Lamda: The Last Mile of the Datacenter Network Does matter

Qiang Li†, Qiao Xiang‡, Derui Liu†, Yuxin Wang‡, Haonan Qiu†, Gexiao Tian†, Xiaoliang Wang⋆
Lulu Chen♦, Ridi Wen‡, Jianbo Dong†, Yuanyuan Gong†‡, Yixiao Gao*, Haozhao Song‡
Zhiwu Wu‡, Shaozong Liu†, Zicheng Luo†, Yuchao Shao†, Yaohui Wu‡, Chao Han†, Chen Tian*
Zhongjie Wu†, Zheng Cao†, Jinbo Wu†, Jiwu Shu‡, Jiesheng Wu†

† Alibaba Group  ⋆ Nanjing University  ‡ Xiamen University  ♦ Fudan University

Abstract

In this paper, we conduct systematic measurement studies to show that the high memory bandwidth consumption of modern distributed applications can lead to a significant drop of network throughput and a large increase of tail latency in high-speed RDMA networks. We identify its root cause as the high contention of memory bandwidth between application processes and network processes. This contention leads to frequent packet drops at the NIC of receiving hosts, which triggers the congestion control mechanism of the network and eventually results in network performance degradation.

To tackle this problem, we make a key observation that given the distributed storage service, the vast majority of data it receives from the network will be eventually written to high-speed storage media (e.g., SSD) by CPU. As such, we propose to bypass host memory when processing received data to completely circumvent this performance bottleneck. In particular, we design Lamda, a novel receiver cache processing system that consumes a small amount of CPU cache to process received data from the network at line rate. We implement a prototype of Lamda and evaluate its performance extensively in a Clos-based testbed. Results show that for distributed storage applications, Lamda improves network throughput by 4.7% with zero memory bandwidth consumption on storage nodes, and improves network throughput by up 17% and 45% for large block size and small size under the memory bandwidth pressure, respectively. Lamda can also be applied to latency-sensitive HPC applications, which reduces their communication latency by 35.1%.

1 Introduction

Memory access has become a bottleneck in the development of computer system architecture [17, 36]. Though memory capacity has increased dramatically, the memory bandwidth has not kept with CPU performance scaling [33, 41].

Device vendors and service providers are aware of the memory bandwidth problem and have been introducing new hardware and computation optimization to alleviate memory bandwidth [10, 31, 36]. For example, Direct Cache Access (DCA) and Data Direct I/O technology (DDIO) technologies were introduced to place the I/O data directly in the processor’s cache rather than main memory [12, 16, 23] (see Sec.2.2.1). The development of cache injection technique [4, 12, 13, 21] enables to access the cache directly to reduce memory bandwidth for applications like Key-Value [15, 28], RPC [14, 21, 40], etc. Processing-in-memory is another promising solution to place the processing logic closer to the memory, which reduces data movement during operation and utilize the internal bandwidth of chip [5, 11, 27, 39].

Moreover, the network process also needs to read/write memory. With the increasing deployment of distributed computation and storage systems, they demand for high-speed network to maintain good application-level performance. For example, storage nodes in the Cloud (Storage Area Network) SAN require 5-10 microseconds latency and 40-100 Gbps throughput [24]. Specifically, Remote Direct Memory Access (RDMA) [7] has been widely applied by current computation and storage systems [8, 10, 31, 32, 43, 44] for the high bandwidth and ultra-low latency. Meanwhile, applications maintain a high-level memory bandwidth usage, where the average memory throughput is over 90% in one production storage system.

However, the impact of high memory utilization on network performance is not well addressed in the literature. To reveal the influence of memory bandwidth contention between application and networking processes, we conduct extensive measurement studies on the leading techniques of DDIO, RDMA and RoCEv2 protocol in datacenter networks. We observe that the high memory bandwidth consumption will cause network throughput drop by 30%-50%, and tail-latency deteriorate by 2x (see Sec. 2.2.2). We then perform in-depth empirical analysis and identify the root cause. Specifically, the contention leads to frequent packet drops, and this problem can not be solved by DDIO write in the shared cache space (see Sec. 3). Packet drops trigger the congestion control mechanism of the network and eventually result in network
performance degradation and affects the performance of all applications in the datacenter.

We aim to release the network capacity by eliminating the impact of the high memory bandwidth consumption on networks. The objective is two-fold: 1) add no more memory to the host; and 2) reduce the bandwidth consumption of memory, such that we can scale to the future 200Gbps and 400Gbps networks (see Sec. 4.1). With regard to the storage system, we make a key observation that the vast majority of data it receives from the network will be eventually written to high-speed storage media (e.g., SSD). As such, we propose to bypass host memory when processing received data to completely circumvent this performance bottleneck.

In this paper, we propose Lamda, a novel system that allows remote devices to directly write data to storage media through processor’s cache, and hence prevents unnecessary memory accesses. Lamda introduces three key techniques: 1) To avoid memory contention between application and network processes, it ensures performance isolation by carefully assigning cache to application and network processes (see Sec. 4.3.1). The assigned cache space is proportional to the ration of incoming network data and local data used by the application. 2) To avoid ping-ponging data between main memory and cache, it realizes fast packet forwarding by using proper RDMA operations for small messages and large message respectively, and pipeline operation to speed up packets processing (see Sec. 4.3.2 and Sec. 4.3.3). 3) To avoid the system crash, it adopts the escape mechanism to exploit backup channel as the last resort for the system (see Sec. 4.3.4).

We implement and evaluate Lamda in a Clos-based testbed, both realistic storage and HPC systems. Results show that for distributed storage applications, Lamda improves network throughput by 4.7% with zero memory bandwidth consumption on storage nodes, and improves network throughput by up 17% and 45% for large block size and small size under the memory bandwidth pressure, respectively ((see Sec. 5.3.1)). For latency-sensitive HPC applications, Lamda reduces communication latency by 35.1% (see Sec. 5.3.2).

