Challenges and Solutions for Fast Remote Persistent Memory Access

Anuj Kalia*
Microsoft Research
anuj.kalia@microsoft.com

David Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Michael Kaminsky
BrdgAI, Carnegie Mellon University
kaminsky@cs.cmu.edu

ABSTRACT

Non-volatile main memory DIMMs (NVMMs), such as Intel's Optane DC Persistent Memory modules, provide data durability with orders of magnitude higher performance than prior durable technologies. This paper explores the unique challenges that arise when building high-performance networked systems for NVMM. Compared to DRAM, we find that NVMMs have distinctive fundamental properties that pose unique challenges for networked access to NVMM, both from the NIC and the CPU. We show that much of the challenges in efficient access to remote NVMM arises from the fact that CPU caches are not optimized for NVMM. To address these challenges, we propose a menu of solutions for current hardware and evaluate their benefits.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Networks → Network protocols**.

KEYWORDS

Persistent memory, Remote Procedure Calls, Remote Direct Memory Access, Intel I/O Acceleration Technology

ACM Reference Format:
Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and Solutions for Fast Remote Persistent Memory Access. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421294

*Work done as a student at CMU

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.
SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421294

1 INTRODUCTION

The arrival of NVMM DIMMs with sub-microsecond latency requires rethinking the design of high-performance networked systems for modern datacenters. NVMM breaks the storage latency barrier that has long limited the performance of such systems. With NVMMs such as Intel's Optane DC Persistent Memory Modules (referred to as Optane DIMMs in this work), making data durable now requires less time (∼100 ns) than a datacenter network round trip (≈2 µs), reversing a historical trend.[1] In the past, persistent media had high latency (e.g., ≈10 µs for the fastest SSDs), and systems designed to operate at near-network latencies avoided syncing data to stable storage on the critical path of requests, often sacrificing consistency guarantees. Several networked systems used in industry now support NVMM, including key-value stores [33, 41], databases [8, 38], and object stores [3].

Researchers have redesigned distributed systems in anticipation of NVMMs. (We use the term NVMM to refer to DIMMs with persistent media, excluding DRAM-based approaches to provide non-volatile memory, such as such as battery-backed DRAM.) These systems include distributed transaction processing systems and key-value stores [19, 27, 35, 50, 54], network stacks [23], distributed file systems [9, 51], disaggregated persistent memory [46], etc. These projects use emulated NVMM (e.g., using battery-backed DRAM) to guide their design, with the (sometimes implicit) expectation that the observations will carry over to real NVMM when it becomes available.

In this paper, we investigate network-level challenges that arise when building high-performance distributed systems for NVMM, with Optane memory as a representative technology. Similar to recent research by Yang et al. [53] on single-machine systems, we find that NVMMs have distinctive (compared to prior DRAM-based emulation setups) yet fundamental properties that affect performance. In addition

[1] Although datacenter network bandwidth is increasing rapidly, network latency has stagnated at around 1–2 µs per intra-rack round trip.

to NVMM's well-understood distinctions, such as higher performance for sequential accesses than random accesses, and lower performance for writes than reads, we discover new distinctions that arise specifically in the networked context. These include the interplay between PCIe or DMA accesses and the NVMM's internal block size, and performance regressions that occur in particular workloads such as networked counters and timestamps.

What makes fast remote access to NVMM challenging? We find that a recurring cause is that CPU caches are not optimized for NVMM. We identify sub-optimal interactions between caches and NVMM that future hardware architects may target, and we design methods for fast remote NVMM access that work well on current hardware. Our methods address both methods of remote NVMM access: one-sided RDMA access that bypasses the remote CPU, as well as RPC-based access in which the remote CPU handles all NVMM access. In addition, we make the following contributions:

(1) We present, to our knowledge, the first detailed empirical evaluation of networking approaches (one-sided RDMA and RPCs) to access remote Optane DIMMs. We show that for small persistent writes to remote NVMM, RPCs have comparable latency as one-sided RDMA.
(2) We show that, counter-intuitively, disabling the Data Direct I/O optimization, with which NICs inject data into the CPU's L3 cache instead of DRAM/NVMM, *improves* bandwidth of bulk RDMA writes by avoiding random accesses to NVMM that DDIO generates.
(3) We show how RPC-based approaches can use the CPU's DMA engines to achieve 2.3x higher bulk write bandwidth.
(4) We present as case studies two distributed systems that demonstrate the effectiveness of our techniques. We build a state machine replication (SMR) system whose 99th percentile latency for three-way replication is 10.2 µs, which is only 12% higher than a non-persistent DRAM-based version. We build a networked log server whose append rate improves by up to 90% with our new optimizations.

A note on the paper's organization: After covering background and our experimental setup, we dive into challenges and solutions for low-latency (Section 4) and high-bandwidth (Section 5) access to remote NVM. Section 6 presents our persistent log case study. We cover our state machine replication system in the context of low-latency remote NVMM access in Section 4.

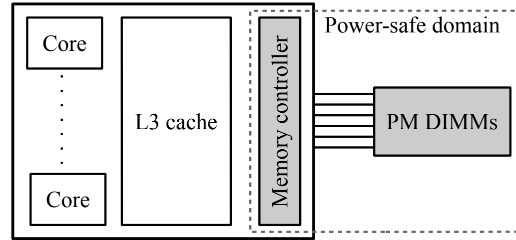

Figure 1: The power-safe domain with Optane DIMMs. The dotted box shows components whose contents survive power failure.

2 BACKGROUND

2.1 Non-volatile main memory

NVMM DIMMs attach to the CPU over the memory bus, and aim to provide latency and bandwidth close to DRAM. Optane DIMMs provide durability with a few hundred nanoseconds of latency in commodity servers. Optane DIMMs have higher density, and lower price per gigabyte than DRAM. For example, for 128 GB modules, the cost per GB is $35 for DRAM, and $5 for Optane DIMMs.

Applications use memory-mapped files to access NVMM. Loads and stores to the mapped region are cached by CPU caches. All NVMM data path operations run in userspace. The `libpmem` library provides support for managing persistent files, and fast SIMD-based persistent building blocks (e.g., memory copies). The `clwb` instruction initiates a write-back of a cache line to NVMM. To wait for writes to become persistent, applications follow one or more `clwb` instructions by a store fence (`sfence`), which blocks the processor's pipeline until all the preceding stores complete. Because NVMMs attach to the memory bus, they are directly accessible from DMA-capable peripherals such as NICs.

A write-back completes when it reaches the memory controller. In machines with Optane DIMMs, the Asynchronous DRAM Refresh feature guarantees that the contents of these DIMMs, as well as buffers in the write pending queue of the CPU's on-die memory controller survive power failure. These two components form the platform's power-safe domain. CPU cache contents may not survive power failure. This is because caches are much larger than memory controller buffers, and server power supply units may not have sufficient capacitance to flush large CPU caches after a power failure [42]. Figure 1 shows this distinction. We also consider machines with power-safe caches in our work. Battery-backed servers already provide power-safe caches [19], and future platforms will likely extend the power-safe domain to include CPU caches [42].

2.1.1 *Intel's Optane DC Persistent Memory.*

We provide a brief overview of Optane DIMM internals, based on Intel's

manual [6] . Each Optane DIMM contains several hundred gigabytes (currently up to 512 GB) of persistent media, internally divided into 256 B blocks. It includes a controller that receives 64 B commands over the DDR4 bus, and translates them into 256 B reads and writes to the persistent media. The on-DIMM controller in Optane DIMMs implements write-combining, coalescing 64 B DDR writes into larger media 256 B writes using a write-combining buffer. It also implements wear-leveling and bad block management.

