PASTE: A Network Programming Interface for Non-Volatile Main Memory

Michio Honda[†], Giuseppe Lettieri[‡], Lars Eggert[⋆] and Douglas Santry[⋆]

NEC Laboratories Europe[†], Università di Pisa[‡], NetApp[⋆]

Abstract

Non-Volatile Main Memory (NVMM) devices have been integrated into general-purpose operating systems through familiar file-based interfaces, providing efficient byte-granularity access by bypassing page caches. To leverage the unique advantages of these high-performance media, the storage stack is migrating from the kernel into user-space. However, application performance remains fundamentally limited unless network stacks explicitly integrate these new storage media and follow the migration of storage stacks into user-space. Moreover, we argue that the storage and the network stacks must be considered together when being designed for NVMM. This requires a thoroughly new network stack design, including low-level buffer management and APIs.

We propose PASTE, a new network programming interface for NVMM. It supports familiar abstractions—including busy-polling, blocking, protection, and run-to-completion—with standard network protocols such as TCP and UDP. By operating directly on NVMM, it can be closely integrated with the persistence layer of applications. Once data is DMA'ed from a network interface card to host memory (NVMM), it never needs to be copied again—even for persistence. We demonstrate the general applicability of PASTE by implementing two popular persistent data structures: a write-ahead log and a B+ tree. We further apply PASTE to three applications: Redis, a popular persistent key-value store, pKVS, our HTTP-based key value store and the logging component of a software switch, demonstrating that PASTE not only accelerates networked storage but also enables conventional networking functions to support new features.

1   Introduction

Non-volatile main memory (NVMMs) [49] has the potential to change the way modern systems are designed and implemented[1]. 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.

Durable main memory will precipitate sweeping changes to how systems are designed end-to-end. The

---

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

entire processing cycle of an application will change. Storage and networking, in the form of user-level libraries, will become inextricably intertwined with application logic, instead of maintaining the clean separation offered by the kernel APIs today (e.g., POSIX).

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, an 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 a DMA, then the kernel 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. Using Write-ahead logs is much faster than updating a primary data structure, as it simply involves serially appending to a log of accepted transactions. The primary data structure is persisted periodically and corresponding 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.

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

With NVMM, a transaction could in principle become durable when the NIC DMAs 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. Further, because of low-latency random access on durably-stored data, NVMMs
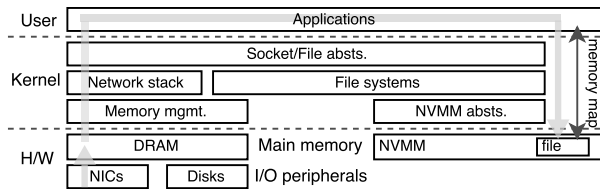
Figure 1: Today's OS organization: No integration between network stack and NVMM abstractions. A light gray arrow indicates path of data from NIC to NVMM.

could even obviate write-ahead logging depending on the primary database—creating a new opportunity of organizing primary data structures with packet buffers to which the NIC DMAs. However, since there is no integration between NVMM abstractions and the network stack, packet data and persistent data are treated separately. This leads to superfluous copying by applications (see Figure 1).

This paper proposes a fast new networking interface for persistent data on NVMMs, which we call PAcket STorE (PASTE). It allows applications to organize persistent data structures directly with network buffers, eliminating the superfluous data copies inherent in today's transactional systems. 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 *points* to arbitrary packet data. Network buffers are not recycled by the network stack until the owning application gives the stack permission. We implement PASTE as a Linux kernel module by extending the netmap [58] framework and exploiting the OS NVMM abstraction.

Our microbenchmarks that involve persistent data show PASTE outperforms a well-tuned Linux stack by up to 108 % in throughput and by up to 51 % in latency; It outperforms StackMap [68], the state-of-the art network stack by up to 43 % in throughput and 30 % in latency.

We apply PASTE to three applications: Redis, a popular persistent key-value store (up to 133 % improvement), pKVS, our custom key value store that runs over HTTP (up to 56 % improvement), and the logging component of software switch (up to 50 % improvement), in order to demonstrate that PASTE not only accelerates networked storage systems but also enables traditional networking functions to support new features.

The remainder of this paper is organized as follows: Section 2 describes background and analyzes the costs of durably storing data from an end-to-end perspective; Section 3 describes design and implementation of PASTE. Section 4 evaluates PASTE; Section 5 shows PASTE's use cases of key-value stores and software switch. Section 6 discusses PASTE's applicability and future work. Section 7 describes related work, and the paper concludes

with Section 8. Appendix A provides supplemental information and advanced experiment results.

2 Motivation

To motivate the proposed reorganization of the network stack, this section briefly reviews literature around persistent data. We then perform case studies to see what happens in reality.

2.1 Background

A transactional data transfer is an essential operation in many networked storage systems, such as blob stores [7, 48, 51], key-value stores [13, 37] and databases [1, 8, 26]. 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 (e.g., on the order of milliseconds). As discussed above, applications frequently use a write-ahead log to speed up transaction persistence instead of directly updating primary data structures, such as a B tree, which involves durably updating multiple blocks and hence many random seeks. The client-perceived transaction commit time is thus increased. 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 persistent NVMM directly (a black arrow in Figure 1). It should be emphasized that NVMM DIMMs are expected within months. While they will be more expensive than NAND Flash, they are expected to be cheaper than DRAM; DRAM is what they will be replacing so adoption is expected to be rapid and wide-spread. File systems can be put on top of NVMM much as they are today for RAM disks—except the contents will survive reboots and power failures. Applications can `mmap()` files into their address space, without a buffer cache interposed, and access their data directly with unprivileged CPU load and store instructions. System calls will be far too slow in comparison, so applications will just flush data from CPU caches into NVMM, typically using the `clflush` instruction. Thus,

Memory	Measurement	Time [µs]
—	Network only (H/W, stack, HTTP)	23.32
NVMM	Network + `memcpy()`	25.57
	Network + `memcpy()`/`clflush`	27.17
	Network + `read()`/`clflush`	27.41
SSD	Network + `memcpy()`/`msync()`	1320
	Network + `read()`/`msync()`	1300
	Network + `write()`/`fdatasync()`	1370
	Network + `write()`/`fsync()`	3490

Table 1: End-to-end transaction latency with various persistence methods: NVMM dramatically reduces end-to-end latency and data copy comes at a significant cost.
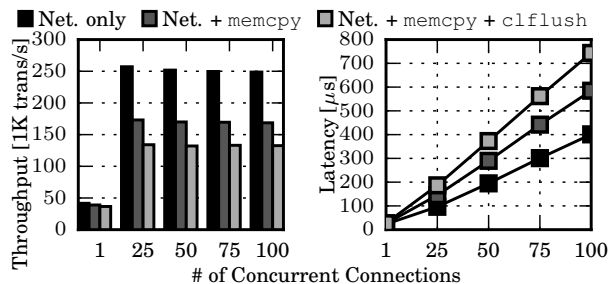

Figure 2: Throughput (left) and transaction latency (right) for concurrent requests and connections: Durably storing data still significantly reduces throughput and increases end-to-end transaction latency.

accessing storage in this new world will be two to three orders of magnitude faster than it is today.

2.2 End-to-End Transaction Latencies

To better understand the impact of logging on end-to-end latency, we wrote a simple HTTP server that implements three methods to durably log data. In all three cases, data arrives on a socket and is read by the server into a buffer. The methods of logging the data are:

(i) `write()` the buffer to a file, followed by either `fsync()` or `fdatasync()`.

(ii) `memcpy()` the buffer to a `mmap()`-ed file, followed by `msync()` for SSD or `clflush` for NVMM.

(iii) Pass the address of a `mmap()`-ed file to `read()` for use as the buffer, followed by `msync()` for SSD or `clflush` for NVMM.

The last method merges steps 5 and 6 of the general transactional data transfer (see Section 2.1), avoiding one of the two data copies that would occur otherwise. The data movement is depicted as a light gray arrow in Figure 1.

We examine two types of persistent media: a PCIe-attached SSD (Samsung 950 Pro, 256 GB) and an NVMM (HPE, 8 GB NVDIMM) attached to a DIMM slot. Both are formatted with the XFS file system that supports the Linux page-cache bypass mechanism, DAX [44] ("NVMM absts." in Figure 1). This NVMM has been available since early 2016, and costs approximately $900 for 8 GB [22].

