PASTE: Network Stacks Must Integrate with NVMM Abstractions

Michio Honda
NEC, NetApp[†]

Lars Eggert
NetApp

Douglas Santry
NetApp

ABSTRACT

This paper argues that the lack of explicit support for non-volatile main memory (NVMM) in network stacks fundamentally limits application performance. NVMM devices have been integrated into general-purpose OSes by providing familiar file-based interfaces and efficient byte-granularity access by bypassing page caches. However, this powerful property cannot be fully utilized unless network stacks also support it and applications exploit such support. This requires a thoroughly new network stack design, including low-level buffer management and APIs. We propose such a new network stack architecture to support NVMM and demonstrate its advantages for efficient write-ahead logging, a popular technique to implement transactions.

1. INTRODUCTION

Non-volatile main memory (NVMMs) [25][1] has the potential to change the way modern systems are designed and implemented. The memory hierarchy, with CPU registers at the top and persistent storage at the bottom, has changed little since the early 1970s. The media available at the bottom of the hierarchy, i.e., block-based persistent storage, has grown to offer a wider spectrum of choices, but ephemeral DRAM has ruled supreme as main memory. The ephemeral nature of DRAM has profoundly affected the design of the storage stack and all of its clients, from the humble word processor to transactional database systems. A great deal of thought has been given by the storage community to how NVMM will change the storage stack [6, 32, 29], but the thinking has been done in isolation from application logic and without regard to the implications for the complete hierarchy.

[†] Work done while at NetApp.

[1] We define NVMM as byte-addressable memory that is persistent, connected to the memory bus and directly addressable by the CPU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets-XV, November 09 - 10, 2016, Atlanta, GA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4661-0/16/11. . . $15.00

DOI: http://dx.doi.org/10.1145/3005745.3005761

Durable main memory will precipitate sweeping changes to how systems are designed end-to-end; it is a mistake to confine one's thinking about the ramifications of NVMM to the storage system in isolation. The entire processing cycle of an application will change, and storage will become inextricably intertwined with application logic, instead of maintaining the clean separation offered by the kernel POSIX API today. This paper addresses this space by examining the ramifications of NVMM from the perspective of an application—not the storage system—and offers a means of leveraging NVMM from the earliest stage of a server's request cycle. In particular, this paper addresses the following question: *what should the end-to-end data path—across a NIC, the network stack, application and a persistent data store—look like?*

Consider a transactional data transfer. The NIC on the receiver writes an incoming packet to main memory via DMA, then a network stack processes the packet. The application then reads the packet data (if the socket API is used, this involves a data copy) and processes it. Processing a transaction can result in side-effects to persistent data structures (e.g., adding a row to a table). The semantics typically require the application to accept and persist a transaction prior to acknowledging it as successful. As persistence is required, and updating the primary data structure on disk (e.g., in a database table) is very slow, it is common practice to use a write-ahead log to speed up transaction processing. Write-ahead logs are much faster than updating the primary data structure, as it simply involves serially appending to a log of accepted transactions. The primary data structure is persisted periodically and log entries are discarded. Accepting a transaction thus involves updating the primary data structure in memory (but not pushing it to disk) and copying data to a write-ahead log.

With NVMM, a transaction could in principle become durable when the NIC DMA's the data to host memory, rather than after an explicit data copy to a write-ahead log by the application. The data copy is particularly problematic in systems with NVMM, because it introduces latency and pollutes the CPU caches. Once the persistence tier is shifted from block devices with microsecond or millisecond order access times to NVMM devices with latencies of tens of nanoseconds, the CPU caches could be used more aggressively.

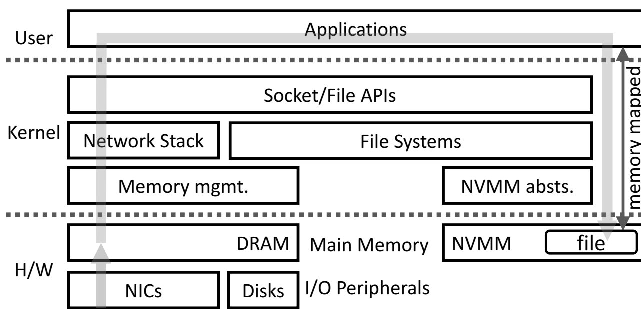

Figure 1: Today's OS organization: No integrated management between network stack and NVMM abstractions. Arrow indicates path of data movement from NIC to NVMM.

