JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters

Yilong Geng*, Vimalkumar Jeyakumar†, Abdul Kabbani‡, Mohammad Alizadeh§
*Stanford University, †Tetration Analytics, ‡Google, §MIT CSAIL

Abstract

We present JUGGLER, a practical reordering resilient network stack for datacenters that enables any packet to be sent on any path at any level of priority. JUGGLER adds functionality to the Generic Receive Offload layer at the entry of the network stack to put packets in order in a best-effort fashion. JUGGLER's design exploits the small packet delays in datacenter networks and the inherent burstiness of traffic to eliminate the negative effects of packet reordering almost entirely while keeping state for only a small number of flows at any given time. Extensive testbed experiments at 10Gb/s and 40Gb/s speeds show that JUGGLER is effective and lightweight: it prevents performance loss even with severe packet reordering while imposing low CPU overhead. We demonstrate the use of JUGGLER for per-packet multipath load balancing and a novel system that provides bandwidth guarantees by dynamically prioritizing packets.

Categories and Subject Descriptors Networks [*Network types*]: Data center networks; Networks [*Network components*]: End nodes—Network servers

Keywords Datacenter, Packet reordering, Generic Receive Offload

1. Introduction

Datacenter networks have demanding and diverse performance requirements, from massive bandwidth for "big-data" workloads, to microsecond-level tail latency for request response workloads, to performance predictability and high utilization in a shared public cloud. To meet these requirements, researchers have made significant progress in recent years, developing better network architectures and systems, including new topologies and routing techniques [4, 5, 8, 20, 24, 35, 41, 49], transport mechanisms [7, 9, 22, 25], and quality of service solutions [28, 29, 34, 43].

Many of these systems benefit from flexible, fine-grained control over the routing and scheduling of packets. For example, per-packet load balancing schemes have been shown to attain near-ideal tail latency at high network utilization [18, 24, 49]. Flow scheduling mechanism such as pFabric [7] have been shown to deliver near-optimal flow completion times by dynamically increasing the scheduling priority of a flow's packets as it nears completion. As datacenter networks evolve to higher speeds, more stringent latency targets, and more diverse workloads, it is likely that future systems will need even more precise control over the routing and scheduling of packets (§2).

To enable precise control, a datacenter network should ideally allow *any packet to take any path at any level of priority*. Unfortunately, this level of flexibility is not currently possible due to a longstanding "soft" requirement in TCP/IP networks: the packets of a TCP flow need to be delivered *in order* to the TCP receiver. Historically, this requirement stems from the fact that TCP treats holes in packet arrivals as a signal for packet loss. Packet reordering thus severely degrades TCP throughput. Current deployments therefore typically route all packets of a flow on the same path (using ECMP routing [20, 48]) and at the same priority. Many research designs also take great care to eliminate or minimize packet reordering [7, 8, 30, 46], often at the expense of added complexity and a penalty in performance.

In this paper, we present JUGGLER, a practical reordering resilient network stack for datacenter networks that enables per-packet path and priority decisions. The goal of JUGGLER is to remove the constraint of in order packet delivery from datacenter network design. Taken to the extreme, JUGGLER enables systems that *systematically* reorder packets, such as the aforementioned per-packet load balancing and dynamic prioritization systems.

JUGGLER sits at the entry point to the network software stack. It buffers out of order packets for a small number of active flows over short timescales (e.g., a few hundred microseconds). It then delivers packets in order to the above layers in a best effort manner. This minimal design is based

EuroSys '16, April 18 - 21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901334

on two key observations about the impact and nature of packet reordering in high speed datacenter networks.

First, the impact of packet reordering in high speed networks extends beyond TCP protocol issues. As observed in recent work [24], packet reordering also imposes a high *CPU overhead* at 10Gb/s and higher speeds. The reason is that critical optimizations such as *Generic Receive Offload (GRO)* [17], which merges bursts of incoming packets to reduce per-packet processing overhead, rely on in-order delivery. Therefore, JUGGLER operates at the GRO layer to handle reordering while merging packets for efficiency. This architectural choice has important practical implications for JUGGLER's design (§3). For instance, naively maintaining state for every TCP connection in GRO is not possible as it opens up a vector for potential memory exhaustion denial of service attacks (§3.3).

Second, although packet reordering can be frequent, the *delay* for out of order packets can be kept small in datacenters (e.g., to a few hundred microseconds or less). The reason is that datacenter networks have low and uniform packet delays; by design, datacenter fabrics have nearly identical latency on different paths and datacenter transports are optimized to keep queueing delays small [6].[1] This insight enables JUGGLER to hide almost all out of order packets from the transport layer despite tracking state for only a small number of flows over short timescales. This keeps the design simple, efficient, and safe because of the limited resource requirements. We describe JUGGLER's design in depth in §4.

We implement JUGGLER in the Linux stack (§4.4), and evaluate it extensively using 10Gb/s and 40Gb/s hardware testbeds (§5). Our evaluation shows that JUGGLER is easy to tune and imposes almost zero overhead in the absence of reordering. JUGGLER handles severe packet reordering with no impact on throughput and low CPU overhead. For example, in adversarial stress tests, it uses less than 5% and 20% of a single core at 10Gb/s and 40Gb/s speeds respectively. We also demonstrate two systems enabled by JUGGLER: per-packet load balancing with near-ideal performance at up to 90% network utilization, and a novel system that passively provides bandwidth guarantees to a set of flows by dynamically prioritizing flows which send below their guarantee.

The source code for JUGGLER is publicly available at: `https://github.com/gengyl08/juggler`.

2. The Case for Reorder Resiliency in Datacenter Networks

In this section, we use several use-cases to argue why a reordering resilient network stack that enables packets to take any route at any priority is desirable in datacenter networks.

[1] Recent measurements from Microsoft datacenters [23] show that the 99th percentile server-server RTT is slightly above 1 millisecond, with only tens of microseconds of that time due to network queuing delay.

2.1 Flexible and dynamic packet scheduling

Dynamic packet scheduling is the ability to change the network priority of packets in real time based on the needs of the flow or other stochastically varying network conditions. Changing a packet's priority relative to another of the same flow can cause the packets to arrive out of order, because packets of different priorities are queued separately in switches and can experience different queueing delays. However, dynamically changing a flow's priority is a powerful technique for fine-grained traffic differentiation and flow scheduling controlled by end-hosts [7, 38, 39]. For example, pFabric [7] dynamically increases a flow's priority as it nears completion to implement the *Shortest Remaining Processing Time (SRPT)* scheduling policy and achieve near-optimal flow completion times.

We showcase a novel application of this technique: providing a minimum bandwidth guarantee to one or more traffic flows of a VM (or an application running at an end-host), using only two priority levels in the network. The key idea is for end-hosts to mark packets of the flows as high or low priority based on the measured rate relative to the guarantee. One way to do this is to mark packets as high priority with a *probability*, p, and periodically adapt p as:

$$p \leftarrow p + \alpha(\mathrm{R_t} - \mathrm{R_m}). \tag{1}$$

Here, R_t and R_m are the target and measured rates, and α is a gain factor chosen to stabilize the control loop. Notice that if $R_m < R_t$, p will increase, causing a larger fraction of the flow's packets to be sent at high priority. This in turn causes the flow's rate to increase. It is not difficult to see that as long as the high priority class is not over-committed (i.e. the guarantees are feasible), this mechanism will ensure that the guarantees are met. However, for this to work, the end-host must be resilient to packet reordering induced by dynamically changing packet priorities. Otherwise, as Figure 1 shows, the flow will fail to achieve its guaranteed bandwidth using a standard "vanilla" network stack.

The above approach has some interesting advantages over existing systems for providing bandwidth guarantees. Most prior designs [10, 29, 42, 47] require adding a rate control layer to the hypervisor, which buffers packets and throttles traffic to ensure that guarantees are met. Besides additional complexity, these designs do not co-exist with techniques such as SRIOV [3] that bypass the hypervisor. By contrast, the above design is entirely *passive*: it simply measures the achieved rate and sets packet priorities. Further, the approach is quite general; notice that R_t and R_m could be other metrics such as 99^{th} percentile flow completion time. On the other hand, the mechanism does rely on the TCP stack of the VMs to function properly, and may only be appropriate in trusted environments such as a private datacenter.

We evaluate this technique further in §5.3.1.

