affected by the following factors: the cycles needed for the memory safety codepath in Cornflakes (which all zero-copy transmissions must take), the ring buffer API (the cycles needed to add an extra scatter-gather entry), and the speed of copying a certain data chunk (including any

[5] Cornflakes automatically copies integer fields directly into the object header, regardless of the copy threshold.

[6] Prior work [26] has shown that NIC characteristics can significantly impact the performance of RDMA systems; here, scatter-gather is much less resource intensive than RDMA.

prefetching that may occur). The threshold would increase if copying became cheaper (e.g., via a custom offload), or adding an extra scatter-gather entry became significantly more expensive. If the memory safety codepath became less expensive somehow, the threshold would decrease. Another factor that could affect the tradeoff is if more offloads were used in the NIC (e.g., encryption or RDMA). Given the difficulty in isolating the effects of some of these factors (e.g., the codepaths) and our lack of access to a wide variety of hardware platforms to validate with, we chose to empirically measure the threshold rather than analytically derive it. If any of the factors change, we recommend rerunning the microbenchmark (varying the number of scatter-gather entries and element size) to re-measure the threshold.

6 Evaluation

We evaluate Cornflakes to answer five questions:

1. Compared to software serialization approaches, what per-core maximum throughput and throughput for a particular p99 latency can Cornflakes sustain?
2. How do Cornflakes and other approaches perform with different distributions of field sizes in serialized messages?
3. Can Cornflakes be integrated into existing systems and what performance benefits does it provide?
4. Can Cornflakes be implemented for multiple NICs and how sensitive is its heuristic to NIC hardware?
5. How do design decisions within Cornflakes contribute to its performance?

6.1 Methodology

This section describes our hardware, applications, baselines, and metrics.

6.1.1 Setup.

Hardware. The measurement study in Section 5 and all other end-to-end experiments (Section 6) use 2 24-core AMD EPYC ROME 7402P 2.80GHz servers (Cloudlab c6525-100g servers [10]), with C-States turned off, running Linux 5.04 with Ubuntu 20.04; they have about 134 MB of L1, L2 and L3 cache. These servers are connected by Dual-port Mellanox ConnectX-5Ex 100 Gb NICs and a 100 Gb Dell Z9264 switch with jumbo frames turned on (MTU = 9000 bytes). The cycles breakdown microbenchmark in Section 6.4 uses 2 32-core AMD EPYC ROME 7452 2.35Ghz servers (Cloudlab d6515 servers) connected by Dual-port Mellanox ConnectX-5 100 Gb NICs and a 2 x 100 Gbs Dell Z9264F-ON switch. Section 6.3 compares Intel e810-CQDA2 NICs and Mellanox Connect X-6 NICs on AMD EPYC Milan 7313P 3.0Ghz servers. Section 6.3 uses a client (with a Mellanox or Intel NIC) connected back-to-back to one server (with a Mellanox or Intel NIC, respectively), without a switch.[7]

Load generator. Unless otherwise noted, all experiments use a 16-threaded client load generator running on top of DPDK, which generate requests for various offered loads for up to 25 or 30 seconds, with Poisson arrivals.[8] Unless otherwise noted, experiments report latency (at various quantiles) from a histogram that records round trip times at 1000-microsecond precision, and throughput using the total number of responses received over the total time. The client supports each serialization format compared to Cornflakes.

Metrics. We study both 99th percentile latency and achieved throughput. We show throughput-latency curves with points where achieved load is within 95% of the offered load. Not all systems achieve the same latency targets (only offered load is controlled); to compare systems (e.g., Cornflakes has X% higher throughput) we use data points where Cornflakes's tail latency is lower than that of the baselines. We additionally compare highest achieved load across all offered loads. We directly measure latency and throughput on a single core rather than measuring cycles used by each baseline as measuring cycles requires subtracting out the cycles spent busy spinning.

6.1.2 Applications.

Custom key-value store. To compare Cornflakes to general-purpose serialization, we use a custom key-value store implemented in Rust designed to be able to easily change the serialization approach, as systems such as Redis have their own serialization. Keys are `strings` and values are single buffers, linked lists of DMA-safe buffers, or vectors of DMA-safe buffers. Client `get` queries fetch values, `list` queries request the entire linked list, and `get_from_list` queries request a specific index within a particular vector.