However, since there is no integrated management between NVMM abstractions and the network stack, packet data and logging are treated separately. This leads to superfluous copying by applications (see Figure 1).

Today, since the end-to-end latencies of transactional data transfers are dominated by slow block-device I/O (even for PCIe-attached SSDs) the impact of network stack performance is negligible. However, when applications store their write-ahead logs on NVMM the time scales are such that they become sensitive to both networking and storage stack performance (see Section 2).

This paper proposes a new network stack architecture called PAcket STorE (PASTE). PASTE places static packet buffers into an NVMM region, statically named by a file so that they can be located across OS reboots. Therefore, applications can locate packet buffers across reboots using private metadata that *point* to arbitrary packet data. Network buffers are not recycled by the network stack until the owning application gives the stack permission. Thus, write-ahead logs can be built directly from network buffers, eliminating the superfluous data copies inherent in today's transactional systems.

The reminder of this paper is organized as follows: Section 2 analyzes the costs of durably storing data from an end-to-end perspective; Section 3 explores a new network stack architecture that exploits the NVMM abstractions of modern OS kernels and defines an APIs for applications or data storage middleware. Section 5 discusses impact of NVMM latency, possibility to apply PASTE to kernel-bypass network stacks, and applications that could benefit from PASTE. Section 6 describes related work. The paper concludes with Section 7.

2. MOTIVATION

2.1 End-to-End Transaction Latencies

To motivate the proposed reorganization of the network stack, this section explores a case study of a transactional data transfer, an essential operation in many networked storage systems, such as blob stores [24, 26, 3], key-value stores [7, 18, 9] and databases [4, 14, 1]. A general transactional data transfer consists of following steps:

1. A client transmits data to a server.
2. The server receives packets at a NIC.
3. The NIC DMAs the packets to memory.
4. The packets are processed by the network stack.
5. A server application reads the data.
6. The server application durably stores a record of the transaction (e.g., on an SSD).
7. The server application replies to the client; the client now knows the transaction has been accepted and persisted.

Step 6 is where the largest contribution to end-to-end latency comes from. As discussed above, applications frequently use a write-ahead log to speed up transaction persistence, so that the client-perceived transaction commit time is minimized.

Today, logs are implemented as files and are updated with the `write()` followed by `fsync()` or `fdatasync()` system calls (the latter differs only in that it does not update file metadata, so is faster). As NVMM becomes available, applications will migrate away from using system calls and access NVMMs directly. File systems can be installed on the NVMM much as they are today for RAM disks—except the contents will survive reboots and power failures. Applications `mmap()` files directly into their address space and access their data directly with unprivileged CPU load and store instructions. System calls will be far too slow in comparison. Thus, accessing storage in this new world will be 3–6 orders of magnitude faster than it is today.

To better understand the impact of logging on total latency, we wrote a simple HTTP server that implements three methods to durably log data:

1. `write()` from a buffer into which a previous `read()` has written data, followed by either `fsync()` or `fdatasync()`
2. `memcpy()` from the same buffer to an `mmap()`-ed file, followed by `msync()`
3. preceding `read()` that has already stored data to the `mmap()`-ed file followed by `msync()`

The last method merges step 5 and 6 of a general transactional data transfer, avoiding one of the two data copies that occur otherwise. The data movement of the last method is depicted as an arrow in Figure 1.

We employ two types of persistent media: a PCIe-attached SSD (Samsung 950 Pro 256GB) and an NVMM that is emulated through a reserved region of DRAM managed by the NVDIMM driver. Both are formatted with the XFS file system that supports the Linux page-cache bypass mechanism, DAX [22] (NVMM absts. in Figure 1).