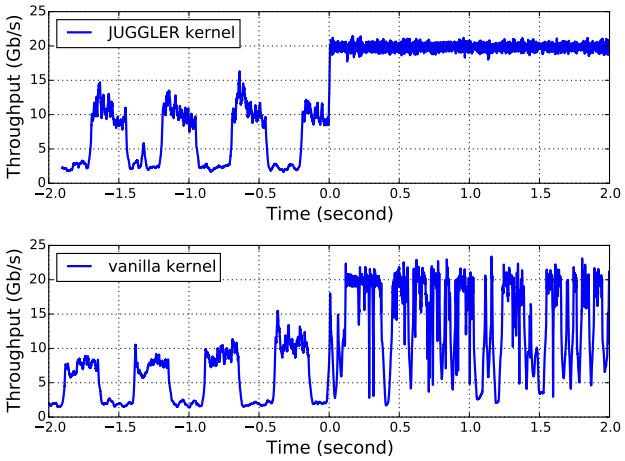

Figure 1. Bandwidth guarantee using dynamic packet scheduling: 8 flows share a 40Gb/s link. Before time 0, each flow gets 5Gb/s on average. At time 0, we begin dynamically prioritizing the packets of one of the flows to give it a 20Gb/s bandwidth guarantee. With JUGGLER, the flow quickly achieves the desired throughput. The vanilla kernel (without JUGGLER) however has widely variable throughput because of its inability to handle packet reordering.

2.2 Fine-grained network load balancing

Prior work has shown that flow-level load balancing (e.g., ECMP [26]) creates hotspots and hurts network utilization [8, 24, 44, 49]. Therefore, state-of-the-art load balancing designs such as CONGA [8] and Presto [24] split flows across different paths to achieve more accurate traffic balance.

Per packet load balancing would of course be most accurate. However, to avoid the detrimental affects of reordering on TCP, most existing designs avoid per packet decisions [8, 30, 46] and use other workarounds. For example, CONGA [8] operates at the level of short bursts of traffic called *flowlets* [31] to eliminate almost all packet reordering seen at the end-host. This detection logic requires new network hardware and is not as precise as per packet decisions. Presto [24] gets closer to ideal per packet load balancing by load balancing fixed-size *TCP segmentation offload (TSO)* blocks [15].[2] However, per-packet load balancing can outperform per-TSO load balancing at high network utilization (see Figure 20 for an illustration).

Our point is not to criticize the above works. On the contrary, our point is that if the network stack were fully reorder-resilient, it would simplify the above designs, and help them achieve even better performance.

[2] TSO sends as much as 64KB of data (45 MTU-sized packets) to the network interface as one large packet to reduce CPU overhead.

2.3 Simpler and cheaper datacenter switches

High-radix "chassis" switches employed at the spine tiers of datacenter networks are designed in a modular fashion [1, 27], with multiple switching ASICs that are internally connected in a Clos-like topology. To deliver non-blocking performance, these switches load balance traffic across parallel paths through the internal interconnect. Existing switches use various techniques to maintain packet order, such as adding reordering buffers at the egress devices, employing flow-level hash-based load balancing with internal speedup (extra capacity) to compensate for load imbalances, or centralized arbitration of the internal fabric. These designs add complexity and cost and could be avoided if the end-host network stack was resilient to reordering.

3. Design Considerations

At a high level, the task of a reorder-resilient network stack is simple: buffer out of order (OOO) packets and deliver them in the right order to the application. However, several practical challenges arise at high speeds such as 10Gb/s and 40Gb/s. We now discuss the main architectural considerations for developing a reorder-resilient network stack for high speed networks.

3.1 Where should it be implemented?

Figure 2 shows a representative overview of today's network stack. The network device driver first processes packets arriving on the network device and hands them off to the generic receive offload (GRO) layer that handles per-flow packet batching. GRO assumes the first packet of a flow in a batch is in sequence and continues to merge packets as long as the *packet arrivals are in the sequence number order*. It flushes the batched packet (sending it to the higher layer) whenever its size exceeds a preconfigured maximum (64KB) or when the next packet is not in sequence. The batched packets are then demultiplexed through several layers; the network filtering layer handles both stateless and stateful packet and flow operations implemented by iptables modules (e.g., `conntrack` [2]), followed by the protocol layer that handles protocol-specific functionality (e.g., TCP) at the socket layer. Finally, the application is woken up to receive data from its socket.

In an experiment where we deliberately reordered packets (see §5 for details), we found that there are two issues that need fixing. First, reordering breaks GRO's batching functionality resulting in high CPU usage. Second, the TCP stack treats mis-sequenced packets as a signal of packet loss due to an increased number of duplicate acknowledgements. We confirmed that these two issues are orthogonal in the sense that they can occur independently of one another and can both contribute to reducing the flow's throughput.

There are three main candidates for where to "fix" reordering: (1) add OOO queues at the GRO layer to fix batching and reordering together; (2) fix the transport layer (TCP)

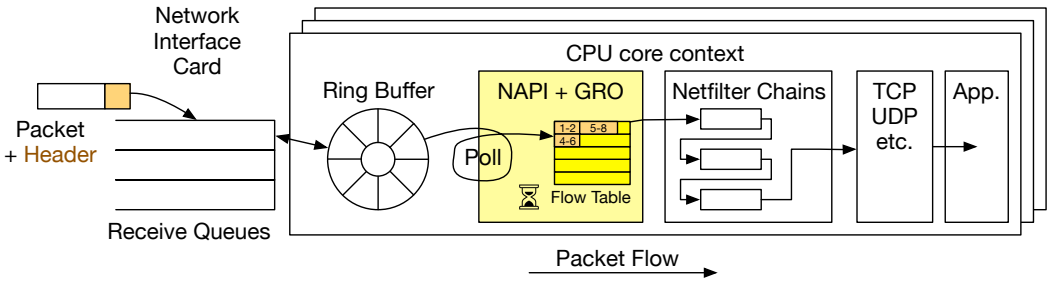

Figure 2. The architecture of JUGGLER. JUGGLER uses OOO queues to put packets back in order. In-sequence packet flushing decisions are made after merging every packet. Timeouts are checked at polling completions.

to gracefully handle out of order packets; or (3) a combination of batching packets *regardless of order* at GRO and correcting the order at TCP.

Since putting packets back in order requires understanding protocol semantics such as TCP flags and sequence numbers, dealing with reordering at the TCP layer is architecturally appealing. However, we argue that the first solution (fixing both batching and reordering in GRO) is necessary in practice. There are two reasons. The first reason is that the other solutions have higher CPU overhead. Specifically, TCP changes cannot prevent the per-packet overhead caused by the inability of GRO to batch out of order packets [24]. Also, batching packets regardless of order in GRO also has notably higher CPU overhead relative to the first approach. The reason is that non-contiguous packet payloads cannot be merged into a larger segment (`sk_buff`). Instead multiple `sk_buffs` would have to be *chained* in a linked list (see Figure 3). We implemented this approach and found that it causes 50% more CPU usage due to more cache misses in a simple experiment with in-order traffic.

Another pragmatic advantage of fixing reordering at the GRO layer is that several modules after GRO (iptables modules, stateful connection tracking `conntrack`) rely on in-order delivery to correctly infer TCP state machine for stateful packet filtering. Therefore, from a software-engineering perspective, encapsulating a common packet reordering functionality outside the netfilter layer provides a cleaner interface of in-order packets to downstream modules.

3.2 How many packets should be buffered?

Buffering a large number of packets in OOO queues at the GRO layer may incur a high memory overhead. More importantly, it can make searching the queue for the right position for each incoming packet costly in terms of CPU overhead. Though this is a valid concern, we argue that it can easily be overcome in datacenters.

The extent to which packets arrive out of order is a function of the difference in latency across different network paths and priority levels. However, the latency difference is expected to be small in a well-engineered datacenter network. In fact, datacenter topologies [4, 20] and transports such as DCTCP [6] are designed to achieve low per packet latency, even at high utilization. Measurements from a production datacenter in Microsoft [23] show that the latency is typically less than a millisecond. These measurements used `ping` and included delays at the end-host stack, which contributes to the `ping` latency, but does not cause *out of order* packets. Further, they were from a network with per-flow load balancing. As we show in §5, per-packet load balancing can significantly reduce queueing latency and further lower the latency difference between network paths. Exploiting the low latency in datacenter networks allows JUGGLER to eliminate almost all reordering by buffering as little as 100s of microseconds worth of packets.

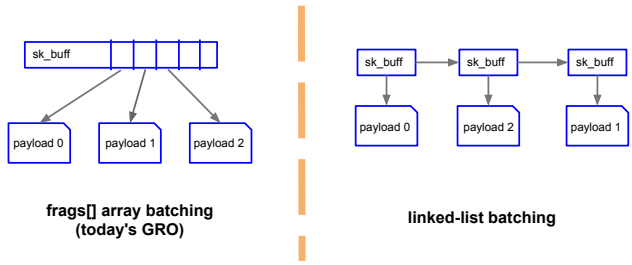

Figure 3. Two options to merge multiple `sk_buffs` into a larger segment. The left figure shows how today's GRO merges in-sequence packets using `sk_buff`'s `frags[]` array, while the right figure shows a linked-list of out of order `sk_buffs`. The latter increases CPU overhead due to higher cache misses.

