Lamda: The Last Mile of the Datacenter Network Does Matter

Paper # XXX, 12 pages excluding reference

Abstract

Datacenters host a variety of applications such as distributed storage, high performance computing and big data processing etc. However, the CPU overhead of different services sharing the machine with the applications using them is increasingly a burden. And with the rapid development of the network bandwidth, the pressure on memory and PCIe bandwidth caused by network I/O data flow has gradually increased. Meanwhile, more and more intense contention also occurs in different data flows, causing skewed QoS.

How to reduce the bottleneck on host caused by growing network and maximize the distributed application performance become a high-profile issue. We argue the essence of the problems is the bottleneck caused by frequent data flow under loosely coupled architecture. In this paper, We first introduce and summarize the host bottlenecks caused by the growing network systematically. And present the design of Lamda, a hardware software co-design high-performance IO framework with support for distributed storage system. Then introduce the effective solution based on Lamda which will mitigate the affection of the QoS issues. We shown that Lamda can alleviate host memory bandwidth by ???% and PCIe bandwidth by ??%, while achieving ??% better performance.

1 Introduction

Datacenters host a variety of applications such as distributed storage [5, 29], high performance computing [7, 35, 37], big data processing [28] etc. These applications desire high bandwidth and ultra low latency. For example, storage nodes in the current Cloud SAN (Storage Area Network) storage system require 5-10 microseconds network latency and 40-100 Gbps throughput to maintain good application-level performance [14, 20]. Similarly, in heterogeneous computing environments, different computing chips, e.g. CPU, FPGA, and GPU, also need high-speed interconnections [8, 38].

However, the CPU overhead of different services sharing the machine with the applications using them is increasingly a burden to application performance in the datacenter [12, 18]. Meanwhile, with the rapid development of the network bandwidth, which is developing towards 200Gbps and 400Gbps, the pressure on memory and PCIe bandwidth caused by network I/O data flow has gradually increased. Finally, more and more intense contention also occurs in different data flows between multiple applications, causing skewed Quality of Service (QoS). As the network and end-side distributed application complement each other in the whole system, the bottleneck on the host will further back pressure to the network, which will appreciably degrade the performance of the running applications. Due to the stagnation of CPU, memory and PCIe bandwidth performance [31], operators also wish to dedicate as many CPU cycles and memory and PCIe bandwidth to applications as possible, rather than network.

The question we ask in this paper is how to reduce the bottleneck on host caused by growing network and maximize distributed application performance?

There have been some recent research efforts that focus on some of the above bottlenecks. On the one hand, a large body of works has sought to place the processing logic closer to the memory [3, 4, 11, 23], which is a DRAM or DIMM moduled-based near-data processing architecture which can avoid multiple memory accesses during data computing. It accelerate the machine learning application by placing processing elements dedicated to each rank in the buffer chip of DRAM, thereby utilizing the internal bandwidth which is equal to the channel bandwidth multiplied by the number of ranks in the channel. On the other hand, there have been some recent research efforts that offload part of the functions in the whole system onto SmartNICs [19, 25, 26, 33, 34]. They take traditional domain-specific acceleration methods to incorporate lots of application logic onto SmartNICs, which reduce load on the general purpose CPU, reduce latency and increase throughput.

These approaches are suitable for specific class applications, which have sufficient parallelism, deterministic program logic and regular data structures. However, none of the existing solutions consider the real role of the NIC to solve

the bottleneck of von-Neumann and the need to further evolve the NIC for distributed applications.

Our observation for the essence of the problems is the bottleneck caused by frequent data flow under loosely coupled architecture. We argue that the tightly coupled design with the network and part of the application at the NIC will provide significant benefits to applications running on it. These benefits include application specific control of the network flows, the ability to run code at precisely the right location and controllable network delay access, e.g.

In this paper, we present Lamda, a high-performance IO framework with a hardware and software co-design, which can separate the control path and data path of distributed applications. In order to better verify its effect, we combine it with distributed storage system, that is an important building block for modern online and offline services that uses the network to connect thousands of storage servers to solve ultra-large-scale storage problems which cannot be completed by a single storage server. Lamda achieves group-based QoS isolation by providing abstract primitives to the storage application. Lamda offloads processing-intensive storage tasks such as replication, data publication and hot data store. Leveraging dedicated hardware acceleration modules, Lamda reduces network utilization by conducting file- and operation-specific (de)compression, improves security by performing data-specific encrypt/decrypt. Moreover, Lamda reduces system fail-over time and number of connections by providing virtual connection for application. Finally, Lamda achieves communication efficiency by asynchronously aggregating work at all inputs and outputs, which can be used to significantly optimize communications over the wire and PCIe, alleviating the host bottleneck caused by the growth of the network bandwidth.