Emulating NVMM with DRAM is reasonable, since some NVMMs offer latency on the order of DRAM. They normally operates on a DRAM front-end; upon power loss, they use the capacity of an integrated battery to flush all content to solid-state back-end memory. This type of NVMM has been standardized as NVDIMM-N [16] and has been available since early 2016; HP sells an 8 GB NVDIMM-N at approximately $900 [12]. We explore other types of NVMM in Section 5.

Memory	Measurement	Time [µs]
—	Network only (H/W, stack, HTTP)	39.86
NVMM (emul.)	Network + read()/msync()	41.76
	Network + memcpy()/msync()	41.13
	Network + write()/fdatasync()	41.50
	Network + write()/fsync()	59.19
SSD	Network + write()/fdatasync()	2030
	Network + write()/fsync()	4510

Table 1: End-to-end transaction latency: NVMM nearly eliminates log latency.

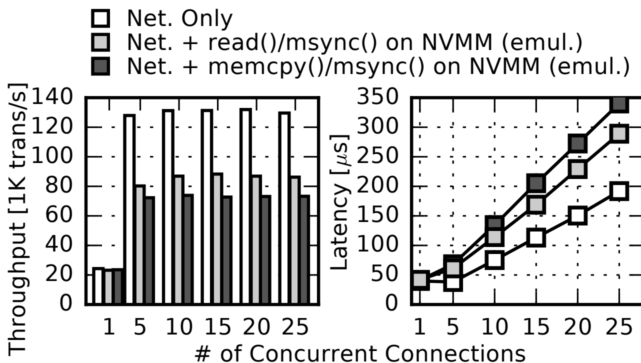

Figure 2: Throughput (left) and transaction latency (right) for concurrent requests and connections: Durably storing data still significantly reduces throughput and increases end-to-end transaction latency.

On the client, we instrument `wrk`, a popular HTTP benchmark tool, to send 1135 B HTTP POSTs. The HTTP OK returned by the server is 127 B long. The server and client are equipped with intel Xeon E5-2680 CPUs clocked at 2.8 GHz and have 64 GB of quad-channel DDR3 RAM. They are connected by Intel 82599 10 G NICs. Both run Linux kernel 4.6. The server uses a single CPU core.

Table 1 shows end-to-end transaction latency that `wrk` reports. Storing data on emulated NVMM is almost two orders of magnitude faster than on an SSD. In addition to the network-only latency (top row in Table 1), persisting data using `read()/msync()` and `memcpy()/msync()` only adds 1.90 µs and 1.27 µs, respectively. Note that systems on block devices typically use `write()/fdatasync()` to ensure durability because of atomicity. But for rest of discussion, we explore `memcpy()/msync()` and `read()/msync()`, because they highlight the costs of persisting data due to the fewer number of system calls and/or data copies. `memcpy()` would also suit the needs of user-level data management to achieves atomicity and thread safety [5, 30].

Unfortunately, the costs of durably storing data are still high. Figure 2 plots throughputs and transaction latencies for concurrent requests over parallel TCP connections, which is a common situation for busy servers. Because the processing of requests is serialized at each CPU core, this significantly increases transaction latencies and reduces throughput. Note that while our experiments are using a single CPU core, real deployments could serve similar or larger numbers of connections or requests on each core.

2.2 CPU Cache Misses

To shed light on the overheads of persisting data, we measure CPU cache misses using the Linux `perf` tool. Table 2 summarizes the results. The reported numbers include references and misses at all CPU cache levels, and are averaged over ten second measurements for ten parallel requests over the same number of TCP connections.

When we do not persist data, cache misses are very rare, and mostly happen during network stack processing, including constructing a packet representation structure (i.e., `sk_buff`), as well as IPv4 and TCP processing. Note that although cache-miss rates might seem too low, this result is due to Direct Data I/O (DDIO) that the NIC uses to directly store data in the lowest-level cache of the target CPU core.