3.3 How many flows should be tracked?

Implementing an OOO queue at GRO raises further questions. It is not easy to scale a naive implementation that keeps track of every open connection at this layer, as a datacenter server today can have millions [45] of established connections. Keeping state for all such established connections can quickly become prohibitive.

Even if tracking millions of connections were feasible through careful engineering, it raises concerns regarding security and accountability. For instance, consider the crucial task of accounting the memory allocated for a flow table and the OOO queue to guard against memory resource exhaustion at the kernel. At GRO, the kernel does not yet know to which process (or accounting entity) the packet is destined to. This requires further processing, such as routing the packet to the specific application context, which could be complex as most network stacks support expressive user-defined routing policies. Executing these checks can further increase the CPU cost in the critical path of processing packets that have only *just* been received from the network device. Failure to carefully manage memory by flushing out of order packets to downstream modules can render JUGGLER ineffective, increasing CPU usage.

Thus, any implementation has to not only efficiently keep track of network state to put reordered packet back in order, but also take precautions to not introduce functionality that could be exploited as a vector for denial of service attacks. Such attacks are not hypothetical [16], hence, there must be strict upper limit on the number of flows tracked in JUGGLER. If the limit is surpassed, flows must be evicted and their memory garbage collected.

Fortunately, the number of active flows on the timescale of the out-of-order delay is small. Consider a extreme case where JUGGLER buffers 1 millisecond worth of packets per flow and every received 1500B packet is from a new flow. With a 40Gb/s NIC and 16 receive queues, each receive queue needs to track only about 200 flows.

Based on the above insights, we design eviction strategies for JUGGLER to only keep track of active flows on the timescale of the out-of-order delay. In practice, we find that due to the small out-of-order delay and the inherent burstiness of traffic, the number of distinct active flows that JUGGLER must track is typically much smaller than the worst-case bound above (e.g., a few 10s in our evaluations). See §5.2.2 for details.

4. Design

We now delve into the design of JUGGLER. JUGGLER is an extension to the GRO layer in the network stack. Since GRO is in the critical path of receiving packets, our design choices are biased towards those that minimize CPU usage, cache misses, and memory usage. Thus, JUGGLER operates in a best-effort manner. It addresses a (tunable) bounded amount of reordering, and aggressively evicts stale flow state. We primarily focus on the handling of TCP traffic, however, our design principles hold for other transports such as SCTP that impose packet order as well.

Figure 4 shows the high-level overview of GRO with JUGGLER's modules. The NIC processes packets on the wire, typically hashing packets with the same canonical five-tuple flow to the same receive (RX) queue. From this point, different RX queues operate independently and have their private data structures; thus we focus our discussion on a single RX queue. Once the kernel receives a receive interrupt from the NIC, it switches to polling mode until it empties all packets on the queue, or up to a brief interval of time (at most 2 milliseconds). The kernel then hands off packets to GRO, whose batching interval is the same as the driver's polling interval. When the kernel finishes polling, standard GRO flushes *all* its packets and starts fresh from the next polling interval. This is where JUGGLER differs from GRO.

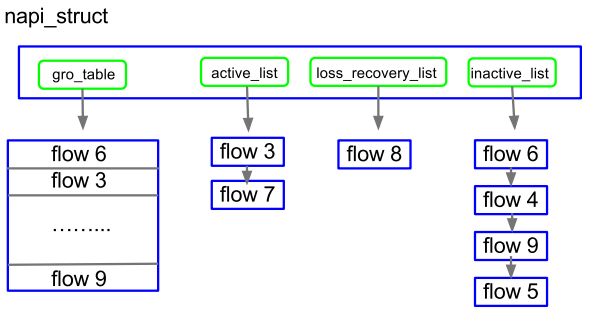

Figure 4. The data structures of JUGGLER. Each flow is tracked in `gro_table` and is part of exactly one of three doubly linked lists. Flows in the inactive list are safe to evict while the others are not.

4.1 Data structures

At a high level, JUGGLER keeps a list of flows in a structure called `gro_table`.[3] The flow entries are keyed by the canonical five-tuple and some additional state shown below:

```
struct flow_entry {
  struct five_tuple key;
  struct sk_buff_head *ofo_queue;
  u64 flush_timestamp;
  u32 seq_next;
  u32 lost_seq;
}
```

JUGGLER uses this state to determine when to buffer packets to reduce reordering and improve batching, and when to flush packets up the stack to avoid unnecessary delays. Additionally, JUGGLER uses two timeout conditions `inseq_timeout` and `ofo_timeout` to avoid buffering packets for too long. Flushing packets in a timely manner is especially important for the underlying TCP protocol to detect and recover from packet losses as quickly as possible.

The `key` field identifies the five-tuple flow, while `ofo_queue` is a doubly-linked list that stores packets sorted in sequence number order. The flow entry also tracks three other pieces of state: `flush_timestamp`, `seq_next`, and `lost_seq`. The field `flush_timestamp` is the last time (nanoseconds since

[3] For simplicity, our current implementation uses a linked list for `gro_table`. It could also be implemented as a hash table.

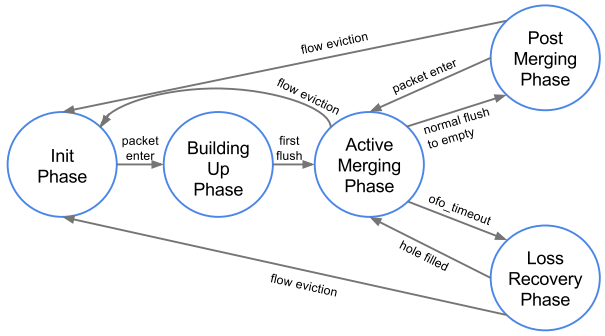

Figure 5. The life cycle of a flow in JUGGLER. Each phase corresponds to one of the lists in JUGGLER's data structure.

epoch) at which JUGGLER flushed packets. Unlike GRO, JUGGLER uses `flush_timestamp` to hold onto packets across polling intervals, and selectively flush packets when the last `flush_timestamp` exceeds the preconfigured value (`inseq_timeout`). The `seq_next` field is JUGGLER's best guess of the largest sequence number that has already been flushed. We discuss the rationale and semantics of this field in §4.2.2. The `lost_seq` field helps JUGGLER quickly identify lost packets from those arriving out of order (§4.2.5).

Finally, JUGGLER also maintains three other linked lists: the active list, the inactive list, and loss recovery list, which further optimize JUGGLER's performance as we explain shortly. To prevent memory resource exhaustion attacks, we aggressively limit the total size of per-flow state that we track in JUGGLER by evicting flows in the inactive list. In practice, we find that tracking a few tens of flows per `gro_table` is sufficient to handle realistic workloads (§5.2.2).

4.2 Design details

We walk through the design details by simulating the life of a flow through various phases in the GRO layer as shown in Figure 5. There are five phases in the life cycle of a flow: the initial phase, the build up phase, the active merging phase, the post merging phase and the loss recovery phase. Flows in the build up phase and the active merging phase are in the active list. Flows in the post merging phase are in the inactive list and ones in the loss recovery phase are in the loss recovery list. A flow is only in one list at any point in its lifetime and JUGGLER treats flows in each list differently. Table 1 lists each stage and its rationale, which we elaborate below.

4.2.1 Initial phase

For every packet, JUGGLER looks up the flow in `gro_table` to fetch its state. If this lookup fails, we say JUGGLER sees a flow for the first time. Note that this need not be the first packet of the flow from the perspective of the TCP stack. On the first packet, JUGGLER creates a new entry in the `gro_table` and adds this entry to the active list. Then, the flow transitions to the build up phase.

Phase	Rationale
Initial phase	JUGGLER sees packet for the first time with unknown `seq_next` (§4.2.1).
Build up phase	Learn initial estimate of `seq_next`, which can go backwards (§4.2.2).
Active merge phase	Merge and flush packets ensuring `seq_next` only moves forward (§4.2.3).
Post merge phase	Flow flushed and can be readily evicted if needed (§4.2.4).
Loss recovery phase	Flow flushed but evicting it can cause delays (§4.2.5).

Table 1. This table shows the phases in the lifetime of a single flow in JUGGLER and their design rationale. The rationales are explained in indicated sections in the paper.

Flush condition	Rationale
Packet sequence number is before `seq_next`	Likely to be retransmission
In-sequence segment reaches 64kB	Segment size limit reached
Packet has certain flags (e.g., PUSH, URGENT)	Protocol semantics necessitates urgent delivery
Packet differs from in-sequence segment in TCP options, CE marks, etc	Cannot be merged without losing information important to TCP
`inseq_timeout` triggers	Do not delay in-sequence packets too much
`ofo_timeout` triggers	Missing packet likely to be lost

Table 2. JUGGLER flushing conditions.

