A Linux Kernel Implementation of the Homa Transport Protocol

John Ousterhout, *Stanford University*

https://www.usenix.org/conference/atc21/presentation/ousterhout

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

A Linux Kernel Implementation of the Homa Transport Protocol

John Ousterhout
Stanford University

Abstract

Homa/Linux is a Linux kernel module that implements the Homa transport protocol. Measurements of Homa/Linux reconfirm Homa's superior performance compared to TCP and DCTCP. In a cluster benchmark with 40 nodes, Homa/Linux provided lower latency than both TCP and DCTCP for all message sizes; for short messages, Homa's 99th percentile tail latency was 7–83x lower than TCP and DCTCP. The benchmarks also show that Homa has eliminated network congestion as a significant performance limitation. Both tail latency and throughput are now limited by software overheads, particularly software congestion caused by imperfect load balancing of the protocol stack across cores. Another factor of 5–10x in performance can be achieved if software overheads can be eliminated in the future.

1 Introduction

Montazeri et al. recently introduced a new network transport protocol for datacenters called Homa [25]. Homa uses network priority queues and receiver-driven packet scheduling to favor shorter messages and eliminate congestion at host downlinks. Homa reduces latency significantly, especially for short messages under high network loads. Montazeri et al. demonstrated tail latencies almost 100x better than the best prior measurements of transport protocols that included TCP, DCTCP [1], Infiniband, HULL [2], PIAS [4], and NDP [15].

However, the original evaluation of Homa was done in a research setting, based partly on simulations and partly on a user-level implementation (with kernel bypass) in the RAMCloud storage system [29]. The implementation did not support general applications, and it was difficult to compare it fairly to a kernel TCP implementation, since the user-level implementation of Homa avoided the high software overheads imposed on TCP by the kernel. As a result, the original Homa work left open questions about whether the protocol's benefits could be achieved in a more practical setting.

This paper describes Homa/Linux, a Linux kernel module that implements Homa. I undertook the development of Homa/Linux with three goals: first, to understand how the overheads of a kernel implementation affect Homa's implementation and performance; second, to measure how a kernel version of Homa performs compared to a mature and widely used implementation of TCP and DCTCP; and third, to create a practical implementation of Homa to encourage its use in real applications and evaluate its benefits for production datacenter workloads.

This paper makes two contributions. First, Homa/Linux demonstrates that it is possible to build a competitive production implementation of Homa. Homa/Linux outperforms both TCP and DCTCP by a wide margin, confirming the results in [25]. Using four of the Montazeri workloads and considering short messages under high network loads, P99 (99th percentile) latency under Homa/Linux is 7–83x lower than TCP and DCTCP. All message sizes under all workloads experience lower latency with Homa than either TCP or DCTCP, both at the median at P99. In most cases Homa's P99 latency is lower than the median for TCP and DCTCP. Homa needs only a few priority levels to achieve high performance, and it outperforms TCP and DCTCP even with only one priority level.

Second, this work provides a case study of the challenges in building a performant transport protocol in the high-overhead environment of the Linux kernel. Homa/Linux had to address a variety of issues, such as batching, load balancing, and real-time processing. Homa eliminates almost all network congestion, but software congestion is becoming more problematic as more and more cores must be harnessed to keep up with increasing network speeds. In Homa/Linux, software overheads are now the primary factor limiting performance. This paper quantifies those overheads. For example, at least 18 cores will be required to drive a 100 Gbps network in both directions, and distributing protocol processing across multiple cores increases software overheads by 2–3x.

Although Homa/Linux provides much better performance than TCP, its latency and small-message throughput are still 5–10x worse than raw network speeds. Section 7 argues that this gap cannot be reduced significantly as long as transports are implemented in software. To harness future networks' full performance potential, protocols such as Homa will probably need to be implemented in hardware. This will require new NIC architectures to be developed.

2 Homa Summary

This section summarizes the key elements of the Homa protocol; see Montazeri et al. [25] for details and discussion. Homa is designed as a transport for RPC frameworks in datacenters. It is optimized for networks with one-way hardware latencies as low as 1–2 µs and aims to provide the lowest possible tail latency for short messages, even when operating at high network load with a mix of message lengths. It does so by minimizing the latency impact of congestion at the network edge (host downlinks). Homa also eliminates head-of-line block-

ing that occurs when small messages are delayed behind large ones in TCP streams. Homa does not explicitly deal with congestion in the network core (see Section 8).

SRPT and messages. Homa implements an approximation of SRPT (shortest remaining processing time), prioritizing shorter messages over longer ones. SRPT is most beneficial for short messages, but it also improves latency for large messages compared to the fair sharing approach used in protocols such as TCP. This is because SRPT produces run-to-completion behavior: once a message becomes highest priority, it will remain highest priority until it completes (unless new messages arrive).

Receiver-driven packet scheduling. In Homa, each receiver collects information about all of its incoming messages and schedules incoming packet transmissions. The first few packets of each message (*unscheduled packets*) are transmitted unilaterally by the sender, but the remaining *scheduled packets* are sent only in response to *grants* from the receiver. The number of unscheduled packets is typically chosen to cover the round-trip time, so that an unloaded server can return the first grant before the last unscheduled packet has been sent. This allows messages to use the full network bandwidth in an unloaded system. The grant mechanism allows receivers to limit congestion at their downlinks; buffer buildup occurs only if many senders transmit unscheduled packets simultaneously to the same destination (*incast*). I use the term *RTTbytes* to describe the number of unscheduled bytes, following Montazeri et al.; its role is analogous to windows in other protocols.

In-network priority queues. Homa takes advantage of the priority queues in modern network switches (typically 8 or 16 for each egress port). It divides the priority levels into two groups: the highest levels are used for unscheduled packets, and the lower levels for scheduled ones. Within each group, shorter messages get higher priorities. Receivers choose the priorities for all of their incoming traffic. For scheduled packets, the receiver specifies priorities "just in time" using grants. For unscheduled packets, the receiver specifies in advance the range of message lengths for each priority level; it disseminates new assignments occasionally as its workload changes.

The fraction of priority levels allocated for unscheduled packets is chosen to match the fraction of all incoming bytes that are in unscheduled packets. The cutoffs between unscheduled priorities are assigned so that each priority level is used for about the same number of incoming bytes.

Sender-side SRPT. Homa nodes also implement SRPT when transmitting: given multiple messages with packets to transmit, a sender should transmit packets for the message with the fewest remaining bytes. To do this, Homa must limit the rate at which outgoing packets are passed to the NIC to ensure that long queues do not build up in the NIC (these would delay packets from new shorter messages).

Overcommitment. When a receiver issues a grant there is no guarantee that the sender will immediately transmit the granted packets. In order to keep its downlink fully loaded, receivers thus grant simultaneously to multiple incoming messages; this is called *overcommitment* since it overcommits the downlink and may result in buffering at the switch. Scheduled priority levels are allocated to ensure SRPT among the messages being granted, so overcommitment does not affect the latency of the highest priority message. The degree of overcommitment is typically in the range of 5–10, which is chosen as a balance between excessive buffer buildup (if too large) and inefficient use of downlink bandwidth (if too small).

3 Homa/Linux API

TCP's connection-oriented socket API is a poor match for Homa, and for datacenter applications in general. The first problem is TCP's streaming nature, which means that it has no notion of message boundaries. This is problematic for Homa, which needs message lengths to implement SRPT. Streams are also awkward for most datacenter applications, which communicate via messages. These applications must impose their own message structure on TCP streams; this adds nontrivial complexity, given that a read might return only part of a message, or parts of several messages. It is difficult to share a TCP stream between multiple reading threads (e.g., in a server), since reads don't necessarily return dispatchable units (entire messages).

Streams also have the disadvantage of enforcing FIFO ordering on their messages. As a result, long messages can severely delay short ones that follow them. This head-of-line blocking is one of the primary sources of tail latency measured for TCP in Section 5.

A second problem with TCP sockets is that they are connection oriented, with long-lived state for each peer that an application communicates with. Connections are undesirable in datacenters because applications can have hundreds or thousands of them, resulting in high space and time overheads. Some applications have resorted to proxies or other forms of connection pooling [32] to reduce the overheads. It seems to be an article of faith in the networking community that connections are necessary for desirable properties such as reliable delivery and congestion control, but in fact all of these properties can be achieved without connections.