In contrast, when we do persist data, cache-miss rates increase, although the number of cache references remains similar. 98–99% of them are caused by moving data to the emulated NVMM, which happens at the `read()` step for the `read()/msync()` case and during the `memcpy()` step for the `memcpy()/msync()`, respectively. In the latter case, moving data to a temporary buffer (i.e., `read()` to the source address of `memcpy()`) does not cause many cache misses, because the destination is always the same and thus remains cached. Moving data to the emulated NVMM is always done to a different location when persisting data, causing cache misses. Therefore, we conclude that moving data *in order to persist data* comes at a cost.

Note that our CPU is a relatively high-end one costing approximately $1700 with a relatively large 2.5 MB of L2 cache and 25 MB of lowest-level cache. Therefore, we expect higher cache miss rates with lower-end CPUs.

3. PASTE

The discussion in the previous section clearly shows that in the world of NVMM, end-to-end transaction latencies are affected by both networking and persisting data. In order to improve both of these cost components, we design an efficient network stack that transactional applications can use to implement more efficient systems.

3.1 Network Stack Requirements

First and foremost we must avoid superfluous copying of data, as we have observed there are significant costs attached to such operations, which points towards performing DMA directly into NVMM. However, this is just a starting point. The object is to leverage the fact that NVMM is persistent, and moving data from the NIC to NVMM introduces the opportunity to avoid later copies for persistence. Today's network stacks dynamically allocate packet buffers from a kernel pool of dynamic memory, which are thus *anonymous*. If the packet buffers themselves are to become the persistent

	Network only	Network + memcpy()/msync()	Network + read()/msync()
Cache references [1/sec]	18 292 550	19 265 789	18 031 136
Cache misses [1/sec]	65	850 028	1 504 711
Percentage	0.0004 %	4.4121 %	8.3451 %
Top cache-miss contributor	net_rx_action()	__memcpy_sse2_unaligned()	sys_read()
Percentage	84 %	98 %	99 %

Table 2: Cache misses on transaction processing: Moving data to emulated NVMM is the major contributor.

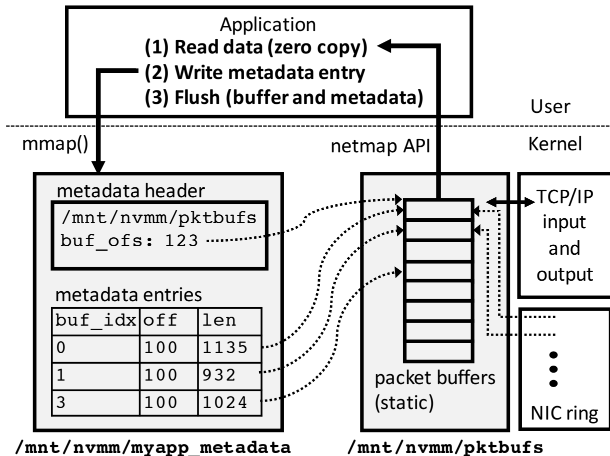

Figure 3: PASTE architecture: Packet buffers are named by a file on NVMM and pointed by application's private metadata.

version of data, then the first requirement is to ensure that such buffers can be found across system reboots and crashes.

To that end, packet buffers must become *named*; we use the well understood and supported *file* abstraction for this purpose. Using files is much more convenient compared to managing the physical addresses of NVMMs directly. This means that a file must be created and NVMM pages allocated for its contents. The network stack must then statically allocate its packet buffers in the physical pages that the file contains. Further, since the application needs to manage and access packet data with its private metadata, the network stack must provide a good representation of individual packet buffers.

3.2 Architecture

Figure 3 illustrates the PASTE architecture and how it stores data. PASTE "pins" a region of NVMM by creating a file (`/mnt/nvmm/pktbufs` in Figure 3). File systems can support NVMM with DAX [22], the Linux kernel NVMM abstraction that also enables file-system buffer cache bypass for `mmap()`-ed regions (Figure 1) [2].

PASTE then enhances the network stack to allocate the packet buffers in the region that this file contains. Applications employ APIs to directly access these packet buffers. The data in a packet buffer is named by a tuple of the form <buffer index, offset, length> so that applications can point particular data easily in their metadata (`/mnt/nvmm/myapp_metadata` in Figure 3).