The contributions of this paper are summarized as follows:

- We conduct systematic measurement studies to understand the impact of memory bandwidth contention on the performance of high-speed networks and identify its root cause as the contention of memory between application and network processes. To the best of our knowledge, this is the first work that identifies this issue.
- To tackle this issue, we design Lamda, a novel host cache processing system that bypasses host memory and consumes a small amount of CPU cache to process received data from the network at line rate.
- We implement and evaluate Lamda through extensive experiments in a cluster with storage and computation applications to demonstrate its efficiency and efficacy.

2 Background and Motivation

2.1 Background

2.1.1 Memory bandwidth limitations

Memory subsystem directly impacts the performance of application systems. However, according to technology trend report, the performance growth of the current memory bandwidth has been degraded by 4x than CPU, and that of memory latency has been degraded by 6x [33] respectively. The memory bandwidth has become a bottleneck in the development of computer system under the Von Neumann architecture [17, 36].

More and more applications such as storage, big data processing, and high-performance computing, are memory bandwidth sensitive rather than memory capacity, which means that the performance of applications is limited by memory bandwidth [18]. Therefore, applications acquire more memory modules to maintain performance. However, due to the limitation of CPU channels, the number of memory modules that a single machine can support is limited. As a result, we have to separate applications to different clusters, which had low utilization (higher cost).

We show the bandwidth of memory from different manufacturers. As shown in Figure 1(a), when memory capacity grows by 50%, the rate of memory Read grows less than 30%. As shown in Figure 1(b), under the 1:1 read/write ratio, the rate of memory bandwidth grows by less than 20% when memory capacity grows by 50%. Therefore, in order to meet the need of memory bandwidth, we have to spend more money and waste part of the memory capacity.

We then evaluate the usage of memory bandwidth for stor-
Figure 4: Different approaches of DMA for transferring data

Figure 4(a) is a standard method to use traditional Direct Memory Access (DMA). This method is inefficient and costly in terms of access latency and memory bandwidth usage. For example, a server has 6.72 ns to process small packets at 100 Gbps, whereas every access to main memory takes 100 ns, 15× more expensive [13].

Figure 4(b) is a faster I/O technology Direct Cache Access (DCA). DCA exploits PCIe Transaction Layer Packet Processing Hint. It prefetches portions of I/O data to the processor’s cache. While this method of implementing DCA can efficiently prefetch the required parts of the I/O data, it is still inefficient in terms of memory bandwidth usage since the entire packet is DMA-ed to main memory [16, 23].

Figure 4(c) is Data Direct I/O technology (DDIO), which is also known as write-allocate-write-update-capable DCA. With DDIO, I/O devices perform DMA directly to/from Last Level Cache (LLC) rather than system memory. For packet processing applications, NICs can send/receive both RX/TX descriptors and the packets themselves via the LLC. Intel DDIO technology enables data delivery operations to be accomplished with a few trips to memory as possible [12].

2.2 Motivation
In this section, we study how to access memory to get the best performance out of the latest hardware.

2.2.1 Data access path
A standard method to transfer data from an I/O device to a processor is Direct Memory Access (DMA). In this mechanism, the processor provides an I/O device with a set of the memory address, aka receive descriptors. After that, I/O devices read/write data directly from/to the main memory without involving the processor. There are three different access methods:

- Figure 4(a) is a standard method to use traditional Direct Memory Access (DMA). This method is inefficient and costly in terms of access latency and memory bandwidth usage. For example, a server has 6.72 ns to process small packets at 100 Gbps, whereas every access to main memory takes 100 ns, 15× more expensive [13].
- Figure 4(b) is a faster I/O technology Direct Cache Access (DCA). DCA exploits PCIe Transaction Layer Packet Processing Hint. It prefetches portions of I/O data to the processor’s cache. While this method of implementing DCA can efficiently prefetch the required parts of the I/O data, it is still inefficient in terms of memory bandwidth usage since the entire packet is DMA-ed to main memory [16, 23].
- Figure 4(c) is Data Direct I/O technology (DDIO), which is also known as write-allocate-write-update-capable DCA. With DDIO, I/O devices perform DMA directly to/from Last Level Cache (LLC) rather than system memory. For packet processing applications, NICs can send/receive both RX/TX descriptors and the packets themselves via the LLC. Intel DDIO technology enables data delivery operations to be accomplished with a few trips to memory as possible [12].

2.2.2 The impact of the memory bandwidth pressure
To meet the application’s demand, network operators maintain a high bandwidth RDMA network running RoCEv2 protocol and deploy different congestion control algorithms under different application requirements even with DDIO enabled. As shown in Figure 5, DCQCN+PFC(CX4) means we used DCQCN of ConnectX-4 RNIC and deployed PFC to prevent packet loss. DCQCN(CX5) means we used DCQCN of ConnectX-5 RNIC and use the packet loss retransmission mechanism which is comes with the RNIC. DCQCN+PFC(CX5) means we used DCQCN of ConnectX-5 RNIC and deployed PFC to prevent packet loss. The client sends data to the server through RDMA write by Perftest tools [2]. Meanwhile, we simulate memory bandwidth pressure on the server through membw application from the RDT benchmark [3], which is a memory bandwidth benchmark for frequently reading and writing memory. Network throughput is severely affected by memory bandwidth pressure problem. For example, the network throughput of DCQCN+PFC(CX4) drops about 25.1% under the memory bandwidth pressure. The network throughput of DCQCN+PFC(CX5) drops about
46.1% under the memory bandwidth pressure. As shown in Figure 6, network latency is also severely affected by memory bandwidth pressure. The results are normalized according to the network latency with no memory bandwidth pressure. As shown in Figure 6(a), 99 percent network latency of DCQCN(CX5) deterioration by 2x, and 999 percent network latency worsened by 4x. As shown in Figure 6(b), 99 percent and 999 percent network latency of DCQCN+PFC(CX5) worsened by about 2x.

