2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00068

PMNet: In-Network Data Persistence

Korakit Seemakhupt*, Sihang Liu*, Yasas Senevirathne*, Muhammad Shahbaz†‡, and Samira Khan*
*University of Virginia †Stanford University ‡Purdue University

***Abstract*—To guarantee data persistence, storage workloads (such as key-value stores and databases) typically use a synchronous protocol that places the network and server stack latency on the critical path of request processing. The use of the fast and byte-addressable persistent memory (PM) has helped mitigate the storage overhead of the server stack; yet, networking is still a dominant factor in the end-to-end latency of request processing. Emerging programmable network devices can reduce network latency by moving parts of the applications' compute into the network (e.g., caching results for read requests); however, for update requests, the client still has to stall on the server to commit the updates, persistently.**

In this work, we introduce *in-network data persistence* that extends the data-persistence domain from servers to the network, and present PMNet, a programmable data plane (e.g., switch or NIC) with PM for persisting data in the network. PMNet logs incoming update requests and acknowledges clients directly without having them wait on the server to commit the request. In case of a failure, the logged requests act as redo logs for the server to recover. We implement PMNet on an FPGA and evaluate its performance using common PM workloads, including key-value stores and PM-backed applications. Our evaluation shows that PMNet can improve the throughput of update requests by 4.31× on average, and the 99th-percentile tail latency by 3.23×.

***Index Terms*—In-Network Processing, Programmable Network, Switch, NIC, Data Center, Persistent Memory, Tail Latency, RPC**

I. Introduction

The benefits of economy of scale and the emergence of cloud computing have caused an increasing number of enterprises to move their workloads to hyper-scale cloud data centers, with an estimated annual growth of about 14.6% during the period 2017–2022 [76]. Today, most of the computation takes place in these hyper-scale cloud data centers—performing more than 89% of the computation world-wide in 2018 [101].

These data centers host workloads ranging from time-critical, interactive jobs (e.g., online data-intensive (OLDI) workloads [11], RAMCloud [86, 87], and financial analysis [57]) to long-running, batch jobs (e.g., MapReduce [31] and machine-learning training [1]) with large memory footprints. In most of these workloads, data is typically managed and maintained in a persistent way across multiple servers, with clients accessing and updating this data remotely over a network of interconnected switches, using 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. Thus, the latency of an RPC is significantly affected by the processing time of each of these stages. As the computation performed by modern workloads is dominated by these RPCs, i.e., read and update requests, the time it takes to access remote data is of major consideration when deploying workloads on modern data centers [2, 3, 13, 82, 110].

These RPCs can be either *synchronous* or *asynchronous*. Though, asynchronous RPCs can enable clients to continue execution while updates are being processed at the remote server; yet, building such applications is quite challenging, especially "at scale, when a typical end-to-end application can span multiple small closely interacting systems" [13]. In contrast, applications using synchronous RPCs are easy to write, tune, and debug—Google is known to strongly prefer a synchronous programming model [25, 97]. Therefore, in this paper, our aim is to improve the performance (specifically the tail latency) for synchronous RPCs by minimizing the access time to remote persistent data.

Recently, as programmable network devices become available [15], a 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 computation scheme is known as in-network compute, spanning a wide range of applications, such as query processing [41, 63], key-value stores [51, 64, 70, 103], data aggregation [26, 93, 116], and even computational-intensive machine-learning tasks [36, 66, 67].

Though promising, in-network computing mainly mitigates the latency of computational tasks and requests that do not change the server state (such as read queries); the data persistence is still maintained by the servers, and update requests still need to traverse the entire network and server's 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 acknowledgement from the server—before it can proceed to the next step.

To minimize the request processing time on the servers, data centers [10, 14] are deploying new persistent memory (PM) technologies, such as Intel's Optane [44] and NVDIMM [91]. 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× [32, 49, 62], thereby enabling software systems (such as databases [6, 48, 61, 75] and key-value stores [20, 109, 111]) to perform at much faster speeds. Even though the integration of PM significantly reduces the request processing time at individual servers, the network is still a dominant factor when processing these requests in a data center—causing clients to stall for a complete RTT. Moreover, as the network is a shared resource, the contention for bandwidth, switch queues, and links

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00068

can lead to variable delays and long tail latencies [27]–[29, 38, 81, 88, 99, 100, 118]. We identify that the fundamental cause of the limitations of in-network computing is that operations are stateless, being unable to accelerate a stateful, persistent operation on the server. On the other hand, persistent memory in the server improves the performance of persistent updates but still puts the network and server network-stack latency on the critical path.

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 acknowledgements 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,[1] 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 with sub-RTT latency. First, PMNet mirrors the incoming update requests while they are traversing the network device. It logs the update requests in its PM. Second, as the requests have already entered a persistent domain of the network device, PMNet immediately sends an acknowledgement to clients, allowing them to move forward. Therefore, compared to the original scenario, the latency is significantly reduced as the client no longer needs to wait for the whole RTT. Third, PMNet invalidates the logged entry upon reception of an acknowledgement from the server, which indicates that the server has completed the request.

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 using Redis as the backend [92], 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 application-level ordering using synchronization primitives to ensure that only one client can update a critical value. For example, a TPCC workload [104] puts the modification of the stock price in a critical section using locking primitives. PMNet treats the lock operations in a critical section as regular read requests and forwards them to the server. Therefore, the ordering is enforced on the server and subsequent lock requests from other clients fail on the server, but subsequent update requests from the same client operate with sub-RTT latency by persisting these update requests in the network device.

We implemented PMNet in an FPGA-based programmable switch (PMNet-Switch) and a NIC (PMNet-NIC). Our evaluation shows that both designs provide a significant benefit over the baseline system. However, the latency difference between PMNet-Switch and PMNet-NIC is negligible as the round-trip time is dominated by the server network stack and the server processing time. 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 proposed to cache read requests in a switch [51]. *(2) PMNet-Switch with replication:* We develop an in-switch replication mechanism that builds upon PMNet's logging protocol. In summary, we make the following contributions:

- 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 incomplete update requests.
- We adapt common PM workloads, including Intel's PMDK-based key-value stores [45], a PM-optimized Redis database [48], Twitter [92], and TPCC [104] to PMNet. Our evaluation shows that PMNet improves the throughput of update requests by 4.31× and the 99th-percentile tail latency by 3.23× in these workloads.
- We also demonstrate that PMNet improves read-caching and state-replication latency by 3.36× and 5.88× respectively, over traditional, baseline systems.

II. Background and Motivation

In this section, we first discuss the performance bottlenecks in performing update requests in persistent applications and then describe our proposed solution.

A. Synchronous Programming Model

Data centers host a wide range of workloads, such as online data-intensive (OLDI) workloads, RAMCloud, and financial analytics [1, 11, 31, 57, 86, 87], where data is typically managed in a persistent way through multiple servers, and accessed via queries. If we categorize these requests, there are mainly two types: synchronous and asynchronous. A synchronous request guarantees its completion at the server by blocking the client

[1]PMNet is publicly available at https://pmnet.persistentmemory.org.

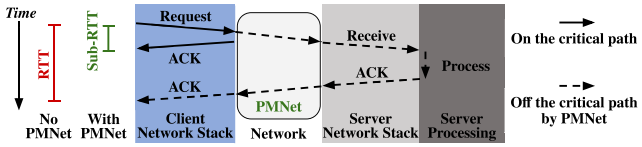

Fig. 1: Round-trip time (RTT) of a single request.

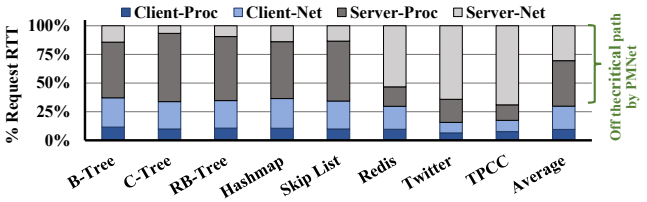

Fig. 2: Latency breakdown of an update request.

until the server responds. In comparison, an asynchronous request lets the client proceed immediately while the request is being delivered and processed at the server. However, the client risks losing data in the face of common failures, such as network packet loss. When such failure happens, the application needs to ensure that the clients and the servers remain in-sync—complicating the design and development of the application. Therefore, programmers usually prefer the synchronous programming model [13, 25, 80, 97].

B. Mitigation of the Synchronous Overhead

Although the synchronous model alleviates the programming burdens, it places the entire query RTT on the critical path of the application, as future requests are blocked until the in-flight request has been processed by the server. Figure 1 demonstrates the steps involved in query processing, including the client's network stack, the network latency, the server's network stack, and the server's request processing. The dashed arrows demonstrate the RTT of processing a single request. Further breakdown in Figure 2 shows that server-side latency, including the server's network stack (in the kernel) and request processing time (in the user-space), makes up the majority of the overhead (70% on average). To mitigate the synchronous overhead, one of the promising solutions in the literature is to offload tasks to the network. By placing computational devices—such as programmable switches [9] and SmartNICs [112]—on the network path, a large portion of the server's processing and network stack latency can be eliminated. Examples of in-network computing include load balancing [54, 79], congestion control [95, 96] and packet scheduling [40, 98], query processing [41, 63], key-value stores [51, 64, 70, 103], and machine-learning acceleration [36, 66, 67].

