Generic External Memory for Switch Data Planes

Daehyeok Kim[1], Yibo Zhu[2], Changhoon Kim[3], Jeongkeun Lee[3], Srinivasan Seshan[1]
[1]Carnegie Mellon University, [2]Microsoft Research, [3]Barefoot Networks

Abstract

Network switches are an attractive vantage point to serve various network applications and functions such as load balancing and virtual switching because of their in-network location and high packet processing rate. Recent advances in programmable switch ASICs open more opportunities for offloading various functionality to switches. However, the limited memory capacity on switches has been a major challenge that such applications struggle to deal with. In this paper, we envision that by enabling network switches to access remote memory purely from data planes, the performance of a wide range of applications can be improved. We design three remote memory primitives, leveraging RDMA operations, and show the feasibility of accessing remote memory from switches using our prototype implementation.

1 Introduction

Modern data center applications such as key-value stores and network functions such as load balancing demand high I/O rates or high packet processing rates. Traditionally, many of these applications are implemented in software that runs on general purpose CPUs. However, technology trends make this solution less desirable – CPU performance improvements (*i.e.,* Moore's Law) have slowed and application demands continue to increase at a staggering rate.

The research community has turned to specialized processors, specifically the high speed packet processing ASICs at the core of modern Ethernet switches, to help meet the processing demands of these applications [19, 23, 26, 29]. Given similar energy and monetary budget as a commodity server, switches offer 1000× higher packet processing rates and processing latency of sub-microseconds. Recent advances in programmable P4 switches [3] enable more flexible application logic on these processors.

While programmable switches provide some important performance benefits, they also create unique challenges, one of the most significant being limited memory space. Data center switches require memory bandwidths of 32×100 *Gbps* (32 100 Gbps ports) or more. As a result, they rely on only using relatively expensive but fast SRAM or TCAM. Because these are located on the chip, they are limited to tens of MB, which already contributes to a significant fraction of switch hardware costs.

Unfortunately, *all* of the above on-switch applications have trade-offs between performance and memory usage. For example, memory size directly determines the cache hit rate of in-network caches [19], and the accuracy of sketches [23]. Applications such as load balancing [26] and monitoring [29] have to fall back to slower, CPU-based workarounds when memory is exhausted. Last but not the least, even the most basic function of switches – packet forwarding – can suffer from limited memory space. These issues are further exacerbated when these applications run on the same switch and must share memory with each other and basic forwarding.

One solution to this issue is designing a switching ASIC that has internal custom logic and wires to access external DRAM [4] on the switch. Unfortunately, this proves to be very *expensive* for several reasons. To provide the bandwidth needed for switching, a ASIC chip would need numerous parallel DRAM modules. This combined with the necessary DRAM controller and wiring within the chip adds significant complexity and cost to the switch. This approach also leads to an *inflexible* usage of the DRAM because the memory can be used either for packet buffers or one or a few select look-up tables depending on how the DRAM is physically connected to the associated modules within the ASIC chip. Moreover, the chip has a fixed amount of external-memory-capacity that is determined at the chip design and manufacturing time, resulting in *limited scalability*. The end result is that on-switch external DRAM is not popularly used by vendors in today's single-chip switches.[1]

In this paper, we explore the possibility of repurposing the affordable DRAM installed on data center servers to expand the memory available to a switch's *data plane*. To achieve this goal, instead of building another switching ASIC that has a custom external-DRAM-access capability, we simply reuse an existing programmable switching ASIC built only with on-chip memory. More specifically, we trade the data plane bandwidth of such a chip for the bandwidth needed to access the external DRAM. For example, a ToR switch may reserve some DRAM space on the servers under it, and turn them into a remote memory pool accessible from the ToR switch directly through the Ethernet links connecting between the servers and the ToR.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA

ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286063

[1]Some high-end multi-chip switches have additional DRAM connected to the data plane. Such switches are much more expensive and thus uncommon in data centers [32].

