Datacenter RPCs can be General and Fast

Anuj Kalia Michael Kaminsky† David G. Andersen
Carnegie Mellon University †*Intel Labs*

Abstract

It is commonly believed that datacenter networking software must sacrifice generality to attain high performance. The popularity of specialized distributed systems designed specifically for niche technologies such as RDMA, lossless networks, FPGAs, and programmable switches testifies to this belief. In this paper, we show that such specialization is not necessary. eRPC is a new general-purpose remote procedure call (RPC) library that offers performance comparable to specialized systems, while running on commodity CPUs in traditional datacenter networks based on either lossy Ethernet or lossless fabrics. eRPC performs well in three key metrics: message rate for small messages; bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, congestion, and background request execution. In microbenchmarks, one CPU core can handle up to 10 million small RPCs per second, or send large messages at 75 Gbps. We port a production-grade implementation of Raft state machine replication to eRPC without modifying the core Raft source code. We achieve 5.5 µs of replication latency on lossy Ethernet, which is faster than or comparable to specialized replication systems that use programmable switches, FPGAs, or RDMA.

1 Introduction

"Using performance to justify placing functions in a low-level subsystem must be done carefully. Sometimes, by examining the problem thoroughly, the same or better performance can be achieved at the high level."

— End-to-end Arguments in System Design

Squeezing the best performance out of modern, high-speed datacenter networks has meant painstaking specialization that breaks down the abstraction barriers between software and hardware layers. The result has been an explosion of co-designed distributed systems that depend on niche network technologies, including RDMA [18, 25, 26, 38, 50, 51, 58, 64, 66, 69], lossless networks [39, 47], FPGAs [33, 34], and programmable switches [37]. Add to that new distributed protocols with incomplete specifications, the inability to reuse existing software, hacks to enable consistent views of remote memory—and the typical developer is likely to give up and just use kernel-based TCP.

These specialized technologies were deployed with the belief that placing their functionality in the network will yield a large performance gain. In this paper, we show that a general-purpose RPC library called eRPC can provide state-of-the-art performance on commodity datacenter networks without additional network support. This helps inform the debate about the utility of additional in-network functionality vs purely end-to-end solutions for datacenter applications.

eRPC provides three key performance features: high message rate for small messages; high bandwidth for large messages; and scalability to a large number of nodes and CPU cores. It handles packet loss, node failures, congestion control, and long-running background requests. eRPC is *not* an RDMA-based system: it works well with only UDP packets over lossy Ethernet without Priority Flow Control (PFC), although it also supports InfiniBand. Our goal is to allow developers to use eRPC in unmodified systems. We use as test-cases two existing systems: a production-grade implementation of Raft [14, 54] that is used in Intel's distributed object store [11], and Masstree [49]. We successfully integrate eRPC support with both without sacrificing performance.

The need for eRPC arises because the communication software options available for datacenter networks leave much to be desired. The existing options offer an undesirable trade-off between performance and generality. Low-level interfaces such as DPDK [24] are fast, but lack features required by general applications (e.g., DPDK provides only unreliable packet I/O.) On the other hand, full-fledged networking stacks such as mTCP [35] leave significant performance on the table. Absent networking options that provide both high performance and generality, recent systems often choose to design and implement their own communication layer using low-level interfaces [18, 25, 26, 38, 39, 55, 58, 66].

The goal of our work is to answer the question: Can a general-purpose RPC library provide performance comparable to specialized systems? Our solution is based on two key insights. First, we optimize for the common case, i.e., when messages are small [16, 56], the network is congestion-free, and RPC handlers are short. Handling large messages, congestion, and long-running RPC handlers requires expensive code paths, which eRPC avoids whenever possible. Several eRPC components, including its API, message format, and wire protocol are optimized for the common case. Second, restricting each flow to at most one bandwidth-delay product (BDP) of outstanding data effectively prevents packet loss caused by switch buffer overflow for common traffic patterns. This is because datacenter switch buffers are much larger than the network's BDP. For example, in our two-layer testbed that resembles real deployments, each switch has 12 MB of dynamic buffer, while the BDP is only 19 kB.

eRPC (*efficient* RPC) is available at https://github.com/efficient/eRPC. Our research contributions are:

1. We describe the design and implementation of a high-performance RPC library for datacenter networks. This includes (1) common-case optimizations that improve eRPC's performance for our target workloads by up to 66%; (2) techniques that enable zero-copy transmission in the presence of retransmissions, node failures, and rate limiting; and (3) a scalable implementation whose NIC memory footprint is independent of the number of nodes in the cluster.
2. We are the first to show experimentally that state-of-the-art networking performance can be achieved without lossless fabrics. We show that eRPC performs well in a 100-node cluster with lossy Ethernet without PFC. Our microbenchmarks on two lossy Ethernet clusters show that eRPC can: (1) provide 2.3 µs median RPC latency; (2) handle up to 10 million RPCs per second with one core; (3) transfer large messages at 75 Gbps with one core; (4) maintain low switch queueing during incast; and (5) maintain peak performance with 20000 connections per node (two million connections cluster-wide).
3. We show that eRPC can be used as a high-performance drop-in networking library for existing software. Notably, we implement a replicated in-memory key-value store with a production-grade version of Raft [14, 54] without modifying the Raft source code. Our three-way replication latency on lossy Ethernet is 5.5 µs, which is competitive with existing specialized systems that use programmable switches (NetChain [37]), FPGAs [33], and RDMA (DARE [58]).

2 Background and motivation

We first discuss aspects of modern datacenter networks relevant to eRPC. Next, we discuss limitations of existing networking software that underlie the need for eRPC.

2.1 High-speed datacenter networking

Modern datacenter networks provide tens of Gbps per-port bandwidth and a few microseconds round-trip latency [73, §2.1]. They support polling-based network I/O from userspace, eliminating interrupts and system call overhead from the datapath [28, 29]. eRPC uses userspace networking with polling, as in most prior high-performance networked systems [25, 37, 39, 56].

eRPC works well in commodity, lossy datacenter networks. We found that restricting each flow to one BDP of outstanding data prevents most packet drops even on lossy networks. We discuss these aspects below.

Lossless fabrics. Lossless packet delivery is a link-level feature that prevents congestion-based packet drops. For example, PFC for Ethernet prevents a link's sender from overflowing the receiver's buffer by using pause frames. Some datacenter operators, including Microsoft, have deployed PFC at scale. This was done primarily to support RDMA, since existing RDMA NICs perform poorly in the presence of packet loss [73, §1]. Lossless fabrics are useful even without RDMA: Some systems that do not use remote CPU bypass leverage losslessness to avoid the complexity and overhead of handling packet loss in software [38, 39, 47].

Unfortunately, PFC comes with a host of problems, including head-of-line blocking, deadlocks due to cyclic buffer dependencies, and complex switch configuration; Mittal et al. [53] discuss these problems in detail. In our experience, datacenter operators are often unwilling to deploy PFC due to these problems. Using simulations, Mittal et al. show that a new RDMA NIC architecture called IRN with improved packet loss handling can work well in lossy networks. Our BDP flow control is inspired by their work; the differences between eRPC's and IRN's transport are discussed in Section 5.2.3. Note that, unlike IRN, eRPC is a real system, and it does not require RDMA NIC support.

Switch buffer $\gg$ BDP. The increase in datacenter bandwidth has been accompanied by a corresponding decrease in round-trip time (RTT), resulting in a small BDP. Switch buffers have grown in size, to the point where "shallow-buffered" switches that use SRAM for buffering now provide tens of megabytes of shared buffer. Much of this buffer is dynamic, i.e., it can be dedicated to an incast's target port, preventing packet drops from buffer overflow. For example, in our two-layer 25 GbE testbed that resembles real datacenters (Table 1), the RTT between two nodes connected to different top-of-rack (ToR) switches is 6 µs, so the BDP is 19 kB. This is unsurprising: for example, the BDP of the two-tier 10 GbE datacenter used in pFabric is 18 kB [15].

In contrast to the small BDP, the Mellanox Spectrum switches in our cluster have 12 MB in their dynamic buffer pool [13]. Therefore, the switch can ideally tolerate a 640-way incast. The popular Broadcom Trident-II chip used in datacenters at Microsoft and Facebook has a 9 MB dynamic buffer [9, 73]. Zhang et al. [70] have made a similar observation (i.e., buffer $\gg$ BDP) for gigabit Ethernet.

In practice, we wish to support approximately 50-way incasts: congestion control protocols deployed in real datacenters are tested against comparable incast degrees. For example, DCQCN and Timely use up to 20- and 40-way incasts, respectively [52, 73]. This is much smaller than 640, allowing substantial tolerance to technology variations, i.e., we expect the switch buffer to be large enough to prevent most packet drops in datacenters with different BDPs and switch buffer sizes. Nevertheless, it is unlikely that the BDP-to-buffer ratio will grow substantially in the near future: newer 100 GbE switches have even larger buffers (42 MB in Mellanox's Spectrum-2 and 32 MB in Broadcom's Trident-III), and NIC-added latency is continuously decreasing. For example, we measured InfiniBand's RTT between nodes under different ToR's to be only 3.1 µs, and Ethernet has historically caught up with InfiniBand's performance.

2.2 Limitations of existing options

Two reasons underlie our choice to design a new general-purpose RPC system for datacenter networks: First, existing datacenter networking software options sacrifice performance or generality, preventing unmodified applications from using the network efficiently. Second, co-designing storage software with the network is increasingly popular, and is largely seen as necessary to achieve maximum performance. However, such specialization has well-known drawbacks, which can be avoided with a general-purpose communication layer that also provides high performance. We describe a representative set of currently available options and their limitations below, roughly in order of increasing performance and decreasing generality.

Fully-general networking stacks such as mTCP [35] and IX [17] allow legacy sockets-based applications to run unmodified. Unfortunately, they leave substantial performance on the table, especially for small messages. For example, one server core can handle around 1.5 million and 10 million 64 B RPC requests per second with IX [17] and eRPC, respectively.

Some recent RPC systems can perform better, but are designed for specific use cases. For example, RAMCloud RPCs [56] are designed for low latency, but not high throughput. In RAMCloud, a single dispatch thread handles all network I/O, and request processing is done by other worker threads. This requires inter-thread communication for every request, and limits the system's network throughput to one core. FaRM RPCs [25] use RDMA writes over connection-based hardware transports, which limits scalability and prevents use in non-RDMA environments.

Like eRPC, our prior work on FaSST RPCs [39] uses only datagram packet I/O, but requires a lossless fabric. FaSST RPCs do not handle packet loss, large messages, congestion, long-running request handlers, or node failure; researchers have believed that supporting these features in software (instead of NIC hardware) would substantially degrade performance [27]. We show that with careful design, we can support all these features and still match FaSST's performance, while running on a lossy network. This upends conventional wisdom that losslessness or NIC support is necessary for high performance.

2.3 Drawbacks of specialization

Co-designing distributed systems with network hardware is a well-known technique to improve performance. Co-design with RDMA is popular, with numerous examples from key-value stores [25, 38, 50, 65, 66], state machine replication [58], and transaction processing systems [21, 26, 41, 66]. Programmable switches allow in-network optimizations such as reducing network round trips for distributed protocols [37, 43, 44], and in-network caching [36]. Co-design with FPGAs is an emerging technique [33].

While there are advantages of co-design, such specialized systems are unfortunately very difficult to design, implement, and deploy. Specialization breaks abstraction boundaries between components, which prevents reuse of components and increases software complexity. Building distributed storage systems requires tremendous programmer effort, and co-design typically mandates starting from scratch, with new data structures, consensus protocols, or transaction protocols. Co-designed systems often cannot reuse existing codebases or protocols, tests, formal specifications, programmer hours, and feature sets. Co-design also imposes deployment challenges beyond needing custom hardware: for example, using programmable switches requires user control over shared network switches, which may not be allowed by datacenter operators; and, RDMA-based systems are unusable with current NICs in datacenters that do not support PFC.

In several cases, specialization does not provide even a performance advantage. Our prior work shows that RPCs outperform RDMA-based designs for applications like key-value stores and distributed transactions, with the same amount of CPU [38, 39]. This is primarily because operations in these systems often require multiple remote memory accesses that can be done with one RPC, but require multiple RDMAs. In this paper (§ 7.1), we show that RPCs perform comparably with switch- and FPGA-based systems for replication, too.

3 eRPC overview

We provide an overview of eRPC's API and threading model below. In these aspects, eRPC is similar to existing high-performance RPC systems like Mellanox's Accelio [4] and FaRM. eRPC's threading model differs in how we sometimes run long-running RPC handlers in "worker" threads (§ 3.2).

eRPC implements RPCs on top of a transport layer that provides basic unreliable packet I/O, such as UDP or InfiniBand's Unreliable Datagram transport. A userspace NIC driver is required for good performance. Our primary contribution is the design and implementation of end-host mechanisms and a network transport (e.g., wire protocol and congestion control) for the commonly-used RPC API.

3.1 RPC API

RPCs execute at most once, and are asynchronous to avoid stalling on network round trips; intra-thread concurrency is provided using an event loop. RPC servers register request handler functions with unique request types; clients use these request types when issuing RPCs, and get continuation callbacks on RPC completion. Users store RPC messages in opaque, DMA-capable buffers provided by eRPC, called msgbufs; a library that provides marshalling and unmarshalling can be used as a layer on top of eRPC.

Each user thread that sends or receives RPCs creates an exclusive `Rpc` endpoint (a C++ object). Each `Rpc` endpoint contains an RX and TX queue for packet I/O, an event loop, and several *sessions*. A session is a one-to-one connection between two `Rpc` endpoints, i.e., two user threads. The client endpoint of a session is used to send requests to the user thread at the other end. A user thread may participate in multiple sessions, possibly playing different roles (i.e., client or server) in different sessions.

User threads act as "dispatch" threads: they must periodically run their `Rpc` endpoint's event loop to make progress. The event loop performs the bulk of eRPC's work, including packet I/O, congestion control, and management functions. It invokes request handlers and continuations, and dispatches long-running request handlers to worker threads (§ 3.2).

Client control flow: `rpc->enqueue_request()` queues a request msgbuf on a session, which is transmitted when the user runs `rpc`'s event loop. On receiving the response, the event loop copies it to the client's response msgbuf and invokes the continuation callback.

Server control flow: The event loop of the `Rpc` that owns the server session invokes (or dispatches) a request handler on receiving a request. We allow *nested* RPCs, i.e., the handler need not enqueue a response before returning. It may issue its own RPCs and call `enqueue_response()` for the first request later when all dependencies complete.

3.2 Worker threads

A key design decision for an RPC system is which thread runs an RPC handler. Some RPC systems such as RAMCloud use dispatch threads for only network I/O. RAMCloud's dispatch threads communicate with *worker* threads that run request handlers. At datacenter network speeds, however, inter-thread communication is expensive: it reduces throughput and adds up to 400 ns to request latency [56]. Other RPC systems such as Accelio and FaRM avoid this overhead by running all request handlers directly in dispatch threads [25, 38]. This latter approach suffers from two drawbacks when executing long request handlers: First, such handlers block other dispatch processing, increasing tail latency. Second, they prevent rapid server-to-client congestion feedback, since the server might not send packets while running user code.

Striking a balance, eRPC allows running request handlers in both dispatch threads and worker threads: When registering a request handler, the programmer specifies whether the handler should run in a dispatch thread. This is the only additional user input required in eRPC. In typical use cases, handlers that require up to a few hundred nanoseconds use dispatch threads, and longer handlers use worker threads.

3.3 Evaluation clusters

Table 1 shows the clusters used in this paper. They include two types of networks (lossy Ethernet, and lossless InfiniBand), and three generations of NICs released between 2011 (CX3) and 2017 (CX5). eRPC works well on all three clusters, showing that our design is robust to NIC and network technology changes. We use traditional UDP on the Ethernet clusters (i.e., we do not use RoCE), and InfiniBand's Unreliable Datagram transport on the InfiniBand cluster.

Currently, eRPC is primarily optimized for Mellanox NICs. eRPC also works with DPDK-capable NICs that support flow steering. For Mellanox Ethernet NICs, we generate UDP packets directly with `libibverbs` instead of going through DPDK, which internally uses `libibverbs` for these NICs.

Our evaluation primarily uses the large CX4 cluster, which resembles real-world datacenters. The ConnectX-4 NICs used in CX4 are widely deployed in datacenters at Microsoft and Facebook [3, 73], and its Mellanox Spectrum switches perform similarly to Broadcom's Trident switches used in these datacenters (i.e., both switches provide dynamic buffering, cut-through switching, and less than 500 ns port-to-port latency.) We use 100 nodes out of the 200 nodes in the shared CloudLab cluster. The six switches in the CX4 cluster are organized as five ToRs with 40 25 GbE downlinks and five 100 GbE uplinks, for a 2:1 oversubscription.

4 eRPC design

Achieving eRPC's performance goals requires careful design and implementation. We discuss three aspects of eRPC's design in this section: scalability of our networking primitives, the challenges involved in supporting zero-copy, and the design of sessions. The next section discusses eRPC's wire protocol and congestion control. A recurring theme in eRPC's design is that we optimize for the common case, i.e., when request handlers run in dispatch threads, RPCs are small, and the network is congestion-free.

4.1 Scalability considerations

We chose plain packet I/O instead of RDMA writes [25, 66, 69] to send messages in eRPC. This decision is based on prior insights from our design of FaSST: First, packet I/O provides completion queues that can scalably detect received packets. Second, RDMA caches connection state in NICs, which does not scale to large clusters. We next discuss *new* observations about NIC hardware trends that support this design.

4.1.1 Packet I/O scales well

RPC systems that use RDMA writes have a *fundamental* scalability limitation. In these systems, clients write requests directly to per-client circular buffers in the server's memory; the server must poll these buffers to detect new requests. The number of circular buffers grows with the number of clients, limiting scalability.

With traditional userspace packet I/O, the NIC writes an incoming packet's payload to a buffer specified by a descriptor pre-posted to the NIC's RX queue (RQ) by the receiver host; the packet is dropped if the RQ is empty. Then, the

Name	Nodes	Network type	Mellanox NIC	Switches	Intel Xeon E5 CPU code
CX3	11	InfiniBand	56 Gbps ConnectX-3	One SX6036	2650 (8 cores)
CX4	100	Lossy Ethernet	25 Gbps ConnectX-4 Lx	5x SN2410, 1x SN2100	2640 v4 (10 cores)
CX5	8	Lossy Ethernet	Dual-port 40 Gbps ConnectX-5	One SX1036	2697 v3 (14 c) or 2683 v4 (16 c)

Table 1: Measurement clusters. CX4 and CX3 are CloudLab [59] and Emulab [68] clusters, respectively.

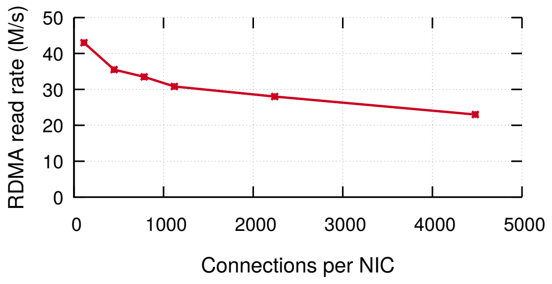


Figure 1: Connection scalability of ConnectX-5 NICs

NIC writes an entry to the host's RX completion queue. The receiver host can then check for received packets in constant time by examining the head of the completion queue.

To avoid dropping packets due to an empty RQ with no descriptors, RQs must be sized proportionally to the number of independent connected RPC endpoints (§ 4.3.1). Older NICs experience cache thrashing with large RQs, thus limiting scalability, but we find that newer NICs fare better: While a Connect-IB NIC could support only 14 2K-entry RQs before thrashing [39], we find that ConnectX-5 NICs do not thrash even with 28 64K-entry RQs. This improvement is due to more intelligent prefetching and caching of RQ descriptors, instead of a massive 64x increase in NIC cache.

We use features of current NICs (e.g., multi-packet RQ descriptors that identify several contiguous packet buffers) in novel ways to guarantee a *constant* NIC memory footprint per CPU core, i.e., it does not depend on the number of nodes in the cluster. This result can simplify the design of future NICs (e.g., RQ descriptor caching is unneeded), but its current value is limited to performance improvements because current NICs support very large RQs, and are perhaps overly complex as a result. We discuss this in detail in Appendix A.

4.1.2 Scalability limits of RDMA

RDMA requires NIC-managed connection state. This limits scalability because NICs have limited SRAM to cache connection state. The number of in-NIC connections may be reduced by sharing them among CPU cores, but doing so reduces performance by up to 80% [39].

Some researchers have hypothesized that improvements in NIC hardware will allow using connected transports at large scale [27, 69]. To show that this is unlikely, we measure the connection scalability of state-of-the-art ConnectX-5 NICs, released in 2017. We repeat the connection scalability experiment from FaSST, which was used to evaluate the older Connect-IB NICs from 2012. We enable PFC on CX5 for this

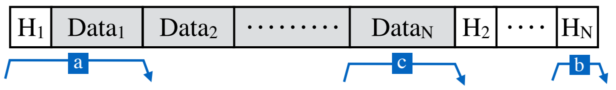


Figure 2: Layout of packet headers and data for an N-packet msgbuf. Blue arrows show NIC DMAs; the letters show the order in which the DMAs are performed for packets 1 and N.

experiment since it uses RDMA; PFC is disabled in all experiments that use eRPC. In the experiment, each node creates a tunable number of connections to other nodes and issues 16-byte RDMA reads on randomly-chosen connections. Figure 1 shows that as the number of connections increases, RDMA throughput decreases, losing ≈50% throughput with 5000 connections. This happens because NICs can cache only a few connections, and cache misses require expensive DMA reads [25]. In contrast, eRPC maintains its peak throughput with 20000 connections (§ 6.3).

ConnectX-5's connection scalability is, surprisingly, not substantially better than Connect-IB despite the five-year advancement. A simple calculation shows why this is hard to improve: In Mellanox's implementation, each connection requires ≈375 B of in-NIC connection state, and the NICs have ≈2 MB of SRAM to store connection state as well as other data structures and buffers [1]. 5000 connections require 1.8 MB, so cache misses are unavoidable.

NIC vendors have been trying to improve RDMA's scalability for a decade [22, 42]. Unfortunately, these techniques do not map well to RPC workloads [39]. Vendors have not put more memory in NICs, probably because of cost and power overheads, and market factors. The scalability issue of RDMA is exacerbated by the popularity of *multihost* NICs, which allow sharing a powerful NIC among 2–4 CPUs [3, 7].

eRPC replaces NIC-managed connection state with CPU-managed connection state. This is an explicit design choice, based upon fundamental differences between the CPU and NIC architectures. NICs and CPUs will both cache recently-used connection state. CPU cache misses are served from DRAM, whereas NIC cache misses are served from the CPU's memory subsystem over the slow PCIe bus. The CPU's miss penalty is therefore much lower. Second, CPUs have substantially larger caches than the ∼2 MB available on a modern NIC, so the cache miss *frequency* is also lower.

4.2 Challenges in zero-copy transmission

eRPC uses zero-copy packet I/O to provide performance comparable to low-level interfaces such as DPDK and RDMA. This section describes the challenges involved in doing so.

4.2.1 Message buffer layout

eRPC provides DMA-capable message buffers to applications for zero-copy transfers. A msgbuf holds one, possibly multi-packet message. It consists of per-packet headers and data, arranged in a fashion optimized for small single-packet messages (Figure 2). Each eRPC packet has a header that contains the transport header, and eRPC metadata such as the request handler type and sequence numbers. We designed a msgbuf layout that satisfies two requirements.

1. The data region is contiguous to allow its use in applications as an opaque buffer.
2. The first packet's data and header are contiguous. This allows the NIC to fetch small messages with one DMA read; using multiple DMAs for small messages would substantially increase NIC processing and PCIe use, reducing message rate by up to 20% [40].

For multi-packet messages, headers for subsequent packets are at the end of the message: placing header 2 immediately after the first data packet would violate our first requirement. Non-first packets require two DMAs (header and data); this is reasonable because the overhead for DMA-reading small headers is amortized over the large data DMA.

4.2.2 Message buffer ownership

Since eRPC transfers packets directly from application-owned msgbufs, msgbuf references must never be used by eRPC after msgbuf ownership is returned to the application. In this paper, we discuss msgbuf ownership issues for only clients; the process is similar but simpler for the server, since eRPC's servers are passive (§ 5). At clients, we must ensure the following invariant: *no eRPC transmission queue contains a reference to the request msgbuf when the response is processed.* Processing the response includes invoking the continuation, which permits the application to reuse the request msgbuf. In eRPC, a request reference may be queued in the NIC's hardware DMA queue, or in our software rate limiter (§ 5.2).

