Packets as Persistent In-Memory Data Structures

Michio Honda
University of Edinburgh
michio.honda@ed.ac.uk

ABSTRACT

Networked storage applications cannot fully benefit from fast persistent memory, because of *data management overheads* incurred to implement storage properties, such as integrity, consistency, search efficiency and flexibility. To address this problem, we explore a new approach that turns networking overheads into assets, repurposing the transport protocol and network stack features, some of which can be offloaded to the NIC hardware, for implementing the storage properties particularly for the PM devices.

CCS Concepts • **Software and its engineering → Operating systems**; • **Networks → Transport protocols**; • **Information systems → Data structures**;

Keywords Persistent memory, transport protocols

ACM Reference Format:
Michio Honda. 2021. Packets as Persistent In-Memory Data Structures. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3484266.3487386

1 INTRODUCTION

Storage systems serve as the primary repository of data that the users or operators cannot afford to lose. In the form of a file system, object store and relational or non-relational database, they support a number of systems and applications, such as web applications and data analytics systems. Since reliability is of importance, storage systems host their data while ensuring various *storage properties*, such as durability to survive system reboots, integrity against silent hardware or software failures, and consistency on concurrent requests and failure recovery. Storage systems also optimize their data structures to serve requests efficiently.

HotNets '21, November 10–12, 2021, Virtual Event, United Kingdom

ACM ISBN 978-1-4503-9087-3/21/11…$15.00
https://doi.org/10.1145/3484266.3487386

Persistent memory (PM), also known as non-volatile main memory, motivates researchers and practitioners to redesign system software. This is because it is not just a fast storage media, but provides a new access method, which allows the applications to access persistent data in byte granularity without system calls nor DRAM buffer cache. Unsurprisingly, file systems and databases have pioneered the use of PM, coping with those unique features to offer fast, reliable data store to the applications (§ 2.1).

Although PM has those attractive features, would it improve realistic deployment? In data centers, most machines are interconnected over a high-speed network fabric, and storage nodes are often *disaggregated* into rack-scale clusters [39, 24, 32], serving the computing nodes installed in other racks. Edge clouds, including Content Delivery Networks (CDNs), are deployed in close proximity to the residential or mobile clients, and directly connect to high bandwidth uplinks; the storage servers push the data at rates of millions of requests per second [1]. In either deployment, as both networks and storage devices are fast (e.g., several or tens of µs in latency) and request rates are high, individual servers have little CPU cycle or time budget to process a single request moved between storage and network devices.

Unfortunately, processing a request involves a number of steps throughout the host system. When a packet with a request arrives at the NIC from the network, the network stack processes it in the protocol implementations (e.g., IPv4 and TCP). The stack then links the payload data to the socket buffer of the receiver application, which is identified by the transport port number. The storage stack application, such as a key-value store, reads the data from the socket buffer and processes the data based on its own semantics; for example, it parses the data for a key-value pair and `get` or `put` command. If the request is a `put`, the application writes the data in its own persistent data structures with metadata, such as identifier, links to other data, checksum and timestamp, so that it can efficiently locate and retrieve the data even after a reboot. These steps over the network and storage stack overwhelm the CPU cycles thus diminish the performance benefit of the PM devices (§ 3).

Existing approaches to exploiting the performance of PM in the context of networked systems, such as Mojim [49], Octopus [27] and FileMR [45], rely on RDMA to eliminate networking latency and processing overheads, which thus leave more CPU cycles for the storage software stack or

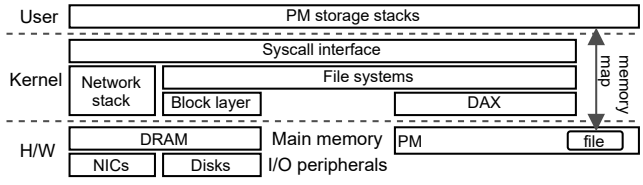

Figure 1: Networked PM host stack. A storage stack (§ 2.1), maps a PM-backed file in its address space.

application. However, the reliance on RDMA restricts deployment, even if the requirement of the lossless network fabric is obviated as in IRN [31]. For example, clients and network fabric also need to be replaced or reconfigured. Further, RDMA systems cannot serve the requests over the Internet, which is unacceptable for edge clouds.