The contributions of this paper are:

- We first introduce and summarize the host bottlenecks caused by the growing network systematically, which make us realize that the network problems no longer occur in the intermediate link, and will gradually migrate to the end side.
- We present the design of Lamda, a hardware software co-design high-performance IO framework with support for distributed storage system. To demonstrate the benefits of Lamda, we apply it to Pangu [14], an unified distributed storage system for Alibaba.
- We introduce the effective solution based on Lamda which will mitigate the affection of the QoS issues. Lamda provides abstract semantics based on groups, and on this basis, the realization of QoS can naturally avoid the coordination problems of multiple copies in storage application.
- We show that Lamda can completely remove the Host CPU from the critical data path. Through parallel data-path publication operations for Lamda, we can separate storage operations on the critical-path from data operations that can execute asynchronously.
- We observed that by eliminating data-copy operations, offloading critical computations and achieving predictable replicated primitives in Lamda, we can alleviate the von-Neumann bottleneck in the post-Moore era, that is, the memory and PCIe bandwidth bottleneck caused by the growth of the network bandwidth. By combining Lamda with Pangu, we can reduce host memory bandwidth by ??% and PCIe bandwidth by ??%, while achieving ??% better IOPS.
- We showcase the ability of Lamda to maintain a high-performance, availability and Service-Level Agreement (SLA). Lamda can setup different priorities for different storage tasks of the upper application, meeting SLA through flexible queue scheduling. Even during failure, Lamda enables queue escape to maintain similar throughput levels across the replication chain, as seen without failures.

2 Background and Motivation

Pangu plays a major in the core Alibaba businesses (e.g., e-bussiness and online payment, cloud computing, enhanced solid state drive backed cloud disk, elastic compute service and distributed database). As seen in our prior work [14], we focus on Storage Cluster in this paper, which mainly involves multiple replications, distributed protocols and IO concurrency control modules,e.g.

With the development of the networks and storage devices, NIC resources, memory and PCIe bandwidth have become precious commodities. We observed that such valuable resources become bottleneck when upgrading Pangu to a 100Gbps network. Firstly we elaborated on three resource contention issues on the Host which can significantly degrade Pangu performance (2.1). We then talked about a series of problems and challenges encountered by the existing solutions in combination with Pangu (2.2).

2.1 Host Resources Bottleneck

2.1.1 Memory bottleneck

As shown in Figure 1, there are three methods to transfer data from an I/O device to a processor. ① is a standard method to use traditional Direct Memory Access (DMA). In this mechanism, a processor provides a set of memory address, aka receive (RX) descriptors, to the I/O device. Later, the I/O device directly reads/writes data from/to main memory without involving the processor. Next, the processor fetches the I/O data from main memory to its cache in order to process the data. ② is a faster I/O technology which means Direct Cache

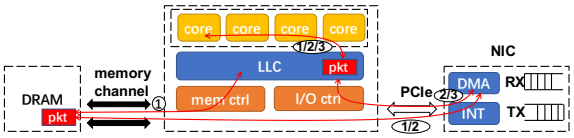
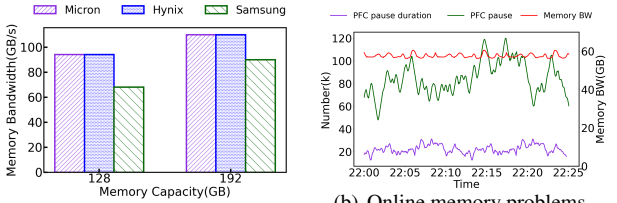


Figure 1: Different approaches of DMA for transferring data from an I/O device. ① means traditional DMA, ② means DCA, ③ means DDIO



(a) Different memory bandwidth (b) Online memory problems

Figure 2: Online Memory Condition

Access (DCA). DCA exploits PCIe Transaction Layer Packet Processing Hint, making it possible to prefetch portions of I/O data to the processor's cache. ③ is the latest technology, Data Direct I/O, which is also known as write-allocate-write-update-capable DCA. It uses this policy to update cache lines in an n-way set associative Last Level Cache (LLC), where n cache lines form one set.

Although the number of read/write to the memory is different in the above ways, when storage services incorporate persistent memory (PM), we need to flush the data from the LLC to the PM [36]. At this time, we need to write data to the memory and then read it, which means the memory bandwidth consumption is twice the network bandwidth. Several memory products used in the industry are as shown in Figure 2(a), we can find the memory bandwidth of different manufacturers' memory modules is different under the same capacity. With 200Gbps network bandwidth, we need 200/8*2=50GBps memory bandwidth for network I/O, which means nearly 50% of the memory bandwidth has been exhausted.

In distributed storage, data from the client application to the PM often needs to undergo multiple different processing (e.g., data serialization, CRC calculate and verification). And all of these storage operations will need memory bandwidth, which leads to memory bandwidth contention with network I/O. The pressure caused by the memory bandwidth will further back pressure to the network, resulting in a poor throughput. As shown in Figure 2(b), in a real online cluster with 160 nodes of Pangu with dual 25Gbps RDMA network. (half of them are storage nodes and the other are compute nodes), we randomly selected one of the storage nodes with abnormal monitoring indicators. We can see that the PFC frames number of RDMA is closed to 30k and the number of pause duration fluctuates around 80k, which is a light NIC PFC storm. And the memory bandwidth used is about 60GBps, which means the