The netmap framework [28] enables this organization with the StackMap extension [34] that integrates the kernel TCP/IP implementation[3]. The netmap framework implements pointers to packet buffers as relative offsets from a base address. The only necessary extension to netmap is to allocate packet buffers in an NVMM region indicated by a file name, instead of a region allocated by the kernel dynamic memory subsystem.

3.3 Durably Storing Data

PASTE is designed to work as follows: The network stack notifies the application when data arrives and is made available after TCP processing. The application then examines the data. A quick digression into modern DMA is required to understand what follows this step.

Modern NICs DMA packets into main memory *logically*, whereas *physically* they are placed into the lowest-level CPU cache. Thus, even if NVMM is backing the physical DMA target address, the contents of a packet are *not* persistent after a DMA. Thus, PASTE must *explicitly* push a packet to NVMM after a DMA to be certain that it is made persistent. Because this operation is costly, we do not want to perform it for every packet. Instead, the application examines a packet while it is still in the CPU cache and decides, based on the type of data it contains, whether to persist the contents to NVMM. On the current generation of Intel CPUs, this can be done with the `clflush` machine instruction.

Applications must distinguish between request data that should be persisted and request data that may remain ephemeral. Requests that are idempotent, such as SQL `select` queries, do (by definition) not have side-effects and need not be persisted. Mutable transactions that must be logged must be persisted (e.g., inserting a row in a table). When an application identifies such a transaction, portions of the packets (bytes in the TCP stream) are pushed to NVMM and made persistent. It is trivial to implement a write-ahead log based on this primitive. A linked list with entries pointing to requests inside the packet buffer can be superimposed onto them. The result is a write-ahead log that is temporally ordered and serves the same purpose as the journals stored on block devices today—but in a much faster fashion.

Applications can store their own write-ahead log in their own NVMM-backed file (`/mnt/nvmm/myapp_metadata` in Figure 3). In our example, the nodes of the linked list that comprise the log can be stored in said file, while the data

[2] For simplicity of packet buffer management, we would like the file to be laid out as a single and physically contiguous region of NVMM. The XFS file system "stripe unit" enables this.

[3] StackMap extends the netmap API, so that applications can traverse buffers in TCP byte stream order while skipping TCP/IP headers. See [34] for details.

they point to are in /mnt/nvmm/pktbufs. The primary data structure (not shown in the figure), such as a database table, may also be stored in NVMM, or stored on a block device (at much lower cost per byte) and updated at leisure (the objective of a write-ahead log is to mask the cost of updating a primary data structure and permitting faster responses to waiting clients). Periodically, the primary data structure is updated to reflect the write-ahead log and the log contents can be safely discarded. At this point the packet buffers can be recycled.

4. EXPERIMENTAL EVALUATION

This section quantifies the potential effects of PASTE. As discussed in Section 3.2, we extended the netmap framework [28] to allocate packet buffers in a file backed by NVMM, and use StackMap [34] for TCP/IP processing.

Recent fast network stacks, such as mTCP [35], IX [2], StackMap and Fastsocket [20] improve end-to-end latency by themselves, because of techniques like batching, run-to-completion, zero-copy APIs and lightweight buffer management. However, since Section 2 has observed that storing data durably comes at a significant cost in, we expect that accelerating *only* the network stack has a marginal effect. We are thus interested in what efficiencies can be realized by integrating the network stack with the application logic in the presence of NVMM.

In order to distinguish the effects of PASTE from the improvements introduced by an enhanced network stack, we use StackMap as a baseline for comparison by extending the server application used in Section 2 to run on top of it. The baseline simulates a purely application-managed write-ahead log that stores data independently using memcpy() followed by a clflush instruction, which is slightly faster than a msync() system call. We then compare these results with those obtained by PASTE, where the network stack stores packets in a file backed by an emulated NVMM and we replace the application write-ahead log with one built based on the mechanism described in Section 3.3. We use the same hardware setup and client application including message sizes of HTTP POST and OK as during the experiments described in Section 2.