Because of these issues, Homa/Linux provides a message-based API. Specifically, it implements *remote procedure calls* (RPCs). Each RPC consists of a *request* message sent from a client to a server, followed by a corresponding *response* message in return. An RPC protocol has two advantages over a pure messaging approach. First, the response serves as acknowledgment for the request, reducing the number of packets that must be processed. Second, it results in a deeper interface [28] because the transport can implement the timers and retries needed to ensure end-to-end completion. In a pure messaging protocol, timeouts must be implemented by applications (the protocol can ensure that individual messages are delivered, but cannot ensure that servers generate responses).

Homa/Linux has no notion of connections, only RPCs. A client can use a single socket for simultaneous communication

```
int socket(AF_INET, SOCK_DGRAM, IPPROTO_HOMA);
int bind(int sockfd, const struct sockaddr *addr,
        socklen_t addrlen);
int close(int sockfd);

int homa_send(int sockfd, const void *request,
              size_t reqlen,
              const struct sockaddr *dest_addr,
              socklen_t addrlen, uint64_t *id);
int homa_recv(int sockfd, void *buf, size_t len,
              int flags, struct sockaddr *src_addr,
              socklen_t addrlen, uint64_t *id);
int homa_reply(int sockfd, const void *response,
               size_t resplen,
               const struct sockaddr *dest_addr,
               socklen_t addrlen, uint64_t id);
```

Figure 1: The API provided by Homa for applications; `socket`, `bind`, and `close` are existing Linux system calls.

with any number of servers. A server can use a single socket to receive requests from any number of clients, to reply to client requests, and to issue its own requests as a client. Homa/Linux maintains state only for active RPCs (rarely more than a few at a time). This eliminates the overheads associated with connections.

Each Homa RPC is independent. Any number of RPCs may be active simultaneously for a given socket. If a client issues multiple concurrent RPCs, they may complete in any order; this eliminates the TCP's head-of-line blocking problem. If order matters among concurrent RPCs, Homa clients can add their own sequence numbers to messages.

Figure 1 shows Homa/Linux's system call interface. The existing `socket` and `close` calls are used to create and delete Homa sockets; Homa/Linux defines a new protocol type `IPPROTO_HOMA`. If a socket is going to receive incoming requests, the existing `bind` call is used to associate the socket with a well-known port number. There is no need to invoke `bind` for client sockets; Homa/Linux automatically assigns port numbers for them from the upper half of the 16-bit port space.

Homa/Linux defines three new functions: `homa_send`, `homa_recv`, and `homa_reply`. All are implemented via `ioctl`s on the Homa socket; Homa/Linux does not add new system calls to Linux. A client invokes `homa_send` to issue a request; it returns a 64-bit unique identifier for the RPC, which can be used to wait for the corresponding response. `homa_recv` receives incoming messages; it returns a message and its unique identifier. The arguments to `homa_recv` can restrict it to return only requests, only responses, or only a response with a given identifier. `homa_reply` is used by servers to send responses; it is similar to `homa_send` except that the RPC identifier is an input argument.

The Homa/Linux API is not backwards-compatible with existing applications based on TCP or UDP sockets, due to the API's need for explicit RPC ids. However, most datacenter applications are layered on a few RPC frameworks such as gRPC [12] or Apache Thrift [34]; adding Homa support to them should enable transparent usage by many applications.

Homa/Linux ensures reliable delivery of messages and eventual completion of each RPC issued by a client. An RPC fails only if the server becomes nonresponsive or there is no socket matching the port in the request. There is no limit on the processing time for an RPC.

4 Implementation

Homa/Linux is a dynamically loadable Linux kernel module; it does not require any changes to Linux itself. The current version runs on Linux version 5.4.80 and contains about 10,000 lines of heavily commented C code. Source for Homa/Linux is freely available on GitHub along with unit tests, instrumentation tools, and all of the benchmarks discussed in this paper [16]. The module was implemented with the goal of achieving production quality, and I believe the current version is mature enough to run a variety of applications.

Most of the implementation effort for Homa/Linux focused on three obstacles to high performance. The first is the high cost of pushing a packet through the protocol stack. This cost is particularly high in Linux, due to its many layers and features. To amortize the protocol stack overheads, packets must be collected into batches, so that the stack traversal cost is only incurred once per batch. Unfortunately, batching necessarily incurs latency, since the first packet in a batch must be delayed until the last is available. Larger batches can be processed more efficiently, but incur a larger latency penalty.

The second obstacle is that a single core is too slow to handle all the protocol processing for a modern high-speed network, especially for small packets. This problem is worsening over time, since network speeds are increasing rapidly while CPU speeds are not. A transport protocol must be able to balance its load across a large number of cores. Even so, small-packet performance will be severely limited by software overheads; for example, neither Homa nor TCP can utilize more than about one-third of the bandwidth of a 25 Gbps network for short-message workloads, even with 20 cores. The greatest challenge with load balancing is software congestion in the form of hot spots that form when too much work is assigned to one core; even after considerable optimization, hot spots remain the single greatest source of tail latency for Homa/Linux.

The third obstacle to high performance is real-time processing. In the case of Homa, real-time processing is needed to implement a transmit rate limiter as discussed in Section 4.3. Implementing such a mechanism, which requires response times on the order of a few microseconds, is challenging in the Linux kernel and creates additional software overheads.

These problems are likely to occur in any transport protocol, and Linux contains mechanisms to facilitate both batching and load-balancing. However, the Linux mechanisms are biased towards TCP, even when implemented in code that is ostensibly protocol independent. To implement Homa efficiently, it was necessary to reshape Homa/Linux to fit the existing mechanisms. In some cases, Homa/Linux subverts the mechanisms to achieve results that are not possible with TCP.

One recurring issue is that TCP depends on in-order processing of a stream's packets. This requires flow-consistent

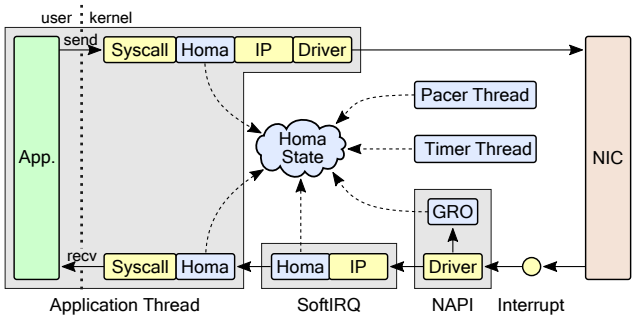

Figure 2: Structure of Homa/Linux. Homa components are shown in blue; existing Linux kernel modules are in yellow. Gray areas represent different cores. Only the primary sending and receiving paths are shown; other Homa elements such as the pacer thread and timer thread also transmit packets.

routing in the network, and also requires consistency in protocol processing on the hosts: for example, to preserve packet order, each packet of a flow must pass through the same set of cores. Homa does not require packets to be processed in order; by relaxing this constraint it can achieve better batching and load balancing. As a result, Homa/Linux attempts to bypass TCP's constraints; this is not always easy or clean, since the constraints tend to be enforced by Linux.

The rest of the section discusses these three primary issues in detail, plus a few other smaller issues. The design described here evolved over about two years and was driven by extensive performance measurements. For example, throughput for Homa/Linux improved by about 3x over this process (the initial version did not implement batching). Space limitations prevent inclusion of the detailed measurements.

4.1 Packet flow and batching

Figure 2 shows the major components of Homa and the basic packet flow for sending and receiving messages. Homa exists as a layer just above IP in the Linux networking stack, parallel to TCP and UDP; the packet flow discussed in this section is similar to that for TCP and UDP. Homa packets are encapsulated as IPv4 datagrams.

The basic path for packet transmission is shown in the top part of Figure 2. When an application invokes `homa_send` or `homa_reply`, the resulting `ioctl` kernel call is dispatched to Homa/Linux. Homa/Linux copies the message data from user space into packet buffers and passes the first few packets to the IP stack, which eventually calls a device driver to transmit the packets.

Linux offers two mechanisms for batching on the transmit path: TSO (TCP Segmentation Offload) and GSO (Generic Segmentation Offload). In either case, the transport protocol creates a single large packet (up to 64 KB) for stack traversal. Once the packet reaches the device driver it is broken up into smaller segments for transmission. With TSO the segmentation occurs in the NIC. For NICs that don't support TSO, or for protocols that cannot use TSO, GSO can perform segmentation in software, just above the driver layer. Homa/Linux currently supports TSO but not GSO. TSO assumes that packets have TCP headers, so Homa packet headers mimic the TCP fields that TSO depends on. Unfortunately, TSO and GSO enable batching only for data packets that are part of the same message. For short messages and control packets, each transmitted packet must traverse the protocol stack independently.