In this paper, we propose an approach to turn networking features into *assets* to enable storage properties, instead of considering them as overheads, which is the case in RDMA-based approaches. We aim to *repurpose* the information and features available in the network protocols and stack, some of which can be accelerated by hardware offloading available in modern NICs, for implementing storage properties. We believe this is a right direction, because TCP stacks have been optimized and new reliable transport protocols with low-latency congestion control algorithms have been recently developed in software but with the possibility of being implemented in hardware [33, 13] (§ 5.2).

The key insight here is that *network packets and their representation in the network stack are flexible, efficient in-memory data structures accompanied by rich metadata designed to survive unreliable hardware.* Reliable transport protocols, such as TCP, attach checksum and timestamp to every segment; the network stack can also represent data that spans across multiple packets. These features could be useful particularly to manage the PM-backed data, which can be persistent without serialization. Both transport protocols and storage stacks have been designed to manipulate faulty, dumb network and storage hardware, respectively, but they have had to be implemented in isolation, because the former is in-memory data structures and the latter is on-disk ones. However, when data resides in PMs, its data structures resemble in-memory data structures, motivating us to unify the both.

2 BACKGROUND AND RELATED WORK

To better understand the overheads incurred in the end-to-end request processing cycle (§ 1), this section reviews the features and techniques in the storage and network stacks.

2.1 Persistent Storage Stacks

Persistent data structures have originally been designed for block I/O devices as file systems. Their main properties are naming, durability and crash consistency. The disk space is addressed by blocks, typically 4KB each, and divides into three sections: the data region that holds user data contents, inode table that maintains file metadata and pointers to data blocks, and superblock that identifies the filesystem itself. Popular design patterns include grouping a portion of the inode table and corresponding data blocks to reduce the expensive *seeks* on a spinning disk (e.g., FFS [30]) and appending writes to a sequential journal (or log) to defer or batch updating the primary data repository, which involves multiple disk access or writes (e.g., `ext3/ext4fs`). Database systems, which are another class of persistent storage stacks but run in the user space, manage their own data structures using similar techniques and `*sync` system calls.

PM devices have precipitated changes in both the storage software architecture and data structures. File systems use the DAX subsystem (Figure 1) to export a PM-backed file region to the application address space over `mmap`, so that the application can access persistent data using `load`/`store` CPU instructions without DRAM buffer caches nor `read`/`write` system calls (right arrow in the figure). To ensure persistence, the application flushes the CPU cache into PM using `clflush(opt)` instructions, which operate in a cache-line granularity, instead of `*sync` system calls, which operate in a block or page granularity. User-space PM storage stacks (top rectangle in the figure) exploit those features while solving PM-specific challenges. Some works tackle read and/or write efficiency, such as in-place update in a log-structured merge tree [19] and optimization of B+trees [4], and some tackle consistency problems caused by out-of-order CPU cache flushes [6, 46, 21].

Ensuring storage properties, such as consistency, integrity, and search and I/O efficiency is the primary source of the storage stack overheads. For example, in addition to persistence and consistency, file systems (e.g., `zfs`, `btrfs`, NOVA [44] and SplitFS [16]) and databases (e.g., pDPM [41], SLM-DB [17] and NoveLSM [19]) ensure integrity using checksums, and use metadata like timestamps. Software overheads matter in PM-based systems, because those are highlighted by the fast storage medium access. This is also why many PM storage systems rely on RDMA for networking [18, 45, 49, 27].