On the client, we instrument `wrk`, a popular HTTP benchmark tool, to send 1412 B HTTP POSTs. The HTTP OK returned by the server is 127 B long. The server and client setup is described in Section 4.1.

Table 1 shows end-to-end transaction latencies that `wrk` reports. Storing data on NVMM is almost two orders of magnitude faster than on SSD. An interesting artifact is observed when comparing time to persist of the `read()`/`clflush` (4.09 µs) and `memcpy()`/`clflush` (3.85 µs) cases. Contrary to intuition, reading data directly into a `mmap()`'ed area is slower. This is because this case

is more likely to require full virtual to physical address translation, so it is slower than reading into a temporary buffer; the same temporary buffer is used every time so the CPU cache has the physical addresses already. Further, `memcpy()` moves data into the log using SSE instructions. In the later discussion, we focus on the variant that uses `memcpy()`, also because it is more realistic, e.g., it applies to user-level NVMM management systems [10, 67].

The majority of costs to durably store data stem from the data copy. We ran the same measurements without flushing data after copying it (the `Network + memcpy()` row in Table 1), which exhibits 1.6 µs lower latency. This is because the access latency with our NVDIMM is almost the same as for DRAM, which is on the order of tens of nanoseconds. Since we flush 1412 B—or 23 cache lines—in a write-through manner, we expect a latency around 1 to 2 µs, which matches out measurement result rather well.

2.3 Implications

We claim that these costs of durably storing data should be regarded as high because of two reasons. First, we expect that network stacks will become faster, as demonstrated by mTCP [30], IX [4] and StackMap [68], which could further emphasize the costs of durably storing data. Second, increased latencies—which we have already observed in a single request-response transaction—amplify in realistic scenarios, because server applications typically serve a large number of clients.

Figure 2 plots throughputs and transaction latencies over concurrent requests over parallel TCP connections. We confirm these reduced throughputs and increased transaction latencies as the number of concurrent requests increases. 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.

We think that these costs are unavoidable as long as we design storage and network stacks in isolation. For

example, Decibel [50] leverages DPDK for the network stack and SPDK for the storage stack, but it needs to move data between them, experiencing similar costs to those we identified above.

3 PASTE

In this section we describe PASTE, our integrated network and storage (in the form of NVMM) stack and API.

3.1 Design Principles

The persistence tier has been literally *secondary* storage, due to the costs of durably storing data on disks or SSDs, which we have quantified in the last section. NVMMs provide persistence primitives at the speed and with an interface comparable to main memory, and we envision ubiquitous deployment of them across many different applications. This includes not only storage systems, but also, for example, software switches and middleboxes for fault tolerance and fine-grained real-time monitoring, and different contexts, such as bare-metal servers in private data centres or virtual machines in the cloud [2]. For broadest deployability of PASTE, we do not rely on RDMA networking (we discuss it in Section 6.1).

PASTE is a new network programming interface for persistent data on NVMMs. There are a number of requirements for networking and persistent storage APIs, including blocking for efficiency and scalability, busy-polling for low latency and fault isolation between the stack and applications, which are benefits provided by the socket APIs today. In addition to these general requirements, PASTE achieves the following properties that concern applications:

Persisting data without a copy: This is essential, as in the previous subsection we identified that data copies to durably store data come at significant costs.

Crash recovery and consistency: Data must be randomly accessible from applications over reboots, otherwise persisting it is useless. Further, applications must be able to write and recover data *consistently*, so that they can reason about the validity of data and the metadata accompanying it (e.g., pointers and extent information) after system crash.

Avoiding unnecessary data persistence: Most network service also offer idempotent operations, some form of *read*. As seen above, persisting data is expensive so only mutable requests should be persisted.

Support for large data stores: Large capacity NVMMs are expected to store even a primary database [10, 40, 66] as opposed to fast, lower-capacity NVMMs that

---

²Virtualization and pass-through of NVMMs are active topics in both academia and industry [33, 63]
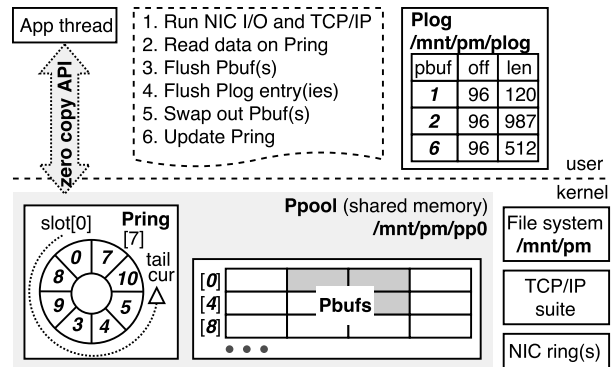

Figure 3: PASTE architecture: Packet buffers are named by a file backed by NVMM and pointed to by private application metadata.

are expected to store logs or journals [15, 36]. Therefore, we must design a networking framework that can manage large persistent data stores.

Support for network protocols: Applications must be able to use existing and new network protocols such as IP, UDP, TCP and NDP [20] for reliability, congestion control and/or compatibility with remote end systems.

Obviating serialization of application state: In general, applications have to maintain two forms of state: their in-memory state and their persistent state on disk/SSD. They are different, because DRAM is byte-addressable and exacts only minor performance penalties for random accesses whereas disks/SSDs offer awful random access performance. These conflicting characteristics lead to different interfaces that leads to different formats. The process of converting between the states is serialization [5, 6, 21, 59]. The DRAM image is the state that application actually wants, as that is the state that it actually computes with. Serialization leads to data corruptions bugs and performance problems. NVMM offers the opportunity to dispense with serialization where applications only need 1 state with NVMM, their in-memory state.

We describe details next, then show how useful these features are in building applications in the later section.

3.2 Architecture

PASTE is designed to execute NIC I/O, protocol stack processing and application logic synchronously in a batch, so called run-to-completion. This model is familiar and used by some recent systems such as Seastar [9], Sandstorm [42], IX [4] and StackMap [68], but PASTE extends it to accelerate persistence in cooperation with NVMM abstractions and provides suitable APIs for it. Figure 3 illustrates the PASTE architecture with its building blocks.

DMA into NVMM: 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. We must 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 version of data, then the first requirement is to ensure that such buffers can be found across system reboots and crashes.

Named packet buffers: 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 are 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. PASTE uses fixed-size packet buffers that are indexed by 32-bit integers, supporting several TB of data using 2 kB buffers. Hence, an application can represent data in its private data structures (`Plog` in Figure 3) by a simple tuple of a 32-bit buffer index, length and offset, which are 16-bit each in size (enough to accommodate an Ethernet jumbo frame).

PASTE initializes `Ppool` (see Figure 3) deterministically for a given memory region, which is "pinned" by a file. Hence, `Plog` and `Pbufs` are consistent over reboots.

Selective persistence: A quick digression into modern DMA is required to understand this issue. Modern NICs DMA packets into main memory *logically*, *physically* they are placed into the lowest-level CPU cache. 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. Since this operation is costly, we do not want to perform it for every packet. Instead, the application examines a packet first while the packet is still in the CPU cache (applications are oblivious to this as the CPU manages its cache transparently). Only if the application decides that the packet needs to be persisted is the packet moved to NVMM. On the current generation of Intel CPUs, this can be done with the `clflush(opt)` 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.

Lightweight ordered journaling: It is trivial to implement a *log* or *journal* [18] 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 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 log in their own NVMM-backed file (`/mnt/pm/plog` 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 they point to are in `/mnt/pm/pp0`.

Journaling with PASTE can be done as follows (see the pseudo-code in Figure 4, line 1–6): First, the application flushes buffer contents (lines 2–4), then durably writes a buffer extent that is a tuple of buffer index, offset and length (lines 5–6). The order ensures consistency against system failures. We analyse data integrity in detail in Appendix A.1.

Applications can perform each step over multiple buffers to journal long data. Since a tuple of buffer index, offset and length is 8 B in size, and Intel CPUs write an entire cache line of 64 B, it is possible to atomically commit up to eight entries. For longer data, the applications may put logs between additional "begin" and "end" entries like in conventional transaction logging.

Copy-on-write style free-space management: Committed buffers and logs comprise either write-ahead logs or primary data structures, such as a B+ tree. In either case, persisted buffers need to be moved out of the NIC ring (i.e., DMA target) so that buffers containing live data are not over-written. Since the `Pring` (in Figure 3) contains only *slots*—each of which includes a buffer index, length and offset (i.e., pointers to buffers)— this can be easily done by swapping buffers containing data with new empty ones outside the ring. The new empty buffers are thus attached to the slots of the ring and returned to the kernel to be eventually used as DMA targets.