Redis. We integrate Cornflakes into Redis [44] to compare Cornflakes to application-specific serialization. We modify three Redis commands: `get` (one value), `mget` (multiple values) and `lrange` (query from a linked list) to use Cornflakes. Given Redis by default uses Linux TCP, we modified Redis to use the Cornflakes UDP networking stack built on the Mellanox drivers, in order to fairly compare Redis and Cornflakes over the same UDP networking stack.

Echo server. The final application is an echo server, with almost no application-level processing. Clients send a serialized data structure and the server echoes it back after deserializing and reserializing it: the server has no working set and so data is always in cache.

6.1.3 Evaluated Systems.

We compare Cornflakes with three libraries: Protobuf [53], FlatBuffers [52], and Cap'n Proto [54]. We use the "protobuf" [41], "flatbuffers" [12], and "capnp" [5] Rust crates to interface with each library's in-memory objects. We also compare Cornflakes to Redis serialization within Redis [44]. All systems run over the Cornflakes UDP networking stack (using the Mellanox drivers), unless

[7] There were compatibility issues with the DAC cables used to connect the NICs to the switch, hence, we connected the servers back to back.

[8] For some workloads, we have a limited number of queries so we run for a fixed time or until the trace runs out.

otherwise specified. Within the Cornflakes networking stack, we implement a specialized network API for each library, depending on what each library produces as output, to minimize unnecessary copies in the baselines: FlatBuffers and Redis use a contiguous buffer, Cap'n Proto provides a non-contiguous list of buffers that represent the object, and Protobuf serializes from Protobuf structs into DMA-safe memory directly.

6.1.4 Workloads.

Google protobuf bytes size distribution (read-only). To evaluate a workload with mostly small objects, we build a synthetic trace generator using object sizes taken from Figure 4c of Google's fleetwide study on Protobufs [23]. Because the paper does not specify how many `bytes` fields there are per serialized object, we generate objects with a linked list of 1, 1-4, 1-8, or 1-16 fields, associated with 64-byte keys. Each field size is sampled from the distribution (and resampled if the total size exceeds an MTU). Most of the objects are small (34% of the sampled field sizes are 8 bytes or less and 94.9% are 512 or less); we expect Cornflakes should have equivalent performance to the baselines, given the hybrid solution.

Twitter cache traces (read-write). To evaluate a workload with a mixture of small and large buffers, we use the Twitter cache trace #4 [56]. The store pre-loads values for the first 4 million unique keys queried in the trace. About 32% of the requests query objects larger than 512, and about 8% of requests are put requests. Because Cornflakes's prototype UDP networking stack does not support segmentation, we break objects larger than an MTU into separate MTU and remainder sized keys and values. The requests come with timestamps at second granularity; we vary offered load by running the same distribution within smaller and smaller fractions of a second.

CDN object size distribution (read-only). To evaluate performance benefits with larger objects, we use the Tragen cache trace generator [45] to create a trace of 1 million object sizes, using the "image" trace class. The object sizes are between 1000 bytes and approximately 116 MB; the mean object size is approximately 20K bytes. We use 64-byte keys. Given the Cornflakes prototype only supports sending single-frame messages (up to a jumbo frame), our trace associates each key with a vector of sub-objects (large objects are broken up into jumbo-frame-sized segments). An individual client request queries a single sub-object, but all sub-objects within an object are requested sequentially, so we report throughput in terms of full objects received. The trace only contains 1 million requests, so we loop over the trace at various offered loads until 10 seconds has passed; though queries are repeated, we expect the key-value store to serve more requests from memory rather than cache as the objects take up approximately 10000 MB. For each offered load, we measure the achieved load and report the highest achieved load for each baseline.

YCSB (read-only). For the measurement study (§5) and to test Redis commands that necessitate more than one scatter-gather entry (`mget` and `lrange`), we run a workload based on the YCSB-C trace [4] . We generate the trace with 1 million 30-31 byte keys and 40 million Zipf distributed queries (Zipf coefficient 0.99). For Redis, the queried object has 2 2048-byte or 1 4096-byte buffers, depending on the command.

System	1 val	1-4 vals	1-8 vals	1-16 vals
Cornflakes	844.7	727.2	584.5	441.2
Protobuf	852.5	741.9	583.8	402.0
Cap'n Proto	678.9	612.9	507.1	412.0
FlatBuffers	778.1	646.4	516.8	388.9

Table 1: Throughput achieved in thousands of requests per second when querying linked lists with element sizes drawn from the Google `bytes` size distribution (≈94% are smaller than 512 bytes, so Cornflakes rarely uses scatter-gather). The length of each list is uniformly distributed across the range specified. For 1 or 1-4 values per list, Cornflakes is within 98% of the dominant baseline (Protobuf), while for 1-8 and 1-16 it outperforms all existing libraries.