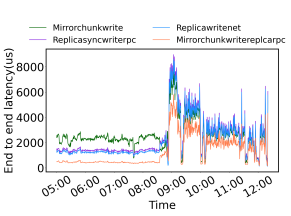


Figure 3: Online PCIe loopback problems

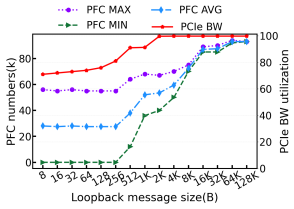


Figure 4: Bandwidth and PFC numbers of PCIe Gen 3x16 link

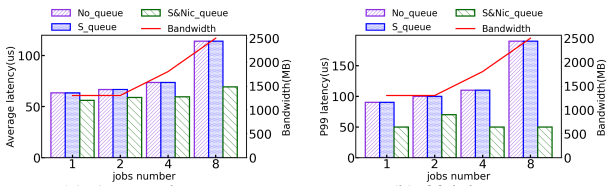
memory bandwidth is basically be used up. This is because the memory bandwidth consumption caused by dual 25Gbps is about 25*2/8*2=12.5GBps, which accounts for about 1/4 of the total memory bandwidth. At the same time, when the data arrives at the storage node, it needs to go through the fellow steps such as memory copy, data format CRC calculate and verification and data serialization. Each storage node will deploy multiple processes, which make the memory bandwidth exhausted, creating the pressure back to the network, and leading to poor throughput and delay.

2.1.2 PCIe bottleneck

PCIe is a widely used and well-established server I/O interconnect technology, which is used to connect off-chip storage, network and accelerator devices to the processor chip by the PCIe root complexes. A generic PCIe Gen 3.0*16 provides a theoretical bandwidth of 128 Gbps.

PCIe contention could occur on the PCIe Root Complex. When multiple DMA (Direct memory access) devices (e.g. GPU, SSD, NIC) initiate the read/write request to the host memory, the request should first reach the root complex [9]. As the bandwidth of the host on-chip interconnect and the main memory interface is much higher than that of the PCIe bus, the aggregated traffic may oversubscribe to the PCIe root complex, causing PCIe contention accordingly. When data frequently accesses PCIe through the NIC, the pressure caused by the above PCIe contention will further back to the network, resulting poor throughput.

Moreover, even only one NIC attached to the PCIe complex, PCIe contention will also occur because of the frequent DMA operations of the NIC. When the data needs to be transmitted through PCIe at this time, data accumulation will occur which means PCIe pressure. And this will generate back pressure to receive packet buffer and lead to poor network throughput. For example, as is shown in Figure 3, the server is equipped with 100Gbps lossless Mellanox ConnectX-5 NIC and PCIe Gen3×16. When the user opened the local file based on RDMA traffic at around 9:00, the delay of various operations rapidly increases. This is a real scene of online business, which has brought us irreparable losses.



(a) Average latency (b) 99th latency

Figure 5: NIC queue contention comparison

We make a test with the same two servers to find the cause of this problem in detail. For one server, we use loopback RDMA message with different sizes as background traffic, which is in a specific port from one QP (Queue Pair) to another/same QP under the same port, and the destination address is equal to the address of the source port. And we also generate RDMA write traffic with 8KB size from another server. As is shown in Figure 4, when loopback message size is less than 256B, although the bandwidth utilization of PCIe is less than 80%, it also generates lots of PFC pauses. With the increase of loopback message size, the number of PFC pauses gradually increases and tends to stabilize. The reason is that the MPS of the server is 256B, which means that when the message size is less than 256B, data transfer can be completed in one PCIe operation. In this process, the PPS (Packets Per Second) of the NIC processing small messages is greater than PCIe, so PCIe pressure will be generated and cause PFC pause. With the increase of loopback message size, the bandwidth utilization of PCIe is gradually increasing and the competition between local traffic and remote traffic becomes tenser. This will cause more PCIe pressure and produce more PFC pauses.

2.1.3 NIC bottleneck

Given the limited priority queues in commodity NIC, which is 8, it is not enough to serve the datacenter applications. Even in storage application, traffic of different storage services need to be isolated. However the number of traffic flows to be isolated is much larger than the hardware priority queue of the NIC. The contention of the same queue will lead to degraded performance of running applications. For example, under search-oriented storage service, which will provide storage services for online advertising search and offline advertising training. The latency of online search will 3 times worse if the traffic of online search and offline training has a contention. The reason is that the flow of online search can be understood as delay-sensitive flow, and the flow of offline training can be understood as bandwidth-sensitive flow. In production environments, multiple processes are often co-existed on the same server and are assigned to the same queue. The bandwidth-sensitive flow will block the delay-sensitive flow, causing the delay of the delay-sensitive flow to increase rapidly.

We run a set of tests to evaluate the skewed QoS. In detail, we use three devices with ConnectX-5 RNIC, two of them are used as clients and one is used as the server. Client1 generates 128KB request messages (elephant flows) and client2 generates 128B request messages (mice flows), both of them will receive 8KB response messages from the server. The latency of mice flows is as shown in Figure 5. No_queue indicates that both the switch and the NIC use a single queue. S&Nic_queue indicates both the switch and the NIC use multiple queues. S_queue indicates that only switch uses multiple queues with the single queue of NIC. Bandwidth represents the NIC bandwidth of the server. When the NIC bandwidth is full, the latency of mice flows will rise rapidly. Compared with a single NIC queue, different NIC priority queues can reduce the latency of mice flows by about 3 times.