This invariant is maintained trivially when there are no retransmissions or node failures, since the request must exit all transmission queues before the response is received. The following **example** demonstrates the problem with retransmissions. Consider a client that falsely suspects packet loss and retransmits its request. The server, however, received the first copy of the request, and its response reaches the client before the retransmitted request is dequeued. Before processing the response and invoking the continuation, we must ensure that there are no queued references to the request msgbuf. We discuss our solution for the NIC DMA queue next, and for the rate limiter in Appendix C.

The conventional approach to ensure DMA completion is to use "signaled" packet transmission, in which the NIC writes completion entries to the TX completion queue. Unfortunately, doing so reduces message rates by up to 25% by using more NIC and PCIe resources [38], so we use unsignaled packet transmission in eRPC.

Our method of ensuring DMA completion with unsignaled transmission is in line with our design philosophy: we choose to make the common case (no retransmission) fast, at the expense of invoking a more-expensive mechanism to handle the rare cases. We flush the TX DMA queue after queueing a retransmitted packet, which blocks until all queued packets are DMA-ed. This ensures the required invariant: when a response is processed, there are no references to the request in the DMA queue. This flush is moderately expensive (≈2 µs), but it is called during rare retransmission or node failure events, and it allows eRPC to retain the 25% throughput increase from unsignaled transmission.

During server node failures, eRPC invokes continuations with error codes, which also yield request msgbuf ownership. It is possible, although extremely unlikely, that server failure is suspected while a request (not necessarily a retransmission) is in the DMA queue or the rate limiter. Handling node failures requires similar care as discussed above, and is discussed in detail in Appendix B.

4.2.3 Zero-copy request processing

Zero-copy reception is harder than transmission: To provide a contiguous request msgbuf to the request handler at the server, we must strip headers from received packets, and copy only application data to the target msgbuf. However, we were able to provide zero-copy reception for our common-case workload consisting of single-packet requests and dispatch-mode request handlers as follows. eRPC owns the packet buffers DMA-ed by the NIC until it re-adds the descriptors for these packets back to the receive queue (i.e., the NIC cannot modify the packet buffers for this period.) This ownership guarantee allows running dispatch-mode handlers without copying the DMA-ed request packet to a dynamically-allocated msgbuf. Doing so improves eRPC's message rate by up to 16% (§ 6.2).

4.3 Sessions

Each session maintains multiple outstanding requests to keep the network pipe full. Concurrently requests on a session can complete *out-of-order* with respect to each other. This avoids blocking dispatch-mode RPCs behind a long-running worker-mode RPC. We support a constant number of concurrent requests (default = 8) per session; additional requests are transparently queued by eRPC. This is inspired by how RDMA connections allow a constant number of operations [10]. A session uses an array of *slots* to track RPC metadata for outstanding requests.

Slots in server-mode sessions have an MTU-size preallocated msgbuf for use by request handlers that issue short responses. Using the preallocated msgbuf does not require user input: eRPC chooses it automatically at run time by examining the handler's desired response size. This optimization avoids the overhead of dynamic memory allocation, and improves eRPC's message rate by up to 13% (§ 6.2).

4.3.1 Session credits

eRPC limits the number of unacknowledged packets on a session for two reasons. First, to avoid dropping packets due to an empty RQ with no descriptors, the number of packets that may be sent to an Rpc must not exceed the size of its RQ ($|RQ|$). Because each session sends packets independently of others, we first limit the number of sessions that an Rpc can participate in. Each session then uses *session credits* to implement packet-level flow control: we limit the number of packets that a client may send on a session before receiving a reply, allowing the server Rpc to replenish used RQ descriptors before sending more packets.

Second, session credits automatically implement end-to-end flow control, which reduces switch queueing (§ 5.2). Allowing BDP/MTU credits per session ensures that each session can achieve line rate. Mittal et al. [53] have proposed similar flow control for RDMA NICs (§ 5.2.3).

A client session starts with a quota of C packets. Sending a packet to the server consumes a credit, and receiving a packet replenishes a credit. An Rpc can therefore participate in up to $|RQ|/C$ sessions, counting both server-mode and client-mode sessions; session creation fails after this limit is reached. We plan to explore statistical multiplexing in the future.

4.3.2 Session scalability

eRPC's scalability depends on the user's desired value of C, and the number and size of RQs that the NIC and host can effectively support. Lowering C increases scalability, but reduces session throughput by restricting the session's packet window. Small values of C (e.g., $C = 1$) should be used in applications that (a) require only low latency and small messages, or (b) whose threads participate in many sessions. Large values (e.g., BDP/MTU) should be used by applications whose sessions individually require high throughput.

Modern NICs can support several very large RQs, so NIC RQ capacity limits scalability only on older NICs. In our evaluation, we show that eRPC can handle 20000 sessions with 32 credits per session on the widely-used ConnectX-4 NICs. However, since each RQ entry requires allocating a packet buffer in host memory, needlessly large RQs waste host memory and should be avoided.

5 Wire protocol

We designed a wire protocol for eRPC that is optimized for small RPCs and accounts for per-session credit limits. For simplicity, we chose a simple *client-driven* protocol, meaning that each packet sent by the server is in response to a client packet. A client-driven protocol has fewer "moving parts" than a protocol in which both the server and client can independently send packets. Only the client maintains

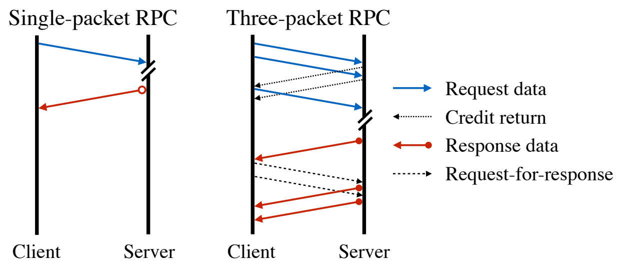


Figure 3: Examples of eRPC's wire protocol, with 2 credits/session.

wire protocol state that is rolled back during retransmission. This removes the need for client-server coordination before rollback, reducing complexity. A client-driven protocol also shifts the overhead of rate limiting entirely to clients, freeing server CPU that is often more valuable.

5.1 Protocol messages

Figure 3 shows the packets sent with $C = 2$ for a small single-packet RPC, and for an RPC whose request and response require three packets each. Single-packet RPCs use the fewest packets possible. The client begins by sending a window of up to C request data packets. For each request packet except the last, the server sends back an explicit *credit return* (CR) packet; the credit used by the last request packet is implicitly returned by the first response packet.

Since the protocol is client-driven, the server cannot immediately send response packets after the first. Subsequent response packets are triggered by *request-for-response* (RFR) packets that the client sends after receiving the first response packet. This increases the latency of multi-packet responses by up to one RTT. This is a fundamental drawback of client-driven protocols; in practice, we found that the added latency is less than 20% for responses with four or more packets.

CRs and RFRs are tiny 16 B packets, and are sent only for large multi-packet RPCs. The additional overhead of sending these tiny packets is small with userspace networking that our protocol is designed for, so we do not attempt complex optimizations such as cumulative CRs or RFRs. These optimizations may be worthwhile for kernel-based networking stacks, where sending a 16 B packet and an MTU-sized packet often have comparable CPU cost.

5.2 Congestion control

Congestion control for datacenter networks aims to reduce switch queueing, thereby preventing packet drops and reducing RTT. Prior high-performance RPC implementations such as FaSST do not implement congestion control, and some researchers have hypothesized that doing so will substantially reduce performance [27]. Can effective congestion control be implemented efficiently in software? We show that optimizing for uncongested networks, and recent advances in software rate limiting allow congestion control with only 9% overhead (§ 6.2).

5.2.1 Available options

Congestion control for high-speed datacenter networks is an evolving area of research, with two major approaches for commodity hardware: RTT-based approaches such as Timely [52], and ECN-based approaches such as DCQCN [73]. Timely and DCQCN have been deployed at Google and Microsoft, respectively. We wish to use these protocols since they have been shown to work at scale.

Both Timely and DCQCN are rate-based: client use the congestion signals to adjust per-session sending rates. We implement Carousel's rate limiter [61], which is designed to efficiently handle a large number of sessions. Carousel's design works well for us as-is, so we omit the details.

eRPC includes the hooks and mechanisms to easily implement either Timely or DCQCN. Unfortunately, we are unable to implement DCQCN because none of our clusters performs ECN marking[1]. Timely can be implemented entirely in software, which made it our favored approach. eRPC runs all three Timely components—per-packet RTT measurement, rate computation using the RTT measurements, and rate limiting—at client session endpoints. For `Rpc`'s that host only server-mode endpoints, there is no overhead due to congestion control.

5.2.2 Common-case optimizations

We use three optimizations for our common-case workloads. Our evaluation shows that these optimizations reduce the overhead of congestion control from 20% to 9%, and that they do not reduce the effectiveness of congestion control. The first two are based on the observation that datacenter networks are typically uncongested. Recent studies of Facebook's datacenters support this claim: Roy et al. [60] report that 99% of all datacenter links are less than 10% utilized at one-minute timescales. Zhang et al. [71, Fig. 6] report that for Web and Cache traffic, 90% of top-of-rack switch links, which are the most congested switches, are less than 10% utilized at 25 µs timescales.

When a session is uncongested, RTTs are low and Timely's computed rate for the session stays at the link's maximum rate; we refer to such sessions as *uncongested.*

1. **Timely bypass.** If the RTT of a packet received on an uncongested session is smaller than Timely's low threshold, below which it performs additive increase, we do not perform a rate update. We use the recommended value of 50 µs for the low threshold [52, 74].
2. **Rate limiter bypass.** For uncongested sessions, we transmit packets directly instead of placing them in the rate limiter.
3. **Batched timestamps for RTT measurement.** Calling `rdtsc()` costs 8 ns on our hardware, which is substantial when processing millions of small packets per second. We reduce timer overhead by sampling it once per RX or TX batch instead of once per packet.

5.2.3 Comparison with IRN

IRN [53] is a new RDMA NIC architecture designed for lossy networks, with two key improvements. First, it uses BDP flow control to limit the outstanding data per RDMA connection to one BDP. Second, it uses efficient selective acks instead of simple go-back-N for packet loss recovery.

IRN was evaluated with simulated switches that have small (60–480 kB) static, per-port buffers. In this buffer-deficient setting, they found SACKs necessary for good performance. However, dynamic-buffer switches are the de-facto standard in current datacenters. As a result, packet losses are very rare with only BDP flow control, so we currently do not implement SACKs, primarily due to engineering complexity. eRPC's dependence on dynamic switch buffers can be reduced by implementing SACK.

With small per-port switch buffers, IRN's maximum RTT is a few hundred microseconds, allowing a ~300 µs retransmission timeout (RTO). However, the 12 MB dynamic buffer in our main CX4 cluster (25 Gbps) can add up to 3.8 ms of queueing delay. Therefore, we use a conservative 5 ms RTO.

5.3 Handling packet loss

For simplicity, eRPC treats reordered packets as losses by dropping them. This is not a major deficiency because datacenter networks typically use ECMP for load balancing, which preserves intra-flow ordering [30, 71, 72] except during rare route churn events. Note that current RDMA NICs also drop reordered packets [53].

On suspecting a lost packet, the client rolls back the request's wire protocol state using a simple go-back-N mechanism. It then reclaims credits used for the rolled-back transmissions, and retransmits from the updated state. The server never runs the request handler for a request twice, guaranteeing at-most-once RPC semantics.

In case of a false positive, a client may violate the credit agreement by having more packets outstanding to the server than its credit limit. In the extremely rare case that such an erroneous loss detection occurs *and* the server's RQ is out of descriptors, eRPC will have "induced" a real packet loss. We allow this possibility and handle the induced loss like a real packet loss.

6 Microbenchmarks

eRPC is implemented in 6200 SLOC of C++, excluding tests and benchmarks. We use static polymorphism to create an `Rpc` class that works with multiple transport types without the overhead of virtual function calls. In this section, we evaluate eRPC's latency, message rate, scalability, and bandwidth using microbenchmarks. To understand eRPC's performance in commodity datacenters, we primarily use the large CX4

[1]The Ethernet switch in our private CX5 cluster does not support ECN marking [5, p. 839]; we do not have admin access to the shared CloudLab switches in the public CX4 cluster; and InfiniBand NICs in the CX3 cluster do not relay ECN marks to software.

Cluster	CX3 (InfiniBand)	CX4 (Eth)	CX5 (Eth)
RDMA read	1.7 µs	2.9 µs	2.0 µs
eRPC	2.1 µs	3.7 µs	2.3 µs

Table 2: Comparison of median latency with eRPC and RDMA

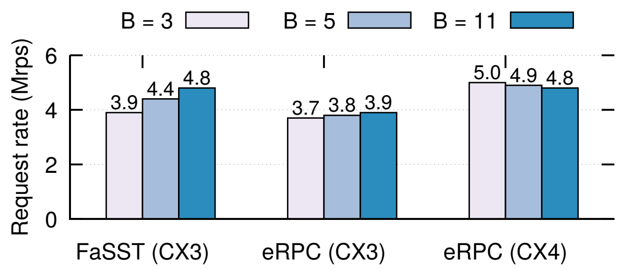


Figure 4: Single-core small-RPC rate with B requests per batch

cluster. We use CX5 and CX3 for their more powerful NICs and low-latency InfiniBand, respectively. eRPC's congestion control is enabled by default.

6.1 Small RPC latency

How much latency does eRPC add? Table 2 compares the median latency of 32 B RPCs and RDMA reads between two nodes connected to the same ToR switch. Across all clusters, eRPC is at most 800 ns slower than RDMA reads.

eRPC's median latency on CX5 is only 2.3 µs, showing that latency with commodity Ethernet NICs and software networking is much lower than the widely-believed value of 10–100 µs [37, 57]. CX5's switch adds 300 ns to every layer-3 packet [12], meaning that end-host networking adds only ≈850 ns each at the client and server. This is comparable to switch-added latency. We discuss this further in § 7.1.

6.2 Small RPC rate

What is the CPU cost of providing generality in an RPC system? We compare eRPC's small message performance against FaSST RPCs, which outperform other RPC systems such as FaRM [39]. FaSST RPCs are *specialized* for single-packet RPCs in a lossless network, and they do not handle congestion.

We mimic FaSST's experiment setting: one thread per node in an 11-node cluster, each of which acts each acts as both RPC server and client. Each thread issues batches of B requests, keeping multiple request batches in flight to hide network latency. Each request in a batch is sent to a randomly-chosen remote thread. Such batching is common in key-value stores and distributed online transaction processing. Each thread keeps up to 60 requests in flight, spread across all sessions. RPCs are 32 B in size. We compare eRPC's performance on CX3 (InfiniBand) against FaSST's reported numbers on the same cluster. We also present eRPC's performance on the CX4 Ethernet cluster. We omit CX5 since it has only 8 nodes.

Figure 4 shows that eRPC's per-thread request issue rate is at most 18% lower than FaSST across all batch sizes, and

Action	**RPC rate**	**% loss**
Baseline (with congestion control)	4.96 M/s	–
Disable batched RTT timestamps (§5.2)	4.84 M/s	2.4%
Disable Timely bypass (§5.2)	4.52 M/s	6.6%
Disable rate limiter bypass (§5.2)	4.30 M/s	4.8%
Disable multi-packet RQ (§4.1.1)	4.06 M/s	5.6%
Disable preallocated responses (§4.3)	3.55 M/s	12.6%
Disable 0-copy request processing (§4.2.3)	3.05 M/s	14.0%

Table 3: Impact of disabling optimizations on small RPC rate (CX4)

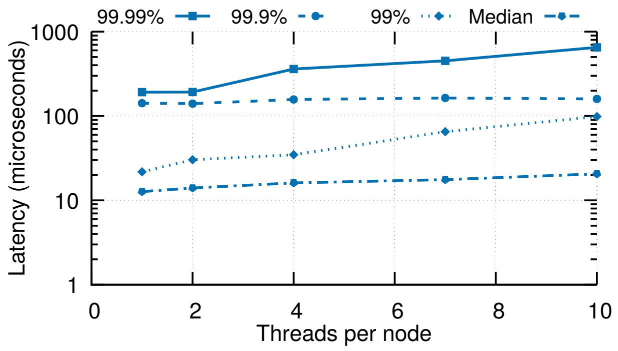


Figure 5: Latency with increasing threads on 100 CX4 nodes

only 5% lower for $B = 3$. This performance drop is acceptable since eRPC is a full-fledged RPC system, whereas FaSST is highly specialized. On CX4, each thread issues 5 million requests per second (Mrps) for $B = 3$; due to the experiment's symmetry, it simultaneously also handles incoming requests from remote threads at 5 Mrps. Therefore, each thread processes 10 million RPCs per second.

Disabling congestion control increases eRPC's request rate on CX4 ($B = 3$) from 4.96 Mrps to 5.44 Mrps. This shows that the overhead of our optimized congestion control is only 9%.

Factor analysis. How important are eRPC's common-case optimizations? Table 3 shows the performance impact of *disabling* some of eRPC's common-case optimizations on CX4; other optimizations such as our single-DMA msgbuf format and unsignaled transmissions cannot be disabled easily. For our baseline, we use $B = 3$ and enable congestion control. Disabling all three congestion control optimizations (§ 5.2.2) reduces throughput to 4.3 Mrps, increasing the overhead of congestion control from 9% to 20%. Further disabling preallocated responses and zero-copy request processing reduces throughput to 3 Mrps, which is 40% lower than eRPC's peak throughput. *We therefore conclude that optimizing for the common case is both necessary and sufficient for high-performance RPCs.*

6.3 Session scalability

We evaluate eRPC's scalability on CX4 by increasing the number of nodes in the previous experiment ($B = 3$) to 100. The five ToR switches in CX4 were assigned between 14 and

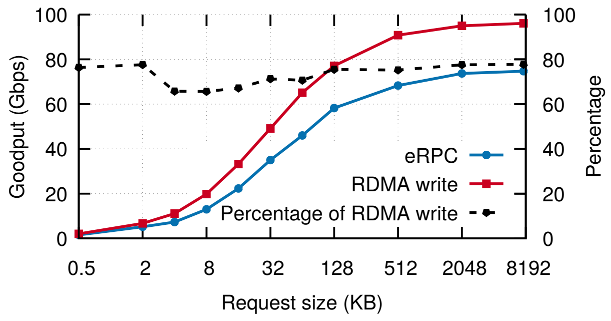


Figure 6: Throughput of large transfers over 100 Gbps InfiniBand

27 nodes each by CloudLab. Next, we increase the number of threads per node: With T threads per node, there are $100T$ threads in the cluster; each thread creates a client-mode session to $100T - 1$ threads. Therefore, each node hosts $T * (100T - 1)$ client-mode sessions, and an equal number of server-mode sessions. Since CX4 nodes have 10 cores, each node handles up to 19980 sessions. This is a challenging traffic pattern that resembles distributed online transaction processing (OLTP) workloads, which operate on small data items [26, 39, 66, 69].

With 10 threads/node, each node achieves 12.3 Mrps on average. At 12.3 Mrps, each node sends and receives 24.6 million packets per second (packet size = 92 B), corresponding to 18.1 Gbps. This is close to the link's achievable bandwidth (23 Gbps out of 25 Gbps), but is somewhat smaller because of oversubscription. We observe retransmissions with more than two threads per node, but the retransmission rate stays below 1700 packets per second per node.

Figure 5 shows the RPC latency statistics. The median latency with one thread per node is 12.7 µs. This is higher than the 3.7 µs for CX4 in Table 2 because most RPCs now go across multiple switches, and each thread keeps 60 RPCs in flight, which adds processing delay. Even with 10 threads per node, eRPC's 99.99th percentile latency stays below 700 µs.

These results show that eRPC can achieve high message rate, bandwidth, and scalability, and low latency in a large cluster with lossy Ethernet. Distributed OLTP has been a key application for lossless RDMA fabrics; our results show that it can also perform well on lossy Ethernet.

6.4 Large RPC bandwidth

We evaluate eRPC's bandwidth using a client thread that sends large messages to a remote server thread. The client sends R-byte requests and keeps one request outstanding; the server replies with a small 32 B response. We use up to 8 MB requests, which is the largest message size supported by eRPC. We use 32 credits per session. To understand how eRPC performs relative to hardware limits, we compare against R-byte RDMA writes, measured using `perftest`.

On the clusters in Table 1, eRPC gets bottlenecked by network bandwidth in this experiment setup. To understand eRPC's performance limits, we connect two nodes in the CX5 cluster to a 100 Gbps switch via ConnectX-5 InfiniBand NICs. (CX5 is used as a 40 GbE cluster in the rest of this paper.) Figure 6 shows that eRPC achieves up to 75 Gbps with one core. eRPC's throughput is at least 70% of RDMA write throughput for 32 kB or larger requests.

In the future, eRPC's bandwidth can be improved by freeing-up CPU cycles. First, on-die memory copy accelerators can speed up copying data from RX ring buffers to request or response msgbufs [2, 28]. Commenting out the memory copies at the server increases eRPC's bandwidth to 92 Gbps, showing that copying has substantial overhead. Second, cumulative credit return and request-for-response (§ 5.1) can reduce packet processing overhead.

Table 4 shows the throughput with R = 8 MB (the largest size supported by eRPC), and varying, artificially-injected packet loss rates. With the current 5 ms RTO, eRPC is usable while the loss probability is up to .01%, beyond which throughput degrades rapidly. We believe that this is sufficient to handle packet corruptions. RDMA NICs can handle a somewhat higher loss rate (.1%) [73].

Loss rate	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}
Bandwidth (Gbps)	73	71	57	18	2.5

Table 4: eRPC's 8 MB request throughput with packet loss

Incast degree	**Total bw**	**50% RTT**	**99% RTT**
20	21.8 Gbps	39 µs	67 µs
20 (no cc)	23.1 Gbps	202 µs	204 µs
50	18.4 Gbps	34 µs	174 µs
50 (no cc)	23.0 Gbps	524 µs	524 µs
100	22.8 Gbps	349 µs	969 µs
100 (no cc)	23.0 Gbps	1056 µs	1060 µs

Table 5: Effectiveness of congestion control (cc) during incast

6.5 Effectiveness of congestion control

We evaluate if our congestion control is successful at reducing switch queueing. We create an incast traffic pattern by increasing the number of client nodes in the previous setup (R = 8 MB). The one server node acts as the incast victim. During an incast, queuing primarily happens at the victim's ToR switch. We use per-packet RTTs measured at the clients as a proxy for switch queue length [52].

Table 5 shows the total bandwidth achieved by all flows and per-packet RTT statistics on CX4, for 20, 50, and 100-way incasts (one flow per client node). We use two configurations: first with eRPC's optimized congestion control, and second with no congestion control. Disabling our common-case congestion control optimizations does not substantially affect the RTT statistics, indicating that these optimizations do not reduce the quality of congestion control.

Congestion control successfully handles our target workloads of up to 50-way incasts, reducing median and 99th percentile queuing by over 5x and 3x, respectively. For 100-way incasts, our implementation reduces median queueing by 3x, but fails to substantially reduce 99th percentile queueing. This is in line with Zhu et al. [74, § 4.3]'s analysis, which shows that Timely-like protocols work well with up to approximately 40 incast flows.

The combined incast throughput with congestion control is within 20% of the achievable 23 Gbps. We believe that this small gap can be further reduced with better tuning of Timely's many parameters. Note that we can also support ECN-based congestion control in eRPC, which may be a better congestion indicator than RTT [74].

Incast with background traffic. Next, we augment the setup above to mimic an experiment from Timely [52, Fig 22]: we create one additional thread at each node that is not the incast victim. These threads exchange latency-sensitive RPCs (64 kB request and response), keeping one RPC outstanding. During a 100-way incast, the 99th percentile latency of these RPCs is 274 µs. This is similar to Timely's latency (≈200-300 µs) with a 40-way incast over a 20 GbE lossless RDMA fabric. Although the two results cannot be directly compared, this experiment shows that the latency achievable with software-only networking in commodity, lossy datacenters is comparable to lossless RDMA fabrics, even with challenging traffic patterns.

7 Full-system benchmarks

In this section, we evaluate whether eRPC can be used in real applications with unmodified existing storage software: We build a state machine replication system using an open-source implementation of Raft [54], and a networked ordered key-value store using Masstree [49].