2.2 Network Stacks and Protocols

Network stacks have been improved since 2010, initially by raw packet I/O frameworks, namely PacketShader [9] and netmap [37], for user-space routers or middleboxes to support high packet rates over modern networks, 10 Gbit/s and beyond. Host TCP/IP stacks have been improved by better multi-core scalability (Affinity Accept [34]), new APIs for system call batching (MegaPipe [10]), streamlined datapath (Stackmap [47] and TAS [20]) and lightweight user-space stacks (mTCP [15] and Sandstorm [28]). Those stacks focus on networking only, assuming the RPC-like applications that do not serve persistent data.

Diskmap [29], Reflex [22] and i10 [12] enable new networked storage systems, but on block devices, which require on-disk data management (§ 2.1). PASTE [11] organizes packet buffers into a PM region, allowing applications to persist data without data copy. However, storage systems are also responsible for other properties, such as data organization optimized for target workloads, integrity to ensure that data is intact over time, consistency typically provided by transaction primitives, and timestamps.

Networked non-persistent in-memory key-value stores, such as MICA [26], eliminate networking overheads using kernel-bypass framework and custom UDP-based protocol. However, these systems need custom clients and do not support storage properties typically offered by persistent storage systems, such as durability and crash consistency.

3 NETWORKING AND STORAGE OVERHEADS

To understand the impact of the software overheads incurred to achieve storage properties, this section measures end-to-end latency of storage stacks over the network. We divide the overheads of networked storage systems into three: *networking*, *persistence* and *data management* overheads. Networking overheads consist of the network stack and protocol processing, in addition to the network fabric latency. Persistence overheads are the cost of storage medium access to persist the application data. We define data management overheads as the cost to achieve storage properties such as consistency, integrity and data layout for read or write efficiency or flexibility. Relative data management overheads are generally higher in PM-based systems than in disk-based systems, because persistence overheads are lower.

The most important goal of this section is to confirm that networking overheads are within the order of the other overheads in networked PM-based systems, and storage stacks impact end-to-end performance; if these assumptions go false, the TCP/IP networking would be unviable for the networked PM systems.

Methodology. To understand the degree of those three overheads, we measure the round-trip time (RTT) that the client sends a write request to a remote server with different storage properties and receives the application-level acknowledgement. The communication protocol is HTTP over TCP. We first measure networking overheads using the server application to simply discard the request and return the response as if the request was processed by the storage stack.

We then measure the RTT with a fully-fledged PM-optimized storage stack (NoveLSM, described next) that implements the storage properties of durability, integrity, consistency (on crash recovery and concurrent access) and searchability (e.g., efficient range query support), and thus incurs persistence and data management overheads. To obtain the persistence overheads, we also measure the RTT with the same storage stack but disabling the persistence operations by modifying the source code. Subtracting the networking-only RTT from this RTT indicates the data management overheads. We obtain the breakdown of the data management overheads by further modifying the storage stack to skip one or more logical operations.

Network and storage stack. To highlight those overheads in a state-of-the-art system, we use PASTE [11] for the network stack, because it achieves comparable performance to kernel-bypass stacks while using the matured kernel TCP/IP implementations that include modern TCP extensions, which are missing in most of the kernel-bypass stacks. We use busy-polling to optimize for latency and configure PASTE to process data in DRAM, because NoveLSM, introduced next, organizes data into PM.

We use NoveLSM [19] as the PM storage stack. NoveLSM extends LevelDB, which is a write-optimized persistent key-value store based on the log structured merge (LSM) tree, designed for both spinning and solid-state disks. LevelDB writes requests in *memtable*, which is a DRAM-backed table whose data is also persisted in the log backed by a disk. To optimize for PM, NoveLSM replaces memtable with PM-backed one without the log, which also implements new data structures for efficient search. Since we are interested in the data management in the PM, we configure NoveLSM to not move (*a.k.a.* compaction) the data to disks during the experiment. We implement checksum calculation in NoveLSM; although it is disabled in the current version of NoveLSM, it is enabled in LevelDB to ensure data integrity.