Current commodity hardware and network deployments are well-suited for enabling this approach. First, since half or more than half of the ToR ports are connected to servers, and data center traffic is sparse [35], there is usually sufficient amount of spare bandwidth between the ToR and this potential memory pool. Second, RDMA-capable NICs (RNICs) can expose the memory access at line rate using the RDMA over Converged Ethernet (RoCE) [16, 17] protocol, without *any* CPU involvement. Third, emerging programmable switches can implement a subset of the RoCE protocol and maintain the needed connection state to communicate with RNICs.

We particularly consider three use cases for remote switch memory. First, we describe remote packet buffer that aims to increase the switch buffer size from O(10 MB) to O(10 GB), or by 1000×. With this, we may be able to largely reduce, if not to eliminate, the microburst incast drops at last hop. Second, we consider a remote lookup table that increases the exact-matching table size by 1000× or more. This may help applications that demand large tables avoid the CPU-based slow path. Finally, we examine remote state stores that improves the performance of data plane monitoring. We will elaborate these example scenarios in §2 and preliminary design in §4.

To verify the feasibility of using remote memory, we implement and evaluate three basic primitives designed for the above use cases on commodity RNICs and a P4 switch. In §5, we show that the remote memory on a server offers up to 95.6% RNIC bandwidth with 1-2 μs extra latency, with absolutely 0% CPU overhead. Multiple leading cloud providers have widely deployed RNICs [28, 36] and are also gradually on-boarding P4 switches. In such environments, there is no additional infrastructure costs of deploying our design, except for the reserved DRAM on servers.

Fundamentally, this paper introduces the notion of "memory hierarchy" to programmable switches, for which such concept was absent. Current P4 switches just have an on-chip memory. Once the switch faces workloads that need larger memory than the cache, it cannot gracefully handle it with low cost. This memory hierarchy we introduce draws parallels to what DRAM did for CPU and CPU cache. We believe that this work can inspire and ease the design of future in-network applications that require large memory on the data plane. Future steps include concretizing the systems described in §2, and addressing the challenges explained in §7.

2 Motivating Examples

In this section, we examine three representative applications that are limited by memory availability on switches. We use these applications to motivate the need for remote memory primitives. The below provides a high-level description of how the primitives can benefit each use case and we leave the design details to the later sections and future work.

2.1 Mitigating Packet Losses

The problem: Congestion drops due to bursty incasts are very common in data center networks. For example, consider the typical last-hop congestion in Figure 1a. Suppose all links are 40 Gbps, the ToR switch has 12 MB packet buffer, and 50 MB traffic comes from eight uplinks at line rate and goes towards a single receiving server. It will take at least $50\ MB/40\ Gbps = 10\ ms$ to receive all the traffic, however the 12 MB packet buffer will be filled within $12\ MB/(8-1)/40\ Gbps \approx 0.34\ ms$ and start dropping packets! This is well explained by past literature [8, 35].

To avoid such packet drops, Priority Flow Control (PFC) [15] has been proposed. Unfortunately, it leads to other serious problems [36] such as occasional deadlocks. Also, while packet detour [34] can reduce packet drops, it requires switch and end host modifications. Another solution is to increase packet buffer on switches, *e.g.,* to more than 50 MB, so that the whole burst can be absorbed. However, data center operators and switch vendors are reluctant to do that as it significantly increases costs and cannot scale to larger incast.

