PMNet: In-Network Data Persistence*

Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz†‡, and Samira Khan

University of Virginia †*Stanford University* ‡*Purdue University*

1 Introduction

Today, most of the computation takes place in hyper-scale cloud data centers (e.g., Amazon AWS, Microsoft Azure, and Google Cloud). These data centers host workloads ranging from latency-sensitive interactive jobs [2] to long-running workloads with large memory footprints [1, 6]. In most of these workloads, data is persistently maintained in multiple dedicated servers, with clients accessing and updating this data remotely via network, using (synchronous) remote procedure calls (RPCs). During each invocation of an RPC, the request is processed by the client's IO stack, the network of intermediate switches, the server's IO stack as well as the request handler on the server as illustrated in Figure 1. Thus, the latency of an RPC is significantly affected by the processing time of these stages. As the computation performed by modern workloads is dominated by these RPCs, i.e., read and update requests, the access latency of remote data is critical when deploying workloads on modern data centers [3]. Therefore, our goal is to improve the performance (specifically the tail latency) for synchronous RPCs by minimizing the access time to remote persistent data.

2 Background

Recently, with programmable network devices becoming a commodity [4], the trend is to offload application logic to those devices. This way, a large fraction of the procedure, including server's network stack and processing time, is no longer handled by the server but accelerated by those network devices. This newer scheme is known as in-network compute, spanning a wide range of applications, such as key-value stores [10], data aggregation [5], and even computational-intensive machine-learning tasks [11].

Though promising, in-network computing mainly mitigates the latency of computational tasks and requests that do not change the server state (e.g., read queries); the data persistence is still maintained by the servers, and update requests still need to traverse the server's entire network and IO stack to complete the update. Therefore, as in the original case, the client needs to wait for an entire round-trip time (RTT)—for an acknowledgment from the server—before it can proceed.

To minimize the server-side request processing time, data centers are deploying new persistent memory (PM) technologies, such as Intel's Optane. Compared to traditional storage devices (e.g., SSD and HDD), PM provides high-speed and direct, byte-addressable access to persistent data, while bypassing OS indirections (e.g., file systems). PM reduces the server's storage latency by 10~50× [9], thereby enabling software systems, such as key-value stores [8] to perform at much faster speeds. Despite the faster server-side processing, the network is still a dominant factor—causing clients to stall for a complete RTT. Moreover, as the network is a shared resource, the contention for bandwidth, switch queues, and links can further lead to variable delays and long tail latencies [3]. We identify that the fundamental limitations of in-network computing are that switch-local operations are stateless, being unable to accelerate a stateful, persistent operation residing on the server. PMs in the server improve the performance of persistent updates, but the network and server network-stack latency are still on the critical path of the request.

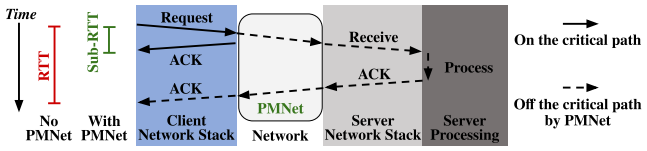

Figure 1. Round-trip time (RTT) of a single request.

3 In-Network Data Persistence

We found that it is possible to expose the persistent state to the network and persist update requests in-network. Therefore, we introduce the notion of *in-network data persistence*, which enables a sub-RTT latency when processing update requests. To expose data-persistence domain to the network, we *log updates* in the network using persistent memory and send acknowledgments to clients as soon as a request enters the persistent domain. The update requests are then forwarded to the server, but this way, the server processing happens off the critical path. As the requests have entered a persistent state before being processed by the server, the client can now proceed before the server acknowledges.

In this work, we design and implement PMNet, a mechanism necessary to provide *persistent logging support* in programmable network devices. However, designing PMNet has many challenges. First, how can a network device track requests and persistently maintain their state? Second, given the requests have been persisted in the network, how can the system recover after a failure? Third, how can PMNet maintain the same application-level ordering guarantees with in-network persistence? Next, we describe our key insights.

Persistent logging. PMNet uses a simple protocol to ensure that updates are logged persistently in the network device

*Published as PMNet: In-Network Data Persistence, Proceedings of the 46th International Symposium on Computer Architecture (ISCA), 2021.

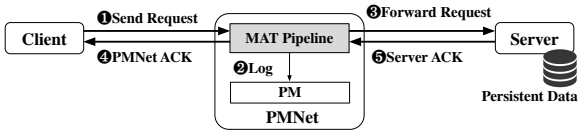

Figure 2. Persistent logging in PMNet.

with sub-RTT latency, as illustrated in Figure 2. First, when incoming update requests are traversing the network device (step ❶), PMNet logs those requests in its PM (step ❷) and forwards them to the server (step ❸). Second, as the requests have already entered a persistent domain of the network device, PMNet immediately sends an acknowledgment to clients (step ❹), allowing them to progress. Therefore, the latency is significantly reduced as the client no longer needs to wait for the whole RTT. Third, PMNet invalidates the logged entry when the server has completed the requests and has sent an acknowledgment to PMNet (step ❺).