Firmware on each Optane DIMM provides hardware counters for the number of 64 B commands received by the controller on the DDR bus, and the number of 256 B commands issued by the controller to the persistent media. We use the term "in-DIMM write amplification" to refer to the ratio of bytes written to persistent media, to bytes received for writing over the DDR bus. Smaller values of in-DIMM write amplification indicate more efficient use of Optane DIMMs.

NVMM emulation. Before the availability of Optane DIMMs, researchers used emulation to guide the design of NVMM-based single-machine systems and distributed systems [10, 20, 47, 51, 55, 56]. Emulation techniques include using software approaches to add latency to memory accesses, as well as hardware-based emulation platforms [10].

Recent empirical analysis by Yang et al. [53] with Optane DIMMs in single-machine systems shows that, because of fundamental but distinctive properties of NVMMs that became evident only after the real DIMMs became available, the emulation techniques fail to capture the distinctive characteristics of Optane DIMMs. They find that, compared to DRAM, the performance of Optane DIMMs depends much more on access type (reads vs writes), access patterns (random vs sequential), and access concurrency (number of threads accessing Optane DIMMs) than prior emulation accounts for. Our work finds similar evidence in networked access to NVMM.

2.2 High-performance networking

Modern commodity datacenter networks support single-digit microsecond round-trip latency and up to 100 Gbps of network bandwidth per server [21, 22]. In addition to faster NIC and switch hardware, high-performance userspace networking in the form of Remote Direct Memory Access (RDMA) [22] and the Data Plane Development Kit (DPDK) [17] is now commonplace. Researchers have redesigned several distributed systems to take advantage of fast networks, including object stores [26, 37], distributed transaction processing [16, 19, 27, 35, 49], and state machine replication [25, 28, 39].

We expect network latency to remain far above NVMM latency for the foreseeable future. Although network bandwidth continues to improve, with 200 Gbps and 400 Gbps networks on the horizon, reducing network latency is much harder. This is because in addition to propagation delay [43], a network round trip requires at least two switch port crossings (300–800 ns one-way) and two PCIe bus round trips (~400 ns per round trip). These components are already optimized for latency, so reducing network round trip time below 1.4 µs (2×300 ns + 2×400 ns) is extremely challenging.

We review the two high-level methods of accessing remote NVMM next.

One-sided RDMA. Accessing NVMM directly from NICs via RDMA is a popular approach, which has the benefit of reducing or eliminating remote CPU use. This includes one-sided RDMA accesses in application such as transaction processing [19, 50], state machine replication [39], and distributed file systems [9, 31, 51, 53].

Remote procedure calls. Another method to use NVMM in distributed systems is to treat it as conventional storage that clients access using RPCs: Clients send requests to the server's CPU, which copies volatile network buffers to NVMM and sends responses. With RPCs, the networking subsystems at the server and client are unaware of NVMM.

Although recent high-performance RPC libraries provide latency, message rate, and bandwidth that is comparable to one-sided RDMA in volatile use cases, we find that existing RPCs are much slower than RDMA NICs for bulk writes to NVMM. We improve RPC performance for this workload by using DMA engines present on the CPU die (Section 5.5). We use eRPC [28] for remote procedure calls in this work, although other libraries that provide similar performance are also sufficient. eRPC runs over both lossy Ethernet and lossless InfiniBand networks, and supports end-to-end reliability and congestion control.

2.3 Goals of this paper

A large number of prior distributed systems designed for high-speed networks target applications (e.g., transactions and state machine replication) that require data durability [16, 19, 25, 27, 28, 35, 39, 49]. These systems use DRAM to store data, often as a placeholder for future NVMM technologies. This is because the latency overhead of persisting data to SSDs on the critical path of requests is prohibitive on fast networks, and low-latency NVMMs such as Optane were unavailable until recently.

Our goal is to help future designers of high-performance NVMM-based distributed systems in answering the following question: What aspects of the system should be designed differently for real NVMM compared to DRAM? As an example, we suggest the following change in Section 5: for

efficient bulk RDMA writes to remote NVMM, the DDIO optimization, which benefits writes to remote DRAM, should be turned off.

We limit this paper primarily to primitives used to build distributed systems, including small latency-sensitive writes, bulk bandwidth-sensitive writes, and networked counters, although we include end-to-end case studies with real applications. There are two reasons for our focus on a microbenchmark analysis of the primitive building blocks: First, understanding the characteristics of these building blocks is crucial for end-to-end efficiency. Second, as we show later, the distinctive properties of NVMM make the behavior of even these simple building blocks quite complex. For example, explaining the performance of these primitives requires carefully studying the DIMMs' hardware counters.

3 EVALUATION SETUP

Our experiments use three machines with Cascade Lake Xeon CPUs (24 cores, 2.9 GHz). Each CPU has six memory channels. Each channel connects to one Optane DIMM, and one 32 GB DDR4 DRAM. Our primary evaluation machine has 256 GB Optane DIMMs; the other two machines have 128 GB. Unless stated otherwise, we measure performance on the primary machine. We run Linux kernel 4.17, which is the minimum required to expose NVMM over RDMA. We collect hardware counters for Optane DIMMs using the `ipmctl` utility. We use the `libpmem` library for persistent memory programming, configured to use AVX-512 instructions for 64 B loads and stores.

Each machine has a single-port 56 Gbps Mellanox ConnectX-3 InfiniBand NIC (PCIe 3.0 x8), connected to a Mellanox SX6036 56 Gbps InfiniBand switch. Our results also apply to newer NICs (Section 4.4), and to modern Ethernet networks, which offer similar latency and bandwidth as InfiniBand [28]. We use InfiniBand's Reliable Connected (RC) transport for one-sided RDMA. All source code used in this paper will be made publicly available.

4 LOW-LATENCY WRITES

We begin by studying the performance of latency-critical writes to remote NVMM. The key takeaway from the experiments in this section is that one-sided RDMA loses most of its latency advantage over RPCs for durable writes to remote NVMM. With this takeaway, we later choose to design our low-latency state machine replication system with RPCs instead of one-sided RDMA.

4.1 Persistent RDMA background

With RDMA, the required sequence of operations to write durably to remote NVMM is an RDMA write followed by an RDMA read, with DDIO disabled. (We term this combination an RDMA pwrite.) The reasoning is as follows: When an RDMA write completes at the client, the written data is *not* guaranteed to be present in the server's memory hierarchy, i.e., CPU caches, memory controller, and DIMMs. At this point, the written data may be in the server's NIC or PCIe buffers. The subsequent RDMA read from the client generates a DMA read at the server that flushes prior DMA writes from the server's NIC and PCIe buffers into the server CPU's memory hierarchy. With DDIO disabled, the DMA writes go directly to the NVMM instead of the L3 cache.

4.2 Durability guarantee of RDMA

Although our processor and NIC vendors (i.e., Intel and Mellanox, respectively) require using pwrites for durability, is the RDMA read-after-write sequence needed for durability in practice? Answering this question is both important and challenging. It is important because RDMA NICs sometimes provide stronger guarantees in practice than what the vendors officially support, and system designers may use such guarantees for higher performance. For example, production systems such as MPICH [30] and FaRM [14] rely on the left-to-right byte ordering of RDMA writes, which the RDMA specification and the NIC vendor prohibits [5, 34]. Similarly, if the RDMA read after the RDMA write is not necessary for durability in practice, developers will choose to omit the RDMA read.

Answering the question is challenging because it requires checking a time-based ordering relationship between two nodes in a distributed system: The server must check if the written data is persistent after the RDMA write completes at the client. This is difficult to do without synchronized clocks.

RDMA write visibility test. We designed a novel test to show that an RDMA write completed at the client may not be durable at the server. The test works by proving that such an RDMA write may not even be *visible* in the server's memory hierarchy, and is therefore outside the server's durable power-safe domain. Therefore, designers must include the RDMA read after the RDMA write if they require persistence.