Hardware and other software. The server is equipped with 128 GB of RAM, 512GB of Intel Optane DC Persistent Memory with the App-Direct mode configuration, and two Xeon Gold 5218R CPUs clocked at 2.10Ghz. The client is equipped with two Xeon E5-2620v3 CPUs clocked at 2.40Ghz. Both machines connect to a switch with an Intel XXV710 25 Gbps NIC, and install well-tuned Linux kernel 5.9, disabling `netfilter`, debugging features, hyper threading, deep CPU sleep states and turbo boost. Both machines enable checksum offloading. The server uses only one CPU core throughout the paper; the client uses all the cores when multiple TCP connections are used. The client runs the regular Linux stack and `wrk` as the application to issue storage requests over one or more TCP connections and measure the end-to-end latency. We report average RTTs over 3s of continual requests.

Results. Table 1 shows the RTT and its breakdown of a 1KB write request. The networking-only RTT is 26.71µs. When the application stores the request in NoveLSM, the RTT increases to 34.79µs, including 6.39µs of the data management overheads and 1.94µs of persistence overheads. These results show that data management overheads are significant.

Within 6.39µs of the data management overheads, preparing the LevelDB-specific request data structure takes 0.7µs,

Overhead	Operation	Time [µs]	
Networking	TCP/IP & HTTP in client and server, and network fabric		**26.71**
Data mgmt.	Request preparation	0.70	
	Checksum calculation	1.77	
	Data copy	1.14	
	Buffer allocation and insertion	2.78	
			6.39
Persistence	Flush CPU caches to PM		**1.94**
Total			**34.79**

Table 1: Latency breakdown of RTT for a 1KB write request. The mismatch (0.25µs) of the total and the sum is because each of them is measured by separate requests.

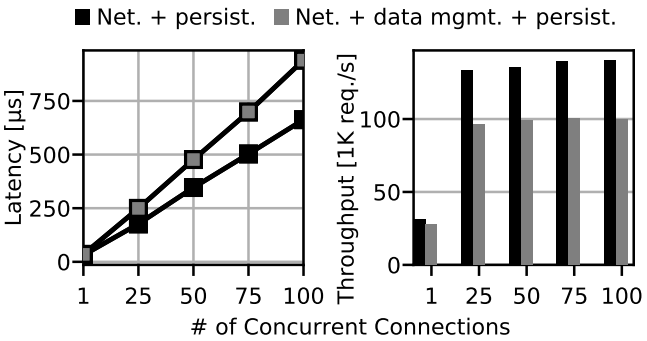

Figure 2: Latency and throughput of continual 1KB writes over parallel persistent TCP connections. Data management overheads reduce throughput by 9–28% and increase latency by 11–41%.

and calculating the checksum for the value takes 1.77µs. Inserting data involves allocating a PM-backed buffer, data copy and linking the buffer into the storage data structure, which is a persistent skip list in NoveLSM. This process takes 3.92µs including 1.14µs of data copy.

6.39µs of the data management overheads may seem low, but this is untrue. Figure 2 plots RTT and throughput with and without data management overheads over concurrent connections or requests, which mimic multiple clients. The former uses a simple application that copies and persists data in the PM region without NoveLSM (thus no data management overheads), and the latter uses NoveLSM. This measurement shows that the data management operations reduce throughput by 9–28% and increase latency by 11–42%.

The additional penalty in latency and throughput in comparison to the single connection case is due to the queue at the server application that processes the TCP connections with the requests in turn; an increased processing time (in the storage stack) delays subsequent requests to be processed. One might think the use of additional CPU cores, but in reality the server receives far more concurrent connections, resulting in a queue at each of the cores.

3.1 Implications

Those experiment results indicate that the data management cost on top of the persistence overhead is significant in the networked PM systems, even over regular TCP/IP networks. Although the current experiment shows the networking latency is still dominant, since both transport protocol and network fabric are improving in latency, we believe the networking latency will be reduced (§ 5.2). Further, the NICs keep improving; not only are they reducing latencies, but also they are implementing new offload capabilities, such as TLS encryption/decryption. Advanced NIC design will further accelerate these trends [7, 40].