Under different network bandwidth and congestion algorithm configurations, the performance of the network will be affected by the memory bandwidth pressure problem. The reason is that network process also needs to consume memory bandwidth, and contention with the application for memory bandwidth. With the development of the network bandwidth, it occupies an increasing percentage of memory bandwidth, which will lead to more serious contention and make more seriously network performance degradation.

3 Understanding details of network performance fluctuations

This section discusses two questions: (1) Why network performance suffers from memory bandwidth pressure even with DDIO enabled? (2) Can we continue to take advantage of LLC under the memory bandwidth pressure?

Testbed. We use a testbed with the configuration shown in Table 1 running Centos 7.2 (Linux kernel-3.10.0) with Haswell. PFC PAUSE frames are disabled to avoid taking into account pause duration in the end-to-end latency. We rewrite the Mellanox RDMA tests to implement send and receive packets and generate memory bandwidth pressure with storage FIO test and perform precise cache management/partitioning (e.g., LLC way partitioning with CAT [3], Cache Allocate Technology presented by Intel). To monitor DDIO and its interaction with I/O devices, we use Intel Performance Counter Monitor (PCM) tool to count the number of PCIe write hits/misses (represented as an ItoM event) and PCIe read hits/misses (represented as a PCIeReadCur event).

3.1 Why network suffers?

This section scrutinizes the performance of DDIO in different scenarios and the goal is to find out why network performance is still affected by the memory bandwidth pressure problem with DDIO enabled. Generally, the I/O device data delivery operations using DDIO technology are achieved with fewer trips to memory, and in the ideal case, without any trips to memory. More specifically, there are two different write methods for DDIO write operation i.e. write allocate and write update. Intel announced that write update causes an in-place update in the cache and write allocate causes allocation in the last-level cache of the CPU’s caching hierarchy, but it did not mention the reason that DDIO is ineffective in the real scenarios which has been addressed in previous work [9, 13, 42]. Based on our experience, we speculate analyzing the difference between the two methods in real scenarios is the key to solving applications not taking advantage of DDIO.

The network performance affected by different write methods of DDIO are various. We assumed that write allocate requires more serial steps to trip the memory using an experiment in which we co-run an I/O intensive application and a cache-sensitive application on the same machine. We used an RDMA-based application as the I/O intensive application. Specifically, the server receives consecutive packets of 64KB at a 50 Gbps rate with four QPs using different memory buffer sizes (64KB-16MB). And for the cache and memory bandwidth sensitive application, we used membw. At the same time, we isolated the cache between the two applications with CAT, which limits different applications to a subset of LLC ways to perform precise cache management/partitioning. The DDIO is allocated to 0xC in 0xFFFFF LLC set-associativity (e.g., two of the twenty ways in Haswell processors). Each application was run on a different CPU core through taskset command to
avoid the impact of CPU process switching. ①Exp1 is configured with cache management/partitioning and DDIO enabled. ②Exp2 is configured without cache management/partitioning and DDIO enabled. ③Exp3 is configured without cache management/partitioning and DDIO disabled.

As shown in Figure 7(a), with the increase of memory occupied by data, the network throughput gradually decreases, and eventually stabilizes when the memory buffer size is larger than 2MB. Compared with Exp1, the average network throughput of Exp2 is decreased by about 10% when the memory buffer size is 512KB. However, when the memory buffer size is larger than 2MB (the default cache size used for DDIO), the network throughput of Exp1 and Exp2 are almost equal. As shown in Figure 7(b), When the buffer size of memory is less than 2MB, the cache miss rate of Exp1 gradually increases. The cache miss rate of Exp2 also gradually increases but always greater than that of Exp1. When the buffer size of memory is larger than 2MB, the cache miss rate of them are both about 100%. Exp3 has the worst network performance among them. The network throughput of Exp3 is decreased by about 29% and 17% compared with Exp1 and Exp2 respectively.

We can see that isolating cache between the applications through CAT can alleviate memory bandwidth pressure problem. When the buffer size of memory is less than 512KB, the whole memory required for the data can be mapped into the cache. Under such circumstances, when a network packet is received, the NIC will only access the LLC but not the memory since the arrival of data will trigger the write update of DDIO. When the memory buffer size is larger than 512KB, the whole memory required for the data cannot be mapped into the LLC, which means the cache line miss, thus trigger the write allocate of DDIO. This process will first invalidate part of the cache in LLC through the LRU algorithm, and it needs to flush part of the data from LLC to memory, contending with the application for memory bandwidth. The memory bandwidth pressure problem is severe with DDIO disabled indeed. Moreover, we have found that memory bandwidth pressure will affect network performance during DDIO write allocate.

3.2 Can we continue to take advantage of LLC?

This section scrutinizes cache contention in different isolation configurations among the DDIO, the network process and the application. The goal is to find out if we can continue to take advantage of LLC even if under memory bandwidth pressure.

Why we need to isolate the cache for application, DDIO and network? When an application is limited to using cache ways that are also used by DDIO, cache lines allocated in LLC for DDIO may evict the code/data of the application itself. This issue was discussed by Tootoonchian and Alireza Farshin et al. [13, 42]. They prove that overlapping any application with DDIO ways in LLC can reduce the performance of both applications. Specifically, the value for Model Specific Register (MSR) with the address of “0xC8B” indicate the LLC ways occupied by DDIO in the system. We need to set up LLCs for the application that does not overlap with it through CAT.

Meanwhile, the network and application share the cache also cause cache contention problem. Specifically, for run-to-completion thread model, each thread needs to process both the network I/O logic and the application’s main work logic. The coupling of code segments and data segments between the network and the application under the run-to-completion model makes it impossible to isolate cache between them. Therefore, the data between the network and the application will contend cache with each other, resulting in unnecessary memory bandwidth because of ping-pong data between the memory and cache.

