SmartDS: Middle-Tier-centric SmartNIC Enabling Application-aware Message Split for Disaggregated Block Storage

Jie Zhang[†], Hongjing Huang[†], Lingjun Zhu[◇], Shu Ma[◇], Dazhong Rong[†], Yijun Hou[◇], Mo Sun[†], Chaojie Gu[‡], Peng Cheng[‡], Chao Shi[◇], Zeke Wang[†]

[†]Collaborative Innovation Center of Artificial Intelligence, College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[◇]Alibaba Group, Hangzhou, China
[‡]State Key Laboratory of Industrial Control Technology, College of Control Science and Engineering, Zhejiang University, Hangzhou, China

ABSTRACT

The widespread deployment of storage disaggregation in the cloud has facilitated flexible scaling and storage overprovisioning, allowing for high utilization of storage capacity and IOPS. Instead of utilizing remote storage protocols to access remote disks, a middle-tier is introduced between compute servers and storage servers in order to serve I/O requests from compute servers and provide computations such as compression and decompression. However, due to the need for a cloud to concurrently serve millions of VMs that require access to disaggregated storage, the middle-tier requires a massive number of servers to process network traffic between computing and storage nodes. For example, a major cloud company may deploy hundreds of thousands of high-end servers to provide such a service for its cloud storage, because the existing CPU-based middle-tier suffers from a severe issue of compute-intensive compression/decompression on high-throughput storage traffic. To address this issue, we introduce SmartDS, a middle-tier-centric SmartNIC that serves storage I/O requests with low latency and high throughput, while maintaining high flexibility and programmability. The key idea behind SmartDS is the application-aware message split (AAMS) mechanism, which allows for the processing of the message's header on the host CPU to achieve high flexibility, and the message's payload on the SmartDS. Experimental results demonstrate that SmartDS provides up to 4.3× more throughput than a CPU-based middle-tier and enables the linear scale-up of multiple network ports and multiple SmartNICs, thus significantly reducing cloud infrastructure costs for disaggregated block storage.

CCS CONCEPTS

• **Hardware** → *Networking hardware*; • **Networks** → *Layering*.

KEYWORDS

SmartNIC, Middle Tier, Disaggregated Block Storage, Payload/-Header Split

ACM Reference Format:
Jie Zhang[†], Hongjing Huang[†], Lingjun Zhu[◇], Shu Ma[◇], Dazhong Rong[†], Yijun Hou[◇], Mo Sun[†], Chaojie Gu[‡], Peng Cheng[‡], Chao Shi[◇], Zeke Wang[†] . 2023. SmartDS: Middle-Tier-centric SmartNIC Enabling Application-aware Message Split for Disaggregated Block Storage. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589077

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589077

1 INTRODUCTION

Storage disaggregation is a widely adopted approach in cloud computing, with prominent examples including Amazon's Elastic Block Store (EBS) [6], Alibaba cloud's Elastic Block Storage [25, 53], and Windows Azure Storage (WAS) [15]. By decoupling compute servers (hosting virtual machines) and storage servers (hosting disks), storage disaggregation enables scaling CPU and flash resources independently in a cost-effective manner [41]. Additionally, cloud vendors can over-provision storage resources, thus decreasing monetary costs.

Rather than relying on remote storage protocols such as iSCSI and NVMe-oF to access remote disks, disaggregated block storage requires a *middle-tier* between compute servers and storage servers in real clouds [15, 25, 41, 53]. The middle tier is composed of a large number of middle-tier servers, which serve storage I/O requests from compute servers and forward data from these requests to remote storage servers. Furthermore, the middle-tier provides maintenance services such as replication, fail-over, and snapshot functionality, making them a critical component of modern storage disaggregation systems. Given that a cloud must serve a vast number of VMs simultaneously, a significant number of middle-tier servers are present in commodity block storage systems. For example, Alibaba Cloud's Elastic Block Storage contains over 100,000 middle-tier servers [53]. In the following, we outline the key characteristics of middle-tier servers, which include high computing intensity and flexibility.

R1: High Computing Intensity. To reduce storage costs, the middle-tier compresses data blocks before writing them to remote storage server disks. When serving read requests, the middle-tier must decompress the data returned from the remote storage server and return it to virtual machines (VMs). Although data compression is beneficial in terms of storage costs, it comes at the expense of computational cost [1, 16].

R2: High Flexibility. Commodity disaggregated block storage systems must not only provide high performance but also high flexibility [2, 21]. As block storage systems are fundamental components of cloud applications, vendors must ensure that they can evolve quickly enough to keep up with customers' various needs and the development of new technologies such as faster networking

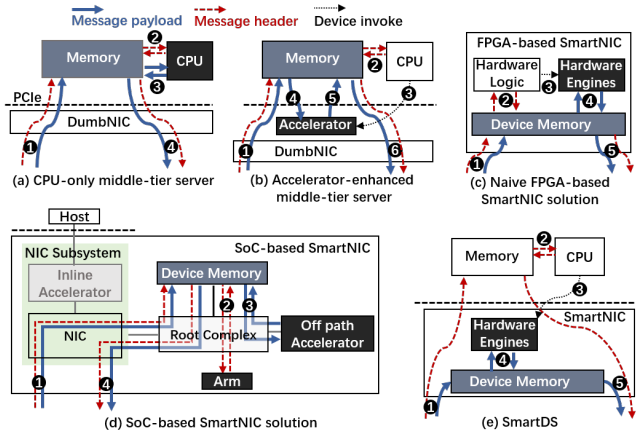


Figure 1: Comparison of different middle-tier designs. The colored components indicate that they are heavily used.

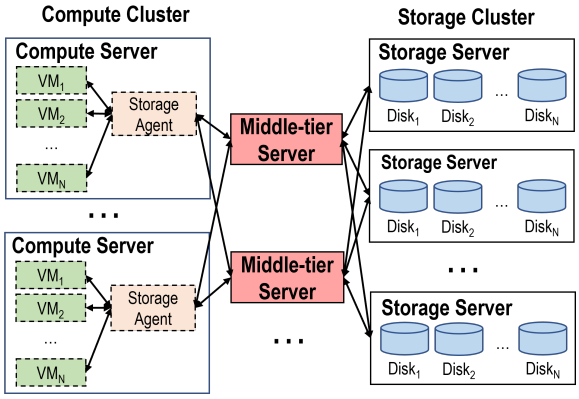


Figure 2: The typical architecture of a disaggregated block storage system

and storage mediums. This agility allows vendors to offer competitive solutions to customers.

However, the existing middle-tier design fails to satisfy the above two requirements concurrently.

CPU-only Middle-tier. Compression occupies the majority of CPU cores in a CPU-based middle-tier [53], as shown in Figure 1a. Even worse, this traditional design faces two trends that exacerbate its performance issue. First, network speeds grow rapidly [62], with 100 [11, 51] and 200 [52] Gigabit Ethernet (GbE) network interface controllers (NICs) already widely available, 400 [10] GbE arriving in 2021 and 800 [19] GbE expected in the near future. Second, PCIe-based flash devices are increasingly prevalent in clouds, providing high-performance storage services with I/O operations per second (IOPS) in the millions and latencies in the tens of microseconds [34].

Accelerator-based Middle-tier. In order to address computing challenges from compression, prior studies leverage GPUs [60, 72, 86], FPGAs [23, 66], and specialized accelerators [31] such as Intel QuickAssist Technology (QAT) [33] to alleviate the computational cost of the compression/decompression. We categorize the middle-tier that leverages these approaches as the accelerator-enhanced middle-tier. The accelerator-enhanced middle-tier can save a lot of CPU cores from compression. However, such a design doubles PCIe traffic, which can easily become a new bottleneck as shown in Figure 1b.

Naive FPGA-based SmartNIC. In order to address the PCIe overhead, offloading the entire middle-tier logic into FPGA-based SmartNIC becomes a new option. However, middle-tier logic iterates very fast, thus offloading control logic to FPGA hardware logic lacks flexibility, as shown in Figure 1c.

SoC-based SmartNIC. SoC-based SmartNIC features small CPU cores and hardware engines to provide both flexibility and computing intensity as shown in Figure 1d. However, current off-the-shelf and upcoming SoC-based SmartNIC have certain limitations in the scenario of middle-tier due to the form factor and power constraints. Firstly, these SmartNICs lack sufficient compression engines, and their CPUs are too wimpy to perform compute-intensive tasks on line-rate network traffic. Secondly, their memory subsystem is relatively weak, compared with their networking capabilities.