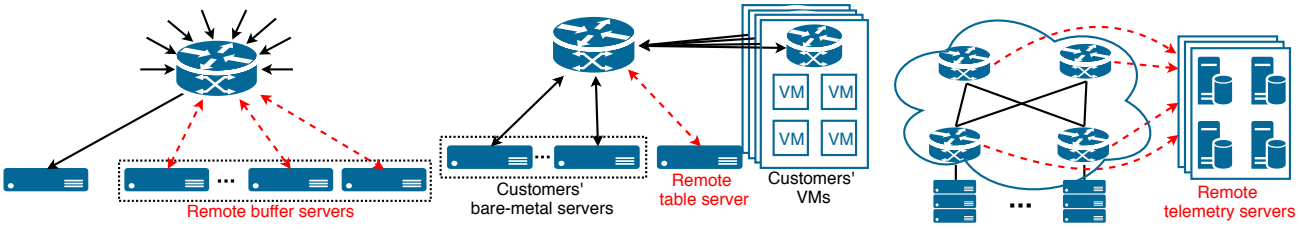
A potential solution: Imagine the ToR switch has a data-plane channel that enables it to *extend its packet buffers* by reading and writing DRAM placed in any servers under the ToR (the dashed box in Figure 1a). When incast occurs and the queue on switch starts to build up, the switch can start writing *all* the following packets bound to the congested queue into a remote buffer located in one or multiple servers (the red dashed lines in the figure). Then, once the queue on the switch is fully drained, the switch can start to read the packets back from the remote buffer and continue forwarding them.

This packet buffer extension would significantly reduce, if not eliminate, last-hop congestion drops, because in a common Clos topology, the total uplink bandwidth is equal to or smaller than the downlink bandwidth. Since DRAM is much more affordable than on-chip buffer memory, operators can reserve O(1 GB) memory on each server for absorbing the bursts. Before that >10 GB remote memory is all filled, any bursty incast conditions should have passed, or (in the case of persistent congestion) end-to-end congestion control based on ECN [36] or delay [28] should have slowed traffic. In short, this solution enables a "lossless" last-hop ToR switch, without the caveats of PFC.

2.2 Extending Lookup Tables

The problem: Some applications and network functions require large lookup tables for handling different flows. We use a concrete example in Microsoft Azure to illustrate this. Azure offers a bare-metal hosting service, where customers provide their "blackbox" servers, usually specialized database systems, or security-related middleboxes. The cloud provider hosts them in data centers, and must provide connectivity to the VMs of the same customer, as shown in Figure 1b. This means that the virtual IPs of bare-metal boxes and VMs should be translated to physical IPs that can be forwarded in the physical network.

Since the cloud provider cannot easily install specialized hardware (e.g., smartNICs [14]) or software (e.g., virtual switch [30]) on these blackboxes, such translation has to be done outside the blackboxes. One option is pairing each of the blackboxes with a dedicated server that runs smartNIC



(a) Extending packet buffers for mitigating packet losses due to incast. **(b) Extending lookup tables for bare-metal hosting.** **(c) Extending state stores for network telemetry.**

Figure 1: Example use cases of remote memory for network switches. The solid black and dashed red lines indicate the data path for transmitting original packets and accessing remote memory, respectively.

or virtual switch. The cost of such an approach is very high. Another option is moving a virtual switch functionality into the ToR switch. However, the most modern merchant silicon switches are equipped with tens of MBs of SRAM for look-up tables. Based on our experience in production, this is at least one order of magnitude less than a typical virtual switch consumes for virtual network address mappings.

A potential solution: A pure data-plane channel to *extend the lookup table* to remote memory may be a good answer to this scenario. We maintain the complete virtual-to-physical address mapping table on servers in a sharded fashion and let the switch fetch the entries on-demand. The local memory on switches serves as high-speed cache to reduce the remote memory access. However, even if the traffic pattern leads to frequent cache misses and remote fetching, there is no CPU overhead or software latency.

Although we use a virtual switch as an example, this idea can benefit many other on-switch applications including key-value stores (*e.g.,* NetCache [19]) or load balancers (*e.g.,* SilkRoad [26]). These applications typically fall back to the software (*i.e.,* either on server or switch's CPU) whenever the memory in the data plane is insufficient for the size of their working set. With the remote lookup table, however, such slow-path forwarding through the software can be eliminated or minimized. Of course, this will require careful co-design of data structure in remote memory and the switch data plane, as we will explain in §7.