However, in-network compute only handles stateless requests that do not change the persistent state. In case of an update request, the communication and processing overhead remains. Fortunately, recent advancement in memory technology provides an alternative, high-performance storage system, persistent memory (PM) [44]. By managing persistent data on PM, the server can perform update requests more efficiently and reduce the server-processing latency. For example, databases and key-value stores [6, 20, 48, 61, 75, 109, 111], and PM-optimized file systems [34, 59, 60, 74, 106, 108, 114, 115] optimize their data management to take advantage of PM's high performance. Despite these improvements, the network stack and the processing time remain on the critical path. Even with an optimized network stack, such as using libVMA [77] that enables applications to bypass the server's network stack, the server-processing time is still a major overhead (we evaluate an optimized network stack in Section VI-B7).

In this work, we ask the following question: *how can we move the server's network stack and processing time off the critical path and process the client requests in sub-RTT?* A promising solution is to log and persist the update requests ahead of time before they enter the server stack and processing time. A dedicated logging module can bypass the majority of system overhead. Unfortunately, a dedicated software module can introduce additional CPU and memory utilization (evaluated in Section VI-B2), and a dedicated hardware module needs a redesign of the CPU architecture, making fast deployment challenging. The trend in data centers is to move dedicated logic into NICs and switches [26, 36, 41, 51, 63, 64, 66, 67, 70, 93, 103, 116]. This method is effective in handling network traffic, without putting extra load on the server. Following this trend, our *goal* is to *design and prototype* an efficient logging mechanism using programmable network devices.

C. In-Network Data Persistence

For data to become persistent in the network, we need to extend the data-persistence domain from within servers to the network. By maintaining the persistent state of ongoing requests in the network, an update request can become equivalently persistent *before* having been processed by the server. In case of a failure on the server, the already persisted requests can be resent from network devices and re-applied to the server for recovery. As the solid arrows in Figure 1 demonstrate, clients no longer need to wait for the entire RTT for servers to process the requests and reach a persistent state. Instead, they can proceed once the requests have entered the persistence domain of the network. Consequently, the entire server-side latency (as pointed out in Figure 2), including the request handler's processing time and the network stack latency, can be placed off the critical path. Therefore, persisting these requests in the network can significantly improve performance by taking the server off the critical path.

III. High-level Ideas

We present PMNet, a PM-integrated programmable network device that realizes in-network data persistence. By keeping a persistent copy of in-flight requests on the network device, the update will end up in a persistent state *before* having been received and processed by the server. In case a server fails (e.g., due to a power outage or kernel crash), the persistent copy of the request can act as a redo log for the in-flight requests when the server recovers from failure. At a high level, the design and implementation of PMNet have three major challenges: (1) How can PMNet move the network and server processing

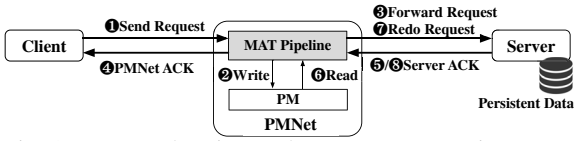

Fig. 3: Request logging and system recovery in PMNet.

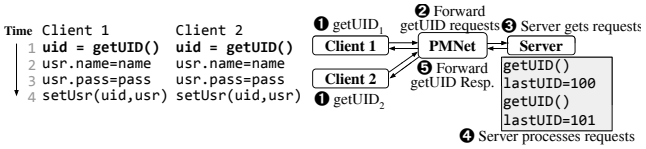

Fig. 4: No ordering in a Twitter workload [92].

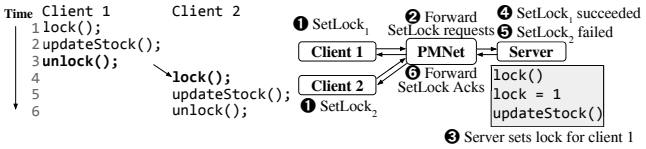

Fig. 5: Application-level ordering in a TPCC workload [104].

time off the critical path by logging the requests? (2) How can the system recover using the logged requests? (3) How can the existing client and server applications maintain ordering after having PMNet integrated? Next, we describe the high-level ideas of PMNet that address these challenges.

A. Persistent Logging

With the integration of PM, PMNet can maintain a copy of in-the-flight update requests in PM, to serve as a redo log for the server. This way, the request reaches a persistent state before it has been processed by the server (Figure 1). PMNet then sends the acknowledgement of the request to the client once it has been persisted in the network device's PM, rather than waiting for the server's acknowledgement. Figure 3 shows the workflow of the request logging procedure: Upon receiving an update request, PMNet writes it to the integrated PM (step ❶-❷). While the request is being written to PM, PMNet forwards it to the destination server (step ❸). After the request becomes persistent, PMNet sends an acknowledgement (`PMNet-ACK`) to the client, indicating that this request has entered a persistent state (step ❹). When the server has actually processed the request, it acknowledges (`server-ACK`) PMNet to invalidate its copy and reclaim the log entry for future use (step ❺). On the client-side, upon receiving an acknowledgement from PMNet, the client proceeds without waiting for the completion of the request at the server (details in Section IV-B). PMNet can work as a *switch* as well as a *NIC*, which we refer to as *PMNet-Switch* and *PMNet-NIC*, respectively.

B. System Recovery

The next challenge is to recover the server after a failure. Figure 3 shows the recovery procedure. Upon detection of a server failure (e.g., through heartbeat signals), PMNet reads the logged requests from the device's PM (step ❻) and resends them to the server (step ❼). PMNet ensures that the server commits the resend requests in the original order by adding an extra *sequence number* in the header (details in Section IV-E). Once the server has committed a request, it notifies PMNet to invalidate the log entry (step ❽). Besides this sequence number, the update requests need additional information to ensure correct in-network logging and recovery. PMNet encodes this information as a new PMNet header (details in Section IV-A) to existing network protocols (e.g., IP or VXLAN), and provides software support for the client and server to process this encoding with minimum changes to the source code.

C. In-order Delivery

PMNet guarantees the same ordering as the traditional, baseline systems. We already discussed that PMNet maintains ordering within a client by keeping additional sequence information in the header of the packets from each client. However, one may wonder how PMNet ensures the ordering between multiple clients. Can another client read the up-to-date state from the server, while the update is being logged in PMNet and not yet committed to the server?

PMNet targets the common cases, where the majority of client-server connections are independent. This is unsurprising because the RTT between a client and server is as high as tens of µs—having dependencies (and synchronization) with other clients can inflate the end-to-end latency of requests even further. Therefore, large-scale workloads typically mitigate dependencies and optimize for independent clients. For example, different tasks in a microservice use separate storage backends without dependencies with others [38, 100], worker nodes in distributed machine-learning systems send new weights to a parameter server without synchronizing with other workers [1, 30], and client-independent databases, such as Memcached and Redis, are commonly used as service backends [7, 89]. Figure 4 demonstrates two update requests to a shared variable `lastUID` in the *Twitter* workload [92]. There is no ordering constraint among clients, and each client independently executes the `getUID` function (line 1) and uses that `UID` for consecutive requests.

However, in rare cases where ordering must be enforced among multiple clients—i.e., one client cannot update the server until another completes—the client needs to make sure the request has actually been processed by the server before making any forward progress. These workloads typically use synchronization primitives to ensure the ordering at the application level. The client uses an update request to access the synchronization primitive on the server and acquire the lock on the critical section; other clients are blocked from entering the critical section, thus enforcing an ordering among clients. Figure 5 shows this synchronization scheme in the TPCC workload [104], where PMNet directly forwards the locking requests to the server and only *client 1* can access the critical section. And, subsequent update requests from *client 1* can still benefit from PMNet. As lock requests are forwarded to the server, the ordering of lock-acquire/release is enforced. Because lock requests are infrequent, the majority of requests can be logged by PMNet. In our experiment, most workloads are lock-free. Only 13.7% of the requests in the TPCC workload access the locking primitive (i.e., bypass PMNet).

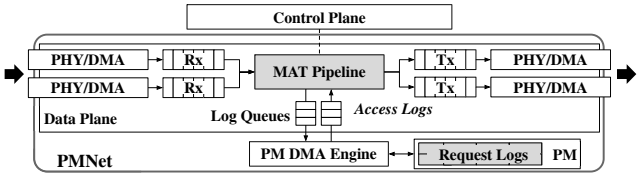

Fig. 6: A high-level view of the PMNet architecture.

IV. PMNet Design

So far, we have introduced the high-level ideas of PMNet, that extend the persistence domain from servers to the network by logging in-flight update queries in the network devices' PMs, and redo them in the event of failures. This way, PMNet effectively moves server-stack overhead off the critical path of request processing.