In the byte-addressable PM devices, fewer opportunities of hardware offloading are available inside the storage device in comparison to block devices, which are implemented as I/O peripherals and often equipped with co-processors or acceleration engines [25, 38, 48]. The use of separate accelerator devices would not be an option, because data transfer to and from these devices over the PCIe bus could incur additional overheads. Further, extending NoveLSM to use the persistent buffer offered by PASTE could reduce the data copy overhead, which is 1.14µs (Table 1), but not other components in the data management overheads, which amount for 5.25µs.

Therefore, we argue that the most promising approach to reducing the data management overhead of PM storage stacks be to exploit the NIC or network stack features where all the storage data go through.

4 OPPORTUNITIES IN NETWORKING

The design goal of storage stacks is to enable a reliable, yet flexible data repository on top of the faulty, dumb storage devices, such as disks and PMs. These storage devices are faulty, because their data can corrupt either silently [2, 8] or with an error on access [3], and writes can be incomplete even after the device returns the completion signal to the software [23]. They are also dumb because they reorder flush commands, defying consistency guarantees [43, 5]. Storage stacks employ many techniques (§ 2.1) in software to offer reliability over such storage devices as efficiently as possible.

This motivation somewhat resembles the design goal of transport protocols and host network stacks. TCP was designed to offer reliability over the Internet that is faulty, because the packets can be lost or reordered, and dumb, because the networks do not perform many tasks other than delivering packets in a best-effort manner. Host network stacks also have been designed to cope with various network conditions and events efficiently, such as organizing packets arrived or acknowledged out-of-order, maintaining large data that the user wishes to send but does not fit into the path MTU, scheduling packet transmissions at the intended time, and sharing the packets between multiple consumers, such as receiver application and packet capture pseudo device.

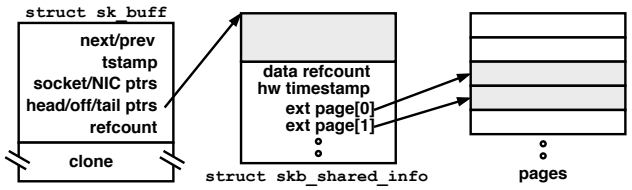

Figure 3: Linux network packet metadata (`sk_buff`). Gray parts indicate packet headers and data.

We therefore examine what and if networking features could be repurposed to implement storage properties. We are motivated particularly because more and more storage stacks are now going to be built on top of PM whose performance characteristics and access methods resemble those of DRAM, meaning that the data is organized into persistent in-memory data structures rather than on-disk ones (§ 2.1).

4.1 Packet Representation in Network Stacks

A packet arriving at the NIC or application data being written in the network stack is accompanied by a network metadata structure (`sk_buff` in Linux and `mbuf` in FreeBSD). Figure 3 illustrates the metadata that represents a single packet data. As shown in the figure, the packet data can span across multiple memory pages to accommodate arbitrary sized data. The metadata structure contains reference counts to the packet data and metadata, timestamps provided by either software or NIC, links to other metadata instances, and pointers to the vantage points in the packet data or kernel subsystems, including protocol headers, sockets and network devices.

Packet metadata can be stateful. For example, when the TCP implementation transmits a segment, it passes the packet metadata to the IP layer which further passes them down to the NIC driver. However, the TCP implementation needs to keep the packet until the segment is acknowledged by the receiver. This is achieved by the *clone* mechanism. A clone points to the same packet data as the original with an additional reference count (data refcount in the figure). Although the lower layer protocols and driver update or release the cloned metadata, the packet data is kept intact until the final reference, which is hold by the transmission queue of the TCP implementation, is released. Therefore, the TCP implementation can retransmit the packet in the event of packet loss. These features could be useful to share the data between the network stack and storage stack, because the storage stack must maintain the metadata and data independently from its transient state (e.g., in-transmission) in the network stack.