Figure 4 shows our results and verifies our expectations[4]. StackMap already improves performance by 14 to 187 % in throughput and 13 to 65 % in average latency, as illustrated by comparing the baseline numbers (white bars and boxes) in Figure 4 against those in Figure 2. However, persisting data without an integration with the network stack decreases throughput by 17 to 61 % and increases average latency by 22 to 161 % (light gray bars and boxes in Figure 4)[5].

[4] The slightly higher latencies in the single-connection case compared to that with five connections is due to the power-saving feature of the client CPU.

[5] Since we use the netmap API, we examined batching clflushes over multiple mutable requests. But we did not observe a meaningful effect, because clflush works at a cache-line granularity.

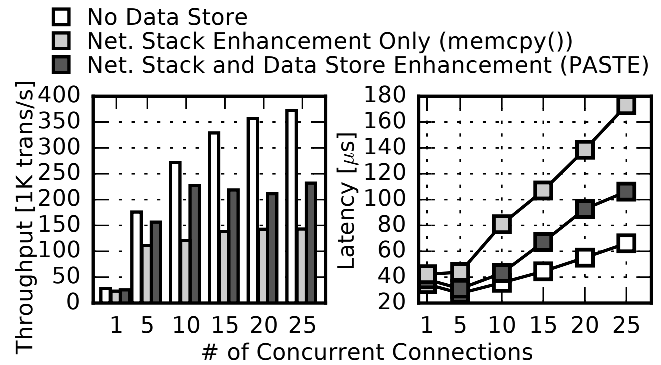

Figure 4: Throughputs (left) and transaction latency (right) with and without the PASTE durable data store. (Network stack uses StackMap in all the cases.)

On the other hand, persisting data with PASTE significantly improves throughput by 10 to 88 % and decreases average latency by 9 to 46 % (dark gray bars and boxes in Figure 4). There is, however, still a cost associated with persisting data with PASTE, which underscores the importance of persisting only those packet buffers that contain mutable data. The indiscriminate persistence of all network buffers would incur a prohibitive cost.

We thus conclude that enhancing the network stack itself is not sufficient for low-latency network transactions with NVMM, and that it is important that network stacks explicitly support NVMM *and* that applications exploit APIs that enable efficient durable data store.

5. DISCUSSION

5.1 Impact of NVMM Latency

Many different NVMM technologies are anticipated, with write and read latencies from tens to hundreds of nanoseconds [16, 23]. As explained in Section 3.3, idempotent requests are only DMA'ed to the CPU cache, thus the performance of idempotent requests is decoupled from the characteristics of the underlying storage. PASTE would only be exposed to the underlying media for the mutable transactions. This is unavoidable and inherent in storing data. We anticipate that PASTE would be suitable for many different NVMM types, but the performance of PASTE transactions would depend on the performance of underlying media.

5.2 Kernel-Bypass Networking

The architecture of PASTE relies on sharing network buffers between the network stack, the application and NVMM abstractions (e.g., files). Our implementation employs netmap, which executes in the kernel and thus allows PASTE to exploit a Linux file system with DAX [22] support. PASTE could also be implemented on systems such as Arrakis [27], IX [2] or Intel SPDK [15]. However, these systems have to implement NVMM abstractions by themselves.

Some user-space TCP/IP stacks, including Sandstorm [21] and UTCP [13], are easier to support PASTE, because their buffers can be managed by the netmap framework in the kernel. The only difference from our current design is that the TCP/IP protocol implementations reside in user space (i.e., in Figure 3, the "TCP/IP input and output" box moves to user-space).

5.3 Use Cases

We are currently examining the following systems to exploit PASTE. Networked applications that embrace a local database, such as SQLite and Berkeley DB, will be straightforward to port to PASTE, as they perform the same steps as described in Section 2 when durably storing data. The same holds true for stand-alone databases that employ write-ahead logging (journaling), such as MySQL, PostgreSQL and Apache Cassandra.