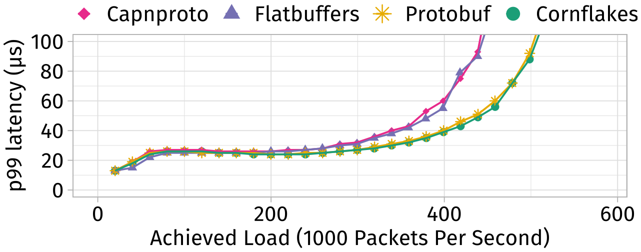


Figure 6: Throughput-tail latency tradeoff for the custom key-value store serving linked lists of 1-8 elements, with object sizes queried from the Google `bytes` size distribution. In workloads dominated by small values, Cornflakes relies on copying and performs as well as Protobuf.

6.2 Comparison to Software-Only Serialization

We first evaluate whether Cornflakes achieves its main performance goal: to achieve better throughput than software-only serialization libraries for similar p99 latencies.

6.2.1 Rust Key-Value Store.

Cornflakes can achieve similar performance to existing approaches when most of the values are small (and zero-copy won't provide any benefits) and better performance than existing approaches as the median value size of the trace increases.

Google bytes size distribution trace. Figure 6 and Table 1 show Cornflakes in comparison to the serialization baselines for the Google `bytes` distribution trace. Cornflakes does not outperform the baselines as zero-copy does not help for small objects other than the case of 1-16 values per list, but it is competitive with Protobuf, FlatBuffers and Cap'n Proto. For 16 elements, Cornflakes may beat Protobuf because the Cornflakes implementation uses arena allocation for vectors inside generated data structures, which this Protobuf implementation does not provide.

Twitter cache trace. Figure 7 shows the latency-throughput

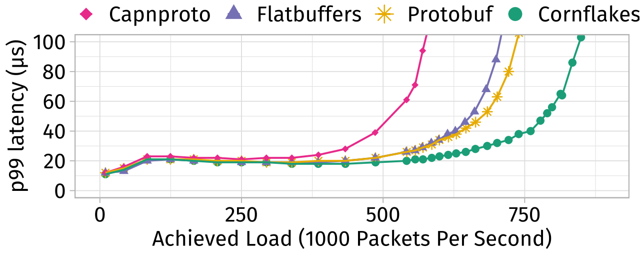


Figure 7: Throughput-tail latency tradeoff for the custom key-value store serving elements from the Twitter cache trace, where about 32% of the objects queried are 512 bytes or larger. Cornflakes outperforms all other serialization approaches: at a tail latency of about 53 µs, Cornflakes achieves 15.4% higher throughput than Protobuf.

Cap'n Proto	FlatBuffers	Protobuf	Cornflakes
161.0	181.2	186.1	366.5

Table 2: Throughput achieved in thousands of objects per second for the key-value store serving the CDN image trace. Cornflakes achieves between 97-128% higher throughput than current approaches. When large fields dominate, zero-copy greatly outperforms copying.

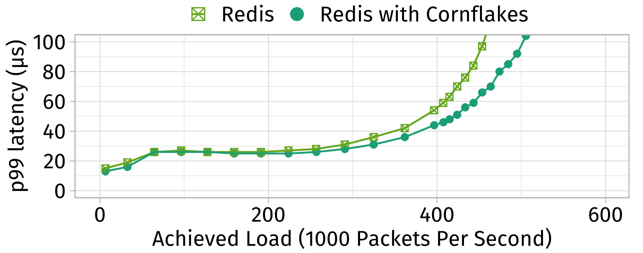


Figure 8: Throughput-tail latency tradeoff for standard Redis handwritten serialization and Redis using Cornflakes for serialization, serving requests drawn from the Twitter trace. For the tail latency SLO of 59 µs, Cornflakes achieves 8.8% higher throughput than Redis. Cornflakes can be integrated into existing systems to improve performance.

curve for the single get put requests on the Twitter cache trace workload. Cornflakes achieves between 15.4% and 45.9% higher throughput than the baselines (these percentages are based on a p99 latency of 52 µs for Cornflakes and 53 µs (FlatBuffers, Protobuf) and 61 µs (Cap'n Proto) for the baselines). Eliminating copies for larger values can allow the key-value store to more effectively use the CPU cache for smaller values.
CDN trace. Table 2 compares the highest throughput achieved on the CDN trace across Cornflakes and the software baselines. Given the minimum object size is 1 KB, which is above the threshold of 512 bytes, Cornflakes uses zero-copy exclusively. Eliminating copies not only saves CPU cycles but also leaves space in the cache for application data (e.g., keys).