There are two main reasons for this problem. Firstly, both flows are in the same physical queue of the receiver NIC, when the NIC queue is build up, the mice flow will be blocked by the elephant flows. Secondly, when sending data, the NIC needs to schedule the application's QP and send the corresponding data through the physical queue. The process is reversed when receiving data. However, ConnectX-5 RNIC has 32 Parallel Units (PUs), each PU can independently handle the WRs in a QP in MTU granularity. If the QP number of elephant flows is above 32, each PU needs to handle multiple QPs which increases the latency of mice flows. Multiple priority queues can ensure that mice flow of high priority can be processed first, thereby ensuring the delay of mice flows.

2.2 Existing challenges with Pangu

In the actual scenario Pangu, RDMA traffic, TCP traffic and user mode traffic share the same RNIC, which makes RDMA traffic can only be allocated to just one hardware queue. Given the current RNIC implementation cannot provide performance isolation. Meanwhile, for different upper-layer applications, data will be stored in the storage node according to different size.

3 Design

3.1 Overview

Lamda is a hardware software co-design high-performance IO framework with support for distributed storage systems using RDMA. The key idea of Lamda is remove the Host CPU from the critical data path and avoid frequent data flow movement while reducing the amount of data through the tightly coupled design of application and network. Due to differences in computing and network capabilities between hardware and software, we adopt pipeline parallelism design principles to offset the gap. Lamda exploits pipeline parallelism to publish and replicate the data, and through the coordination and cooperation between the various engines to ensure the streaming of data.

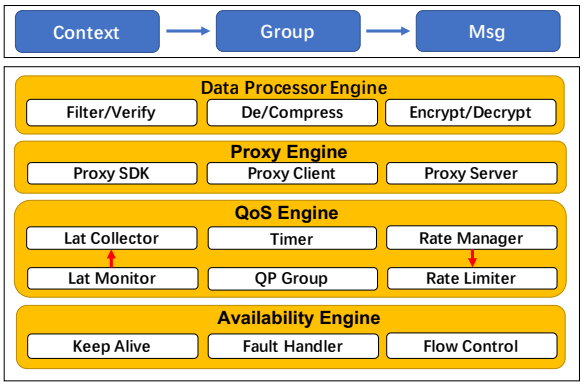


Figure 6: Overall Architecture

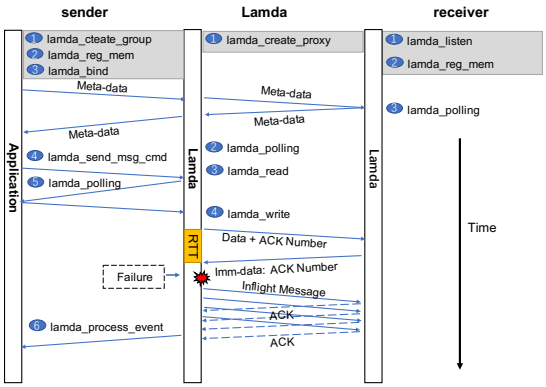


Figure 7: Per thread workflow

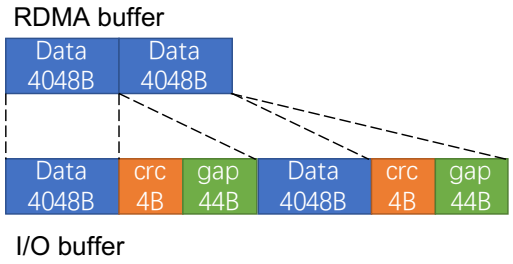


Figure 8: Potential data format

As shown in Figure 6, Lamda can be abstracted to two layers: *LibLamda* and *HidLamda*. *LibLamda* is linked to application processes running on host cores and *HidLamda* runs on the hardware. In the upper layer, Lamda provides three highly abstracted data structures to simplify user programming. In the lower layer, Lamda implements useful components for proxy, QoS, availability and data processor engines. Firstly, by providing diversified data processing engine to meet the data requirements. Secondly, this layer introduces a proxy engine to connect the user layer and the network layer. By providing a proxy virtual connection to the upper layer, the lower layer maintains the real network connections to achieve group-based primitive mapping. Through proxy engine we can achieve QP connection multiplexing and reduce network resource overhead. Furthermore, we introduce the effective solution which means QoS engine to mitigate the affection of the group QoS issues, providing unlimited priorities support to the upper layer. Through QoS engine we can guarantee low latency for small messages and high throughput for large messages. Finally, the availability engine consists of three module (keep alive, fault handler and flow control) to guarantee system availability.

Overall, by hardening parts of the engines, Lamda can avoid frequent data flow movement through the tightly coupled design of application and network. By asynchronous parallel processing through host and the Lamda, we can shield the network status for the upper later and increasing application performance for large-scale production.

What's more, the multiple engines are loosely coupled and users can reorganize different engines according to application characteristics. We aim at the characteristics of distributed storage applications and the combination of each engine is shown in the figure 6.