PASTE was designed to support systems with persistence guarantee semantics. Systems that accept transactions but do not guarantee their durability may not prove to be a great fit for PASTE. Systems such as HyperDex [9] perform asynchronous durable commits by acknowledging client requests prior to actually safely persisting them, sacrificing durability for speed and only guaranteeing consistency and atomicity. With NVMM and PASTE, such systems can provide durability without sacrificing performance, which may suggest a new replication strategy.

Recent research that focuses on write-ahead logging to NVMM, such as REWIND [4] and NVWAL [17] could be extended to use PASTE. Finally, distributed file systems that perform journaling, such as Ceph [31] could also benefit from PASTE.

6. RELATED WORK

Special-Purpose Network Stacks: Optimizing a network stack by leveraging application knowledge has been proposed several times [10, 21, 19]. Conversely, we design a network stack and its APIs so that applications can exploit a newly emerging hardware opportunity.

Exploiting RDMA is also popular for distributed computing [8] and storage [36] to avoid network stack overheads and use NVMM for efficient logging. Instead of bypassing it, our approach integrates a network stack with an NVMM stack, leveraging existing abstractions such as files and network protocols.

Enhanced Network Stacks: IX [2], mTCP [35], Fastsocket [20] and StackMap [34] are fast network stacks. Since they do not assume DMA on NVMM, they do not address the overheads of durably storing data, as described in Section 2.

General-Purpose Networking API: On the transmit path, the `sendfile()` system call enables applications to directly transmit data from in-kernel buffer caches identified by a file descriptor. However, doing something similar on the receive path (i.e., directly receiving data into the buffer cache) is not trivial, because applications also need to see the data to make processing decisions. SoftNIC [11] provides abstractions for multi-core processing, pipe-lining (chaining) and scheduling, but it limits its focus to NICs.

NVMM-Aware Persistent Data Store: There exists a large body of work on efficiently managing data in NVMM. Examples include a data center storage system [36], efficient in-memory, persistent data structures for file systems [6, 32, 29] and databases [4, 5], synchronization mechanisms for reducing memory traffic while preserving atomicity and durability [17], and consistency guarantees in the presence of write reordering caused by the CPU and memory controller [33].

However, none of this work integrates a network stack with NVMM. For example, NVWAL [17] employs byte-granularity differential logging to reduce the amount of data to log, resulting in a reduced number of memory copies and cache-line flushes. However, PASTE allows applications to log only a packet buffer index, offset and length (8 B in total) per packet, which is much smaller than the differential data set.

7. CONCLUSION

In current transactional network services the latency observed by clients is dominated by the persistence layer of the service. The current state-of-the-art for masking the latency of transactional services is write-ahead logging. Even with this technique, the inherent characteristics of the underlying block devices attached to the host with PCIe, or other means, presents an unsurmountable lower-bound on achievable latency.

In this paper we quantified this lower-bound and showed that it is still so high that the networking stack's performance has little bearing on the overall performance. We further showed that a host using NVMM for its write-ahead log would render the application sensitive to the performance of the network stack.

The imminent wide-scale adoption of NVMM suggests that the current bounds will soon be obsolete and makes it imperative that network stacks adapt to this impending world. Moreover, NVMM presents an opportunity to leverage the juxtaposition of the network stack, application logic and the persistence layer to provide excellent transactional latency. PASTE is working towards this goal by exploring new APIs and abstractions for the next generation of high-performance transactional systems.

Acknowledgments

We are thankful to Luigi Rizzo for his insightful comments, as well as anonymous HotNets reviewers for their thoughtful feedback. This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

8. REFERENCES

[1] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. *Proc. acm sigmod*. Melbourne, Victoria, Australia, 2015, pp. 707–722.

[2] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: a protected dataplane operating system for high throughput and low latency. *Proc. usenix osdi*. Broomfield, CO, 2014, pp. 49–65.

[3] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. *Proc. acm sosp*. Cascais, Portugal, 2011, pp. 143–157.

[4] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: recovery write-ahead system for in-memory non-volatile data-structures. *Proc. vldb endow.*:497–508, Jan. 2015.