7.1 Raft over eRPC

State machine replication (SMR) is used to build fault-tolerant services. An SMR service consists of a group of server nodes that receive commands from clients. SMR protocols ensure that each server executes the same sequence of commands, and that the service remains available if servers fail. Raft [54] is such a protocol that takes a *leader*-based approach: Absent failures, the Raft replicas have a stable leader to which clients send commands; if the leader fails, the remaining Raft servers elect a new one. The leader appends the command to replicas' logs, and it replies to the client after receiving acks from a majority of replicas.

SMR is difficult to design and implement correctly [31]: the protocol must have a specification and a proof (e.g., in TLA+), and the implementation must adhere to the specification. We avoid this difficulty by using an existing implementation of Raft [14]. (It had no distinct name, so we term it LibRaft.) We did not write LibRaft ourselves; we found it on GitHub and used it as-is. LibRaft is well-tested with fuzzing over a network simulator and 150+ unit tests. Its only requirement is that the user provide callbacks for sending and handling RPCs—which we implement using eRPC. Porting to eRPC required no changes to LibRaft's code.

Measurement	System	Median	99%
Measured at client	NetChain	9.7 µs	N/A
	eRPC	5.5 µs	6.3 µs
Measured at leader	ZabFPGA	3.0 µs	3.0 µs
	eRPC	3.1 µs	3.4 µs

Table 6: Latency comparison for replicated PUTs

We compare against recent consistent replication systems that are built from scratch for two specialized hardware types. First, NetChain [37] implements chain replication over programmable switches. Other replication protocols such as conventional primary-backup and Raft are too complex to implement over programmable switches [37]. Therefore, despite the protocol-level differences between LibRaft-over-eRPC and NetChain, our comparison helps understand the relative performance of end-to-end CPU-based designs and switch-based designs for in-memory replication. Second, Consensus in a Box [33] (called ZabFPGA here), implements ZooKeeper's atomic broadcast protocol [32] on FPGAs. eRPC also outperforms DARE [58], which implements SMR over RDMA; we omit the results for brevity.

Workloads. We mimic NetChain and ZabFPGA's experiment setups for latency measurement: we implement a 3-way replicated in-memory key-value store, and use one client to issue PUT requests. The replicas' command logs and key-value store are stored in DRAM. NetChain and ZabFPGA use 16 B keys, and 16–64 B values; we use 16 B keys and 64 B values. The client chooses PUT keys uniformly at random from one million keys. While NetChain and ZabFPGA also implement their key-value stores from scratch, we reuse existing code from MICA [45]. We compare eRPC's performance on CX5 against their published numbers because we do not have the hardware to run NetChain or ZabFPGA. Table 6 compares the latencies of the three systems.

7.1.1 Comparison with NetChain

NetChain's key assumption is that software networking adds 1–2 orders of magnitude more latency than switches [37]. However, we have shown that eRPC adds 850 ns, which is only around 2x higher than latency added by current programmable switches (400 ns [8]).

Raft's latency over eRPC is 5.5 µs, which is substantially lower than NetChain's 9.7 µs. This result must be taken with a grain of salt: On the one hand, NetChain uses NICs that have higher latency than CX5's NICs. On the other hand, it has numerous limitations, including key-value size and capacity constraints, serial chain replication whose latency increases linearly with the number of replicas, absence of congestion control, and reliance on a complex and external failure detector. The main takeaway is that microsecond-scale consistent replication is achievable in commodity Ethernet datacenters with a general-purpose networking library.

7.1.2 Comparison with ZabFPGA

Although ZabFPGA's SMR servers are FPGAs, the clients are commodity workstations that communicate with the FPGAs over slow kernel-based TCP. For a challenging comparison, we compare against ZabFPGA's commit latency measured at the leader, which involves only FPGAs. In addition, we consider its "direct connect" mode, where FPGAs communicate over point-to-point links (i.e., without a switch) via a custom protocol. Even so, eRPC's median leader commit latency is only 3% worse.

An advantage of specialized, dedicated hardware is low jitter. This is highlighted by ZabFPGA's negligible leader latency variance. This advantage does not carry over directly to end-to-end latency [33] because storage systems built with specialized hardware are eventually accessed by clients running on commodity workstations.

7.2 Masstree over eRPC

Masstree [49] is an ordered in-memory key-value store. We use it to implement a single-node database index that supports low-latency point queries in the presence of less performance-critical longer-running scans. This requires running scans in worker threads. We use CX3 for this experiment to show that eRPC works well on InfiniBand.

We populate a Masstree server on CX3 with one million random 8 B keys mapped to 8 B values. The server has 16 Hyper-Threads, which we divide between 14 dispatch threads and 2 worker threads. We run 64 client threads spread over 8 client nodes to generate the workload. The workload consists of 99% GET(key) requests that fetch a key-value item, and 1% SCAN(key) requests that sum up the values of 128 keys succeeding the key. Keys are chosen uniformly at random from the inserted keys. Two outstanding requests per client was sufficient to saturate our server.

We achieve 14.3 million GETs/s on CX3, with 12 µs 99th percentile GET latency. If the server is configured to run only dispatch threads, the 99th percentile GET latency rises to 26 µs. eRPC's median GET latency under low load is 2.7 µs. This is around 10x faster than Cell's single-node B-Tree that uses multiple RDMA reads [51]. Despite Cell's larger key/-value sizes (64 B/256 B), the latency differences are mostly from RTTs: At 40 Gbps, an additional 248 B takes only 50 ns more time to transmit.

8 Related work

RPCs. There is a vast amount of literature on RPCs. The practice of optimizing an RPC wire protocol for small RPCs originates with Birrell and Nelson [19], who introduce the idea of an implicit-ACK. Similar to eRPC, the Sprite RPC system [67] directly uses raw datagrams and performs retransmissions only at clients. The Direct Access File System [23] was one of the first to use RDMA in RPCs. It uses SEND/RECV messaging over a connected transport to initiate an RPC, and RDMA reads or writes to transfer the bulk of large RPC messages. This design is widely used in other systems such as NFS's RPCs [20] and some MPI implementations [48]. In eRPC, we chose to transfer all data over datagram messaging to avoid the scalability limits of RDMA. Other RPC systems that use RDMA include Mellanox's Accelio [4] and RFP [63]. These systems perform comparably to FaRM's RPCs, which are slower than eRPC at scale by an order of magnitude.

Co-design. There is a rapidly-growing list of projects that co-design distributed systems with the network. This includes key-value stores [38, 46, 50, 65], distributed databases and transaction processing systems [21, 25, 66, 69], state machine replication [33, 58], and graph-processing systems [62]. We believe the availability of eRPC will motivate researchers to investigate how much performance these systems can achieve without sacrificing the networking abstraction. On the other hand, there is a smaller set of recent projects that also prefer RPCs over co-design, including RAMCloud, FaSST, and the distributed data shuffler by Liu et al. [47]. However, their RPCs lack either performance (RAMCloud) or generality (FaSST), whereas eRPC provides both.

9 Conclusion

eRPC is a fast, general-purpose RPC system that provides an attractive alternative to putting more functions in network hardware, and specialized system designs that depend on these functions. eRPC's speed comes from prioritizing common-case performance, carefully combining a wide range of old and new optimizations, and the observation that switch buffer capacity far exceeds datacenter BDP. eRPC delivers performance that was until now believed possible only with lossless RDMA fabrics or specialized network hardware. It allows unmodified applications to perform close to the hardware limits. Our ported versions of LibRaft and Masstree are, to our knowledge, the fastest replicated key-value store and networked database index in the academic literature, while operating end-to-end without additional network support.

Acknowledgments We received valueable insights from our SIGCOMM and NSDI reviewers, Miguel Castro, John Ousterhout, and Yibo Zhu. Sol Boucher, Jack Kosaian, and Hyeontaek Lim helped improve the writing. CloudLab [59] and Emulab [68] resources were used in our experiments. This work was supported by funding from the National Science Foundation under awards CCF-1535821 and CNS-1700521, and by Intel via the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS). Anuj Kalia is supported by the Facebook Fellowship.

Appendix A. eRPC's NIC memory footprint

Primarily, four on-NIC structures contribute to eRPC's NIC memory footprint: the TX and RX queues, and their corresponding completion queues. The TX queue must allow sufficient pipelining to hide PCIe latency; we found that 64 entries are sufficient in all cases. eRPC's TX queue and TX completion queue have 64 entries by default, so their footprint does not depend on cluster size. The footprint of on-NIC page table entries required for eRPC is negligible because we use 2 MB hugepages [25].

As discussed in Section 4.3.1, eRPC's RQs must have sufficient descriptors for all connected sessions. If traditional RQs are used, their footprint grows with the number of connected sessions supported. Modern NICs (e.g., ConnectX-4 and newer NICs from Mellanox) support *multi-packet* RQ descriptors that specify multiple contiguous packet buffers using base address, buffer size, and number of buffers. With eRPC's default configuration of 512-way RQ descriptors, RQ size is reduced by 512x, making it negligible. This optimization has the added advantage of almost eliminating RX descriptor DMA, which is now needed only once every 512 packets. While multi-packet RQs were originally designed for large receive offload of one message [6], we use this feature to receive packets of independent messages.

What about the RX completion queue (CQ)? By default, NICs expect the RX CQ to have sufficient space for each received packet, so using multi-packet RQ descriptors does not reduce CQ size. However, eRPC does not need the information that the NIC DMA-writes to the RX CQ entries. It needs only the number of new packets received. Therefore, we shrink the CQ by allowing it to *overrun*, i.e., we allow the NIC to overwrite existing entries in the CQ in a round-robin fashion. We poll the overrunning CQ to check for received packets. It is possible to use a RX CQ with only one entry, but we found that doing so causes cache line contention between eRPC's threads and the CPU's on-die PCIe controller. We solve this issue by using 8-entry CQs, which makes the contention negligible.

Appendix B. Handling node failures

eRPC launches a session management thread that handles sockets-based management messaging for creating and destroying sessions, and detects failure of remote nodes with timeouts. When the management thread suspects a remote node failure, each dispatch thread with sessions to the remote node acts as follows. First, it flushes the TX DMA queue to release msgbuf references held by the NIC. For client sessions, it waits for the rate limiter to transmit any queued packets for the session, and then invokes continuations for pending requests with an error code. For server-mode sessions, it frees session resources after waiting (non-blocking) for request handlers that have not enqueued a response.

Appendix C. Rate limiting with zero-copy

Recall the request retransmission example discussed in § 4.2.2: On receiving the response for the first copy of a retransmitted request, we wish to ensure that the rate limiter does not contain a reference to the retransmitted copy. Unlike eRPC's NIC DMA queue that holds only a few tens of packets, the rate limiter tracks up to milliseconds worth of transmissions during congestion. As a result, flushing it like the DMA queue is too slow. Deleting references from the rate limiter turned out to be too complex: Carousel requires a bounded difference between the current time and a packet's scheduled transmission time for correctness, so deletions require rolling back Timely's internal rate computation state. Each Timely instance is shared by all slots in a session (§ 4.3), which complicates rollback.

We solve this problem by dropping response packets received while a retransmitted request is in the rate limiter. Each such response indicates a false positive in our retransmission mechanism, so they are rare. This solution does not work for the NIC DMA queue: since we use unsignaled transmission, it is generally impossible for software to know whether a request is in the DMA queue without flushing it.

References

[1] Private communication with Mellanox.

[2] Fast memcpy with SPDK and Intel I/OAT DMA Engine. `https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine`.

[3] A peek inside Facebook's server fleet upgrade. `https://www.nextplatform.com/2017/03/13/peek-inside-facebooks-server-fleet-upgrade/`, 2017.

[4] Mellanox Accelio. `http://www.accelio.org`, 2017.

[5] Mellanox MLNX-OS user manual for Ethernet. `http://www.mellanox.com/related-docs/prod_management_software/MLNX-OS_ETH_v3_6_3508_UM.pdf`, 2017.

[6] Mellanox OFED for Linux release notes. `http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_Release_Notes_3_2-1_0_1_1.pdf`, 2017.

[7] Oak Ridge leadership computing facility - Summit. `https://www.olcf.ornl.gov/summit/`, 2017.

[8] Aurora 710 based on Barefoot Tofino switching silicon. `https://netbergtw.com/products/aurora-710/`, 2018.

[9] Facebook open switching system FBOSS and Wedge in the open. `https://code.facebook.com/posts/843620439027582/facebook-open-switching-system-fboss-and-wedge-in-the-open/`, 2018.

[10] RDMAmojo - blog on RDMA technology and programming by Dotan Barak. http://www.rdmamojo.com/2013/01/12/ibv_modify_qp/, 2018.

[11] Distributed asynchronous object storage stack. https://github.com/daos-stack, 2018.

[12] Tolly report: Mellanox SX1016 and SX1036 10/40GbE switches. http://www.mellanox.com/related-docs/prod_eth_switches/Tolly212113MellanoxSwitchSXPerformance.pdf, 2018.

[13] Jim Warner's switch buffer page. https://people.ucsc.edu/~warner/buffer.html, 2018.

[14] C implementation of the Raft consensus protocol. https://github.com/willemt/raft, 2018.

[15] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, Hong Kong, China, Aug. 2013.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[17] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[18] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. In *Proc. VLDB*, New Delhi, India, Aug. 2016.

[19] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 1984.

[20] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, 2003.

[21] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[22] D. Crupnicoff, M. Kagan, A. Shahar, N. Bloch, and H. Chapman. Dynamically-connected transport service, May 19 2011. URL https://www.google.com/patents/US20110116512. US Patent App. 12/621,523.

[23] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The direct access file system. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, 2003.

[24] DPDK. Data Plane Development Kit (DPDK). http://dpdk.org/, 2017.

[25] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[26] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[27] A. Dragojevic, D. Narayanan, and M. Castro. RDMA reads: To use or not to use? *IEEE Data Eng. Bull.*, 2017.

[28] M. D. et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.

[29] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, Apr. 2018.

[30] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[31] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[33] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proc. 13th USENIX NSDI*, Santa Clara, CA, May 2016.

[34] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. Aug. 2017.

[35] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[37] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM*

SIGCOMM, Chicago, IL, Aug. 2014.

[39] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[40] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high-performance RDMA systems. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[41] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. HyperLoop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[42] M. J. Koop, J. K. Sridhar, and D. K. Panda. Scalable MPI design over InfiniBand using eXtended Reliable Connection. In *2008 IEEE International Conference on Cluster Computing*, 2008.

[43] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say no to Paxos overhead: Replacing consensus with network ordering. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[44] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[45] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ISCA*, 2015.

[46] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, Apr. 2014.

[47] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proc. 12th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2017.

[48] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.

[49] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

[50] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference*, San Jose, CA, June 2013.

[51] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the Cell distributed B-Tree store. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[52] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[53] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting network support for RDMA. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, Philadelphia, PA, June 2014.

[55] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[56] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM TOCS*, 2015.

[57] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[58] M. Poke and T. Hoefler. DARE: High-performance state machine replication on RDMA networks. In *HPDC*, 2015.

[59] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[60] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[61] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proc. ACM SIGCOMM*, Los Angeles, CA, Aug. 2017.

[62] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[63] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proc. 12th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2017.

[64] Y. Wang, X. Meng, L. Zhang, and J. Tan. C-hint: An effective and reliable cache management for RDMA-accelerated key-value stores. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Seattle, WA, Nov.

2014.

[65] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: A resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[66] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[67] B. B. Welch. The Sprite remote procedure call system. Technical report, Berkeley, CA, USA, 1986.

[68] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[69] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. In *Proc. VLDB*, Munich, Germany, Aug. 2017.

[70] J. Zhang, F. Ren, X. Yue, R. Shu, and C. Lin. Sharing bandwidth by allocating switch buffer in data center networks. *IEEE Journal on Selected Areas in Communications*, 2014.

[71] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, 2017.

[72] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proc. 9th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.

[73] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015.

[74] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye. ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY. In *Proc. CoNEXT*, Dec. 2016.

Using RDMA Efficiently for Key-Value Services

Anuj Kalia Michael Kaminsky[†] David G. Andersen
Carnegie Mellon University [†]Intel Labs
{akalia,dga}@cs.cmu.edu michael.e.kaminsky@intel.com

ABSTRACT

This paper describes the design and implementation of HERD, a key-value system designed to make the best use of an RDMA network. Unlike prior RDMA-based key-value systems, HERD focuses its design on reducing network round trips while using efficient RDMA primitives; the result is substantially lower latency, and throughput that saturates modern, commodity RDMA hardware.

HERD has two unconventional decisions: First, it does not use RDMA reads, despite the allure of operations that bypass the remote CPU entirely. Second, it uses a mix of RDMA and messaging verbs, despite the conventional wisdom that the messaging primitives are slow. A HERD client writes its request into the server's memory; the server computes the reply. This design uses a single round trip for all requests and supports up to 26 million key-value operations per second with 5 µs average latency. Notably, for small key-value items, our full system throughput is similar to native RDMA read throughput and is over 2X higher than recent RDMA-based key-value systems. We believe that HERD further serves as an effective template for the construction of RDMA-based datacenter services.

Keywords

RDMA; InfiniBand; RoCE; Key-Value Stores

1. INTRODUCTION

This paper explores a question that has important implications for the design of modern clustered systems: What is the best method for using RDMA features to support remote hash-table access? To answer this question, we first evaluate the performance that, with sufficient attention to engineering, can be achieved by each of the RDMA communication primitives. Using this understanding, we show how to use an unexpected combination of methods and system architectures to achieve the maximum performance possible on a high-performance RDMA network.

SIGCOMM'14, Aug 17-22 2014, Chicago, IL, USA
ACM 978-1-4503-2836-4/14/08.
http://dx.doi.org/10.1145/2619239.2626299

Our work is motivated by the seeming contrast between the fundamental time requirements for cross-node traffic vs. CPU-to-memory lookups, and the designs that have recently emerged that use multiple RDMA (remote direct memory access) reads. On one hand, going between nodes takes roughly 1-3 µs, compared to 60-120 ns for a memory lookup, suggesting that a multiple-RTT design as found in the recent Pilaf [21] and FaRM [8] systems should be fundamentally slower than a single-RTT design. But on the other hand, an RDMA read bypasses many potential sources of overhead, such as servicing interrupts and initiating control transfers, which involve the host CPU. In this paper, we show that there is a better path to taking advantage of RDMA to achieve high-throughput, low-latency key-value storage.

A challenge for both our and prior work lies in the lack of richness of RDMA operations. An RDMA operation can only read or write a remote memory location. It is not possible to do more sophisticated operations such as dereferencing and following a pointer in remote memory. Recent work in building key-value stores [21, 8] has focused exclusively on using RDMA reads to traverse remote data structures, similar to what would have been done had the structure been in local memory. This approach invariably requires multiple round trips across the network.

Consider an ideal RDMA read-based key-value store (or cache) where each `GET` request requires only 1 small RDMA read. Designing such a store is as hard as designing a hash-table in which each `GET` request requires only one random memory lookup. We instead provide a solution to a simpler problem: we design a key-value cache that provides performance similar to that of the ideal cache. However, our design does not use RDMA reads at all.

In this paper, we present HERD, a key-value cache that leverages RDMA features to deliver low latency and high throughput. As we demonstrate later, RDMA reads cannot harness the full performance of the RDMA hardware. In HERD, clients transmit their request to the server's memory using RDMA writes. The server's CPU polls its memory for incoming requests. On receiving a new request, it executes the GET or PUT operation in its local data structures and sends the response back to the client. As RDMA write performance does not scale with the number of outbound connections, the response is sent as a SEND message over a datagram connection.

Our work makes three main contributions:

- A thorough analysis of the performance of RDMA verbs and expose the various design options for key-value systems.
- Evidence that "two-sided" verbs are better than RDMA reads for key-value systems, refuting the previously held assumption [21, 8].
- Describing the design and implementation of HERD, a key-value cache that offers the maximum possible performance of RDMA hardware.

The following section briefly introduces key-value stores and RDMA, and describes recent efforts in building key-value stores using RDMA. Section 3 discusses the rationale behind our design decisions and demonstrates that messaging verbs are a better choice than RDMA reads for key-value systems. Section 4 discusses the design and implementation of our key-value cache. In Section 5, we evaluate our system on a cluster of 187 nodes and compare it against FaRM [8] and Pilaf [21].

2. BACKGROUND

This section provides background information on key-value stores and caches, which are at the heart of HERD. We then provide an overview of RDMA, as is relevant for the rest of the paper.

2.1 Key-Value stores

DRAM-based key-value stores and caches are widespread in large-scale Internet services. They are used both as primary stores (e.g., Redis [4] and RAMCloud [23]), and as caches in front of backend databases (e.g., Memcached [5]). At their most basic level, these systems export a traditional GET/PUT/DELETE interface. Internally, they use a variety of data structures to provide fast, memory-efficient access to their underlying data (e.g., hash table or tree-based indexes).

In this paper, we focus on the communication architecture to support both of these applications; we use a cache implementation for end-to-end validation of our resulting design.

Although recent in-memory object stores have used both tree and hash table-based designs, this paper focuses on hash tables as the basic indexing data structure. Hash table design has a long and rich history, and the particular flavor one chooses depends largely on the desired optimization goals. In recent years, several systems have used advanced hash table designs such as Cuckoo hashing [24, 17, 9] and Hopscotch hashing [12]. Cuckoo hash tables are an attractive choice for building fast key-value systems [9, 31, 17] because, with K hash functions (usually, K is 2 or 3), they require only K memory lookups for GET operations, plus an additional pointer dereference if the values are not stored in the table itself. In many workloads, GETs constitute over 95% of the operations [6, 22]. This property makes cuckoo hashing an attractive backend for an RDMA-based key-value store [21]. Cuckoo and Hopscotch-based designs often emphasize workloads that are read-intensive: PUT operations require moving values within the tables. We evaluate both balanced (50% PUT/GET) and read-intensive (95% GET) workloads in this paper.

To support both types of workloads without being limited by the performance of currently available data structure options, HERD internally uses a *cache* data structure that can evict items when it is full. Our focus, however, is on the network communication architecture—our results generalize across both caches and stores, so long as the implementation is fast enough that a high-performance communication architecture is needed. HERD's cache design is based on the recent MICA [18] system that provides both cache and store semantics. MICA's cache mode uses a lossy associative index to map keys to pointers, and stores the values in a circular log that is memory efficient, avoids fragmentation, and does not require expensive garbage collection. This design requires only 2 random memory accesses for both GET and PUT operations.

2.2 RDMA

Remote Direct Memory Access (RDMA) allows one computer to directly access the memory of a remote computer without involving the operating system at any host. This enables zero-copy transfers, reducing latency and CPU overhead. In this work, we focus on two types of RDMA-providing interconnects: InfiniBand and RoCE (RDMA over Converged Ethernet). However, we believe that our design is applicable to other RDMA providers such as iWARP, Quadrics, and Myrinet.

InfiniBand is a switched fabric network widely used in high-performance computing systems. RoCE is a relatively new network protocol that allows direct memory access over Ethernet. InfiniBand and RoCE NICs achieve low latency by implementing several layers of the network stack (transport layer through physical layer) in hardware, and by providing RDMA and kernel-bypass. In this section, we provide an overview of RDMA features and terminology that are used in the rest of this paper.

2.2.1 *Comparison with classical Ethernet*

To distinguish from RoCE, we refer to non-RDMA providing Ethernet networks as "classical Ethernet." Unlike classical Ethernet NICs, RDMA NICs (RNICs) provide reliable delivery to applications by employing hardware-based retransmission of lost packets. Further, RNICs provide kernel bypass for all communication. These two factors reduce end-to-end latency as well as the CPU load on the communicating hosts. The typical end-to-end ($\frac{1}{2}$RTT) latency in InfiniBand/RoCE is 1 μs while that in modern classical Ethernet-based solutions [2, 18] is 10 μs. A large portion of this gap arises because of differing emphasis in the NIC design. RDMA is increasing its presence in datacenters as the hardware becomes cheaper [21]. A 40 Gbps ConnectX-3 RNIC from Mellanox costs about $500, while a 10 Gbps Ethernet adapter costs between $300 and $800. The introduction of RoCE will further boost RDMA's presence as it will allow sockets applications to run with RDMA applications on the same network.