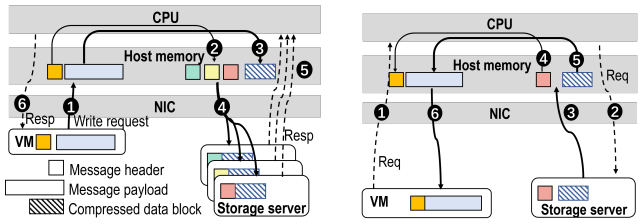
To this end, we propose SmartDS, a middle-tier-centric SmartNIC that can serve storage I/O requests with low latency and high throughput while maintaining high flexibility and high programmability. SmartDS consists of three key innovations. First, it features an application-aware message split (AAMS) mechanism that enables the processing of message headers on the host CPU, providing flexibility, while the message payload is processed on the SmartDS, achieving high throughput. Second, due to its low CPU and PCIe overheads, SmartDS enables efficiently utilizing multiple network ports on the SmartNIC and enables linear scaling by installing multiple SmartNICs on a server. Third, SmartDS exposes a RDMA-like high-level API to guarantee high programmability.

We prototype SmartDS on an HBM-enhanced Xilinx VCU128 FPGA. SmartDS enables us to fully leverage multiple 100Gbps network ports (up to 6) on SmartNIC. The experimental results show that our prototype can provide up to 4.3× throughput than CPU-based middle-tier, and reduce the average latency, 99th percentile latency, and 999th percentile latency by 2.6×, 3.4× and 3.5×, respectively. And SmartDS enables linear scale-up by installing multiple SmartNICs (up to 8 in our 4U server) to improve achievable throughput. As such, SmartDS reduces the required number of middle-tier servers by 51.6× and thus significantly reduces cloud infrastructure cost for disaggregated block storage.

2 BACKGROUND

2.1 Disaggregated Block Storage

Figure 2 demonstrates the typical architecture of a block storage system that adopts a storage disaggregation design. It provides persistent data hosting in virtualized disks (VDs) to cloud users. This architecture places compute servers (that host virtual machines, VMs) and storage servers (that host VDs) in separate clusters. VMs in a compute server organizes data in logical block addressing (LBA). There is a mapping of LBA to the segment address of the physical disks. Segments (e.g., 32GB) are managed by middle-tier, which would divide them into chunks (e.g., 64MB), and each I/O request (e.g., 4KB data block) from a VM targets a chunk. Each write request would be replicated to several (usually three) different storage servers, which are in charge of the standalone back-end storage of



(a) Write requests (b) Read requests

Figure 3: Middle-tier server data flow when serving I/O requests

the blocks and device management. There are three main advantages of the disaggregated design. First, compute and storage servers can be designed independently to optimize either type of server for its target workloads, resulting in more cost-efficient specialized servers [53]. Second, vendors can over-provision storage resources, thus decreasing monetary costs. Third, with persistent states stored in storage servers, migrating application services across compute servers is fast and straightforward.

When a VM sends an I/O request through its storage agent, the request is forwarded to the corresponding middle-tier server. The addition of a middle-tier between compute servers and storage servers provides three main benefits. Firstly, running storage operations such as LSM-Tree compaction and periodical data scrubbing on compute servers requires additional processing and maintenance services (e.g., snapshot, fail-over, replication), all of which can cause significant and often imbalanced overhead to the precious CPU resources in compute servers. Secondly, in the absence of a middle-tier, sharing a Virtual Disk (VD) among compute servers requires maintaining shared states among them with strong consistency, which is much more challenging than maintaining them in the middle tier. Thirdly, to reduce the monetary cost of storage, data is compressed before being written to disk, and this operation is expensive for the precious CPU resources in compute servers.

2.2 Roles of Middle-tier

In this section, we provide a detailed description of the roles that the middle-tier plays in modern commodity disaggregated block storage systems. We can categorize these roles into two classes: real-time services (serving read or write requests) and maintenance services (LSM-tree compaction, fail-over, snapshot, etc).

2.2.1 Serving Write Requests. As illustrated in Figure 3a, the process of writing a data block (usually 4 KB) from a VM involves issuing a network request to the middle-tier server responsible for the chunk to which the block belongs (❶). The network message comprises a block storage header containing the VM's unique ID, service type, block offset, segment ID, and other relevant information, as well as a payload (the data block to be written). The reliability of the message is guaranteed by the transport layer, which is typically RDMA or a variant.

When the message arrives at the middle-tier server's memory, the server first parses the block storage header (❷) and makes some decisions including:

- Choose several remote storage servers (usually three [53]) according to disk usage, distribution of switches, loads of storage servers, and disaster recovery strategy.
- Whether the block should be compressed and what compression effort should be used according to service type and CPU load. Generally, workloads' higher tolerance for latency and more idleness of the middle-tier server CPU means that the data block would be compressed with more computing time (thus a better compression ratio). Some data blocks may even be compressed many times for a better compression ratio.

After that, the software would compress the data block (❸) and send the compressed block to three chosen storage servers (❹). Storage servers would write the data into the disk in an appended way and return with success to the middle-tier server (❺). Once all the storage servers have confirmed the successful write, the middle-tier would acknowledge the VM with a success (❻) and then the write request completes.

Why Compression is in Middle Tier? Compression at the compute server can reduce network traffic earlier while saving storage capacity. However, in the modern cloud, compression at the compute server means each compute server's hypervisor card must reserve enough compression resources (CPU/FPGA/ASIC) according to the peak storage traffic, wasting computing power when storage traffic is under-loaded. Compression at the middle tier means that compression ability can serve different compute servers dynamically, achieving high utilization of compression resources.

2.2.2 Serving Read Requests. As shown in Figure 3b, when a VM issues a read request to the middle-tier server (❶), the middle-tier server identifies the corresponding data block and sends read requests to one of the storage servers (❷). The storage server then read the requested data blocks from the disk and sends them back to the middle-tier server (❸). The middle-tier server parses the message header sent from the VM (❹). Based on the parsed results, the middle-tier server decompresses the received payload (❺). Finally, the middle-tier server generates a new message header and assembles it with the decompressed data into a complete message, which is sent back to the VM (❻).

2.2.3 Maintenance Services. There are maintenance services running in the middle-tier such as LSM-tree compaction, disk garbage collection, fail-over, and snapshot.

When handling write requests, the middle-tier server would not release the memory that holds the write request even if the request has finished. Once the number of writes in a chunk reaches a threshold, the LSM-tree compaction service running in the middle-tier server performs compaction on the blocks stored in memory. The result of the compaction is sent to remote storage servers for persistence. Additionally, the garbage collection service releases the disk space of the data blocks that have been compacted.

In a commodity middle-tier server, real-time services occupy more than half of the CPU resources. This paper focuses on the processing of write requests for two reasons: (1) The number of write requests is much more than that of read requests (around 5× [53]); (2) A CPU core's decompression throughput is much higher than compression (more than 7× [49]).

3 MOTIVATION

A middle-tier server needs to be capable of (1) handling I/O pressure on the host memory subsystem and PCIe interconnect, (2) handling computation pressure, and (3) providing flexibility for the entire middle-tier server applications. In the following, we analyze existing solutions that only partially meet these requirements, which motivates the design of SmartDS.

3.1 Traditional CPU-based Middle-tier Server

Figure 1a illustrates the primary data flow of a CPU-based middle-tier server. Network messages are transmitted to the middle-tier server's host memory via PCIe and NIC (❶). Two types of large-size messages are primarily handled: the write request (from VM), which includes the data block that needs compression, and the read reply (from the storage server), which contains the data block that needs to be decompressed. After receiving the message, the CPU parses the message header (❷) and performs compression or decompression (❸) on the message payload. Finally, the CPU forwards the processed message to storage servers (or VMs) through PCIe and NIC (❹). A CPU-based middle-tier server allows for explicit control over each network message by the host CPU, providing adequate flexibility. However, the high volume of messages can overwhelm the host memory subsystem and PCIe bandwidth. Moreover, compute-intensive compression on message payloads can exhaust CPU cores.

3.1.1 Computation Pressure. Compression can reduce the storage cost and alleviate the network amplification caused by replication. However, the benefits of data compression come at the expense of computational cost [1, 16].

An Intel i7-9700K CPU @ 4.9GHz with turbo boost can run LZ4 [8] lossless compression at 780 MB/s [49] per core, while 100/200 Gigabit Ethernet (GbE) network interface controller is already widely available [52] in modern clouds. In a typical middle-tier server, to saturate network bandwidth, nearly half of the CPU resource is occupied by data compression.

Moreover, network bandwidth is increasing at a faster rate than CPU performance [61], making it increasingly difficult for CPU-based middle-tier servers to keep up with the demands of modern clouds.

3.1.2 Memory Bandwidth Pressure. The growth of network bandwidth not only places computational pressure on the CPU but also on the memory subsystem, which has been extensively explored in prior works [3, 74, 77, 78].

To illustrate the memory interference caused by network packets, we use Intel Memory Latency Check (MLC) [36] tool to inject dummy memory requests at different rates. All CPU cores in the server are running MLC. The server has eight memory channels and its achievable memory bandwidth is around 120 GB/s. The server has a 16 MB last-level cache and DDIO occupies 2 out of total 11 ways. To minimize the influence of CPU interference, we use one-sided RDMA to simulate a packet forwarding application. The client uses `RDMA READ` and `RDMA WRITE` to access remote memory in the server. Both machines are equipped with a 100 GbE NVIDIA ConnectX-5 NIC [51] and the RDMA message size is large (4 MB).