The pseudo-code in lines 8–19 of Figure 4 shows the typical workflow. The application `poll()`s the receive ring for incoming requests (line 11); when it returns, it examines and generates a reply to each request (lines 12–19). Whenever it receives an update request (test at line 14) it also permanently stores the buffer containing data (lines 15–16) and then replaces it with a free one (lines 17–18) to preserve it. Buffers containing read requests are simply left in the receive ring to be reused. Responses are sent to the network in a batch at the next `poll()` (line 11).

Figure 3 illustrates an example. Initially, the `Pring` slots 0–7 pointed to buffers 0–7. Assume that the NIC has received packets on buffers 0–6 and the application has

```
1   flush_buf(buf, off, len, buf_idx, *log)
2     buf += off;
3     for (int i = 0; i < len; i += CACHE_LINE_SIZE)
4       clflush(buf + i);
5     *log = buf_idx << 32 | off << 16 | len;
6     clflush(log);
7
8   paste_eventloop(nmd, plog, plogsiz)
9     rx, tx = get_netmap_rings(nmd);
10    for (;;)
11      poll(/* on the rx ring */);
12      for each new slot s in rx
13        char *buf = get_netmap_buf(s, rx);
14        if (is_write_request(buf))
15          uint64_t *log = next_log(rx, plog, plogsiz);
16          flush_buf(buf, s->offset, s->len-s->offset,
                    s->buf_idx, log);
17          netmap_slot *extra = next_free_buf();
18          swap(s, extra); // swap buffer indices
19        write_response(tx);
20
21  main(pm_file, size, plog_file, plog_size, netmap_port)
22    fd = open(pm_file); // Ppool
23    p = mmap(fd, size);
24    netmap_pools_info *pi = p;
25    pi->memsize = size;
26    pi->buf_pool_objtotal = HOW_MANY_BUFFERS;
27    nmreq nmr = { .cmd = POOLS_CREATE, .extm = pi };
28    nm_desc *nmd = nm_open(netmap_port, &nmr);
29    plog_fd = open(plog_file);
30    plog_map = mmap(plog_fd, plog_size);
31    paste_event_loop(nmd, plog_map, plog_size);
```

Figure 4: Durably writing data and log in `flush_buf()`, event loop in `paste_eventloop()` and Ppool initialization in `main()`. Figure 12 in Appendix A.1 illustrates buffer state over time.

consumed them (indicated by advancing the "cur" ring pointer from slot 0 to 6) and persisted and logged buffers 1, 2 and 6 (the gray ones in the Ppool). The application thus has swapped the persisted buffers with free ones: in the example, these are the buffers 8, 9 and 10, respectively.

Although the Plog is depicted as an array of tuples for simplicity, it can be of an arbitrary form, such as a B+ tree accompanying more structured metadata (e.g., sorted by keys), as long as a single buffer extent can still be flushed atomically, thus ensuring consistency.

As it turns out, PASTE is suitable for copy-on-write operations, as opposed to in-place updates, because new data is always (DMA-)written to free space. Further, when data is stored in the primary database as with B+tree, since new data is written prior to logging, PASTE achieves *write-behind logging* [2].

If PASTE is used only for logging, primary data structures (not shown in Figure 3), 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 (one of the purposes 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. The corresponding buffers can now be returned to the free pool.

Network protocols: A protocol suite operates directly on Pbufs where the NIC or application reads or writes. On RX, the protocol suite sets only buffers whose data are *ready* (e.g., in-order TCP segments) to the application ring (Pring) with providing application data offset, so that the application can see useful data only. PASTE can hold non-ready data packets (e.g., out-of-order TCP segments) out of the NIC's DMA target, which are inserted to the Pring when the protocol suite indicates they are ready.

To exploit system call and I/O batching, a Pring multiplexes multiple streams (e.g., TCP connections); PASTE thus sets a file descriptor to each ring slot such that the application can distinguish them.

Protection: To be a generic programming interface, fault isolation is an essential property. Despite of the direct access to NVMM, PASTE only exposes data buffers to applications using the shared memory primitive in the kernel; it does not expose NIC registers or data structures managed by file systems or network protocols. When an application crashes, the rest of the system is unaffected.

3.3 API

In order to promote wide deployment, PASTE is designed to smoothly integrate with the netmap framework [58]; it runs in the kernel and mediates physical or virtual NIC ring(s) and applications via shared memory in which kernel- and user-owned regions are synchronized by `poll()` (blocking or non-blocking) or `ioctl()` (non-blocking) system calls. Therefore, PASTE inherits most parts of the netmap API.

Data semantics contained in ring slots depend on port types. When PASTE is used with the kernel TCP/IP implementation, each ring slot points to a buffer that contains an in-order TCP segment with offsets to payload data and a file descriptor as a ring may contain data from multiple TCP connections. On RX, buffers that belong to the same descriptor are grouped in the ring, so that the application needs to process each descriptor only once in an event loop. TX is opposite. The application puts data on Pbufs pointed by available slots (or sets existing Pbufs to the slots, avoiding data copy) with providing a file descriptor and headroom for protocol headers to each of them. PASTE relies on regular socket APIs (e.g., `socket()`, `bind()`, `listen()`, `accept()`) for control operations. When PASTE is used for a user-level TCP/IP implementation or a middlebox that perform raw packet I/O, a ring is just a replica of the physical or virtual NIC ring where packets are placed in arrival order.

To (re)initialize the Ppool (which also includes all packet buffers), an application first `open()`s and `mmap()`s a file backed by NVMM (lines 22–23 in Figure 4). If this is the first time the file is opened, the application initializes a header that describes how the memory region should be

organized (lines 24–26). In any case, it prepares a netmap request pointing at the memory region (line 27) and then opens the netmap port, binding it to the region (line 28). The kernel validates the user-space virtual addresses and obtains the corresponding kernel-space virtual address, then initializes the `Ppool` using them.

Recovery: PASTE deterministically initializes `Ppool` for the given NVMM region. Therefore, after reboot, the application can restore previous buffers by simply re-initializing `Ppool` with `nm_open()`, and reason about application-specific organization of these buffers using `Plog` which can be a write-ahead log or a primary data structure like B+ tree. In Figure 4, lines 29–30, the `Plog` is also allocated on the NVMM, in a separate file.

3.4 Implementation

We implemented PASTE by heavily extending netmap, inserting approximately 4K lines of code and removing approximately 0.4K, which also contains the kernel TCP/IP support and software switch extension which we explain in Section 5.2. PASTE is a loadable kernel module and it supports Linux kernel versions of 4.6–4.12 (the latest version at the time of this writing). No modification to the main Linux kernel is needed.

We rely on the Linux NVMM kernel subsystem that provides standard NVMM abstractions [62], such as pages, namespaces [28] and DAX [44], a file system interface to access a physical NVMM device without buffer caches. Thus, applications can create their packet buffers, journals, data structures on their favorite file systems, including ones whose file operations (e.g., directory scan) are optimized for NVMMs [66].

PASTE is open source and under active development. It is available at https://github.com/luigirizzo/netmap/tree/paste with all the PASTE applications we use for experiments. We also provide some implementation details in Appendix A.4.

4 Evaluation

We begin with microbenchmarking PASTE in comparison to state-of-the art systems. We evaluate PASTE with more realistic applications in Section 5.