2.3 Extending State Store for Telemetry

The problem: Switches have unique advantages in network telemetry or monitoring. First, they have access to state information available only in the network, such as the true progression of queue length, the exact extent of congestion, and exact table entries that are matched by each packet. Second, the switching ASIC can perform monitoring tasks, *e.g.,* per packet counting, at the rate of billions of packets per second. However, the limited memory space either directly determines the performance (like sketch systems [23]), or again forces the monitoring system to fall back to software frequently [29].

A potential solution: Programmable switches can generate new packets based on the content of original packets. The commodity RNICs support not only reading/writing memory, but also atomic Fetch-and-Add operation. Combining these two, it is possible to *extend the state-store* for telemetry systems, as shown in Figure 1c. For example, the switch can extract fields from original packets and perform RDMA WRITE into certain remote memory address. This eliminates the CPU cycles required for capturing and parsing packets in previous systems. In addition, updating counters, as one of the most important telemetry primitives, can be implemented using RDMA atomic Fetch-and-Add. The number of counters can increase by $10^3\times$ (*e.g.,* 100 GB DRAM vs. less than 100 MB switch SRAM), without any CPU overhead.

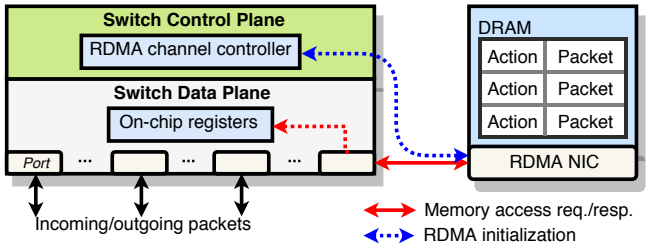


Figure 2: Overview of the proposed remote memory architecture. Using our primitives, the switch data plane can utilize remote memory region registered to RNICs by servers which are connected to the switch. While this figure illustrates a remote lookup table as an example, different types of data structures are used for different primitives (*e.g.,* ring buffer for remote packet buffer).

2.4 Summary

The three examples above show that a pure data-plane channel between switch and remote memory can benefit a wide range of network functions. Fortunately, as we shall see soon in §4, commodity hardware is ready for this architecture – P4 switches can craft RoCE protocol packets that communicates with RNICs using one-sided RDMA operations.

3 Overview

In this section, we overview the design of the three remote memory primitives to support the use cases described in §2: (1) Packet buffer (2) Lookup table, and (3) State-store primitive. The general architecture is shown in Figure 2. We design the primitives with the following goals in mind.

Goal #1 – Easy to deploy on commodity hardware: The remote memory primitives can be used in any Ethernet networks that deploy RDMA using the RoCE standard and programmable switches. For now, we assume that the remote memory servers are equipped with RDMA-capable NICs

(RNICs) that are directly connected to the switches.[2] We do not require any hardware modifications on commodity RNICs and switches. We design the primitives to be generic and modular, so that any data plane programs can use the primitives to utilize remote memory while processing packets.

Goal #2 – No CPU involvement: Except for initialization, the primitives do not require any involvement of CPUs, neither remote servers' or switches', while switches access the remote memory. All memory access requests and responses are processed entirely by the RNICs on the servers (the red solid line in Figure 2). The CPUs are used only when initializing and registering memory regions that will be accessed by switches, and establishing an RDMA channel between the RNIC and switches (the blue dashed line in the figure). This property makes the server-side memory-extension architecture low cost (no CPU overhead) and energy efficient.

The key design idea: Our key idea is to create an RDMA channel between switches and remote servers so that the switches can access remote memory via the channel even without having RNICs on the switches and without involving CPUs on the servers.

We realize this idea based on three insights. First, on Ethernet networks with RoCE, RDMA requests and responses are merely regular Ethernet packets with additional headers supporting RDMA operations. This means that we can generate and process RDMA packets via any devices connected to the networks. Second, emerging programmable switching ASICs [3] can manipulate, adding, or removing packet headers at line rate. This enables us to design data plane actions to generate and process RDMA request/response packets with low latency. Third, one-sided RDMA operations such as RDMA WRITE, READ, and atomic Fetch-and-Add are processed entirely by RNICs without the involvement of remote servers' CPUs. By leveraging such operations, we can make the primitives simple and scalable since it does not require any complex server-side components to control the memory access from switches.