6.2.2 Redis Integration.

Figure 8 shows that Cornflakes serialization improves Redis throughput by 8.8% when serving the Twitter trace. Table 3 shows a 15-40.1% improvement for a range of commands on the YCSB workload for objects with a total size of 4096 bytes; `mget-2` and `lrange-2` involve retrieving 2 non-contiguous 2048-byte buffers. Cornflakes uses zero-copy exclusively, leaving space in the cache for other Redis data such as keys. Note that the Redis baseline was modified to use the Cornflakes networking stack.

System	`get`	`mget-2`	`lrange-2`
Redis w/Cornflakes	23.37 Gbps	14.12 Gbps	15.07 Gbps
Redis	18.49 Gbps	12.27 Gbps	10.75 Gbps

Table 3: Highest throughput achieved for three commands within Redis, `get`, `mget` (2 values) and `lrange` (2 values), comparing Redis with Cornflakes serialization and Redis's own handwritten serialization, transferring payloads totaling to 4096 bytes, serving the YCSB workload. Cornflakes provides between 15 and 40.1% higher throughput. `lrange-2` sees a larger improvement than `mget-2` because details in the experimental setup cause `mget-2` to have more cache misses on keys.

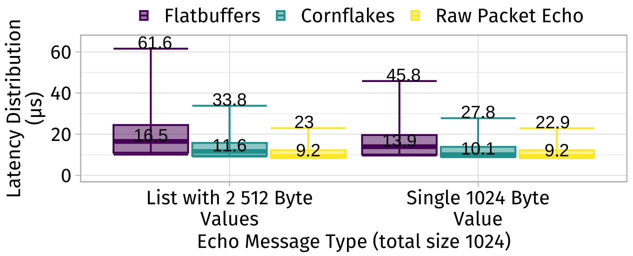
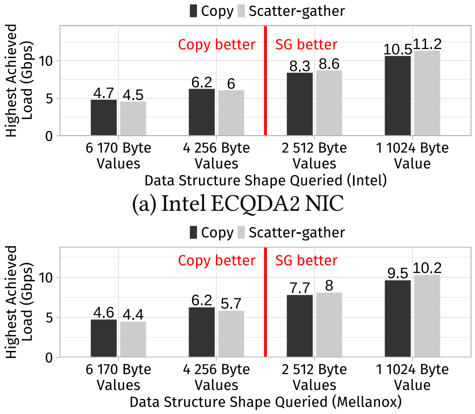


Figure 9: Latency of a TCP echo application on top of Demikernel at a load of 175,000 1024-byte requests per second, comparing serialization with FlatBuffers, serialization with Cornflakes, and a raw packet echo (an L3 forwarder without serialization). Box boundaries show p25 to p75 latencies, while whiskers show p5 and p99 latencies; p50 and p99 are labeled. Cornflakes provides between 18 µs and 27.8 µs lower tail latenies than FlatBuffers, while imposing only 4.9 – 10.8 µs more overhead than a plain packet echo. Cornflakes can support TCP and provide performance improvements compared to existing libraries.

6.2.3 TCP Integration.

Figure 9 shows latency results for integrating Cornflakes with the Demikernel [58] TCP stack for the echo server application. Raw packet echo is an L3 forwarder using Demikernel: Demikernel pops and pushes the same packet back. Here, all baselines include a separate scatter-gather entry for the packet header (which Cornflakes writes the object header into). We use a TCP load generator on top of DPDK and record the latencies for 10 seconds, after a warm-up period of 10 seconds, at a rate of 175,000 packets per second.[9] At this offered load, for a 1024-byte payload, Cornflakes provides between 18 µs and 27.8 µs lower tail latency than FlatBuffers. This is because this load approaches the

[9] We show latency here as we encountered an issue with sending at high packet rates.



(a) Intel ECQDA2 NIC

(b) Mellanox CX-6 NIC

Figure 10: Highest achieved throughput for ≈1024-byte messages on Intel and Mellanox NICs as messages are split into fewer elements, comparing only scatter-gather and only copy, on the YCSB workload. The Intel NIC supports only up to 8 scatter-gather entries; as one is needed for the message header, the final bars are for messages with 6 170-byte values. On both NICs, scatter-gather outperforms copy when values are 512-bytes or larger, showing the 512-byte threshold is consistent for two different NICs.

maximum load that FlatBuffers can sustain, but Cornflakes is able to sustain higher load.