The receive path is shown at the bottom of Figure 2. When an interrupt occurs, the interrupt handler doesn't actually read packets; instead, it schedules an *NAPI* action to execute driver code. NAPI actions execute at interrupt level on the same core that received the interrupt, just before returning from the interrupt and with interrupts reenabled. To avoid high overheads associated with interrupts, the NAPI layer polls the NIC for more packets until it reaches either a packet count limit or a time limit.

Once the NAPI layer has finished collecting packets, it passes them to the *SoftIRQ* layer, where they work their way through the networking and IP layers, eventually reaching Homa. Homa performs protocol processing and message reassembly. When a message is complete, Homa signals the waiting application thread (or queues the message if there are no waiting threads); the application thread copies the message data to user space and then returns from the `homa_recv` system call. SoftIRQ code executes in the same fashion as NAPI (at interrupt level with interrupts enabled) but it may run on a different core than the NAPI code (see Section 4.2).

Before forwarding incoming packets to SoftIRQ, the NAPI layer organizes them into batches. It does this by passing each packet to a transport-specific function using a mechanism called GRO (Generic Receive Offload). The transport-specific GRO function decides how to group packets into batches. For TCP, each batch contains packets from a single flow (this ensures that all packets from a flow go to the same core for SoftIRQ). Unfortunately, protocol-independent GRO code partially segregates batches by flow before invoking the transport-specific function. This segregation is neither necessary nor desirable for Homa. Fortunately, I found a way for Homa/Linux to defeat the GRO segregation and collect all incoming packets into a single batch (see the Appendix for details). This allows Homa to batch more aggressively than TCP, and is one of the reasons Homa has higher throughput for short messages (in TCP, each short message becomes a separate batch).

4.2 Load balancing

As mentioned in the introduction to this section, the networking subsystem must distribute load across multiple cores to keep up with a high-speed network. Load balancing is easy for packet output, because the output stack executes entirely on the sending thread's core, with a separate NIC channel per core. The Linux scheduler balances threads across cores, and this distributes the packet transmission load as well.

The remainder of this section focuses on packet input, where Linux employs two different load balancers. The first is RSS, which runs in NICs and distributes incoming packets across cores for NAPI processing using a hash of packet

header fields. This hash function ensures that all packets from a given flow are assigned to the same core, as required by TCP. The second load balancer runs in the NAPI layer: after incoming packets have been grouped into batches, the NAPI layer chooses a SoftIRQ core for each batch. It does this using another hash function on header fields of the first packet in each batch; the hash is different from the one used by the NIC, but it ensures that all batches for a given flow are delegated to the same SoftIRQ core.

Input load balancing is problematic because it results in hot spots. Hot spots occur because the Linux scheduler's load balancer is unaware of the networking subsystem. For example, the NAPI layer can consume a core for hundreds of microseconds when a burst of packets arrives for a single long message. Short messages can still be processed through the NAPI and SoftIRQ layers on other cores, but if a message's target thread is assigned to the core processing the local burst, it will not run until after the burst is handled. This form of hot spot is the primary contributor to P99 tail latency in Homa/Linux.

Homa exacerbates this form of hot spot because it tries to use the entire incoming link bandwidth for one message at a time. TCP's fair-sharing scheduler tends to divide incoming bandwidth among multiple flows, which will probably be processed on different cores, so this form of hot spot is less likely to occur. Increasing packet size does not help, because NAPI polls for more packets: if the MTU increases then NAPI will spend less time processing packets but more time polling.

Homa/Linux has been unable to eliminate NAPI-thread conflicts because it has no control over the RSS hash function in the NICs. Hot spots can also occur with the SoftIRQ layer, where it conflicts with either application threads or NAPI. I eliminated most of these hot spots by repurposing an alternate mechanism for SoftIRQ core selection (the socket flow table, which steers packets to a specific application thread's core) so that Homa's GRO function can select a specific SoftIRQ core for each packet batch. After experimenting with several policies, I settled on one that records for each core the most recent time when it processed a Homa packet at either NAPI or SoftIRQ level. To choose a SoftIRQ core, NAPI consults the times for the next few cores after the one where it is running (in circular order) and selects the one with the oldest time. This policy has two desirable properties. First, it avoids conflicts between NAPI and SoftIRQ. Second, it tends to distribute SoftIRQ batches across multiple cores, so that no single core is occupied continuously: this leaves time for application threads to execute on each core. This policy improved Homa's P99 latency by about a factor of 2x.

I experimented with using core affinity to reduce conflicts between application threads and NAPI/SoftIRQ threads. For example, I tried partitioning the cores, with one set used exclusively for application threads and the others for NAPI and SoftIRQ processing. However, I was unable to find a configuration where core affinity improved tail latency.

A second problem with load balancing is that it hurts performance at low load. At low load it is best to concentrate all processing on a single core: ideally, RSS should be disabled so that all interrupts pass through a single core, and that same core should execute both the NAPI and SoftIRQ layers. This maximizes cache locality, eliminates inter-core synchronization, eliminates the overhead of cross-core invocation between the NAPI and SoftIRQ layers, and makes the NAPI polling mechanism more efficient. In contrast, load balancing employs many cores but at low load they are all underutilized. These cores are likely to enter power-saving *C-states*, resulting in wakeup delays of 50–100 µs the next time work is assigned to them. Furthermore, the continual arrival of packets on all cores reduces the effectiveness of power-saving mechanisms.

Ideally, the load balancing configuration should change dynamically with load, but there does not appear to be a way to do this in Linux. Thus, Homa/Linux is optimized for high loads. As a result, P99 latency for short messages is actually worse at low load than high load (see Section 5.4). However, Homa/Linux includes one optimization for low load. The algorithm for choosing a SoftIRQ core checks to see if the current batch contains only a single packet (an indicator of low load). If so, the NAPI core is also used for SoftIRQ. This reduces round-trip latency by 3–4 µs when the system is underloaded.

4.3 Real-time processing: the pacer

As discussed in Section 2, Homa/Linux must limit the length of the NIC's internal transmit queue to enforce SRPT during output: if a large queue builds up in the NIC, then it will delay small messages. When the NIC queue length exceeds a threshold, Homa/Linux no longer transmits packets immediately; their messages are added to a queue and a *pacer* thread is awakened. The pacer thread manages the queue, passing packets to the NIC in SRPT order as NIC queue length permits.

The NIC queue length is not directly observable, so Homa/Linux maintains `nic_empty_time`, an estimate of the time when all queued packets will have been transmitted. As each transmitted packet is passed to the IP stack, this variable is updated based on the packet length and link speed. A system parameter controls how far into the future `nic_empty_time` is allowed to get; the current value is 2 µs.

The pacer operates under severe real-time constraints: to keep the NIC queue short while fully utilizing the uplink, it must queue new packets at a time granularity of 1–2 µs. There is no sleep/wakeup mechanism in Linux that can operate at this timescale, so the pacer thread must poll the processor cycle counter to wait for the next transmission time. As a result, the pacer consumes most of a core under heavy output load.

Even with a dedicated core, the pacer can fall behind. In the original design, once the pacer queue became nonempty, all output packets passed through the pacer. Unfortunately, a single thread cannot push small packets through the IP stack fast enough to drive the network at full speed. In addition, the scheduler occasionally deschedules the pacer; this can produce gaps of several milliseconds with no packets transmitted.

Homa/Linux includes three additional mechanisms to ensure full usage of the uplink. First, packets smaller than a threshold (currently 1000 bytes) bypass the pacer mechanism: they are passed to the NIC immediately without consulting or updating `nic_empty_time` (short packets transmit so quickly that it isn't possible to queue them faster than the NIC sends them, so the pacer is superfluous). Second, other cores help push packets through the network stack when the pacer falls behind. Before queuing a message for the pacer, Homa/Linux checks `nic_empty_time`; if it has dropped below the threshold for sending more packets, it indicates that the pacer isn't keeping the uplink fully utilized, so packets are transmitted directly, even if the pacer queue is occupied. Third, the pacer is invoked explicitly by the SoftIRQ layer after processing each batch of Homa packets; if the pacer thread has fallen behind, this invocation will queue more packets (but it returns without polling).