As mentioned earlier, we design the primitives as data plane actions so that switch data plane programs can easily adopt the primitives to utilize remote memory. The primitive actions generate corresponding RDMA requests by adding or manipulating RDMA headers on top of original or cloned packets. An RDMA channel controller running on the switch control plane and a server is responsible to allocate memory regions on the server, set up an RDMA channel, and pass the channel information including a remote queue pair number (QPN), a base address of the registered memory region, and a remote access key (Rkey) for the region to the data plane via the switch control plane APIs.

4 Proof-of-Concept Design

We now explain in detail the current design of the primitives.

Packet buffer primitive: Using the packet buffer primitive, the switch can store a packet to the remote packet buffer or load it back. Storing can be triggered by certain states in the data plane (*e.g.,* the egress queue length exceeds some threshold or drains). The switch generates a RDMA WRITE request containing the original packet to the remote RNIC. To create such RDMA operation requests, the primitive adds RoCE headers on top of a original packet and fills the header fields with necessary information, such as an operation type (*i.e.,* WRITE or READ), a remote memory address, a QPN, an Rkey to access remote packet buffer. We design the remote packet buffer as a ring buffer and make the primitive maintain the write and read offset pointer to the buffer so that it can store and load packets from the right position of the buffer. In the current prototype, we allocate the buffer to store full-sized Ethernet frame in each entry.

Similar to storing, packet loading starts when a pre-defined event (*e.g.,* the egress queue length becomes some threshold) occurs in the data plane. Specifically, the switch generates a READ request and the RNIC immediately replies a READ response. The switch must parse the READ response, decapsulate the RoCE headers, and passes the original packet to the egress pipeline. In the current prototype, each load operation fetches a single entire entry regardless of the original packet size. The received READ response packet is also used as a trigger for generating another READ request to fetch the next packet in the remote buffer if exists. Also, to avoid packet reordering, the primitive makes sure that until all packets in remote buffer are read, the following new packets must also be written to the remote buffer and read out in order.

As mentioned above, packet storing and loading starts or ends based on a pre-defined condition (*e.g.,* the current egress queue length). Depending on the condition, end-to-end performance may be affected (*e.g.,* latency increases due to a packet loaded too late). Finding a right condition to start loading packets from remote buffer is our ongoing work.

Lookup table primitive: This primitive enables the switch to look up the match-action table on remote memory for a given packet. For example, the primitive action can be triggered when a look-up miss happens on a local match-action table on the switch. In the current design, the action generates corresponding RDMA requests (1) to store the original packet to the table on remote memory and (2) to load an ⟨action, packet⟩ pair from the table. By bouncing the original packet to and from the remote buffer, the switch does not need to store the packet when waiting for the table entry. Upon receiving the response from the RNIC, the switch parses the action and applies it to the packet.

We design the lookup table primitive again using RDMA WRITE and READ. Once the primitive is triggered, it first calculates the target entry index in the remote table, based on a user-defined hash function (*e.g.,* hashing based on the packet's 5-tuple). Combined with entry size and base memory address, which is initialized when setting up the RDMA connection, it gets the target memory address. Then, it creates a RDMA WRITE request based on the address and the packet length and issues it to store the packet to the packet buffer

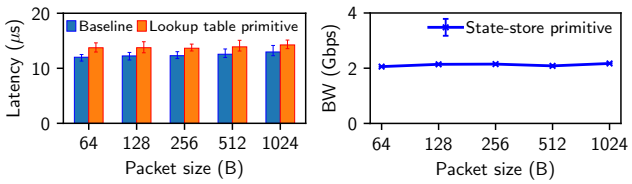
[2] In future work, it is possible to use any remote servers in the same RoCE network after some technical challenges are addressed (§7).

corresponding to the entry. Then it immediately generates a RDMA READ request to fetch the entry ⟨action, packet⟩. Upon receiving a RDMA READ response packet from the RNIC, it extracts the action from the packet and remove RoCE headers. Then, it applies the action to the original packet. Finally, the switch can (optionally) cache the table entry in local SRAM, so that the same action can be applied to subsequent packets of the same flow without accessing the remote table. We verify that this can be done purely in the data plane.