The packet metadata is released and headers are stripped when the data is passed to the application (e.g., `read`), which acts as a storage stack (Figure 1). The storage stack, including NoveLSM used in the experiment in § 3, then examines the data. If the request is a `get`, it searches for the database, otherwise it inserts the key-value pair to its private data structures, such as a persistent skip list, with its own metadata, such as checksum and timestamp. If the storage stack retrieved the data with packet metadata, many of these data management tasks could be obviated or simplified.

4.2 Enabling Storage Properties

How would the networking features useful to accelerate realistic PM storage stacks? We take a look at NoveLSM used in our case study in § 3 and file systems, and find several opportunities.

NoveLSM implements a persistent, mutable skip list in the PM to quickly find a key requested by a `get` request. This data structure is implementable using packet metadata, although some additional list entries may be needed, depending on the number of levels. Linked lists with packet metadata are common. For example, a socket buffer is a linked list of packet metadata. Also, a TCP receiver maintains out-of-order segments in a red-black tree to quickly identify and retrieve any of these segments when it receives a relevant in-order segment. This is a good example that exhibits flexibility of packet metadata to build an efficient in-memory data structure.

Further, LevelDB, which NoveLSM extends, computes the checksum of the data when storing the data. NoveLSM could obviate this task using the checksum values available in the TCP header that has carried the request over the network. As checksumming is known to be expensive, CompoundFS [36] proposes to use an in-storage embedded processor. However, since such a processor is unavailable in PM devices. Therefore, the use of TCP checksum computed by the NIC hardware appears a more promising option. This could save 1.77µs of the request processing (Table 1).

NoveLSM employs a user-space persistent memory allocator. It could obviate this allocator if the network buffer and metadata allocators are exported to the storage stack; as discussed in the previous subsection, a network metadata can point to one or more external buffers. If NoveLSM organized its data into the packet data structures, it could reduce the costs of sending data to the network, because it can avoid memory deallocation in its own allocator and memory allocation inside the network stack. This could be significant, as [36] reports high memory allocation overheads in both the user and kernel space with LevelDB.

To exploit those opportunities, we need to extend the network stack. The first step could be persisting the packet metadata in the PM device. We also need to locate them after a reboot, otherwise persisted data is useless. One straightforward method to enable persistent packet metadata is to extend PASTE to name and persist the packet metadata in a PM-backed file, in addition to the packet data. Further,

we need to define new APIs to share the packet metadata between the stack and application.

File systems designed for PM devices are another example that could benefit from persistent network metadata. Using persistent network metadata to ensure durability, consistency, timestamp and integrity, sending or receiving file data could be accelerated. File systems manage on-disk data using metadata (i.e., inode) that typically contains name, timestamp, checksum and links that point to either on-disk data blocks or other metadata (§ 2.1). Most of these information and structures can be achieved by packet metadata if allocated in a PM device. Therefore, current inode structures would be simplified, and packet metadata blocks will be maintained by the file system alongside inode blocks.

When the data is originated by the application (e.g., `write`), the file system would also allocate a packet metadata from its metadata block; this application data can be larger than a single packet size, because it can be split into multiple MTU-sized packets on network transmission, either by software (GSO) or hardware (TSO). Likewise, the file system could manipulate the NIC to allocate the packet metadata and data buffer from its packet-metadata and data blocks, respectively. Efficiently routing packets to an intended file could need hardware support, such as a programmable network chip or embedded CPUs in SmartNICs.

5 RESEARCH AGENDA

Enabling aforementioned networked storage stacks needs to address a number of research challenges.

5.1 Packet/Storage Metadata Structures

As we observed in the previous discussion, extending or redesigning packet metadata to be persistent is essential to upcycle packets for PM-based storage stacks, but it must be done carefully. It appears feasible to persist packet metadata. This is because, although access latency to a PM device is higher (346ns [14]) than DRAM (70ns), packet metadata is designed to be compact and cache friendly to minimize cache misses in the network stack data path, as the network stack needs to process millions of packets per second. However, at the same time, we may need further optimization, because the impact of a cache miss is higher than DRAM.