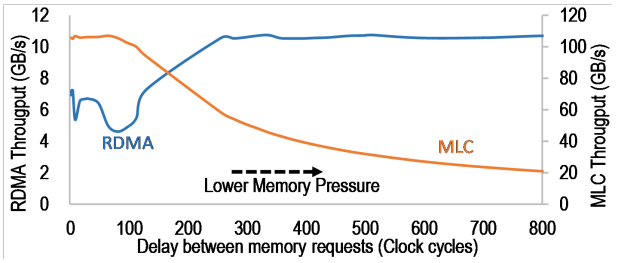


Figure 4: RDMA throughput at different memory pressure levels

Table 1: PCIe latency under different pressure

	H2D Latency (us)	D2H Latency (us)
Under Loaded	1.4	1.4
Heavily Loaded	11.3	6.6

Figure 4 illustrates the achieved RDMA throughput and total MLC throughput of all cores when varying the delay between injected memory requests. We observe that RDMA throughput significantly drops when memory pressure from MLC increases. At the maximum memory pressure (i.e., no delay), the one-sided RDMA-based packet forwarding can only deliver ~46% of the achieved bandwidth without interference. This demonstrates that memory pressure can significantly intervene with achievable network throughput.

3.1.3 Interconnect Pressure. In cloud servers, the PCIe interconnect is typically used to connect the NIC and host. However, when the network is heavily loaded, the PCIe interconnect can become saturated and cause performance degradation.

We designed a micro-benchmark to illustrate that heavily loaded PCIe interconnect can contribute to extra network latency. As NIC leverages its DMA to read data from or write data to host memory through PCIe, we use a Xilinx U280 FPGA [82] to call its DMA to read from or write to host memory and measure PCIe latency. We adjust the duration that the FPGA issues request to make PCIe interconnect under-loaded or heavily loaded.

We measure the H2D (Host to Device, DMA read) latency and D2H (Device to Host, DMA write) latency and the results are presented in Table 1. Both H2D and D2H latency increase significantly when the PCIe interconnect is heavily loaded. This suggests that when the network is heavily loaded, the PCIe interconnect may introduce additional latency to the end-to-end network latency.

This issue cannot be resolved by simply upgrading the PCIe interconnect, as current NIC products provide comparable PCIe bandwidth and network bandwidth. For example, NICs with PCIe 3.0X16 can achieve a throughput of around 104 Gbps and are typically equipped with a single 100 Gbps networking port [51], while NICs with PCIe 4.0X16 are equipped with a 200 Gbps networking port [52].

3.2 Accelerator-enhanced Middle-tier Server

Since CPU-based middle-tier server suffers from severe computation pressures, it is natural to offload compute-intensive compression/decompression into accelerators and thus relieve the pressures. To our knowledge, this design has been deployed in the real disaggregated block storage system at a large scale.

Figure 1b describes the data flow in an accelerator-enhanced middle-tier server. First, data blocks are written into the middle-tier server's host memory through NIC and PCIe (❶). Second, the host CPU parses the message header (❷) and controls the accelerator (❸) to fetch the message payload in the host memory (❹). When processing completes, the accelerator writes results back to the host memory and notifies the CPU (❺). At last, the CPU sends the compressed message out through PCIe and NIC (❻).

Therefore, the accelerator-enhanced middle-tier server can relieve computation pressure while maintaining flexibility, because each network message is explicitly controlled by the host CPU. However, the messages' payload still stays at the host memory, thus the memory pressure still keeps the same as the CPU-based middle-tier server (Subsection 3.1), and PCIe interconnect pressure doubles because messages' payload goes through PCIe twice as that of CPU-based middle-tier server.

Potential Optimization with DDIO. After a received packet that occupies a cache line has been processed and sent to the network, the occupied cache line does not need to be evicted to host memory, but instead, can be immediately reused for the following packets. Ideally, a packet-forwarding application with a receive ring small enough to fit within the Direct data I/O (DDIO) capacity [32] would not occupy memory bandwidth. DDIO technology can serve DMA reads of I/O devices (e.g., NIC, accelerators) from the last level cache (LLC), and it allows DMA writes to allocate up to two LLC ways, thereby bypassing memory [62]. Thus NIC and the accelerator can exchange data using LLC. However, Subsection 2.2.3 mentions that the middle-tier server requires to store the exchanged data between NIC and accelerator in the host memory. We collect the intermediate buffer lifetime in a commodity cloud middle-tier server (equipped with a 100 GbE NIC), and the average lifetime is around 32 ms. According to Little's law, the intermediate buffer size is around 100Gbps * 32ms = 400MB, which is much more than the LLC capacity of a modern CPU. Therefore, DDIO cannot avoid storing messages in host memory. Meanwhile, DDIO cannot relieve PCIe pressure.

3.3 Naive FPGA-based SmartNIC solution

In recent years, there are many studies on offloading tasks to SmartNICs [7, 14, 17, 21, 28, 44, 46–48, 61, 63, 68]. Figure 1c describes the data flow of the naive FPGA-based SmartNIC solution that offloads both computation-intensive and control logic to SmartNIC. First, network messages are written into SmartNIC's device memory (❶). Second, the hardware logic parses the message header (❷) and invokes hardware engines (❸). Hardware engines fetch message payloads and process them, then write the results into the device memory (❹). At last, hardware logic sends the results out through the network (❺).

Compared to a conventional middle-tier server, the naive FPGA-based SmartNIC solution is cheaper and has lower active power [55]. However, offloading the control logic to hardware sacrifices flexibility, which is unacceptable in the cloud as it provides the cloud with the ability to evolve quickly. From the information of a mainstream cloud vendor, the control logic software has released around 7 major updates over the last 4 months, while the compression computation only releases 2 major updates over the last 2 years.

3.4 SoC-based SmartNIC solution

Current off-the-shelf and upcoming SoC-based SmartNIC (e.g., Nvidia BlueField [58, 59], Broadcom Stingray [13], Intel IPU [35]) features both CPU cores and hardware engines to provide flexibility and computing intensity. Figure 1d illustrates the data flow of a middle-tier server implemented with a typical SoC-based SmartNIC. First, network messages are written into SmartNIC's device memory (❶). Second, the Arm core parses the message header ❷. Third, the off-path accelerator fetches message payloads and processes them, then write the results into the device memory (❸). At last, the Arm core sends the results out through the network (❹). However, today's SoC-based SmartNICs are not well provisioned due to the form factor and power constraints. There are mainly two limitations of SoC-based SmartNIC solution:

First, these SmartNICs have limited compression ability compared with their networking ability. BlueField-2 [58] only delivers ~40 Gbps compression throughput while providing up to 2x100 Gbps networking. Broadcom Stingray PS1100R [13] does not support compression. The upcoming BlueField-3 [59] does not have a compression engine neither. BlueField-3 has a 16-core RISC-V programmable datapath accelerator (PDA) that sits in the place of "Inline Accelerator" in Figure 1d. BlueField-3 engineers say that PDA is not suitable for compression, but can do lightweight computation along the datapath. They suggest performing compression in BlueField-3's 16-core Arm that can only deliver ~50 Gbps compression throughput, which is significantly less than the network bandwidth (400 Gbps).

Second, these SmartNICs have limited memory bandwidth compared with their networking ability. As shown in Figure 1d, SoC-based SmartNIC has two kinds of accelerator: inline accelerator and off-path accelerator. SoC-based SmartNIC is more expert at accelerating inline simple applications using inline accelerators (traffic from the network is directly processed by inline accelerators and then forwarded to the host, without interacting with the Arm subsystem). SoC-based SmartNIC's Arm CPU is usually wimpy and the memory bandwidth is usually designed to match the ability of the Arm CPU. However, the middle-tier application requires network traffic to be temporarily stored in large DRAM, which results in the payload going through device memory four times as shown in Figure 1d. Considering the effect of compression and 3-way replication, this number is around 3.5× in reality. Let's assume that these SoC-based SmartNICs are equipped with a hypothetical compression engine that can consume the network data at line rate. Take BlueField-3 (400 Gbps networking) for example, 400 Gbps write request needs 3.5× memory bandwidth 1400 Gbps. However, BlueField-3 features two 5600MT/s DDR5 channels to provide theoretical 716.8 Gbps memory bandwidth. Its achievable memory bandwidth is around 0.7× the theoretical bandwidth, around 500 Gbps, which is far less than the required bandwidth. BlueField-2,

Stingray PS1100R, and IPU also have similar memory bandwidth shortages. Given the cost, power, and form factor constraints, the limitation of memory bandwidth on SoC-based SmartNICs is not likely to be solved in the near future.