2.2.2 *Verbs and queue pairs*

Userspace programs access RNICs directly using functions called *verbs*. There are several types of verbs. Those most relevant to this work are RDMA read (READ), RDMA write (WRITE), SEND, and RECEIVE. Verbs are posted by applications to queues that are maintained inside the RNIC. Queues always exist in pairs: a *send queue* and a *receive queue* form a *queue pair* (QP). Each queue pair has an associated *completion queue* (CQ), which the RNIC fills in upon completion of verb execution.

The verbs form a semantic definition of the interface provided by the RNIC. There are two types of verbs semantics: memory semantics and channel semantics.

Memory semantics: The RDMA verbs (READ and WRITE) have memory semantics: they specify the remote memory address to operate upon. These verbs are *one-sided*: the responder's CPU is unaware of the operation. This lack of CPU overhead at the responder makes one-sided verbs attractive. Furthermore, they have the lowest latency and highest throughput among all verbs.

Channel semantics: SEND and RECEIVE (RECV) have channel semantics, i.e., the SEND's payload is written to a remote memory address that is specified by the responder in a pre-posted RECV. An analogy for this would be an unbuffered sockets implementation that required `read()` to be called before the packet arrived. SEND and RECV are *two-sided* as the CPU at the responder needs to post a RECV in order for an incoming SEND to be processed. Unlike the memory verbs, the responder's CPU is involved. Two-sided verbs also have slightly higher latency and lower throughput than one sided verbs and have been regarded unfavorably for designing key-value systems [21, 8].

Although SEND and RECV verbs are technically RDMA verbs, we distinguish them from READ and WRITE. We refer to READ and WRITE as *RDMA verbs*, and refer to SEND and RECV as *messaging verbs*.

Verbs are usually posted to the send queue of a QP (except RECV, which is posted to the receive queue). To post a verb to the RNIC, an application calls into the userland RDMA driver. Then, the driver prepares a Work Queue Element (WQE) in the host's memory and rings a doorbell on the RNIC via Programmed IO (PIO). For ConnectX and newer RNICs, the doorbell contains the entire WQE [27]. For WRITE and SEND verbs, the WQE is associated with a payload that needs to be sent to the remote host. A payload up to the maximum PIO size (256 in our setup) can be *inlined* in the WQE, otherwise it can be fetched by the RNIC via a DMA read. An inlined post involves no DMA operations, reducing latency and increasing throughput for small payloads.

When the RNIC completes the network steps associated with the verb, it pushes a completion event to the queue pair's associated completion queue (CQ) via a DMA write. Using completion events adds extra overhead to the RNIC's PCIe bus. This overhead can be reduced by using *selective signaling*. When using a selectively signaled send queue of size S, up to $S-1$ consecutive verbs can be *unsignaled*, i.e., a completion event will not be pushed for these verbs. The receive queue cannot be selectively signaled. As S is large ($\sim$ 128), we use the terms "selective signaling" and "unsignaled" interchangeably.

2.2.3 *Transport types*

RDMA transports can be connected or unconnected. A connected transport requires a connection between two queue pairs that communicate exclusively with each other. Current RDMA implementations support two main types of connected transports: Reliable Connection (RC) and Unreliable Connection (UC). There is no acknowledgement of packet reception in UC; packets can be lost and the affected message can be dropped. As UC does not generate ACK/NAK packets, it causes less network traffic than RC.

In an unconnected transport, one queue pair can communicate with any number of other queue pairs. Current implementations provide only one unconnected transport: Unreliable Datagram (UD). The RNIC maintains state for each active queue in its queue pair context cache, so datagram transport can scale better for applications with a one-to-many topology.

InfiniBand and RoCE employ lossless link-level flow control, namely, credit-based flow control and Priority Flow Control. *Even with unreliable transports (UC/UD), packets are never lost due to buffer overflows.* Reasons for packet loss include bit errors on the wire and hardware failures, which are extremely rare. Therefore, our design, similar to choices made by Facebook and others [22], sacrifices transport-level retransmission for fast common case performance at the cost of rare application-level retries.

Some transport types support only a subset of the available verbs. Table 1 lists the verbs supported by each transport

Verb	RC	UC	UD
SEND/RECV	✓	✓	✓
WRITE	✓	✓	✗
READ	✓	✗	✗

Table 1: Operations supported by each connection type. UC does not support READs, and UD does not support RDMA at all.

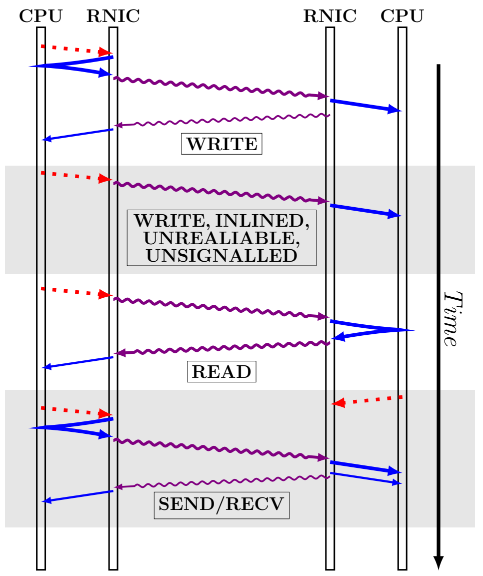


Figure 1: Steps involved in posting verbs. The dotted arrows are PCIe PIO operations. The solid, straight arrows are DMA operations: the thin ones are for writing the completion events. The thick wavy arrows are RDMA data packets and the thin ones are ACKs.

type. Figure 1 shows the DMA and network steps involved in posting verbs.

2.3 Existing RDMA-based key-value stores

Pilaf [21] is a key-value store that aims for high performance and low CPU use. For GETs, clients access a cuckoo hash table at the server using READs, which requires 2.6 round trips on average for single GET request. For PUTs, clients send their requests to the server using a SEND message. To ensure consistent GETs in the presence of concurrent PUTs, Pilaf's data structures are *self-verifying*: each hash table entry is augmented with two 64-bit checksums.

The second key-value store we compare against is based upon the store designed in **FaRM** [8]. It is important to note that FaRM is a more general-purpose distributed computing platform that exposes memory of a cluster of machines as a shared address space; we compare *only* against a key-value store implemented on top of FaRM that we call FaRM-KV. Unlike the client-server design in Pilaf and HERD, FaRM is symmetric, befitting its design as a cluster architecture: each machine acts as both a server and client.

FaRM's design provides two components for comparison. First is its key-value store design, which uses a variant of Hopscotch hashing [12] to create a locality-aware hash table. For GETs, clients READ several consecutive Hopscotch slots, one of which contains the key with high probability. Another READ is required to fetch the value if it is not stored inside the hash table. For PUTs, clients WRITE their request to a circular buffer in the server's memory. The server polls this buffer to detect new requests. This design is not specific to FaRM—we use it merely as an extant alternative to Pilaf's Cuckoo-based design to provide a more in-depth comparison for HERD.

The second important aspect of FaRM is its symmetry; here it differs from both Pilaf and HERD. For small, fixed-size key-value pairs, FaRM can "inline" the value with the key. With inlining, FaRM's RDMA read-based design still achieves lower maximum *throughput* than HERD, but it uses less CPU. This tradeoff may be right for a cluster where all machines are also busy doing computation; we do not evaluate the symmetric use case here, but it is an important consideration for users of either design.

3. DESIGN DECISIONS

Towards our goal of supporting key-value servers that achieve the highest possible throughput with RDMA, we explain in this section the reasons we choose to use—and not use—particular RDMA features and other design options. To begin with, we present an analysis of the performance of the RDMA verbs; we then craft a communication architecture using the fastest among them that can support our application needs.

As hinted in Section 1, one of the core decisions to make is whether to use memory verbs (RDMA read and write) or messaging verbs (SEND and RECV). Recent work from the systems and networking communities, for example, has focused on RDMA reads as a building block, because they bypass the remote network stack and CPU entirely for GETs [21, 8]. In contrast, however, the HPC community has made wider use of messaging, both for key-value caches [14] and general communication [16]. These latter systems scaled to thousands of machines, but provided low throughput—less than one million operations per second in memcached [14]. The reason for low throughput in [14] is not clear, but we suspect application design that makes the system incapable of leveraging the full power of the RNICs.

There remains an important gap between these two lines of work, and to our knowledge, HERD is the first system to provide the best of both worlds: throughput even higher than that of the RDMA-based systems while scaling to several hundred clients.

HERD takes a hybrid approach, using both RDMA and messaging to best effect. RDMA reads, however, are unattractive because of the need for multiple round trips. In HERD, clients instead write their requests to the server using RDMA writes over an Unreliable Connection (UC). This write places

Name	Nodes	Hardware
Apt	187	Intel Xeon E5-2450 CPUs. ConnectX-3 MX354A (56 Gbps IB) via PCIe 3.0 x8
Susitna	36	AMD Opteron 6272 CPUs. CX-3 MX353A (40 Gbps IB) and CX-3 MX313A (40 Gbps RoCE) via PCIe 2.0 x8

Table 2: Cluster configuration

the `PUT` or `GET` request into a per-client memory region in the server. The server polls these regions for new requests. Upon receiving one, the server process executes in conventional fashion using its local data structures. It then sends a reply to the client using messaging verbs: a SEND over an Unreliable Datagram.

To explain why we use this hybrid of RDMA and messaging, we describe the performance experiments and analysis that support it. Particularly, we describe why we prefer using RDMA writes instead of reads, not taking advantage of hardware retransmission by opting for unreliable transports, and using messaging verbs despite conventional wisdom that they are slower than RDMA.

3.1 Notation and experimental setup

In the rest of this paper, we refer to an RDMA read as READ and to an RDMA write as WRITE. In this section, we present microbenchmarks from Emulab's [29] Apt cluster, a large, modern testbed equipped with 56 Gbps InfiniBand. Because Apt has only InfiniBand, in Section 5, we also use the NSF PRObE's [11] Susitna cluster to evaluate on RoCE. The hardware configurations of these clusters are shown in Table 2.

These experiments use one server machine and several client machines. We denote the server machine by M_S and its RNIC by $RNIC_S$. Client machine i is denoted by C_i. The server and client machines may run multiple server and client processes respectively. We call a message from client to server a *request*, and the reply from server to client, a *response*. The host issuing a verb is the *requester* and the destination host *responder*. For unsignaled SEND and WRITE over UC, the destination host does not actually send a response, but we still call it a responder.

For throughput experiments, processes maintain a window of several outstanding verbs in their send queues. Using windows allows us to saturate our RNICs with fewer processes. In all of our throughput experiments, we manually tune the window size for maximum aggregate throughput.

3.2 Using WRITE instead of READ

There are several benefits to using WRITE instead of READ. WRITEs can be performed over the UC transport, which itself confers several performance advantages. Because the responder does not need to send packets back, its RNIC performs less processing, and thus can support higher throughput than with READs. The reduced network bandwidth similarly benefits both the server and client throughput. Finally, as one might expect, the latency of an unsignaled WRITE is about half that ($\frac{1}{2}$ RTT) of a READ. This makes it possible to replace one READ by two WRITEs, one client-to-server and one server-to-client (forming an application-level request-reply pair), without increasing latency significantly.

3.2.1 WRITEs have lower latency than READs

Measuring the latency of an unsignaled WRITE is not straightforward as the requester gets no indication of completion. Therefore, we measure it indirectly by measuring the latency of an *ECHO*. In an ECHO, a client transmits a message to a server and the server relays the same message back to the client. If the ECHO is realized by using unsignaled WRITEs, the latency of an unsignaled WRITE is at most one half of the ECHO's latency.

We also measure the latency of signaled READ and WRITE operations. As these operations are signaled, we use the completion event to measure latency. For WRITE, we also measure the latency with payload inlining.

Figure 2 shows the average latency from these measurements. We use inlined and unsignaled WRITEs for ECHOs. On our RNICs, the maximum size of the inlined payload is 256 bytes. Therefore, the graphs for WR-INLINE and ECHO are only shown up to 256 bytes.

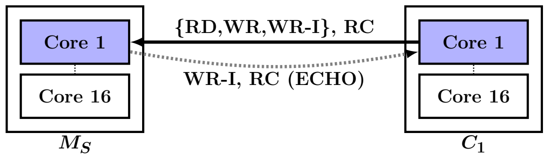
Unsignaled verbs: For payloads up to 64 bytes, the latency of ECHOs is close to READ latency, which confirms that the one-way WRITE latency is about half of the READ latency. For larger ECHOs, the latency increases because of the time spent in writing to the RNIC via PIO.

Signaled verbs: The solid lines in Figure 2 show the latencies for three signaled verbs—WRITE, READ, and WRITE with inlining (WR-INLINE). The latencies for READ and WRITE are similar because the length of the network/PCIe path travelled is identical. By avoiding one DMA operation, inlining reduces the latency of small WRITEs significantly.

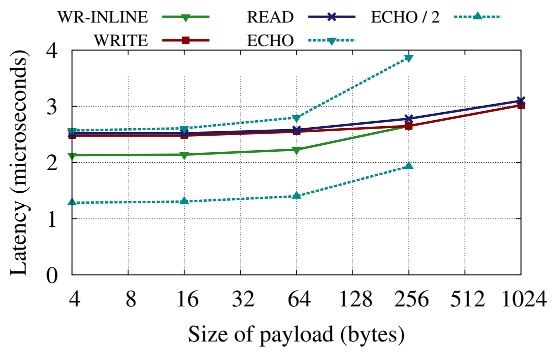
3.2.2 WRITEs have higher throughput than READs

To evaluate throughput, it is first necessary to observe that with many client machines communicating with one server, different verbs perform very differently when used at the clients (talking to one server) and at the server (talking to many clients).

Inbound throughput: First, we measured the throughput for *inbound* verbs, i.e., the number of verbs that multiple remote machines (the clients) can issue to one machine (the server). Using the notation introduced above, $C_1, ..., C_N$ issue operations to M_S as shown in Figure 3a. Figure 3b shows the cumulative throughput observed across the active machines. For up to 128 byte payloads, WRITEs achieve 35 Mops, which is about 34% higher higher than the maximum

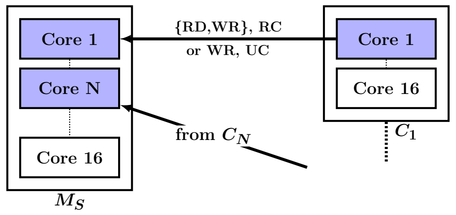


(a) Setup for measuring verbs and ECHO latency. We use one client process to issue operations to one server process

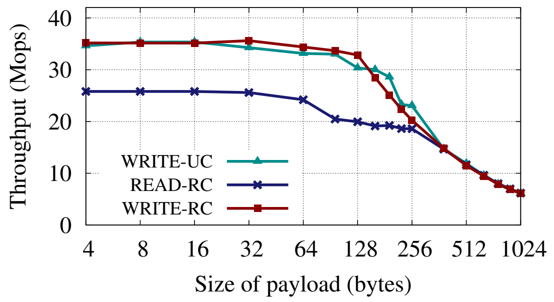


(b) The one-way latency of WRITE is half of the ECHO latency. ECHO operations used unsignaled verbs.

Figure 2: Latency of verbs and ECHO operations

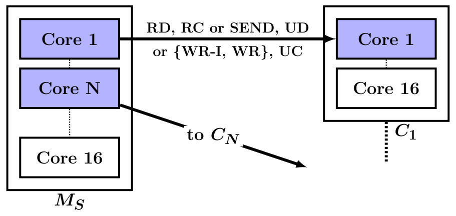


(a) Setup for measuring inbound throughput. Each client process communicates with only one server process

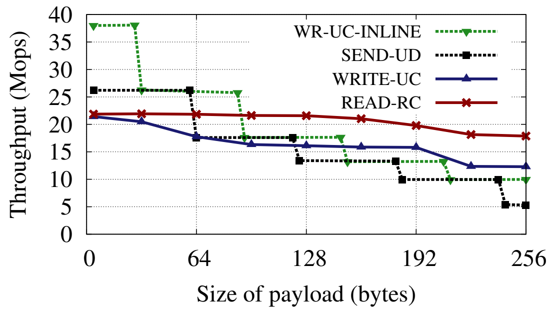


(b) For moderately sized payloads, WRITE has much higher inbound throughput than READs.

Figure 3: Comparison of inbound verbs throughput



(a) Setup for measuring outbound throughput. Each server process communicates with only one client process.



(b) For small payloads, WRITE with inlining has a higher outbound throughput than READ.

Figure 4: Comparison of outbound verbs throughput

READ throughput (26 Mops). Interestingly, reliable WRITEs deliver significantly higher throughput than READs despite their identical InfiniBand path. This is explained as follows: writes require less state maintainance both at the RDMA and the PCIe level because the initiator does not need to wait for a response. For reads, however, the request must be maintained in the initiator's memory till a response arrives. At the RDMA level, each queue pair can only service a few outstanding READ requests (16 in our RNICs). Similarly, at the PCIe level, reads are performed using non-posted transactions, whereas writes use cheaper, posted transactions.

Although the inbound throughput of WRITEs over UC and RC is nearly identical, using UC is still beneficial: It requires less processing at $RNIC_S$, and HERD uses this saved capacity to SEND responses.

Outbound throughput: We next measured the throughput for *outbound* verbs. Here, M_S issues operations to $C_1, ..., C_N$. As shown in Figure 4a, there are N processes on M_S; the i^{th} process communicates with C_i only (the scalability problems associated with all-to-all communication are explained in Section 3.3). Apart from READs, WRITEs, and inlined WRITEs over UC, we also measure the throughput for inlined SENDs over UD for reasons outlined in Section 3.3. Figure 4b plots the throughput achieved by M_S for different payload sizes. For small sizes, inlined WRITEs and SENDs have significantly higher outbound throughput than READs. For large sizes, the throughput of all WRITE and SEND variants is less than for READs, but it is never less than 50% of the READ throughput. Thus, even for these larger items, using a single WRITE (or SEND) for responses remains a better choice than using multiple READs for key-value items.

ECHO throughput is interesting for two reasons. First, it provides an upper bound on the throughput of a key-value cache based on one round trip of communication. Second, ECHOs help characterize the processing power of the RNIC: although the advertised message rate of ConnectX-3 cards is 35 Mops, bidirectionally, they can process many more messages.

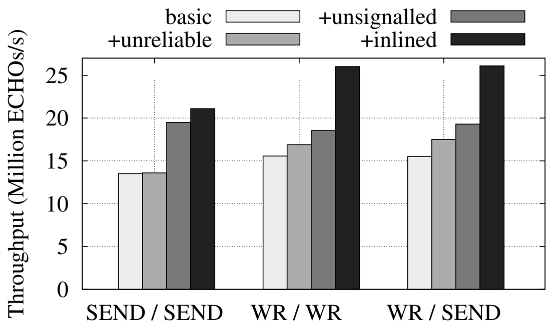


Figure 5: Throughput of ECHOs with 32 byte messages. In WR-SEND, the response is sent over UD.

An ECHO consists of a request message and a response message. Varying the verbs and transport types yield several different implementations of ECHO. Figure 5 shows the throughput for some of the possible combinations and for 32 byte payloads. The figure also shows that using inlining, selective signaling, and UC transport increases the performance significantly.

ECHOs achieve maximum throughput (26 Mops) when both the request and the response are done as RDMA writes. However, as shown in Section 3.3, this approach does not scale with the number of connections. HERD uses RDMA writes (over UC) for requests and SENDs (over UD) for responses. *An ECHO server using this hybrid also achieves 26 Mops—it gives the performance of WRITE-based ECHOs, but with much better scalability.*

By avoiding the overhead of posting RECVs at the server, our method of WRITE based requests and SEND-based responses provides better throughput than purely SEND-based ECHOs. Interestingly, however, after enabling all optimizations, the throughput of purely SEND-based ECHOs (with no RDMA operations) is 21 Mops, which is more than three-fourths of the peak inbound READ throughput (26 Mops). Both Pilaf and FaRM have noted that RDMA reads vastly outperform SEND-based ECHOs, which our results agree with *if our optimizations are removed*. With these optimizations, however, SENDs significantly outperform READs in cases where a single SEND-based ECHO can be used in place of multiple READs per request.

Our experiments show that several ECHO designs, with varying degrees of scalability, can perform better than multiple-READ designs. *From a network-centric perspective, this is fortunate: it also means that designs that use only one cross-datacenter RTT can potentially outperform multiple-RTT designs both in throughput and in latency.*

Discussion of verbs throughput: The ConnectX-3 card is advertised to support 35 million messages per second. Our experiments show that the card can achieve this rate for inbound WRITEs (Figure 3b) and slightly exceed it for very small outbound WRITEs (Figure 4b). All other verbs are slower than 30 Mops regardless of operation size. While the manufacturer does not specify bidirectional message throughput, we know empirically that $RNIC_S$ can service 30 million ECHOs per second (WRITE-based ECHOs achieve 30 Mops with 16 byte payloads; Figure 5 uses 32 byte payloads), or at least 60 total Mops of inbound WRITEs and outbound SENDs.

The reduced throughputs can be attributed to several factors:

- For outbound WRITEs larger than 28 bytes, the RNIC's message rate is limited by the PCIe PIO throughput. The sharp decreases in the WR-UC-INLINE and SEND-UD graphs in Figure 4b at 64 byte intervals are explained by the use of write-combining buffers for PIO acceleration. With the write-combining optimization, the unit of PIO writes is a cache line instead of a word. Due to the larger datagram header, the throughput for SEND-UD drops for smaller payload sizes than for WRITEs
- The maximum throughput for inbound and outbound READs is 26 Mops and 22 Mops respectively, which is considerably smaller than the advertised 35 Mops message rate. Unlike WRITEs, READs are bottlenecked by the RNIC's processing power. This is as expected. Outbound READs involve a PIO operation, a packet transmission, a packet reception, and a DMA write, whereas outbound WRITEs (inlined and over UC) avoid the last two steps. Inbound READs require a DMA read by the RNIC followed by a packet transmission, whereas inbound WRITEs only require a DMA write.

3.3 Using UD for responses

Our previous experiments did not show that as the number of connections increases, connected transports begin to slow down. To reduce hardware cost, power consumption, and design complexity, RNICs have very little on-chip memory (SRAM) to cache address translation tables and queue pair contexts [26]. A miss in this cache requires a PCIe transaction to fetch the data from host memory. When the communication fan-in or fan-out exceeds the capacity of this cache, performance begins to suffer. This is a potentially important effect to avoid both for cluster scaling, but also because it interacts with the cache or store architectural decisions. For example, the cache design we build on in HERD partitions the keyspace between several server processes in order to achieve efficient CPU and memory utilization. Such partitioning further increases the fan-in and fan out of connections to a single machine.

To evaluate this effect, we modified our throughput experiments to enable *all-to-all* communication. We use N client processes (one process each at $C_1,...,C_N$) and N server processes at M_S. For measuring inbound throughput, client processes select a server process at random and issue a WRITE to it. For outbound throughput, a server process selects a client at random and issues a WRITE to it. The results of these experiments for 32 byte messages are presented in Figure 6. Several results stand out:

Outbound WRITEs scale poorly: for $N = 16$, there are 256 active queue pairs at $RNIC_S$ and the server-to-clients throughput degrades to 21% of the maximum outbound WRITE throughput (Figure 4b). With many active queue pairs, each posted verb can cause a cache miss, severely degrading performance.

Inbound WRITEs scale well: Clients-to-server throughput is high even for $N = 16$. The reason for this is that queueing of outstanding verbs operations is performed at the requesting RNIC and very little state is maintained at the responding RNIC. Therefore, the responding RNIC can support a much larger number of active queue pairs without incurring cache misses. The higher requester overhead is amortized because the clients outnumber the server.

In a different experiment, we used 1600 client processes spread over 16 machines to issue WRITEs over UC to one server process. HERD uses this many-to-one configuration to reduce the number of active connections at the server (Section 4.2). This configuration also achieves 30 Mops.

Outbound WRITEs scale poorly only because $RNIC_S$ must manage many connected queue pairs. This problem cannot be solved if we use connected transports (RC/UC/XRC) because they require at least as many queue pairs at M_S as the number of client machines. Scaling outbound communication therefore mandates using datagrams. UD transport supports one-to-many communication, i.e., a single UD queue can be used to issue operations to multiple remote UD queues. The main problem with using UD in a high performance application is that it only supports messaging verbs and not RDMA verbs.

Fortunately, messaging verbs only impose high overhead at the receiver. Senders can directly transmit their requests; only the receiver must pre-post a RECV before the SEND can be handled. For the sender, the work done to issue a SEND is identical to that required to post a WRITE. Figure 6 shows that, when performed over Unreliable Datagram transport, SEND side throughput is high and scales well with the number of connected clients.