Is isolating cache good enough? Previous works illustrate that applications that share the LLC without overlapping with I/O can get better application performance than with overlapping [6, 13, 42]. However, the network performance is also impacted by the memory bandwidth pressure problem after cache isolating. We further isolate network and application to analyze the impact on the memory bandwidth pressure.

We build a network-application isolated ping pong benchmark. We simulate network and the application through different processes and deploy them to the same physical node with
a shared receive memory pool. They are bound to different CPU cores and different LLC ways without overlapping. The client sends data to the network process and then network process notifies the application to process data under the different message sizes. ①Exp1: Application reads the data through the shared memory pool directly and then forwards it to the client. ②Exp2: Application sends the data through its own memory directly ③Exp3: Application copies the data from the shared memory pool to the fixed memory and then send it.

We measure the memory bandwidth of the physical node where the network and application are located. As shown in Figure 8, Exp1 consumes almost zero memory bandwidth but Exp2 consumes memory bandwidth in memory read direction. Although network and application are assigned to different LLC ways, this does not characteristics of LLC shared by all cores. Therefore, if the memory address is mapped into the cache allocated by network, the application can still cache hit the same memory address. In Exp1, the arrival of network data is written to the LLC through DDIO. When the application send data, NIC will read the data from the LLC directly and does not consume memory bandwidth. In Exp2, the application send data through its own memory. Since the memory address is not mapped into the cache, consuming memory bandwidth in read direction. When the NIC DMA data in read direction, if the memory address of the data is not in the cache, the memory will be accessed always. This is in contrast to the fact that DDIO forces the memory address to be mapped into the cache when the NIC receives data. Exp3 also consumes almost zero memory bandwidth. The reason is that Exp3 maps the memory address of the data into the cache through once memory copy. Since the destination memory address of the copy is always the same, an cache-to-cache copy also consumes zero memory bandwidth.

Through the above experiments, we know that relying on cache isolation between DDIO, network and application can still consume memory bandwidth. When sending data, the NIC will first send control information to the cache to determine whether the data is in the cache. If cache hits, the NIC will read data from cache directly. Otherwise, then the NIC will read data from the memory, consuming memory bandwidth in read direction. In the case where data needs to be sent multiple times such as data replication, loading the data into the cache first can greatly reduce the required memory bandwidth. Meanwhile, reasonable copying data between cache can avoid the impact under the memory bandwidth pressure.

4 Lamda

4.1 Design rationale

In this section, we describe the design goals and architecture of Lamda. Two goals dominate the Lamda:

- Minimizing performance impact on applications: The fluctuations of network performance including both throughput and latency could seriously hurt the predictability of application performance. As the network bottleneck shifting from network core to end-host in hundreds of Gbps network, the design and running of micro routing PCIe network in end-host are also significant for applications’ end-to-end performance.

- Minimizing the involvement of memory when processing received data: We make a key observation that in modern distributed applications such as distributed storage systems, HPC computing and big data analytics, memory is only an intermediate stop for the massive amounts of data received from the network. The eventual destination of these data is either high-speed storage media (e.g., SSD), or computation unit (e.g., GPU). As such, the network performance can gain substantially if we can minimize the involvement of memory on processing the received data.

4.2 Key ideas

We start by explaining the key ideas that make Lamda possible. The core insight of Lamda is that LLC of modern CPUs is usually large (tens of MB) and high performance (hundreds of GBps) and the most important point is that applications don’t take good advantage of it. Lamda could minimize memory bandwidth consumption and stabilize application performance by shifting the middle station from memory to cache for data forwarding, with good control of cache space usage. In detail, Lamda designs three key mechanisms:

- Cache isolation between network and applications: To ensure network performance and mitigate cache contention among applications and I/O devices, Lamda allocates dedicated cache group for IO devices and applications. During data forwarding, I/O devices would read or write data from or to dedicated cache group by DDIO. Then applications running over computing devices could also read or write data on same cache group but could not change the memory-to-cache mapping. When applications access their other memory area, cache swapping would be executed on their dedicated cache group, without generating unexpected cache swapping on cache group for IO devices and unnecessary memory bandwidth consumption. This mechanism is inspired by Alireza [13].

- Network buffering with small-size cache resident occupy To avoid memory bandwidth consumption from frequent cache swapping and worse memory bandwidth contention with computing devices when I/O devices forward data. Lamda first limits the address range and the whole size of memory area that I/O devices could access by DDIO, enforcing that the dedicated cache group
could always accommodate memory area accessed by I/O devices and cache swapping would almost never happen. Then Lamda loads the memory area to dedicated cache group periodically and make it always cache resident. Finally, Lamda limits the total size of in-flight inbound data at receiver-side and adapt to the total size of memory area that I/O devices could access. As a result, I/O devices would always access and hit the dedicated cache group for each I/O and execute cache write update, without experiencing heavy cache swapping, cache allocation and memory bandwidth contention under cache miss.

- **Cache recycle with fast forwarding during transient period:** To effectively limit the whole occupied memory area that IO devices could access, Lamda requires that the cache area be recycled quickly, i.e., applications should process the received data from network and release related cache area back to Lamda as much as possible. To achieve fast data forwarding, Lamda uses separated threads to handle network IO and builds efficient cache management for all kinds of data size.

### 4.3 Architecture

We take the distributed storage system as an example to describe Lamda in detail and describe the performance obstacles and our corresponding solutions. The overall architecture of Lamda is shown in the Figure 9. There are three main modules in Lamda: 1. shared circular buffer, which is size-limited and shared between network thread and work thread of application. 2. network thread, which handles network I/O and manage memory area that I/O devices could access by DDIO. 3. work thread, which executes applications’ tasks and interacts with I/O devices to accomplish data forwarding. We explain them separately through cache isolation, cache management, cache pipeline processing and fault recovery.