Our test works around the clock synchronization issue by using one machine with two RDMA NICs. We run a server and a client thread on this machine, which use different NICs. The client thread sends an RDMA write to the server thread, and sets an in-memory flag after it gets the RDMA write completion. This write goes through the network switch, so our experiment accurately emulates a setup with the client and server on different machines. On detecting a raised flag, the server thread checks if the written data is visible. We find that the written data is frequently (many times every second) invisible to the server.

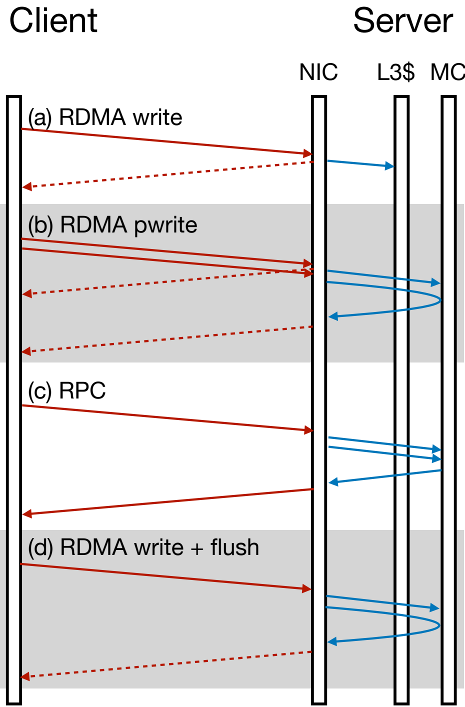

Figure 2: Network and PCIe operations involved in writing to remote NVMM with different methods. Red arrows from the client to the server's NIC are network packets. The dotted arrows are NIC-generated RDMA acknowledgments. Straight blue arrows between the server's NIC and its cache (L3) or memory controller (MC) are PCIe DMA or MMIO writes; the curved ones are DMA reads. The server's CPU (not shown) is involved in persisting RPC requests.

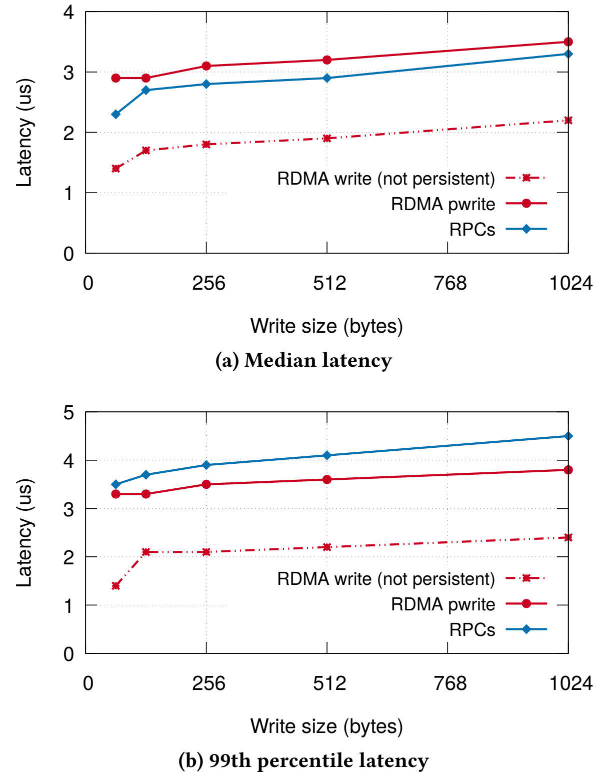

Figure 3: Latency of sequential writes to remote Optane memory with RDMA and with RPCs

4.3 Measurements

We next evaluate the latency added by disabling DDIO and the additional RDMA read. We measure the latency of different methods of writing small items (64–1024 B) to remote NVMM. We use two machines in our cluster (Section 3). We run a single-threaded server on the machine with 256 GB Optane DIMMs, and a single-threaded client on another machine. The client measures the round-trip latency of writes to sequential locations at the server. Figure 3 shows the median and 99th percentile latency of three methods, shown as parts (a)–(c) in Figure 2.

(1) **RDMA write.** As a baseline, we measure the latency of an RDMA write issued by the client, with DDIO enabled at the server. Recall that this method does not provide durability because the server's NIC may reply before its DMA write reaches the server's memory hierarchy. As a result, the median latency of a 64 B RDMA write is only 1.4 µs.

(2) **RDMA pwrite.** Disabling DDIO and adding a flushing RDMA read pipelined with the RDMA write increases median latency of 64 B writes by over 2x to 2.9 µs. In our test implementation, the client's RDMA write is "unsignaled," so the client's NIC does not generate a completion for the RDMA write. The client measures the latency of the read-after-write operation using the RDMA read's completion. Of the 1.5 µs latency increase (from 1.4 µs to 2.9 µs), only 100 ns is due to disabling DDIO, and the remaining 1.4 µs is due to the flushing RDMA read.

(3) **RPC.** With RPCs, the server's NIC writes packets into volatile ring buffers in the server's L3 cache. The server's CPU detects new requests via busy polling, persists them to Optane memory, and replies to the client. Median latency with RPCs is up to 20% lower than an RDMA pwrite, e.g., 2.3 µs for 64 B writes. 99th percentile latency with RPCs is up to 18% higher than RDMA pwrites, but we believe the higher tail latency of RPCs is an artifact of the old ConnectX-3 NICs in our cluster (more in Section 4.6). 99.9th percentile latency

(not shown in Figure 3) for 1024 B writes is 4.3 µs with RDMA pwrites, and 4.8 µs with RPCs.

The takeaway from our measurements is that switching from volatile DRAM writes to durable NVMM writes greatly reduces the latency advantage of one-sided RDMA over RPCs. Overall, the latency of the two approaches is similar: RPCs have slightly better median latency, and slightly worse tail latency. There is a fundamental reason behind this similarity: RPCs and pwrites have similar network and PCIe operation on their critical path (Figure 2), which are the factors that largely govern the end-to-end latency of simple primitives [26]. In contrast, an RDMA write avoids one PCIe round trip at the server on its critical path.

In addition to comparable latency as RDMA pwrites, RPCs are simpler to use and can reduce round trips needed to complete distributed system operations [26, 32]. For these reasons, we find that RPCs are well-suited to building low-latency distributed systems with NVMM.

4.4 Newer NICs

Although our cluster has old Mellanox ConnectX-3 NICs, our results apply to clusters with newer NICs. (Unfortunately, we cannot update the NICs on our NVMM cluster because we do not have physical access to the cluster.) This is because although newer RDMA NICs such as ConnectX-4 and ConnectX-5 NICs greatly improve the message rate and scalability of RDMA [28, 35] by 5x or more, factors relevant to our work such as network latency, PCIe latency, and PCIe DMA accesses are largely unchanged.

For example, on our 56 Gbps ConnectX-3 cluster, the median latency of an RDMA write is 1.4 µs, and the median latency of an RDMA pwrite is 2.9 µs (Section 4.3). We repeat these latency tests on a different cluster that has newer 100 Gbps ConnectX-5 InfiniBand NICs. Since this cluster does not have NVMM, we keep the target buffers of RDMA operations in DRAM, which gives our ConnectX-5 setup a small (≈100 ns) latency advantage. With ConnectX-5, the latency of an RDMA write is 1.3 µs, and the latency of an RDMA read-after-write is 2.5 µs. Despite the advantage, these latencies are only slightly better than the corresponding ConnectX-3 latencies.

4.5 Future RDMA extensions

There is ongoing work to specify and create a new RDMA primitive called "RDMA flush" for NVMM, available in an RFC by Talpey et al. [44]. If the RDMA flush specified in this RFC becomes available, one can use an RDMA write plus RDMA flush instead of an RDMA write plus RDMA read to achieve durability. In addition, the RDMA flush primitive will allow keeping DDIO enabled system-wide because the RDMA flush implementation will handle the flushing of CPU caches.