The slight degradation of SEND throughput beyond 10 connected clients happens because the SENDs are unsignaled, i.e., server processes get no indication of verb completion. This leads to the server processes overwhelming $RNIC_S$ with too many outstanding operations, causing cache misses inside the RNIC. As HERD uses SENDs for responding to requests, it can use new requests as an indication of the completion of old SENDs, thereby avoiding this problem.

4. DESIGN OF HERD

To evaluate whether these network-driven architectural decisions work for a real key-value application, we designed and implemented an RDMA-based KV cache, called HERD, based upon recent high-performance key-value designs. Our HERD setup consists of one server machine and several client machines. The server machine runs N_S server processes. N_C client processes are uniformly spread across the client machines.

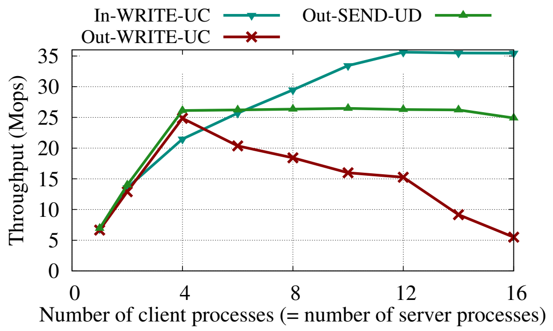


Figure 6: Comparison of UD and UC for all-to-all communication with 32 byte payloads. Inbound WRITEs over UC and outbound SENDs over UD scale well up to 256 queue pairs. Outbound WRITEs over UC scale poorly. All operations are inlined and unsignaled.

4.1 Key-Value cache

The fundamental goal of this work is to evaluate our networking and architectural decisions in the context of key-value systems. We do not focus on building better back-end key-value data structures but rather borrow existing designs from MICA [18].

MICA is a near line-rate key-value cache and store for classical Ethernet. We restrict our discussion of MICA to its cache mode. MICA uses a lossy index to map keys to pointers, and stores the actual values in a circular log. On insertion, items can be evicted from the index (thereby making the index lossy), or from the log in a FIFO order. In HERD, each server process creates an index for 64 Mi keys, and a 4 GB circular log. We use MICA's algorithm for both `GET`s and `PUT`s: each `GET` requires up to two random memory lookups, and each `PUT` requires one.

MICA shards the key space into several partitions based on a keyhash. In its "EREW" mode, each server core has exclusive read and write access to one partition. MICA uses the Flow Director [3] feature of modern Ethernet NICs to direct request packets to the core responsible for the given key. HERD achieves the same effect by allocating per-core request memory at the server, and allowing clients to WRITE their requests directly to the appropriate core.

4.1.1 Masking DRAM latency with prefetching

To service a `GET`, a HERD server must perform two random memory lookups, prepare the SEND response (with the key's value inlined in the WQE), and then post the SEND verb using the `post_send()` function. The memory lookups and the `post_send()` function are the two main sources of latency at the server. Each random memory access takes 60-120 *ns* and the `post_send()` function takes about 150 *ns*. While the latter is unavoidable, we can mask the memory access

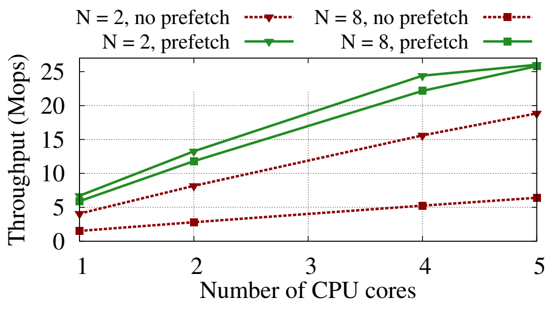


Figure 7: Effect of prefetching on throughput

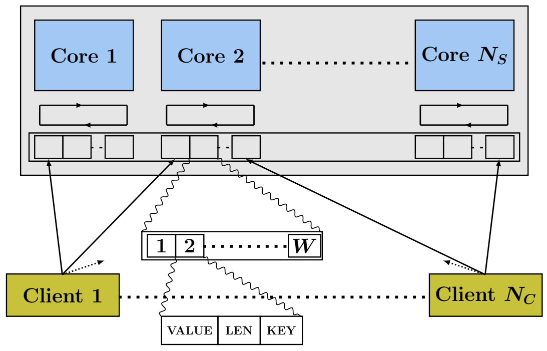


Figure 8: Layout of the request region at the server

latency by overlapping memory accesses of one request with computation of another request.

MICA and CuckooSwitch [18, 31] mask latency by overlapping memory fetches and prefetches, or request decoding and prefetches. HERD takes a different approach: we overlap prefetches with the `post_send()` function used to transmit replies. To process multiple requests simultaneously in the absence of a driver that itself handles batches of packets [2, 18, 31]), HERD creates a pipeline of requests at the application level.

In HERD, the maximum number of memory lookups for each request is two. Therefore, we create a request pipeline with two stages. When a request is in stage *i* of the pipeline, it performs the *i*-th memory access for the request and issues a prefetch for the next memory address. In this way, requests only access memory for which a prefetch has already been issued. On detecting a new request, the server issues a prefetch for the request's index bucket, advances the old requests in the pipeline, pushes in the new request, and finally calls `post_send()` to SEND a reply for the pipeline's completed request. The server process expects the issued prefetches to finish by the time `post_send()` returns.

Figure 7 shows the effectiveness of prefetching. We use a WRITE/SEND-based ECHO server but this time the server performs N random memory accesses before sending the response. Prefetching allows fewer cores to deliver higher throughput: 5 cores can deliver the peak throughput even with $N = 8$. We conclude that there is significant headroom to implement more complex key-value applications, for instance, key-value stores, on top of HERD's request-reply communication mechanism.

With a large number of server processes, this pipelining scheme can lead to a deadlock. A server does not advance its pipeline until it receives a new request, and a client does not advance its request window until it gets a response. We avoid this deadlock as follows. While polling for new requests, if a server fails for 100 iterations consecutively, it pushes a no-op into the pipeline.

4.2 Requests

Clients WRITE their `GET` and `PUT` requests to a contiguous memory region on the server machine which is allocated during initialization. This memory region is called the *request region* and is shared among all the server processes by mapping it using `shmget()`. The request region is logically divided into 1 KB slots (the maximum size of a key-value item in HERD is 1 KB).

Requests are formatted as follows. A `GET` request consists only of a 16-byte keyhash. A `PUT` request contains a 16-byte keyhash, a 2-byte LEN field (specifying the value's length), and up to 1000 bytes for the value. To poll for incoming requests, we use the left-to-right ordering of the RNIC's DMA writes [16, 8]. We use the keyhash field to poll for new requests; therefore, the key is written to the rightmost 16 bytes of the 1 KB slot. A non-zero keyhash indicates a new request, so we do not allow the clients to use a zero keyhash. The server zeroes out the keyhash field of the slot after sending a response, freeing it up for a new request.

Figure 8 shows the layout of the request region at the server machine. It consists of separate chunks for each server process which are further sub-divided into per-client chunks. Each per-client chunk consists of W slots, i.e., each client can have up to W pending requests to each server process. The size of the request region is $N_S \cdot N_C \cdot W$ KB. With N_C = 200, N_S = 16 and W = 2, this is approximately 6 MB and fits inside the server's L3 cache. Each server process polls the per-client chunks for new requests in a round robin fashion. If server process s has seen r requests from client number c, it polls the request region at the request slot number $s \cdot (W \cdot N_c) + (c \cdot W) + r \bmod W$.

A network configuration using bidirectional, all-to-all, communication with connected transports would require $N_C \cdot N_S$ queue pairs at the server. HERD, however, uses connected transports for only the request side of communication, and thus requires only N_C connected queue pairs. The configuration works as follows. An initializer process creates the request region, registers it with $RNIC_S$, establishes a UC connection with each client, and goes to sleep. The N_S server processes then map the request region into their address space via `shmget()` and do not create any connections for receiving requests.

4.3 Responses

In HERD, responses are sent as SENDs over UD. Each client creates N_S UD queue pairs (QPs) whereas each server process uses only one UD QP. Before writing a new request to server process s, a client posts a RECV to its s-th UD QP. This RECV specifies the memory area on the client where the server's response will be written. Each client allocates a *response region* containing $W \cdot N_S$ response slots: this region is used for the target addresses in the RECVs. After writing out W requests, the client starts checking for responses by polling for RECV completions. On each successful completion, it posts another request.

The design outlined thus far deliberately shifts work from the server's RNIC to the client's, with the assumption that client machines often perform enough other work that saturating 40 or 56 gigabits of network bandwidth is not their primary concern. The servers, however, in an application such as Memcached, are often dedicated machines, and achieving high bandwidth is important.

5. EVALUATION

We evaluate HERD on two clusters: Apt and Susitna (Table 2). Due to limited space, we restrict our discussion to Apt and only present graphs for RoCE on Susitna. A detailed discussion of our results on Susitna may be found in [15]. Although Susitna uses similar RNICs as Apt, the slower PCIe 2.0 bus reduces the throughput of all compared systems. Despite this, our results on Susitna remain interesting: just as ConnectX-3 cards overwhelm PCIe 2.0 x8, we expect the next-generation Connect-IB cards to overwhelm PCIe 3.0 x16. Our evaluation shows that:

- HERD uses the full processing power of the RNIC. A single HERD server can process up to 26 million requests per second. For value size up to 60 bytes, HERD's request throughput is *greater than native READ throughput* and is much greater than that of READ-based key-value services: it is over 2X higher than FaRM-KV and Pilaf.
- HERD delivers up to 26 Mops with approximately 5 μs average latency. Its latency is over 2X lower than Pilaf and FaRM-KV at their peak throughput respectively.
- HERD scales to the moderately sized Apt cluster, sustaining peak throughput with over 250 connected client processes.

We conclude the evaluation by examining the seeming drawback of the HERD design relative to READ-based designs—its higher server CPU use—and put this in context with the *total* (client + server) CPU required by all systems.

5.1 Experimental setup

We run all our throughput and latency experiments on 18 machines in Apt. The 17 client machines run up to 3 client processes each. With at most 4 outstanding requests per client, our implementation requires at least 36 client processes to saturate the server's throughput. We over-provision slightly by using 51 client processes. The server machine runs 6 server processes, each pinned to a distinct physical core. The machine configuration is described in Table 2. The machines run Ubuntu 12.04 with Mellanox's OFED v2.2 stack.

Comparison against stripped-down alternatives: In keeping with our focus on understanding the effects of network-related decisions, we compare our (full) HERD implementation against simplified implementations of Pilaf and FaRM-KV. These simplified implementations use the same communication methods as the originals, but *omit* the actual key-value storage, instead returning a result instantly. We made this decision for two reasons. First, while working with Pilaf's code, we observed several optimization opportunities; we did not want our evaluation to depend on the relative performance tuning of the systems. Second, we did not have access to the FaRM source code, and we could not run Windows Server on our cluster. Instead, we created and evaluated emulated versions of the two systems which do not include their backing data structures. This approach gives these systems the maximum performance advantage possible, so the throughput we report for both Pilaf and FaRM-KV may be higher than is actually achievable by those systems.

Pilaf is based on 2-level lookups: a hash-table maps keys to pointers. The pointer is used to find the value associated with the key from flat memory regions called *extents*. FaRM-KV, in its default operating mode, uses single-level lookups. It achieves this by inlining the value in the hash-table. It also has a two-level mode, where the value is stored "out-of-table." Because the out-of-table mode is necessary for memory efficiency with variable length keys, we compare HERD against both modes. In the following two subsections, we denote the size of a key, value, and pointer by S_K, S_V, and S_P respectively.

5.1.1 *Emulating Pilaf*

In *K-B* cuckoo hashing, every key can be found in K different buckets, determined by K orthogonal hash functions. For associativity, each bucket contains B slots. Pilaf uses 3-1 cuckoo hashing with 75% memory efficiency and 1.6 average probes per `GET` (higher memory efficiency with fewer, but slightly larger, average probes is possible with 2-4 cuckoo hashing [9]). When reading the hash index via RDMA, the smallest unit that must be read is a bucket. A bucket in Pilaf has only one slot that contains a 4 byte pointer, two 8 byte checksums, and a few other fields. We assume the bucket size in Pilaf to be 32 bytes for alignment.

`GET`: A `GET` in Pilaf consists of 1.6 bucket READs (on average) to find the value pointer, followed by a S_V byte READ to fetch the value. It is possible to reduce Pilaf's latency by issuing concurrent READs for both cuckoo buckets. As this comes at the cost of decreased throughput, we wait for the first READ to complete and issue the second READ only if it is required.

PUT: For a PUT, a client SENDs a $S_K + S_V$ byte message containing the new key-value item to the server. This request may require relocating entries in the cuckoo hash-table, but we ignore that as our evaluation focuses on the network communication only.

In emulating Pilaf, we enable all of our RDMA optimizations for both request types; we call the resulting system Pilaf-em-OPT.

5.1.2 *Emulating FaRM-KV*

FaRM-KV uses a variant of Hopscotch hashing to locate a key in approximately one READ. Its algorithm guarantees that a key-value pair is stored in a small neighborhood of the bucket that the key hashes to. The size of the neighborhood is tunable, but its authors set it to 6 to balance good space utilization and performance for items smaller than 128 bytes. FaRM-KV can inline the values in the buckets, or it can store them separately and only store pointers in the buckets. We call our version of FaRM-KV with inlined values FaRM-em and without inlining FaRM-em-VAR (for variable length values).

GET: A GET in FaRM-em requires a 6 * $(S_K + S_V)$ byte READ. In FaRM-em-VAR, a GET requires a 6 * $(S_K + S_P)$ byte READ followed by a S_V byte READ.

PUT: FaRM-KV handles PUTs by sending messages to the server via WRITEs, similar to HERD. The server notifies the client of PUT completion using another WRITE. Therefore, a PUT in FaRM-em (and FaRM-em-VAR) consists of one $S_K + S_V$ byte WRITE from a client to the server, and one WRITE from the server to the client. For higher throughput, we perform these WRITEs over UC unlike the original FaRM paper that used RC (Figure 5).

5.2 Workloads

Three main workload parameters affect the throughput and latency of a key-value system: relative frequency of PUTs and GETs, item size, and skew.

We use two types of workloads: *read-intensive* (95% GET, 5% PUT) and *write-intensive* (50% GET, 50% PUT). Our workload can either be *uniform* or *skewed*. Under a uniform workload, the keys are chosen uniformly at random from the 16 byte keyhash space. The skewed workload draws keys from a Zipf distribution with parameter .99. This workload is generated offline using YCSB [7]. We generated 480 million keys once and assigned 8 million keys to each of the 51 client processes.

5.3 Throughput comparison

We now compare the end-to-end throughput of HERD against the emulated versions of Pilaf and FaRM.

Figure 9 plots the throughput of these system for read-intensive and write-intensive workloads for 48-byte items (S_K = 16, S_V = 32). We chose this item size because it is representative of real-life workloads: an analysis of Facebook's general-purpose key-value store [6] showed that the 50-th percentile of key sizes is approximately 30 bytes, and that of value sizes is 20 bytes. To compare the READ-based GETs of Pilaf and FaRM with Pilaf's SEND/RECV-based PUTs, we also plot the throughput when the workload consists of 100% PUTs.

In HERD, both read-intensive and write-intensive workloads achieve 26 Mops, which is slightly larger than the throughput of native RDMA reads of a similar size (Figure 3b). For small key-value items, there is very little difference between PUT and GET requests at the RDMA layer because both types of requests fit inside one cacheline. Therefore, the throughput does not depend on the workload composition.

The GET throughput of Pilaf-em-OPT and FaRM-em(-VAR) is directly determined by the throughput of RDMA READs. A GET in Pilaf-em-OPT involves 2.6 READs (on average). Its GET throughput is 9.9 Mops, which is about 2.6X smaller than the maximum READ throughput. For GETs, FaRM-em requires a single 288 byte READ and delivers 17.2 Mops. FaRM-em-VAR requires a second READ and has throughput of 11.4 Mops for GETs.

Surprisingly, the PUT throughput in our emulated systems is much larger than their GET throughput. This is explained as follows. In FaRM-em(-VAR), PUTs use small WRITEs over UC that outperform the large READs required for GETs. Pilaf-em-OPT uses SEND/RECV-based requests and replies for PUT. Both Pilaf and FaRM assume that messaging-based ECHOs are much more expensive than READs. (Pilaf reports that for 17 byte messages, the throughput of RDMA reads is 2.449 Mops whereas the throughput of SEND/RECV-based ECHOs is only 0.668 Mops.) If SEND/RECV can provide only one fourth the throughput of READ, it makes sense to use multiple READs for GET.

However, we believe that these systems do not achieve the full capacity of SEND/RECV. After optimizing SENDs by using unreliable transport, payload inlining, and selective signaling, SEND/RECV based ECHOs, as shown in Figure 5, achieve 21 Mops, which is considerably more than half of our READ throughput (26 Mops). Therefore, we conclude that SEND/RECV-based communication, when used effectively, is more efficient than using multiple READs per request.

Figure 10 shows the throughput of the three systems with 16 byte keys and different value sizes for a read-intensive workload. For up to 60-byte items, HERD delivers over 26 Mops, which is slightly greater than the peak READ throughput. Up to 32-byte values, FaRM-em also delivers high throughput. However, its throughput declines quickly with increasing value size because the size of FaRM-em's READs grow rapidly (as 6 * $(S_V + 16)$). This problem is fundamental to the Hopscotch-based KV design which amplifies the READ size to reduce round trips. FaRM-KV quickly saturates link bandwidths (PCIe or InfiniBand/RoCE) with smaller items

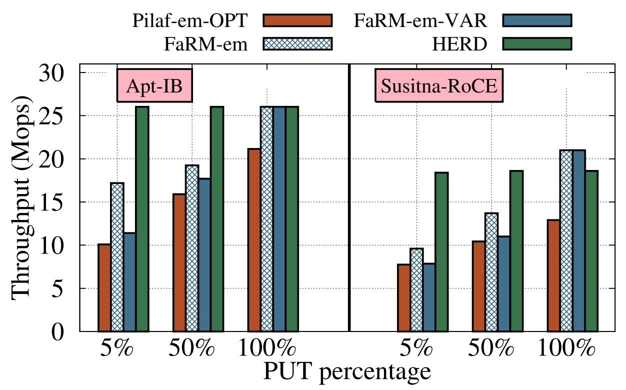


Figure 9: End-to-end throughput comparison for 48 byte key-value items

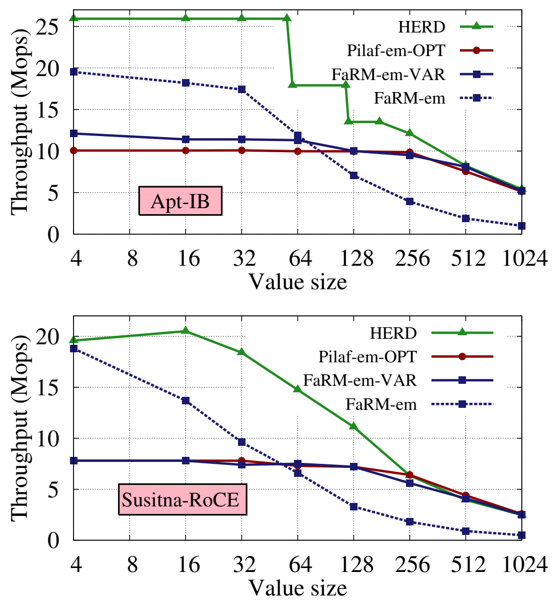


Figure 10: End-to-end throughput comparison with different value sizes

than HERD, which conserves network bandwidth by transmitting only essential data. Figure 10 illustrates this effect. FaRM-em saturates the PCIe 2.0 bandwidth on Susitna with 4 byte values, and the 56 Gbps InfiniBand bandwidth on Apt with 32 byte values. HERD achieves high performance for up to 32 byte values on Susitna, and 60 bytes values on Apt, and is bottlenecked by the smaller PCIe PIO bandwidth.

With large values (144 bytes on Apt, 192 on Susitna), HERD switches to using non-inlined SENDs for responses. The outbound throughput of large inlined messages is less than non-inlined messages because DMA outperforms PIO for large payloads (Figure 4b). For large values, the performance of HERD, FaRM-em, and Pilaf-em-OPT are within 10% of each other.

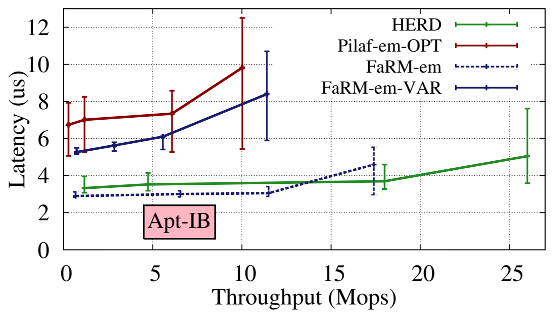


Figure 11: End-to-end latency with 48 byte items and read-intensive workload

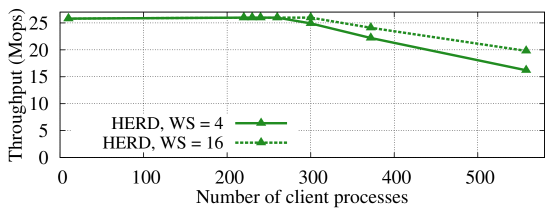


Figure 12: Throughput with variable number of client processes and different window sizes

5.4 Latency comparison

Unlike FaRM-KV and Pilaf, HERD uses only one network round trip for any request. FaRM-KV and Pilaf use one round trip for PUT requests but require multiple round trips for GETs (except when FaRM-KV inlines values in the hashtable). This causes their GET latency to be higher than the latency of a single RDMA READ.

Figure 11 compares the average latencies of the three systems for a read-intensive workload; the error bars indicate the 5th and 95th percentile latency. To understand the dependency of latency on throughput, we increase the load on the server by adding more clients until the server is saturated. When using 6 CPU cores at the server, HERD is able to deliver 26 million requests per second with approximately 5 μs average latency. For fixed-length key-value items, FaRM-em provides the lowest latency among the three systems because it requires only one network round trip (unlike Pilaf-em-OPT) and no computation at the server (unlike HERD). For variable length values, however, FaRM's variable length mode requires two RTTs, yielding worse latency than HERD.

The PUT latency for all three systems (not shown) is similar because the network path traversed is the same. The measured latency for HERD was slightly higher than that of the emulated systems because it performed actual hash table and memory manipulation for inserts, but this is an artifact of the performance advantage we give Pilaf-em and FaRM-em.

5.5 Scalability

We conducted a larger experiment to understand HERD's number-of-clients scalability. We used one machine to run 6 server processes and the remaining 186 machines for client processes. The experiment uses 16 byte keys and 32 byte values.

Figure 12 shows the results from this experiment. HERD delivers its maximum throughput for up to 260 client processes. With even more clients, HERD's throughput starts decreasing almost linearly. The rate of decrease can be reduced by increasing the number of outstanding requests maintained by each client, at the cost of higher request latency. Figure 12 shows the results for two window sizes: 4 (HERD's default) and 16. This observation suggests that the decline is due to cache misses in $RNIC_S$, as more outstanding verbs in a queue can reduce cache pressure. We expect this scalability limit to be resolved with the introduction of Dynamically Connected Transport in the new Connect-IB cards [1, 8],

Another likely scalability limit of our current HERD design is the round-robin polling at the server for requests. With thousands of clients, using WRITEs for inbound requests may incur too much CPU overhead; mitigating this effect may necessitate switching to a SEND/SEND architecture over Unreliable Datagram transport. Figure 5 shows there is a 4-5 Mops decrease to this change, but once made, the system should scale up to many thousands of clients, while still outperforming an RDMA READ-based architecture.[1] We expect the performance of the SEND/SEND architecture relative to WRITE-SEND to increase with the introduction of inlined RECVs in Connect-IB cards. This will reduce the load on RNICs by encapsulating the RECV payload in the RECV completion.

5.6 HERD CPU Use

The primary drawback of not using READs in HERD is that `GET` operations require the server CPU to execute requests, in exchange for saving one cross-datacenter RTT. While at first glance, it might seem that HERD's CPU usage should be higher than Pilaf and FaRM-KV, we show that in practice these two systems also have significant sources of CPU usage that reduce the extent of the difference.