#### 4.3.1 Cache isolation between network and application

Lamda firstly splits CPU LLC into two cache groups by CAT tool, assigns them to network thread (which manages memory area that IO devices could access) and work thread (which executes applications’ tasks). To determine the cache size for I/O devices, a simple I/O model could be used for analysis. In the simple I/O model, the processing capability among computing devices, network devices and storage devices are matched. A cache recycle is defined as the total time $T$ cost by data size $S$ passing from network devices to computing devices / storage devices and return to network devices. In this assumption, the cache size $C$ for I/O devices could be computed as $C = T \cdot S$. In our storage application, $T$ is about 3 5 us for typical 4KB data under peak network throughput and $T$ is no more than 1 ms, finally the maximum value of $C$ is about 4MB. For 100Gbps network, 4MB cache area could sustain about 320 us without cache overflow which is enough to accomplish cache recycle. In our environment, we use 4 LLC cache groups (0xF0000) of 20 cache groups (0xFFFFF), which means 4MB of the total 20MB is used for I/O devices to execute DDIO alone. The rest 16 cache group are reserved for work thread to execute application tasks.

#### 4.3.2 Cache reuse between network and application

After isolating cache group, I/O devices acquire the dedicated cache group for DDIO and they expose the cache group as a shared circular buffer for applications to do data forwarding. Lamda need to effectively management these cache groups. The work thread of application needs to be responsible for memory registration and all control operations are handled by the network thread. The network thread is responsible for inserting/deleting/updating corresponding to the data address in the shared circular queue. We design different optimizations for different RDMA transfer semantics.

**Shared Receive Queue (SRQ) based small message optimization.** For small messages less than 8KB, we use a two-side RDMA Send/Receive operation through a shared receive buffer. The work thread of application applies for memory through a shared memory pool with the network thread and registers it. Then the network thread submits the address information to the SRQ. Each time a new network message is received, a receive request in the shared receive queue will be consumed. The network thread notifies the work thread after receiving the completion event notification from the NIC. When the work thread processes the message, the memory corresponding to the data is back to the shared buffer. Then the network thread adds receive request corresponding to the memory address to the shared receive queue again. In the storage node of our distributed storage system, when the network thread receives the message, a single work thread will process the message and place it to the disk directly. Under the network bandwidth of 100 Gbps, it takes 320 us for the 4MB shared receive buffer to be fully filled. Therefore, the shared
Receiver-side control based large message optimization. For large messages larger than 8KB, we use a one-side RDMA Read operation. Since the memory address of the data is changing every time, the incoming data cannot be written to the memory address corresponding to the work request in the receive queue. The receiver initiates a data read request, and the request carries the memory address of the sender’s data memory address. We use receiver-side control to ensure the total amount of in-flight data remains within a certain range so that the corresponding memory address can be completely mapped into the cache. The newly arrived message can trigger the write update of DDIO instead of writing allocation, so it can avoid the data being flushed back to the memory under the memory bandwidth pressure influence. The work thread of the application slice the data to avoid the influence of large blocks of data. When the data block size is larger than 128KB, it will divide the large data into multiple requests according to the granularity of 128KB.

4.3.3 Fast forwarding and pipeline processing

In the distributed storage system, when a network message arrives, a series of data operations including CRC calculation, de/serialization, and data format conversion need to be performed. Then the data will be further placed to the disk or other storage nodes. The data flow process can be abstracted into the producer-consumer model of the network and application, as shown in Figure 10. The producer module in the network thread pushes data to the shared circular buffer, and the consumer module of the work thread process the data and forward it.

By slicing the large message according to the granularity of 128KB, the overhead of software processing and the arrival of network messages are in pipeline. Ideally, as shown in Figure 9, the message enters the shared buffer through ① and then directly writes to the disk through ④. Producers and consumers are not one-to-one. Data put per producer can be processed by any consumer. Due to the performance limitation of a single CPU, multiple work threads are required to ensure the parallel of the pipeline. In detail, according to the software processing overhead of the application, the incoming bandwidth and outgoing bandwidth of the message in the cache could be matched by adjusting the number of work threads, so that the cache can be reused as much as possible to further avoid the consumption of memory bandwidth. For example, for 100Gbps network bandwidth and 4MB buffer size, we take a fixed size of 8KB to split the data block. If the production interval of each 8KB data block at full rate is 0.64 us, if the processing time of each IO thread is 6.4 us, 10 IO threads are required to ensure the complete parallelism of the pipeline. Therefore, no unnecessary memory bandwidth is consumed through fast forwarding and pipeline processing.

4.3.4 Escape mechanism

There is a cumulative effect in the process of fast-forwarding. In this scene, if the reclamation speed of memory address decreases, it will lead to disrupt the normal execution of the application-network pipeline, causing unpredictable performance fluctuations. The reason is that network packets may arrive out of sequence, and the storage application is processed strictly in order, which makes the memory address corresponding to the out-of-order data packet may not be recycled in time. And the sender cannot send new data if the receiver does not have available memory addresses.

Lamda use a FIFO queue to keep the total amount of concurrent in-flight data within a certain range size. Specifically, each network thread has an initial minimum request window. And multiple network threads dynamically limit their request window according to the maximum request window, which means that the total amount of window size can not exceed a threshold, a.k.a the size of the cache allocated to execute DDIO by CAT. Meanwhile, Lambda achieve a timeout escape mechanism to avoid deadlock. Specifically, if the receiver does not have a memory address available for a certain time interval (milliseconds-level), it will apply a new memory address from the spare memory pool to receive new network data. This will lead to a burst of memory bandwidth read/write, resulting in a slight memory bandwidth consumption. Meanwhile, the escape mechanism can avoid network throughput jitter and achieve high network utilization.