3.2 workflow

The workflow for each thread is shown in Figure 7. To establish the connection, the semantics exposed to the sender and receiver are different. The real network connection is established by Lamda and the receiver. Firstly, receiver starts out in listening state first, sender starts out in creating group which means it establishes a connection with Lamda (①). After that, the developer should request RDMA-enabled memory from the memory cache or register RDMA-enabled memory manually (②). Meanwhile, sender calls bind semantics to the real receiver, the parameter of this API is five-tuple information of the receiver. At this time, the sender will send the meta-data of the receiver to Lamda, and Lamda establishes a real network connection with the receiver based on the local proxy engine (③). For an instance of data transmission with polling mode, the sender only need to send meta data, Lamda will actively poll the data in the application memory and send it to the receiver (④). The receiver will send back the response with an ACK number attached to the Lamda. Additionally, Lamda can change configuration dynamically to adjust running state. Once the receiver receive all the data and the processing is completed, it will send a completion semantic to the sender through the application level.

3.3 Data Processor Engine

To improve performance and reduce memory footprint for distributed storage, it is especially important to reduce the amount of data and the number of memory copies as much as possible. Well-designed memory management and data seri-

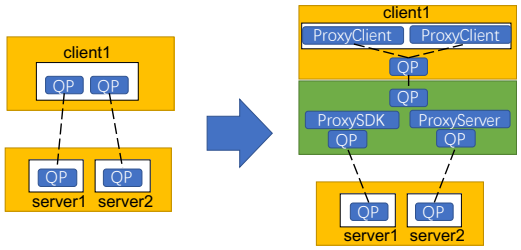


Figure 9: Lamda proxy details

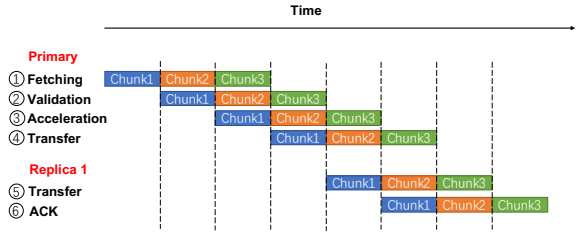


Figure 10: Replication pipeline

alization are key to achieving zero-copy during data access. Here we briefly introduce the format of data storage in Pangu as shown in Figure 8.

The received data is split into chunks of 4KB, with a 4B CRC value and a 44B gap added to each chunk. This operation is a memory- and computation-intensive operation as the calculations are applied to the entire dataset. The data are also copied when they are written into the disks in order to include CRC footers. At the same time, part of the data is also accompanied by the encryption and decryption process. All of these operations require multiple memory copies.

Lamda harden the above process and take the order of CRC first, then compression and finally encryption at the sender. In this way, stream-based data processing can be achieved and memory copies can be reduced.

3.4 Proxy Engine

As the core layer of the Lamda, the proxy engine serves as a key to connect user semantics with the hardware engines. The proxy engine in Lamda follows the three design principles.

- *The key to for software and hardware co-design:* When multiple applications share the same hardware, we should ensure the synergy between multiple applications and the resource utilization of the hardware.
- *The effective of resource reuse for diversified application:* Hardware resources are precious, we need to ensure that resources can be reused.
- *Pipeline parallelism for data replication and publication:* We should make data flow as little as possible on host and combine data operations of multiple clients through pipeline.

3.4.1 Resource reuse

As shown in Figure 9, the proxy engine is hardened to reduce resource consumption on the host. For users, proxy still provides users with the concept of two virtual connections which means proxy client. The users send meta-data through the proxy client to, and proxy server uses one-sided RDMA to fetch proxy client entries corresponding to the memory according to the control information in the meta-data. Then it finds the correct QP connection through the proxy SDK and sends the data to the receiver. Multiple applications can share the real QP connection through proxy SDK.

Take one client thread and two server threads with 36 IODepths for example. In the traditional case, we will need 2*2*36 = 144 resources. When Lamda is introduced, we only need 2*36 = 72 resources, which is half of the original resources to host.

3.4.2 Pipeline replication

Through the proxy engine in Lamda, we can replicate data asynchronously and proactively. Lamda synchronously replicates any remaining data because it uses pipeline parallelism to accelerate asynchronous replication. The replication pipeline consists of five stages:fetch, validation, acelerator, transfer and acknowledgement (ACK).

As shown in Figure 10, a pipeline chunk is first fetched (①) and validated (②). Then Lamda accelerate the data (compression, encryption e.g.) (③). Finally, it transfers the chunk data to the next replica's Lamda (④). Each replica sends an ACK to the primary, after copying the pipeline chunk to the local PM. These steps happen proactively in the background and the host application does not need to be informed.

3.5 QoS Engine

The principles of the QoS engine in Lamda follow from its requirements.

- *Isolation without sacrificing utilization:* When small and large messages coexists, performance isolation cannot be guaranteed. We should ensure the low latency of small messages, while increasing RNIC resource utilization.
- *No admission control of small messages:* The latency of small message is less than 10 microseconds. So we cannot add additional delay to it, we should allow to send them immediately whenever application want.
- *High precision rate control for large messages:* Given that RDMA messages can range from bytes to gigabytes, and the bandwidth of RNIC is up to 25/40/100Gbps, the

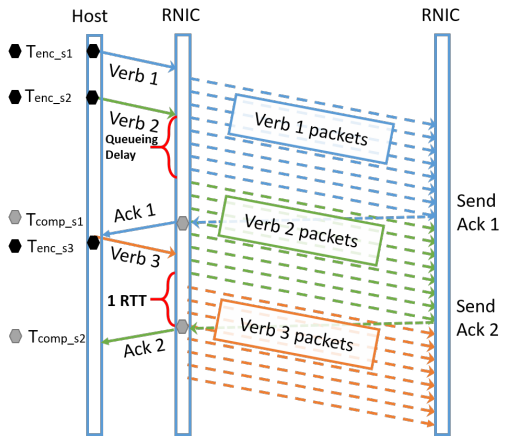


Figure 11: Queueing Delay Estimation

design need to respond to high latency faster and adjust the sending rate of large messages more precisely.