4.1 Hardware and Software Setup

We use two machines connected back-to-back with two Intel X540-T2 10 Gbit/s NICs and a direct attached cable. The server machine that runs PASTE has two Intel Xeon E5-2640v4 processors clocked at 2.4 GHz. For NVMM, we use an HPE 8 GB NVDIMM and format it with XFS with DAX enabled (See Figure 1 for an architecture diagram.) The client machine has an Intel Xeon E5-2690v4

	64 B	256 B	768 B	1280 B	2560 B
Net. only	22.2 $\sigma = 1.4$	22.9 $\sigma = 1.1$	23.9 $\sigma = 1.2$	24.7 $\sigma = 1.2$	28.0 $\sigma = 1.2$
Linux	21.5 $\sigma = 3.5$	22.8 $\sigma = 5.7$	25.0 $\sigma = 8.9$	27.2 $\sigma = 11.0$	33.1 $\sigma = 14.1$
StackMap	22.7 $\sigma = 3.6$	23.9 $\sigma = 5.7$	26.2 $\sigma = 8.8$	28.4 $\sigma = 10.9$	31.6 $\sigma = 10.7$
PASTE	22.6 $\sigma = 1.9$	23.2 $\sigma = 1.9$	24.7 $\sigma = 2.1$	26.4 $\sigma = 1.8$	29.4 $\sigma = 2.1$

Table 2: **Mean roundtrip latencies in µs with standard deviations σ for WAL without concurrent requests.**

processor. Both the server and the client disable "turbo boost", hyper-threading and all the C-states. They both run Linux kernel 4.11 and compiled with `gcc` version 6.3. Unless otherwise stated, we use a single CPU core at the server and the `wrk` HTTP benchmark tool with fourteen CPU cores at the client to saturate the server. Unless otherwise stated, we use busy-wait and TCP on all the systems except for Section 5.2.

4.2 Methodology

We compare PASTE against a well-tuned Linux stack and StackMap, which is the state-of-the art network stack that achieves comparable performance to user-space network stacks while using the feature-rich kernel TCP/IP implementation [68]. We refer to a version of PASTE that uses a DRAM region organized by the regular netmap as StackMap, because it resembles the architecture while details differ (e.g., PASTE does not modify the kernel, scales better to multiple cores and offers simpler API). PASTE's improvements over this StackMap thus indicates effect of reduction of data copy to persist data.

In the end, comparing Linux, StackMap and PASTE, which all use the same TCP/IP implementation, precisely exposes the stack architecture differences without any performance difference that could arise from different TCP/IP protocol implementations, which is (only) a subset of the network stack. This is important, because TCP/IP implementations have largely different features and supported protocol extensions, and adopt different software architectures to implement them.

4.3 Microbenchmarks

4.3.1 Write-Ahead Log

Write-ahead logs (WALs) are the simplest data structure to persist data in practice. We arrange the NVMM to accommodate as many WAL entries and packet buffers, which amount to roughly 3.5 million entries and buffers. The client continually generates a fixed-size HTTP POST on each experiment.

Table 2 and Figure 5 shows end-to-end throughput and mean latency of Linux, StackMap and PASTE. To see how each method compares to a networking-only performance
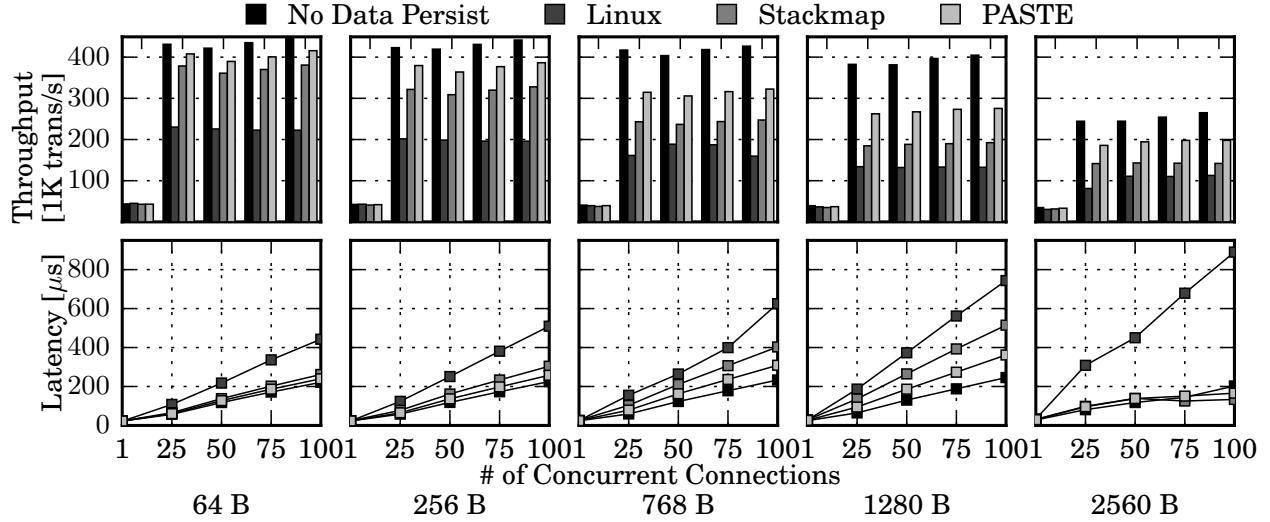
Figure 5: **Write throughput and mean latency with WAL**. Latency plots have standard deviations of at most 20 %, 43 %, 46 % and 9 % in No Data Persist, Linux, StackMap and PASTE, respectively.

baseline, we also plot PASTE without any persisting of data (it simply discards received messages and returns "HTTP OK" as if the transaction had been recorded).

For 64 to 1280 B message sizes, PASTE increases throughput by up to 108 % over Linux, and by up to 43 % over StackMap; it reduces latency by up to 51 % over Linux, and by up to 30 % over StackMap. In each method, throughput stays at almost flat on and after 25 parallel connections while latency keeps increasing. This is because the server (i.e. consumer) always has backlog requests to process. We observe improvements over StackMap by larger margins with increased message sizes. This is expected, because the cost of a data copy is small when messages are small. Latencies for the 2560 B case have different characteristics from the others, because each request now consists of two packets. In StackMap and PASTE, the lower latencies compared to that of smaller message cases is because queueing latency at the server becomes much lower due to decreased packet rates.

4.3.2 B+ Tree

Having identified that PASTE speeds up transactions to a write-ahead log, which is a temporary data structure, we now evaluate if PASTE can accelerate the case in which the data are directly stored in a primary database. We implement a B+ tree as a `Plog` on NVMM (Figure 3), a self-balanced, ordered tree which is widely used to organize primary data structures of file systems and databases. We instrument the B+ tree to store a `Plog` entry whose format is the same as in the WAL case (i.e., a tuple of buffer index, offset and length) as a value for a key. Recall from Section 3.2, the server flushes data for a network buffer prior to inserting the entry for this buffer to the B+tree.
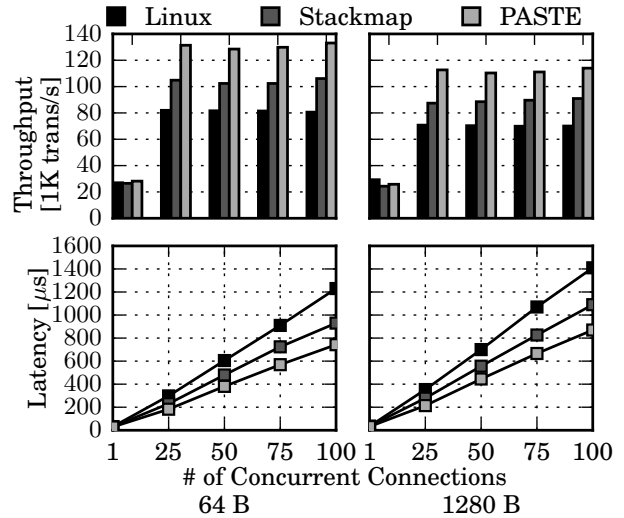

Figure 6: **Write throughput and mean latency with a B+ tree.** Latency plots have standard deviations of at most 39 %, 39 % and 7 % for in Linux, StackMap and PASTE respectively.

Figure 6 shows throughput and latency of Linux, StackMap and PASTE. The client continually transmits an "HTTP POST" message whose first 8 bytes indicate a "key" used by the B+ tree which contains 1 million random values. In the Linux and StackMap cases, the B+tree contains entire values copied from the network buffers. All the POST messages are served as insert or update. We test 64 B and 1280 B value cases.

While peak throughput is lower than in the WAL case because of tree traversal operations, we observe that PASTE improves throughput by up to 65 % over Linux
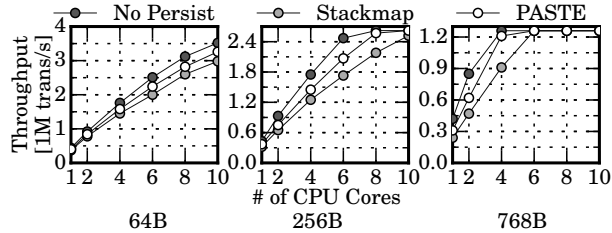
Figure 7: **WAL throughput over multiple CPU cores.**

and up to 28 % over StackMap, as well as reducing latency by up to 39 % over Linux and 23 % over StackMap.