User-space TCP/IP stacks typically use smaller metadata than kernel ones to achieve high performance, at the expense of support for fewer network protocols and subsystems, such as modern TCP extensions and packet filters, than kernel stacks. Designing a cache-efficient packet metadata will be even more important in PM devices to low average and tail processing latency.

In addition to the size of metadata, ensuring crash consistency is also challenging. Although packet metadata will be persisted by the application (or intelligent NIC), their reference to external objects, such as socket and other packet metadata, must be kept consistent so that the storage data can be recovered after a reboot.

Finally, interaction between the storage and network stacks needs new abstractions beyond the POSIX APIs. Sharing packet metadata is relatively straightforward or could be done by memory mapping. However, sending or receiving packets will need new APIs. In FreeBSD, there exist in-kernel APIs that can pass the packet metadata that points to data to a network socket (e.g., `sosend`), which could be useful to implement as a system call interface.

5.2 NIC Offloading and Transport Protocols

Utilizing NIC offloading capabilities significantly improves networking performance, and if the offloaded processing results are also utilized by the storage stack, the benefit would be doubled. Checksum offload, TCP segmentation offload and hardware timestamps are straightforward examples. As discussed in § 4, some use cases, such as file systems, will need advanced NIC features, most likely those in SmartNICs, to route incoming packets to the specific PM address. Serialization at the NIC [35, 42] advances zero-copy in terms of reducing application data movement. Our work could complement it when used for PM by reducing metadata management overheads.

New transport protocols will further highlight the benefit of repurposing packets, because the networking latency, which is 26.71µs with TCP in our experiment (Table 1), will be lower. The Linux kernel implementation of Homa [33], a new transport protocol specifically designed for data center networking, uses regular Linux packet metadata and TCP headers to exploit the NIC offloading feature with unmodified drivers. This implies that the approach of repurposing the networking features is feasible not only for TCP but also future transport protocols.

6 CONCLUSION

This paper characterised networking, persistence and data management overheads of a PM-based storage stack in TCP/IP networking. We identified that data management overheads are significant, and thus proposed a new approach to reducing them by repurposing the networking features available in the network protocols and stacks to implement storage properties, such as integrity, consistency, flexibility and search efficiency. We believe this is an important step for storage systems to benefit from PM devices in regular IP-based networks without relying on RDMA.

ACKNOWLEDGMENTS

I am grateful to anonymous HotNets reviewers. This work was in part supported by EPSRC grant EP/V053418/1.

REFERENCES

[1] Joao Taveira Araujo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the edge: transport affinity without network state. *USENIX NSDI*. (April 2018).

[2] Lakshmi N Bairavasundaram et al. 2008. An analysis of data corruption in the storage stack. *USENIX FAST*. (February 2008).

[3] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. *ACM SIGMETRICS*.

[4] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, (February 2015).

[5] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. *ACM SOSP*.

[6] Joel Coburn et al. 2011. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM ASPLOS*.

[7] Alex Forencich, Alex C Snoeren, George Porter, and George Papen. 2020. Corundum: an open-source 100-gbps nic. *IEEE FCCM*. IEEE.

[8] Laura M. Grupp et al. 2009. Characterizing flash memory: anomalies, observations, and applications. *IEEE/ACM MICRO*.

[9] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM*.

[10] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. Megapipe: a new programming interface for scalable network i/o. *USENIX OSDI*.

[11] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: a network programming interface for non-volatile main memory. *USENIX NSDI*. (April 2018).

[12] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP≈RDMA: cpu-efficient remote storage access with i10. *USENIX NSDI*. (February 2020).

[13] Stephen Ibanez et al. 2021. The nanopu: a nanosecond network stack for datacenters. *USENIX OSDI*. (Jul. 2021).

[14] Joseph Izraelevitz et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*. arXiv: 1903.05714.

[15] Eun Young Jeong et al. 2014. Mtcp: a highly scalable user-level tcp stack for multicore systems. *USENIX NSDI*.