PMNet realizes in-network data persistence by augmenting programmable network devices (e.g., switches and NICs) with PM, as the architecture overview in Figure 6 shows. To have PMNet fully functioning end-to-end, from the client-server application to the hardware implementation, we introduce four major aspects in this section. First, we introduce PMNet protocol for update requests that can benefit from in-network data persistence, which encodes metadata that is necessary for logging and recovery into a PMNet header (Section IV-A). Second, we describe the request processing procedure of PMNet, according to the PMNet protocol (Section IV-B). Third, based on the design of PMNet, we further introduce two use-cases of PMNet. One case that integrates PMNet into a replication system (Section IV-C), and another case that implements a read cache on top of PMNet's logs (Section IV-D). Finally, having with all the details described, we illustrate the recoverability of a PMNet-based system (Section IV-E).

A. PMNet Protocol

We now describe the PMNet protocol, including its packet header, delivery method, and ordering guarantees for queries.

*1) **Header format**:* The PMNet header is placed in the application layer of each network packet (i.e., L4) (Figure 8). A PMNet header consists of the following fields:

- **Type** (8 bits) differentiates the type of PMNet's requests (details in IV-B1).
- **SessionID** (16 bits) keeps track of sessions a client sends requests from and differentiates connections among clients.
- **SeqNum** (32 bits) tracks the order of packets sent over a given session, such that the server can process the queries in the original order. Furthermore, the latest `SeqNum` informs PMNet to avoid redoing already completed queries during the recovery (Section IV-E).
- **HashVal** (32 bits) is a CRC-32 hash of the entire header that is computed by the sender's network stack. The PMNet uses this hash value to index a packet in the PM of PMNet.

*2) **Delivery method**:* The PMNet protocol is built upon UDP, similar to other in-network compute works [50, 65, 67, 103]. To differentiate from other network traffic, PMNet reserves specific UDP ports (range: 51000–52000) and encodes PMNet header into the UDP packet. In case an application originally uses TCP, PMNet's software library converts TCP packets to UDP packets while maintaining a reliable delivery guarantee of TCP (similar to [96]). We present how PMNet protocol ensures the packet ordering and integrity guarantees in Section IV-A4.

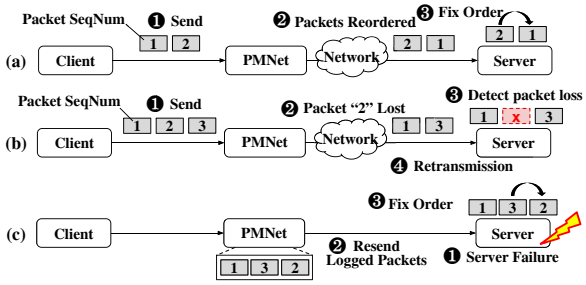

Fig. 7: Per-client packet ordering guarantee in three cases: (a) reordered packets, (b) packet loss, and (c) failure.

*3) **MTU-sized packets**:* PMNet obtains the packet size from the UDP header. Though a UDP packet typically has a maximum transmission unit (MTU) of 1.5 kB, a query can be larger than this limit. PMNet's software library transparently divides the queries larger than MTU into smaller packets and uses a sequence number (`SeqNum`) to maintain packet order. PMNet handles MTU-sized packets by sending a per-packet `PMNet-ACK` to the client once each packet has been logged in PMNet. And, the client needs to collect *all* `PMNet-ACK`'s to make sure the corresponding update request has been completely logged in PMNet. The PMNet's software library also tracks the number of `PMNet-ACK`'s in a similar way.

*4) **Ordering guarantees**:* As UDP does not guarantee packet ordering, PMNet protocol implements such ordering for servers to execute queries from the same client in the *original order*. We discuss the ordering guarantee in three scenarios. (1) During normal execution: Figure 7a demonstrates a scenario where the client first sends packets in the original order (step ❶), and during network transmission, some packets are reordered (step ❷). On detection of out-of-order packets, the server's PMNet library corrects the order based on `SeqNum` of each packet (step ❸). (2) On detection of a packet loss: Figure 7b demonstrates a scenario where the client sends a series of packets (step ❶), but some packets are lost (packet-#2 in this example) during network transmission (step ❷). The server's PMNet library detects nonconsecutive `SeqNum` (step ❸) and requests for retransmission (step ❹). Section IV-B describes more details about retransmission. (3) During failure recovery: Figure 7c demonstrates a scenario where the server fails (step ❶) and PMNet retransmits the logged packets to the server for recovery (step ❷). Similar to scenario (1), the server reorders the packets on reception (step ❸). In conclusion, although the network may reorder or lose packets, PMNet protocol guarantees their ordering and delivery.[2]

[2]Note that handling reordering at the network level is rare as datacenter networks typically use flow-consistent load balancing (e.g., ECMP [53]).

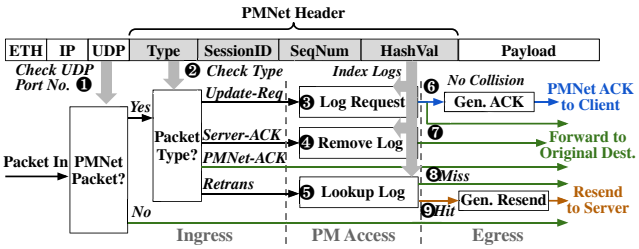

Fig. 8: The data-plane MAT pipeline in PMNet.

Multi-client ordering: PMNet enforces multi-client ordering at the application level by forwarding synchronization requests directly to the server (described in Section III-C). Doing so allows the server to enforce ordering across critical sections of the application code, while the code within the critical sections can still benefit from PMNet's logging.

B. PMNet Request Processing and Log Management

1) ***Packet handling:*** In PMNet, a match-action table (MAT) pipeline (Figure 6) handles the following types of PMNet packets, distinguished via the "`Type`" field, along with other non-PMNet packets.

- **Update requests from the clients (`update-req`).** PMNet needs to maintain the persistent state of these queries in order to recover. Therefore, upon reception of a packet that belongs to an update request, PMNet immediately forwards the packet to the destination server, and in the meantime, logs the packet in the persistent memory. The `HashVal` in the PMNet header serves as the index to the log entry. Once the whole packet has been persisted to the network device's PM, PMNet sends a `PMNet-ACK` back to the client, as the packet has reached a persistent state. Note that in cases where the device PM is full or the `HashVal` of the packet collides with an existing entry, PMNet directly forwards the packet to the destination server without logging it (or acknowledging the client).
- **Bypass request from the client (`bypass-req`).** For purposes such as read request and synchronization, where the client does not need to receive an early acknowledgement from PMNet, the client sets the packet type to `bypass-req`. PMNet directly forwards it to the destination without logging. As a result, these packets do not enter a persistent state until processed by the server.
- **ACK from another PMNet (`PMNet-ACK`).** In a system with multiple PMNets, a PMNet may receive a PMNet-ACK from another PMNet. In this case, PMNet directly forwards the packet along its path to the destination.
- **ACK from a server (`server-ACK`).** Upon reception of a `server-ACK`, PMNet looks up the request log with the `HashVal` in the packet. If the request log hits, PMNet forwards this `server-ACK` to the destination (the next PMNet in this route may log the request).
- **Retransmission request from a server (`Retrans`).** In case the server detects any packet loss, it sends a `Retrans` request to the client, going through PMNet. If PMNet has the requested packet logged, PMNet directly sends it to the server and drops this `Retrans` request. Otherwise, PMNet forwards this `Retrans` request directly to the target client.
- **Non-PMNet packets.** As PMNet also serves as a regular network device, it also handles non-PMNet packets by directly forwarding them to the destination.

2) ***MAT pipeline workflow:*** PMNet implements the MAT pipeline for packet processing, as shown in Figure 8. The pipeline contains three stages: *ingress*, *PM-access*, and *egress*. The *ingress* pipeline first checks whether this packet is a PMNet packet based on its port number in the UDP header (step ❶); non-PMNet packets are directly forwarded to the destination port. Second, for the remaining PMNet packets, it checks the type of the packet based on the `Type` field in the header (step ❷), and forwards `PMNet-ACK` packets to the destination. The *PM-access* stage operates on the request logs. It creates a log upon receiving an `update-req` packet (step ❸), removes it upon a `server-ACK` packet (step ❹), and looks up a log upon a `Retrans` packet (step ❺), by using `HashAddr` as the index. The *egress* stage processes the outgoing packets based on the outcome of the (log) lookup. For `update-req` packets, it first forwards all of them to the destination server (step ❼); and for those that can be logged in its PM, it additionally generates and sends a `PMNet-ACK` to the client (step ❻). For `Retrans` packets, if the log entry is present, the egress pipeline generates and resends the requested packet to the server; otherwise, it forwards this `Retrans` to the destination client.

During the PM-access stage, PMNet manages log entries in its PM through a DMA engine (i.e., to add/remove/read a log entry). Different from the rest of the MAT pipeline, the PM access stage can suffer from the longer PM access latencies. To prevent blocking incoming packets, PMNet maintains *log queues* (separate queues for reads and writes) that buffer PM access requests (as shown in Figure 6). This way, PMNet can handle all incoming packets at line-rate (Section VI-B).