4.4 Performance optimization

The workflow of the distributed storage system is as shown in Figure 11. Compute nodes generate real data and each segment is assigned to a BlockServer for I/O processing. Each ChunkServer initiates the RPC servers, and storage operations are performed by issuing pre-registered RPCs from BlockServer to ChunkServer. After the data reaches the ChunkServer, it is processed and placed to the disk. Moreover, a mix of BlockServers and ChunkServers will be deployed on each physical machine.
4.4.1 Performance hindrance

Due to part of the processing overhead, and the performance of a single CPU is insufficient, we need more CPUs to fully utilize the pipeline performance of the network application. We found that the overhead barriers of the storage system suffered from interaction between the network layer and storage layer. The newly arrived network message cannot be directly used by the storage protocol, so the CPU needs a series of computational processing. Not only does it induce more latency, but also leads to the computational bottleneck of the CPU.

As shown in Figure 12, for an RPC initiated from the storage domain to the network domain, the discrete objects in the storage layer would be firstly serialized into a continuous byte stream for inter-node network communication. The byte stream had to be deserialized into structured data at the Chunkserver side. The mismatch between storage protocol and network protocol format required redundant de/serialization operated through Protobuf. The overhead brought by RPC is non-negligible.

Meanwhile, data movement between network and storage also cause data translation overhead. Typical storage write operation is shown in Figure 13. The data from Blockserver was transferred to a continuous network memory buffer through RDMA communication, and then it was sliced and copied to the SSD memory buffer according to the storage’s data format, and then placed to the SSD. In detail, multiple process tasks such as CRC, data format conversion, serialization, and deserialization are required, which is the root cause of storage system exhausting memory bandwidth.

4.4.2 Optimization methodology

Lamda achieves fast path through removing RPC at the Read/Write data operation and realizes inline processing of network message through full-link offloading by RNIC. Lamda eliminated the protocol translation overhead by targeting the Read/Write RPC operations. Specifically, we designed a customized data structure while maintaining the flexibility and performance to replace the protobuf during Read/Write RPC. At the same time, we customized a fast path for storage Read/Write operation by offloading storage protocol to the network, which took the advantage of the lightweight data header of Read/Write data structure.

Meanwhile, Lambda enables network message to be written directly to the SSD through the cache, while consuming zero memory bandwidth. The data translation overhead brings great pressure on memory bandwidth when processing data. The RDMA UMR (User Mode Registration) and CRC offloading were used to inline data format construction for the storage layer. It makes the overhead caused by memory copy and the CRC calculation offload to the communication phase, which also reduces CPU utilization and memory bandwidth consumption.

5 Evaluation

We evaluate the performance of Lamda to validates our design principles.

5.1 Testbed Setup

Network Topology: The testbed is a Clos network, which mimics a small-scale PoD in the production datacenter. The testbed consists of two ToR switches and one spine switch. Each server is configured with two Mellanox NIC and connects to two ToR switches for high availability.

Workload: For micro-benchmark, we use dual Mellanox ConnectX-5 card (50Gbps). RDMA traffic is generated by using the Mellanox PerfTest tool, for performance evaluation in RDMA. Then we use standard network benchmarks to further measure the memory bandwidth consumption of Lamda on the host. For application-benchmark, we use dual Mellanox ConnectX-6 cards (200Gbps) to support real high-performance application deployment environment. We use FIO benchmark tools to generate traffic for distributed storage system. And use MVAPICH benchmark [35] tools to verify Lamda performance for high-performance computing.
At the same time, we deploy the DCQCN that comes with Mellanox NIC and cancel the PFC to avoid the impact of PFC to the network latency.

5.2 Micro-benchmark

We seek to understand: (1) can Lamda match the throughput for fast forwarding? (2) how is the network performance of Lamda under the memory bandwidth pressure? (3) can Lamda adapt to reduce memory bandwidth consumption?

**Effectiveness of fast forwarding:** In order to evaluate Lamda performance for fast forwarding, we measure the optimization gain in terms of fast forwarding throughput and end-to-end latency per work thread. As shown in Figure 14, Lamda improves by up to 3.8x fast forwarding throughput per work thread of the application at most. For 4KB block size, Lamda increases 1.1x network throughput supported for fast forwarding while maintains the almost same application latency. For 128KB block size, Lamda increase 3.7x network throughput and reduces 7% application latency. By network storage co-design for critical paths and offloading storage protocols to the network, Lamda avoids the overhead of processing and the overhead of frequently writing/reading data to memory. Therefore, Lamda can greatly increase the forwarding throughput that per work thread can support, allowing it to match network throughput with fewer work threads.

**Network performance:** To evaluate the network performance of Lamda under the memory bandwidth pressure, we measure the network throughput and the PCIe average latency of the receiver. We simulate memory bandwidth pressure on the receiver using `membw` application, and then keep on sequentially sending messages of fixed size at the different memory buffer size.

As shown in Figure 15(a), with Lamda optimization, the loss of network throughput is little. The improvement of throughput is notable from 512KB memory buffer size, and it up by 32% under the 4MB. It protects the application with large memory buffer size requirement from performance degradation. The PCIe latency from NIC to the host represents the average latency of the NIC writing data. The PCIe latency of each write operation fluctuates heavily, so we gather the each write latency during processing and calculate the average latency as the evaluation metric. As shown in Figure 15(b), the average PCIe latency decreases with Lamda optimization, and its reduction is up to 60% under the 4MB memory buffer size. This improvement thanks to the receive side control that reduces the cache miss rate when the memory size requested by the application is larger than the DDIO allocated.

We then evaluate the cache miss rate at the receiver, which provides more insight into the achieved performance gain as motioned above. With the increasing of memory buffer size, the DDIO method gradually shifts from DDIO write update to write allocation. And the cache miss rate of the NIC also gradually increases as shown in Figure 16. At the memory buffer size from 4MB, the cache miss rate before the optimization is close to 100%. At this time, under the memory bandwidth pressure, the cache is ineffective for acceleration of memory access even though DDIO is enabled. The receive side control of Lamda makes the miss rate reduced to about 2%, which improves the throughput and shorts the average latency of PCIe prominently.