4 DESIGN AND IMPLEMENTATION OF SMARTDS

In addition to the limitations of existing approaches, SmartDS is motivated by two key observations.

- First, network speed grows faster than other relevant components within a host system [62]: CPU [26, 56], memory [50, 76] and the PCIe interconnect [42, 57, 73]. Various products or sub-products from different vendors provide higher network volume than PCIe links [9, 12, 19, 51, 52, 58, 59, 71, 82, 84].
- Second, the I/O size in the middle-tier is relatively large (e.g., 4 KB). The majority of the network message needs heavy computation, while only a small part (e.g., 64 bytes) requires flexible processing.

Considering hundreds of thousands of middle-tier servers deployed in the cloud, it is of utmost importance to design a high-performance middle-tier server architecture to meet the performance and flexibility requirements at the same time. When designing SmartDS, we keep the following three goals in mind.

G1: High throughput and low latency. With the same amount of storage I/O bandwidth, the higher throughput of a middle-tier server means that fewer servers are needed, thus reducing the cloud's total cost of ownership (TCO). Therefore, SmartDS must provide as higher throughput as possible. The explicit goal of SmartDS is to deliver near-peak network bandwidth even on SmartNICs with multiple networking ports. Over the past few years, there has been a significant reduction in network and storage device latencies within the datacenter [25, 29, 53, 69]. Our design must provide possibly lower latency when serving storage I/O requests.

G2: High Flexibility. As we discussed in Section 3.3, flexibility is one of the most important factors of the cloud to enable various strategies and to meet the requirements of different types of workloads at a low cost.

G3: High Programmability. SmartDS must be easy to use and allow developers to focus solely on implementing the high-level functionalities, rather than low-level details regarding SmartNIC and RDMA mechanism. The high programmability also enables fast upgrades of high-level software in the cloud.

To achieve the above three goals, we propose SmartDS, a middle-tier-centric SmartNIC that can serve storage I/O requests with low latency and high throughput, while maintaining high flexibility and programmability. The key idea behind SmartDS is a SmartNIC-based application-aware message split (AAMS) mechanism that allows processing the message's header on the host CPU, so as to achieve high flexibility (**G2**), and to process the message's payload on the SmartNIC, so as to achieve high throughput (**G1**). The header/payload split mechanism has been studied in several prior works to accelerate data mover applications [27, 62, 67]. CacheDirector [20] and IDIO [3] split packets' headers and payloads to better utilize DDIO. SmartDS is the first to process different parts of a network message with different computing resources, thus achieving lower costs and minimizing resource interference.

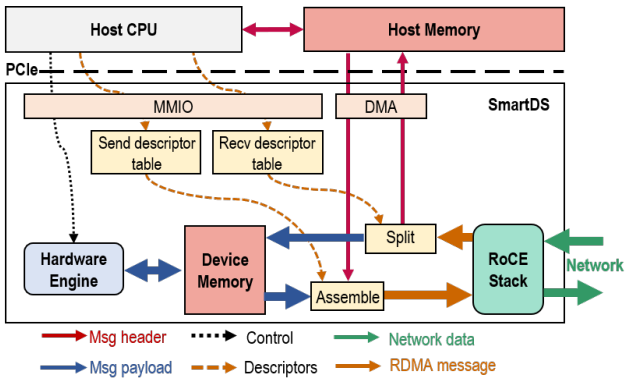


Figure 5: Architecture of SmartDS that enables application-aware message-split mechanism

In the following, we present the detailed design of AAMS (Subsection 4.1), followed by the multiple port design (Subsection 4.2) and a running example with SmartDS (Subsection 4.3).

4.1 Application-aware Message Split Mechanism

Figure 5 demonstrates the overall architecture of the middle-tier server equipped with SmartDS. SmartDS connects the host through PCIe, and SmartDS consists of a hardware engine, a large off-chip device memory, and a RoCE network transport.

Network Module. The network stack used in SmartDS is a modified version of earlier work [18, 70], which implements a RoCE stack on FPGAs. The changes introduced in this paper aim to implement application-aware message split on RDMA messages. In addition to accessing host memory using one-sided and two-sided RDMA verbs, the key difference of our RoCE stack is to directly access SmartDS's large device memory and to allow a single RDMA message to span host memory and SmartDS's device memory according to the descriptors posted by the software (Subsection 4.3).

On the receive side, the application posts recv descriptors to the *Recv descriptor table* associated with the *Split* module. When an RDMA message from the *RoCE* module arrives in the *Split* module, it will find the corresponding descriptor in the *Recv descriptor table*. The descriptor contains a host memory address h_buf, a host memory buffer size h_size, a device memory address d_buf, a device memory buffer size d_size and information of the queue pair (e.g., queue id). The *Split* module would write the first h_size bytes of the RDMA message into h_buf, and write the remaining bytes of the RDMA message into d_buf. At last, it notifies the host CPU.

On the send side, the application posts send descriptors to the *Send descriptor table* associated with the *Assemble* module. Similarly, the descriptor contains a host memory address h_buf, a host memory buffer size h_size, a device memory address d_buf, a device memory buffer size d_size and information of the queue pair. The *Assemble* module would read h_size bytes from h_buf and read d_size bytes from d_buf. When data is returned by the *DMA* module and device memory, the *Assemble* module uses the returned data to make up an RDMA message using the information

in the descriptor. The *Assemble* module sends the message to the *RoCE* stack, which would send the message to the network. Finally, it notifies the host CPU.

Offloaded hardware engine. The software can explicitly invoke a hardware engine in SmartDS to process message payload residing in SmartDS's device memory. To simplify the programming model, the engine follows a simple I/O mechanism. The engine fetches data from the device memory and writes results to the device memory. In the scenario of the middle-tier server, we implement LZ4 compression [8] as it is the most computation-intensive operation in the middle-tier server. Nevertheless, SmartDS provides a simple interface to deploy different hardware engines according to the application scenario.

Benefits. In our design, only a small portion of the network message, i.e., the message header, would be forwarded to the CPU, thus reducing much PCIe bandwidth and memory bandwidth occupation. In particular, there are mainly two benefits:

- Reduced memory bandwidth occupation indicates that the performance of serving I/O requests would be less affected by other maintenance services in the middle-tier server. We could achieve performance isolation without partitioning memory bandwidth.
- Reduced PCIe bandwidth occupation indicates that we could easily scale up SmartDS with much more network bandwidth than PCIe bandwidth.

4.2 Extending to Multiple Networking Ports

There is a wide variety of NIC/SmartNIC products from different vendors that provide more network volume than PCIe link bandwidth. The CPU-based and the accelerator-enhanced middle-tier servers can not benefit from this asymmetric network/PCIe bandwidth, because all network messages have to travel through PCIe whose bandwidth becomes a bottleneck. Therefore, the remaining networking ports can only be used as a backup link to improve reliability.

In SmartDS, all networking ports can be fully leveraged, because SmartDS only forwards the message header to host memory through PCIe, while the majority of the network message stays in SmartDS's device memory, leaving the PCIe link under-loaded. All the networking ports can be utilized as long as the SmartDS's device memory bandwidth is enough. Fortunately, many FPGA-based SmartNICs (with multiple networking ports) have a large off-chip high-bandwidth memory. For example, our prototype is built with Xilinx VCU128 FPGA [83], which has a 6x100 Gbps networking volume and an 8 GB High Bandwidth Memory (HBM). Its HBM has 16 memory channels, providing up to 3.4 Tbps memory bandwidth [79].

Figure 6 demonstrates the overall architecture of SmartDS which is able to take advantage of multiple networking ports. For each networking port, SmartDS instantiates an extended RoCE stack instance, configured with an individual IP address. The extended RoCE stack includes the RoCE stack in prior work [70], *Split* module, and *Assemble* module mentioned in Subsection 4.1. At the same time, a hardware engine is instantiated to process the message payload from the extended ROCE stack. As such, SmartDS is able to concurrently provide real-time services coming from multiple networking ports. This design enables SmartDS fully utilize the asymmetric bandwidth of PCIe and network.

4.3 A Running Example with SmartDS

In order to allow programmers to easily harvest the performance potential of SmartDS, SmartDS exposes RDMA-like high-level application programming interface (API) to guarantee high programmability. Table 2 demonstrates the APIs of SmartDS. Listing 1 demonstrates a simplified example using SmartDS to serve write requests in a disaggregated block storage system:

Memory allocation. The software first allocates both host memory for headers (Lines 2-3) and SmartDS's device memory for payloads (Lines 4-5).

Connect queue pair. After memory allocation, the software calls function *open_roce_instance* to get the context of the first RoCE network instance in SmartDS (Line 8). Then it connects one queue pair with a remote VM (Line 10) and the other queue pair with a remote storage server (Line 11). The code here is simplified for better understanding.

Receive message. The software posts a recv work descriptor to the SmartDS (Line 15) and polls its completion. After receiving the write request, the software can know the received payload size through the completion event *e*.