4.4 Grants

The goal of the grant mechanism is to maintain RTTbytes of outstanding grants (data granted but not yet received) for each *active message*. The number of active messages is limited by the degree of overcommitment; only the highest priority messages (according to SRPT) are active. In addition, Homa/Linux will not grant to more than one message from a given peer at a time, since the peer will only transmit the shortest one. To implement this, Homa/Linux maintains a global 2-level priority queue of incoming messages, consisting of a list of messages from each peer, plus a list of all peers with non-empty lists. Each is sorted in increasing order of the number of bytes not yet granted; the active messages consist of the first message from each of the first few peer lists.

Sending grants is not triggered by a clock, but rather by packet arrivals: after the SoftIRQ layer processes each batch of incoming packets, it checks to see if any active messages need additional grants. Packet arrivals may also change the structure of the grant queues (a new message may appear or an existing message may become fully granted).

The original design of Homa called for one grant packet to be sent for each scheduled data packet. However, this approach results in high overheads. Furthermore, the use of TSO means that packets are transmitted in groups, so there is no benefit in sending grants at per-packet granularity. Thus, Homa/Linux provides a parameter that specifies a minimum increment for grants; it is currently set to 10000 bytes, which is the same as the size of Homa's TSO packets.

4.5 Other implementation issues

Locking. The Linux networking stack is designed around per-socket locks. This approach works well for TCP because each flow is associated with its own socket; thus different flows can be processed concurrently without lock conflicts. However, a Homa application typically only has one socket, which is used for all messages. Homa initially used socket-level locking, but this resulted in severe contention for socket locks.

Homa was eventually refactored to use RPC-level locks as the primary synchronization mechanism. However, per-RPC locks created additional complications. For example, when the first packet arrives for an RPC, an entry must be created in a hash table of RPCs associated with the socket; future packets must use the existing RPC structure. However, multiple "first" packets for an RPC might be processed concurrently on different cores; the natural way to synchronize them is to use the socket's lock. To process packets without acquiring the socket lock, Homa/Linux associates a lock with each bucket in the socket's hash table; this lock covers all RPCs in that bucket. Bucket locks synchronize the lookup of an existing RPC with the creation of a new one, while allowing RPCs in other buckets to be processed concurrently.

Another issue with locking is the potential for deadlock. There are several places in Homa/Linux where multiple locks must be held. RPC locks are normally acquired before any others, but there are a few places where an RPC lock needs to be acquired while holding a different lock; this risks deadlock. Code had to be reorganized on a case-by-case basis to prevent deadlock. For example, in some situations the RPC lock can be acquired conditionally; if it is not available, then the operation can be deferred until later.

Combining FIFO with SRPT. Homa's SRPT priority mechanism is ideal for reducing small-message tail latency, and it also works well for most large messages because of its run-to-completion nature. However, under high load, a few of the largest messages may suffer very high tail latency. To mitigate this problem, Homa/Linux directs a small configurable fraction of the total network bandwidth to the *oldest* message instead of using SRPT; this occurs both when issuing grants for incoming messages and in the pacer for outgoing messages. The current fraction is 5%, which eliminates most of the penalty for the largest messages without affecting tail latency for short messages. This improves tail latency for the largest messages by about 2x in pathological cases.

Reaping RPCs. Any long-running operation creates a threat to tail latency because it may delay other work. One example is RPC reaping, which frees the resources for an RPC after it completes. For clients, this is just before returning from `homa_recv`; for servers, it is after the response has been sent. Reaping can take tens of microseconds for large RPCs, most of which is spent freeing packet buffers. Reaping was originally done immediately when RPCs were freed, but it had a noticeable impact on tail latency. On servers, for example, reaping could occur in the SoftIRQ layer while handling an incoming packet, causing unpredictable delays for subsequent packets.

To reduce the impact of reaping on tail latency, Homa/Linux now defers reaping so that it does not occur when an RPC is freed. Instead, reaping occurs in `homa_recv` while waiting for incoming messages: large messages are reaped incrementally in chunks of a few packet buffers, checking for incoming messages after each chunk. This approach

hides the cost of reaping unless the system is so loaded that homa_recv never waits. If the pool of unreaped messages grows too large, then homa_recv will stop and reap despite the waiting messages; this situation is rare in practice.

Receive polling. If a thread blocks in homa_recv, it takes about 2.5 µs to wake it on message arrival; this contributes significantly to latency. To avoid this cost, Homa polls briefly in homa_recv before blocking the thread. This saves about 2 µs if a message arrives during the polling interval. The interval is a system parameter, currently 50 µs. Polling is useful primarily when the system is lightly loaded; it has little impact on the tail latency measurements in Section 5.

Timer thread. Homa contains a dedicated in-kernel thread that wakes up at 1 ms intervals, checks for overdue packets, requests retransmissions, and eventually declares a peer dead if it fails to respond. Lost packets are rare, so slow response is usually due to overload on the peer; to minimize the extra load from retransmission requests, the timer thread will only request retransmission for a single RPC at a time for each peer (the oldest one). If a peer is declared dead, then all RPCs to/from that peer are aborted.

Computing unscheduled priorities. Each Homa host must send information to its peers about the priorities to use for unscheduled packets. Priorities are computed by a user-level daemon, homa_prio, which runs regularly (currently every 500 ms) and collects information from Homa/Linux about the size distribution for recently received messages. It then decides how many priorities to use for unscheduled packets and computes cutoff values (the largest message size for each priority level) as described in Section 2. If the cutoffs have changed significantly, homa_prio passes them to Homa/Linux using the sysctl mechanism. Homa/Linux does not immediately notify all its peers; it waits until the next message from each peer, and then sends the new cutoffs.

Long-term state. As mentioned previously, Homa stores no long-term connection information. It does, however, keep long-term state for each peer that it has communicated with. A peer's state is about 200 bytes, of which almost half consists of cached routing information. The remainder includes information about unscheduled priorities for that peer and information for detecting timeouts and managing grants. This is far less than the 2000 bytes that TCP stores for each open socket. Peer state is created on demand and never discarded.

Configuration parameters. One disadvantage of Homa/Linux is that it has about 30 configuration parameters. Many were added solely for evaluating the implementation and need not be considered in practice. For many others the exact value has little impact on system performance. About 5–10 parameters can have a significant impact on performance; for them the best value depends on hardware characteristics such as network speed, CPU speed, and number of cores. I believe that it is possible to compute their values automatically by running benchmarks, but I leave this to future work. The only configuration parameter for which the best value might vary from application to application is the polling interval; this should probably be made an argument of the homa_recv kernel call.

CPU:	E5-2640v4 (10 cores, 2.4 GHz)
RAM:	4x 16 GB DDR4-2400 DIMMs
Disk:	Intel DC S3520 SSD (480 GB 6G SATA)
NIC:	Mellanox ConnectX–4 25 Gbps
Switch:	Mellanox 2410

Table 1: CloudLab [10] xl170 hardware configuration used for benchmarking. All nodes ran Linux 5.4.80. Hyperthreads were enabled (2 hyperthreads per core). TSO and RSS were enabled for all protocols, with separate transmit/receive channels for each hyperthread. C-States were enabled, Meltdown mitigations were disabled, and interrupt moderation was disabled. All 40 nodes were connected to a single switch.

	Homa	TCP	DCTCP
100B latency (µs)	15.1	23.4	24.1
500KB throughput (Gbps)	10.0	20.3	20.5
Client throughput (Gbps)	23.8	23.9	21.4
Server throughput (Gbps)	23.7	23.6	22.4
Client RPC rate (Mops/sec)	1.6	1.0	1.0
Server RPC rate (Mops/sec)	1.6	1.0	1.0

Table 2: Basic Homa and TCP performance. The top two lines used a single client thread issuing back-to-back requests to a single server. Latency was measured end-to-end at application level with 100-byte requests and responses; throughput was measured with 500 KB requests and responses. For the remaining measurements each client had multiple threads; each thread issued multiple concurrent RPCs. Client performance was measured with a single client node spreading requests across 9 server nodes; server performance was measured with 9 client nodes all issuing requests to a single server node. Throughput was measured with 500 KB requests and responses and counts only message payloads; RPC rate was measured with 100-byte requests and responses. Each number represents the best average across five 5-second runs.