C. PMNet Replication

Replication is a common fault-tolerance mechanism that maintains multiple copies of the same data block across various storage servers in a distributed system [12, 17, 21, 55]. In case one server fails, the other can be used to recover the corrupted data. To enable fault-tolerance, an update request must commit to all the replication servers. PMNet can accelerate these fault-tolerance systems, further, by replicating data using in-network PMs. Figure 9a demonstrates a scheme where the server replicates data. Accordingly, we place two switches in series to maintain two copies of logs as well.

An update request is processed in the following steps: The client sends an update request (step ❶), and the first PMNet switch (#1) logs the request (step ❷) and sends a `PMNet-ACK` (#1) (step ❸). Then, the second PMNet switch (#2) receives the forwarded request, performs logging (step ❹), and sends a second `PMNet-ACK` (#2) (step ❺). On the client side, it continues processing only after it has received *both* `PMNet-ACK` (#1) and (#2). On the server side, the primary server receives the update request and process it (step ❻).

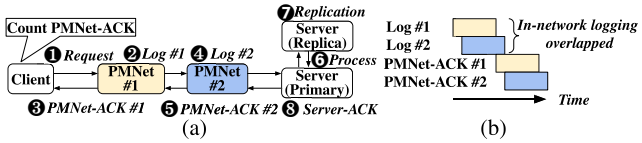

Fig. 9: (a) A replication scheme with two PMNet switches. (b) Timeline of in-network replication.

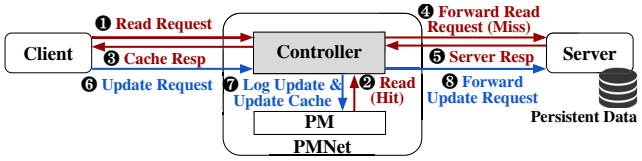

Fig. 10: Caching steps in a PMNet switch.

Afterward, it sends the updated data to the replication server. Only after replication completes (step ❼), the primary server sends a `server-ACK` to invalidate both logs in the two PMNet switches (step ❽). Even though such in-network replication requires all switches to persist the log prior to sending an acknowledgement, the latencies for persisting data at each switch are overlapped. Figure 9b shows the latency benefits of this overlap in the replication procedure for two switches (evaluation in Section VI-B5).

D. PMNet Read Caching

Prior works have used programmable switches as a cache for key-value stores [51, 70, 103] but they do not maintain a persistent state of the data cache and cannot mitigate RTT for update requests. To serve both read and update requests, we add a read cache on top of PMNet's persistent log. With read caching enabled, PMNet is implemented as the server's ToR switch, similar to the prior works on in-network read cache, to simplify consistency issues [51]. It maintains a persistent key-value cache that logs update requests and responses to read requests. Figure 10 shows this procedure. When a read request arrives at PMNet (step ❶), it looks up the cache. If the request is a hit in the cache (step ❷), PMNet sends a cache-response to serve the read (step ❸). In case of a miss, the read request is forwarded to the server as normal (step ❹), and the response is cached in PMNet when the server replies (step ❺). On the other hand, an update request is first logged in PMNet (step ❼) to move server processing off the critical path (step ❽). In case the key in the update request is a hit to the cache (i.e., has been cached already), it updates the cache to maintain consistency (step ❼).

Figure 11 describes a state diagram, where each entry in PMNet has four states: (1) **`Invalid`**: the entry is empty (initial). (2) **`Stale`**: the entry is not up-to-date when there is an in-the-flight update to the same key (3) **`Persisted`**: the request logged by this entry has been persisted on the server, and (4) **`Pending`**: the request has been logged by PMNet but not persisted by the server. When the state is `Pending` or `Persisted`, the entry can serve for read cache. Here are the state transitions. **T1:** Upon receiving an `update-req` from the client, PMNet logs the request. As the server has not

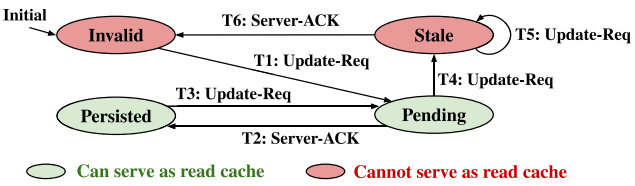

Fig. 11: A state diagram for PMNet with integrated read cache.

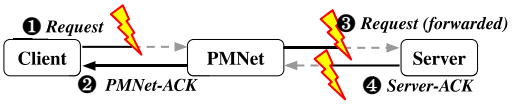

Fig. 12: Intermittent failure scenarios.

persisted it, the state is now `Pending`. **T2:** After receiving a `server-ack`, PMNet is notified that the server has persisted the request. Thus, the status becomes `Persisted`. **T3:** If the same entry (indexed by the key) is updated again (via an `update-req`), PMNet directly bypasses this request and the state goes back to `Pending`. **T4:** When an entry is `Pending` but receives another `update-req`, the state becomes `Stale` as the server will be updated with a new value. **T5:** A `Stale` entry remains `Stale`, after getting another `update-req`. **T6:** Once a `Stale` entry receives a `server-ACK`, it becomes `Invalid` as its prior `update-req` has been persisted.

E. PMNet Failure Recovery

Our work focuses on a system with client-server architecture located in a modern data center. Here, we consider both intermittent failures (such as power outage, software bug and other hardware failures that cause the system to temporarily become unavailable) and permanent hardware failures that are handled via replication (details in Section IV-C).

*1) **Intermittent failures**:* In this case, all hardware regains their functionality after the failure ends. However, data that is outside the persistent domain is lost. To ensure data integrity, PMNet ensures that accepted requests are safely stored in the persistent domain. We categorize such failures into three cases.

- **PMNet fails before receiving the request from the client** (during step ❶ in Figure 12). In this case, as the packet has not been accepted by either PMNet or the server, the client is not acknowledged. Therefore, the client will continue to be stalled by the in-flight request and simply needs to resend the request after timeout (or during recovery).
- **PMNet fails after accepting a request but before a server receives the forwarded request** (during step ❸ in Figure 12). We discuss two cases. **(1)** The `PMNet-ACK` was sent to the client before failure (step ❷ is complete). In this scenario, the client has assumed the packet has been processed but the server has not processed it, in fact. During recovery, the server polls PMNet for logged requests with the sequence number starting from the last packet it receives. Then, the server applies logged requests in the same order as they were sent by the client. Note, PMNet itself is agnostic of the packet ordering for better performance, but the server uses `SeqNum` to maintain the order of requests. **(2)** The

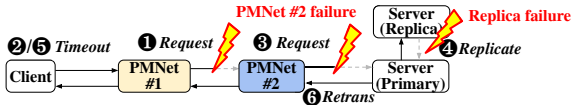

Fig. 13: Failure-recovery in a PMNet system with replication.

PMNet-ACK was *not* sent to the client before the failure (step ❷ is incomplete). In this scenario, the client has not received the acknowledgement and, therefore, it will be stalling on the current request. After the client times out, it will resend the request again.

- **PMNet fails before receiving the `server-ACK`** (during step ❹ in Figure 12). PMNet first resends the logged requests to the server. Upon reception of the resent request packets, the server checks the `SeqNum` of each packet. As the server has processed the request, the latest `SeqNum` is greater than that in the resent request and, therefore, the server drops these requests and sends a make-up `server-ACK` to invalidate the log entries in PMNet.

*2) **Permanent failures**:* In this type of failure, the hardware that stores persistent data (PMNet or server) cannot be recovered. Such failures are handled by replicating data in the persistent domain across multiple devices. We further categorized permanent failures into three cases.

- **A PMNet fails permanently after accepting a request but before the server receives the forwarded request.** For PMNet to handle permanent failure, the update request must persist in all PMNet all the way to the server before it is accepted (as in Section IV-C). We discuss two cases. **(1) PMNet #2 fails before both PMNet devices have accepted the request** (step ❶ in Figure 13). Because the client has not received both `PMNet-ACK`'s to satisfy the replication requirement, it will be stalling on the current request until it times out[3] and resends the request (step ❷ in Figure 13). **(2) PMNet #2 fails after both PMNet devices have accepted the request and send `PMNet-ACK`** (step ❸ in Figure 13). During recovery, the server polls PMNet for the logged requests. Because all PMNet devices have logged the request, any surviving PMNet can retransmit the request to the server.
- **Server replication system fails before sending `server-ACK`**. In a replication system, the primary server sends `Server-ACK` only after it has committed the update in all replicas. Thus, when a replication server fails (step ❹ in Figure 13), the server will not send `server-ACK` to invalidate the logged requests in PMNet. Eventually, when PMNet's log entries are full, PMNet forwards newer requests directly to the server. However, those requests will not be processed by the failing replica and the client will eventually timeout[3] (step ❺ in Figure 13). All the in-flight requests are now logged in PMNet with equivalent replication strength (the same number of PMNet devices as the replicas). Once the replication system is up, the primary server sends `Retrans` (step ❻ in Figure 13) to get the logged requests from PMNet. And, the retransmitted requests will be processed by the server in-order based on the sequence number. Note that these systems typically monitor servers' status using heartbeats. Therefore, the client will be notified as soon as a replica fails—the client would not need to wait until it times out.