For 64 B writes we see improvements by larger margins than WAL cases, because a B+tree is more memory intensive and the effect of reducing memory traffic is larger. We conclude that PASTE improves not only ephemeral persistent data structures, but also more complex primary data structures. In Section 5.1.1, we extend this PASTE B+ tree to a realistic key-value store that also serves read requests efficiently.

4.4 Multicore Scalability

Next, we evaluate PASTE's scalability to multiple CPU cores. We dedicate a single thread to each CPU core, and the NIC is configured to have one TX/RX pair of rings per core. Each thread independently processes a single pair of rings (`Prings` in Figure 3) with the `poll()` loop in Figure 4. It also persists data in its own `Plog` of WAL. The rings are mapped to the NIC rings, to which TCP connections are balanced by the NIC based on the connection hash value or the tuple of source-destination addresses and ports. All the rings share the same packet buffer pool or `Ppool` on NVMM. To saturate the server, we use an additional identical client machine. We use a ratio of 25 TCP connections to the number of cores.

PASTE reasonably increases throughout with additional cores. It reaches the 10 Gbit/s line rate at 8 and 6 cores with 256 and 768 B data, respectively.

5 Use Cases

In order to demonstrate how PASTE accelerates realistic applications and provides new opportunities, we have built three applications with it.

5.1 Key-Value Store

A popular use case is a key-value store (KVS) with durability support. The performance of a KVS is usually constrained by the network, because of lightweight `put/get` operations as opposed to relational databases, which require more computation to process client requests. While
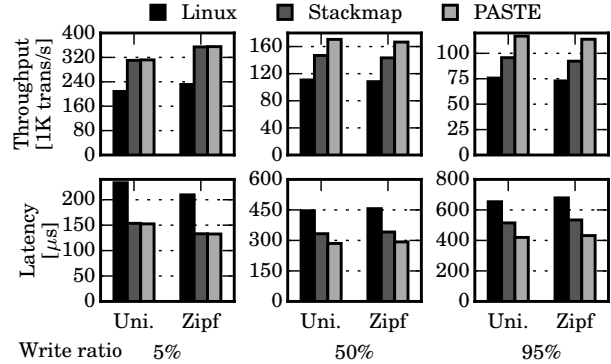

Figure 8: **Throughput and mean latency with pKVS.** Latency plots have standard deviations at most 30 % (Linux), 17 % (StackMap) and 11 % (PASTE).

joint-optimization of the network stack and volatile main-memory management has been explored (see Section 7), efficiently supporting data durability requires PASTE.

5.1.1 pKVS

pKVS is our custom KVS, which builds on top of PASTE and organizes data in a B+ tree. It uses HTTP as a communication protocol, mapping "set" and "get" commands into HTTP "POST" and "GET" methods, respectively. In addition to durable zero-copy writes, which we share in Section 4.3.2, pKVS also performs opportunistic zero-copy reads. On the "set" command, the server records a pointer to a buffer slot, and on the "get" command, the server first searches for the key in the B+ tree to obtain the buffer index and extent, then further obtains the slot which contains this buffer. The result is a complete form of the previous POST message in a packet buffer. The server thus simply places this buffer into a TX ring. In order to enable zero-copy, we tailor the length of the HTTP POST and OK to be identical.

Figure 8 shows throughput and average latency on different write ratio and key skewness. 50% and 5% of write ratios with Zipfian 0.99 distribution correspond to YCSB [12] workload A (Update heavy) and B (Read mostly), respectively. We use the default YCSB parameters for the key space (1K) and size (1KB). We use 50 concurrent TCP connections.

Because of large benefit of zero-copy durable write, PASTE improves throughput and latency as the write ratio increases. PASTE increases throughput by up to 56 %, and reduces latency by up to 36 % in comparison to Linux. PASTE increases throughput by up to 23 %, and reduces latency by up to 19 % in comparison to StackMap.

5.1.2 Redis

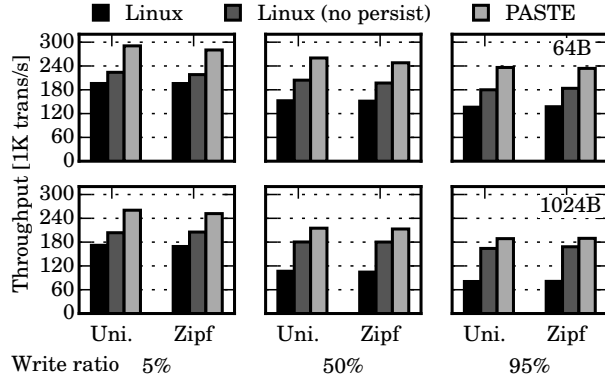Redis [57] is a popular named "data structure" on a network service. The server offers many services including

Figure 9: **Redis transaction rates.**

Figure 10: **mSwitch with packet logging support.**

counters, hyperlog estimators and a key/value store to name but a few. To demonstrate both the advantages of using PASTE, and the feasibility, we extended Redis 3.2.8.

Redis uses the socket API to communicate with clients over TCP and the POSIX file I/O interface. The source code was modified to receive events from PASTE instead of `read()` and data was persisted in PASTE buffers. Around 200 lines of source code were added to the 65K line base system. We discuss porting effort in Appendix A.4.

Figure 9 plots throughput of the regular and PASTE-enabled Redis with a single CPU core over different write ratios and key distribution patterns, including two default YCSB's workloads: read-mostly (5 % writes with key skewness of Zipfian 0.99 for 1KB data) and update-heavy (50 % writes with the same distribution and data size). For comparison, we test the regular Redis with and without persisting write operations (HSETs). To be fair, PASTE does not use busy-polling in this test.

Since the data structure is a relatively lightweight hash table, peak throughputs with PASTE are similar to the WAL case in Figure 5. PASTE outperforms Redis by 43 to 133 %. Even in comparison to Redis without persistence, PASTE outperforms it by 12 to 31 %, indicating PASTE offers persistence for more than free.

5.2 Software Switch

There is a growing interest by operators in the reliability of network middleboxes whose failure impacts on many end systems [35, 56, 61]. Network Function Virtualization (NFV) has led to deploying and consolidating middleboxes on commodity servers, enabling better resource utilization and fine-grained isolation [41, 60, 69]. Fault-tolerant middlebox (FTMB) [61] allows them to recover with states after crash. It relies on input packets stored in stable storage at the virtualization backend, which are replayed after the middlebox fails [61] since the last VM snapshot.

Programmable traffic monitoring systems [14, 32, 47] could also benefit from real-time packet logging, which is now often performed by dynamically activating a mir-
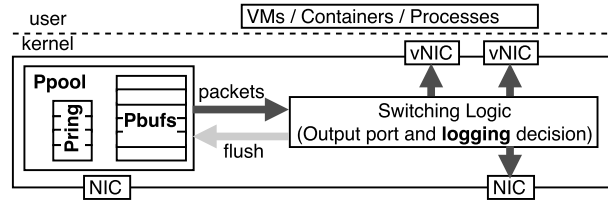
roring port on a hardware or software switch [53, 64] to capture traffic. PASTE already maintains a file with captured packets that is a snapshot of the most recent packets decided to be recorded, which can be directly leveraged for this use-case.

In order to support FTMB and other applications that benefit from packet logging, we implement a logging feature in mSwitch, a fast, modular software switch that supports a large number of virtual and physical ports to serve an NFV backend [24]. Its switching logic is modular and can implement arbitrary packet processing, such as a learning bridge, L3 forwarding, the Open vSwitch datapath and a subset of P4 [54]. A module takes packets as input from the switching fabric, and returns packet action values indicating destination switch port, drop or broadcast.

mSwitch acts as an application of PASTE despite that it runs in the kernel (Figure 10). We implement a new packet action of "logging" which is used in conjunction with the existing actions by switching logic modules. When the module indicates a packet to be logged, mSwitch swaps out the buffer from the receive ring slots.

Figure 11 shows throughput with PASTE in comparison to a variant of mSwitch that implements packet logging without PASTE, by copying and flushing packets from the DRAM to the NVMM. We use the default learning bridge module that has moderate overhead consisting of two hash calculations for source and destination MAC addresses. Packets are forwarded between the two 10 Gbit/s NIC ports. For the latency measurement, we increase *burst* sizes, which indicate the number of packets arriving at the input NIC of the mSwitch at a line rate. This models a very common situation, for example, a TCP sender is allowed to send up to ten packets at once (i.e., at line rate) even at the beginning of a connection.