6.3 Generality Across NICs

The heatmap in Figure 5 that informed a 512 byte copy threshold was generated with the Mellanox CX-5Ex NIC on Cloudlab. Figure 10 shows how highest achieved throughput varies between copying and scatter-gather for payloads totaling to 1024 bytes across an Intel e810 CQDA2 NIC and a Mellanox CX-6 NIC, on the YCSB workload. The Intel e810 NIC series support only up to 8 scatter-gather entries; given one is needed for the packet header, we only compare up to 6 scatter-gather elements. For both NICs, scatter-gather outperforms copying with 1 1024-byte value and 2 512-byte values; the threshold of 512 bytes holds across both NICs.

6.4 Cornflakes Overheads

This section shows where Cornflakes spends CPU cycles. Figure 11 breaks down the average cycles for different parts of request handling within the CDN trace, for Cornflakes, FlatBuffers and Protobuf, for the achieved load of about 100K objects per second.[10] Given the minimum object size is 1K bytes, Cornflakes always uses zero-copy. Applying zero-copy leaves more cache space for application keys, also causing get operations to complete faster. Cornflakes deserialization is shorter because Cornflakes defers utf-8 verification for the `string` key field until the key field is accessed; the other baselines perform utf-8 verification at deserialization time.

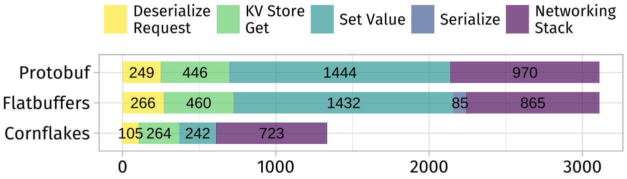


Figure 11: Average cycles taken for various steps within the key-value store, serving the CDN workload, averaged over 2 executions, at a rate of about 100K objects per second, where all systems meet the offered load. Cornflakes avoids a copy into the serialization data structure during "Set Value" and only needs to access a reference count, while FlatBuffers and Protobuf copy once into the serialization structure ("Set Value") and once into a networking buffer (within "Networking Stack")

.

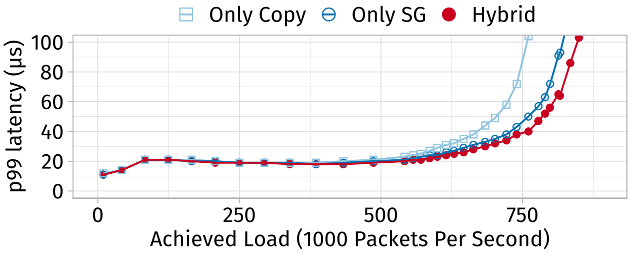


Figure 12: Throughput-tail latency tradeoff on the Rust key-value store, serving the Twitter cache trace. Cornflakes's hybrid approach achieves 2.3-3.9% higher throughput than the only scatter-gather approach at about 50 µs and 3.1% improvement for the highest achieved throughput overall (not visible). For workloads containing a mix of sizes, Cornflakes's hybrid approach provides performance benefits.

System	1 val	1-4 vals	1-8 vals	1-16 vals
Hybrid	844.7	<u>727.2</u>	<u>584.5</u>	<u>441.2</u>
Only SG	<u>846.9</u>	717.2	557.4	387.1
Only Copy	819.0	693.2	547.4	411.8

Table 4: Highest throughput achieved in thousands of requests per second on the Rust key-value store, serving the Google `bytes` size distribution workload. The hybrid approach imposes a 0.3% overhead in the case of 1 scatter-gather entry, while outperforming scatter-gather by 1.4-14.0% when response have more values. Cornflakes allows applications to leverage one library, regardless of the workload.

6.5 Cornflakes's Design Decisions

This section explores Cornflakes's size based hybrid threshold and its send and serialize API.

6.5.1 Cornflakes Hybrid Size Threshold.

Figure 12 compares the throughput-latency curves for hybrid, only scatter-gather, and only copy on the Twitter workload; the hybrid approach performs about 2.3-3.9% better than only scatter-gather in terms of throughput at the latency SLO of approximately 50 µs. On the Google workload, Table 4 shows that hybrid outperforms only scatter-gather by between 1.4% and

[10] Note this experiment uses a different machine type, Cloudlab d6515, from Table 2, which uses the Cloudlab c6525-100g type.