5 Performance Evaluation

This section evaluates Homa/Linux's performance, comparing it with the Linux implementations of TCP and DCTCP. The most important metric is tail latency for short messages under high load, but the benchmarks also measure performance for large messages and lower loads. This section also evaluates concerns that have been raised in recent papers about Homa's use of priorities and network buffers.

The benchmarks ran on a 40-node cluster described in Table 1. Each server had 10 cores with two-way hyperthreading enabled. Unless otherwise indicated below, the term "core" refers to a single hyperthread. Unless otherwise stated, all measurements used a maximum packet size (MTU) of 3000 bytes, which produced better results for all protocols than the traditional 1500 bytes. For DCTCP the ECN marking threshold was 70 KB.

5.1 Basic latency and throughput

Table 2 shows latency and throughput for Homa/Linux and TCP under best-case conditions. Homa/Linux round-trip-times for short messages are about one-third lower than those of TCP or DCTCP (15.1 µs vs. 23.4/24.1 µs). This difference is primarily due to Homa's use of polling (4 µs) and its optimization of SoftIRQ core selection (3–4 µs). In addition, Homa's single-socket-per-thread approach eliminated

Name	Mean	Description
W2	433	Search application at Google [33].
W3	2423	Aggregated workload from all applications running in a Google datacenter [33].
W4	60175	Hadoop cluster at Facebook [32].
W5	385315	Web search workload used for DCTCP [1].

Table 3: The message size distributions used for benchmarks. The workloads were taken from [25], except that messsages larger than 1 MB in W4 and W5 were truncated to 1 MB (the largest size permitted under Homa/Linux). The workloads are ordered by average message size ("Mean"): W2 is most skewed towards small messages and W5 is most heavy-tailed.

the need for `epoll` system calls, which saves 1 µs. Section 6 shows that latencies in real applications are considerably higher than this for both Homa/Linux and TCP. For comparison, the lowest achievable round-trip latency for this hardware (using kernel bypass) is 3.7 µs [19].

Homa's throughput is only about half that of TCP or DCTCP when a single client sends large RPCs back-to-back to an unloaded server (10 Gbps vs. 20 Gbps). This is because Homa's message-based interface reduces opportunities for pipelining. In particular, Homa/Linux cannot transfer any part of a message to a waiting application until the entire message has arrived, whereas TCP can overlap network transmission and copying to user space. When multiple RPCs are active simultaneously, Homa's throughput is equivalent to TCP's, while DCTCP's throughput is slightly lower.

Table 2 also shows the maximum small message request rate. Homa's throughput is 60% above TCP or DCTCP. One reason is Homa's ability to batch received packets more effectively than TCP, as described in Section 4.1. Homa generated average batches of 2–3 packets at the NAPI level, whereas TCP did not batch packets at all for this benchmark.

5.2 Cluster benchmarks

Most of the remaining evaluation is based on a cluster benchmark that uses all 40 nodes. Each node operates simultaneously as both client and server, with multiple client and server threads chosen for each protocol to maximize its performance. Each node has multiple independent Homa server sockets or TCP listen sockets. Clients send request messages to servers chosen at random, and servers return response messages of the same size as the requests. Message sizes are chosen at random to match one of four workload distributions, W2–W5, described in Table 3. These distributions are the same as those used in Montazeri et al. except that W1 is omitted because of space limitations (W1's behavior is almost identical to W2). The timing of new requests is based on a Poisson arrival function that produces a particular average throughput.

Figure 3 compares Homa with TCP and DCTCP for each of the four workloads under high network loads. The figure uses *slowdown* as a measure of latency, which allows comparisons between messages with different lengths. The slowdown for a given RPC consists of the end-to-end round-trip time (RTT) observed by the client application divided by the RTT for RPCs of the same length measured with Homa under the ideal conditions of Table 2; smaller slowdowns are better.

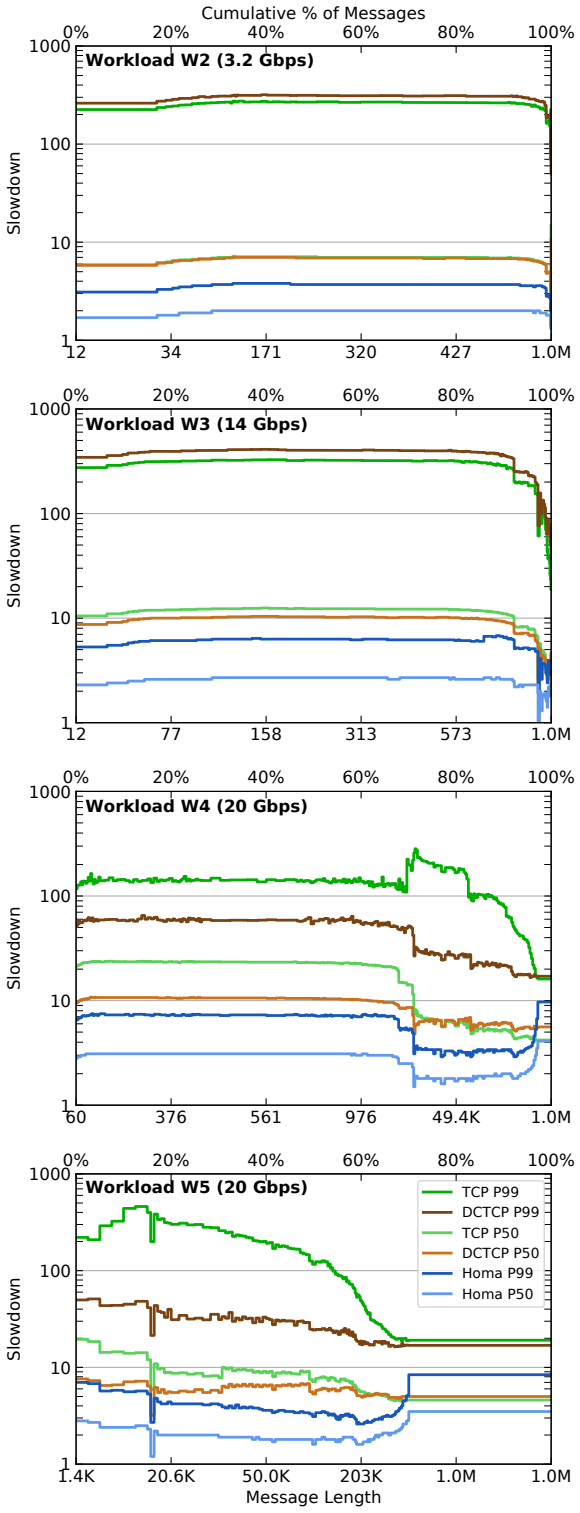

Figure 3: Median and 99th-percentile slowdown as a function of message size for workloads W2-W5. Each x-axis is linear in number of RPCs (i.e. it reflects the CDF of message length for that workload). The network load for each workload (e.g. 20 Gbps for W4 and W5) was chosen so that the protocols operated at 80–90% of their maximum sustainable rates; load is measured in units of message payload bytes sent by each host (an equal amount was also received).

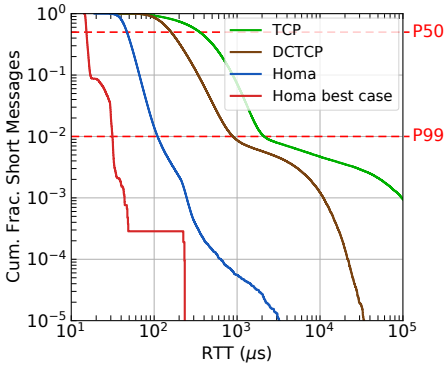

Figure 4: Complementary CDF of round-trip latency for the shortest 10% of RPCs for W4 (messages with 315 bytes or fewer) in Figure 3. A point (x, y) indicates that a fraction y of RPCs had round-trip times longer than x. "Homa best case" was measured under the conditions of Table 2.

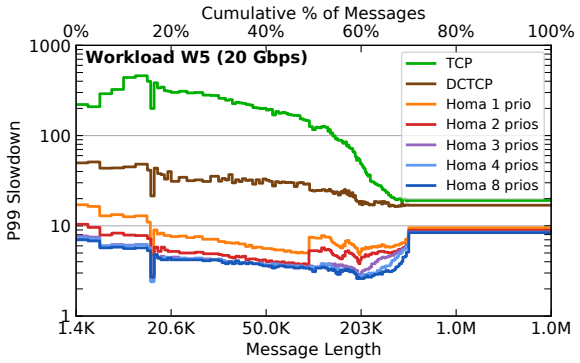

Figure 5: The impact of the number of available priority levels on Homa's P99 slowdown for workload W5 (other workloads are omitted because of space limitations; they are less sensitive to the number of priorities than W5). TCP and DCTCP are shown for comparison.