We confirm that PASTE improves throughput and latency by up to 50 % and 15 %, respectively. A higher latency with increasing burst sizes is due to batching in order to amortize device I/O cost (on the order of hundreds of ns) and improve packet processing locality [24]. Thus, reduction of per-packet logging costs with PASTE reduces latency by larger margins in the presence of larger numbers of packets processed within the same batch.
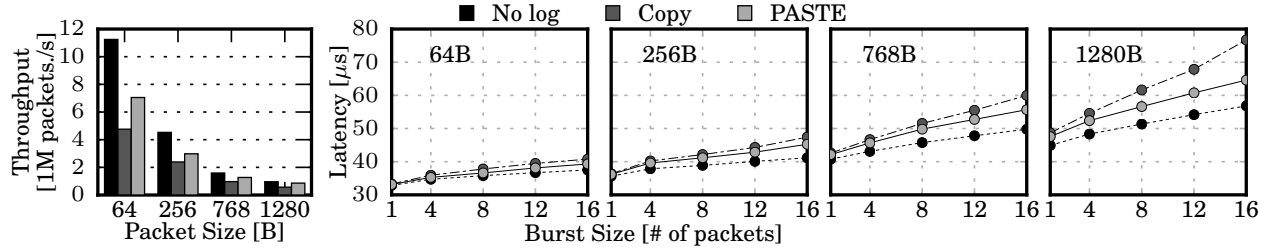
Figure 11: **mSwitch packet forwarding throughput and latency with packet logging.**

6 Discussion

In this section we discuss applicability of PASTE to various systems, and describe future work.

6.1 Kernel-Bypass Networking

PASTE relies on sharing network buffers between the network stack, the application and NVMM abstractions (e.g., files). Our implementation employs netmap [58], which executes in the kernel and thus allows PASTE to exploit a Linux file system with DAX [44] support. PASTE could also be implemented on systems such as Arrakis [55], IX [4] or Intel SPDK [27]. However, these systems have to implement NVMM abstractions by themselves.

Some user-space TCP/IP stacks, including Sandstorm [42], UTCP [25] and Warpcore [16], are easier to support with PASTE, because their buffers can be managed by netmap in the kernel. The only difference from our current design is that the TCP/IP implementation ("TCP/IP suite" in Figure 3) resides in user space.

As it turns out, our mSwitch extension in the last section follows this line and demonstrates flexibility of PASTE architecture, because it bypasses the vast majority of the kernel network stack to utilize a fast packet I/O framework.

We are starting to see RDMA deployments [19], but it requires loss-less network fabric and individual systems closely tied with low-level hardware details [15, 31, 40]. LITE [65] remedies the latter problem by kernel-level abstraction. This could help PASTE support RDMA, providing a higher-level interface to integrate with NVMMs and to be transparent to TCP/IP networking.

Our latency target range (e.g., 22.6–29.4 µs without parallel requests, see Section 4.3.1) is close to that of RDMA. [19] reports RTTs of several tens of µs over RDMA network fabric likely in the absence of queueing between network and application formed by parallel requests. In addition, recent work also reports comparable latency is achievable over lossy Ethernet fabric without RDMA [20].

6.2 NVMM Access Latency

Many different NVMM technologies are anticipated, with I/O latencies from tens to thousands of nanoseconds [29,

45]. As explained in Section 3.2, 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 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 any underlying media.

The latest generation of Intel CPUs have a faster cacheline flush instruction (`clflushopt`), which also works in a write-back fashion. Therefore, we will be able to *overlap* NVMM access latencies with subsequent processing; and this can be done across multiple requests processed in the same batch, i.e., in the same `poll()` loop (see Figure 4). We can guarantee that all the flushes are done at the time of triggering transmission (i.e., `poll()` using `mfence` instructions. In Appendix A.2 we describe details, and quantify effects with some experiments.

6.3 Generality

PASTE works as a fast, scalable network stack in the absence of NVMM, because it still exploits run-to-completion, system call and I/O batching, and zero copy between the NIC and application. The netmap API that PASTE is based on has been widely used in packet I/O applications and has proven its flexibility and ease of use. Further, PASTE can be used without modifying the kernel, and offers protection provided by the socket API. Therefore, we believe PASTE is a suitable basis which achieves high performance in general and makes applications ready to efficiently support NVMMs.

6.4 Limitations and Future Work

Space utilization: PASTE relies on fixed-size packet buffers for indexing. For better space utilization, we would combine copies for small data depending on workloads.

Multiple applications: Since the application needs to have direct access to NIC's DMA target, isolating multiple applications requires partitioning it. This could be done using Flow Director on multiple NIC queues and Smart NICs (based on more flexible policy).

NVMM wear: DMA-writes would increase wear on NVMMs, while it could be mitigated by DDIO. We leave analysing this effect in future work.

7 Related Work

Previous work discussed PASTE's concept and strategy, and made minimalistic implementation and experiments using an emulated NVMM device [23]; This paper completes our design and implementation, as well as extensive evaluation and case study of applying to applications.

Special-Purpose Network Stacks: Specializing a network stack by leveraging application knowledge has been proposed several times [17, 38, 42, 43]. PASTE takes a different approach with a network stack that is general enough to support different classes of applications.

Enhanced Network Stacks: IX [4], mTCP [30], Fastsocket [39] and StackMap [68] 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, and shown how PASTE improves these approaches in Section 4. We have discussed RDMA approaches in Section 6.1.

General-Purpose Networking API: On the transmit path, the `sendfile()` system call enables applications to directly transmit data from in-kernel buffer caches or NVMMs. However, doing the opposite (i.e., directly receiving data into the buffer cache or NVMM) is not trivial, because applications need to examine the data to make processing decisions. PASTE enables this by the persistent, named packet buffers and their abstraction.

New NIC Interfaces: FlexNIC [34] provides rich abstractions of NIC features, such as scheduling, offloading and classification. These works are complementary to PASTE. For example, they could support isolating multiple applications on the same NIC.

NVMM-Aware Persistent Data Store: There exists a large body of work on efficiently managing data in NVMM. They tend to examine the problem from the perspective of the storage system in isolation. There is little consideration of data arrival from a network or the requirements of application logic. The POSIX API is often their starting point. They can be generally classed into block-oriented storage systems, such as file systems and virtual disks [3, 26, 36, 46], or byte-oriented file systems [11, 66, 67], that is the latter's metadata is byte-oriented, but they still export a POSIX interface. Some NVMM programming systems [10, 59] are designed from the application's perspective. `malloc()` manages NVM and transactions on nodes in linked lists or binary trees are supported instead of file blocks. This approach fits the PASTE approach, the native representation of data is a first class citizen, not serialized data.

None offer a coherent and integrated life-cycle for work arriving from a network that needs to be persisted. For example, NVWAL [36] 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. There is no support efficiently storing data in its final resting place. 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.

DRAM-based Data Store: There is a large body of work which co-design in-memory data store with network stacks. For example, MICA [38] is an extremely fast, scalable key-value store that bypasses the most of the network stack and relies on UDP to tightly map key-value data structures and packets. RAMCloud [52] is a distributed key/value store that avoids the penalty of persisting to media by replicating to multiple physical machines. PASTE could help such systems to support persistence, because it creates and names packet buffers on NVMM and allows applications to organize them with zero-copy, protocol-independent networking API. On NVMM, applications can use the same cache invalidation mechanisms with this class of work.

8 Conclusion

NVMM is not just a faster, more exotic, storage medium. It is a fundamental change in the memory hierarchy. Its introduction and adoption will change the way we design and evaluate systems. The artificial sequestering of networking stacks, storage stacks and application logic will be infeasible with such hardware. The Network File System (NFS) was feasible because the network was not the bottleneck, the bottleneck was the disk. Commodity NVMM is pushing the stack out of the kernel and into user-land. Network stacks are following. As the application, network/storage stacks will be operating in the same address space they need to be co-designed for true efficiency.

In this paper we have quantified the cost of a network service offering reliable storage services under a variety of scenarios. We have shown that by tightly integrating the network stack, application logic and the storage stack large performance improvements can be realized. PASTE is a system that safely permits applications to be built, and back-ported, to gain these performance improvements. It does this while retaining the isolation, protection and software maintenance advantages of modern monolithic kernel stacks. We verified our system by implementing and evaluating PASTE then writing and back porting real applications to use it. We then showed PASTE-based applications' performance to be superior to the state of the art.