State-store primitive: The state-store primitive enables the switch to maintain various stateful objects such as counters and meters on remote memory. We can use this primitive for various purposes which require to store a large amount of stateful objects, such as running multiple sketching algorithms [23] or network measurements [29]. For instance, one can easily implement sketching algorithm such as Count Sketch [11] using the primitive even for a large number of flows. If a packet matches to a certain sampling criteria, the primitive action determines the address of the remote sketch counter for the flow and updates the remote counter by issuing an RDMA atomic Fetch-and-Add request to the RNIC. Network operators can run any estimation algorithms (*e.g.,* heavy-hitter detection) on the remote counter.

RDMA WRITE and atomic Fetch-and-Add operation can be used to implement the state-store primitive. Here, we just describe a state-store that just counts packets on per-flow basis, which only requires Fetch-and-Add operation. Specifically, while an original packet is processed through the regular pipeline, the primitive clones the original packet and truncates the entire headers and payload of cloned packet to generate a packet for an RDMA Fetch-and-Add request. It adds RoCE headers to the cloned packet and fills the header fields with the information corresponding to a Fetch-and-Add operation, including the memory address of the remote store entry for the flow. Since there is a maximum limit of outstanding RDMA atomic requests that an RNIC can handle, we design this primitive to maintain the number of outstanding requests and issue a Fetch-and-Add request only if there is a room to issue more requests. Otherwise, it accumulates the counter value and uses the accumulated value when it can issue a new operation.

Overhead: Though our design does not have any CPU overhead, it does consume extra bandwidth. In an RDMA packet, RoCEv2 protocol adds 40 bytes (52 bytes in the case of RoCEv1) of headers containing routing and transport information in addition to an RDMA operation-specific header of 16 (WRITE/READ) or 28 bytes (Fetch-and-Add). In the case of WRITE for storing packets, the bytes of original packet is directly appended after those headers. These extra bytes will consume link bandwidth between the switch and the RNIC. Also, whenever the switch accesses remote memory, end-to-end latency may increase and the throughput may decrease slightly. We will demonstrate the feasibility of our approach by measuring the end-to-end performance and overheads in §5.



(a) Latency overhead of lookup table primitive. **(b) Bandwidth overhead of state-store primitive.**

Figure 3: Overhead of accessing remote memory.

5 Preliminary Evaluation

In this section, our focus is to demonstrate the feasibility of accessing remote memory from the switch, and to show that it achieves desirable performance using microbenchmarks.

Prototype implementation: We have implemented a prototype of the remote memory primitives and testing data plane programs in approximately 1400 lines of P4 [9] code. We have compiled them to Barefoot Tofino ASIC [3] with Barefoot Capilano SDE [2]. Also, we implement the control plane program in about 1000 lines of C and Python code. It initializes a RDMA channel and allocate/register memory regions on the server's DRAM.

Testbed setup: Our testbed consists of a Barefoot Wedge-100B programmable switch and three servers equipped with a Intel Xeon E5-2609 CPU, 64 GB RAM, and a 40 Gbps Mellanox CX-3 Pro NIC. The servers run Ubuntu 16.04 with the kernel version 4.4.0. All three servers are directly connected to the switch. We use two servers as communication end-hosts and the remaining one as a remote memory server.

Packet buffer primitive: To verify the feasibility of using the remote packet buffer, we evaluate how much traffic can be handled by the primitive using a single remote buffer server. We wrote a P4 program that first stores all incoming packets to the remote buffer, and later loads and forwards them to the destination port. For microbenchmark purpose, we manually start the two steps respectively.