4.2.2 Build phase

This phase starts with the first packet of a flow in the initial phase and ends when JUGGLER flushes packets of this flow for the first time. On the first packet, JUGGLER sets `seq_next` in the flow entry to the packet's sequence number, as it is JUGGLER's best guess of the largest sequence number that has already been flushed to higher layers. As JUGGLER receives more packets, it updates `seq_next` by setting it to the smaller of `seq_next` and the new packet's sequence number.

Deciding when to flush exposes a tradeoff between timeliness and CPU efficiency. If JUGGLER flushes late, it can achieve high batching efficiency, but at the expense of delaying packets and increasing latency. However, flushing too soon reduces opportunities to batch (increasing CPU usage) and risks sending out of order packets up the stack. JUGGLER flushes an OOO queue whenever the flow satisfies one of several intricate conditions summarized in Table 2.

Flushing conditions. There are two types of flushing conditions: event-driven checks and timeout checks. Event driven checks (the first four rows of Table 2) are done with the arrival of a new packet. Most of these checks are self-explanatory and similar to standard GRO, except for the fact that JUGGLER only flushes packets that are in order starting from `seq_next`. Notice that JUGGLER flushes an incoming packet immediately if its sequence number is smaller than `seq_next` (except in the build up phase), as this is likely a retransmitted packet.

There are also two timeout based flushing conditions unique to JUGGLER, which are checked at the end of the polling interval and in *one* high resolution timer call-

back per gro_table. To avoid holding packets in JUGGLER longer than necessary, JUGGLER flushes all partially merged in-order segments that have been held longer than inseq_timeout. JUGGLER also uses a global ofo_timeout, which flushes *all* packets in the out of order queue and transitions the flow to the loss recovery phase (§4.2.5). After a flush, JUGGLER updates flush_timestamp to the current timestamp, and sets seq_next to the next expected (in order) sequence number.

Remark 1. The use of the build up phase to learn seq_next is a subtle aspect of JUGGLER's design. Since JUGGLER aggressively evicts flows from the inactive list (§4.3), all evicted flows will go through the build up phase when they re-enter JUGGLER. At that point, they are indistinguishable from new flows. One might consider setting seq_next to the first packet's sequence number on re-entry. However, it is likely that the first packet is out of order, which would cause subsequent packets in the same arrival burst to be flushed to upper layers. Instead, we wait for one polling interval to finish and allow seq_next to briefly go backwards in this build up phase. In a basic single flow experiment with reordering, we found that this simple optimization resulted in 6% fewer segments sent up the stack at no extra implementation cost.

4.2.3 Active merging phase

Following a flush in the build up phase, the flow enters the active merging phase. It stays in the active merging phase as long as its out of order queue is not empty. In this phase, JUGGLER tries to merge new packets with their adjacent packets, and flushes in-sequence packets on meeting flush conditions discussed above. If the out of order queue becomes empty after flushing in-sequence packets, the flow enters the post-merge phase.

JUGGLER uses seq_next to avoid unnecessarily buffering packets. Unlike in the build up phase where seq_next can go backward, in the active merging phase seq_next is only allowed to go forward. Packets before seq_next are inferred as retransmissions and thus not buffered. Figure 6 shows an example of this scenario.

4.2.4 Post merge phase

The post merging phase is for a flow to temporarily save its flow_entry state before JUGGLER sees packets of the flow again. Flows in this phase are good candidates to aggressively evict since a flow enters this phase from the active merging phase if its out of order queue becomes empty after flushing some in-sequence packets. When entering the post merging phase, the flow entry is removed from the active list and enqueued into the inactive list (Figure 4). If a packet after seq_next from the flow enters JUGGLER, the reverse process happens. The flow goes back to the active merging phase and its flow entry is added back to the active list.

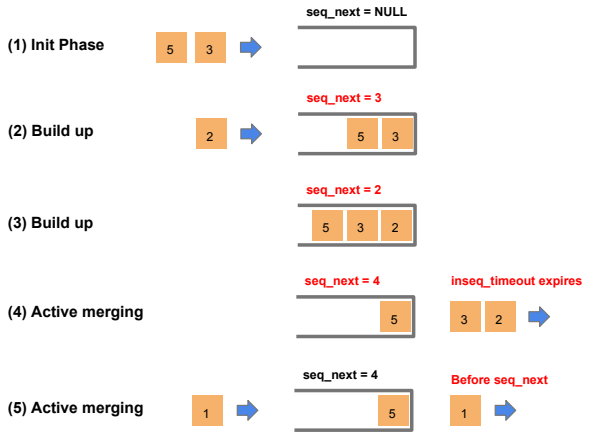

Figure 6. JUGGLER receives and buffers a flow's packets with sequence numbers 3, 5 and 2 in the build up phase. After flushing packets 2 and 3 (e.g., due to inseq_timeout), it sets seq_next to 4 and the flow enters the active merging phase. Now if the flow receives a retransmitted packet with sequence number 1, JUGGLER can immediately infer that it should not buffer the packet in its out of order queue because packets up to seq_next have already been flushed.

4.2.5 Loss recovery phase

Finally, the loss recovery phase helps identify flows that are most likely to have had lost packets. A flow enters the loss recovery phase (from the active merging phase) when its out of order queue is cleared due to an ofo_timeout expiration. JUGGLER infers this event as packet loss, since a missing packet did not arrive for at least a time period of ofo_timeout. JUGGLER treats these flows differently. In particular, it prefers to not evict these flows since their future packets are likely to have "holes" which could confuse JUGGLER into more timeouts (§4.3).

On entering the loss recovery phase, JUGGLER stores the first missing packet's sequence number in lost_seq. The flow then goes back to the active merging phase as soon as the hole is filled later. Figure 7 shows an example of this procedure. Note that JUGGLER operates in a best-effort fashion in that it only maintains the sequence number for the first lost packet, and does not require all the holes to be filled before moving the flow back to the active merging phase.

4.3 Flow eviction

So far, we discussed the life of a flow through JUGGLER's state machine, and focused primarily on the conditions under which JUGGLER flushes packets. We now discuss how JUGGLER manages flow state memory through evictions.

Recall that JUGGLER differs from standard GRO in maintaining per-flow state. It is critical to keep this memory footprint to a minimum so as to not create an opportunity for memory resource exhaustion attacks (as discussed in §3.3). Our design aggressively evicts flows from gro_table.

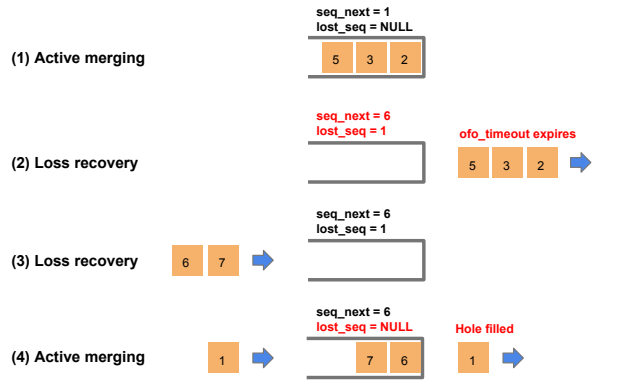

Figure 7. A flow is in the active merging phase with `seq_next` set to 1, and packets 2, 3, 5 in the out of order queue. When `ofo_timeout` expires, packets 2, 3 and 5 are flushed, and `seq_next` advances to 6. Also, `lost_seq` is set to 1 and the flow is added to the loss recovery list. Later, when JUGGLER sees packets 7, 6 and 1 in that order, JUGGLER enqueues 7 and 6, flushes 1 and adds the flow entry back to the active list. For simplicity, JUGGLER operates in a best-effort fashion and does not require all the holes to be filled before moving the flow back to the active merging phase. In this example, the flow exits the loss recovery phase even though JUGGLER never saw packet 4.

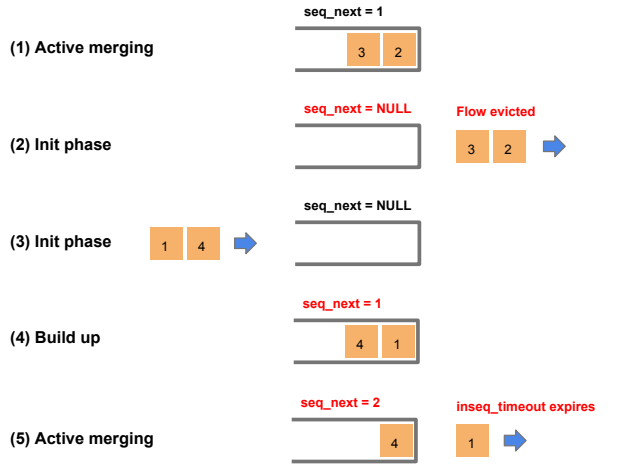

Figure 8. A flow's `seq_next` is 1 and packets 2 and 3 are in the out of order queue. JUGGLER flushes all these packets on eviction. After eviction, packets 4 and 1 arrive in that order, the flow enters the build up phase and `seq_next` is set to 1, which will be flushed after `inseq_timeout`. Packets 2 and 3 will never arrive as they have already been flushed, and packet 4 will therefore be flushed only after `ofo_timeout` expires.