Flexible and changeful processing. The software parses the message header in the host memory *h_buf_recv* (Line 20). Then the software prepares *h_buf_send* according to the parsed results (Line

Table 2: High-level APIs of SmartDS

host_alloc(size)
Allocating *size* bytes buffer in the host memory.
dev_alloc(size)
Allocating *size* bytes buffer in the SmartDS's device memory.
open_roce_instance(instance_index)
Open one of the RoCE instances in the SmartDS and return the context.
dev_mixed_recv(qp, h_buf, h_size, d_buf, d_size)
Post a recv work request, the received RDMA message would be split, the first *h_size* bytes would be stored in host memory *h_buf*, while the remaining would be stored in SmartDS's device memory *d_buf*. It would return an asynchronous event.
dev_mixed_send(qp, h_buf, h_size, d_buf, d_size)
Post a send work request, the SmartDS would assemble the *h_size* bytes in host memory *h_buf* and the *d_size* bytes in the SmartDS's device memory *d_buf* into a RDMA message. It would return an asynchronous event.
dev_func(src, src_size, dest, dest_size, engine)
Invoke the engine to do the offloaded computation, the engine would fetch *src_size* bytes from *src* in SmartDS's device memory. When the engine completes, it writes the result into *dest* in SmartDS's device memory and notifies the application running in the host CPU. The *engine* parameter specifies which engine to use. It would return an asynchronous event.
poll(event)
Poll the asynchronous event until it completes.

21). Note that this stage only involves the host CPU and host memory, so as to guarantee high flexibility and high programmability.

Offloaded computation. If the write request is sensitive to latency, the software directly forwards the data block to a remote storage server and polls its completion (Lines 25-27). Otherwise, the software can invoke a hardware engine in SmartDS to perform compression on the data block in *d_buf_recv* and poll its completion (Lines 30-31). After compression, the software sends the compressed results to a remote storage server and polls its completion (Lines 33-35).

How to guarantee high flexibility and high throughput? In SmartDS, the message header that requires flexible or changeful logic remains in host memory and is processed by the host CPU, we do not offload any control logic to SmartDS. The message payload that needs fixed heavy computations (e.g., compression) remains in SmartDS's device memory. We only build hardware engines in SmartDS to perform fixed heavy computations that hardly change over time on payloads. As such, SmartDS enables the fast upgrades of disaggregated block storage software while keeping high throughput.

How to guarantee high programmability? Programming with SmartDS uses high-level APIs and the SmartDS implementation only requires 145 lines of code, while the baseline uses standard RDMA NIC and LZ4 library [49] and its core functionalities require 130 lines of code. Therefore, SmartDS provides high programmability to allow developers to only focus on high-level functionalities.

```
1  /* Allocating host and device memory buffers*/
2  void* h_buf_recv = host_alloc(MAX_SIZE);
3  void* h_buf_send = host_alloc(MAX_SIZE);
4  void* d_buf_recv = dev_alloc(MAX_SIZE);
5  void* d_buf_send = dev_alloc(MAX_SIZE);
6
7  /*Open Open RoCE instance 0*/
8  ctx = open_roce_instance(INSATANCE_0);
9  /*Connect queue pairs with remote client and storage server*/
10 qp_recv = connect_qp(ctx,remote_VM);
11 qp_send = connect_qp(ctx,remote_storage_server);
12
13 while(true){
14     /* Recv a write request from a client, forward its header to
           host memory, keep the payload in SmartNIC's memory */
15     e = dev_mixed_recv (qp_recv, h_buf_recv, HEAD_SIZE,
           d_buf_recv, MAX_SIZE);
16     poll(e);
17     payload_size = e.size;
18
19     /* User's logic flexibly parses the content in h_buf_recv and
          prepares the necessary send header*/
20     parsed_res = host_parse_header(h_buf_recv, HEAD_SIZE);
21     host_fill_send_h_buf(h_buf_send, parsed_res);
22
23
24     /*Directly send a latency-sensitive block to a storage server
           */
25     if(parsed_res.is_latency_important === true){
26         e = dev_mixed_send(qp_send, h_buf_send, HEAD_SIZE,
               d_buf_recv, payload_size);
27         poll(e);
28     }else{ /*for a block that is not latency-sensitive*/
29         /* compress a data block via hardware engine 0*/
30         e = dev_func(src=d_buf_recv, payload_size, dest=
               d_buf_send, MAX_SIZE, engine=COMPRESS_ENGINE_0);
31         poll(e);
32         /* Send the compressed block to a remote storage server*/
33         compressed_size = e.size;
34         e = dev_mixed_send(qp_send, h_buf_send, HEAD_SIZE,
               d_buf_send, compressed_size);
35         poll(e);
36     }
37 }
```

Listing 1: An example of programming with SmartDS

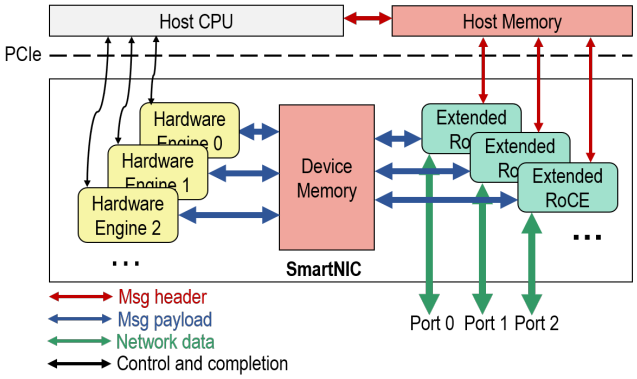


Figure 6: SmartDS Architecture with multiple network ports

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

Our experimental platform consists of four AMAX XP04A201G servers. Each has two 24-logical-core (12 physical cores with 2-way SMT) 2.2 GHz Xeon Silver 4214 CPUs, 256 GiB (8x32 GiB) 2400 MHz DDR4 memory, and a 16 MiB LLC. Each server is equipped with a Mellanox ConnectX-5 NIC. We implement our prototype on the Xilinx VCU128 FPGA [83], which is plugged into one of the above servers.

Workloads. We carry out our experiments on Silesia compression corpus [75], a well-known data set that provides a data set of files that covers the typical data types used nowadays. We build a typical scenario of disaggregated block storage. One server keeps issuing write requests to a middle-tier server. After receiving the write request, the middle-tier server compresses the data block in the request and sends it to three storage servers, which would store the compressed data on the disk. When the middle-tier server receives successes from storage servers, it replies a success to the server that issues the request. After that, the write request completes.

CPU-based middle-tier server baseline. Our CPU-based middle-tier server, "CPU-only", is equipped with a Mellanox ConnectX-5 NIC. The network stack is RDMA and we use LZ4 library [49] for compression.

Accelerator-enhanced middle-tier server baseline. The baseline, "Acc", is equipped with a Mellanox ConnectX-5 NIC and Xilinx Alveo U280 FPGA card [82]. When the host CPU receives an RDMA message from the client, it invokes compression engines on the FPGA to read the message payload by DMA. The compression engines then write the results back and notify the host CPU. Then the host CPU sends it to three remote storage servers. After receiving replies from remote storage servers, it sends a success to the client. The engine's compression throughput can be up to 100 Gbps.

SoC-based SmartNIC baseline. We use Nvidia BlueField-2 SmartNIC to implement SoC-based SmartNIC baseline "BF2". It has eight Arm A72 cores and two 100 Gbps networking ports. We perform the compression on the compression engine of BlueField-2.

SmartDS implementations. We present four implementations of SmartDS on a Xilinx VCU128 FPGA: "SmartDS-1", "SmartDS-2",

Table 3: FPGA resource consumption

Name	LUTs (K)	REGS (K)	BRAMs
"Acc"	112 (8.6%)	109 (4.2%)	172 (8.5%)
"SmartDS-1"	157 (12.0%)	143 (5.4%)	292 (14.5%)
"SmartDS-2"	313 (24.0%)	285 (10.9%)	584 (29.0%)
"SmartDS-4"	627 (48.1%)	571 (21.9%)	1168 (58.0%)
"SmartDS-6"	941 (72.2%)	857 (32.9%)	1752 (86.9%)

"SmartDS-4", "SmartDS-6".[1] The number indicates the number of utilized networking ports. Each utilized networking port is equiped with a compression engine and a RoCE instance. Each compression engine can process 4 KB data blocks at the rate of 100 Gbps. Table 3 shows the FPGA resource consumption of accelerator-enhanced middle-tier server "Acc" and four implementations of SmartDS.

5.2 Effect of Application-aware Message Split Mechanism

We examine the effect of our proposed application-aware message split mechanism, by comparing the throughput and latency of serving write requests.