We measure the maximum traffic rate that the remote buffer can handle without losing any packets. We use `raw_ethernet_bw` in the Mellanox Perftest suite [7] to generate packets at configurable data rate, up to 40 Gbps line rate. The results show that the primitive can store 1500B MTU sized packets arriving at the rate of 34.1 Gbps to the remote buffer and forward the packets to their original destination at the rate of 37.4 Gbps without packet loss. Beyond these rates, though the peak throughput can be higher, we start to observe that RDMA requests were occasionally dropped at the NIC. As a baseline, we test native server-to-server RDMA WRITE and READ throughput. The baseline is only 4.4% faster.

Lookup table primitive: We evaluate the lookup table primitive by analyzing its impact on the end-to-end latency. Using the primitive, we wrote a P4 program that fetches an action entry from the remote table for every incoming packet, applies the action to each packet, and forward to the destination port. As an example, we defined a custom action that modifies the value of the DSCP field of IPv4 header to a specific value stored in the remote table entries. We use `NPtcp` [6] to

measure the end-to-end latency with different packet sizes (64-1 KB). Figure 3a shows the median end-to-end latency when packets traverse the baseline and our prototype. Compared to the baseline, a simple P4 implementation of L2 switch without doing anything special, it only adds 1-2 μs latency on average.

State-store primitive: To evaluate the state-store primitive, we wrote a P4 program that counts the number of packets transmitted between two end hosts using a counter in the remote memory. We measure a link bandwidth consumed by RDMA atomic Fetch-and-Add requests/responses to update the remote counter as well as verify the accuracy of the value in the counter. We ran `raw_ethernet_bw` to generate traffic with different packet sizes. Figure 3b shows that on average, memory access requests generated by the primitive consume 2.1 Gbps of link bandwidth between the switch and the RNIC to update the remote counter while the updated value is 100% accurate. This overhead is capped by RNIC Fetch-and-Add throughput. As explained in §4, the switch keeps track of RNIC progress, and it aggregates the counts locally until it finds that the RNIC can catch up. We also verify that there is no end-to-end throughput degradation compared with the baseline, a simple L2 switch.

All the primitives have zero CPU overhead.

6 Related Work

Accessing remote memory with RDMA: As RDMA enables ultra low-latency remote memory access with minimal CPU consumption, previous works have proposed various applications that utilize remote memory in a network. These include key-value stores [12, 20, 27], distributed shared-memory [12], transactional systems [10, 13, 21], and distributed NVM systems [24, 31]. While all these applications adopt RDMA to let a server access remote memory on another server, our work demonstrates a novel use of RDMA, which allows network switches to leverage remote memory.

Applications on programmable switch data planes: With recent advances in programmable switch data plane [3, 5], studies have shown that various network functions such as load balancers [26] and network telemetry [1, 29], and distributed applications such as key-value stores [18, 19] and sequencers [22] can be implemented on a switch to improve application performance and network efficiency. However, such systems can have limited scalability due to scarce memory resource on the switch. Our primitives can potentially benefit those applications by providing a memory hierarchy with remote memory pool.

Lookup table updates via the control plane: Previous studies on software-defined networking have proposed ways of updating flow-based rules to switches, either proactively or reactively, via the control plane [25]. Some approaches [33] attempt to scale the controller by distributing rules to multiple authority switches. However, all these past approaches still require control plane resources for moving the rules, making them less applicable to data plane programs. We propose more general remote memory access primitives. They can be used for not only extending tables, but also packet buffer and state-store, run purely in data plane and does not require control plane involvement.

7 Discussion and Future Work

We conclude by highlighting a subset of open challenges and future research directions.

Co-design of remote memory data structure and switch data plane: The current design based on commodity switch and RNICs can only support address-based memory access. They do not natively support ternary or exact matching. Thus, we design our prototypes using the most basic data structure like FIFO queues and fixed-size array. It would be interesting to co-design the data structure and switch data plane for supporting ternary matching and other more complicated data layouts in remote memory.