Replacing the flushing RDMA read with an RDMA flush will have lower latency than pwrites if NIC designers allow merging the RDMA flush request into the preceding RDMA write packet; Kim et al. [29] implement such an approach by modifying an RDMA NIC's firmware. Figure 2(d) shows the network and PCIe operations used when the flush is merged with the write. In contrast, the flushing RDMA read in pwrites requires a separate packet. Similar to the RDMA read, the RDMA flush operation will require issuing a DMA read or atomic operation on the PCIe bus, which are the only fast methods of flush pending DMA writes over PCIe.

However, we expect that the latency reduction from using an RDMA flush instead of RDMA read will be small, and the resultant latency to be not much better than RPCs. This is because both approaches require similar network and PCIe operations on their critical path. Comparing parts (c) and (d) of Figure 2 conveys this idea visually.

4.6 Low-latency state machine replication

We next design and evaluate an example low-latency durable distributed system—state machine replication—using RPCs. State machine replication is an important low-latency application in datacenters, often used to store small metadata items. In the past, SMR systems that achieved microsecond-scale latency stored their data in volatile memory [25, 28, 39]. NVMM allows microsecond-scale SMR latency with durability.

Leader-based SMR protocols such as Raft [36] roughly follow the following failure-free operation: a client sends a state machine command to the leader. The leader records the command in its log and forwards it to followers; followers reply after recording the command in their own log. After receiving acknowledgments from a majority of followers, the leader replies to the client that the command has been successfully replicated.

We add persistence support to the volatile, RPC-based Raft state machine replication provided by Kalia et al. [28]. Their implementation uses a production-grade Raft codebase [2], so our results are relevant to real systems. Unlike Kalia et al. [28]'s Raft implementation that stores the SMR command log in DRAM, we store commands in NVMM. The key-value store is *volatile* in our implementation, too: only the command log is persistent, which is sufficient to recreate the key-value store after failure. We implement the command log using a simple array. Saving an entry to our log requires two dependent persistent writes: one to insert an entry into the log, and one to update the log's tail pointer. There are four dependent persistent writes to NVMM on the end-to-end critical path of each request issued by our client: two

Log storage	Median (µs)	99% (µs)
DRAM	5.6	9.1
Optane DIMMs	6.6	10.2

Table 1: Three-way Raft replication latency

at the leader, and two at the followers. Followers work in parallel, so only one of them is on the critical path.

We also use Kalia et al. [28]'s benchmark workload: We run a three-way replicated key-value store that maps 16 B keys to 64 B values. We use one client that issues PUT requests to this store, keeping one request outstanding at a time. Since we have only three machines in our NVMM cluster, we colocate the client with the third replica. We ensure that the third replica is not the Raft leader, so the client's request to the leader goes over the network instead of looping back through its NIC.

Table 1 compares the three-way replication latency of our system measured at the client, with the SMR command log stored either in DRAM or in NVMM. Using NVMM instead of DRAM adds only 1 µs and 1.1 µs to the median and 99th percentile latency, respectively.[2]

Comparison with one-sided RDMA. The current state-of-the-art RDMA-based SMR system is DARE [39]. However, DARE replicates to DRAM and its codebase does not support NVMM. We chose to not port DARE to NVMM because the best-case latency of such a system (named DARE-persistent) would be substantially worse than ours. Replication in DARE-persistent requires three network round trips: one volatile RPC from the client to the SMR leader, and two dependent *persistent* writes from the leader to followers for log replication. We measured that the median latency of volatile RPCs in our cluster is 2.3 µs. The median latency of one persistent write with one-sided RDMA is 2.9 µs (Figure 3). Therefore, the end-to-end median latency of DARE-persistent is at least 8.1 µs (2.3 µs + 2×2.9 µs), substantially higher than the 6.6 µs with RPCs. This comparison ignores other sources of latency in DARE such as computation at the leader, which are included in our system's latency.

5 HIGH-BANDWIDTH BULK WRITES

We next study challenges and solutions for bulk writes to remote NVMM for both networking approaches (i.e., RDMA and RPCs). Efficient bulk writes are important for performance in applications such as NVMM-based distributed file systems [9, 31, 51, 53], distributed logging [12], backups, etc. We contribute new optimizations that improve the performance of bulk writes to remote NVMM for both RDMA and RPC approaches. The key takeaways from the experiments in this section are as follows.

First, for RDMA writes, we find that, counter-intuitively, *disabling* DDIO at the server improves bandwidth for bulk writes. Put briefly, DDIO reduces bulk write bandwidth because it injects the sequential DMA writes into L3 cache, which later evict into NVMM in near-random order, reducing performance. We discuss this mechanism in detail later in this section.

Second, for RPCs, we find that existing RPC libraries provide low throughput for bulk writes to remote NVMM. This happens because CPU cores are much slower than RDMA or DMA engines at writing to NVMM. While eRPC can achieve up to 75 Gbps for bulk writes to remote *volatile* memory [28] with one core, it achieves only 22 Gbps for bulk writes to remote NVMM. We show how RPC-based approaches can use an on-CPU DMA engine to offload the copying of volatile RPC buffers to NVMM buffers, improving bandwidth by 2.3x (reaching line rate on our setup).

5.1 Discussion on disabling DDIO

In subsequent sections, we advocate disabling DDIO as a crucial optimization for efficient bulk RDMA writes to remote NVMM. Because disabling DDIO is necessary for durability with one-sided RDMA with current power-safe technologies, it might seem that all RDMA-based NVMM systems will obviously disable DDIO. We emphasize that this is not the case. Developers may choose to not disable DDIO in such systems for four reasons.

1. Recent RDMA-based NVMM systems such as Orion [51], Assise [9] and PASTE [23] use RDMA (or PCIe DMA in PASTE) writes to place network data in remote NVMM, and then use the remote CPU to flush the written cache lines to NVMM. These systems provide durability without disabling DDIO. However, they suffer from the DDIO-induced NVMM access pattern randomization.
2. DDIO is an important optimization in volatile systems, so developers may wish to keep it in NVMM-based systems that do not require persistence, e.g., cases where NVMM serves as a large DRAM, or in persistent applications that require weak consistency.
3. Disabling DDIO is unnecessary for durability in current battery-backed servers [19], and in future platforms in which CPU caches survive power failure [42].
4. DDIO is on by default, and developers may misconfigure the system by failing to disable it.

[2]The 99th percentile latency in our cluster with the command log in DRAM is noticeably higher (by 44%) than in the evaluation of Kalia et al. [28]. This is because our cluster uses older ConnectX-3 InfiniBand NICs, whereas Kalia et al. [28] use newer and faster ConnectX-5 NICs.

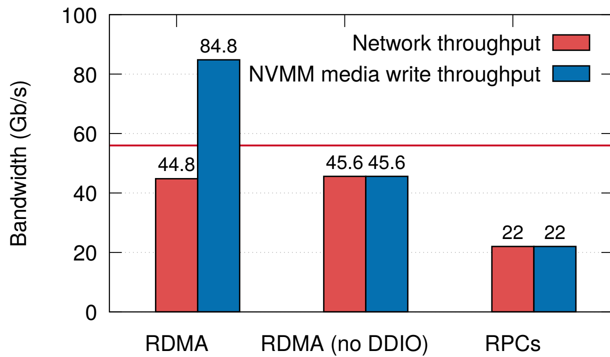

Figure 4: The red bars show the bandwidth of bulk writes to remote NVMM with one-sided RDMA (with and without DDIO) and RPCs. The red line shows our network's line rate (56 Gbps). The blue bars show the throughput of 256 B writes to the DIMMs' persistent media, which can exceed network bandwidth due to in-DIMM write amplification (Section 2.1.1).