First, issuing extra READs adds CPU overhead at the Pilaf and FaRM-KV clients. To issue the second READ, the clients must poll for the first READ to complete. HERD shifts this overhead to the server's CPU, making more room for application processing at the clients.

Second, handling `PUT` requests requires CPU involvement at the server. Achieving low-latency `PUT`s requires dedicating server CPU cores that poll for incoming requests. Therefore, the exact CPU use depends on the fraction of `PUT` throughput that server is *provisioned* for, because this determines the CPU resources that must be allocated to it, not the dynamic amount actually used. For example, our experiments show that, even ignoring the cost of updating data structures, provisioning for 100% `PUT` throughput in Pilaf and FaRM-KV requires over 5

[1]Figure 5 uses SENDs over UC, but we have verified that similar throughput is possible using SENDs over UD.

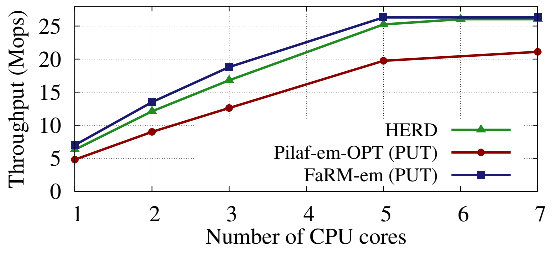


Figure 13: Throughput as a function of server CPU cores

CPU cores. Figure 13 shows FaRM-em and Pilaf-em-OPT's `PUT` throughput for 48 byte key-value items and different numbers of CPU cores at the server. Pilaf-em-OPT's CPU usage is higher because it must post RECVs for new `PUT` requests, which is more expensive than FaRM-em's request-region polling.

In Figure 13, we also plot HERD's throughput for the same workload by varying the number of server CPU cores. HERD is able to deliver over 95% of its maximum throughput with 5 CPU cores. The modest gap to FaRM-em arises because the HERD server in this experiment is handling hash table lookups and updates, whereas the emulated FaRM-KV is handling only the network traffic.

We believe, therefore, that HERD's higher throughput and lower latency, along with the significant CPU utilization in Pilaf and FaRM-KV, justifies the architectural decision to have the CPU involved on the `GET` path for small key-value items. For a 50% `PUT` workload, for example, the moderate extra cost of adding a few more cores—or using the already-idle cycles on the cores—is likely worthwhile for many applications.

5.7 Resistance to skew

To understand how HERD's behavior is impacted by skew, we tested it with a workload where the keys are drawn from a Zipf distribution. HERD adapts well to skew, delivering its maximum performance even when the Zipf parameter is .99. HERD's resistance to skew comes from two factors. First, the back-end MICA architecture [18] that we use in HERD performs well under skew; a skewed workload spread across several partitions produces little variation in the partitions' load compared to the skew in the workload's distribution. Under our Zipf-distributed workload, with 6 partitions, the most loaded CPU core is only 50% more so than the least loaded core, even though the most popular key is over 10^5 times more popular than the average.

Second, because the CPU cores share the RNIC, the highly loaded cores are able to benefit from the idle time provided by the less-used cores. Figure 13 demonstrates this effect: with a *uniform* workload and using only a single core, HERD can deliver 6.3 Mops. When the system is configured to use 6 cores—the minimum required by HERD to deliver its peak throughput—the system delivers 4.32 Mops *per core*. The per-core performance reduction is not because of a CPU

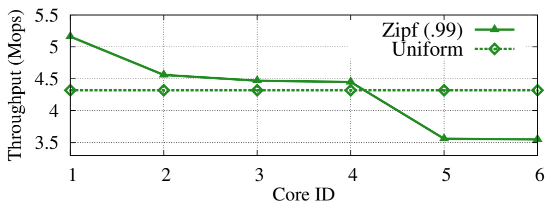


Figure 14: Per-core throughput under skewed and uniform workloads. Note that the y-axis does not begin at 0.

bottleneck, but because the server processes saturate the PCIe PIO throughput. Therefore, even if the workload is skewed, there is ample CPU headroom on a given core to handle the extra requests.

Figure 14 shows the per-core throughput of HERD for a skewed workload. The experimental configuration is: 48-byte items, read-intensive, skewed workload, 6 total CPU cores. The per-core throughput for a uniform workload is included for comparison.

6. RELATED WORK

RDMA-based key-value stores: Other than Pilaf and FaRM, several projects have designed memcached-like systems over RDMA. Panda et al. [14] describe a memcached implementation using a hybrid of UD and RC transports. It uses SEND/RECV messages for all requests and avoids the overhead of UD transport (caused by a larger header size than RC) by actively switching connections between RC and UD. Although their cluster (ConnectX, 32 Gbps) is comparable to Susitna (ConnectX-3, 40 Gbps), their request rate is less than 1.5 Mops. Stuedi et al. [25] describe a SoftiWARP [28] based version of memcached targeting CPU savings in wimpy nodes with 10GbE.

Accelerating systems with RDMA: Several projects have used verbs to improve the performance of systems such as HBase, Hadoop RPC, PVFS [30, 13, 20]. Most of these use only SEND/RECV verbs as a fast alternative to socket-based communication. In a PVFS implementation over InfiniBand [30], `read()` and `write()` operations in the filesystem use both RDMA and SEND/RECV. They favor WRITEs over READs for the same reasons as in our work, suggesting that the performance gap has existed over several generations of InfiniBand hardware. There have been several versions of MPI over InfiniBand [16, 19]. MPICH2 uses RDMA writes for one-sided messaging: the server polls the head of a circular buffer that is written to by a client. HERD extends this messaging in a scalable fashion for all-to-all request-reply communication. While [30, 13, 20, 16, 19] have benchmarked verbs performance before, it has been for large messages in the context of applications like NFS and MPI. Our work exploits the performance differences that appear only for small messages and are relevant for message rate-bound applications like key-value stores.

User level networking: Taken together, we believe that one conclusion to draw from the union of HERD, Pilaf, FaRM, and MICA [18] is that the biggest boost to throughput comes from bypassing the network stack and avoiding CPU interrupts, *not* necessarily from bypassing the CPU entirely. All four of these systems use mechanisms to allow user-level programs to directly receive requests or packets from the NIC: the userlevel RDMA drivers for HERD, Pilaf, and FaRM, and the Intel DPDK library for MICA. As we discuss below, the *throughput* of these systems is similar, but the batching required by the DPDK-based systems confers a latency advantage to the hardware-supported InfiniBand systems. These lessons suggest profitable future work in making user-level classical Ethernet systems more portable, easier to use, and lower-latency. One ongoing effort is NIQ [10], an FPGA-based low-latency NIC which uses cacheline-sized PIOs (without any DMA) to transmit and receive small packets. Inlined WRITEs in RDMA use the same mechanism at the requesters's side.

General key-value stores: MICA [18] is a recent key-value system for classical Ethernet. It assigns exclusive partitions to server cores to minimize memory contention, and exploits the NIC's capability to steer requests to the responsible core [3]. A MICA server delivers 77 Mops with 4 dual-port, 10 Gbps PCIe 2.0 NICs, with 50 µs average latency (19.25 Gbps with one PCIe 2.0 card). This suggests that, comparing the state-of-the-art, classical Ethernet-based solutions can provide comparable throughput to RDMA-based solutions, although with much higher latency. RAMCloud [23] is a RAM-based, persistent key-value store that uses messaging verbs for low latency communication.

7. CONCLUSION

This paper explored the options for implementing fast, low-latency key-value systems atop RDMA, arriving at an unexpected and novel combination that outperforms prior designs and uses fewer network round-trips. Our work shows that, contrary to widely held beliefs about engineering for RDMA, single-RTT designs with server CPU involvement can outperform the "optimization" of CPU-bypassing remote memory access when the RDMA approaches require multiple RTTs. These results contribute not just a practical artifact—the HERD low-latency, high-performance key-value cache—but an improved understanding of how to use RDMA to construct future DRAM-based storage services.

Acknowledgements We thank our shepherd Chuanxiong Guo and the anonymous reviewers for their feedback that helped improve the paper. We also thank Miguel Castro and Dushyanth Narayanan for discussing FaRM with us, Kirk Webb and Robert Ricci for getting us early access to the Apt cluster, and Hyeontaek Lim for his valuable insights. This work was supported by funding from the National Science Foundation under awards CNS-1314721 and CCF-0964474, and Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). The PRObE cluster [11] used for many experiments is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE).

References

[1] Connect-IB: Architecture for Scalable High Performance Computing. URL http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf.

[2] Intel DPDK: Data Plane Development Kit. URL http://dpdk.org.

[3] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. URL http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[4] Redis: An Advanced Key-Value Store. URL http://redis.io.

[5] memcached: A Distributed Memory Object Caching System, 2011. URL http://memcached.org.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*, 2012.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[8] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *USENIX NSDI*, 2014.

[9] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX NSDI*, 2013.

[10] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX ATC*, 2013.

[11] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[12] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *DISC*, 2008.

[13] J. Huang, X. Ouyang, J. Jose, M. W. ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *IPDPS*, 2012.

[14] J. Jose, H. Subramoni, K. C. Kandalla, M. W. ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *CCGRID*. IEEE, 2012.

[15] A. Kalia, D. G. Andersen, and M. Kaminsky. Using RDMA Efficiently for Key-Value Services. In *Technical Report CMU-PDL-14-106*, 2014.

[16] J. Li, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.

[17] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*, 2011.

[18] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX NSDI*, 2014.

[19] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *IPDPD*, 2004.

[20] X. Lu, N. S. Islam, M. W. ur Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In *ICPP*, 2013.

[21] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, 2013.

[22] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *USENIX NSDI*, 2013.

[23] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[24] R. Pagh and F. F. Rodler. Cuckoo Hashing. *J. Algorithms*, 2004.

[25] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *USENIX ATC*, 2012.

[26] S. Sur, A. Vishnu, H.-W. Jin, W. Huang, and D. K. Panda. Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems? In *HOTI*, 2005.

[27] S. Sur, M. J. Koop, L. Chai, and D. K. Panda. Performance Analysis and Evaluation of Mellanox ConnectX Infiniband Architecture with Multi-Core Platforms. In *HOTI*, 2007.

[28] A. Trivedi, B. Metzler, and P. Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *APSys*, 2011.

[29] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

[30] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Ohio State University Tech Report*, 2003.

[31] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*, 2013.

Raising the Bar for Using GPUs in Software Packet Processing

Anuj Kalia, Dong Zhou, Michael Kaminsky*, and David G. Andersen
*Carnegie Mellon University and *Intel Labs*

Abstract

Numerous recent research efforts have explored the use of Graphics Processing Units (GPUs) as accelerators for software-based routing and packet handling applications, typically demonstrating throughput several times higher than using legacy code on the CPU alone.

In this paper, we explore a new hypothesis about such designs: For many such applications, the benefits arise less from the GPU hardware itself as from the expression of the problem in a language such as CUDA or OpenCL that facilitates memory latency hiding and vectorization through massive concurrency. We demonstrate that in several cases, after applying a similar style of optimization to algorithm implementations, a CPU-only implementation is, in fact, *more resource efficient* than the version running on the GPU. To "raise the bar" for future uses of GPUs in packet processing applications, we present and evaluate a preliminary language/compiler-based framework called G-Opt that can accelerate CPU-based packet handling programs by automatically hiding memory access latency.

1 Introduction

The question of matching hardware architectures to networking requirements involves numerous trade-offs between flexibility, the use of off-the-shelf components, and speed and efficiency. ASIC implementations are fast, but relatively inflexible once designed, and must be produced in large quantities to offset the high development costs. Software routers are as flexible as code, but have comparatively poor performance, in packets-per-second (pps), as well as in cost (pps/$) and energy efficiency (pps/watt). Both ends of the spectrum are successful: Software-based firewalls are a popular use of the flexibility and affordability of systems up to a few gigabits per second; commodity Ethernet switches based on high-volume ASICs achieve seemingly unbeatable energy and cost efficiency.

In the last decade, several potential middle grounds emerged, from network forwarding engines such as the Intel IXP, to FPGA designs [12], and, as we focus on in this paper, to the use of commodity GPUs. Understanding the advantages of these architectures, and how to best exploit them, is important both in research (software-based implementations are far easier to experiment with) and in practice (software-based approaches are used for low-speed applications and in cases such as forwarding within virtual switches [13]).

Our goal in this paper is to advance understanding of the advantages of GPU-assisted packet processors compared to CPU-only designs. In particular, noting that several recent efforts have claimed that GPU-based designs can be faster even for simple applications such as IPv4 forwarding [23, 43, 31, 50, 35, 30], we attempt to identify the *reasons* for that speedup. At the outset of this work, we hypothesized that much of the advantage came from the way the GPUs were *programmed*, and that less of it came from the fundamental hardware advantages of GPUs (computational efficiency from having many processing units and huge memory bandwidth).

In this paper, we show that this hypothesis appears correct. Although GPU-based approaches are faster than a straightforward implementation of various forwarding algorithms, it is possible to transform the CPU implementations into a form that is more resource efficient than GPUs.

For many packet processing applications, the key advantage of a GPU is *not* its computational power, but that it can transparently hide the 60-200ns of latency required to retrieve data from main memory. GPUs do this by exploiting massive parallelism and using fast hardware thread switching to switch between sets of packets when one set is waiting for memory. We demonstrate that insights from code optimization techniques such as group prefetching and software pipelining [17, 51] apply to typical CPU packet handling code to boost its performance. In many cases, the CPU version is more resource efficient than the GPU, and delivers lower latency because it does not incur the additional overhead of transferring data to and from the GPU.

Finally, to make these optimizations more widely usable, both in support of practical implementations of software packet processing applications, and to give future research a stronger CPU baseline for comparison, we present a method to automatically transform data structure lookup code to overlap its memory accesses and computation. This automatically transformed code is up to 1.5-6.6x faster than the baseline code for several common

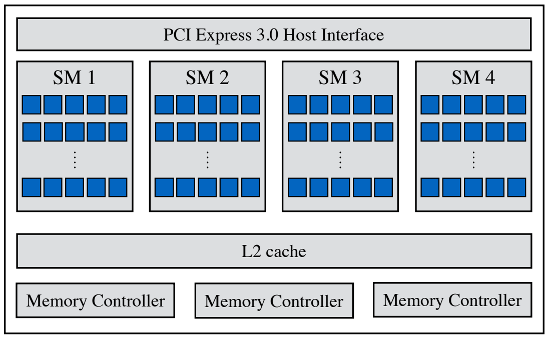


Figure 1: Simplified architecture of an NVIDIA GTX 650. The global memory is not shown.

lookup patterns, and its performance is within 10% of our hand-optimized version. By applying these optimizations, we hope to "raise the bar" for future architectural comparisons against the baseline CPU-based design.

2 Strengths and weaknesses of GPUs for packet processing

In this section, we first provide relevant background on GPU architecture and programming, and discuss the reasons why previous research efforts have used GPUs as accelerators for packet processing applications. Then, we show how the fundamental differences between the requirements of packet processing applications and conventional graphics applications make GPUs less attractive for packet processing than people often assume. Throughout this paper, we use NVIDIA and CUDA's terminology for GPU architecture and programming model, but we believe that our discussion and conclusions apply equally to other discrete GPUs (e.g., GPUs using OpenCL).

2.1 GPU strengths: vectorization and memory latency hiding

A modern CUDA-enabled GPU (Figure 1) consists of a large number of processing cores grouped into Streaming Multiprocessors (SMs). It also contains registers, a small amount of memory in a cache hierarchy, and a large global memory. The code that runs on a GPU is called a *kernel*, and is executed in groups of 32 threads called *warps*. The threads in a warp follow a SIMT (Single Instruction, Multiple Thread) model of computation: they share an instruction pointer and execute the same instructions. If the threads "diverge" (i.e., take different execution paths), the GPU selectively disables the threads as necessary to allow them to execute correctly.

Vectorization: The large number of processing cores on a GPU make it attractive as a vector processor for packets. Although network packets do have some inter-packet ordering requirements, most core networking functions such as lookups, hash computation, or encryption can be executed in parallel for multiple packets at a time. This parallelism is easily accessible to the programmer through well-established GPU-programming frameworks such as CUDA and OpenCL. The programmer writes code for a single thread; the framework automatically runs this code with multiple threads on multiple processors.

Comparison with CPUs: The AVX2 vector instruction set in the current generation of Intel processors has 256-bit registers that can process 8 32-bit integers in parallel. However, the programming language support for CPU-based vectorization is still maturing [8].

Memory latency hiding: Packet processing applications often involve lookups into large data structures kept in DRAM. Absent latency-hiding, access to these structures will stall execution while it completes (300-400 cycles for NVIDIA's GPUs). Modern GPUs hide latency using hardware. The warp scheduler in an SM holds up to 64 warps to run on its cores. When threads in a warp access global memory, the scheduler switches to a different warp. Each SM has *thousands* of registers to store the warp-execution context so that this switching does not require explicitly saving and restoring registers.

Comparison with CPUs: Three architectural features in modern CPUs enable memory latency hiding. First, CPUs have a small number of hardware threads (typically two) that can run on a single core, enabling ongoing computation when one thread is stalled on memory. Unfortunately, while each core can maintain up to ten outstanding cache misses [51], hyperthreading can only provide two "for free". Second, CPUs provide both hardware and software-managed prefetching to fetch data from DRAM into caches before it is needed. And third, after issuing a DRAM access, CPUs can continue executing independent instructions using out-of-order execution. These features, however, are less able to hide latency in unmodified code than the hardware-supported context switches on GPUs, and leave ample room for improvement using latency-hiding code optimizations (Section 3).

2.2 GPU weaknesses: setup overhead and random memory accesses

Although GPUs have attractive features for accelerating packet processing, two requirements of packet processing applications make GPUs a less attractive choice:

Many networking applications require low latency. For example, it is undesirable for a software router in a datacenter to add more than a few microseconds of latency [20]. In the measurement setup we use in this paper, the RTT through an unloaded CPU-based forwarder is 16µs. Recent work in high-performance packet processing reports numbers from 12 to 40µs [32, 51].

Unfortunately, merely communicating from the CPU to the GPU and back may add more latency than the total RTT of these existing systems. For example, it takes ∼ 15µs to transfer one byte to and from a GPU, and ∼ 5µs to launch the kernel [33]. Moreover, GPU-accelerated systems must assemble large batches of packets to process on the GPU in order to take advantage of their massive parallelism and amortize setup and transfer costs. This batching further increases latency.

Networking applications often require random memory accesses into data structures, but the memory subsystem in GPUs is optimized for contiguous access. Under random accesses, GPUs lose a significant fraction of their memory bandwidth advantage over CPUs.

We now discuss these two factors in more detail. Then, keeping these two fundamental factors in mind, we perform simple experiments through which we seek to answer the following question: *When is it beneficial to offload random memory accesses or computation to a GPU?*

2.3 Experimental Setup

We perform our measurements on three CPUs and three GPUs, representing the low, mid, and high end of the recent CPU and GPU markets. Table 1 shows their relevant hardware specifications and cost. All prices are from `http://www.newegg.com` as of 9/2014. The K20 connects to an AMD Opteron 6272 socket via PCIe 2.0 x16, the GTX 980 to a Xeon E5-2680 via PCIe 2.0 x16, and the GTX 650 to an i7-4770 via PCIe 3.0 x16.

2.4 Latency of CPU-GPU communication

We first measure the minimum time required to involve a GPU in a computation—the minimum extra latency that a GPU in a software router will add to every packet. In this experiment, the host transfers an input array with N 32-bit integers to the GPU, the GPU performs negligible computations on the array, and generates an output array with the same size. To provide a fair basis for comparison with CPUs, we explored the space of possible methods for this CPU-GPU data exchange in search of the best, and present results from two methods here:

Asynchronous CUDA functions: This method performs memory copies and kernel launch using asynchronous functions (e.g., `cudaMemcpyAsync`) provided by the CUDA API. Unlike synchronous CUDA functions, these functions can reduce the total processing time by overlapping data-copying with kernel execution. Figure 2 shows the timing breakdown for the different functions. We define the time taken for an asynchronous CUDA function call as the time it takes to return control to the calling CPU thread. The extra time taken to complete all the pending asynchronous functions is shown separately.

Polling on mapped memory: To avoid the overhead of CUDA functions, we tried using CUDA's mapped memory feature that allows the GPU to access the host's memory over PCIe. We perform CPU-GPU communication using mapped memory as follows. The CPU creates the input array and a flag in the host memory and raises the flag when the input is ready. CUDA threads continuously poll the flag and read the input array when they notice a raised flag. After processing, they update the output array and start polling for the flag to be raised again. This method does not use any CUDA functions in the critical path, but all accesses to mapped memory (reading the flag, reading the input array, and writing to the output array) that come from CUDA threads lead to PCIe transactions.

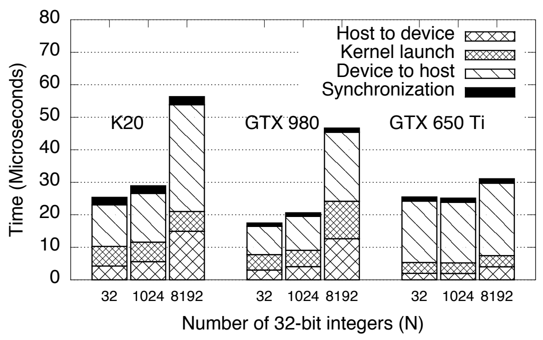


Figure 2: Timing breakdown of CPU-GPU communication with asynchronous CUDA functions.

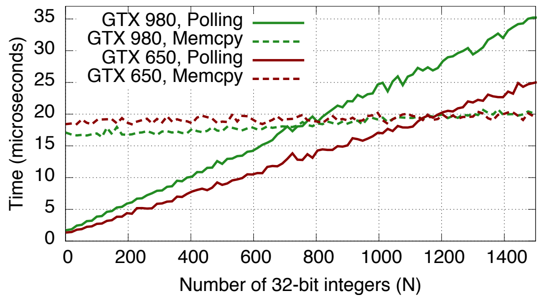


Figure 3: Minimum time for GPU-involvement with kernel-polling on mapped memory.

Figure 3 shows the time taken for this process with different values of N. The solid lines show the results with polling on mapped memory, and the dotted lines use the asynchronous CUDA functions. For small values of N, avoiding the CUDA driver overhead significantly reduces total time. However, polling generates a linearly increasing number of PCIe transactions as N increases, and becomes slower than CUDA functions for $N \sim 1000$. As GPU-offloading generally requires larger batch sizes to be efficient, we only use asynchronous CUDA functions in the rest of this work.

2.5 GPU random memory access speed

Although GPUs have much higher *sequential* memory bandwidth than CPUs (Table 1), they lose a significant fraction of their advantage when memory accesses are random, as in data structure lookups in many packet processing applications. We quantify this loss by measuring

Name	# of cores	Memory b/w	Arch., Lithography	Released	Cost	Random Access Rate
Xeon E5-2680	8	51.2 GB/s	SandyBridge, 32nm	2012	$1,748	595 M/s
Xeon E5-2650 v2	8	59.7 GB/s	IvyBridge, 22nm	2013	$1,169	464 M/s
i7-4770	4	25.6 GB/s	Haswell, 22nm	2013	$309	262 M/s
Tesla K20	2,496	208 GB/s	Kepler, 28nm	2012	$2,848	792 M/s
GTX 980	2048	224 GB/s	Maxwell, 28nm	2014	$560	1260 M/s
GTX 650 Ti	768	86.4 GB/s	Kepler, 28nm	2012	$130	597 M/s

Table 1: CPU and GPU specifications, and *measured* random access rate

the random access rate of CPUs and GPU as follows. We create a 1 GB array `L` containing a random permutation of $\{0, \ldots, 2^{28} - 1\}$, and an array `H` of B random offsets into `L`, and pre-copy them to the GPU's memory. In the experiment, each element of `H` is used to follow a chain of random locations in `L` by executing `H[i] = L[H[i]]` D times. For maximum memory parallelism, each GPU thread handles one chain, whereas each CPU core handles all the chains simultaneously. Then, the random access rate is $\frac{B*D}{t}$, where t is the time taken to complete the above process.