[3]The timeout value is 2× 99th-percentile RTT, which is 700 µs.

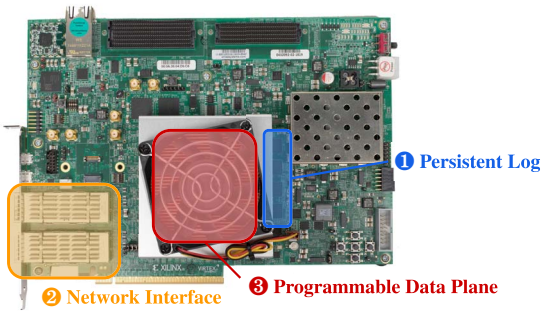

Fig. 14: FPGA platform for PMNet implementation.

*3) **Failure of other components**:* When components outside of the persistence domain fail, such as the client or other non-persistent network devices, the system follows the original procedure for recovery. The recovery procedure does not change as the persistence guarantee of those devices remains the same.

V. PMNet Implementation

A. Hardware Implementation

We express PMNet's processing pipeline in the P4 language [16]. We choose Xilinx UltraScale+ VCU118 platform (Figure 14) as the hardware for the programmable network device and maintain the request logs in its 2GB on-board DRAM (labeled as component ❶). The DRAM write latency in the FPGA is 273 ns (due the slow DMA engine on the FPGA) which is close to Optane PM's write latency [107]. We adapt the open-source code from NetFPGA-SUME (with 10G Ethernet MAC) [84] to our UltraScale+ platform and integrate PMNet into it. The network interface is labeled as component ❷. The whole design uses 13% of LUT, 19% of BRAM, and 31% of IO resources of the FPGA. The FPGA chip is labeled as component ❸. We use a Li-ion battery module to back the device during power failure.

***Design choices**:* The in-network PM needs to log all the in-flight update requests. In our evaluation, the 99th-percentile RTT of update requests is 350 µs. If we conservatively take 500 µs as the maximum RTT and 10 Gbps as the bandwidth, the bandwidth-delay product (BDP) is:

$$BDP_{Net} = RTT \times BW = 500 \cdot 10^{-6} \times 10 \cdot 10^{9} \approx 5Mbits. \quad (1)$$

Our FPGA board has sufficient memory capacity to log all these on-going update requests. Because of the slower PM access latency, we buffer the accesses to the in-network PM on the PMNet device (switch or NIC) to process packets at a line rate. The required buffer size also follows a BDP calculation where the delay equals to the memory-access latency.

$$BDP_{PM} = PMLatency \times BW = 100 \cdot 10^{-9} \times 10 \cdot 10^{9} \approx 1kbits. \quad (2)$$

TABLE I: PMNet software interface.

Client Software Interface	
`PMNet_send_update()`	Send an `update-req` to server
`PMNet_bypass()`	Send a `bypass-req` to server
`PMNet_start_session()`	Start a session
`PMNet_end_session()`	End a session
Server Software Interface	
`PMNet_recv()`	Receive requests from client
`PMNet_ack()`	Send ACK to PMNet

TABLE II: System configuration.

Server Configuration	
CPU	Intel Cascade Lake, 2.1GHz, 20 cores
DRAM	6×32GB DDR4, 2666MT/s
PM	2×128GB Intel DCPMM, Interleaved, 2-1-1 Config, App Direct Mode, Mounted as EXT4-DAX
NIC	Mellanox ConnectX-3 MCX314A
Client Configuration	
CPU	Intel Haswell, 3.6GHz, 6 cores
DRAM	4×16GB DDR4, 2133MT/s
NIC	Mellanox ConnectX-3 MCX314A
Software System	
OS	Ubuntu 19.10, Linux kernel v5.3.0
Tools & Libs	gcc/g++-9.2, PMDK-1.8, daxctl/ndctl-65 (server only)

We conservatively use 4 KB of SRAM as the queue size for logging both reads and writes to PM. Section VII discusses support for even higher network bandwidth with PMNet.

B. Software Implementation

We develop an easy-to-use software interface that allows programmers to adapt existing workloads to a PMNet system (`PMNet_interface.h`). Table I lists the interface functions. Programmers need to overwrite the existing send and receive functions of the system's socket interface with the PMNet version. Then, the PMNet library operates on these functions and encapsulates payload in PMNet-compatible formats. In addition, PMNet library serves two major purposes. First, it accepts incoming packets from PMNet to mitigate the RTT for the client (Section IV-B). Second, it maintains the ordering and integrity of PMNet packets for the server (Section IV-A4).

VI. Evaluation

A. Methodology

*1) **System setup**:* We evaluate PMNet using a testbed, described in Table II. The server is equipped with Intel's DC Persistent Memory [44] that are mounted in DAX-FS mode for workloads to directly manage PM. The clients contain normal DRAMs on the machines and send requests to the server for persistent data access. Both the server and clients use Mellanox NICs for network connection. In total, we have 4 client machines, each running up to 16 client instances (64 in total), and 3 Xilinx UltraScale+ FPGAs that are programmed as PMNet-NICs/Switches. In the PMNet-Switch configuration, the client machines are connected to a top-of-the-rack PMNet switch (Section V-A). Due to the limited number of Ethernet ports on the FPGA board, we place a regular switch (with sub-microsecond latency) in the middle of clients and the FPGA to merge their traffic. In the PMNet-NIC configuration, the client machines are connected to a regular switch directly; for the server, the FPGA is placed as a bump-in-the-wire between the server's NIC and the ToR switch, similar to recent Microsoft's SmartNIC setup [35, 36].

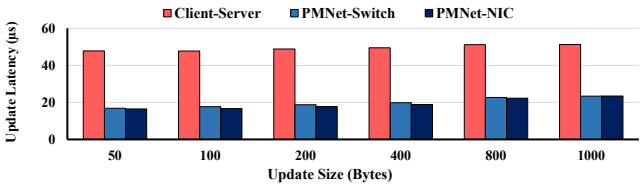

Fig. 15: Update latency of an ideal request handler with variable request sizes.

*2) **Evaluated workloads**:* We evaluate workloads from Intel's PMDK library [45] and a PM-optimized version of Redis from Intel [48]. We use a YCSB-like client [24] to generate and send read/update requests to the server. We also evaluate the performance of PMNet with two real workloads: a Twitter workload based on the Twitter Clone tutorial [92] and the online transaction processing benchmark, TPCC [104]. All server workloads manage persistent data in PM directly through the DAX-FS support. To support these workloads, we modified 11 and 7 lines of code in Redis and PMDK workloads, respectively. The payload of each read/update request is 100 Bytes, by default, unless specified otherwise. During evaluation, we skip the first 10k (warm-up) requests for more precise results.

*3) **Baseline protocol**:* We implement the driver program for PMDK workloads (B-Tree, C-Tree, RB-Tree, Hashmap, and Skip List) using UDP. Therefore, both the baseline and PMNet of these workloads use UDP. Redis, Twitter, and TPCC are originally based on TCP. Although UDP is faster, adapting them to UDP introduces a 9% slowdown due to the conversion overhead. Thus, we keep the original TCP-based communication as the baseline. This way, all the baselines are evaluated with their *best-performing* protocols.

*4) **Design points**:* We test PMNet under three system configurations for comparison.

- **PMNet-Switch:** A system with PMNet in the ToR switch of the server rack.
- **PMNet-NIC:** A system with PMNet as server's NIC.
- **Client-Server:** A baseline system which forwards all network packets to the destination.

B. Evaluation Results

*1) **Latency of microbenchmarks**:* We start with evaluating the raw latency and bandwidth of PMNet. In a practical system, the server-processing time can be the bottleneck. Therefore, we implement a microbenchmark with an *ideal request handler* on the server-side that acknowledges the client upon reception of the request, without processing it. Hence, the network latency becomes the primary bottleneck.

We first evaluate the latency benefit of PMNet. Figure 15 shows the round-trip time latency of PMNet-Switch and PMNet-NIC as we vary the payload size from 50B to 1000B, using a

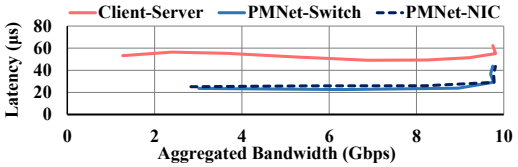

Fig. 16: Bandwidth vs. latency under stress test.

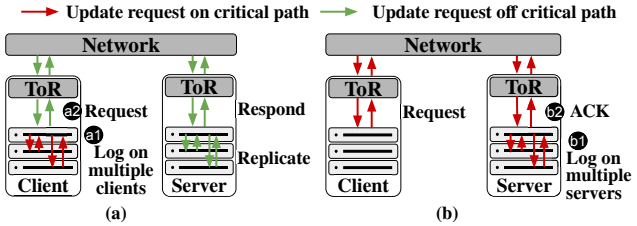

Fig. 17: Alternative designs: (a) client-side logging with replication and (b) server-side logging with replication.