5.2 Improving RDMA bandwidth

We measure the performance of bulk writes to remote NVMM with RDMA and RPCs as follows. We run a single-threaded server and client process on two separate machines. The client issues large writes to remote NVMM, pipelining its write requests (implemented with either RDMA or asynchronous RPCs) to avoid blocking. In the RPC mode, an application-level request handler at the server manages all NVMM access, i.e., we do not modify eRPC's internals. Figure 4 shows the client's throughput for 2 MB writes with RDMA (with and without DDIO) and RPCs. For each approach, we also show the throughput of writes to the persistent media at the server, reported by the Optane DIMMs' hardware counters. We discuss only RDMA throughput in this section. In the next section, we show how we improve the low throughput of RPCs.

RDMA writes achieve 45 Gbps, which is close to the maximum achievable throughput on our 56 Gbps InfiniBand network. Although RDMA writes saturate our network, our measurements using the Optane DIMMs' hardware counters show that the RDMA writes are inefficient because they cause a high in-DIMM write amplification of around 2x. This means that 2x more bytes are written to the DIMM's persistent media than are received over the network (Section 2.1.1). Figure 4 also shows that in our test, bandwidth of data written to the Optane DIMMs is around 10.6 GB/s—2x higher than the bandwidth achieved over the network.

This write amplification is problematic for two reasons. First, even in servers where all six memory channels are populated with Optane DIMMs, it leaves little bandwidth for use by other applications. The six Optane DIMMs on our server support a total write throughput of around 13 GB/s [53], so writing to the DIMMs at 10.6 GB/s leaves only 2.4 GB/s for other applications. Second, because Optane DIMMs are expensive, we expect some datacenter operators to use fewer DIMMs per server. We re-ran the previous experiment with only one Optane DIMM at the server, which shifts the bottleneck from the InfiniBand network to the Optane DIMM. In this setup, the bulk RDMA write throughput is 1.15 GB/s, half of the one DIMM's peak 2.3 GB/s persistent media write throughput.

We found that the high in-DIMM write amplification happens because the CPU cache is not optimized for NVMM. Specifically, the DDIO feature of the L3 cache turns the sequential RDMA accesses into near-random writes to the Optane DIMMs, as follows. With DDIO enabled, RDMA NICs inject data into L3 cache, and the cache lines eventually evict to the Optane DIMMs in some proprietary eviction order that is near-random in practice. Although CPU caches typically implement some form of least recently used eviction policy within a cache set, consecutive cache lines typically map to distinct cache sets. When an Optane DIMM fails to coalesce a 64 B write with writes to adjacent locations, it issues 256 B read-modify-write commands to its persistent media, resulting in write amplification. Yang et al. [53, Sec 5.2] make a related observation for CPU cores writing data to caches instead of writing directly to NVMM with non-temporal stores.

Disabling DDIO eliminates the access pattern randomization and write amplification. Figure 4 shows that disabling DDIO makes bulk RDMA writes more efficient. Although the network throughput is still near line rate, the write bandwidth to the Optane DIMMs measured using the DIMMs' hardware counters decreases from 84.8 Gbps to 45.6 Gbps.

Challenges for hardware designers. To our knowledge, existing proposals for cache eviction policies for NVMM operate at a cache line granularity [13, 40, 48]. Compared to eviction policies for DRAM, these policies account for the higher energy cost of writes than reads with NVMM. However, they do not handle the important case where the NVMM's block size is larger than the 64 B cache line size, e.g., 256 B in the case of Optane DIMMs. The results from this paper in the networked context, and from Yang et al. [53] in the single machine context, suggest that an efficient cache eviction policy for NVMM should account for block sizes larger than cache lines.

RPC performance. Figure 4 also shows that the RPC-based approach achieves low performance—only around 22 Gbps. This is as expected: the server thread receives data over the network at line rate (around 46 Gbps), and copies it to Optane DIMMs at around 34 Gbps (equal to the single-core NVMM write speed [53]). The resulting bandwidth is

$(1/46 + 1/34)^{-1} = 19.5$ Gbps, which is close to 22 Gbps. Next, we show how we can improve RPC throughput to match the network speed by leveraging DMA engines present on modern CPUs.

5.3 DMA engine background

An on-die DMA engine, such as Intel's I/O Acceleration Technology (IOAT) DMA engine, aims to accelerate memory copies in applications such as high-speed networking and storage. Processors have included a DMA engine for over a decade, but these accelerators have seen little use. One notable recent use case is in high-speed networking at Google, where the IOAT DMA engine copies network packets from the NIC's ring buffers to application buffers [32]. In contrast to their use case that targets volatile buffers, our work focuses on DMA acceleration for NVMM. Prior to us, Assise [9] uses on-die DMA engines to bypass cache coherence traffic in cross-NUMA copies to Optane DIMMs. We focus on DMA optimizations within one NUMA node, and present new DMA engine techniques and measurements.

One reason why developers have ignored DMA acceleration is that, until recently, their offered copy speed was fairly low, even lower than the basic `memcpy` speed of a single CPU core. For example, Intel's evaluation of their IOAT DMA engine on Broadwell processors, which were released in 2016, finds that for bulk transfers, the DMA engine's copy speed (4.4 GB/s) is around half that of one CPU core (8.0 GB/s) [4]. Intel recently improved their DMA engine speed to ~15 GB/s in Skylake and newer processors. DMA engines on AMD's second-generation EPYC processors offer even higher rates up to 70 GB/s [1], indicating that DMA acceleration is a promising strategy for the faster networks and NVMMs in the future, too.

Software support for IOAT DMA. We use a userspace driver for the IOAT DMA engine from the DPDK library [24]. An application thread interacts with the DMA engine with an asynchronous, lock-free API by posting work descriptors to per-thread DMA "channels." One limitation of the IOAT DMA engine is that it operates on physical addresses. On Linux, privileged applications can translate virtual addresses to physical addresses by parsing `/proc/self/pagemap`. We use 2 MB hugepages allocated at boot time, and cache physical addresses after the first translation. This is safe because on Linux, the kernel does not change the physical address of such hugepages. In the steady state, our benchmarks do not perform any virtual-to-physical translation.

5.4 IOAT DMA microbenchmarks

This section presents, to our knowledge, the first in-depth study of the performance of the IOAT DMA engine on modern Intel CPUs. We focus on aspects that affect performance of bulk DMA writes to NVMM.

Recall that we are using the IOAT DMA engine to improve single-core RPC performance for bulk writes to remote NVMM, which is substantially lower than RDMA (Figure 4). Therefore, our microbenchmarks compare the copy performance of the IOAT DMA engine to software memcpy running on one CPU core. We do so by measuring the speed at which each approach can copy 1 kB–128 kB buffers to a large 4 GB buffer that is much larger than the CPU's cache. This benchmark is representative of real networked workloads that copy data from cached RPC message buffers to in-memory logs or key-value stores.

Persistence of DMA operations. By default, the IOAT DMA engine writes data to CPU cache using a feature called Direct Cache Access (DCA). DCA for DMA engines is similar to DDIO for NICs. We found that we can disable DCA, and thereby force the engine's writes to go directly to the memory controller, achieving durability in theory. Although Intel does not yet officially support this approach for durable writes to NVMM with IOAT DMA, the mechanism likely works because it is identical to RDMA writes with DDIO disabled. (RDMA writes are also handled by a DMA engine on the CPU.) By showing the large improvement from IOAT DMA for NVMM access, we hope to convince CPU vendors to officially support using DMA engines for durable NVMM writes.