	Google	Twitter	YCSB
	1-4 vals		1024x4
With serialize-and-send	727 krps	899 krps	18.9 Gbps
W/o serialize-and-send	675 krps	814 krps	16.1 Gbps

Table 5: Highest throughput achieved by Cornflakes with and without the combined serialize-and-send optimization. The optimization increases throughput by between 7.7-17.4% on different workloads, suggesting it is crucial to squeeze the best performance out of the scatter-gather hardware.

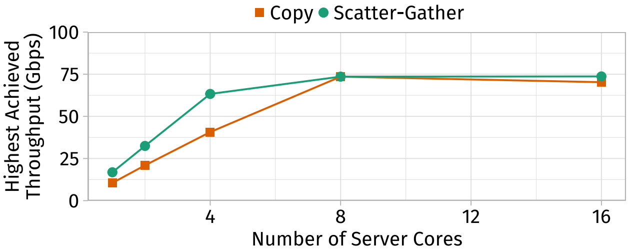


Figure 13: Highest achieved throughput on a scatter-gather microbenchmark comparing copy raw scatter-gather as the number of cores increase. Clients send requests querying 2 non-contiguous 512-byte segments from an array that is about 10 × larger than the size of L3 cache. The array is sharded across the number of cores specified. The behavior indicates that scatter-gather performance scales linearly as the number of cores increase.

14.0% when responses have more than 1 scatter-gather entry. These results demonstrate two benefits that Cornflakes's hybrid approach provides. First, users can leverage one serialization library regardless of the workload, rather than needing to switch between a copy-only library and a scatter-gather-only library, depending on the object size. Second, for workloads that mix buffers of different sizes, Cornflakes provides better performance overall.

6.5.2 Combined Serialize-And-Send. Cornflakes's combined serialize-and-send API allows it to achieve between 7.7% and 17.4% better throughput on various workloads, as shown in Table 5. Without serialize-and-send, Cornflakes allocates a scatter-gather array that contains the object header and copied data into a DMA-safe buffer the first entry, and the zero-copy data in further entries. The networking stack transmits the array, placing an extra scatter-gather entry at the front for the packet header. Combined serialize-and-send allows Cornflakes to avoid materializing an intermediate scatter-gather array and avoid one less scatter-gather entry (with the optimization, Cornflakes can place the object header and copied data in the scatter-gather entry with the network header).

6.6 Discussion on Scalability

Realistic applications benefit from parallelizing work across cores. To understand Cornflakes's potential to scale, we ran the microbenchmark from Section 2.4, comparing copy and raw scatter-gather, on a server that shards the full application memory across cores, with a working set that is about 10× larger than the L3 cache, and a message shape of two 512-byte buffers. Figure 13 shows that scatter-gather throughput starts at 16.8 Gbps and scales linearly with the number of cores for 1, 2, and 4 cores, before plateauing at about 73.5 Gbps. Copy throughput starts at 10.5 Gbps and similarly scales linearly with the number of cores; the corresponding throughput at higher number of cores is about 33% lower than scatter-gather throughput, until both systems plateau. The fact that both systems scale linearly with the number of cores suggests that our end-to-end results should extrapolate to multiple cores. However, given the complexity of carefully optimizing even a single-core implementation, we leave a full multicore implementation to future work.

7 Limitations and Future Work

Memory safety. There is a tradeoff between different levels of memory protection, transparency and efficiency; Cornflakes optimizes for transparency and efficiency, at the cost of less protection than other methods. Cornflakes does not prevent the application from modifying memory during sends. Previous work such as zIO [48] provides transparent write protection by calling `mprotect` on data that is sent; adding the overhead of a system call to each Cornflakes transmission would likely make zero-copy more expensive and push the zero-copy threshold higher than 512 bytes. Userspace control of page permissions (via Intel memory protection keys) could help control write access as data is sent [35], but would require careful consideration to integrate safely and efficiently. Cornflakes could provide a library of smart pointers for developers where writes to the smart pointer automatically trigger new allocations and raw pointer swaps, but this would require more developer effort.

Static zero-copy threshold. Cornflakes relies on a per-system threshold to determine the field size at which Cornflakes adds an extra zero-copy entry to the transmission. However, the threshold could change dynamically based on the current memory bandwidth pressure (e.g., due to additional applications running). With higher memory bandwidth pressure, adding extra PCIe requests could become more expensive. If Cornflakes automatically monitored the cache and memory bandwidth pressure and adjusted the threshold dynamically the threshold could both become more application-specific and work in multitenant environments.