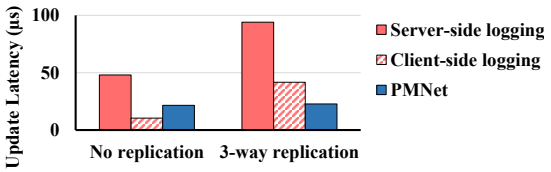

Fig. 18: PMNet vs. alternative designs: server-side logging and client-side logging.

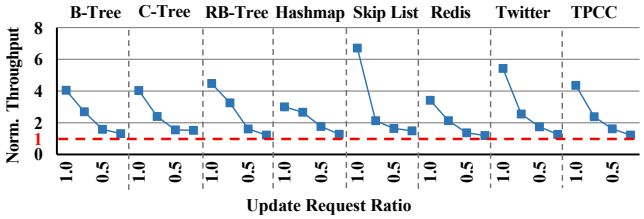

Fig. 19: Throughput normalized to Client-Server with variable update/read ratio.

single client. We make two observations from this figure. First, PMNet-Switch and PMNet-NIC provide 2.83× and 2.90× speedup, respectively, over the Client-Server with a payload size of 50B. However, the benefit decreases with larger payloads as the payload processing time goes up in the network device. For example, both PMNet-Switch and PMNet-NIC provide around 2.19× speedup compared to the Client-Server with 1000B payloads. Second, we observe that the difference in absolute latency between PMNet-Switch and PMNet-NIC is almost negligible (under 1 µs) as wire latency is low and the most benefit comes from moving the server processing and network latency off the critical path.

Next, we stress test the bandwidth using an ideal server-side request handler. On the client-side, we scale the number of client instances and keep sending 1000B requests to the server to saturate the bandwidth. Figure 16 displays stress testing results. First, both PMNet configurations and the Client-Server follow a similar trend where the latency remains the same when the total bandwidth is low, and there is a spike in latency when the bandwidth reaches the physical limit at 10 Gbps. Second, when the bandwidth is less than 10 Gbps, both PMNet-Switch and PMNet-NIC consistently have better latency than the Client-Server as they move the server off the critical path. As both PMNet configurations are equally effective in terms of update request response time and maximum bandwidth, in the next sections, we discuss PMNet performance using switches only.

*2) **Comparison with alternative designs**:* We compare PMNet with two alternative designs (request payload is 100B). To maximize the performance difference due to communication, we use the aforementioned microbenchmark (Section VI-B1).

***Client-side logging**:* (Figure 17a) locally logs the request and then lets the client proceeds (step a1). The client then forwards the request (step a2), such that the server's network stack and processing time are off the critical path. We implement client-side logging in a separate, dedicated software process, following a client-side persistent logging (caching) design [4]. The application directly overwrites the original socket interface to send logs to the client-side logger.

***Server-side logging**:* (Figure 17b) logs the request on the server upon reception (step b1) and immediately notifies the client (step b2), in order to move the server's processing time off the critical path. We implement the server-side logging following a persistent write logging (caching) design [56].

Figure 18 compares the latencies in these two designs. First, without replication, client-side logging is faster than PMNet as it does not go through the client's network stack (10.4 µs compared to PMNet's 21.5 µs). Whereas, server-side logging is much slower than PMNet because a large fraction of the server network latency remains on the critical path (47.97 µs). Second, with 3-way replication enabled, client-side logging becomes inefficient (41.61 µs) because it needs to communicate with other clients to replicate the logs. Similarly, server-side logging latency also increases significantly due to communication (94.02 µs). In comparison, PMNet consistently performs well even with replication (22.8 µs and 21.5 µs with and without replication), as the communication latency among replicas is off the critical path.

*3) **Application performance**:* Prior works have shown that the update/read ratio varies [7, 18]. Figure 19 shows the normalized throughput in real-world applications with PMNet, when we vary the update ratio from 100% to 25%. We make two observations: First, PMNet provides, on average, 4.31× speedup over the Client-Server with 100% update requests. Second, as the ratio of read requests increases, the throughput improvements from PMNet decrease. This trend is expected as PMNet focuses on update requests. We show the performance benefit of PMNet with read caching in the next section.

*4) **PMNet with read caching**:* Our read-cache implementation is based on "key" lookups using the GET/SET interface in the key-value store workloads. As a result, this experiment only includes the key-value store based workloads from PMDK [45] and Redis [48], and excludes workloads with complex queries,

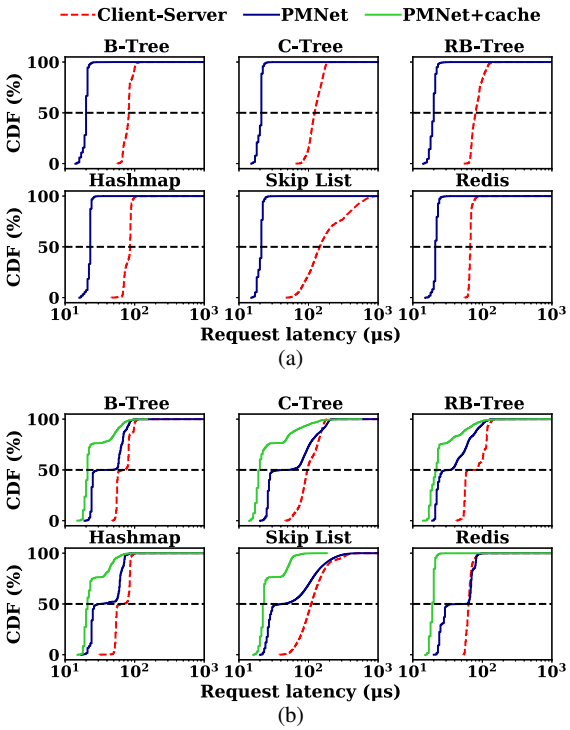

Fig. 20: CDF of request latency with (a) 100% and (b) 50% update request.

such as Twitter [92] and TPCC [104]. Figure 20a and 20b show the cumulative distribution function (CDF) of the latency with 100% and 50% update requests, respectively. We make three observations from this figure. First, the average latency of PMNet with caching is 3.36× lower than the Client-Server system. Second, when 50% of requests are updates, the latency of PMNet without caching has a noticeable transition point at the 50th-percentile (blue lines in Figure 20b), where latencies become close to the Client-Server system afterward. This happens because only half of the requests are updates and have been optimized by PMNet. In comparison, when 100% of the requests are updates (Figure 20a), the latency does not drop and provides 3.23× better tail latency than the Client-Server design. On the other hand, when caching is integrated into PMNet, the latency benefit does not stop at 50th-percentile but keeps continuing (green lines in Figure 20b)—as it serves *all* update requests and *most* read requests (cache hits) with a sub-RTT latency. Third, workloads with a higher hit rate (e.g., Redis) significantly benefit from the 99th-percentile latency with caching. We conclude that, with the integration of read caching, PMNet effectively reduces the latency of both update and read requests.

*5) **PMNet with replication**:* The replication scheme connects three PMNet switches in series to implement a 3-way replication in the network. Figure 21 shows the benefit of in-network replication compared to a Client-Server system that performs replication on the server-side. We make two observations. First, PMNet with replication provides 5.88×

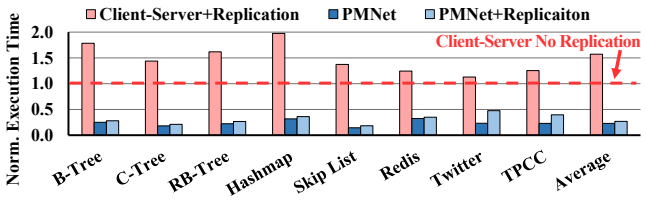

Fig. 21: Update request latency in a 3-way replication system (normalized to the no-replication Client-Server design).

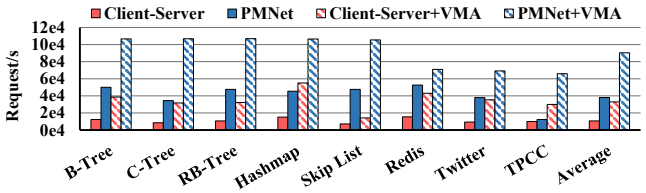

Fig. 22: Update throughput with optimized network stack.

better performance than server-side replication on average. Second, the overhead due to in-switch replication is low, as the latency of persisting logs is overlapped in our mechanism. The 3-way replication introduces 16% overhead over a PMNet system that only logs the updates once.

*6) **Recovering from server failures**:* We evaluate how a PMNet system recovers from server failures. To mimic failure, we manually cut off the power of the server and let PMNet resend logged requests after power has been restored. For each workload, we saturate the network bandwidth to create the worst-case scenario where PMNet has the maximum number of requests logged. On average, it takes 67 µs to resend a single request and 4.4 seconds to resend all pending requests in the log. Even in the worst case, the entire recovery procedure (resend + application recovery) only takes 9.3 seconds which is a small fraction of the server's 2∼3-minute boot-up time [43].