Ordering of IOAT DMA operations. For DMA measurements, we found that keeping a window of multiple DMA operations in flight improves performance; we show performance with one (W = 1) and multiple (W = 8) outstanding DMA operations in flight. We found that pipelining more than eight DMAs did not further improve performance on our CPUs.

NVMM applications often require ordered persistence, i.e., a write should become persistent only after the previously issued write becomes persistent. For DMA operation pipelining to be usable in NVMM applications that require ordered persistence, the DMA engine should provide the following guarantee: updates from an operation should not be visible in the CPU's memory subsystem before all updates from previous operations on the same channel are visible. The publicly-available IOAT DMA documentation does not specify this guarantee. We created a stress test in which a monitoring thread checks for ordering violations, and verified that the IOAT DMA engine upholds this guarantee in practice.

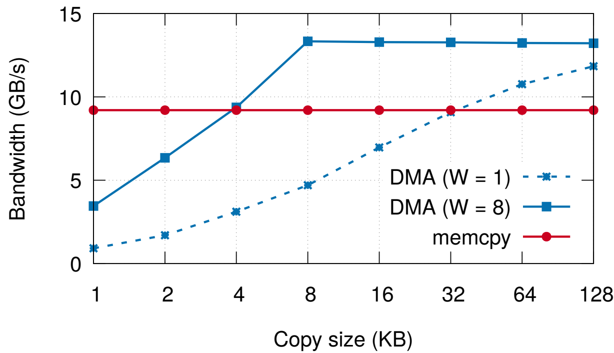

Figure 5: Comparison of IOAT DMA and memcpy for bulk writes to volatile memory

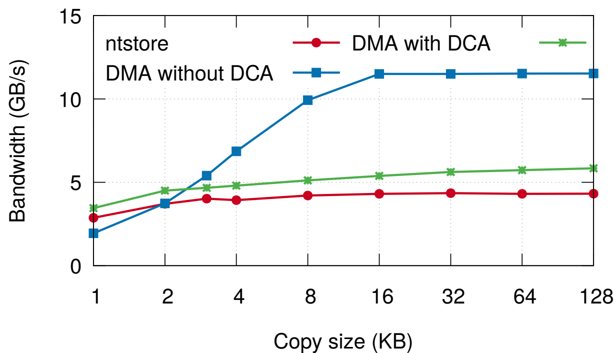

Figure 6: Comparison of IOAT DMA and ntstores for bulk writes to NVMM. We keep eight writes in flight for the DMA experiments. Unlike "ntstore" and "DMA without DCA", "DMA with DCA" (green line) does not provide durability.

5.4.1 Volatile memory copy performance. To establish a baseline of IOAT DMA performance without hitting the NVMM write bandwidth bottleneck, we first run the experiment with both source and destination buffers in volatile memory. Figure 5 shows the results with two window sizes (one and eight). For 8 kB or larger writes, the DMA engine delivers around 13.3 GB/s with a window of eight writes in flight, which is 44% higher than a 9.2 GB/s achieved by one core with memcpy. Our results also show that, even for moderately-sized copies of 4 kB or more bytes, pipelining multiple DMA operations improves performance by up to 3x.

5.4.2 NVMM copy performance. We now run the previous experiment with the destination buffer in Optane DIMMs. For the software baseline, we use an optimized persistent memcpy that uses AVX-512 non-temporal stores (ntstore) from the Persistent Memory Development Kit library. Figure 6 shows the results. We find that the default IOAT DMA configuration achieves at most 5.8 GB/s, which is only slightly better than ntstores (4.3 GB/s). Our single-threaded ntstore performance is the same as that reported previously by Yang et al. [53].

We used the hardware counters on Optane DIMMs to diagnose the DMA engine's low performance. We found that the default IOAT DMA configuration results in a high in-DIMM write amplification of around 2x. We found that the root cause is similar to DDIO's randomization effect in RDMA writes: by default, the IOAT DMA engine writes data to CPU caches using DCA, which trickles down to the Optane DIMMs in semi-random order. *Recall that the fundamental cause of this inefficiency is that the CPU cache is not optimized for NVMM.*

Disabling DCA for the IOAT DMA engine increases peak throughput by 2x from 5.8 GB/s to 11.5 GB/s. Both with and without DCA, the bottleneck lies in the Optane DIMMs' write throughput to persistent media (approximately 13 GB/s for the six Optane DIMMs in our system [53]). The difference is that with DCA enabled, write amplification wastes almost half the bandwidth.

For small 1–2 kB copies, disabling DCA reduces the DMA engine's performance. This happens likely because for small copies, the throughput is much lower than the Optane DIMMs' peak write throughput, and DCA's advantage of writing directly to caches makes the copies faster. However, DMA with DCA enabled does not provide durability.

5.5 Optimizing RPCs with DMA

We chose to add the DMA optimization in the application-level RPC request handler at the server. We considered adding DMA support inside eRPC by using DMA to directly copy application data from the NIC's receive ring buffers to NVMM. However, receive ring buffers are MTU-sized and therefore small—typically ≈1500 B on commodity Ethernet. The DMA engine is more efficient for larger copies (Figure 6). Using DMA at the application level allows us to work with large de-fragmented messages instead of small packets, improving performance.

Our solution involves the following steps at the server. First, eRPC's event loop running at the server receives packets from the network. It de-fragments packets into messages by copying them from ring buffers into the application's volatile buffer. Then, eRPC's event loop invokes an application-level request handler in the same thread. The handler copies the message to NVMM asynchronously using the DMA engine. The handler returns to eRPC before the copy completes, overlapping network packet processing inside eRPC with the DMA copy. The server sends a response message back to the client after the copy completes.

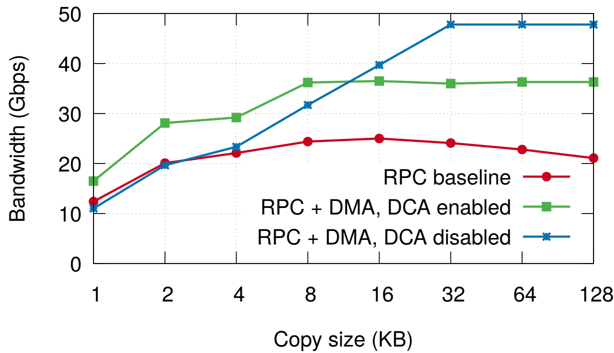

Figure 7: Effect of IOAT DMA acceleration on RPC throughput. Unlike "RPC baseline" and "RPC + DMA, DCA disabled", "RPC + DMA, DCA enabled" does not provide durability.

Figure 7 compares the performance of bulk writes to remote NVMM with default RPCs, and with our DMA acceleration optimization (termed RPC + DMA). We show performance for RPC+DMA both with Direct Cache Access enabled and disabled. Note that RPC+DMA does not provide durability when DCA is enabled. We make two observations:

(1) RPC+DMA with DCA disabled provides the highest performance for 16 kB and larger writes. For smaller writes, the RPC+DMA with DCA *enabled* performs better. This is because the latter benefits from having to write data to only the CPU cache and not the NVMM, which is faster when the NVMM DIMMs are under low load (Section 5.4.2).
(2) RPC+DMA with DCA disabled achieves line rate with 32 kB or larger writes. The other approaches fail to achieve line rate. For large 128 kB copies, RPC+DMA is 2.3x faster than the eRPC baseline, and 32% faster than RPC+DCA with DCA enabled. The latter fails to achieve line rate because of NVMM access pattern randomization caused by DCA.

6 PERSISTENT LOG

This section presents our findings on another important primitive in distributed systems: Persistent logs are building blocks of critical application such as transaction processing systems [12, 18, 19], state machine replication, and remote backups. We focus on small log records up to a few thousand bytes in size, such as those generated by online transaction processing workloads like TPC-C [45]. Techniques from the previous section on optimizing bulk writes to remote NVMM are useful for larger log writes.