Table 1 shows the rate achieved for different CPUs and GPUs with $D = 10$, and the value of B that gave the maximum rate ($B = 16$ for CPUs, $B = 2^{19}$ for GPUs).[1] Although the advertised memory bandwidth of a GTX 980 (224 GB/s) is 4.37x of a Xeon E5-2680, our measured random access rate is only 2.12x. This reduction in GPU bandwidth is explained by the inability of its memory controller to coalesce memory accesses done by different threads in a warp. The coalescing optimization is only done when the warp's threads access contiguous memory, which rarely happens in our experiment.

2.6 When should we offload to a GPU?

Given that involving GPUs takes several microseconds, and their random memory access rate is not much higher than that of CPUs, it is intriguing to find out in which scenarios GPU-offloading is really beneficial. Here, we focus on two widely-explored tasks from prior work: random memory accesses and expensive computations. In the rest of this paper, all experiments are done on the E5-2680 machine with the GTX 980 GPU.

2.6.1 Offloading random memory accesses

Lookups in pointer-based data structures such as IPv4/IPv6 tries and state machines follow a chain of mostly random pointers in memory. To understand the benefit of offloading these memory accesses to GPUs, we perform the experiment in Section 2.5, *but include the time taken to transfer `H` to and from the GPU*. `H` represents a batch of header addresses used for lookups in packet processing. We set B (the size of the batch) to 8192—slightly higher than the number of packets arriving in 100µs on our 40 Gbps network. We use different values of D, representing the variation in the number of pointer-dereferencing operations for different data structures.

Figure 4a plots the number of headers processed per second for the GPU and different numbers of CPU cores. As D increases, the overhead of the CUDA function calls gets amortized and the GPU outperforms an increasing number of CPU cores. However for $D \leq 4$, the CPU outperforms the GPU, indicating that offloading ≤ 4 dependent memory accesses (e.g., IPv4 lookups in PacketShader [23] and GALE [50]) should be slower than using the CPU only.

2.6.2 Offloading expensive computation

Although GPUs can provide substantially more computing power than CPUs, the gap decreases significantly when we take the communication overhead into account. To compare the computational power of GPUs and CPUs for varying amounts of offloaded computation, we perform a sequence of D dependent `CityHash32` [4] operations on a each element of `H` (B is set to 8192).

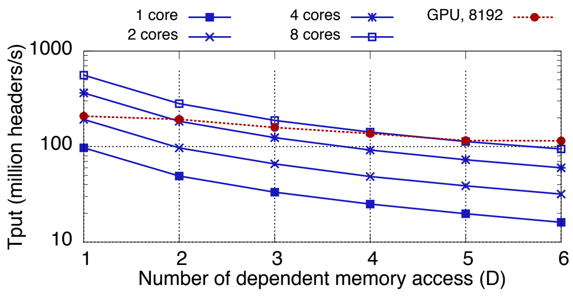
Figure 4b shows that the CPU outperforms the GPU if $D \leq 3$. Computing 3 CityHashes takes $\sim$ 40ns on one CPU core. This time frame allows for a reasonable amount of computation before it makes sense to switch to GPU offloading. For example, a CPU core can compute the cryptographically stronger Siphash [16] of a 16-byte string in $\sim$ 36ns.

3 Automatic DRAM latency hiding for CPUs

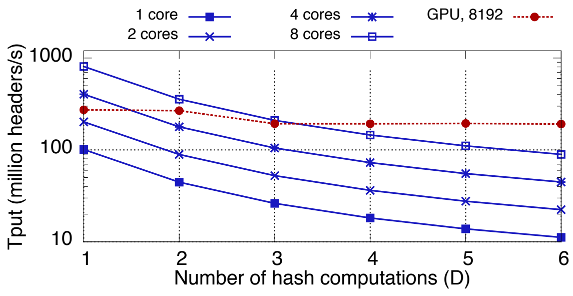
The section above showed that CPUs support respectable random memory access rates. However, *achieving* these rates is challenging: CPUs do not have hardware support for fast thread switching that enables latency hiding on GPUs. Furthermore, programs written for GPUs in CUDA or OpenCL start from the perspective of processing many (mostly)-independent packets, which facilitates latency hiding.

The simple experiment in the previous section saturated the CPU's random memory access capability because of its simplicity. Our code was structured such that each core issued B independent memory accesses—one for each chain—in a tight loop. The CPU has a limited window for reordering and issuing out-of-order instructions.

[1]The K20's rate increases to 1390 M/s if `L` is smaller than 256 MB.



(a) Offloading random memory accesses



(b) Offloading expensive computation

Figure 4: Comparison of CPU and GPU performance for commonly offloaded tasks. Note the log scale on Y axis.

When memory accesses are independent and close in the instruction stream, the CPU *can* hide the latency by issuing subsequent accesses before the first completes. However, as described below, re-structuring and optimizing real-world applications in this manner is tedious or inefficient.

A typical unoptimized packet-processing program operates by getting a batch of packets from the NIC driver, and then processing the packets one by one. Memory accesses *within* a packet are logically dependent on each other, and the memory accesses across multiple packets are spaced far apart in the instruction stream. This reduces or eliminates the memory latency-hiding effect of out-of-order execution. Our goal, then, is to (automatically) restructure this CPU code in a way that hides memory latency.

In this section, we first discuss existing techniques for optimizing CPU programs to hide their memory access latency. As these techniques are not suited to *automatically* hiding *DRAM* latency, we present a new technique called G-Opt that achieves this goal for programs with parallel data structure lookups. Although the problem of automatic parallelization and latency hiding in general is hard, certain common patterns in packet processing applications *can* be handled automatically. G-Opt hides the DRAM latency for parallel lookups that observe the same constraints as their CUDA implementations: independence across lookups and read-only data structures.

```
 1 find(entry_t *h_table, key_t *K,value_t *V) {
 2     int i;
 3     for(i = 0; i < B; i ++) {
 4         int entry_idx = hash(K[i]);
 5         // g_expensive (&h_table[entry_idx]);
 6         value_t *v_ptr = h_table[entry_idx].v_ptr;
 7         if(v_ptr != NULL) {
 8             // g_expensive (v_ptr);
 9             V[i] = *v_ptr;
10         } else {
11             V[i] = NOT_FOUND;
12         }
13     }
14 }
```

Figure 5: Naive batched hash table lookup.

3.1 Existing techniques for hiding memory access latency

3.1.1 Group prefetching

Group prefetching hides latency by processing a batch of lookups at once and by using *memory prefetches* instead of memory accesses. In a prefetch, the CPU issues a request to load a given memory location into cache, but does not wait for the request to complete. By intelligently scheduling independent instructions after a prefetch, useful work can be done while the prefetch completes. This “hiding” of prefetch latency behind independent instructions can increase performance significantly.

A data structure lookup often consists of a series of dependent memory accesses. Figure 5 shows a simple implementation of a batched hash table lookup function. Each invocation of the function processes a batch of **B** lookups. Each hash table entry contains an integer key and a pointer to a value. For simplicity, we assume for now that there are no hash collisions. There are three steps for each lookup in the batch: hash computation (line 4), accessing the hash table entry to get the value pointer (line 6), and finally accessing the value (line 9). Within a lookup, each step depends on the previous one: there are no independent instructions that can be overlapped with prefetches. However, independent instructions do exist if we consider the different lookups in the batch [17, 51].

Figure 6 is a variant of Figure 5 with the *group prefetching* optimization. It splits up the lookup code into three stages, delimited by the expensive memory accesses in the original code. We define an expensive memory access as a memory load operation that is likely to miss all levels of cache and hit DRAM. The optimized code does not directly access the hash table entry after computing the hash for a lookup key; it issues a prefetch for the entry and proceeds to compute the hash for the remaining lookups. By doing this, it does not stall on a memory lookup for the hash table entry and instead overlaps the prefetch with independent instructions (hash computation and prefetch instructions) from other lookups.

```
find(entry_t *h_table, key_t *K,value_t *V) {
    int entry_idx[B], i;
    value_t *v_ptr[B];
    // Stage 1: Hash-Computation
    for(i = 0; i < B; i ++) {
        entry_idx[i] = hash(K[i]);
        prefetch(&h_table[entry_idx[i]]);
    }

    // Stage 2: Access hash table entry
    for(i = 0; i < B; i ++) {
        v_ptr[i] = h_table[entry_idx[i]].v_ptr;
        prefetch(v_ptr[i]);
    }

    // Stage 3: Access value
    for(i = 0; i < B; i ++) {
        if(v_ptr[i] != NULL) {
            V[i] = *v_ptr[i];
        } else {
            V[i] = NOT_FOUND;
        }
    }
}
```

Figure 6: Batched lookup with group prefetching.

Unfortunately, group prefetching does not apply trivially to general lookup code because of control divergence. It requires dividing the code linearly into stages, which is difficult for code with complicated control flow. Even if such a linear layout were possible, control divergence will require a possibly large number of *masks* to record the execution paths taken by different lookups. Divergence also means that fewer lookups from a batch will enter later stages, reducing the number of instructions available to overlap with prefetches.

3.1.2 Fast context switching

In Grappa [36], fast context switching among lightweight threads is used to hide the latency of remote memory accesses over InfiniBand. After issuing a remote memory operation, the current thread yields control in an attempt to overlap the remote operation's execution with work from other threads. The minimum reported context switch time, 38 nanoseconds, is sufficiently small compared to remote memory accesses that take a few microseconds to complete. Importantly, this solution (like the hardware context switches on GPUs) is able to handle the control divergence of general packet processing. Unfortunately, the local DRAM accesses required in most packet processing applications take 60-100 nanoseconds, making the overhead of even highly optimized generic context switching unacceptable.

3.2 G-Opt

We now describe our method, called *G-Opt*, for automatically hiding DRAM latency for data structure lookup algorithms. Our technique borrows from both group prefetching and fast context switching. Individually, each of these techniques falls short of our goal: Group prefetching can hide DRAM latency but there is no general technique to automate it, and fast context switching is easy to automate but has large overhead.

G-Opt is a source-to-source transformation that operates on a batched lookup function, $\mathcal{F}$, written in C. It imposes the same constraints on the programmer that languages such as CUDA [3], OpenCL, and Intel's ISPC [8] do: the programmer must write *batch* code that expresses parallelism by granting the language explicit permission to run the code on multiple independent inputs. G-Opt additionally requires the programmer to annotate the expensive memory accesses that occur within $\mathcal{F}$. To annotate the batch lookup code in Figure 5, the lines with `g_expensive` hints should be uncommented, indicating that the following lines (line 6 and line 9) contain an expensive memory access. `g_expensive` is a macro that evaluates to an empty string: it does not affect the original code, but G-Opt uses it as a directive during code generation. The input function, $\mathcal{F}$ processes the batch of lookups one-by-one as in Figure 5. Applying G-Opt to $\mathcal{F}$ yields a new function $\mathcal{G}$ that has the same result as $\mathcal{F}$, but includes extra logic that tries to hide the latency of DRAM accesses. Before describing the transformation in more detail, we outline how the function $\mathcal{G}$ performs the lookups.

$\mathcal{G}$ begins by executing code for the first lookup. Instead of performing an expensive memory access for this lookup, $\mathcal{G}$ issues a prefetch for the access and switches to executing code for the second lookup. This continues until the second lookup encounters an expensive memory access, at which point $\mathcal{G}$ switches to the third lookup, or back to the first lookup if there are only two lookups in the batch. Upon returning to the first lookup, the new code then accesses the memory that it had previously prefetched. In the optimal case, this memory access does not need to wait on DRAM because the data is already available in the processor's L1 cache.

We now describe the transformation in more detail by discussing its action on the batched hash table lookup code in Figure 5. The code produced by G-Opt is shown in Figure 7. The key characteristics of the transformed code are:

1. **Cheap per-lookup state-maintenance**: There are two pieces of state for a lookup in $\mathcal{G}$. First, the function-specific state for a lookup is maintained in local arrays derived from the local variables in $\mathcal{F}$: the local variable named `x` in $\mathcal{F}$ is stored in `x[I]` for the `I`th lookup in $\mathcal{G}$. Second, there are two G-Opt-specific control variables for lookup `I`: `g_labels[I]` stores its `goto` target, and `g_mask`'s `I`th bit records if it has finished execution.

2. **Lookup-switching using gotos**: Instead of stalling on a memory access for lookup `I`, $\mathcal{G}$ issues a prefetch for the memory address, saves the `goto` target at the next line of code into `g_labels[I]`, and jumps to the `goto` target for the next lookup. We call this procedure a "Prefetch, Save, and Switch", or PSS. It acts as a fast switching mechanism between different lookups, and is carried out using the `G_PSS` macro that takes two arguments: the address to prefetch and the label to save as the `goto` target. G-Opt inserts a `G_PSS` macro and a `goto` target before every expensive memory access; this is achieved by using the annotations in $\mathcal{F}$.
3. **Extra initialization and termination code**: G-Opt automatically sets the initial `goto` target label for all lookups to `g_label_0`. Because different lookups can take significantly different code paths in complex applications, they can reach the label `g_end` in any order. $\mathcal{G}$ uses a bitmask to record which lookups have finished executing, and the function returns only after all lookups in the batch have reached `g_end`.

We implemented G-Opt using the ANTLR parser generator [2] framework. G-Opt performs 8 passes over the input function's Abstract Syntax Tree. It converts local variables into local arrays. It recognizes the annotations in the input function and emits labels and `G_PSS` macros. Finally, it deletes the top-level loop (written as a `foreach` loop to distinguish it from other loops) and adds the initialization and termination code based on the control variables. Our current implementation does not allow pre-processor macros in the input code, and enforces a slightly restricted subset of the ISO C grammar to avoid ambiguous cases that would normally be resolved subsequent to parsing (e.g., the original grammar can interpret `foo(x);` as a variable declaration of type `foo`).

3.3 Evaluation of G-Opt

In this section, we evaluate G-Opt on a collection of synthetic microbenchmarks that perform random memory accesses; Section 4 discusses the usefulness of G-Opt for a full-fledged software router. We present a list of our microbenchmarks along with their possible uses in real-world applications below. For each microbenchmark, we also list the source of expensive memory accesses and computation. The speedup provided by G-Opt depends on a balance between these two factors: G-Opt is not useful for compute-intensive programs with no expensive memory accesses, and loses some of its benefit for memory-intensive programs with little computation.

Cuckoo hashing: Cuckoo hashing [37] is an efficient method for storing in-memory lookup tables [19, 51]. Our 2-8 cuckoo hash table (using the terminology from MemC3 [19]) maps integer keys to integer values. *Computation*: hashing a lookup key. *Memory*: reading the corresponding entries from the hash table.

```
// Prefetch, Save label, and Switch lookup
#define G_PSS(addr, label) do {
    prefetch(addr); \
    g_labels[I] = &&label; \
    I = (I + 1) % B;  \
    goto *g_labels[I]; \
} while(0);

find(entry_t *h_table, key_t *K,value_t *V) {
    // Local variables from the function
    int entry_idx[B];
    value_t *v_ptr[B];

    // G-Opt control variables
    int I = 0, g_mask = 0;
    void *g_labels[B] = {g_label_0};

g_label_0:
    entry_idx[I] = hash(K[I]);
    G_PSS(&h_table[entry_idx[I], g_label_1);
g_label_1:
    v_ptr[I] = h_table[entry_idx[I]].v_ptr;
    if(v_ptr[I] != NULL) {
        G_PSS(v_ptr[I], g_label_2);
g_label_2:
        V[I] = *v_ptr[I];
    } else {
        V[I] = NOT_FOUND;
    }

g_end:
    g_labels[I] = &&g_end;
    g_mask = SET_BIT(g_mask, I);
    if(g_mask == (1 << B) - 1) {
        return;
    }
    I = (I + 1) % B;
    goto *g_labels[I];
}
```

Figure 7: Batched hash table lookup after G-Opt transforming.

Pointer chasing: Several algorithms that operate on pointer-based data structures, such as trees, tries, and linked lists, are based on following pointers in memory and involve little computation. We simulate a pointer-based data structure with minimal computation by using the experiment in Section 2.5. We set D to 100, emulating the long chains of dependent memory accesses performed for traversing data structures such as state machines and trees. *Computation*: negligible. *Memory*: reading an integer at a random offset in `L`.

IPv6 lookup: To demonstrate the applicability of G-Opt to real-world code, we used it to accelerate Intel DPDK's batched IPv6 lookup function. Applying G-Opt to the lookup code required only minor syntactic changes and one line of annotation, whereas hand-optimization required significant changes to the code's logic. We populated DPDK's Longest Prefix Match (LPM) structure with 200,000 random IPv6 prefixes (as done in Packet-

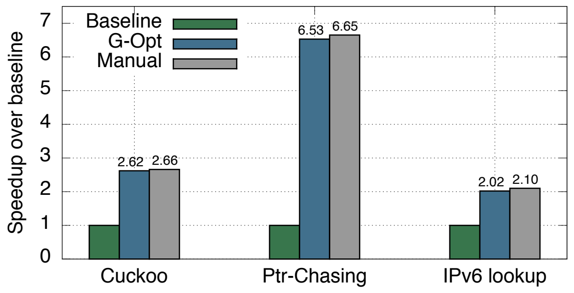


Figure 8: Speedup with G-Opt and manual group prefetching.

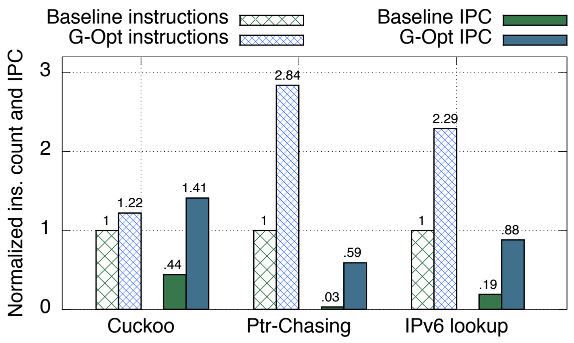


Figure 9: Instruction count and IPC with G-Opt.

Shader [23]) with lengths between 48 and 64 bits,[2] and used random samples from these prefixes to simulate a worst case lookup workload. *Computation*: a few arithmetic and bitwise operations. *Memory*: 4 to 6 accesses to the LPM data structure.

Our microbenchmarks use 2 MB hugepages to reduce TLB misses [32]. We use `gcc` version 4.6.3 with -O3. The experiments in this section were performed on a Xeon E5-2680 CPU with 32 GB of RAM and 20 MB of L3 cache. We also tested G-Opt on the CPUs in Table 1 with similar results.

3.3.1 Speedup over baseline code

Figure 8 shows the benefit of G-Opt for our microbenchmarks. G-Opt speeds up cuckoo hashing by 2.6x, pointer chasing (with $D = 100$) by 6.6x, and IPv6 lookups by 2x. The figure also shows the speedup obtained by manually re-arranging the baseline code to perform group prefetching. There is modest room for further optimization of the generated code in the future, but G-Opt performs surprisingly well compared to hand-optimized code: the manually optimized code is up to 5% faster than G-Opt. For every expensive memory access, G-Opt issues a prefetch, saves a label, and executes a `goto`, but the hand-optimized code avoids the last two steps.

3.3.2 Instruction overhead of G-Opt

G-Opt's output, $\mathcal{G}$, has more code than the original input function $\mathcal{F}$. The new function needs instructions to switch between different lookups, plus the initialization and termination code. G-Opt also replaces local variable accesses with array accesses. This can lead to additional load and store instructions because array locations are not register allocated.

Although G-Opt's code executes more instructions than the baseline code, *it uses fewer cycles* by reducing the number of cycles that are spent stalled on DRAM accesses. We quantify this effect in Figure 9 by measuring the total number of instructions and the instructions-per-cycle (IPC) for the baseline and with G-Opt. We use the PAPI tool [9] to access hardware counters for total retired instructions and total cycles. G-Opt offsets the increase in instruction count by an even larger increase in the IPC, leading to an overall decrease in execution time.

[2] This prefix length distribution is close to worst case; only 1.5% and 0.1% of real-world IPv6 prefixes are longer than 48 and 64 bits, respectively [14].

4 Evaluation

We evaluate four packet processing applications on CPUs and GPUs, each representing a different balance of computation, memory accesses, and overall processing required. We describe each application and list its computational and memory access requirements below. Although the CPU cycles used for packet header manipulation and transmission are an important source of computation, they are common to all evaluated applications and we therefore omit them from the per-application bullets. As described in Section 4.2, G-Opt also overlaps these computations with memory accesses.

Echo: To understand the limits of our hardware, we use a toy application called *Echo*. An Echo router forwards a packet to a uniformly random port P based on a random integer X in the packet's payload ($P = X \ mod\ 4$). In the GPU-offloaded version, we use the GPU to compute P from X. As this application does not involve expensive memory accesses, we do not use G-Opt on it.

IPv4 forwarding: We use Intel DPDK's implementation of the DIR-24-8-BASIC algorithm [22] for IPv4 lookups. It creates a 32 MB table for prefixes with length up to 24 bits and allocates 128 MB for longer prefixes. We populate the forwarding table with 527,961 prefixes from a BGP table snapshot [14], and use randomly generated IPv4 addresses in the workload. *Computation*: negligible. *Memory*: ~ 1 memory access on average (only 1% of our prefixes are longer than 24 bits).

IPv6 forwarding: As described in Section 3.3.

Layer-2 switch: We use the CuckooSwitch design [51]. It uses a cuckoo hash table to map MAC addresses to output ports. *Computation*: 1.5 hash-computations (on average) for determining the candidate buckets for a destination MAC address; comparing the destination MAC address with the addresses in the buckets' slots. *Memory*: 1.5 memory accesses (on average) for reading the buckets.

Named Data Networking: We use the hash-based algorithm for name lookup from Wang et al. [46], but use cuckoo hashing instead of their more complex perfect hashing scheme. We populate our name lookup table with prefixes from a URL dataset containing 10 million URLs [45, 46]. We make two simplifications for our GPU-accelerated NDN forwarding. First, because our hash function (`CityHash64`) is slow on the GPU, we use a *null kernel* that does not perform NDN lookups and returns a response immediately. Second, we use fixed-size 32-byte URLs (the average URL size used in Zhang et al. [49]) in the packet headers for both CPU and GPU, generated by randomly extending or truncating the URLs from the dataset.[3]

4.1 Experimental Setup

We conduct full-system experiments on a Xeon E5-2680 CPU (8 cores @2.70 GHz)-based server.[4] The CPU socket has 32 GB of quad-channel DDR3-1600 DRAM in its NUMA domain, 2 dual-port Intel X520 10 GbE NICs connected via PCIe 2.0 x8, and a GTX 980 connected via PCIe 2.0 x16. To generate the workload for the server, we use two client machines equipped with Intel L5640 CPUs (6 cores @2.27 GHz) and one Intel X520 NIC. The two 10 GbE ports on these machines are connected directly to two ports on the server. The machines run Ubuntu with Linux kernel 3.11.2 with Intel DPDK 1.5 and CUDA 6.0.

4.2 System design

Network I/O: We use Intel's DPDK [5] to access the NICs from userspace. We create as many RX and TX queues on the NIC ports as the number of active CPU cores, and ensure exclusive access to queues. Although the 40 Gbps of network bandwidth on the server machine corresponds to a maximum packet rate of 59.52 (14.88 * 4) million packets per second (Mpps) for minimum sized Ethernet frames, only 47.2 Mpps is achievable; the PCIe 2.0 x8 interface to the dual-port NIC is the bottleneck for minimum sized packets [51]. As the maximum gains from GPU acceleration come for small packets [23], we use the smallest possible packet size in all experiments.