When we ran JUGGLER without eviction under real workloads, we found that most flow entries are perpetually in the inactive list. There are several reasons for this behavior. First, most datacenter flows are short, lasting only a few round-trip times [6]. Second, large high throughput flows are bursty and exhibit ON-OFF patterns at 10–100 microsecond timescales due to TSO offload [15, 32]. Third, the delay difference between different paths in datacenter networks is very small (typically less than a few hundred microseconds). JUGGLER essentially only needs to track flows during TSO bursts, and on the timescale of the out of order delay. It can rapidly evict inactive flows without risk of reordering or loss in batching efficiency.

JUGGLER triggers flow eviction when it sees a new flow and `gro_table` is full. Eviction removes the flow's state and flushes all its packets to higher layers. However, there are subtleties on when JUGGLER should evict a flow. Specifically, evicting a flow which has holes in its out of order queue may cause it to get stuck and have to wait for a timeout when it re-enters JUGGLER in the future. Figure 8 shows how such a situation could happen: JUGGLER may wait on packets already flushed after the flow re-enters.

Thus, evicting flows in the active merging phase is counter-productive because there could be holes in the out of order queue. Similarly, it is easy to see that evicting flows in the loss recovery phase can be equally counter-productive. Thus, JUGGLER *first evicts a flow in the post merge phase*, because JUGGLER knows that these flows have empty out of order queues and their previous packets were flushed sequentially, leaving no holes. If there is no room for the new flow, JUGGLER evicts flows in an FIFO fashion from the active merge list.

4.4 Implementation

We implemented JUGGLER for Linux 4.1. It is about 1000 lines of code and involves changing the GRO layer without touching the TCP stack. We ensured that JUGGLER is API-compatible with GRO so no code outside GRO needs any change. Moreover, our implementation behaves identically to GRO when handling in-order traffic. JUGGLER is open source and the patch is available at `https://github.com/gengyl08/juggler`.

5. Evaluation

Our evaluation consists of two parts. First, we compare status-quo ("vanilla" Linux kernel) with a JUGGLER-enabled kernel and confirm that the latter incurs negligible CPU overhead, can be easily tuned, and is able to operate by only tracking a very small number of flows. Second, we showcase JUGGLER's utility by prototyping new bandwidth guarantee and load balancing applications that require the end-host to be reorder resilient to achieve their goal.

Unless otherwise noted, we set `inseq_timeout` to 15μs and `ofo_timeout` to 50μs, and run our experiments on a 40Gb/s two-stage Clos network with two uplinks from each of the ToR switch to the Stage 2 switches (Figure 19).

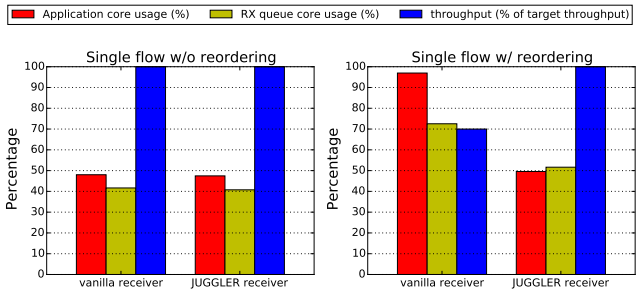

Figure 9. The CPU overhead of JUGGLER in the single flow case. Without reordering, JUGGLER does not introduce extra CPU overhead. In presence of reordering, the vanilla receiver's application core gets saturated and it loses throughput, while JUGGLER uses 10% more CPU and handles reordering gracefully.

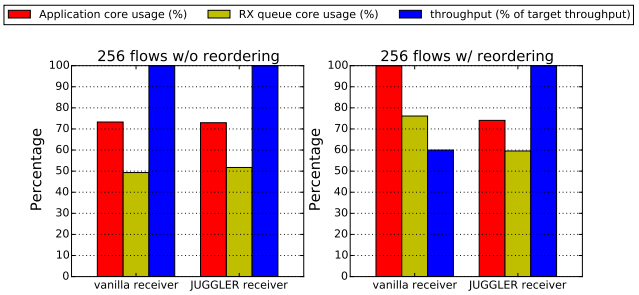

Figure 10. The CPU overhead of JUGGLER in the multiple flow case. The comparisons and results are similar to that of the single flow case.

5.1 CPU and latency overhead

5.1.1 CPU overhead

By the virtue of its design, JUGGLER's performance is identical to standard GRO when processing in-order traffic. For out of order traffic, however, JUGGLER's CPU overhead should be significantly less than that of the vanilla kernel. To understand the extent to which JUGGLER affects the CPU performance, we conduct two experiments, one with a single large flow, and another with many flows, trying to push JUGGLER to its flow tracking limits. The topology in this experiment is a two-stage Clos (similar to Figure 19)—the receiver is connected to the first ToR and the senders are connected to the other 4 ToRs.

In order to cause out of order packet delivery in the case when packets are sprayed, we generate some background traffic such that the average load on the sending ToR uplinks is 50%. We evaluate vanilla kernel and JUGGLER-enabled kernel under four scenarios, with 1 and 256 flows (32 senders across the 4 ToRs, each generating 8 flows), and with equal-cost multi-path (ECMP) and per-packet load balancing. ECMP load balancing provides a baseline without reordering, while per-packet load balancing creates packet reordering because of the variations in queuing on different paths caused by the background traffic. In all cases we aim all flows on a single RX queue on the receiver and rate limit the total throughput to 20Gb/s.[4]

We found that the CPU usage varies with different CPU affinity settings of the RX queue and the application. For the vanilla kernel receiving in-order traffic, we found that it is always the most efficient to pin the RX queue and the application on two different cores belonging to the same CPU socket. We use this same affinity setting for all the experiments for a fair comparison.

[4] We didn't use 40Gb because a single core cannot handle 40Gb/s of traffic in our tested.

Figure 9 and 10 show the CPU usage and throughput results in each setup. The key takeaways are as follows. The left subgraphs in the two figures show that when processing in order traffic, JUGGLER introduces no CPU overhead compared to the vanilla kernel. The right subgraphs in the two figures show that in the presence of reordering, the vanilla kernel's CPU overhead reaches almost 100% on the CPU core the application runs on (hence, falling short of reaching 20Gb/s in this case). However, with JUGGLER, we can sustain line-rate.

As soon as we start reordering in both Figure 9 and 10, the vanilla kernel TCP stack roughly sees 15 times more segments (of which 40% are out of order), sends 15 times more ACKs, and loses 35% of its throughput. On the other hand, JUGGLER hides almost all of the reordering from TCP, allowing it to operate as normal.

For a fair comparison with the vanilla kernel, we compare JUGGLER with reordering to the vanilla kernel without reordering. From Figure 9, by comparing "vanilla receiver" in left and "JUGGLER receiver" in the right graphs, we can see that JUGGLER uses less than 10% additional CPU on a single core to deal with 20Gb/s of traffic with reordering. Figure 10 shows similar results with 256 flows. While there is still scope for improving this overhead, we believe this is a modest cost to pay considering the advantages of per-packet load balancing.

5.1.2 Latency overhead

Since JUGGLER buffers packets and does extra processing in the critical receive path, one potential concern is the extra latency it could add to short flows. To investigate this, we set up an experiment in which one client sends 150 Byte RPC messages to a server, with no competing traffic in the network to avoid queueing latency. Given that without any reordering, JUGGLER is identical to standard GRO, we have found the median end-to-end latency is the same, with and without JUGGLER. Of course, as we show in §5.3.2, the overall latency improves significantly when JUGGLER is used together with per-packet network load balancing compared to existing per-flow load balancing schemes.

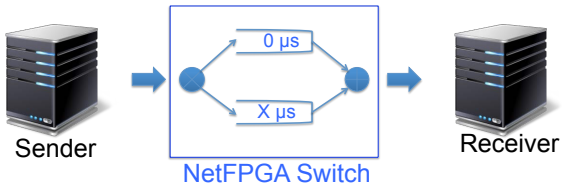

Figure 11. Testbed to understand the timeout parameters. Two hosts are connected by a NetFPGA-10G switch, which hashes each inbound packet to one of two output queues uniformly at random. The delay of each output queue can be configured per-packet to precisely control the amount of reordering seen by the hosts.