**Memory bandwidth consumption:** The improvement of application performance comes from the efficiency of memory bandwidth consumption. In other word, use less memory bandwidth to write more data. Therefore, we measure the memory bandwidth consumption of writing the same size data to verify the effect of cache reusing and optimization of RDMA two-side operation. We run the storage network benchmark and kill other processes such as memory recycling to avoid influence. Figure 17 shows that Lamda can greatly reduce the consumption of memory bandwidth to background
To evaluate the performance of Lamda under realistic application, we test Lamda under distributed storage application and high-performance computing application. For storage application, the deployed ratio of computing clients nodes and storage nodes is 1:3 in a mixed deployment, which means each physical node is both a client and a storage server. Client nodes send I/O requests with different block size, and storage nodes backup data and respond to the request of client. For high-performance application, each physical node deploys 8 processes, and a total of 32 processes for MPI communication.

5.3 Storage Application

Network throughput and memory bandwidth optimization: To illustrate the effectiveness of Lamda in the storage system, we use FIO to generate large block data (128KB) and access the storage application server. We evaluate the network throughput and the memory bandwidth during deployment. As shown in Figure 18(a), Lamda improves by up to 8.4% network throughput. And as shown in Figure 18(b), Lamda decrease nearly 83% memory bandwidth consumption in average. At the same time, Lamda can reduce network throughput and memory bandwidth jitter. The reason is that NIC could directly write data to the dedicated cache and data is forwarded to the SSD directly with Lamda, which can avoid extra memory access. Meanwhile, Lamda can avoid the calculation of CRC by the CPU, thereby avoiding once memory access. Therefore, Lamda can avoid twice memory access in total, and the data is directly written to the SSD through the cache, achieving nearly zero memory bandwidth consumption.

Application performance optimization: To evaluate the network performance, we select one of the storage application nodes under the memory bandwidth pressure. We evaluate the average network throughput and the average 99.9th end-to-end latency from FIO client to the storage node while varying FIO’s block size. As shown in Figure 19(a), Lamda increase 17% FIO’s average great network throughput under the large block size (128KB) using Receive-side control while increase 45% average great throughput under the small block size (8KB) using SRQ. Under same bandwidth pressure, small block are more sensitive to IOPS than large block and achieved more significant improvement in throughput when consuming less memory bandwidth under Lamda. As shown in Figure 19(b), for average latency of FIO, Lamda reduce up to 18.1% and 47% for large block size and small block size, respectively. As shown in Figure 19(c), for 99.9th tail latency, Lamda could reduce up to 63% and 92% for large block size and small block size, respectively. The reason is that Lamda ensure that data received from the network is always and completely written to the dedicated cache group, and then forwarded to the disk directly, avoiding extra latency induced by the bandwidth contention under heavy memory bandwidth pressure.

Escape mechanism optimization: Slow recycle of memory area could quickly consume all receive memory area and lead to no available area to hold inbound data, resulting in back pressure to sender and reducing remote send rate. Frequent slow recycle of memory area would seriously hurt performance of application and should be avoided. Through Lamda’s escape mechanism, FIO could always achieve stable and peak throughput about 200Gbps. It is worth noting that escape mechanism of Lamda hardly consumes additional memory bandwidth consumption (about 1GBps) in comparison with throughput achieved under normal case. In summary, the escape mechanism of Lamda is necessary and useful for exception case.

5.3.2 Compute Application

The field of High-Performance Computing (HPC) has been seeing steady growth during the last 25 years. Most of the parallel applications such as weather forecasting, earthquake simulations, and chemical simulations, continue to use Message Passing Interface (MPI) libraries conforming to the MPI Standard. Thus, ensuring the performance and quality of the MPI library is important to the progress of the HPC field. We also deployed Lamda with part of the MPI library, and we verify its effect by running MVAPICH benchmark. As shown in Figure 20, Lamda can reduce the communication time of the MPI library under the pressure of the memory bandwidth bottlenecks. For all-to-all and all-gather communication, Lamda can reduce latency by up to 35.1% and 25% than before respectively. For all-reduce communication, Lamda can reduce latency by up to 5.5% than before. Lamda can achieve better performance optimization for communication-intensive MPI library. The reason is that communication-intensive MPI library is more affected by network performance than computation-intensive MPI library.
6 Related work

Novel Network Architecture: Many works use FPGA for network programmability to minimize the number of data moves on the host. Fukuda [15] and Lim [28] use FPGA to implement a data cache, they use the memory in the programmable NIC as an extension of the host. StRoM [40] and FlexNIC [19] offload the application layer based on the programmable NIC. Fine-grained memory access operations can be achieved through the NIC. The NIC can aggregate or filter the passing data. NetDIMM [4] and Dagger [26] connect programmable NIC and CPU via the memory bus, which means the arrival of network packets is no longer transmitted through the traditional PCIe bus, but through the memory bus. It changes the way data flows to the host and completely offloads the RPC software protocol stack. PMNet [38] uses FPGA to implement a programmable data plane, which further extends the original data persistence operation to the NIC. After the NIC receives the data, it is directly written to the persistent memory.

Network System Optimization: Meanwhile, many works analyze the unknown details of the system to make us design more reasonable applications. Larsen [25] measures the overhead of descriptors in the process of DMA data of the NIC. It abandons the complex IO control that the hardware relied on the memory system. Farshin [13] studies the details of the direct cache access technology in Intel, which remind us that DDIO optimization can reduce the latency of I/O intensive network functions. It focuses on the cache contention between DDIO and application, but does not consider the impact of memory bandwidth pressure and does not study how to use the cache efficiently with application. Rolf [34] shows that PCIe, alongside its interaction with the root complex and device drivers, can significantly impact the performance of end-host networking. They propose a methodology to analyze the cost of PCIe in detail.