Homa's latencies are significantly lower than both TCP and DCTCP. For example, Homa's P99 latencies for short messages are 19–72x lower than those for TCP and 7–83x lower than DCTCP. Homa's P50 latencies for short messages are 3.5–7.5x lower than TCP and 2.7–3.8x lower than DCTCP. Homa's P99 slowdown is actually lower than P50 slowdown for TCP and DCTCP, except for the the largest messages in W4 and W5. DCTCP generally outperforms TCP, as expected.

W2 is different from the other workloads in that network congestion is not an issue: almost all messages are short, so none of the protocols can support more than about 40% network utilization. For this workload performance is limited primarily by software overheads; Homa still provides much lower latency than either TCP or DCTCP. An analysis of TCP's P99 behavior showed that it is caused by scheduler anomalies where two threads end up assigned to the same core, even though there are fewer application threads than cores. One of them eventually migrates to a different core, but by the time it resumes execution, many milliseconds have been lost. Anomalies like this did not occur for Homa.

One potential concern with Homa's SRPT policy is its impact on large messages. Figure 3 shows that this is not a problem in practice. Slowdowns for workloads W4 and W5 do increase for large message sizes, but Homa still beats both TCP and DTCP on P50 and P99 latency. There is no message size in any workload where TCP or DCTCP outperformed Homa.

Figure 4 shows more detail on the round-trip latency for short messages in W4. Homa's latencies are better than TCP and DCTCP at every percentile, typically by at least an order of magnitude.

Homa's slowdown in Figure 4 is entirely due to software overheads: Homa has eliminated network congestion as a significant factor. To verify this, I used a timetracing package to extract precise end-to-end traces of short RPCs with latencies near the 99th percentile. Latency for these RPCs was primarily due to hot spots in load balancing: an application thread cannot execute to handle a short message because its core is occupied by NAPI processing for unrelated large messages. These NAPI bursts can last 100 μs or more.

In contrast, timetraces for TCP RPCs near P99 showed different causes of tail latency. Roughly two-thirds of the traces had head-of-line blocking where a short message was stuck behind a long one in the same stream. In the remaining traces the extra time occurred after a packet was accepted by the source NIC and before an interrupt occurred on the destination, suggesting buffer buildup in the NIC or at the switch's egress port. This analysis shows that the P99 latency difference between TCP/DCTCP and Homa is because of protocol features, as opposed to quality of the implementations.

5.3 Number of priorities

One of the concerns that has been raised about Homa is its use of switch priorities (e.g., [21]): priority queues are a limited resource, and in some datacenters they are already allocated to applications with critical QoS needs. Figure 5 shows that Homa/Linux needs only a few levels to achieve maximum performance: it does well with 2 priorities, and additional levels benefit only a small range of message sizes. Even with only a single priority level, Homa's P99 latency is still much better than DCTCP or TCP.

5.4 Reduced load

The measurements in Section 5.2 were made under high load (80–90% of the maximum that the protocols and network can sustain). Figure 6 shows slowdown for Homa and DCTCP when the load is reduced by a factor of 2x or 10x. Because of space limitations, measurements are shown only for W4, and only for Homa and DCTCP (other workloads are similar).

Figure 6 shows that Homa is more stable than DCTCP. Homa's latency for short messages varies by less than 2x across a 10x load change, whereas DCTCP's latency varies by 7x. Homa's P99 short-message latency at the highest load is 40% lower than DCTCP's P99 latency at a 10x reduced load.

Homa displays a performance inversion in Figure 6. Once the load drops below 50% of maximum, short-message latency begins to increase; Homa latencies at the lowest load are about 25% *higher* than those at the highest load. The inversion is due to C-states. At low loads, cores may be idle for

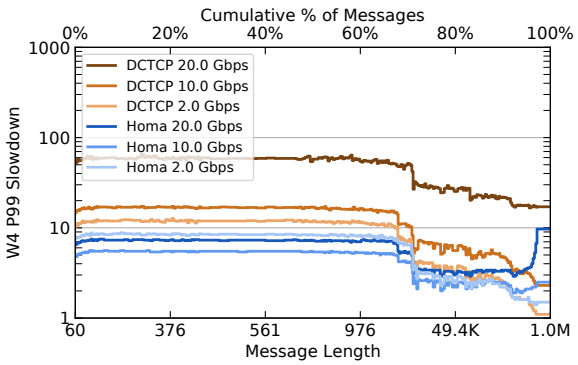

Figure 6: P99 slowdown as a function of message size for W4 under varying loads.

significant periods, causing them to enter C-states; it takes 50–100 µs to awaken them when packets arrive. I confirmed this behavior by running the benchmark with C-states disabled: in this case, Homa latency continues to improve with lower loads.

Figure 6 suggests that Homa's automatic priority allocation mechanism may outperform the manual approaches currently used in datacenters. Consider an application using DCTCP whose workload matches W4, and suppose it has exclusive use of the highest network priority. It will then behave roughly as if it had a private network, so its performance will match the DCTCP curve in Figure 6 corresponding to the load generated by that application. Now suppose that, instead of manually allocating QoS levels, all applications switch to Homa, and Homa uses all the priority levels to support them without preference. The performance will now match the Homa curve in Figure 6 corresponding to the combined load of all the applications. There is almost no overlap in these curves: shared Homa provides better latency than differentiated DCTCP for all but the largest 20% of messages. If those particular message sizes are not crucial for the application's performance, it would be better off in a shared environment under Homa, even though Homa offers no particular QoS support. The shared approach would also eliminate the administrative burden of assigning and managing QoS levels.

5.5 Buffer usage

Homa uses buffers relatively economically compared to TCP, but its performance will degrade if buffer space is exhausted and packets are dropped. Hai et al. measured Homa with switch buffers limited to 200 KB per port [14]. They found that this resulted in very poor Homa performance due to dropped packets and the subsequent timeouts and retransmissions. I analyzed buffer usage in our cluster to get a better understanding of this issue, and found that Homa fits easily in the buffer space available in our 25 Gbps environment.

Hai et al. assumed a fixed-size buffer for each port. However, many modern switches have dynamic buffer pools that can be shared between ports. For example, our switches have 16 MB of buffer space, of which 13 MB can be dynamically shared among 40 ports.

	W2	W3	W4	W5	W5-6K
`homa_send`	1.35	1.38	0.47	0.45	0.38
`homa_recv`	2.11	2.54	1.29	1.32	1.22
`homa_reply`	1.60	1.69	0.56	0.50	0.44
NAPI	1.29	1.76	1.18	1.37	1.01
SoftIRQ	1.31	1.58	0.90	0.87	0.82
Pacer	0.02	0.34	0.71	0.70	0.66
Timer	0.01	0.01	0.01	0.01	0.01
Total	**7.69**	**9.29**	**5.11**	**5.03**	**4.53**
Polling	7.30	6.25	1.05	0.18	0.30

Table 4: Total core usage for the components of Homa in the benchmarks from Figure 3; 1.0 corresponds to 100% usage of one hyperthread (work is actually spread across many cores). Polling time is listed separately, and is not included in `homa_recv`. The times for system calls do not include time to enter and leave the kernel. "W5-6K" shows W5 with the MTU increased from 3000 to 6000 bytes.

Using statistics maintained by the switch, I found that Homa's maximum buffer occupancy across all ports ranged from 246 KB for W2 to 8.5 MB for W5, well within the pool's 13 MB capacity. To confirm the number for W5, I reduced the pool size to 9 MB and reran the benchmark: its performance was unaffected and the switch recorded no packet drops. I then further reduced the pool size to 7 and then 6 MB. At 7 MB there was still no performance degradation, but Homa dropped 500–1000 packets per second per port under the W5 workload. With 6 MB of buffer space, Homa dropped about 90,000 packets per second per port under W5 (a rate of 0.13%) and performance degraded severely. For comparison, TCP performance dropped noticeably when buffer space was reduced to 10 MB and severely at 6 MB; DCTCP experienced no degradation until buffer space dropped below 2 MB.