*7) **PMNet with an optimized network stack**:* The latency breakdown in Figure 2 (Section II-C) shows the client- and server-side latency consists of both network stack and processing time. With an optimized network stack, the network stack overhead can be significantly reduced. In this experiment, we use libVMA [77] to reduce the network stack time by moving network procedures into the user-space and avoiding the expensive context switching from the kernel. To evaluate an ideal scenario where the server-side overhead is minimal, we use the microbenchmark introduced in Section VI-B1. And, both the client and the server are running optimized network stacks. Figure 22 compares the update-throughput among four designs: Client-Server, PMNet, Client-Server + libVMA, and PMNet + libVMA. The result shows that, without libVMA, PMNet provides 3.08× better throughput. After applying libVMA, the server network stack overhead is significantly reduced. Nonetheless, the integration of PMNet still provides 3.56× better throughput. Although the speedup is lower—part of the server-side overhead has been reduced by libVMA—the benefit from PMNet is still significant as PMNet moves the remaining server processing time off the critical path.

VII. Discussion

In this section, we discuss the network bandwidth and PM performance, and alternatives to PMNet.

Reaching Higher Network Bandwidths: Fundamentally, PMNet supports higher bandwidths. First, the relatively slower PM access is decoupled from the network traffic by queuing the PM access in log queues. By increasing the log queue size according to the bandwidth-delay product (BDP) (Equation 1) of a high-bandwidth network, PMNet can buffer incomplete accesses to PM. To support a 100 Gbps network, only a 10 kbit (or 1.25 kB) log queue buffer would suffice. Second, the PMNet only needs to buffer ongoing update requests that have not been committed to the server. Only 500 Mbit (or 62.5 MB) of PM is needed to buffer the in-flight update requests in a 100 Gbps network (Equation 2).

PM Write Bandwidth: In our implementation, we use battery-backed DRAM on the FPGA board, that has a bandwidth of 2.5 GB/s, similar to the per-DIMM bandwidth of Intel's Optane PM. We expect that future PM technologies will enable much higher bandwidth, such as the higher-bandwidth battery-backed NVDIMMs [91], the emerging persistent cache [90], and alternative PM media (e.g., STT-RAM [58], ReRAM [113]). With a higher PM bandwidth, PMNet can handle higher update request bandwidth.

External Persistent Storage: We integrated PM into a network device. Alternatively, other types of storage can also maintain persistent data. For example, switches can access a network-attached PM device [102] (or SSD [78]) instead of keeping persistent data on-board. However, such designs add additional network latency to persist data in the network device, eventually inflating the critical path of client execution. Further, as our BDP calculations have shown (Section V-A), the PM capacity requirement is relatively small, and therefore, it is unnecessary to use an external device for persisting requests.

VIII. Related Work

In this section, we discuss the related works in PM systems, network optimization, and in-network compute.

Persistent memory systems: The integration of PM improves the performance of managing persistent data over conventional storage devices, such as SSD and HDD. Various software systems take advantage of PM for better storage performance. PM-optimized file systems allow existing programs to manage data efficiently on PM [23, 34, 115]. Distributed, PM-based file systems allow clients to access PM remotely while benefiting from PM's high performance [74, 94, 119]. There are also PM-optimized databases and key-value store applications that accepts requests from clients [8, 46]–[48, 61, 111]. Applications can also choose to manage data on PM without software indirections, by maintaining their own PM data structures [5, 19, 22, 42, 85, 105, 117]. However, even with a faster storage backend, the client still needs to go through the network and wait for the entire RTT to update persistent data on the servers. In comparison, the integration of PMNet can improve the performance by moving both network stack and processing time off the critical path.

Performance of the PM system is one of the major aspects, and correctness is another, as ensuring a consistent recovery in PM systems is hard and error-prone. Recent works have provided tools that ensure PM-based applications recover to a consistent state in event of a failure [33, 39, 71]–[73, 83]. These testing methods can be adapted to in-network data persistence systems, to validate not only the ordering in one application but also the persist ordering among clients and servers. Further, verification tools for programmable data plane [37, 68] can work in cooperation with PM testing tools and guarantee end-to-end correctness of a system with in-network data persistence. We leave this direction as a future work.

In-network compute: In-network compute reduces latency for a variety of tasks by moving computation off the server and into the network. Prior works proposed programmable switches for network functions such as load balancing [54, 79, 79] and packet scheduling [40, 98]. In addition, programmable switches can also offload part of application's logic into the network such as data aggregation [26, 93, 116], and machine learning [36, 66, 116]. In-network compute can also accelerate storage workloads through caching [51, 64, 69]. However, the persistence domain in these workloads is still limited to the server. A recent work NetChain [52] utilizes the storage capability of programmable switches; however, it only stores the coordination information rather than persisting data in the network. To maintain data persistence in the network, this work introduces PMNet that places PM on network devices, such as NICs and switches. We expect future works to further integrate other acceleration logic into PMNet to accelerate a wider range of applications.

IX. Conclusion

In this work, we propose in-network data persistence that *exposes* the persistence domain to the network to persist update requests with sub-RTT latency. We design a PM-integrated programmable network device, PMNet, that logs in-flight update requests, and moves the server network stack and processing time off the critical path. We implement PMNet in an FPGA-based programmable switch and NIC and evaluate them in a real system with a variety of workloads. Compared to the existing system, PMNet can improve the throughput of update requests by 4.31× on average, and the 99th-percentile tail latency by 3.23×.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work is supported by the Google fellowship program, NSF awards CCF-1822965 and CNS-2046066, and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

References

[1] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *OSDI*, 2016.

[2] M. Alian and N. S. Kim, "NetDIMM: Low-latency near-memory network interface architecture," in *MICRO*, 2019.

[3] M. Alizadeh *et al.*, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *NSDI*, 2012.

[4] D. Arteaga and M. Zhao, "Client-side flash caching for cloud systems," in *SYSTOR*, 2014.
[5] J. Arulraj *et al.*, "Bztree: A high-performance latch-free range index for non-volatile memory," *VLDB*, 2018.
[6] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *SIGMOD*, 2017.
[7] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.
[8] K. A. Bailey *et al.*, "Exploring storage class memory with key value stores," in *INFLOW*, 2013.
[9] Barefoot Networks, "Tofino." [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/
[10] J. Barr, "Now available – Amazon EC2 high memory instances with 6, 9, and 12 TB of memory, perfect for SAP HANA," 2018. [Online]. Available: https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-sap-hana/
[11] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, 2003.
[12] L. A. Barroso *et al.*, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*, 2018.
[13] L. Barroso *et al.*, "Attack of the killer microseconds," *Commun. ACM*, 2017.
[14] N. Boden, "Available first on Google cloud: Intel Optane DC persistent memory." [Online]. Available: https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory
[15] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *SIGCOMM*, 2013.
[16] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM*, 2014.
[17] N. Bronson *et al.*, "TAO: Facebook's distributed data store for the social graph," in *ATC*, 2013.
[18] S. Chen *et al.*, "TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study," *SIGMOD*, 2011.
[19] S. Chen and Q. Jin, "Persistent B+-Trees in non-volatile main memory," in *VLDB*, 2015.
[20] X. Chen *et al.*, "UDORN: A design framework of persistent in-memory key-value database for NVM," in *NVMSA*, 2017.
[21] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *ATC*, 2013.
[22] N. Cohen *et al.*, "Object-oriented recovery for non-volatile memory," *OOPSLA*, 2018.
[23] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.
[24] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.
[25] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *TOCS*, 2013.
[26] P. Costa *et al.*, "Camdoop: Exploiting in-network aggregation for big data applications," in *NSDI*, 2012.
[27] A. Daglis *et al.*, "Manycore network interfaces for in-memory rack-scale computing," in *ISCA*, 2015.
[28] A. Daglis *et al.*, "RPCValet: NI-drive tail-aware balancing of ms-scale RPCs," in *ASPLOS*, 2019.
[29] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.
[30] J. Dean *et al.*, "Large scale distributed deep networks," in *NeurIPS*, 2012.
[31] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, USA, 2004.
[32] Dell, "Dell EMC PowerEdge R740xd Intel Optane," 2019. [Online]. Available: https://principledtechnologies.com/Dell/PowerEdge-R740xd-Intel-Optane-1019.pdf?linkId=78228051
[33] B. Di *et al.*, "Fast, flexible, and comprehensive bug detection for persistent memory programs," in *ASPLOS*, 2021.
[34] S. R. Dulloor *et al.*, "System software for persistent memory," in *EuroSys*, 2014.
[35] D. Firestone *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *NSDI*, 2018.
[36] J. Fowers *et al.*, "A configurable cloud-scale DNN processor for real-time AI," in *ISCA*, 2018.
[37] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in P4 programs with assertion-based verification," in *SOSR*, 2018.
[38] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019.
[39] H. Gorjiara *et al.*, "Jaaru: Efficiently model checking persistent memory programs," in *ASPLOS*, 2021.
[40] J. Guo *et al.*, "An efficient packet scheduling algorithm in network processors," in *INFOCOMM*, 2005.
[41] A. Gupta *et al.*, "Sonata: Query-driven streaming network telemetry," in *SIGCOMM*, 2018.
[42] Q. Hu *et al.*, "Log-structured non-volatile main memory," in *ATC*, 2017.
[43] IBM, "How to reset/reboot server iMM interface." [Online]. Available: https://www.ibm.com/support/knowledgecenter/ST5Q4U_1.6.0/com.ibm.storwize.v7000.unified.160.doc/ifs_resetserverimminterface.html
[44] Intel, "Intel Optane DC persistent memory." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
[45] Intel, "Persistent memory programming." [Online]. Available: https://pmem.io/
[46] Intel, "Pmem-rocksdb."
[47] Intel, "PMSE - Persistent memory storage engine for MongoDB," 2018. [Online]. Available: https://github.com/pmem/pmem-rocksdb
[48] Intel, "Redis," 2018. [Online]. Available: https://github.com/pmem/redis/tree/3.2-nvml
[49] J. Izraelevitz *et al.*, "Basic performance measurements of the Intel Optane DC persistent memory module," *arXiv*, 2019.
[50] T. Jepsen *et al.*, "Packet subscriptions for programmable ASICs," in *HotNets*, 2018.
[51] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *SOSP*, 2017.
[52] X. Jin *et al.*, "NetChain: Scale-free sub-RTT coordination," in *NSDI*, 2018.
[53] A. Kalia *et al.*, "Datacenter RPCs can be general and fast," in *NSDI*, 2019.
[54] N. Katta *et al.*, "HULA: Scalable load balancing using programmable data planes," in *SOSR*, 2016.
[55] B. Kemme and G. Alonso, "Database replication: A tale of research across communities," *VLDB*, 2010.
[56] S. Kim *et al.*, "Request-oriented durable write caching for application performance," in *ATC*, 2015.
[57] R. R. Kompella *et al.*, "Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator," in *SIGCOMM*, 2009.
[58] E. Kültürsay *et al.*, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
[59] H. Kumar *et al.*, "High-performance metadata integrity protection in the WAFL copy-on-write file system," in *FAST*, 2017.
[60] Y. Kwon *et al.*, "Strata: A cross media file system," in *SOSP*, 2017.
[61] Lenovo, "Memcached-pmem," 2018. [Online]. Available: https://github.com/lenovo/memcached-pmem
[62] Lenovo, "Analyzing the performance of Intel Optane DC persistent memory in storage over app direct mode," 2019. [Online]. Available: https://lenovopress.com/lp1085.pdf
[63] A. Lerner *et al.*, "The case for network accelerated query processing," in *CIDR*, 2019.
[64] B. Li *et al.*, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *SOSP*, 2017.
[65] J. Li *et al.*, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *SOSP*, 2017.
[66] Y. Li *et al.*, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *MICRO*, 2018.
[67] Y. Li *et al.*, "Accelerating distributed reinforcement learning with in-switch computing," in *ISCA*, 2019.
[68] J. o. Liu, "P4v: Practical verification for programmable data planes," in *SIGCOMM*, 2018.
[69] M. Liu *et al.*, "DudeTM: Building durable transactions with decoupling for persistent memory," in *ASPLOS*, 2017.
[70] M. Liu *et al.*, "IncBricks: Toward in-network computation with an in-network cache," in *ASPLOS*, 2017.
[71] S. Liu *et al.*, "PMTest: A fast and flexible testing framework for persistent memory programs," in *ASPLOS*, 2019.
[72] S. Liu *et al.*, "Cross-failure bug detection in persistent memory programs," in *ASPLOS*, 2020.