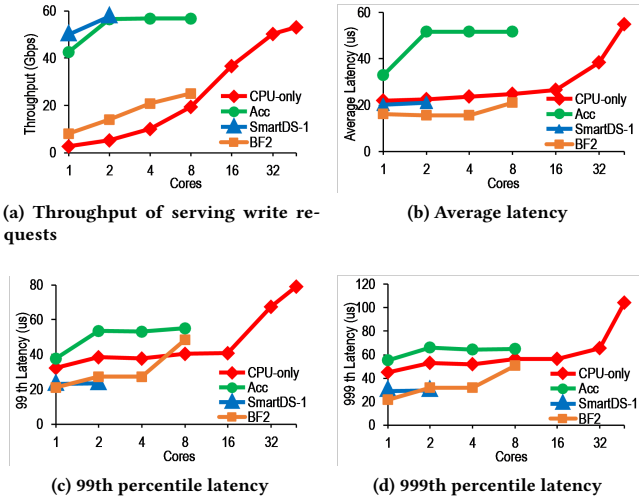
Throughput. Figure 7a demonstrates the achieved throughput of different middle-tier server implementations. "SmartDS-1" and "Acc" only require two threads to reach the peak throughput, while "CPU-only" requires nearly all 48 logical cores in the server to reach the same throughput because CPU compression is slow (~2.1 Gbps for one logical core, ~2.7 Gbps for two logical cores that belong to the same hardware core), while "Acc" and "SmartDS-1" use FPGA to compress data blocks and "BF2" uses compression engine to do compression.

Latency. Figure 7b-d shows the average latency, 99th percentile latency and 999th percentile latency of different approaches. "Acc" has the highest average latency when not overloaded due to two reasons. First, despite the data movement of NIC, "Acc" requires two extra data block movements (uncompressed data block from host memory to FPGA and compressed data from FPGA to host memory). Second, "Acc" performs the compression on the FPGA, and its processing latency is higher than the CPU due to its significantly lower frequency. "SmartDS-1" has nearly the same latency as "CPU-only", the difference is that "CPU-only" latency would dramatically increase when more CPU cores are used. "BF2" has the lowest average latency since it does not need to communicate with the host. But its 99th and 999th percentile latency is higher than SmartDS-1 when more than one CPU core is used.

We corroborate our hypothesis by measuring the host memory usage and PCIe usage of different approaches.

Memory bandwidth. Figure 8a shows the host memory bandwidth occupation of different approaches. SmartDS hardly occupies host memory bandwidth, as only the message header of the entire message goes to host memory, the data block that accounts for the majority of the message stays in SmartDS's memory, demonstrating the great potential of scaling up SmartDS within a middle-tier

[1]There are only four 100 Gbps networking ports in our FPGA, but it can have two more 100Gbps networking ports with a 2-port QSFP28 (Quad Small Form-factor Pluggable) FMC Module [30]. We do have this module, but ours is broken and can not be detected. So we simulate the performance of "SmartDS-6" using the results of the former three implementations.



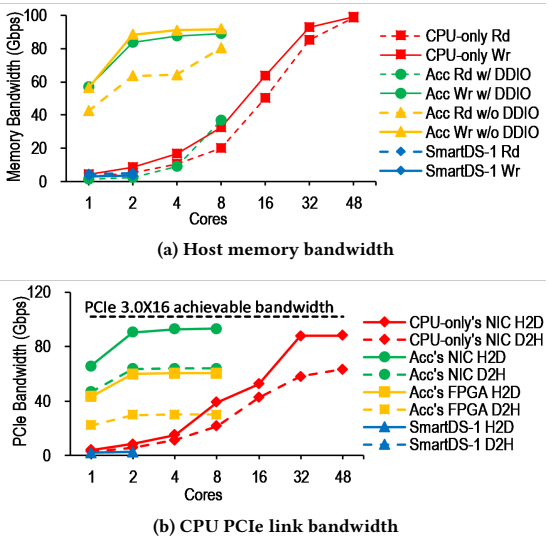
(a) Throughput of serving write requests

(b) Average latency

(c) 99th percentile latency

(d) 999th percentile latency

Figure 7: Throughput and latency of different approaches



(a) Host memory bandwidth

(b) CPU PCIe link bandwidth

Figure 8: Comparison of host memory and CPU PCIe link bandwidth usage between approaches

server. "CPU-only" consumes nearly the same memory read bandwidth and memory write bandwidth, and the occupied bandwidth increases when using more cores. When using more cores, "Acc w/DDIO" occupies more memory write bandwidth, but it hardly consumes memory read bandwidth, because Intel DDIO [32] is enabled in our server. When network data arrives, it would go to LLC (last level cache), then FPGA can directly read the data from LLC instead of the host memory. When FPGA completes compressing the data block, it writes the result into LLC, then NIC can directly read the result from LLC and send it to the network. We verify this by turning off DDIO and repeating the experiment. As expected, the memory read bandwidth (yellow line marked with w/o DDIO) significantly increases when DDIO is disabled.

PCIe bandwidth. Figure 8b shows the PCIe bandwidth occupation comparison of different approaches. D2H indicates the direction

from device to host while H2D indicates from host to device. "CPU-only" only uses one PCIe device (i.e., NIC). When more cores are used, both D2H and H2D PCIe bandwidth increase. H2D PCIe bandwidth nearly reaches the achievable bandwidth of PCIe 3.0X16. "Acc" employs two PCIe devices (NIC and FPGA). When running at peak performance, NIC's H2D bandwidth reaches the achievable bandwidth of PCIe 3.0X16, and FPGA's H2D bandwidth reaches ~70% of the achievable bandwidth. "SmartDS-1" only employs one PCIe device (i.e., FPGA), and occupies only ~2% of PCIe bandwidth because the majority of the message, i.e., message payload, is processed on SmartDS.

5.3 Performance Interference

In this subsection, we validate that SmartDS can achieve performance isolation without partitioning memory bandwidth and caches on the CPU.

As mentioned in Section 2, despite serving I/O requests from VMs, each middle-tier server runs maintenance services. Therefore, serving I/O requests and running maintenance services result in performance inference in each middle-tier server. To achieve performance isolation, developers partition not only CPU cores but also memory bandwidth and caches [24]. To demonstrate the interference from the memory subsystem, we use 16 dedicated cores to run the Intel memory latency checker (MLC) to inject memory requests into the memory subsystem. The remaining cores are dedicated to serving I/O requests. We change the duration (clock cycles) between injected memory requests of Intel MLC to simulate different levels of memory pressure.

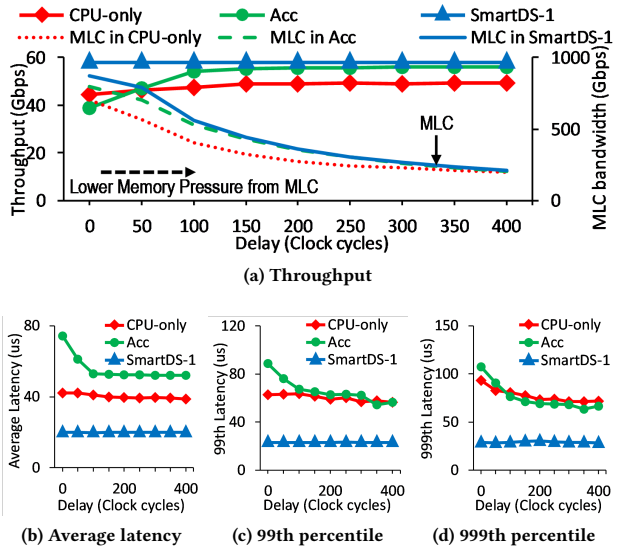
Figure 9 shows the achieved performance under different memory pressure from other cores. "CPU-only" and "Acc" achieve significantly lower throughput with higher memory pressure, while the throughput of "SmartDS-1" hardly changes over different levels of memory pressure. And the MLC bandwidth of "SmartDS-1" is always higher than the other two implementations. The average latency (Figure 9b) and tail latency (Figure 9c, Figure 9d) show similar results: the average latency and tail latency of "SmartDS-1" would not be interfered by memory bandwidth pressure from other cores, while "CPU-only" and "Acc" suffer from severe memory interference, indicating that "SmartDS-1" can achieve performance isolation without partitioning host memory bandwidth and caches.

5.4 Effect of Multiple Networking Ports

We validate that SmartDS can fully utilize more networking ports. Figure 5.5 illustrates the throughput and latency trend when employing different numbers of networking ports. We observe that SmartDS can linearly increase the achieved throughput with an increasing number of networking ports, as shown in Figure 10a. For example, SmartDS-4 reaches 4× achieved throughput of the maximum throughput of SmartDS-1, "CPU-only" or "Acc". Figure 10a shows that the average latency, 99th percentile latency, and 999th percentile latency of SmartDS roughly keep the same with an increasing number of networking ports.