System & Network Co-Design: Recently, there have been increasing amount of work that adopts system and network co-design through SmartNIC, including RPC systems [30,37], storage system [21,22,29] and key-value stores [15,19,20]. When the application runs across machines, the endpoint of the data path is the NIC. They greatly reduce the number of data moves from the NIC to the host by separating part of tasks on the host.

7 Conclusion

This paper shows that memory bandwidth pressure on the host can significantly impact the performance of end-host networking. Past research has reported some of the findings in the context of specific applications such as KVS acceleration and DCA technology. In contrast, we systematically showed that the memory bandwidth cannot perform as needed with increasing link speeds. We present, to our knowledge, the first detailed methodology to understand the impact of memory bandwidth pressure on network traffic systematically. We also present Lambda, a cache-reused high-performance implementation of the methodology for a systematic evaluation in real systems. We showcase the ability of Lambda to maintain a high level of performance even in the high memory bandwidth pressure, whose network performance is hardly affected, and achieve a great application performance.
References

[1] Nas parallel benchmarks. Website. https://www.nas.nasa.gov/software/npb.html.

[2] Mellanox perfest package, 2017.

[3] Intel® resource director technology (intel® rdt), 2021. https://www.intel.cn/content/www/cn/zh/architecture-and-technology/resource-director-technology.html.

[4] M. Alian and N. S. Kim. Netdimm: Low-latency near-memory network interface architecture. In Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2019.

[5] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O’Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim. Application-transparent near-memory processing architecture with memory channel network. In Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).

[6] M. Alian, Y. Yuan, J. Zhang, R. Wang, M. Jung, and N. S. Kim. Data direct I/O characterization for future I/O system exploration. In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2020.

[7] I. T. Association. Supplement to infiniband architecture specification volume 1 release 1.2.2 annex a17: RoCEv2 (IP routable RoCE), 2014.

[8] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In ACM SIGMOD International Conference on Management of Data, SIGMOD, 2015.

[9] H. Basavaraj. A case for effective utilization of direct cache access for big data workloads. Master’s Thesis, 2017.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[11] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In IEEE International Symposium on High Performance Computer Architecture (HPCA), 2015.

[12] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In USENIX Annual Technical Conference (ATC). USENIX Association, July 2020.

[13] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In 2020 USENIX Annual Technical Conference (ATC). USENIX Association, 2020.

[14] D. Firestone, A. Putnam, S. Mundkur, D. Chiu, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zhuhair, D. Bansal, D. Burger, K. Vaid, D. A. Malitz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA, 2018. USENIX Association.

[15] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura. Caching memcached at reconfigurable network interface. In 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014.

[16] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In International Symposium on Computer Architecture (ISCA), 2005.

[17] M. Jung, S. A. McKee, C. Sudarshan, C. Dropmann, C. Weis, and N. Wehn. Driving into the memory wall: The role of memory for advanced driver assistance systems and autonomous driving. In Proceedings of the International Symposium on Memory Systems, MEMSYS, page 377–386, New York, NY, USA, 2018. Association for Computing Machinery.

[18] A. Kalia, D. Andersen, and M. Kaminsky. Challenges and solutions for fast remote persistent memory access. In Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC, New York, NY, USA, 2020. Association for Computing Machinery.

[19] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with flexnic. In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, New York, NY, USA, 2016. Association for Computing Machinery.

[20] H. Kim, S. Rixner, and V. S. Pai. Network interface data caching. IEEE Transactions on Computers, 54(11):1394–1408, nov 2005.

[21] H.-y. Kim, V. S. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. SIGPLAN Not., 37(10):239–250, oct 2002.

[22] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles, SOSP. Association for Computing Machinery, 2021.

[23] A. Kumar and R. Huggahalli. Impact of cache coherence protocols on the processing of network traffic. In Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2007.

[24] G. Kumar, N. Dukkipati, K. Jung, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In Proceedings of the ACM SIGCOMM Conference, SIGCOMM. Association for Computing Machinery, 2020.

[25] S. Larsen, S. Larsen, and B. Lee. Platform io dma transaction acceleration. 2012.

[26] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. ASPLOS. Association for Computing Machinery, 2021.
[27] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. Scope: A stochastic computing engine for dram-based in-situ accelerator. In Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018.

[28] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. SIGARCH Comput. Archit. News, 41(3):36–47, jun 2013.

[29] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, Nov. 2020.

[30] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. SIGCOMM, New York, NY, USA, 2019. Association for Computing Machinery.

[31] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of apache spark with rdma and its benefits on various workloads. In IEEE International Conference on Big Data (Big Data), 2016.

[32] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: An rdma-enabled distributed persistent memory file system. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC ’17, page 773–785, USA, 2017. USENIX Association.

[33] J. McCalpin. Memory bandwidth and system balance in hpc systems. https://blogs.fau.de/hager/archives/9072.

[34] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Bueno, and A. W. Moore. Understanding pcie performance for end host networking. In Proceedings of the ACM SIGCOMM Conference, SIGCOMM, 2018.

[35] N.-B. C. L. N. of The Ohio State University. Mvapich: Mpi over infiniband, omni-path, ethernet/iwarp, and roce. Website. https://mvapich.cse.ohio-state.edu/benchmarks/.

[36] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC, 2021.

[37] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In Proceedings of the ACM Symposium on Operating Systems Principles, SOSP. Association for Computing Machinery, 2021.

[38] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan. Pmnet: In-network data persistence. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021.

[39] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In ACM/IEEE Annual International Symposium on Computer Architecture (ISCA), 2016.

[40] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. Strom: Smart remote memory. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery.

[41] L. Song, X. Qian, H. Li, and Y. Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[42] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling slos in network function virtualization. In Proceedings of USENIX Conference on Networked Systems Design and Implementation, NSDI, USA, 2018. USENIX Association.

[43] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: A resilient rdma-driven key-value middleware for in-memory cluster computing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC. Association for Computing Machinery, 2015.

[44] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: scaling graph computation to the trillions. Proceedings of the Sixth ACM Symposium on Cloud Computing, 2015.