3.5.1 Queueing Delay Measurement

Given that when the application posts the message into the QP, the message will be taken over by RNIC. we can just get the time when the message is posted into the QP(t_{enq}), and cannot get the time when the last byte of the message is sent out by RNIC. So we cannot calculate the queueing delay directly. But RNIC also provides a high precision timer for timestamping the completion of the message(t_{comp}). This means that we can get the time when receiving the last ACK, and the queue delay of Work Completion(WC) in the Completion Queue(CQ) is also excluded.So We can get the accurate enqueueing time(t_{enq}) and completion time of the message(t_{comp}), yet t_{enq} is not always accurate to use as the actual sending time(t_{start}) because of the queueing at end host. RNIC interfaces can only get the uncompleted bytes in one QP, which includes the bytes queueing in the QP, in flight, queueing in the CQ. Furthermore, the queueing delay between the QPs also cannot be measured directly. Figure 11 shows this scene. The enqueueing time of *Verb*2 is not the actual sending time, because RNIC is handling *Verb*1. However the sending time of the first byte of *Verb*2 is one RTT earlier than the completion time of *Verb*1. This one RTT is the time when the last byte of *Verb*1 is sent until the ACK of *Verb*1 is received. So we get the actual sending time of *Verb*2.

Lamda use this property to calculate queuing delay [21]. The completion time of the last message(t_{last_comp}) is known, and this time is exactly one network RTT after the last byte of the message was sent, which is also when the first byte in the current message will be sent(t_{start}). So the actual sending time(t_{start}) is:

$$t_{start} = max(t_{enq}, t_{comp_{last}} - RTT_{last}) \quad (1)$$

And the new RTT can be computed as:

$$RTT = t_{comp} - t_{start} - \frac{size_{msg}}{rate_{cc}} \quad (2)$$

Finally, we can get the queueing delay(t_{queue}):

$$t_{queue} = t_{start} - t_{enq} \quad (3)$$

Lamda does not interrupt or interact with small messages. Given that it takes just 60ns to call *ibv_post_send*() to send RDMA message, and less than 2μs to complete 1KB-message, any scheduling and buffering algorithm would increase the overall delay of small messages. So QoS engine just record the time when application posts the message into the QP, get the timepoint from Work Completion and calculate the queuing delay. This operations just add less than 100*ns* to one small message.

Pseudocode 1 QoS algorithm

```
procedure Calculate_current_rate
    if No_Small_Messages then
        F = F_the_whole_max
    else
        if D_curr > D_target && Util_curr > Util_min then
            F = F * max((1 − α * (D_curr − D_target)/D_curr), 1 − max_mdf)
        else
            F = F + unit * D_min/D_curr
        end if
        if F > F_max then
            F = F_max
        end if
        if F < F_min then
            F = F_min
        end if
    end if
end procedure
```

3.5.2 Rate Limiting for Large Messages:

Given the measurements above, Lamda maintains a sliding window of the most recent $RefCount$ (=10000) queueing delay measurements, and uses a count-min sketch on that window to estimate D_{curr}, This is fed into the F computation algorithm described below. If D_{curr} is higher than D_{target}, Lamda can decreasing the sending rate of large messages. When D_{curr} is lower than D_{target}, Lamda tries to increase the sending rate of large messages.

Consideration for RNIC utilization: In the case of blocking, we use rate control of large messages to guarantee the low latency of small messages. There is an important problem is how to avoid the waste of RNIC utilization in the process

of rate regulation. For example, when the latency of small messages util the target latency is met. In this process, the total bandwidth of all messages is much lower than the RNIC Maximum bandwidth. In addition, how to control the granularity of rate regulation to ensure that it can respond to the change of queuing delay in time.

In the absence of small messages, this situation is simple. F is set to total RNIC bandwidth($F_{the\_whole\_max}$), where $F_{the\_whole\_max}$ is pre-determined on a per-RNIC basis using benchmark messages.

In the presence of small messages, the overarching goal of Lamda boils down to continuously maximizing F while the current D_{curr} estimation is less than D_{target}. To continuously update F, Lamda first uses a simple AIMD scheme that reacts to D_{curr} every $RefPeriod$ interval as follows. If D_{curr} is larger than D_{target}, Lamda decreases F multiplicatively, with the decrease depending on how far the dalay is from the target. If the estimation is below D_{target}, Lamda slowly increases F, and small messages are highly sensitive to the rate adjustment. we define $RateUnit$ as the bandwidth required to send messages in the way that sending next message when the previous one is completed. So we increase the F by just one combination of $RateUnit$ and D_{curr}each time, Which means if D_{curr} is small, we can increase the speed of large messages appropriately and quickly. Considering that the scheduling of messages is based on flows(QPs), we set $RateUnit$ to the basic bandwidth required by a flow(QP). The depth of QP is 1 and the size of message sent by this QP is MTU. We decide that this is the basic bandwidth needed for a flow(QP).