Homa's maximum buffer usage amounts to about 220 KB per port, which is only slightly higher than the value that produced poor results for Hai et al. It seems likely that the dynamic buffer pool explains the difference in our results. Assuming that buffer usage scales with link speed, buffer requirements (and availability) can be characterized in units of Kbytes of buffer space per Gbps of aggregate switch bandwidth. Homa's maximum usage was 8.5 KB/Gbps, though it performed well with only 7 KB/Gbps. Looking forward to 100 Gbps networks, Broadcom's Tomahawk 3 switching chip [35] provides 7.5 KB/Gbps of buffer space, which appears to be adequate for Homa. I believe that Homa's buffer usage can be reduced, but I leave this to future work.

6 Software Overheads

As discussed in the preceding section, Homa/Linux eliminates congestion as a significant performance factor. Tail latency is now limited by software overheads, not network congestion. New congestion control schemes are unlikely to be impactful unless they also reduce software overheads (and simulation results that omit those overheads may be misleading). Significant gains may be had if overheads can be reduced: for example, the RAMCloud implementation of Homa [25] has much lower software overheads than Homa/Linux; as a result, it provides 8x lower P99 latency for short messages in W4

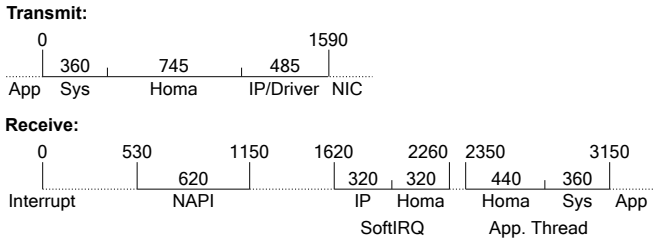

Figure 7: Timelines (in ns) for sending and receiving short messages under the best-case conditions of Table 2; the receiving thread is polling in `homa_recv` when the message arrives (an additional 2000 ns are incurred if the thread is sleeping). All times are in ns. "Sys" refers to Linux code for entering and leaving system calls.

(unfortunately, the RAMCloud implementation is impractical, as discussed below). This section provides more data on software overheads.

As discussed previously, software overheads (in particular, hot spots due to conflicts between the thread scheduler and network load balancers) are the primary source of tail latency for Homa/Linux. In addition, software overheads limit throughput and place excessive demands on cores for packet processing. They particularly impact throughput for short messages: the 1.6 Mops/sec maximum request rate reported for servers in Table 2 consumes resources equal to 18 of the machine's 20 cores, even though no meaningful work is done in servicing the requests. Table 4 shows core utilization for each of the four workloads; W2 and W3, which have the smallest message sizes, consume almost all of the cores, and yet they are unable to fully utilize a 25 Gbps network. W4 and W5 process larger packets, so they use fewer cores. The pacer thread consumes about 0.7 core for W4 and W5, which have many large messages. Extrapolating from Table 4, Homa/Linux will require at least 18 cores (excluding polling) to drive a 100 Gbps network at 80% utilization.

One possible way to reduce software overhead is to increase the MTU. However, Table 4 shows only small benefits: doubling the MTU from 3000 to 6000 bytes reduces core utilization for W5 only 11%, and workloads with smaller messages benefit even less.

Figure 7 breaks down the best-case latency for sending and receiving short messages. The total round-trip software overhead is about 9.5 µs, consisting of 1.6 µs on each node to send a message and about 3.2 µs on each node to receive one. This is notably higher than the network time (from NIC queue to interrupt), which is about 5.6 µs per round-trip. Figure 7 and Table 4 show that software overheads are distributed across many components; there is no single culprit to optimize.

Unfortunately, Figure 7 significantly understates the latency for real workloads. Software overheads increase by 2–3x when load balancing is used. This can be seen in the upper left corner of Figure 4. The "Homa best case" RTT is about 15 µs; in this measurement, all protocol processing occurs on a single core on each machine. However, in the W4 workload, where protocol processing is distributed across cores, the median RTT for short requests is 3x higher (about 47 µs); only 1% of requests complete in under 30 µs. An analysis of detailed timetraces showed that each stage of processing from Figure 7 slows down by a factor of 2–3x (presumably due to cache coherency traffic?); again, there is no single culprit.

7 The Future of Transport Protocols

The preceding sections showed that transport protocols implemented in kernel software carry a painfully high cost in terms of both application latency and core usage. Although some improvements may be possible, such as better thread schedulers that reduce hot spots from load balancing, fundamental challenges will remain. This section discusses the possibility of moving transport protocols out of the kernel.

7.1 Moving transport protocols to user space

Several recent projects have explored moving protocol processing to user space, including MICA [22], RAMCloud [29], IX [5], ZygOS [31], Shenango [27], eRPC [19], and Snap [23]. At first glance, these systems appear to reduce software overheads significantly. For example, Shenango and RAMCloud can serve about 1 M requests/sec/core and eRPC can serve 10 M requests/sec/core, vs. just 0.1 M requests/sec/core for Homa/Linux. eRPC offers best-case latency of 3.7 µs, vs. 15.1 µs for Homa/Linux, and the RAMCloud implementation of Homa provides P99 latency for W4 less than 15 µs, vs. about 100 µs for Homa/Linux.

However, most of these systems have significant simplifications and/or restrictions, such as the following:

- They measure under unrealistic conditions like those in Table 2.
- They don't do load balancing, or do it in a hand-optimized fashion that eliminates hot spots and the 2–3x load-balancing overhead discussed above.
- The measurement workloads contain only short or only long messages (combined workloads are both more realistic and more challenging).
- They don't consider congestion control.
- They assume that every application can implement the protocol independently (see below).

Many of the overheads experienced by Homa/Linux cannot be eliminated; for example, at least one core will still be consumed polling the NIC and receiving packets, and if output rate limiting is needed, another core will be consumed for that.

Of the user-space systems listed above, only Snap provides a full-featured production implementation. However, it reduces software overheads by only about 2x compared to Homa/Linux (Snap requires 7–14 cores to drive a 100 Gbps network at 80% load bidirectional, vs. 18 for Homa/Linux; this is still a steep price to pay). Snap appears to incur load balancing overheads similar to those for Homa/Linux. A single Snap core without load balancing can drive a 100 Gbps network at 80% load in one direction, but when load balancing is enabled (as in the numbers for 7–14 cores above) throughput/core drops by 3.5–7x. Furthermore, Snap reported P99 latencies for short messages of 300–400 µs under high network loads, which is 3–4x higher than Homa/Linux. This suggests

that user-space implementations will not be a panacea for the problem of high software overheads.

One of the challenges with user-space protocol implementations is that they perform best when every application can implement the protocol independently. However, not all protocols fit this model. For example, Homa requires global state for congestion control (for grant management and the pacer). If global state is required, then the protocol must be implemented in a shared service and packets must pass through that service on their way to and from the network. This introduces extra core/thread crossings, adding significant overhead. Snap uses this approach, which could explain why its overhead is higher than many other user-level protocol implementations.

7.2 Moving transport protocols to the NIC

The challenges with software transport protocols will only get worse in the future, since network speeds are increasing while CPU speeds are static. We are approaching a point where it no longer makes sense to implement transport protocols in software. The alternative is to move them entirely to the NIC; applications would communicate directly with the NIC using kernel bypass. Packets would no longer be visible to host software: all communication with the NIC would be in terms of messages. In addition, the NIC would implement intelligent load balancing, such as distributing incoming requests across available threads in a service, so that packets do not need to pass through an additional core just for load balancing.

Moving the transport protocol to the NIC would reduce end-to-end application latency by at least 5x, increase small-message throughput by 5–10x, and use silicon real estate more efficiently, freeing cores currently used for inefficient protocol processing so they can run application code instead. Reducing end-to-end latency would also reduce RTTbytes, cutting buffer consumption in the switches. This transition could be as impactful for system performance as the transition from programmed I/O to direct memory access in the 1960's.

Designing such a NIC will be challenging; it will require a new architecture that combines a line-rate packet processing pipeline with enough programmability to allow a variety of transport features and to support easy transport maintenance and evolution. The NIC will also need to support network virtualization features commonly implemented in software today [30, 9]. There exist systems that provide some of the required features, but none is fully satisfactory. For example:

- RDMA NICs provide kernel bypass and low latency, but their mechanisms for congestion control and load balancing are inadequate. In addition, existing NICs are not open or programmable.
- Today's "smart NICs" have inadequate performance and/or programmability. Smart NICs come in two flavors. The first is implemented with many general-purpose cores. These NICs still implement transport protocols in software, with all the performance problems discussed above. The second flavor is based on FPGAs; they can potentially provide adequate performance, but are difficult to program.
- New packet processing pipelines such as P4 [6, 7] operate at line rate and are programmable, but they do not currently have enough power to implement transport protocols. As one example, P4 does not provide long-lived state that is required for transport protocols.