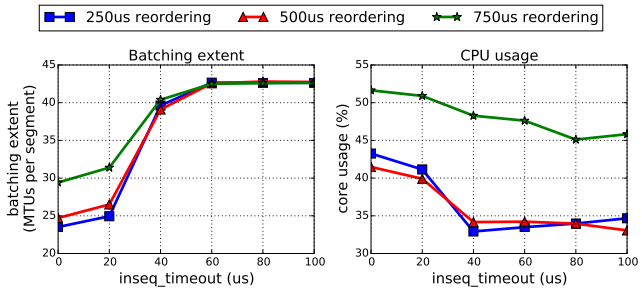

Figure 12. Tradeoff between batching efficiency and `inseq_timeout`. The figure illustrates that to maximize batching efficiency, there is little benefit to increasing `inseq_timeout` beyond the time it takes to transmit the maximum sized TSO segment at line rate (52μs at 10Gb/s).

5.2 Deep dive into JUGGLER

In this section, we run a series of targeted micro benchmarks to understand (1) how various tunable parameters affect JUGGLER's performance (§5.2.1), and (2) the size of `gro_table` required for JUGGLER's flow eviction policies (§5.2.2). For some of the micro benchmarks in this section, we use a NetFPGA-10G [13] testbed to precisely control the amount of reordering, while the other micro benchmarks are run on the 40Gb/s Clos topology in which background traffic and queueing delays cause reordering.

5.2.1 Timeout parameters

Recall that JUGGLER has only two key tunable, *global* parameters: `inseq_timeout`, which trades-off between batching, that is crucial for CPU overhead, and timely packet delivery; and `ofo_timeout`, which trades-off between putting packets back in order and timely reaction to packet loss.

In-sequence timeout: The parameter `insesq_timeout` ensures in-sequence packets are not buffered in GRO for too long. If `inseq_timeout` is too small, JUGGLER tends to flush in-sequence packets before they become large enough, increasing the number of segments seen by TCP (and hence per-packet overhead). If `inseq_timeout` is too large, JUGGLER tends to hold in-sequence packets unnecessarily longer, hurting the RTT of the flow. Therefore, the rule of thumb is that `inseq_timeout` should be configured to the minimum value that suffices for receiving the maximum batch size—a single 64kB TCP segment (or 45 MTUs worth of packets at line rate). This value is 52μs on a 10Gb/s network and 13μs on a 40Gb/s network.

To validate this, we ran an experiment with two machines connected by a NetFPGA-10G switch as in Figure 11. The sender sends a single TCP flow at line rate (10Gb/s). At the receiver, we measure the extent to which JUGGLER batches packets and the CPU usage for various `inseq_timeout` values. We run the same experiment 3 times at different levels of reordering, with the delay of the second queue on the NetFPGA switch set to 250μs, 500μs and 750μs. Figure 12 summarizes the relationship between `inseq_timeout`, JUGGLER's batching extent (average number of MTUs batched into a TCP segment), and CPU usage. Increasing `inseq_timeout` helps with batching and CPU usage until it reaches about 52μs. This threshold is not affected by the amount of reordering seen in the network.

The 52μs threshold is specific to 10Gb/s network speeds. We also verified that at 40Gb/s, the optimal value for `inseq_timeout` is 13μs. Also, notice that the batching extent is initially around 25 MTUs when `inseq_timeout` is 0. This is because we do not check the value of `inseq_timeout` after every packet, but only at polling completions; therefore, JUGGLER is still able to batch MTUs within individual polling cycles. As `inseq_timeout` grows, JUGGLER is able to batch packets spanning multiple polling cycles, reaching its maximum batching efficiency at about 52μs.

Out of order timeout: Since batching is done only for packets that are in-sequence, `ofo_timeout` ensures timely packet delivery in case we do not have in-sequence packets due to packet loss. This ensures quick loss recovery by the higher level TCP transport. Intuitively, `ofo_timeout` should be set to the largest expected delay for out of order packets. For example, for multi-path load balancing, `ofo_timeout` should be set to the maximum difference in packet delays across various network paths.

We use the same testbed as in Figure 11 to understand the trade-offs for `ofo_timeout`. Figure 13 shows the results of an experiment where we run a single TCP flow between the two hosts and measure the throughput of the flow for various `ofo_timeout` values. We run the same experiment with different amounts of delay for out of order packets—τ, set to 250μs, 500μs and 750μs—to see how `ofo_timeout` should be configured. The results show that with more severe reordering, it takes a larger `ofo_timeout` to keep the flow at line rate. The minimum `ofo_timeout` needed is roughly $\tau - \tau_0$, where τ_0 is the interrupt coalescing period [14] (125μs in our testbed). Interrupt coalescing acts as an additional reordering buffer layer before JUGGLER, so

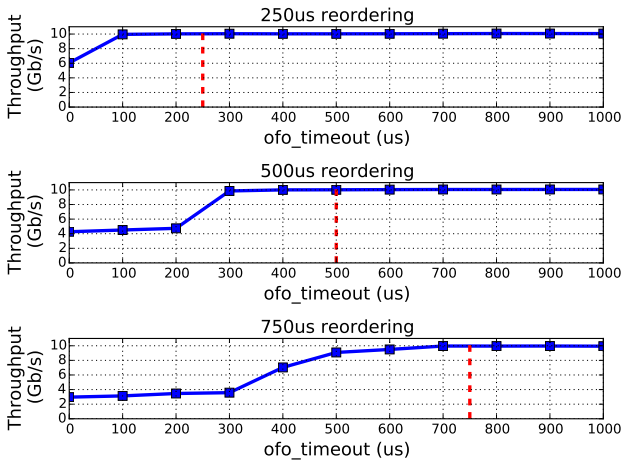

Figure 13. Single flow throughout versus `ofo_timeout`. The amount of reordering seen by the receiver is $250\mu s$, $500\mu s$ and $750\mu s$. In all cases the value of `ofo_timeout` needs to be at least comparable to the amount of reordering to prevent throughput loss.

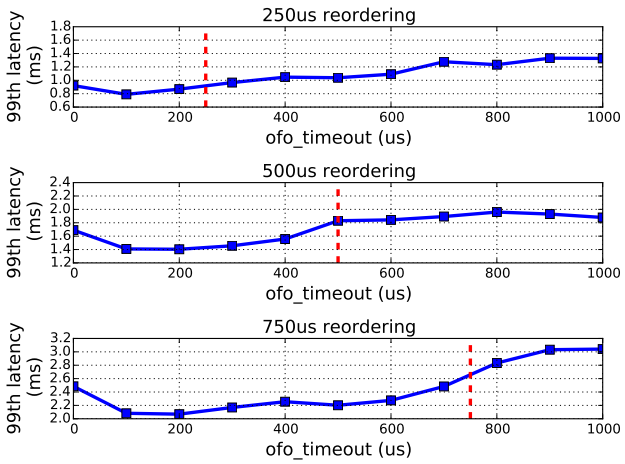

Figure 14. Small RPC tail latency versus `ofo_timeout`. The amount of reordering seen by the receiver is $250\mu s$, $500\mu s$ and $750\mu s$. Note that the plots use different ranges for the y-axis. In all cases the value of `ofo_timeout` should not be configured larger than the amount of reordering in order to avoid high tail latency.

its duration should be subtracted from τ when configuring `ofo_timeout`.

Figure 14 shows the impact `ofo_timeout` has on the latency when packets are being dropped. In this experiment the server sends 10KB RPC messages to the client through the NetFPGA switch. The client drops 0.1% of the packets uniformly at random before they enter JUGGLER. The 99th percentile RPC completion time is measured for various `ofo_timeout` values. Still, we configure the NetFPGA switch to have τ equal to $250\mu s$, $500\mu s$ and $750\mu s$ to see how different amounts of reordering affect the optimal value of `ofo_timeout`. The results show that the tail RPC completion time stays flat when `ofo_timeout` is small, and starts to grow rapidly as soon as `ofo_timeout` becomes larger than $\tau - \tau_0$. Combining with the first experiment, we can conclude that `ofo_timeout` should be set to $\tau - \tau_0$.

Note that we only simulated packet drops in the latency experiments but not the throughput ones. The reason is that the throughput of a flow if less sensitive to a large `ofo_timeout`, in presence of packet losses, compared to its latency. In a separate experiment, at a 0.1% packet loss rate, we see the flow loses throughput only when `ofo_timeout` is larger than 100 *milliseconds*.

Summary: To summarize, we set `inseq_timeout` to a value that gives JUGGLER the maximum batching size with line rate traffic, and set `ofo_timeout` to the expected value of difference in delays across network paths. Generally, it is better to slightly over-estimate `ofo_timeout` since packet loss is rare in datacenters [11].

5.2.2 Size of GRO table

We now look at the size of `gro_table` required to make JUGGLER's eviction policy work without performance degradation. The size of `gro_table` grows linearly with the number of flows tracked by JUGGLER. We discussed in §4.3 that a small limit of a few 10s of entries in this table is sufficient for JUGGLER to start evicting inactive flows (§4.2.4) despite a high flow concurrency at a server, because the length of the active list and the loss recovery list is small. We now experimentally verify that this is indeed the case.