What if Target Queueing Delay is Unreachable? A key consequence of the isolation-utilization tradeoff is that D_{curr} may sometimes be unreachable - e.g., when it is set too low or in the presence of too many QPs sending large messages. This can cause under utilization as QoS engine continuously try to reduce D_{curr} without success while limiting the sending rate of large messages.

We address this issue by providing an extra parameter $Util_{min}$, which means the minimum utilization of the RNIC that can be allowed. If D_{curr} is higher than D_{target} and current RNIC link utilization $Util_{curr}$ is lower than $Util_{min}$, QoS engine assumes that D_{target} is unreachable. Because if we continue to reduce the sending rate of large messages, not only can D_{curr} not meet Q_{target}, but also the RNIC link utilization declines. Qos engine can ignore small messages altogether and focus on equally sharing resources among large messages.

By doing this, RNIC link utilization can be guaranteed to be above $Util_{min}$. If the estimation is below D_{target}, Lamda slowly increases F, because $Util_{curr}$ ranges between $Util_{min}$ and total RNIC bandwidth, and small messages are highly sensitive to the rate adjustment. It may need to come out of this state only when $Util_{curr}$ changes. Specifically, when $Util_{curr}$ becomes even smaller, it can stay in the same state. Only when $Util_{curr}$ increase above $Util_{min}$, QoS engine can go back to the original algorithm and try to attain D_{target} again.

Our conservative AIMD scheme works well in practice and the complete rate control of large messages can be shown in Pseudocode 1.

3.6 Availability Engine

Based on the native reliability of RDMA, Lamda implement Keepalive mechanism. In our implementation, a probe request will be triggered if either side fails to communicate with peer side more than t ms. Lamda will automatically respond to an acknowledgement to declare its aliveness. To reduce any negative impact on overall performance, the payload size of this probe request is zero.

If the connection is broken, corresponding resource will be released immediately to avoid connection leaks. At the same time, Lamdawill select a new QP from the QP pool to avoid time consumption caused by initializing resources. In this stage, the host application is not aware of fault handling.

As a reactive congestion control method, we observed that DCQCN fails to perform very effectively in the large-scale cluster with heavy incast. The flow control here adopts our previous work. As a switch, the user can also adjust the flow control method according to whether or not PFC is needed.

4 Implementation

With the emergence of multi-core SoC (syetem-on-a-chip) SmartNICs [1, 2], they bridge the gap between increasing network bandwidth and stagnating CPU computing power and memory and PCIe bandwidth because they can implement the data path for disaggregated storage operations. SmartNICs integrate compute cores and memory into the network interface, plus accelerators for common packet-processing functions. Such devices also support both block-level remote access protocols, such as NVMe over fabrics (NVMe-oF) and remote direct memory access (RDMA). This provides us with a new opportunity to realize Lamda.

We implement Lamda in x86 Linux and ARM-based Mellanox Bluefield2 SmartNICs. This is a 2x100Gb/s SmartNIC that contains an ARMv8 Cortex-A72 processor with 8 cores. Each core has a 32 KB L1 data cache. Two cores share 1MB of L2 cache and all cores share 6MB of L3 cache, as well as 16GB of memory.

4.1 SmartNIC system

Liu et al. [26, 34] provide an extensive analysis of their characteristics and behavior. Compared to such SmartNICs, Lamda select an off-path SmartNIC, an RDMA switch on the SmartNIC is capable of directly accessing SmartNIC and host memory. The switch can be configured to forward RDMA requests according to various rules. Compared to on-path SmartNIC (LiquidIOII CN2350/CN2360), it allow us to treat

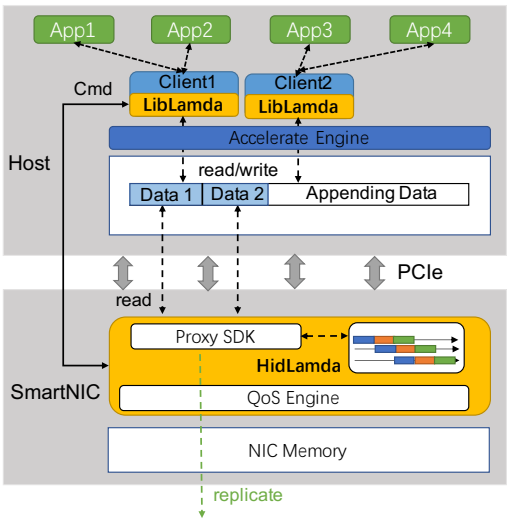


Figure 12: SmartNIC architecture

the SmartNIC a independent node and we can full control the system from any machine, over where to transfer data and whether to interact with the SmartNIC of the host.

As shown in Figure 12, we offload the *HidLamda* to the SmartNIC, and to make offload worthwhile, we run it as an independent process on SmartNIC's Linux OS. *HidLamda* uses one-sided RDMA to fetch client data and caches the hot data in SmartNIC's DRAM to avoid frequent access to host memory via PCIe.

4.2 Active polling

5 Evaluation

We evaluate the performance of Lamda based on the controlled experiments. We apply testbed experiments to validate the effectiveness of Lamda in comparison with the real production solutions.