[5] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *Proc. acm asplos*. Newport Beach, California, USA, 2011, pp. 105–118.

[6] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. *Proc. acm sosp*. Big Sky, Montana, USA, 2009, pp. 133–146.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *Proc. acm sosp*. Stevenson, Washington, USA, 2007, pp. 205–220.

[8] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: fast remote memory. *Proc. usenix nsdi*. Seattle, WA, Apr. 2014, pp. 401–414.

[9] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: a distributed, searchable key-value store. *Proc. acm sigcomm*. Helsinki, Finland, 2012, pp. 25–36.

[10] G. Ganger, D. Engler, M. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *Acm tocs*:49–83, 2002.

[11] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. Softnic: a software nic to augment hardware. Tech. rep. (UCB/EECS-2015-155). May 2015.

[12] Hewlett Packard Enterprise. Turbo-charge performance with HPE Persistent Memory. https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=4AA6-4771ENW&doctype=data%20sheet&doclang=EN_US. Mar. 2016.

[13] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling network protocol innovation with user-level stacks. *Acm sigcomm ccr*:52–58, Apr. 2014.

[14] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proc. vldb endow.*:389–400, Dec. 2014.

[15] Intel. Introduction to the Storage Performance Development Kit (SPDK). https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk. Sep. 2015.

[16] Jeff Chang. NVDIMM-N Cookbook: A Soup-to-Nuts Primer on Using NVDIMM-Ns to Improve Your Storage Performance. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/JeffChang-ArthurSainio_NVDIMM_Cookbook.pdf. 2015.

[17] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. Nvwal: exploiting nvram in write-ahead logging. *Proc. acm asplos*. Atlanta, Georgia, USA, 2016, pp. 385–398.

[18] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: a memory-efficient, high-performance key-value store. *Proc. acm sosp*. Cascais, Portugal, 2011, pp. 1–13.

[19] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: a holistic approach to fast in-memory key-value storage. *Proc. usenix nsdi*. Seattle, WA, Apr. 2014, pp. 429–444.

[20] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel tcp design and implementation for short-lived connections. *Proc. acm asplos*. Atlanta, Georgia, USA, 2016, pp. 339–352.

[21] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. *Proc. acm sigcomm*. Chicago, Illinois, USA, 2014, pp. 175–186.

[22] Matthew Wilcox. DAX: Page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/. 2014.

[23] Micron. Breakthrough Nonvolatile Memory Technology. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. F4: facebook's warm blob storage system. *Proc. usenix osdi*. Broomfield, CO, Oct. 2014, pp. 383–398.

[25] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield. Non-volatile storage. *Commun. acm*:56–63, Dec. 2015.

[26] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. *Proc. usenix osdi*. Hollywood, CA, 2012, pp. 1–15.

[27] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: the operating system is the control plane. *Proc. usenix osdi*. Broomfield, CO, Oct. 2014, pp. 1–16.

[28] L. Rizzo. Netmap: a novel framework for fast packet i/o. *Proc. usenix atc*. Boston, MA, Jun. 2012, pp. 101–112.

[29] D. Santry and K. Voruganti. Violet: a storage stack for iops/capacity bifurcated storage environments. *Proc. usenix atc*. Philadelphia, PA, Jun. 2014, pp. 13–24.

[30] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. *Proc. usenix fast*, 2011.

[31] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. *Proc. usenix osdi*. Seattle, Washington, 2006, pp. 307–320.

[32] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. *Proc. usenix fast*, 2016.

[33] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. *Proc. usenix fast*, 2015.

[34] K. Yasukata, M. Honda, D. Santry, and L. Eggert. Stackmap: low-latency networking with the os stack and dedicated nics. *Proc. usenix atc*. Denver, CO, Jun. 2016, pp. 43–56.

[35] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. *Proc. usenix atc*, 2004, pp. 99–112.

[36] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: a reliable and highly-available non-volatile memory system. *Proc. acm asplos*. Istanbul, Turkey, 2015, pp. 3–18.