Allocation of pinned memory. Typically, to use zero-copy datapaths, applications must modify all I/O allocation sites to use a specific pinned memory allocator that allocates DMA-safe memory, so zero-copy is possible. Making memory allocators aware of DMA-safe memory would make it easier for developers to port existing applications or write new data structures that are automatically allocated within DMA-safe memory (e.g., allocating Rust or C++ collection data structures in pinned memory directly).

8 Related Work

Zero-copy. Cornflakes builds on prior work that applies zero-copy to applications and studies the tradeoffs between copy and zero-copy. zIO [48] transparently removes application copies by interposing on `memmove` and `memcpy` and handling memory safety via page faults. However, zIO mostly studies packet sizes larger than 8000 bytes; Cornflakes shows zero-copy gains for transmitting sub-MTU packets. Linux proposes a zero-copy API [8, 9], but this implementation is not optimized for microsecond-scale applications [8]. Goldenberg et al. [16] use the Sockets Direct API, which has zero-copy and copy modes, to study when zero-copy is beneficial. Chu studies the performance benefits from zero-copy within Solaris [6] and finds it depends on application behavior. Cornflakes makes similar observations about when zero-copy achieves better performance, in the context of modern hardware and serialization workloads specifically. Cornflakes measures a lower zero-copy threshold on the hardware platforms evaluated so far than previous systems (zIO suggests 16kB and the Linux kernel suggests 10kB) because the tradeoffs for zero-copy in these systems differ (e.g., the Linux kernel pins memory on demand and zIO provides write protection with `mprotect`). Demikernel [58] suggests a slightly higher threshold of 1024 bytes but did not show an in-depth measurement study to determine this value.

Serialization acceleration. Many serialization libraries reduce overhead by optimizing the wire format [52, 54], employing SIMD parallelism for decoding [28], or minimizing the cost of type inference in dynamic serialization [27, 31]. These approaches do not remove the fundamental cost of in-memory copies. As a result, recent research offloads serialization or RPC to custom accelerators [19, 23, 29, 38, 39, 55] or directly within SSDs [51]. Unlike these custom accelerators, Cornflakes uses existing functionality in widely used NICs.

Reducing data movement in serialization. A recent proposal for a custom Protobuf hardware [23] accelerates the transformation between the Protobuf in-memory format and the Protobuf wire format for `C++` Protobuf applications. However, applications still move data from in-memory data structures to Protobuf objects unless Protobuf objects are used as mutable application state and manually provide DMA-safe output buffers to the accelerator. Zerializer [55] proposes a custom NIC offload to accelerate serialization. Cornflakes only requires commodity NICs and does not require any data conversions, because its chosen wire format does not require encoding integers. Naos [49] is a recent "serialization-free" runtime that uses one-sided RDMA writes to individually write data fields of large Java objects directly into the receiver's memory. In contrast, Cornflakes works across programming languages and does not require RDMA support.

Kernel bypass systems. Cornflakes requires kernel bypass APIs that expose NIC interfaces directly to applications in userspace [18, 43, 50] to eliminate OS level packet processing overheads. Many recent kernel bypass networking stacks [13, 34, 36, 40, 57] build on top of these interfaces to provide APIs to applications while offering low latency, optimized thread scheduling, or zero-copy I/O. eRPC [21] offers general-purpose RPC for commodity networking hardware, and zero-copy networking. Demikernel and Arrakis [36, 58] are ideal for use with Cornflakes because they expose scatter-gather APIs directly to applications; however, none of these systems have support for serialization.

Scatter-gather capabilities. High-performance computing applications have used scatter-gather to optimize MPI all-to-all communication primitives [14], or provide zero-copy communication over MPI derived datatypes [46]. Kesavan et al. [25] use scatter-gather to measure when zero-copy helps an in-memory database, but does not consider serialization of arbitrary data structures. Derecho [20], a recent SMR system, uses scatter-gather to provide zero-copy I/O for scattered data structures, but relies on specific layouts of data structures provided by their memory allocator. We propose designing general-purpose serialization library for application data in arbitrary memory layouts.

9 Conclusion

This paper describes Cornflakes, a microsecond-scale serialization library that uses NIC scatter-gather to eliminate in-memory copies when it improves performance. Evaluation results show that Cornflakes accelerates serialization, with the greatest benefits for applications that send large application data objects. At the timescales of hundreds of nanoseconds per packet, achieving these results while safely accessing application memory required carefully optimizing Cornflakes's entire stack and data structures: a single cache miss can be the difference between small and significant performance gains.