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/PODS*. 2015.

[2] J. Arulraj, M. Perron, and A. Pavlo. "Write-behind Logging". *Proc. VLDB Endow.* Nov. 2016.

[3] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. "Operating System Implications of Fast, Cheap, Non-Volatile Memory". *Proc. ACM HotOS*. 2011.

[4] 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*. 2014.

[5] A. Birrell, M. Jones, and E. Wobber. "A Simple and Efficient Implementation of a Small Database". *Proc. ACM SOSP*. 1987.

[6] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. "Orleans: cloud computing for everyone". *Proc. ACM SoCC*. ACM. 2011.

[7] B. Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". *Proc. ACM SOSP*. 2011.

[8] A. Chatzistergiou, M. Cintra, and S. D. Viglas. "REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures". *Proc. VLDB Endow.* Jan. 2015.

[9] Cloudius Systems. *Seastar*. http://www.seastar-project.org/.

[10] 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*. 2011.

[11] 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*. 2009.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB". *Proc. ACM SoCC*. ACM. 2010.

[13] 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*. 2007.

[14] S. Donovan and N. Feamster. "Intentional Network Monitoring: Finding the Needle Without Capturing the Haystack". *Proc. ACM HotNets*. 2014.

[15] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. "FaRM: Fast Remote Memory". *Proc. USENIX NSDI*. 2014.

[16] L. Eggert. *warpcore*. Jan. 2017.

[17] G. Ganger, D. Engler, M. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. "Fast and flexible application-level networking on exokernel systems". *ACM ToCS*, 2002.

[18] G. R. Ganger and Y. N. Patt. "Metadata Update Performance in File Systems". *Proc. USENIX OSDI*. 1994.

[19] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. "RDMA over Commodity Ethernet at Scale". *Proc. ACM SIGCOMM*. 2016.

[20] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". *Proc. ACM SIGCOMM*. 2017.

[21] M. P. Herlihy and B. Liskov. "A Value Transmission Method for Abstract Data Types". *ACM Trans. Program. Lang. Syst.* Oct. 1982.

[22] 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.

[23] M. Honda, L. Eggert, and D. Santry. "Paste: Network stacks must integrate with nvmm abstractions". *Proc. ACM HotNets*. ACM. 2016.

[24] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. "mSwitch: A Highly-scalable, Modular Software Switch". *Proc. ACM SOSR*. 2015.

[25] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. "Rekindling Network Protocol Innovation with User-level Stacks". *ACM SIGCOMM CCR*, Apr. 2014.

[26] J. Huang, K. Schwan, and M. K. Qureshi. "NVRAM-aware Logging in Transaction Systems". *Proc. VLDB Endow.* Dec. 2014.

[27] 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.

[28] Intel Corporation. *NVDIMM Namespace Specification*. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

[29] 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. Sep. 2015.

[30] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems". *Proc. USENIX NSDI*. 2014.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. "Design Guidelines for High Performance RDMA Systems". *Proc. USENIX ATC*. 2016.

[32] A. Kangarlou, S. Shete, and J. D. Strunk. "Chronicle: Capture and Analysis of NFS Workloads at Line Rate". *Proc. USENIX FAST*. 2015.

[33] S. Kannan, A. Gavrilovska, and K. Schwan. "pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage". *Proc. ACM EuroSys*. 2016.

[34] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. "High Performance Packet Processing with FlexNIC". *Proc. ACM ASPLOS*. 2016.

[35] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. "Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr." *Proc. USENIX NSDI*. 2016.

[36] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. "NVWAL: Exploiting NVRAM in Write-Ahead Logging". *Proc. ACM ASPLOS*. 2016.

[37] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. "SILT: A Memory-efficient, High-performance Key-value Store". *Proc. ACM SOSP*. 2011.

[38] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". *Proc. USENIX NSDI*. 2014.

[39] 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*. 2016.

[40] Y. Lu, J. Shu, Y. Chen, and T. Li. "Octopus: an RDMA-enabled Distributed Persistent Memory File System". *Proc. USENIX ATC*. 2017.

[41] V. Maffione, L. Rizzo, and G. Lettieri. "Flexible virtual machine networking using netmap passthrough". *Proc. IEEE LANMAN*. IEEE. 2016.

[42] I. Marinos, R. N. Watson, and M. Handley. "Network Stack Specialization for Performance". *Proc. ACM SIGCOMM*. 2014.

[43] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart. "Disk|Crypt|Net: Rethinking the Stack for High-performance Video Streaming". *Proc. ACM SIGCOMM*. 2017.

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

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

[46] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. "Operating System Support for NVM+DRAM Hybrid Main Memory". *Proc. ACM HotOS*. 2009.

[47] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. "Trumpet: Timely and Precise Triggers in Data Centers". *Proc. ACM SIGCOMM*. 2016.

[48] 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*. 2014.

[49] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield. "Non-volatile Storage". *Commun. ACM*, Dec. 2015.

[50] M. Nanavati, J. Wires, and A. Warfield. "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage". *Proc. USENIX NSDI*. 2017.

[51] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. "Flat Datacenter Storage". *Proc. USENIX OSDI*. 2012.

[52] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. "Fast Crash Recovery in RAMCloud". *Proc. ACM SOSP*. 2011.

[53] Open vSwitch. *Basic Configuration*. http://docs.openvswitch.org/en/latest/faq/configuration/.

[54] P4 Consortium. *P4 Language Specification*. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[55] 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*. 2014.

[56] R. Potharaju and N. Jain. "Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters". *Proc. ACM IMC*. 2013.

[57] Redis. *Official Redis Website*. https://redis.io/.

[58] L. Rizzo. "netmap: A Novel Framework for Fast Packet I/O". *Proc. USENIX ATC*. 2012.

[59] D. Santry and K. Voruganti. "Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments". *Proc. USENIX ATC*. 2014.

[60] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. "Design and Implementation of a Consolidated Middlebox Architecture". *Proc. USENIX NSDI*. 2012.

[61] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. "Rollback-Recovery for Middleboxes". *Proc. ACM SIGCOMM*. 2015.

[62] SNIA Technical Position. *NVM Programming Model Version 1.2*. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf. 2017.

[63] Stefan Hajnoczi. *Using NVDIMM under KVM*. https://vmsplice.net/~stefan/stefanha-fosdem-2017.pdf.

[64] O. Tilmans, T. Bühler, S. Vissicchio, and L. Vanbever. "Mille-Feuille: Putting ISP Traffic Under the Scalpel". *Proc. ACM HotNets*. 2016.

[65] S.-Y. Tsai and Y. Zhang. "LITE Kernel RDMA Support for Datacenter Applications". *Proc. ACM SOSP*. 2017.

[66] J. Xu and S. Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories". *Proc. USENIX FAST*. 2016.

[67] 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.

[68] K. Yasukata, M. Honda, D. Santry, and L. Eggert. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". *Proc. USENIX ATC*. 2016.

[69] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood. "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization". *Proc. ACM CoNEXT*. 2016.

A Appendix

A.1 Consistency Analysis

Figure 12 illustrates data states over a single network event loop cycle. If the system crashes before metadata (`Plog` entries) are flushed, extant packet buffers are simply overwritten by the next packets following reboot. If it crashes after the metadata has been written but before the corresponding buffers are swapped out of `Pring`, the application must do so right after re-initializing `Ppool`, before starting network I/O. Note that the application should not have updated the ring's pointer (`cur` in Figure 3) before swapping out the buffers.

The application can identify the buffers to be swapped out by reading its `Plog`. There is no atomicity semantics on buffer swapping, so the application should read `Plog` and ensure that the necessary buffers are in the intended place in either on or outside the `Pring`. The application may also leverage the ring pointer to identify buffers that have been swapped out, because the ring pointer can be updated atomically.

If the system crashes after buffers have been swapped out, buffers are consistent. However, some data sent after that, such as response messages might have been lost before being dispatched to the network. It is the responsibility of the application-level protocol to address or tolerate duplicate responses.

A.2 Overlap Flushes for an Event Loop

In Section 6.2, we have introduced a technique that overlaps flushes and other processing in a network event loop that processes multiple requests, by leveraging `clflushopt` and `mfence`. We set out to test this method using a server equipped with an Intel Xeon Silver 4110 CPU clocked at 2.1 Ghz that supports this instruction. Unfortunately, since this machine does not support our NVMM device, we emulate NVMM using a reserved region of DRAM as prior work does [26, 40].