One promising direction of research is to extend P4 with additional mechanisms to meet the needs of transport protocols [18]; this work is still embryonic.

7.3 The future of TCP

TCP has a long and illustrious history, and it has found use across an extraordinary range of technologies and environments. However, datacenters did not exist when TCP was designed, and virtually every aspect of the TCP design is wrong for the datacenter:

- Its connections create high space and time overheads.
- Its stream orientation is awkward for applications and causes high tail latency due to head-of-line blocking.
- Its fair scheduling increases tail latency for all message sizes.
- Its sender-driven congestion control ensures buffer occupancy at high loads, which drives up tail latency.
- It doesn't take advantage of in-network priority queues.
- It requires in-order delivery, restricting opportunities for load balancing both in network hardware and host software.

Furthermore, TCP's high implementation complexity will make it difficult to implement in hardware. Thus, it is difficult to imagine a path to high performance datacenter networking that is based on TCP.

8 Related Work

The last decade has seen many proposals to improve TCP performance and/or solve the congestion control problem for datacenters. Examples include DCTCP [1], D^3 [37], HULL [2], D^2TCP [36], PDQ [17], pFabric [3], PIAS [4], QJUMP [13], pHost [11], Karuna [8], and NDP [15]. Of these, DCTCP appears to be the only one with a readily-available Linux implementation that can be compared with Homa/Linux.

Homa's congestion control mechanism is effective against congestion at the network edge, but it does not address congestion in the core. Several other recent projects have addressed core congestion, including TIMELY [24], HPCC [21], and Swift [20]. Swift and Homa could probably be synergistically combined, with Swift dynamically adjusting RTTbytes to manage congestion in the core while Homa eliminates it at the edge. At the same time, it seems possible that core congestion is due primarily to TCP's requirement of flow-consistent routing, and that a datacenter using Homa with packet spraying would not experience significant core congestion; this would be an interesting experiment for future work.

Aeolus [14] proposed modifications to Homa to improve performance when buffer space is exhausted. However, measurements in Section 5.5 indicate that buffer exhaustion is unlikely to occur in modern switches with shared buffer pools, so the Aeolus modifications are not necessary.

As mentioned in Section 6, several recent projects have developed network transports outside the kernel in order to avoid the overheads of an in-kernel transport. Examples include MICA [22], RAMCloud [29], IX [5], ZygOS [31], Shenango [26, 27], eRPC [19], and Snap [23].

9 Conclusion

This paper has demonstrated two things. First, it has shown that the Homa transport protocol can be implemented in a practical setting that allows it to be used by a variety of applications. Homa/Linux retains the benefits of Homa's congestion-control mechanism and outperforms TCP and DCTCP by a wide margin, offering order-of-magnitude lower tail latencies across a range of workloads and message sizes.

Second, this paper has shown that the battle for networking performance is shifting from network protocols to software overheads. Increasing network speeds make it ever more difficult to do protocol processing in software, and overheads increase as more cores are harnessed. If these overheads can be eliminated, for example with new NIC architectures, another factor of 5–10x in networking performance is possible.

10 Acknowledgments

This work was supported by Cisco, Fujitsu, Google, Huawei, NEC, and VMware. Bare-metal computing resources were provided by CloudLab [10]. Collin Lee, Yilong Li, Amy Ousterhout, and Seo Jin Park provided helpful comments on drafts of this paper. The paper also benefited from comments and suggestions from Michio Honda, the paper's ATC shepherd, and anonymous reviewers, as well as copy editing by Geoff Kuenning.

11 Appendix

This appendix provides additional details on a few aspects of the Homa/Linux implementation.

11.1 GRO packet batching

As discussed in Section 4.1, Homa/Linux collects incoming packets into batches for SoftIRQ processing without regard to message structure. Unfortunately, the Linux infrastructure segregates incoming packets using a hash of packet header fields. When a new packet arrives, Linux uses its hash to find a list of packets being held for batching (if any) that match that hash. Then it passes the new packet and the held list to a transport-specific function. The transport-specific function can batch the packet with an existing packet on the list (by incorporating it into a sublist within the existing packet). If the transport-specific function chooses not to batch the new packet with an existing one, then Linux adds the new packet to the list as a "root" for future batching.

This default mechanism prevents transports from batching packets that hash to different lists. However, the transport-specific function has a third option, which is to indicate that it has completely processed the packet, so Linux should not add it to the list or take any other actions. Homa/Linux takes advantage of this feature. Homa/Linux keeps track of a distinguished held list for each core (the one corresponding to the first packet received by that core). When packets arrive for any other held list, Homa/Linux ignores that list, merges the packet with the first Homa packet on the distinguished list instead, and indicates to Linux that the packet has been fully processed. As a result, all incoming Homa packets are merged together into a single batch. When the batch is transmitted to SoftIRQ, the distinguished list is reset.

11.2 skbuff management

Homa/Linux uses regular Linux skbuffs for packet buffering. It increments the reference count on transmitted skbuffs in order to retain them until they have been acknowledged. The original skbuff cannot be retransmitted because transmission is not idempotent; for example, lower-level protocol headers get prepended as the buffer traverses the IP stack. Thus, if retransmission is required, Homa/Linux copies the skbuff's data into a new skbuff.

Homa/Linux keeps the skbuffs for an incoming message in a list sorted by offset in the message. New packets are inserted into the list starting at the back (highest offset). With this approach, packet insertion will not require traversing many list elements unless the packet has been delayed a long time.

11.3 Synchronization details

Homa/Linux's approach to socket locking allows it to avoid the awkward mechanism used elsewhere in Linux for locking sockets in SoftIRQ handlers (a SoftIRQ handler could have interrupted a background thread that holds the socket lock needed by SoftIRQ, so SoftIRQ cannot wait for socket locks; when it finds a locked socket, it defers packet processing until the thread releases the lock). Instead, Homa/Linux uses spinlocks for sockets, with interrupts disabled. This prevents an application thread from being interrupted while holding a socket lock.

Homa/Linux takes advantage of the Linux RCU mechanism to prevent sockets from being deleted while operations are underway on them. This eliminates the need to acquire socket locks in some situations, and is particularly useful in situations where it would have been necessary to acquire the socket lock before acquiring RPC locks (this would violate ordering constraints necessary to prevent deadlock).

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[2] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[4] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 455–468, Berkeley, CA, USA, 2015. USENIX Association.

[5] 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)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87—-95, July 2014.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[8] L. Chen, K. Chen, W. Bai, and M. Alizadeh. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *Proceedings of the ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 174–187, New York, NY, USA, 2016. ACM.

[9] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, Apr. 2018. USENIX Association.

[10] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[11] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1:1–1:12, New York, NY, USA, 2015. ACM.

[12] Google. gRPC: A High Performance, Open-Source Universal RPC Framework. `http://www.grpc.io`.

[13] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, 2015. USENIX Association.

[14] S. Hai, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '20, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery.

[15] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichik, and M. Mojcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, SIGCOMM '17, pages 29–42, New York, NY, USA, 2017. ACM.

[16] Homa GitHub repository. `https://github.com/PlatformLab/HomaModule`.

[17] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 127–138, New York, NY, USA, 2012. ACM.

[18] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021.

[19] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, Feb. 2019. USENIX Association.

[20] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '20, pages 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 44—-58, New York, NY, USA, 2019. Association for Computing Machinery.

[22] 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 Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[23] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19,

pages 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[24] R. Mittal, V. 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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[25] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221—-235, New York, NY, USA, 2018. Association for Computing Machinery.

[26] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, Feb. 2019. USENIX Association.

[27] A. E. Ousterhout. *Achieving High CPU Efficiency and Low Tail Latency in Datacenters*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2019.

[28] J. Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, Palo Alto CA, USA, 2018.

[29] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[30] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[31] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[32] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[33] R. Sivaram. Some Measured Google Flow Sizes (2008). Google internal memo, available on request.

[34] Apache Thrift. `https://thrift.apache.org`.

[35] BCM56960 Series: High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series`.

[36] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 115–126, New York, NY, USA, 2012. ACM.

[37] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61, New York, NY, USA, 2011. ACM.