10 Acknowledgements

We thank our shepherd, Edouard Bugnion, the anonymous OSDI 2022, OSDI 2023, and SOSP reviewers, Adam Belay, Akshay Narayan, Anuj Kalia, Dan Ports, Hudson Ayers, Hugo Sadok, Jacob Nelson, John Ousterhout, Josh Fried, Keith Winstein, Sadjad Fouladi, Sagar Karandikar, and members of the Stanford Future Data and Sing Research groups for their invaluable feedback and conversations that have greatly improved this work. We thank Sadjad Fouladi for assistance with the diagrams, and Obi Nnorom for assistance trying the artifact evaluation instructions. We thank Intel for their donation of the e810-series NICs and Jesse Brandeburg and Brian Johnson for helping us getting started with these NICs. We thank Cloudlab [10] for their hardware clusters and technical support. This research was supported in part by affiliate members and other supporters of the Stanford Platform Lab and the Stanford DAWN Project, as well as the NSF under Career Grant CNS-1651570, Graduate Research Fellowship Grant DGE-1656518, and Grant No. 1931750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

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

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

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI*, 2014.

[4] brianfrankcooper. Ycsb. https://github.com/brianfrankcooper/YCSB.

[5] Cap'n Proto Schema Compiler Plugin Library. https://docs.capnproto-rust.org/capnpc/.

[6] H.-k. J. Chu et al. Zero-copy tcp in solaris. In *USENIX Annual Technical Conference*, volume 1, pages 253–264, 1996.

[7] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, 1985.

[8] J. Corbet. https://lwn.net/articles/726917/. https://lwn.net/Articles/726917/, 2017.

[9] W. de Bruijin and E. Dumazet. sendmsg copy avoidance with MSG_ZEROCOPY , 2017.

[10] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[11] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004.

[12] Crate flatbuffers. https://docs.rs/flatbuffers/latest/flatbuffers/.

[13] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[14] A. Gainaru, R. L. Graham, A. Polyakov, and G. Shainer. Using infiniband hardware gather-scatter capabilities to optimize mpi all-to-all. In *EuroMPI 2016*, 2016.

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

[16] D. Goldenberg, M. Kagan, R. Ravid, and M. Tsirkin. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In *13th Symposium on High Performance Interconnects (HOTI'05)*, 2005.

[17] IEEE Standards Association Working Groups. IEEE P802.3df Defines Architecture Holistically to Achieve 800 Gb/s and 1.6 Tb/s Ethernet. https://standards.ieee.org/beyond-standards/ieee-p802-3df-defines-a-holistic-architectural-approach/, 2023.

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

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

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

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

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

[23] 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*, 2021.

[24] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, 2019.

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

[26] X. Kong, Y. Zhu, H. Zhou, Z. Jiang, J. Ye, C. Guo, and D. Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

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

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

[29] 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 *ASPLOS*, 2021.

[30] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

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

[32] Welcome to MLNX_OFED InfiniBand/VPI. https://docs.nvidia.com/networking/category/mlnxofedib, 2022.

[33] NVIDIA Corporation. ConnectX-7 Ethernet Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2023.

[34] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, 2019.

[35] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

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

[37] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir. Autonomous nic offloads. In *ASPLOS*, 2021.

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

[39] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO*, 2021.

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

[41] Crate protobuf. https://docs.rs/protobuf/latest/protobuf/.

[42] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang. Breakfast of champions: Towards zero-copy serialization with nic scatter-gather. In *HotOS*, 2021.

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

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

[45] A. Sabnis and R. K. Sitaraman. Tragen: A synthetic trace generator for realistic cache simulations. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, 2021.

[46] G. Santhanaraman, J. Wu, W. Huang, and D. K. Panda. Designing zero-copy message passing interface derived datatype communication over infiniband: Alternative approaches and performance evaluation. *The International Journal of High Performance Computing Applications*, 2005.

[47] A. Sriraman and A. Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *ASPLOS*, 2020.

[48] T. Stamler, D. Hwang, A. Raybuck, W. Zhang, and S. Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[49] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler. Naos: Serialization-free RDMA networking in java. In *Usenix Annual Technical Conference*, 2021.

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

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

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

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

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

[55] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé. Zerializer: Towards zero-copy serialization. In *HotOS*, 2021.

[56] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

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

[58] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *SOSP*, 2021.