Length of active list: We use the same experiment setup as in Figure 11 to show how the number of concurrent flows and the amount of reordering in the network affects the number of active flows that JUGGLER needs to track. Specifically, we fix reordering to $250\mu s$, $500\mu s$, $750\mu s$ and $1ms$ respectively, sweep the number of concurrent flows from 64 to 1024, and sample the number of active flows in JUGGLER. In all experiments, the sender sends 10Gb/s of total traffic into 4 RX queues of the receiver. Figure 15 shows that the number of active flows grows slowly with the number of concurrent flows and the amount of reordering. The number of active flows start to drop after there are more than 256 concurrent flows because low throughput flows tend to send single-MTU TSOs which are not affected by reordering. In the worst case, JUGGLER needs to track less than 35 active flows.

The previous experiment considered an artificial reordering scenario. To understand how the number of active flows behaves in a more realistic reordering scenario, we reran the experiment described in §5.1.1. On the two-stage Clos topol-

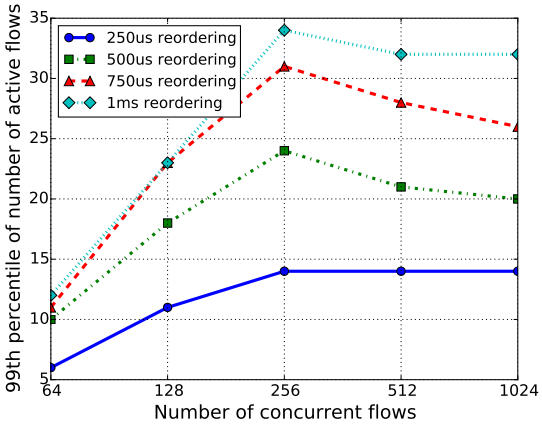

Figure 15. 99th percentile of the number of active flows in JUGGLER. It initially grows with the number of concurrent flows then starts to drop after the latter exceeds 256. The number of active flows also grows with reordering, and the growth slows down when reordering is high.

ogy, 32 senders send 256 flows at 20Gb/s in total into a single NIC receive queue. We generate background traffic such that the total load on each ToR switch is 50%, and use per-packet load balancing at the ToR uplinks to cause reordering. We measured the number of active flows by sampling the size of the active list at the receiver's JUGGLER module every 10 milliseconds; Figure 16 (a) shows the histogram. The average length is less than 1, and 99% of the time it is less than 5. Thus we see even fewer active flows than the last experiment because the reordering caused by real world queueing delay is on the order of 10s of microseconds, which is much smaller than what we experimented with in the NetFPGA setup. Figure 16 (b) shows what happens with a 10Gb/s network interface in the same setup. With the 10Gb/s port, TSO segments spend 3 times more time on the wire and thus stay in JUGGLER longer, resulting a bigger active list. Nonetheless, 99% of the time, the length of the active list is still less than 6.

Length of loss-recovery list: The experiment in Figure 16 (b) has 256 flows competing for the same 10Gb/s-port receiver so it induces packet losses. The flows that experience packet losses are enqueued to the loss recovery list and are dequeued later after the lost packet is retransmitted. The statistics show that on average only 4 flows are enqueued into the loss recovery list per second and the list turns out to be almost always empty, confirming the analysis in §4.3.

Summary: The above experiments show that the active and loss recovery lists are typically small enough to fit within a small 8 entry `gro_table` in applications like per packet load balancing. Even if the application requires JUGGLER to handle up to $1ms$ of reordering, a 64 entry `gro_table` is adequate. Since JUGGLER operates independently on a per-receive queue basis, the memory requirements scale linearly with the number of receive queues.

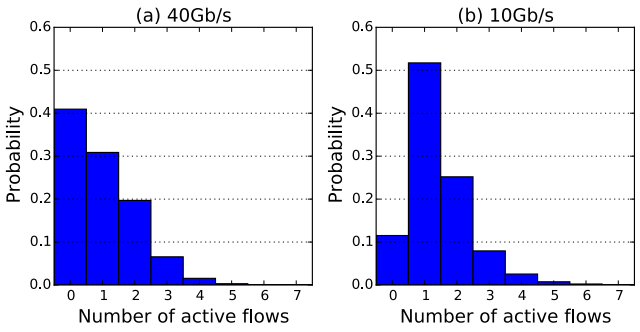

Figure 16. Statistics of length of active list. This length is typically very small because a flow is in the active list only when a TSO burst is accumulating in JUGGLER. A flow spends most of its time in the inactive list.

5.3 Case studies of designs enabled by JUGGLER

As we discussed in §2, JUGGLER opens up new design points to meet the demands of datacenter applications. Below, we show experimental evidence for two examples: guaranteeing network bandwidth to a set of flows and fine-grained network load balancing.

5.3.1 Bandwidth enforcement by dynamic priority adjustment

In §2.1, we discussed how JUGGLER can guarantee some minimum bandwidth to flows using dynamic packet prioritization. Recall that we have a sender module to control the probability that the flow's bytes are transmitted at high priority. This probability is adapted according to the target and achieved throughput (measured by the sender for every ACK received) normalized to the line rate. The gain factor, α in Eq. (1), is set to 0.1 in this experiment.

Figure 17 shows the experiment setup: a 40Gb/s topology with two TORs, where in the switch interconnecting the two ToRs there are two queues with one having strict priority over the other. Two senders are placed under one ToR with their receivers on the other. Sender 1 sends one (target) TCP flow with a guaranteed bandwidth B ($< 40\text{Gb/s}$) to receiver 1. Sender 2 sends 7 antagonist TCP flows, with no bandwidth constraints or guarantees, to receiver 2. All flows start at lowest priority.

Results. Figure 18 shows the throughput of the target flow with a JUGGLER-enabled kernel and the vanilla kernel, as we vary B. We measure the average and the standard deviation of the achieved throughput numbers from 30 runs. As expected, the throughput of the target flow closely tracks B until it reaches 25Gb/s, at which point we hit the CPU limit on a single core.[5] Without JUGGLER, the target flow's

[5] This is because NICs today hash one flow to one receive queue.

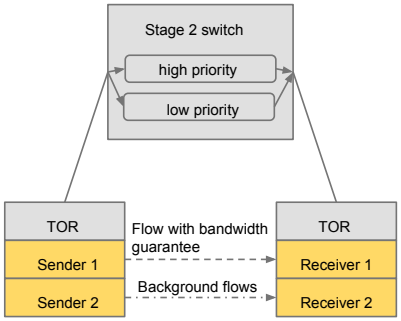

Figure 17. Bandwidth enforcement experiment setup. The stage two switch has two output queues with strict priority. We try to guarantee the bandwidth of a target flow despite the background flows by putting a portion of the packets of the target flow into the high priority queue.

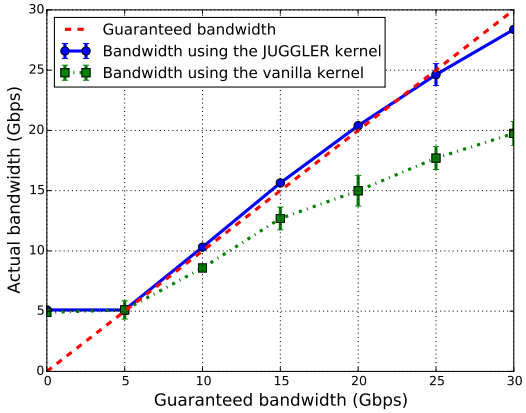

Figure 18. Dynamic packet scheduling to guarantee network bandwidth: JUGGLER's actual achieved bandwidth (y-axis) follows the guaranteed bandwidth closely while the vanilla kernel's TCP deviates from the ideal guaranteed bandwidth. Note that the bandwidth of the target flow won't be smaller than its fair share bandwidth (5Gb/s) even if its guaranteed bandwidth is. This is because the fair share bandwidth is achieved when all packets of the target flow go through the low priority queue.

bandwidth is far lower than the guarantee and variable, because dynamic packet prioritization induces packet reordering which the vanilla kernel cannot handle.

5.3.2 Fine-grained load balancing

Figure 19 shows the experiment setup. There are 8 servers under ToR A sending to 8 clients under ToR B using 40Gb/s NICs. Half of the clients and servers are dedicated for sending and receiving 1MB remote procedure calls (RPCs) and the other half send and receive 150 Byte RPCs, in an all-to-all fashion. The senders generate RPCs in an open-loop fash-

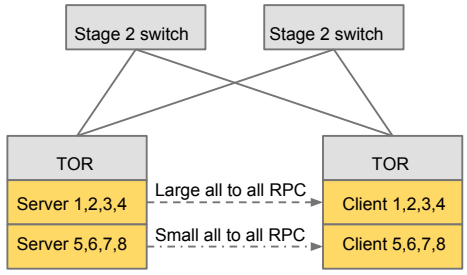

Figure 19. Load balancing experiment setup. We have 4 pairs of server running large all to all RPC and another 4 pairs running small all to all RPC in a 40Gb/s Clos network. The ToR switches are configured to do per-flow, per-packet level load balancing to compare their performance. We implement per-TSO load balancing at end-hosts in a manner similar to Presto [24].