Further improving the telemetry system: We only show a preliminary prototype – per-packet counting in remote memory – as an example telemetry system using the store-state primitive. There is much room to improve. For example, to reduce the bandwidth overhead of Fetch-and-Add packets, we may further combine multiple counter updates into a single operation, at the cost of some delay in updates. Also, designing a general streaming packet trace analysis system with our primitives is another interesting direction.

RDMA packet drops: The RDMA packets between the switch and remote memory servers may get dropped due to congestion or packet corruption. Such packet drops may or may not affect the end-to-end performance and the accuracy of primitive operations. For example, in the packet buffer primitive case, an RDMA packet drop would lead to dropping the original packet. Since Ethernet itself is best-effort, applications and end-to-end protocol should tolerate the packet drops. On the other hand, in the store-state primitive, an RDMA packet drop would affect the accuracy of the state on the remote store.

Future work has a few options to minimize the RDMA packet drops. For example, one could enable PFC, just like today's RoCE deployment, to avoid congestion drops. Alternative, one may prioritize these RDMA packets so that they are less likely to be dropped, and use a bandwidth cap to prevent RDMA packets taking too much bandwidth. Finally, on the switch side, one can implement parsing and handling of RDMA ACKs/NACKs to make certain remote memory reliable, *e.g.,* in the remote counter case.

Other open problems: This paper aims to propose an ambitious vision, thus leaving many possibilities and problems as future work. These include: 1) to design a programming interface for the primitives that general data-plane applications can easily use; 2) to concretize the systems described in §2; 3) to explore alternative system designs, *e.g.,* for the switch table extension, one may recirculate the original packet locally and wait for the pulled entry, instead of depositing the original packet. This can save the bandwidth overhead to the remote memory; and 4) to improve the robustness of the architecture by handling switch and server failures.

Acknowledgments

We would like to thank the anonymous HotNets reviewers and our shepherd, Vincent Liu for their helpful comments. This work was funded in part by NSF awards 1700521 and 1513764.

References

[1] 2018. Advanced Network Telemetry. https://www.barefootnetworks.com/use-cases/ad-telemetry/.

[2] 2018. Barefoot Capilano. https://www.barefootnetworks.com/products/brief-capilano/.

[3] 2018. Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[4] 2018. BCM88690–10 Tb/s StrataDNX Jericho2 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690.

[5] 2018. Cavium Xpliant Ethernet Switches. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[6] 2018. netpipe(1) - Linux man page. https://linux.die.net/man/1/netpipe.

[7] 2018. Perftest package. https://github.com/linux-rdma/perftest.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM* (2010).

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.

[10] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *EuroSys* (2016).

[11] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1530–1541. https://doi.org/10.14778/1454159.1454225

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *USENIX NSDI* (2014).

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *ACM SOSP* (2015).

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI* (2018).

[15] IEEE. 2011. 802.1Qbb – Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/.

[16] Infiniband Trace Association. 2010. Supplement to InfiniBand architecture specification volume 1 release 1.2.1 annex A16: RDMA over converged ethernet (RoCE).

[17] Infiniband Trace Association. 2010. Supplement to InfiniBand architecture specification volume 1 release 1.2.1 annex A17: RDMA over converged ethernet (RoCE).

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI* (2018).

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP* (2017).

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *ACM SIGCOMM* (2014).

[21] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *ACM SIGCOMM* (2018).

[22] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX OSDI* (2016).

[23] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM* (2016).

[24] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *USENIX ATC* (2017).

[25] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74.

[26] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM* (2017).

[27] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC* (2013).

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM* (2015).

[29] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM* (2017).

[30] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *USENIX NSDI* (2015).

[31] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *ACM SoCC* (2017).

[32] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM* (2015).

[33] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. 2010. Scalable Flow-based Networking with DIFANE. In *ACM SIGCOMM* (2010).

[34] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. 2014. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *ACM EuroSys* (2014).

[35] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution Measurement of Data Center Microbursts. In *ACM IMC* (2017).

[36] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *ACM SIGCOMM* (2015).