[16] Rohan Kadekodi et al. 2019. Splitfs: reducing software overhead in file systems for persistent memory. *ACM SOSP*.

[17] Olzhas Kaiyrakhmet et al. 2019. Slm-db: single-level key-value store with persistent memory. *USENIX FAST*. (February 2019).

[18] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. *ACM SoCC*.

[19] Sudarsun Kannan et al. 2018. Redesigning lsms for nonvolatile memory with novelsm. *USENIX ATC*. (Jul. 2018).

[20] Antoine Kaufmann et al. 2019. Tas: tcp acceleration as an os service. *ACM EuroSys*.

[21] Wook-Hee Kim et al. 2016. Nvwal: exploiting nvram in write-ahead logging. *ACM ASPLOS*.

[22] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash ≈ local flash. *ACM ASPLOS* number 1.

[23] Andrew Krioukov et al. 2008. Parity lost and parity regained. *FAST*.

[24] Jialin Li et al. 2020. Pegasus: tolerating skewed workloads in distributed storage with in-network coherence directories. *USENIX OSDI*. (November 2020).

[25] Shengwen Liang et al. 2019. Cognitive SSD: a deep learning engine for in-storage data retrieval. *USENIX ATC*. (Jul. 2019).

[26] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. *USENIX NSDI*. (April 2014).

[27] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an rdma-enabled distributed persistent memory file system. *USENIX ATC*.

[28] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network stack specialization for performance. *ACM SIGCOMM*.

[29] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|crypt|net: rethinking the stack for high-performance video streaming. *ACM SIGCOMM*.

[30] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for unix. *ACM Trans. Comput. Syst.*, (August 1984).

[31] Radhika Mittal et al. 2018. Revisiting network support for rdma. *ACM SIGCOMM*.

[32] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: isolation and sharing in disaggregated rack-scale storage. *USENIX NSDI*.

[33] John Ousterhout. 2021. A linux kernel implementation of the homa transport protocol. *USENIX ATC*. (Jul. 2021).

[34] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving network connection locality on multicore systems. *ACM EuroSys*.

[35] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of champions: towards zero-copy serialization with nic scatter-gather. *Proceedings of the Workshop on Hot Topics in Operating Systems*.

[36] Yujie Ren, Jian Zhang, and Sudarsun Kannan. 2020. Compoundfs: compounding i/o operations in firmware file systems. *USENIX HotStorage*. (Jul. 2020).

[37] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet i/o. *USENIX ATC*.

[38] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: designing in-storage computing system for emerging high-performance drive. *USENIX ATC*. (Jul. 2019).

[39] Vishal Shrivastav et al. 2019. Shoal: a network architecture for disaggregated racks. *USENIX NSDI*. (February 2019).

[40] Brent Stephens, Aditya Akella, and Michael M. Swift. 2018. Your programmable nic should be a programmable switch. *ACM HotNets*.

[41] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: an exploration of passive disaggregated key-value stores. *USENIX ATC*. (Jul. 2020).

[42] Adam Wolnikowski et al. 2021. Zerializer: towards zero-copy serialization. *Proceedings of the Workshop on Hot Topics in Operating Systems*.

[43] Youjip Won et al. 2018. Barrier-enabled IO stack for flash storage. *USENIX FAST*. (February 2018).

[44] Jian Xu and Steven Swanson. 2016. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. *USENIX FAST*. (February 2016).

[45] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. Filemr: rethinking RDMA networking for scalable persistent memory. *USENIX NSDI*. (February 2020).

[46] Jun Yang et al. 2015. Nv-tree: reducing consistency cost for nvm-based single level systems. *USENIX FAST*. (February 2015).

[47] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. Stackmap: low-latency networking with the os stack and dedicated nics. *USENIX ATC*.

[48] Teng Zhang et al. 2020. Fpga-accelerated compactions for lsm-based key-value store. *USENIX FAST*. (February 2020).

[49] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: a reliable and highly-available non-volatile memory system. *ACM ASPLOS*.