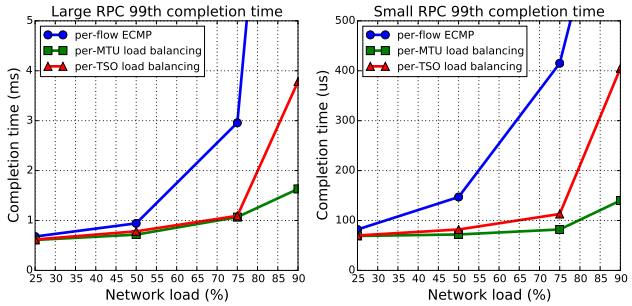

Figure 20. Per-packet load balancing improves tail latency significantly at high network load.

ion, with inter-arrival times drawn from an exponential distribution (Poisson arrivals), such that the total load on the uplinks is 25%, 50%, 75% and 90% of capacity 80Gb/s. Most load is due to large 1MB RPCs; the 150 byte small RPCs generate only 100Mb/s traffic at each server. The traffic generator randomly multiplexes RPCs across 8 long-lived TCP sessions between every client-server pair.

Results. We compare the resulting RPC completion times and the switch buffer occupancy with packet-level load balancing to per-TSO and per-flow (ECMP) load balancing in Figure 20. The results show at least 2× better tail latency for short RPCs with per-packet load balancing compared to per-flow load balancing after network utilization crosses 50%. This is consistent with the observed buffer build-up measured at the ToR switches (figure not shown).

The small RPC tail latency of per-packet load balancing is 30μs less than that of per-TSO load balancing at 75% load, and 250μs less at 90% load. Such large difference in tail latency is crucial for latency-sensitive applications like RAMCloud [40], which can fetch small key-value pairs within 5μs over an uncongested network. This exper-

iment shows that per-packet load balancing coupled with JUGGLER's reorder resiliency is an effective mechanism for achieving low tail latency even at high utilization.

6. Related Work

Combating packet reordering: Most of the previous work has focused on fixing TCP protocol problems [12, 19, 36, 50]. These schemes mitigate TCP's overreaction to out of order packets, but cannot address CPU overhead due to the collapse of optimizations such as GRO. Presto [24] is an end-host based load balancing mechanism that also adds an out of order buffer to GRO to mitigate the CPU overhead of reordering. However, Presto requires the sender to send fixed size chunks and only handles reordering at the TSO level. Also, Presto maintains state for all established connections, which may suffer from performance issues and is vulnerable to memory resource exhaustion attacks. JUGGLER resolves both TCP and CPU overhead problems and handles severe packet level reordering. JUGGLER is practical and safe to deploy as it only keeps state for a very small number of recent flows by making smart eviction decisions.

Network stack optimizations: The CPU overhead of network processing has been shown to be dominated by per-packet overhead for real-world workloads [33]. This has led to receive offload techniques that merge packets in hardware [21] and software [37] to significantly reduce the receiver's CPU usage. JUGGLER preserves software receive offload in presence of reordering to sustain high throughput.

7. Conclusion

JUGGLER is a practical reordering resilient network stack for datacenters. Our work was motivated by the desire to enable future datacenter network systems that can transmit packets on any path and at any level of priority. We demonstrated two designs enabled by this flexibility: per packet load balancing and a novel mechanism for giving bandwidth guarantees. We believe that many other systems can also benefit from flexible control over packet routing and scheduling.

JUGGLER adds a modest amount of functionality to the GRO layer to deal with packet reordering in a best effort fashion. JUGGLER leverages the small network latency and high traffic burstiness in datacenters to provide reordering resiliency with a simple and practical design. It aggressively limits the amount of per-flow state maintained and intelligently selects which flows to track. Extensive experiments with 10Gb/s and 40Gb/s hardware showed that JUGGLER is lightweight and efficient, and can handle even severe packet reordering.

Acknowledgments

We thank our shepherd, Robert Soulé, and the anonymous EuroSys reviewers for their thoughtful feedback that helped improve the presentation of the paper.

References

[1] Arista 7500 switch architecture. `https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7500E_Switch_Architecture.pdf`. [Online; accessed 22-October-2015].

[2] conntrack-tools: Netfilter's connection tracking userspace tools. `https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7500E_Switch_Architecture.pdf`. [Online; accessed 22-October-2015].

[3] Overview of single root i/o virtualization (sr-iov). `https://msdn.microsoft.com/en-us/library/windows/hardware/hh440148(v=vs.85).aspx`. [Online; accessed 22-October-2015].

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). *SIGCOMM*, 2011.

[7] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. *SIGCOMM*, 2013.

[8] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514. ACM, 2014.

[9] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and W. Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 25. ACM, 2014.

[10] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[12] E. Blanton and M. Allman. On making tcp more robust to packet reordering. *ACM SIGCOMM Computer Communication Review*, 32(1):20–30, 2002.

[13] M. Blott, J. Ellithorpe, N. McKeown, K. Vissers, and H. Zeng. Fpga research design platform fuels network advances. *Xilinx Xcell Journal*, 4(73):24–29, 2010.

[14] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed tcp. *Communications Magazine, IEEE*, 39(4):68–74, 2001.

[15] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder. Offload of tcp segmentation to a smart adapter, Aug. 10 1999. US Patent 5,937,169.

[16] conntrack. nf_conntrack: table full, dropping packet. `http://security.stackexchange.com/questions/`

43205/nf-conntrack-table-full-dropping-packet.

[17] J. Corbet. Jls2009: Generic receive offload. https://lwn.net/Articles/358910/, 2009. [Online; accessed 22-October-2015].

[18] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2130–2138. IEEE, 2013.

[19] S. Floyd, J. Mahdavi, M. Podolsky, and M. Mathis. An extension to the selective acknowledgement (sack) option for tcp. 2000.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

[21] L. Grossman. Large receive offload implementation in neterion 10gbe ethernet driver. In *Linux Symposium*, page 195, 2005.

[22] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues dont matter when you can jump them! In *Proc. NSDI*, 2015.

[23] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152. ACM, 2015.

[24] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 465–478. ACM, 2015.

[25] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[26] C. E. Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[27] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[28] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message completion time in the cloud. Technical report, Tech. rep., Microsoft Research, 2013. MSR-TR-2013-95, 2013.

[29] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. *REM*, 1005(A1):A2, 2013.

[30] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 149–160. ACM, 2014.

[31] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, 2007.

[32] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet trains: A study of nic burst behavior at microsecond timescales. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 133–138. ACM, 2013.

[33] J. Kay and J. Pasquale. Profiling and reducing processing overheads in tcp/ip. *IEEE/ACM Transactions on Networking (TON)*, 4(6):817–828, 1996.

[34] J. Lee, M. Lee, L. Popa, Y. Turner, S. Banerjee, P. Sharma, and B. Stephenson. Cloudmirror: Application-aware bandwidth reservations in the cloud. *USENIX HotCloud*, 20, 2013.

[35] Y. J. Liu, P. X. Gao, B. Wong, and S. Keshav. Quartz: a new design element for low-latency dcns. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 283–294. ACM, 2014.

[36] R. Ludwig and R. H. Katz. The eifel algorithm: making tcp robust against spurious retransmissions. *ACM SIGCOMM Computer Communication Review*, 30(1):30–36, 2000.

[37] A. Menon and W. Zwaenepoel. Optimizing tcp receive performance. In *USENIX Annual Technical Conference*, pages 85–98. Boston, MA, 2008.

[38] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. How to improve your network performance by asking your provider for worse service. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 25. ACM, 2013.

[39] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 491–502. ACM, 2014.

[40] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[41] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 307–318. ACM, 2014.

[42] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198. ACM, 2012.

[43] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. Elasticswitch: practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 351–362. ACM, 2013.

[44] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[45] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137. ACM, 2015.

[46] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, optimal flow routing in datacenters via local link balancing. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 151–162. ACM, 2013.

[47] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: performance isolation for cloud datacenter networks. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 1–1. USENIX Association, 2010.

[48] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in googles datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 183–197. ACM, 2015.

[49] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4): 139–150, 2012.

[50] M. Zhang, B. Karp, S. Floyd, and L. Peterson. Rr-tcp: a reordering-robust tcp with dsack. In *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pages 95–106. IEEE, 2003.