GPU acceleration: We use PacketShader's approach to GPU-based packet processing as follows. We run a dedicated master thread that communicates with the GPU, and several worker threads that receive and transmit packets from the network. Using a single thread to communicate with the GPU is necessary because the overhead of CUDA functions increases drastically when called from multiple threads or processes. The worker threads extract the essential information from the packets and pass it on to the master thread using exclusive worker-master

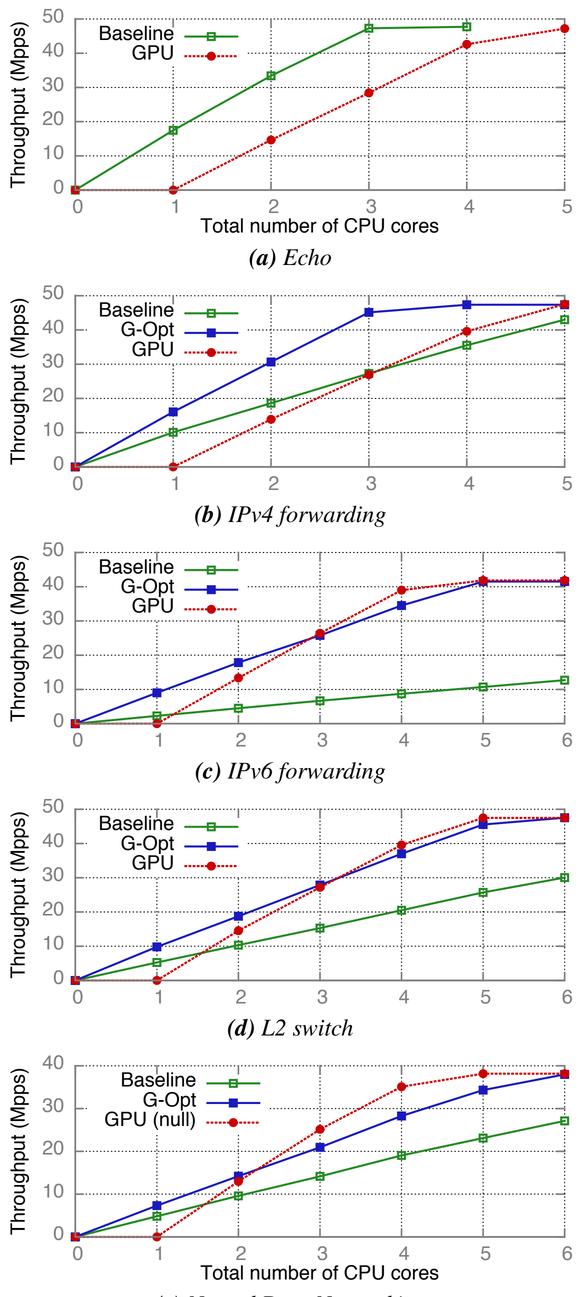


Figure 10: Full system throughput with increasing *total* CPU cores, N. For the GPU, N includes the master core (throughput is zero when $N = 1$ as there are no worker cores). The x-axis label is the same for all graphs.

queues. The workers also perform standard packet processing tasks like sanity checks and setting header fields. This division of labor between workers and master reduces the amount of data that the master needs to transmit to the GPU. For example, in IPv4 forwarding, the master receives only one 4-byte IPv4 address per received packet. In our implementation, each worker can have up to 4096 outstanding packets to the master.

[3]Our CPU version does not need to make these assumptions, and performs similarly with variable length URLs.

[4]The server is dual socket, but we restricted experiments to a single CPU to avoid noise from cross-socket QPI traffic. Previous work on software packet processing suggests that performance will scale and our results will apply to two socket systems [32, 51, 23].

PacketShader's master thread issues a separate CUDA `memcpy` for the data generated by each worker to transfer it to the GPU directly via DMA without first copying to the master's cache. Because of the large overhead of CUDA function calls (Figure 2), we chose not to use this approach.

Using G-Opt for packet processing programs: Intel DPDK provides functions to receive and transmit batches of packets. Using batching reduces function call and PCIe transaction overheads [23, 51] and is required for achieving the peak throughput. Our baseline code works as follows. First, it calls the batched receive function to get a batch of up to 16 packets from a NIC queue. It then passes this batch to the packet processing function $\mathcal{F}$, which processes the packets one by one.

We then apply G-Opt on $\mathcal{F}$ to generate the optimized function $\mathcal{G}$. Unlike the simpler benchmarks in Section 3.3, $\mathcal{F}$ is a full-fledged packet handler: it includes code for header manipulation and packet transmission in addition to the core data structure lookup. This gives $\mathcal{G}$ freedom to overlap the prefetches from the lookups with this additional code, but also gives it permission to transmit packets in a different order than they were received. However, $\mathcal{G}$ preserves the per-flow ordering if forwarding decisions are made based on packet headers only, as in all the applications above.[5] If so, all packets from the same flow are "switched out" by $\mathcal{G}$ at the same program points, ensuring that they reach the transmission code in order.

4.3 Workload generation

The performance of the above-mentioned packet processing applications depends significantly on two workload characteristics. The following discussion focuses on IPv4 forwarding, but similar factors exist for the other applications. First, the distribution of prefixes in the server's forwarding table, and the IP addresses in the workload packets generated by the clients, affects the cache hit rate in the server. Second, in real-world traffic, packets with the same IP address (e.g., from the same TCP connection) arrive in bursts, increasing the cache hit rate.

Although these considerations are important, recall that our primary focus is understanding the relative advantages of GPU acceleration as presented in previous work. We therefore tried to mimic PacketShader's experiments that measure the near worst-case performance of both CPUs and GPUs. Thus, for IPv4 forwarding, we used a real-world forwarding table and generated the IPv4 addresses in the packets with a uniform random distribution. For IPv6 forwarding, we populated the forwarding table with prefixes with randomly generated content, and chose the workload's addresses from these prefixes using uniformly random sampling.[6] We speculate that prior work may have favored these conditions because worst-case performance is an important factor in router design for quality of service and denial-of-service resilience. Based on results from previous studies [31, 48], we also expect that more cache-friendly (non-random) workloads are likely to improve CPU performance more than that of GPUs.

4.4 Throughput comparison

Figure 10 shows the throughput of CPU-only and GPU-accelerated software routers with different numbers of CPU cores. For Echo (Figure 10a), the CPU achieves $\sim$ 17.5 Mpps of single-core throughput and needs 3 cores to saturate the 2 dual-port 10 GbE NICs. The GPU-offloaded implementation needs at least 4 worker cores, for a total of 5 CPU cores including the master thread. This happens because the overhead of communicating each request with the master reduces the single-worker throughput to 14.6 Mpps.

Figure 10b shows the graphs for IPv4 lookup. Without G-Opt, using a GPU provides some benefit: With a budget of 4 CPU cores, the GPU-accelerated version outperforms the baseline by 12.5%. After optimizing with G-Opt, the CPU version is strictly better than the GPU-accelerated version. G-Opt achieves the platform's peak throughput with 4 CPU cores, whereas the GPU-accelerated version requires 5 CPU cores *and* a GPU.

With G-Opt, a single core can process 16 million IPv4 packets per second, which is 59% higher than the baseline's single-core performance and is only 8.9% less than the 17.5 Mpps for Echos. When using the DIR-24-8-BASIC algorithm for IPv4 lookups, the CPU needs to perform only $\sim$ 1 expensive memory access in addition to the work done in Echo. With G-Opt, the latency of this memory access for a packet is hidden behind independent packet-handling instructions from other packets. As GPUs also hide memory access latency, the GPU-accelerated version of IPv4 forwarding performs similarly to its Echo counterpart.

For IPv6 forwarding (Figure 10c), G-Opt increases single-core throughput by 3.8x from 2.2 Mpps to 8.4 Mpps. Interestingly, this increase is *larger* than G-Opt's 2x gain in local IPv6 lookup performance (Figure 8). This counter-intuitive observation is explained by the reduction in effectiveness of the reorder buffer for the baseline code: Due to additional packet handling instructions, the independent memory accesses for different packets in a batch are spaced farther apart in the forwarding code than in the local benchmarking code. These instructions consume slots in the processor's reorder buffer, reducing its ability to detect the inter-packet independence.

[5]For applications that also examine the packet content, the transmission code can be moved outside $\mathcal{F}$ for a small performance penalty.

[6]This workload is the worst case for DPDK's trie-based IPv6 lookup. PacketShader's IPv6 lookup algorithm uses hashing and shows worst-case behavior for IPv6 addresses with random content.

With G-Opt, our CPU-only implementation achieves 39 Mpps with 5 cores, and the platform's peak IPv6 throughput (42 Mpps) with 6 cores. Because IPv6 lookups require relatively heavyweight processing, our GPU-based implementation indeed provides higher *per-worker* throughput—it delivers line rate with only 4 worker cores, *but it requires another core for the master in addition to the GPU*. Therefore, using a GPU plus 5 CPU cores can provide a 7.7% throughput increase over using just 5 CPUs, but is equivalent to using 6 CPUs.

For the L2 switch (Figure 10d), G-Opt increases the throughput of the baseline by 86%, delivering 9.8 Mpps of single-core throughput. This is significantly smaller than the 17.5 Mpps for Echos because of the expensive hash computation required by cuckoo hashing. Our CPU-only implementation saturates the NICs with 6 cores, and achieves 96% of the peak throughput with 5 cores. In comparison, our GPU-accelerated L2 switch requires 5 CPU cores and a GPU for peak throughput.

For Named Data Networking, G-Opt increases single-core throughput from 4.8 Mpps to 7.3 Mpps, a 1.5x increase. With a budget of 4 CPU cores, the (simplified) GPU version's performance is 24% higher than G-Opt, but is almost identical if G-Opt is given one additional CPU core.

Conclusion: For all our applications, the throughput gain from adding a GPU is never larger than from adding just one CPU core. The cost of a Xeon E5-2680 v3 [6] core (more powerful than the cores used in this paper) is \$150. In comparison, the cheapest GPU used in this paper costs \$130 and consumes 110W of extra power. CPUs are therefore a more attractive and resource efficient choice than GPUs for these applications.

4.5 Latency comparison

The GPU-accelerated versions of the above applications not only require more resources than their G-Opt counterparts, but also add significant latency. Each round of communication with the GPU on our server takes $\sim$ 20µs (Figure 2). As the packets that arrive during a round must wait for the next round to begin, the average latency added is $20 * 1.5 = 30$µs.

Our latency experiments measure the round-trip latency at clients. Ideally, we would have liked to measure the latency *added* by the server without including the latency added by the client's NIC and network stack. This requires the use of hardware-based traffic generators [42] to which we did not have access.[7]

In our experiments, clients add a timestamp to packets during transmission and use it to measure the RTT after reception. We control the load offered by clients by tuning the amount of time they sleep between packet transmissions. The large sleep time required for generating a low load, and buffered transmission at the server [32] cause our measured latency to be higher than our system's minimum RTT of 16µs.

For brevity, we present our latency-vs-throughput graphs only for Echo, and IPv4 and IPv6 forwarding. The CPU-only versions use G-Opt. All measurements used the minimum number of CPU cores required for saturating the network bandwidth.

Figure 11a shows that the RTT of CPU-only Echo is 29µs at peak throughput and 19.5µs at low load. The minimum RTT with GPU acceleration is 52µs, which is close to 30µs larger than the CPU-only version's minimum RTT. We observe similar numbers for IPv4 and IPv6 forwarding (Figures 11b and 11c), but the GPU version's latency increases at high load because of the larger batch sizes required for efficient memory latency hiding.

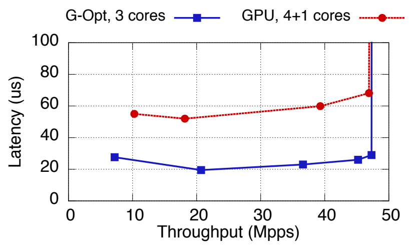
5 Discussion

Other similar optimizations for CPU programs Until now, we have discussed the benefit of an automatic DRAM latency-hiding optimization, G-Opt. We now discuss how intrusion detection systems (IDSes), *an application whose working set fits in cache* [41], can benefit from similar, latency-hiding optimizations.

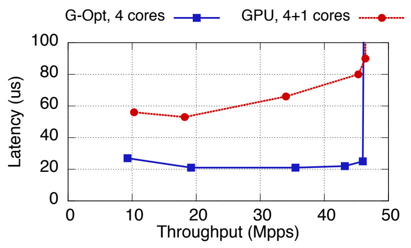
We study the packet filtering stage of Snort [39], a popular IDS. In this stage, each packet's payload is used to traverse one of several Aho-Corasick [15] DFAs. The DFA represents the set of malicious patterns against which this packet should be matched; Snort chooses which DFA to use based on the packet header. For our experiments, we recorded the patterns inserted by Snort v2.9.7 into its DFAs and used them to populate our simplified pattern matching engine. Our experiment uses 23,331 patterns inserted into 450 DFAs, leading to 301,857 DFA states. The workload is a `tcpdump` file from the DARPA Intrusion Detection Data Sets [11].

Our baseline implementation of packet filtering passes batches of B ($\sim$ 8) packets to a function that returns B lists of matched patterns. This function processes packets one-by-one. We made two optimizations to this function. First, we perform a loop interchange: Instead of completing one traversal before beginning another, we interweave them to give the CPU more independent instructions to reorder, reducing stalls from long-latency loads from cache. Second, we collect a larger batch of packets (8192 in our implementation), and sort it—first by the packet's DFA number and then by length. Sorting by DFA number reduces cache misses during batch traversal. Sorting by length increases the effectiveness of loop interchange—similar to minimizing control flow divergence for GPU-based traversals [41].

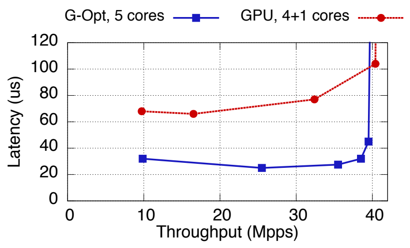
[7]Experiments with a Spirent SPT-N11U [42] traffic generator as the client have measured a minimum RTT of 7-8µs on an E5-2697 v2 server; the minimum RTT measured by our clients is 16µs.



(a) Echo



(b) IPv4 forwarding



(c) IPv6 forwarding

Figure 11: Full system latency with minimum CPU cores required to saturate network bandwidth.

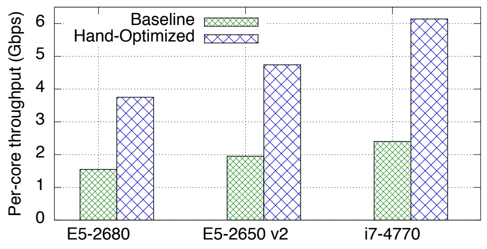


Figure 12: Pattern matching gains (no network I/O)

Figure 12 shows that, for a local experiment without network I/O, these optimizations increase single-core matching throughput by 2.4x or more. We believe that our optimizations also apply to the CPU-only versions of pattern matching in GPU-accelerated IDSes including Kargus [24] and Snap [43]. As we have only implemented the packet filtering stage (Snort uses a second, similar stage to discard false positives), we do not claim that CPUs can outperform GPUs for a full IDS. However, they can reduce the GPU advantage, or make CPU-only versions more cost effective. For example, in an experiment with innocent traffic, Kargus's throughput (with network I/O) improved between 1.4x and 2.4x with GPU offloading. Our pattern matching improvements offer similar gains which should persist in this experiment: innocent traffic rarely triggers the second stage, and network I/O requires less than 10% of the CPU cycles spent in pattern matching.

Additional applications We have shown that CPU implementations can be competitive with (or outperform) GPUs for a wide range of applications, including lightweight (IPv4 forwarding, Layer-2 switching), midweight (IPv6 and NDN forwarding), and heavyweight (intrusion detection) applications. Previous work explores the applicability of GPU acceleration to a number of different applications; one particularly important class, however, is cryptographic applications.

Cryptographic applications, on the one hand, involve large amounts of computation, making them seem attractive for vector processing [25, 23]. On the other hand, encryption and hashing requires copying the full packet data to the GPU (not just headers, for example). Since the publication of PacketShader, the first work in this area, Intel has implemented hardware AES encryption support for their CPUs. We therefore suspect that the 3.5x speedup observed in PacketShader for IPSec encryption would be unlikely to hold on today's CPUs. And, indeed, 6WIND's AES-NI-based IPSec implementation delivers 6 Gbps per core [1], 8x higher than PacketShader's CPU-only IPSec, though on different hardware.

One cryptographic workload where GPUs still have an advantage is processing expensive, but infrequent, RSA operations as done in SSLShader, assuming that connections arrive closely enough together for their RSA setup to be batched.[8] Being compute intensive, these cryptographic applications raise a second question for future work: Can automatic vectorization approaches (e.g., Intel's ISPC [8]) be used to increase the efficiency of CPU-based cryptographic applications?

Revising TCO estimates In light of the speedups we have shown possible for some CPU-based packet processing applications, it bears revisiting total-cost-of-ownership calculations for such machines. The TCO of a machine includes not just the cost of the CPUs, but the motherboard and chipset as well as the total system power draw, and the physical space occupied by the machine.

Although our measurements did not include power, several conclusions are obvious: Because the GPU-accelerated versions required almost as many CPU cores as the CPU-only versions, they are likely to use at least modestly more power than the CPU versions. The GTX 980 in our experiments can draw up to 165W compared to 130W for the E5-2680's 8 cores, though we lack precise power draw measurements.

Adding GPUs requires additional PCIe slots and lanes from the CPU, in addition to the cost of the GPUs. This burden is likely small for applications that require transferring only the packet header to the GPU, such as forwarding—but those applications are also a poor match for the GPU. It can, however, be significant for high-bandwidth offload applications, such as encryption and deep packet inspection.

[8]And perhaps HMAC-SHA1, but Intel's next generation "Skylake" CPUs will have hardware support for SHA-1 and SHA-256.

Future GPU trends may improve the picture. Several capabilities are on the horizon: CPU-integrated GPU functions may substantially reduce the cost of data and control transfers to the GPU. Newer NVidia GPUs support "GPUDirect" [7], which allows both the CPU and certain NICs to DMA directly packets to the GPU. GPUDirect could thus allow complete CPU-bypass from NIC to GPU, or reduce CUDA's overhead by letting the CPU write directly to GPU memory [29]. This technology currently has several restrictions—the software is nascent, and only expensive Tesla GPUs (over $1,700 each) and RDMA-capable NICs are supported. A more fundamental and long-term limitation of removing CPU involvement from packet processing is that it requires entire packets, not just headers, to be transferred to the GPU. The CPU's PCIe lanes would then have to be divided almost equally between NICs and GPUs, possibly halving the network bandwidth that the system can handle.

Alternative architectures such as Tilera's manycore designs, which place over 100 cores on a single chip with high I/O and memory bandwidth, or Intel's Xeon Phi, are interesting and under-explored possibilities. Although our results say nothing about the relative efficiency of these architectures, we hope that our techniques will enable better comparisons between them and traditional CPUs.

Handling updates Currently, G-Opt works only for data structures that are not updated concurrently. This constraint also applies to GPU-accelerated routers where the CPU constructs the data structure and ships it to the GPU. It is possible to hide DRAM latency for updates using manual group prefetching [32]; if updates are relatively infrequent, they also can be handled outside the batch lookup code. Incorporating updates into G-Opt is part of future work.

6 Related Work

GPU-based packet processing Several systems have used GPUs for IPv4 lookups *absent* network I/O [50, 31, 30, 35], demonstrating substantial speedups. Our end-to-end measurements that include network I/O, however, show that there is very little room for improving IPv4 lookup performance—when IPv4 forwarding is optimized with G-Opt, the single-core throughput drops by less than 9% relative to Echo. Packet classification requires matching packet headers against a corpus of rules; the large amount of per-packet processing makes it promising for GPU acceleration [23, 27, 44]. GSwitch [44] is a recent GPU-accelerated packet classification system. We believe that the Bloom filter and hash table lookups in GSwitch's CPU version can benefit from G-Opt's latency hiding, reducing the GPU's advantage.

CPU-based packet processing RouteBricks [18] focused on mechanisms to allocate packets to cores; its techniques are now standard for making effective use of a multicore CPU for network packet handling. User-level networking frameworks like Intel's DPDK [5], `netmap` [38], and PF_RING [10] provide a modern and efficient software basis for packet forwarding, which our work and others takes advantage of. Many of the insights in this paper were motivated by our prior work on hiding lookup latency in CuckooSwitch [51], an L2 switch that achieves 80 Gbps while storing a billion MAC addresses.

Hiding DRAM latency for CPU programs is important in many contexts: Group prefetching and software pipelining has been used to mask DRAM latency for database hash-joins [17], a software-based L2 switch [51], in-memory trees [40, 28], and in-memory key-value stores [32, 34, 26]. These systems required manual code rewrites. To our knowledge, G-Opt is the first method to automatically hide DRAM latency for the independent lookups in these applications.

7 Conclusion

Our work challenges the conclusions of prior studies about the relative performance advantages of GPUs in packet processing. GPUs achieve their parallelism and performance benefits by constraining the code that programmers can write, but this very coding paradigm also allows for latency-hiding CPU implementations. Our G-Opt tool provides a semi-automated way to produce such implementations. CPU-only implementations of IPv4, IPv6, NDN, and Layer-2 forwarding can thereby be more resource efficient and add lower latency than GPU implementations. We hope that enabling researchers and developers to more easily optimize their CPU-based designs will help improve future evaluation of both hardware and software-based approaches for packet processing. Although we have examined a wide range of applications, this work is not the end of the line. Numerous other applications have been proposed for GPU-based acceleration, and we believe that these techniques may be applicable to other domains that involve read-mostly, parallelizable processing of small requests.

Code release The code for G-Opt and the experiments in this paper is available at `https://github.com/efficient/gopt`.

Acknowledgements This work was supported by funding from the National Science Foundation under awards CNS-1314721, CCF-0964474, and 1345305; and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). Emulab [47] and PRObE [21] were used for some experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). We thank Sangjin Han and Hyeontaek Lim for valuable comments, and David Maltz for shepherding.

References

[1] High-Performance Packet Processing Solutions for Intel Architecture Platforms. http://www.lannerinc.com/downloads/campaigns/LIDS/05-LIDS-Presentation-Charlie-Ashton-6WIND.pdf.

[2] ANTLR. http://www.antlr.org.

[3] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[4] CityHash. https://code.google.com/p/cityhash/.

[5] Intel DPDK: Data Plane Development Kit. http://dpdk.org.

[6] Intel Xeon Processor E5-2680 v3. http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz.

[7] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[8] Intel's SPMD Program Compiler. https://ispc.github.io.

[9] Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[10] PF_RING: High-speed packet capture, filtering and analysis. http://www.ntop.org/products/pf_ring/.

[11] DARPA Intrusion Detection Data Sets, . http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/.

[12] NetFPGA, . http://yuba.stanford.edu/NetFPGA/.

[13] Open vSwitch, . http://www.openvswitch.org.

[14] University of Oregon Route Views Project, . http://www.routeviews.org.

[15] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, June 1975.

[16] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. In *INDOCRYPT*, 2012.

[17] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst. 2007*.

[18] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP 2009*.

[19] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.

[20] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM 2014*.

[21] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[22] P. Gupta, S. Lin, and M. Nick. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM 1998*.

[23] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *SIGCOMM 2010*.

[24] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: A Highly-scalable Software-based Intrusion Detection System. In *CCS 2012*.

[25] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI 2011*.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*, 2014.

[27] K. Kang and Y. S. Deng. Scalable Packet Classification via GPU Metaprogramming. In *DATE*, 2011.

[28] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Designing Fast Architecture-sensitive Tree Search on Modern Multicore/Many-core Processors. *ACM TODS 2011*, .

[29] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *OSDI 2014*, .

[30] T. Li, H. Chu, and P. Wang. IP Address Lookup Using GPU. In *HPSR*, 2013.

[31] Y. Li, D. Zhang, A. X. Liu, and J. Zheng. GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers. In *ANCS 2013*.

[32] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX NSDI*, 2014.

[33] D. Lustig and M. Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *HPCA 2013*.

[34] Y. Mao, C. Cutler, and R. Morris. Optimizing RAM-latency Dominated Applications. In *APSys 2013*.

[35] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. IP Routing Processing with Graphic Processors. In *DATE*, 2010.

[36] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications. Technical Report UW-CSE-14-02-01, University of Washington.

[37] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, May 2004.

[38] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC 2012*.

[39] M. Roesch and S. Telecommunications. Snort - Lightweight Intrusion Detection for Networks. 1999.

[40] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB 2011*.

[41] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *ISPASS*, 2009.

[42] Spirent. Spirent SPT-N11U. http://www.spirent.com/sitecore/content/Home/Ethernet_Testing/

Platforms/11U_Chassis.

[43] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *ANCS 2013*.

[44] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-Layer Packet Classification with Graphics Processing Units. In *CoNEXT*, 2014.

[45] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire Speed Name Lookup: A GPU-based Approach. In *NSDI 2013*.

[46] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for Named Data Networking. In *IWQoS*, 2014.

[47] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

[48] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP Lookup Performance with FIB Explosion. In *SIGCOMM*, 2014.

[49] T. Zhang, Y. Wang, T. Yang, J. Lu, and B. Liu. NDNBench: A benchmark for Named Data Networking lookup. In *GLOBECOM*, 2013.

[50] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu. Exploiting Graphics Processors for High-performance IP Lookup in Software Routers. *INFOCOM*, 2011.

[51] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*, 2013.