[73] S. Liu *et al.*, "PMFuzz: Test case generation for persistent memory programs," in *ASPLOS*, 2021.
[74] Y. Lu *et al.*, "Octopus: An RDMA-enabled distributed persistent memory file system," in *ATC*, 2017.
[75] V. J. Marathe *et al.*, "Persistent Memcached: Bringing legacy code to byte-addressable persistent memory," in *HotStorage*, 2017.
[76] Markets and Markets, "Data center colocation market by type (retail and wholesale), end-user (smes and large enterprise, industry (bfsi, it & telecom, government & defense, healthcare, research & academic, retail, energy and manufacturing), and region – global forecast to 2022," 2017. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/colocation-market-1252.html
[77] Mellanox, "Messaging accelerator (VMA)," 2020. [Online]. Available: https://github.com/Mellanox/libvma
[78] C. Mellor, "Toshiba nudges direct-to-ethernet SSD towards the light," 2019. [Online]. Available: https://blocksandfiles.com/2019/08/14/toshiba-ethernet-ssd/
[79] R. Miao *et al.*, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *SIGCOMM*, 2017.
[80] Microsoft, "Asynchronous programming - c#." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/async
[81] A. Mirhosseini *et al.*, "Enhancing server efficiency in the face of killer microseconds," in *HPCA*, 2019.
[82] P. A. Misra *et al.*, "Managing tail latency in datacenter-scale file systems under production constraints," in *EuroSys*, 2019.
[83] I. Neal *et al.*, "AGAMOTTO: How persistent is your persistent memory application?" in *OSDI*, 2020.
[84] NetFPGA, "NetFPGA-SUME Virtex-7 FPGA development board." [Online]. Available: https://netfpga.org/site/#/systems/1netfpga-sume/details/
[85] D. Ongaro *et al.*, "Fast crash recovery in RAMCloud," in *SOSP*, 2011.
[86] J. Ousterhout *et al.*, "The case for RAMClouds: Scalable high-performance storage entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, 2010.
[87] J. Ousterhout *et al.*, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, 2015.
[88] G. Prekas *et al.*, "ZygOS: Achieving low tail latency for microsecond-scale networked tasks," in *SOSP*, 2017.
[89] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *SoCC*, 2012.
[90] A. Rudoff, "Persistent memory programming without all that cache flushing." [Online]. Available: https://www.youtube.com/watch?v=uLOjC6Kuokc
[91] A. Sainio, "NVDIMM - Changes are here so what's next?" 2016. [Online]. Available: https://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What%27s%20Next%20-%20final.pdf
[92] S. Sanfilippo, "antirez/retwis." [Online]. Available: https://github.com/antirez/retwis
[93] A. Sapio *et al.*, "In-network computation is a dumb idea whose time has come," in *HotNets*, 2017.
[94] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *SoCC*, 2017.
[95] N. K. Sharma *et al.*, "Evaluating the power of flexible packet processing for network resource allocation," in *NSDI*, 2017.
[96] N. K. Sharma *et al.*, "Approximating fair queueing on reconfigurable switches," in *NSDI*, 2018.
[97] J. Shute *et al.*, "F1: A distributed SQL database that scales," *VLDB*, 2013.
[98] A. Sivaraman *et al.*, "Programmable packet scheduling at line rate," in *SIGCOMM*, 2016.
[99] A. Sriraman *et al.*, "Deconstructing the tail at scale effect across network protocols," *ArXiv*, 2017.
[100] A. Sriraman and T. F. Wenisch, "µTune: Auto-tuned threading for OLDI microservices," in *OSDI*, 2018.
[101] Y. Sverdlik, "Study: Data centers responsible for 1 percent of all electricity consumed worldwide," 2020. [Online]. Available: https://www.datacenterknowledge.com/energy/study-data-centers-responsible-1-percent-all-electricity-consumed-worldwide
[102] T. Talpey, "SNIA nonvolatile memory programming TWG – Remote persistent memory," 2019. [Online]. Available: https://www.snia.org/educational-library/nonvolatile-memory-programming-twg-remote-persistent-memory-2019
[103] Y. Tokusashi *et al.*, "LaKe: An energy efficient, low latency, accelerated key-value store," *arXiv*, 2018.
[104] Transaction Processing Performance Council (TPC), "TPC-C." [Online]. Available: http://www.tpc.org/tpcc/
[105] S. Venkataraman *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory," in *FAST*, 2011.
[106] H. Volos *et al.*, "Aerie: Flexible file-system interfaces to storage-class memory," in *EuroSys*, 2014.
[107] Z. Wang *et al.*, "Characterizing and modeling non-volatile memory systems," in *MICRO*, 2020.
[108] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *SC*, 2011.
[109] X. Wu *et al.*, "NVMcached: An NVM-based key-value cache," in *ApSys*, 2016.
[110] Z. Wu *et al.*, "CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services," in *NSDI*, 2015.
[111] F. Xia *et al.*, "HiKV: A hybrid index key-value store for DRAM-NVM memory systems," in *ATC*, 2017.
[112] Xilinx, "X2 series Ethernet adapters." [Online]. Available: https://www.xilinx.com/products/boards-and-kits/x2-series.html
[113] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *HPCA*, 2015.
[114] J. Xu *et al.*, "Nova-Fortis: A fault-tolerant non-volatile main memory file system," in *SOSP*, 2017.
[115] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST*, 2016.
[116] F. Yang *et al.*, "SwitchAgg: A further step towards in-network computation," in *FPGA*, 2019.
[117] J. Yang *et al.*, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *FAST*, 2015.
[118] Y. Zhang *et al.*, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *ISCA*, 2016.
[119] Y. Zhang *et al.*, "Mojim: A reliable and highly-available non-volatile memory system," in *ASPLOS*, 2015.