In a straightforward log design, the log consists of two parts: a log buffer, and an 8 B counter that holds the number of bytes written to the log buffer. To append data to the log, we first issue a durable write to the log buffer, followed by a durable write to the counter. (A durable write includes a store instruction, followed by a cache line flush using the `clwb` instruction, and an `sfence`.) We found that this straightforward design performs poorly, which is surprising because sequential writes to NVMM perform well [53].

We found that the repeated writes to the counter cause the poor append performance. To understand the counter's performance in isolation, we wrote a benchmark that repeatedly issues durable writes to one 8 B location in NVMM. We found that we can do only 1.3 million updates per second, corresponding to 770 ns average latency per persistent write. Note that a persistent write begins execution only after the previous persistent write is complete, so inverse throughput of persistent writes equals their average latency. For comparison, the average latency of small sequential durable writes to with Optane DIMMs is only around 120 ns.

6.1 Diagnosis: Cache line invalidation

Why are repeated writes to the same memory location slow? This is an important question because such writes are common in networked systems, e.g., in log-based distributed system components such as write-ahead logging and state machine replication, in systems that use persistent timestamps [11], and in key-value stores while handling skewed workloads [15].

At first, we wrongly hypothesized that slow repeated writes are due to wear-leveling inside the Optane DIMMs: Each persistent block in Optane DIMMs supports a finite number of erase cycles. A controller inside the DIMM prevents repeated erasures of the same block by spreading out the writes over multiple blocks. Repeated writes to the counter trigger wear-leveling, causing a slowdown. After examining the DIMM's hardware counters, we found that this hypothesis in incorrect because the volatile power-safe write-combining buffer inside the Optane DIMMs absorbs almost all writes to the counter. The DIMMs' hardware counters report that almost no writes to the log's counter reach the persistent media. In addition, our experiments showed that the poor performance is not specific to NVMM Optane DIMMs: We also see low performance when we use DRAM for the 8 B location instead of NVMM, while still using the `store-clwb-sfence` instruction combination.

The actual reason for poor performance of repeated writes to the same cache line is that CPU caches are not optimized for NVMM. Contrary to the expected behavior and common understanding, `clwb` invalidates and flushes the target cacheline [7]. `clwb` is supposed to improve upon the older `clflushopt` instruction—which invalidates the target cacheline—by retaining flushed cachelines in the CPU cache. However, on current processors that support NVMM (i.e.,

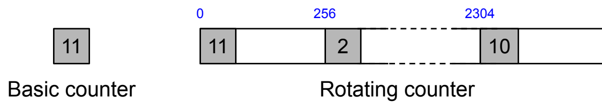

Figure 8: A comparison of the basic counter design and our rotating counter, showing the state of the two counters after 11 increments. The blue numbers show the addresses of the ten chunks in the rotating counter.

up to Intel Cascade Lake CPUs), `clwb` behaves identically to `clflushopt`. The unexpected behavior of `clwb` is correct, however. Per the documentation of `clwb`: "The line *may* be retained in the cache hierarchy in non-modified state." (emphasis is ours). Repeated invalidation of the same cache line results in poor performance.

6.2 Rotating counter

We designed a "rotating counter" that avoids repeated writes to the same memory location by spreading writes to multiple memory blocks. It internally uses ten contiguous 256 B blocks in NVMM. Other values of the number of contiguous blocks or the size of each block showed worse performance. With 256 B blocks, blocks are striped across the six Optane DIMMs, which contributes to the improved performance. We write the first counter update to the 8 B "chunk" at the start of the first block, the second to the start of the second block, and so forth. After ten updates, we wrap around to the first block. During normal operation, we maintain the counter's latest update in volatile memory, so reading its value is cheap. During failure recovery, we restore the counter's state by reading from the persistent memory file, and taking the maximum update value across the ten chunks. Figure 8 diagrams our rotating counter in comparison to a basic 8 B counter.

The rotating counter supports ten million updates per second, 7.6x higher than using one 8 B location. The rotating counter requires more space than the basic counter–2560 B in NVMM instead of 8 B. This overhead may be negligible for persistent logs in database applications, since such logs typically store gigabytes or more of data.

6.3 Extension to rotating registers

The rotating counter technique above requires increasing updates: taking the maximum value among the ten chunks does not work if we wish to also support decrements to the 8 B location. We created a novel method to extend our approach by removing this limitation, thereby supporting arbitrary updates. Our method works by using XOR instead of MAX as the recovery function, and by cleverly constructing updates to the chunks.

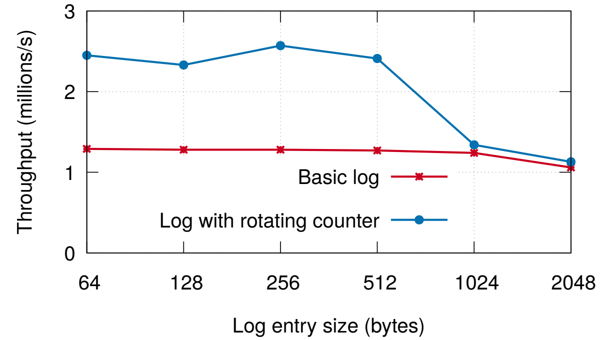

Figure 9: Throughput of a persistent log appends with a basic 8 B counter, and a rotating counter

Our persistent rotating register `R` consists of ten 256 B chunks `R[0], ..., R[9]`. Assume that for the ith update, we wish to set `R = x`. As in the rotating counter, we will write to chunk `j = i % 10` for this update. Instead of simply setting `R[j] = x`, we set `R[j] = R[0] ⊕ ... ⊕ R[j - 1] ⊕ x ⊕ R[j + 1] ⊕ ... ⊕ R[9]`. (We assume `j` is between 2 and 7 for convenience in writing this expression.) During failure-free operation, we maintain copies of the chunks in CPU cache, so computing the right hand side of the expression is cheap because it requires no NVMM accesses.

If the machine crashes after we set `R[j]`, we need to recover `x` without knowing `j`. (Persisting `j` for each update would reduce performance.) The recovery works as follows. Recall that any value XOR-ed with itself is zero. Therefore, after the write to `R[j]` becomes persistent, `R[0] ⊕ ... ⊕ R[9] = x`. Each update maintains this invariant, starting from the initial zero value of all chunks. During failure recovery, we obtain the latest value of of R by XOR-ing `R[0]` through `R[9]`.

6.4 End-to-end performance

We evaluate the benefit of the rotation technique described above in a networked environment. We created a single-threaded log server that receives log entries from multiple remote clients via RPCs. On receiving a log entry, the server thread appends it to a local log in NVMM. The server uses either a single 8 B location or a rotating counter for the head pointer of its log.

Figure 9 compares the persistent log append rate with the two counter choices. For up to 512 B appends, using a rotating counter improves the append rate by 80–90%, reaching 2.6 million updates per second with 256 B updates. For larger appends, most of the time is spent in writing to the log buffer, so the choice of counter has little effect.

7 RELATED WORK

Distributed systems with emulated or real NVMM. Several recent distributed transaction processing systems use DRAM as a placeholder for NVMM to store data and transaction logs [18, 27, 49]. A key difference between the design of these systems is the extent to which they use one-sided RDMA and what they handle with RPCs. Our work shows that we must revisit this longstanding RDMA-vs-RPC debate for NVMM. For example, Wei et al. [49] find that one-sided RDMA writes are much faster than RPCs for logging to remote DRAM in distributed transactions. However, using durable logging to remote NVMM instead of volatile logging to remote DRAM reduces the advantage of RDMA (Figure 3). Because RPC-based designs have other advantages such as simplicity and higher scalability [28, 32], the much-reduced latency benefit of one-sided RDMA over RPCs for logging may shift the balance in favor of RPCs.