5.1 Experimental Setup

5.2 Micro-benchmark

5.3 Macro-benchmark

6 Deep dive

6.1 GPU + DPU

6.2 Discussion

The existing SmartNIC is still insufficient.

7 Related work

In the context of high-performance network applications, several papers have observed part of the bottlenecks and its interaction with the host architecture. Kalia et al. [10, 17] provide a simple evaluation and suggestion for RDMA primitives and how they interact with PCIe and avoid NIC cache misses. Lim et al. [24]. discussed the influence of NUMA on the DMA of the device in detail. Guo et al. [16] raised the issue of slow receiver for the first time, which will slow down network performance because of NIC bottleneck. Flajslik et al. [13] presents a high-level picture of the PCIe interactions between an Ethernet NIC and CPUs. However, all these touch on some aspects covered in this paper but in the context of some higher level applications, which can not satisfy versatility and there is no way to solve the root cause. Neugebauer and Goldhammer et al. [15, 30] argue many runtime factors usually impact communication performance. With respect to latency, DMA engine queueing delay, PCIe request size, and host DRAM access costs will all slow down PCIe packet delivery performance. They provide a reference for the evolution of PCIe and NIC, but can not give the urgently needed evolution directions.

Recent RMT switches and Smart-NICs enable programmability along the packet data plane. Ming et al. [6, 22, 27, 32] are in-network caching fabrics with some basic computing primitives. It has proposed the use of in-network computation to offload compute operations from end-hosts to these network devices. But they ignore the advantages of NIC, which will allow the performance of the application to be improved again and avoid host contention.

8 Future work and Conclusion

We are self-developing a new type of NIC, which could also be called DPU (Data Processing Unit). It is between the memory and CPU on the memory bus, and connects to the NIC. We argue it will become a new intermediate infrastructure node that can accelerate infrastructure functions, including storage virtualization, network virtualization, and security with dedicated protocol accelerators.

In this paper, we propose the host bottlenecks caused by the growing network: memory contention, PCIe contention and NIC contention. What's more, this is the first time that we have conducted a detailed and systematic analysis of the above three contention. We present the design of Lamda, a hardware software co-design high-performance IO framework with support for distributed storage system. And based on it combined with Pangu to achieve real deployment verification. Through effective QoS solution we guarantee the latency of small messages and the throughput of large messages. Finally, we shon that Lamda can completely remove the host CPU from the critical data path and alleviate the host bottlenecks. By combing Lamda with Pangu, we can reduce host mem-

ory bandwidth by ??%, and PCIe bandwidth by ??%, while achieving ??% better application performance.

We hope that through this paper, people can realize the importance of the host bottleneck caused by the growing network, and that the NIC of the whole link should be data-centric and be elevated to the position of the first citizen.

References

[1] Bluefield2 smartnic ethernet, 2021. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[2] Broadcom stingra, 2021. https://www.broadcom.com/blog/at-a-glance--the-broadcom-stingray-ps1100r-delivers-breakthrough-performance-and-efficiency-for-nvme-of-storage-target-applications.

[3] M. Alian and N. S. Kim. Netdimm: Low-latency near-memory network interface architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 699–711, New York, NY, USA, 2019. Association for Computing Machinery.

[4] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 802–814, 2018.

[5] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1463–1475, New York, NY, USA, 2015. Association for Computing Machinery.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, Aug. 2013.

[7] 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.

[8] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[9] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng, Y. Guo, X. Jiang, L. Tang, Y. Du, Y. Zhang, P. Pan, and Y. Xie. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 610–622, 2020.

[10] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[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 *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 283–295, 2015.

[12] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, 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. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, Apr. 2018. USENIX Association.

[13] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 333–346, San Jose, CA, June 2013. USENIX Association.

[14] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, Apr. 2021.

[15] A. Goldhammer and J. Ayer. Understanding performance of pci express systems, 2008.

[16] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[18] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery.

[19] 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 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[20] G. Kumar, N. Dukkipati, K. Jang, 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 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*,

SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Y. Le, B. Stephens, A. Singhvi, A. Akella, and M. M. Swift. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 225–236, New York, NY, USA, 2018. Association for Computing Machinery.

[22] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[23] 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 *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 696–709, 2018.

[24] T. Li, Y. Ren, D. Yu, S. Jin, and T. Robertazzi. Characterization of input/output bandwidth performance models in numa architecture for data intensive applications. In *2013 42nd International Conference on Parallel Processing*, pages 369–378, 2013.

[25] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, Nov. 2020.

[26] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[27] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGPLAN Not.*, 52(4):795–809, Apr. 2017.

[28] 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 *2016 IEEE International Conference on Big Data (Big Data)*, pages 253–262, 2016.

[29] 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.

[30] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[31] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, Feb. 2019. USENIX Association.

[32] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, Oct. 2018. USENIX Association.

[33] Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 772–787, New York, NY, USA, 2021. Association for Computing Machinery.

[34] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[35] 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 '15, New York, NY, USA, 2015. Association for Computing Machinery.

[36] X. Wei, X. Xie, R. Chen, H. Chen, and B. Zang. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 523–536. USENIX Association, July 2021.

[37] 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.

[38] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer. Imagenet training in minutes, 2018.