5.5 Multiple SmartNICs per Server

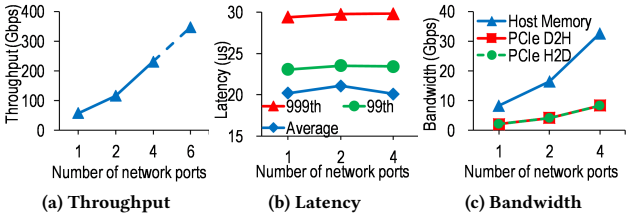
SmartDS can scale up linearly if the server has enough CPU cores (two cores per networking port) since it does not bring much pressure to the host memory and PCIe. We estimate that SmartDS-6



(a) Throughput

(b) Average latency (c) 99th percentile (d) 999th percentile

Figure 9: Performance under different memory pressure



(a) Throughput (b) Latency (c) Bandwidth

Figure 10: Effect of the number of network ports

consumes the host memory bandwidth (49 Gbps) and PCIe memory bandwidth (12.4 Gbps) to consume 348Gbps storage traffic. Therefore, it's practical to equip a middle-tier server with many SmartNICs to reduce the total cost of ownership when the number of CPU cores is enough.

Taking the platform we have used in the prior experiments as an example, it has two 1x4 PCIe 3.0x16 switches, and each PCIe switch can be equipped with four PCIe 3.0x16 devices. We can equip this server with 8 SmartDSs. We can achieve up to 2.8 Tbps throughput (51.6× higher than "CPU-only"), while the host memory occupation is only 392 Gbps, far less than the total host memory bandwidth (theoretically 1228 Gbps from eight memory channels). The root port of each PCIe switch only consumes 49.6 Gbps (4*12.4 Gbps) PCIe bandwidth, which is far less than the achievable bandwidth (102.4 Gbps) of the PCIe 3.0x16 root port.

6 RELATED WORK

To our knowledge, SmartDS is the first middle-tier-centric SmartNIC that serves storage I/O requests with high throughput, while keeping high flexibility and high programmability.

Header-payload split. Header-payload split has been studied in many of the prior works [3, 20, 27, 62, 67].

The first class is relevant to DDIO. CacheDirector [20] steers the header of each network packet into the LLC tile that is closest to the core that will process the packet. IDIO [3] selectively disables Direct Cache Access (DCA) for the payload of received packets according to application classes while always keeping DCA enabled for packet headers. In contrast, based on the observation that many SmartNIC products provide much more network bandwidth than the PCIe link, SmartDS forwards the message header to the host while leaving message payloads in the SmartNIC's device memory, and then invokes SmartNIC hardware engines to process the payloads. It's also possible to apply CacheDirector to SmartDS to put message headers into the closest LLC tile, thus reducing the latency of middle-tier server applications.

The second class forwards packet headers to host memory, while payloads remain in NIC memory [62, 67], or network switch memory [27]. They can only accelerate data mover applications where network payloads are unchanged such as shallow network functions. Besides, they do not provide reliability and their split is performed at the granularity of the packet. In contrast, SmartDS applies to the applications that require computation on payloads, and SmartDS provides reliability and performs our split at the granularity of RDMA message.

The third class forwards the header and data of the received Ethernet frames to different host buffers [54]. As such, the headers occupy a smaller memory region to let more headers fit into a single memory page and even the system cache, reducing the driver stack overhead of memory accesses.

SmartNIC-enhanced Storage Disaggregation. In recent years, SmartNIC-based disaggregated storage [13, 55, 58, 59] has emerged and become increasingly popular because of their low deployment costs and competitive IO performance compared to traditional server-based approaches. SmartNIC-based storage node usually comprises a SmartNIC, a PCIe switch, and a collection of NVMe SSDs. LineFS [40] proposes the persist-and-publish model and pipeline parallelism to offload a PM-optimized distribute file system (DFS) to SmartNICs. It offloads CPU-intensive DFS tasks, like replication, compression, data publication, index management, and consistency management to an SoC-based SmartNIC. It sits in the compute node and offloads many control functions to SmartNIC. The data compression in LineFS is performed in the SmartNIC's CPU, thus its achieved throughput is very low (~10 Gbps). SmartDS is orthogonal to these works, we focus on optimizing the middle tier (i.e., middle-tier servers) instead of the storage node. SmartDS only offload the most computation-intensive operations to SmartNIC.

Offloading to Hardware. Many prior works [5, 7, 17, 21, 28, 37, 39, 44–48, 61, 63, 80, 81] offload host tasks to FPGA-based or SoC-based SmartNICs. For FPGA-based SmartNIC, ClickNP [44] offloads network functions such as firewall, L4 load balancer, and IPSec gateway. AccelNet [21] offloads TCP processing and SDN stack. Tonic [7] enables programmable transport protocols. hXDP [14] offloads eBPF processing. PANIC [46] addresses the performance isolation and fairness problems under the multi-tenant environment. FlowBlaze [63] enables stateful network packet processing on FPGAs. FpgaNIC [80] allows data/control plane offloading for efficient GPU-SmartNIC co-processing. For SoC-based SmartNIC, Floem [61] and iPipe [47] offload applications like key-value store, real-time data analytics to SmartNIC. E3 [48] and λ-NIC [17] offload microservices to SmartNIC. FairNIC [28] tackles performance isolation and fairness problems. Xenic offloads distributed transactions to SmartNIC. These works leverage SmartNIC to alleviate CPU pressure but do not tackle the problem in the middle-tier server, offloading the most computation-intensive compression efficiently while keeping flexibility and programmability. Prior works [38, 43, 64, 65] accelerate RPCs using specialized hardware accelerators or programmable NICs. They mainly focus on data transformation in RPCs (i.e., serialization/deserialization). RAMBDA [85] leverages cache-coherent accelerators and commodity RDMA NICs to offload datacenter applications, but suffers from the high memory bandwidth occupation problem.

Other Novel Network Architecture. Flajslik et at. [22] study different sources of latency overhead in the network stack and find that minimizing the number of PCIe transactions is vital. They propose NIQ to reduce communication latency by leveraging techniques such as packet inline, custom polling, and creative use of caching policies. NetDIMM [4] integrates a full-blown NIC into the buffer device of a DIMM. It reduces the amount of data movement when processing network packets by leveraging in-memory buffer cloning. It can provide the performance of zero-copy networking without its drawbacks. SmartDS focus on middle-tier server applications. It reduces the amount of data movement by only transferring the message header between host memory and SmartNIC, it leverages hardware engines in SmartNIC to process network-intensive message payloads.

7 CONCLUSION

The cloud introduces a middle-tier, in terms of middle-tier servers, between compute servers and storage servers to serve I/O requests from compute servers and to provide computations such as compression and decompression. The existing CPU-based middle-tier server is bottlenecked by compute-intensive compression/decompression. To this end, we present SmartDS, a middle-tier-centric SmartNIC that serves storage I/O requests with low latency and high throughput, while keeping high flexibility and high programmability. The key idea of SmartDS is an application-aware message split mechanism that enables the processing of message headers on the host CPU, providing flexibility, while the message payload is processed on the SmartDS, achieving high throughput. SmartDS enables linear scale-up of multiple network ports and multiple SmartNICs, such that SmartDS reduces the number of middle-tier servers by roughly 51.6×, and thus significantly reduces cloud infrastructure cost for disaggregated block storage.

Acknowledgements. We would like to thank Xueying Zhu and Xuzheng Chen for supporting this work. Also, we thank the anonymous reviewers for their detailed feedback. We are grateful to the AMD-Xilinx University Program for the donation of some of the AMD-Xilinx FPGAs used in the experiments. The work is supported by the following grants: the Program of Zhejiang Province Science and Technology (2022C01044), a research grant from Alibaba Group through Alibaba Innovative Research (AIR) Program, the Fundamental Research Funds for the Central Universities 226-2022-00151, Key Laboratory for Corneal Diseases Research of Zhejiang Province. Zeke Wang is the corresponding author.

REFERENCES

[1] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *IWOCL*, 2014.

[2] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution," in *SOSP*, 2019.

[3] M. Alian, S. Agarwal, J. Shin, N. Patel, Y. Yuan, D. Kim, R. Wang, and N. S. Kim, "Idio: Network-driven, inbound network data orchestration on server processors," in *MICRO*, 2022.

[4] M. Alian and N. S. Kim, "Netdimm: Low-latency near-memory network interface architecture," in *MICRO*, 2019.

[5] G. Alonso, "Technical perspective: Dfi: The data flow interface for high-speed networks," *SIGMOD Rec.*, 2022.

[6] Amazon, "Amazon Elastic Block Store," https://aws.amazon.com/cn/blogs/architecture/category/storage/amazon-elastic-block-storage-ebs, 2022.

[7] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, "Enabling programmable transport protocols in high-speed nics," in *NSDI*, 2020.

[8] M. Bartík, S. Ubik, and P. Kubalik, "Lz4 compression algorithm on fpga," in *ICECS*, 2015.

[9] Broadcom, "Stingray™ PS250," https://docs.broadcom.com/doc/PS250-PB, 2018.

[10] Broadcom, "BCM957508-P2200G," https://docs.broadcom.com/doc/957508-P2200G-DS, 2019.