System recovery. In case a failure happens in the persistence domain (i.e., the network device and/or server), PMNet needs to ensure that logged entries are reflected on the server. When the system is up again, PMNet resends the logged requests so that servers can redo them in the *same order* as they were sent. As such, the server can recover to a consistent state with the logged requests.

In-order delivery. PMNet always maintains the ordering of the original system. As the logged updates are reflected later on the server, one may think that a client will read a stale value from the server. However, we observe that oftentimes large-scale workloads optimize for independent clients. For example, in a Twitter workload [13], the clients update tweets and followers without maintaining any order. Still, PMNet provides ordering guarantees when there is a strict ordering requirement within multiple clients. These workloads enforce ordering using locks to ensure that only one client can update a critical value. In another example of a TPCC workload [14], the modification of the stock price is placed in a critical section using locking primitives. PMNet forwards the lock operations in a critical section and the ordering is enforced on the server. Once the client acquires the lock, lock requests from *other clients* are rejected. However, subsequent update requests from the *same client* can still benefit from PMNet, i.e., sub-RTT latency.

4 Implementation

We implement PMNet on a Xilinx UltraScale+ FPGA platform, by integrating data persistence on top of NetFPGA-SUME [12] (with a 10G network interface). In this platform, we enable data persistence with 2GB of DRAM-emulated PM that has latency adjusted to match Optane PM. Using this platform, we demonstrate a programmable switch (PMNet-Switch) and a NIC (PMNet-NIC). As the server stack dominates the request RTT, both PMNet-Switch and PMNet-NIC provide similar benefits.

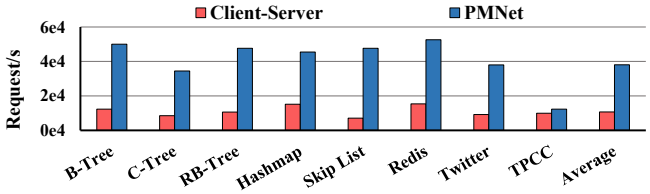

Figure 3. Update throughput of PMNet compared to Client-Server baseline.

On top of PMNet, we further integrate additional functionalities. *(1) PMNet-Switch with caching:* We demonstrate that our logging mechanism for update requests works coherently with a prior work that caches read requests in a switch [10]. *(2) PMNet-Switch with replication:* We develop an in-switch replication mechanism that builds upon PMNet's logging protocol. The whole implementation of PMNet is publicly available at https://pmnet.persistentmemory.org.

5 Contribution and Key Results

In this work, we expose data persistence to the network to improve the performance of update requests. We implement PMNet using a programmable data-plane device, integrated with a persistent memory that logs in-flight update requests. To evaluate PMNet, we adapt eight PM workloads to PMNet, including Intel's PMDK-based key-value stores [7], a PM-optimized Redis database [8], Twitter [13], and TPCC [14]. Figure 3 shows the update throughput of PMNet compared to a Client-Server baseline in eight workloads. The result shows that PMNet improves the throughput of update requests by 4.31× and the 99th-percentile tail latency by 3.23× in these workloads. In addition, PMNet improves read-caching and state-replication latency by 3.36× and 5.88× respectively, over traditional, baseline systems.

References

[1] Abadi et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.
[2] Barroso et al. Web search for a planet: The Google cluster architecture. *MICRO*, 2003.
[3] Barroso et al. Attack of the killer microseconds. *Commun. ACM*, 2017.
[4] Bosshart et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *SIGCOMM*, 2013.
[5] Costa et al. Camdoop: Exploiting in-network aggregation for big data applications. In *NSDI*, 2012.
[6] Dean et al. MapReduce: Simplified data processing on large clusters. In *OSDI*, USA, 2004.
[7] Intel. Persistent memory programming.
[8] Intel. Redis, 2018. https://github.com/pmem/redis/tree/3.2-nvml.
[9] Izraelevitz et al. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv*, 2019.
[10] Jin et al. NetCache: Balancing key-value stores with fast in-network caching. In *SOSP*, 2017.
[11] Li et al. Accelerating distributed reinforcement learning with in-switch computing. In *ISCA*, 2019.
[12] NetFPGA. NetFPGA-SUME Virtex-7 FPGA development board.
[13] Sanfilippo. antirez/retwis. https://github.com/antirez/retwis.
[14] Transaction Processing Performance Council (TPC). TPC-C. http://www.tpc.org/tpcc/.