The work of Yang et al. [52] on reducing the overhead of registering huge NVMM memory regions with RDMA NICs is orthogonal but complementary to ours. Our experiments register only small memory NVMM regions (a few tens of gigabytes) for RDMA, which was sufficient to expose the inefficiencies in current CPU cache designs. File systems such as Orion [51] and Assise [9] seek to bring the benefits of NVMM to file operations. The bulk RDMA writes in these systems could benefit from our optimizations (Section 5)

Single-machine NVMM systems. Our work is closest in spirit to the work on empirical guidelines for NVMM use by Yang et al. [53]. The primary distinction is that we focus on systems issues that arise specifically in networked workloads. Similar to their conclusions, we find that prior results in distributed systems that used emulated NVMM will need re-evaluation. Mnemosyne [47] presents a novel "torn-bit" log for NVMM that requires only one persistent write per log append. They achieve this by using the atomicity of 8 B persistent memory writes, and reserving one bit per 8 B word as a marker. In contrast, our persistent log (Section 6) requires a second write to the log's head pointer. We plan to compare the performance of the two log designs in the future. Our design has the advantage that the log entries are intact instead of garbled as in Mnemosyne's torn-bit log. As a result, it may be faster and simpler (e.g., during debugging) to read from the log. In addition, our rotation technique applies to applications other than logging, such as counters and timestamps. Concurrent to our work, Chen et al. [15] discovered the poor performance of repeated durable writes to the same cache line. They hypothesized that wear-leveling inside the NVMM or some unexplained blocking of `clwb` causes the low performance. Our work provides the exact reason for the low performance, and an optimization for networked counters and registers to achieve high performance.

8 CONCLUSION

Our work presents solutions to a variety of challenges that arise when building networked systems with NVMM. Our contributions include an empirical evaluation of networking options for NVMM, new experiments and optimizations for achieving low latency and high bandwidth access to remote NVMM, and case studies of two high-performance NVMM-based networked systems. We find that current CPU caches are ill-optimized for NVMM, which causes several performance regressions. Put together, these results highlight the importance of careful attention to fundamental and distinctive networking-related properties of real NVMM devices.

Acknowledgments. This work was supported by funding from the National Science Foundation under award 1700521, and by Intel via the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS). We are grateful to Intel for providing us access to a cluster with Intel Optane DC Persistent Memory.

REFERENCES

[1] 2019. Accelerating Intra-Host PVRDMA Storage Traffic in a Future Dell AMD Server. Talk at VMWorld 2019.

[2] 2020. C implementation of the Raft Consensus protocol. https://github.com/willemt/raft.

[3] 2020. Distributed Asynchronous Object Storage Stack. https://github.com/daos-stack.

[4] 2020. Fast memcpy with SPDK and Intel I/OAT DMA Engine. https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.

[5] 2020. InfiniBand Architecture Specification Volume 1. https://cw.infinibandta.org/document/dl/7859.

[6] 2020. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[7] 2020. Intel's CLWB instruction invalidating cache lines. https://stackoverflow.com/questions/60266778/intels-clwb-instruction-invalidating-cache-lines.

[8] Aerospike 2020. Aerospike Performance on Intel Optane Persistent Memory. https://www.aerospike.com/blog/performance-on-intel-optane-persistent-memory/.

[9] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan KostiÄĞ, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2019. Assise: Performance and Availability via NVM Colocation in a Distributed File System. (2019). arXiv:1910.05106 [cs.DC]

[10] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage and Recovery Methods for Non-Volatile Memory Database Systems. In *Proc. ACM SIGMOD*. Melbourne, Australia.

[11] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment.*

[12] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: a shared log design for flash clusters. In *Proc. 9th USENIX NSDI*. San Jose, CA.

[13] Nathan Beckmann, Phillip B. Gibbons, Bernhard Haeupler, and Charles McGuffey. 2019. Writeback-Aware Caching (Brief Announcement). In

The 31st ACM Symposium on Parallelism in Algorithms and Architectures.

[14] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. 2020. A1: A Distributed In-Memory Graph Database. In *Proc. ACM SIGMOD*. Portland, OR, USA.

[15] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[16] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)* (London, UK).

[17] DPDK 2017. Data Plane Development Kit (DPDK). http://dpdk.org/.

[18] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proc. 11th USENIX NSDI*. Seattle, WA.

[19] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[20] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)* (London, UK).

[21] Daniel Firestone et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proc. 15th USENIX NSDI*. Renton, WA.

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proc. ACM SIGCOMM*. Florianopolis, Brazil.

[23] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *Proc. 15th USENIX NSDI*. Renton, WA.

[24] Intel. 2013. Intel Data Plane Development Kit (Intel DPDK). http://www.intel.com/go/dpdk.

[25] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proc. 13th USENIX NSDI*. Santa Clara, CA.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proc. ACM SIGCOMM*. Chicago, IL.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proc. 12th USENIX OSDI*. Savannah, GA.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proc. 16th USENIX NSDI*. Boston, MA.

[29] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. HyperLoop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proc. ACM SIGCOMM*. Budapest, Hungary.

[30] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. 2004. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming* (2004).

[31] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*.

[32] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP)*. Waterloo, Canada.

[33] Memcached 2020. The Volatile Benefit of Persistent Memory. https://memcached.org/blog/persistent-memory/.

[34] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *Proc. ACM SIGCOMM*. Budapest, Hungary.

[35] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. 2019. Storm: a fast transactional dataplane for remote data structures. In *12th ACM International Systems and Storage Conference (SYSTOR)*. ACM, USENIX.

[36] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX Annual Technical Conference*. Philadelphia, PA.

[37] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal.

[38] Oracle TimesTen 2020. Using Intel Optane DC Persistent Memory with Oracle TimesTen In-Memory Database. https://blogs.oracle.com/timesten/using-intel-optane-dc-persistent-memory-with-oracle-timesten-in-memory-database.

[39] Marius Poke and Torsten Hoefler. 2015. DARE: High-performance state machine replication on RDMA networks. In *HPDC*.

[40] Hanfeng Qin and Hai Jin. 2017. Warstack: Improving LLC Replacement for NVM with a Writeback-Aware Reuse Stack. In *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP*.

[41] Redis Labs 2020. Break the Cost and Capacity Barrier with Intel Optane DC Persistent Memory. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/redis-enterprise-brief.pdf.

[42] Andy Rudoff. 2017. Persistent Memory Programming. *USENIX ;login:* (2017).

[43] Shelby Thomas, Geoffrey M. Voelker, and George Porter. 2018. Cachecloud: Towards Speed-of-Light Datacenter Communication. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*.

[44] Tom Talpey RDMA Commit 2020. RDMA Extensions for Enhanced Memory Placement. https://tools.ietf.org/id/draft-talpey-rdma-commit-01.html.

[45] TPC-C 2010. TPC Benchmark C. http://www.tpc.org/tpcc/.

[46] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Towards Low-Cost, Fast, and Scalable Disaggregated Persistent Memory Systems. In *2018 USENIX Annual Technical Conference*.

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proc. 16th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA.

[48] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A. Jiménez. 2013. WADE: Writeback-Aware Dynamic Cache Management for NVM-Based Main Memory System. *ACM Trans. Archit. Code Optim.* (2013).

[49] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *Proc. 13th USENIX OSDI*. Carlsbad, CA.

[50] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[51] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *Proc. USENIX Conference on File and Storage Technologies*. Boston, MA.

[52] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proc. 17th USENIX NSDI*. Santa Clara, CA.

[53] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. *An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.* Technical Report. Santa Clara, CA.

[54] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. In *Proc. VLDB*. Munich, Germany.

[55] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE.

[56] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS âĂŹ15)*.