[11] Broadcom, "BCM957504-N1100G," https://docs.broadcom.com/doc/957504-N1100G-DS, 2020.

[12] Broadcom, "Broadcom N2200G," https://www.broadcom.com/products/ethernet-connectivity/network-adapters/n2200g, 2022.

[13] Broadcom, "Broadcom Stingray PS1100R," https://docs.broadcom.com/doc/PS1100R-PB, 2022.

[14] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hxdp: Efficient software packet processing on fpga nics," *Communications of the ACM*, 2022.

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

[16] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress compute vs. io tradeoffs for mapreduce energy efficiency," in *SIGCOMM*, 2010.

[17] S. Choi, M. Shahbaz, B. Prabhakar, and M. Rosenblum, "λ-nic: Interactive serverless compute on programmable smartnics," in *ICDCS*, 2020.

[18] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research," in *ASPLOS*, 2022.

[19] Ehernet Technology Consortium, "800G specification," https://ethernettechnologyconsortium.org/wpcontent/uploads/2020/03/800G-Specification_r1.0.pdf, 2020.

[20] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *EuroSys*, 2019.

[21] 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 *NSDI*, 2018.

[22] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *ATC*, 2013.

[23] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *FCCM*, 2015.

[24] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *OSDI*, 2020.

[25] 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 *NSDI*, 2021.

[26] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing gpu as packet processing accelerator," in *NSDI*, 2017.

[27] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. Seltzer, "Parking packet payload with p4," in *CoNEXT*, 2020.

[28] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren, "Smartnic performance isolation with fairnic: Programmable networking for the cloud," in *SIGCOMM*, 2020.

[29] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *SIGCOMM*, 2016.

[30] HiTech Global, "2-Port QSFP28 (2x100G) / QSFP+ (2x40G or 2x56G) FMC Module (Vita57.1)," http://www.hitechglobal.com/FMCModules/FMC_2QSFP28.htm, 2022.

[31] X. Hu, F. Wang, W. Li, J. Li, and H. Guan, "Qzfs: Qat accelerated compression in file system for application agnostic and cost efficient data storage," in *ATC*, 2019.

[32] Intel, "Intel data direct i/o technology: A primer," https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012.

[33] Intel, "Intel QuickAssist Technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html, 2019.

[34] Intel, "Intel® SSD D7-P5520 Series," https://ark.intel.com/content/www/us/en/ark/products/213416/intel-ssd-d7p5520-series-1-92tb-2-5in-pcie-4-0-x4-3d4-tlc.html, 2020.

[35] Intel, "Intel® Infrastructure Processing Unit," https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2022.

[36] Intel, "Intel® Memory Latency Checker," https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2022.

[37] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a Box: Inexpensive Coordination in Hardware," in *NSDI*, 2016.

[38] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee, "A specialized architecture for object serialization with applications to big data analytics," in *ISCA*, 2020.

[39] M. Khazraee, A. Forencich, G. C. Papen, A. C. Snoeren, and A. Schulman, "Rosebud: Making FPGA-Accelerated Middlebox Development More Pleasant," in *ASPLOS*, 2023.

[40] 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 *SOSP*, 2021.

[41] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *EuroSys*, 2016.

[42] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *ASPLOS*, 2021.

[43] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *ASPLOS*, 2021.

[44] B. Li, K. Tan, 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 *SIGCOMM*, 2016.

[45] J. Li, Y. Lu, Q. Wang, J. Lin, Z. Yang, and J. Shu, "AlNiCo: SmartNIC-accelerated contention-aware request scheduling for transaction processing," in *ATC*, 2022.

[46] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "Panic: A high-performance programmable nic for multi-tenant networks," in *OSDI*, 2020.

[47] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartnics using ipipe," in *SIGCOMM*, 2019.

[48] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3:energy-efficient microservices on smartnic-accelerated servers," in *ATC*, 2019.

[49] LZ4, "LZ4 Benchmarks," https://github.com/lz4/lz4, 2022.

[50] J. D. McCalpin, "Memory bandwidth and system balance in hpc systems," *UT Faculty/Researcher Works*, 2016.

[51] Mellanox, "ConnectX®-5 En Card Product Brief," https://www.mellanox.com/sites/default/files/relateddocs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf, 2017.

[52] Mellanox, "ConnectX®-6 En Card Product Brief," https://www.mellanox.com/sites/default/files/relateddocs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf, 2017.

[53] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, R. Liu, C. Shi, B. Fu, J. Zhu, J. Wu, D. Cai, and H. H. Liu, "From luna to solar: The evolutions of the compute-to-storage networks in alibaba cloud," in *SIGCOMM*, 2022.

[54] Microsoft, "Introduction to Header-Data Split," https://learn.microsoft.com/en-us/windows-hardware/drivers/network/header-data-split, 2021.

[55] J. Min, M. Liu, T. Chugh, C. Zhao, A. Wei, I. H. Doh, and A. Krishnamurthy, "Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs," in *SIGCOMM*, 2021.

[56] A. Mirhosseini, H. Golestani, and T. F. Wenisch, "Hyperplane: A scalable low-latency notification accelerator for software data planes," in *MICRO*, 2020.

[57] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *SIGCOMM*, 2018.

[58] Nvidia, "NVIDIA BLUEFIELD-2 DPU," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2021.

[59] Nvidia, "NVIDIA BLUEFIELD-3 DPU," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2022.

[60] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined parallel lzss for streaming data compression on gpgpus," in *ICPADS*, 2012.

[61] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A programming system for nic-accelerated network applications," in

OSDI, 2018.
[62] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The benefits of general-purpose on-nic memory," in *ASPLOS*, 2022.
[63] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Bianchi, "Flowblaze: Stateful packet processing in hardware," in *NSDI*, 2019.
[64] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *ASPLOS*, 2020.
[65] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the rpc tax in datacenters," in *MICRO*, 2021.
[66] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled cpu-fpga platforms," in *FCCM*, 2018.
[67] A. Sarma, H. Seyedroudbari, H. Gupta, U. Ramachandran, and A. Daglis, "Nfslicer: Data movement optimization for shallow network functions," *arXiv preprint arXiv:2203.02585*, 2022.
[68] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: Smartnic-accelerated distributed transactions," in *ASPLOS*, 2021.
[69] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag, "A cloud-optimized transport protocol for elastic and scalable hpc," *IEEE Micro*, 2020.
[70] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, "Strom: smart remote memory," in *EuroSys*, 2020.
[71] Silicom, "Silicom FPGA SmartNIC N501x," https://www.silicom.dk/wp-content/uploads/2022/03/Silicom-FPGA-SmartNIC-N501x-Series_v1.0.pdf, 2022.
[72] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *ICPP*, 2016.
[73] I. Smolyar, A. Markuze, B. Pismenny, H. Eran, G. Zellweger, A. Bolen, L. Liss, A. Morrison, and D. Tsafrir, "Ioctopus: Outsmarting nonuniform dma," in *ASPLOS*, 2020.
[74] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The nebula rpc-optimized architecture," in *ISCA*, 2020.
[75] The Silesia corpus, "," https://sun.aei.polsl.pl//~sdeor/index.php, 2022.
[76] S. Thomas, G. M. Voelker, and G. Porter, "Cachecloud: Towards speed-of-light datacenter communication," in *HotCloud*, 2018.
[77] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *NSDI*, 2018.
[78] M. Vemmou, A. Cho, and A. Daglis, "Patching up network data leaks with sweeper," in *MICRO*, 2022.
[79] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking high bandwidth memory on fpgas," in *FCCM*, 2020.
[80] Wang, Zeke and Huang, Hongjing and Zhang, Jie and Wu, Fei and Alonso, Gustavo, "FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs," in *ATC*, 2022.
[81] J. Wirth, J. A. Hofmann, L. Thostrup, C. Binnig, and A. Koch, "Scalable and Flexible High-Performance In-Network Processing of Hash Joins in Distributed Databases," in *FPT*, 2021.
[82] Xilinx, "Xilinx ALVEO™ U280," https://www.xilinx.com/publications/product-briefs/alveo-u280-product-brief.pdf, 2021.
[83] Xilinx, "Virtex UltraScale+ HBM VCU128 FPGA Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/vcu128.html, 2022.
[84] Xilinx, "Xilinx Versal FPGA," https://www.xilinx.com/products/silicon-devices/acap/versal-hbm.html, 2022.
[85] Y. Yuan, J. Huang, Y. Sun, T. Wang, J. Nelson, D. R. Ports, Y. Wang, R. Wang, C. Tai, and N. S. Kim, "Rambda: Rdma-driven acceleration framework for memory-intensive μs-scale datacenter applications," in *HPCA*, 2023.
[86] B. Zhou, H. Jin, and R. Zheng, "A high speed lossless compression algorithm based on cpu and gpu hybrid platform," in *TrustCom*, 2014.