Figure 13a shows WAL throughput and mean latency. The overlap improves throughput by up to 47 % and latency by up to 32 % in StackMap that copies data. It improves throughput by up to 72 % and latency by up to 42 % in PASTE. PASTE with the overlap improves throughput by up to 54 % and latency by up to 35 % in comparison to StackMap.

Figure 13b shows the B+tree case. The overlap improves throughput by up to 93 % in StackMap, and up to 133 % in PASTE; PASTE with the overlap improves throughput by up to 59 % in comparison to StackMap.

We observe higher throughputs in comparison to equivalent results in Section 4, although the CPU clock is lower in this server and the real NVMM used in the other server achieves the same speed with DRAM "in theory". This is perhaps because of higher memory clock frequency of this server (2600 Mhz, as opposed to 2133 Mhz in that section), and the newer CPU generation.

A.3 Effect of High NVMM Access Latency

Using the aforementioned overlap technique, we examine the effect of NVMMs with higher access latency. Since `clflushopt`s are asynchronous, we expect that higher NVMM access latency delays `mfence` to return. We thus insert artificial `sleep()` before `mfence`, and measure impact on overall throughput.

Figure 14 plots results, and they match our expectation. Emulated latency decreases throughput by larger margin as the number of parallel connections or requests decreases, because the NVMM access latency is amortised over the number of requests processed in the same network event loop.

In summary, also including the previous subsection, we conclude that the overlap technique significantly improves performance, and could mask high NVMM access latency. However, there is also a caveat. This method could increase the complexity of consistency guarantees. It certainly avoids compromising data after acknowledging to the client. However, when the system crashes before doing so, the system does not have any guarantee of the correctness of receiving data to be written. We can mitigate this risk by either flushing metadata, or designing the application-level protocol to tolerate duplicate writes where the server thinks the data is written but the client does not so thus precipitating resends of the previous write. We leave the analysis of these approaches for future work.

A.4 Implementation Note

In the OS kernel, network protocols are usually implemented using OS-specific packet representation structures (`sk_buff` in Linux, `mbuf` in *BSD). They typically contain metadata and one or more *pointers* to buffers that contain actual packet data, allowing them to point `Pbuf`s. Once an RX buffer is passed to the TCP/IP implementation (`netif_receive_skb()` in Linux, `ifp->if_input()` in FreeBSD), in order to identify whether it is ready to be set to a `Pring` (e.g., in-order TCP segment), we exploit a callback that is invoked on data enqueued to a socket buffer (`sk_data_ready()` in Linux and `sb_upcall()` in FreeBSD). The socket structure also has interfaces for kernel subsystems (e.g., iSCSI) similar to user-space socket APIs. But in the kernel they also provide zero-copy APIs (`kernel_sendpage()` in Linux and `sosend()` in FreeBSD), which allow PASTE to pass data that reside in `Pbuf`s to the TCP/IP implementation on the TX path.

Further, the OS kernels provide an interface (`get_user_pages()` in Linux and `vm_map_*()` family
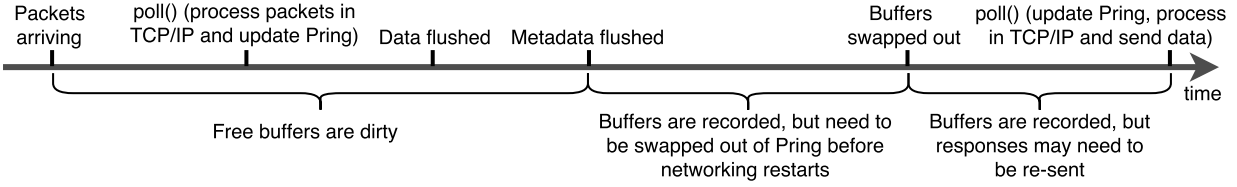
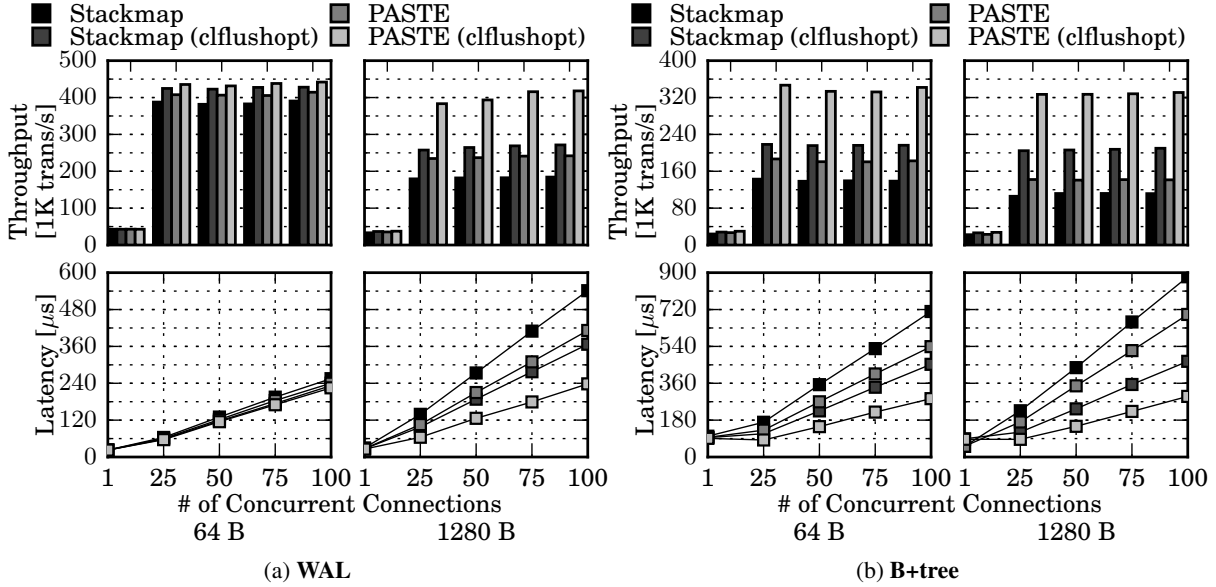Figure 12: **Buffer state over a networking event cycle.**

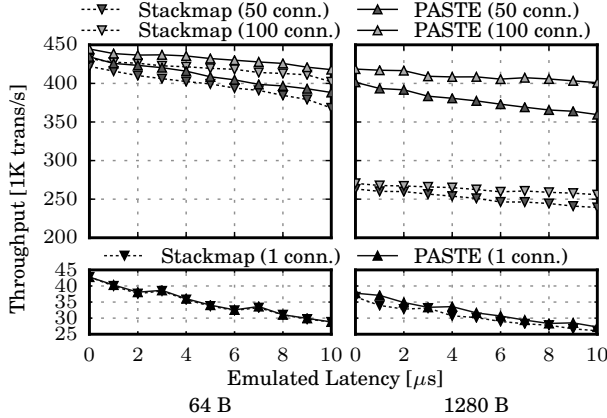Figure 13: **Throughput and mean latency with `clflushopt`.**

Figure 14: **WAL throughput over emulated NVMM access latency.**

in FreeBSD) to obtain kernel-space virtual addresses from the user-space ones `mmap()`ed to the file (e.g., `Ppool`). Therefore, PASTE can be implemented without modifying the OS kernel, using its *good* parts, such as protection mechanisms inherited from the netmap framework,

NVMM abstractions, file systems and extensive network protocol implementations.

FreeBSD support is our ongoing effort. It appears possible once the basic NVMM programming model [62] is supported, because the netmap framework is already there.

The porting effort of existing applications to use PASTE is medium, according to our experience with Redis (where the majority of the effort was to understand how Redis works, a burden that the maintainers would not have to bear). We have a library implemented as a header file to initialize and run an event loop in Figure 4. This library implements two callbacks to be registered by an application: one invoked at `accept()` and the other invoked on every RX packet buffer when traversing the ring (line 12 in the figure). In addition to rearranging Redis to use these features, we extended a function that parses and identifies a write request to flush and swap out the buffer, using the same procedure with `flush_buf()` in the figure.

In order to ease porting existing applications and writing new ones, we plan to extend `libuv`, a popular event-driven networking library, to support PASTE.