DPDPU: Data Processing with DPUs

Jiasheng Hu[1], Philip A. Bernstein[2], Jialin Li[3], Qizhen Zhang[1]
[1]University of Toronto, [2]Microsoft Research, [3]National University of Singapore
[1]{jasonhu, qz}@cs.toronto.edu, [2]philbe@microsoft.com, [3]lijl@comp.nus.edu.sg

ABSTRACT

Improving the performance and reducing the cost of cloud data systems is increasingly challenging. Data processing units (DPUs) are a promising solution. We characterize their capabilities and constraints. We then propose DPDPU, a platform for holistically exploiting DPUs to optimize data processing tasks that are critical to performance and cost. It seeks to fill the semantic gap between DPUs and data processing systems and handle DPU heterogeneity with three engines dedicated to compute, networking, and storage. This paper describes our vision, DPDPU's key components, their associated utilization challenges, as well as the current progress and future plans.

1 INTRODUCTION

Recent trends in computing and data center architectures have made improving the performance and cost efficiency of cloud data systems increasingly challenging. First, speedup of general-purpose processors is not keeping up with data growth. This has led to degraded compute-bound performance gains over large volumes of data. Second, high-bandwidth I/O devices, e.g., solid-state drives (SSDs) and network interface cards (NICs), have greatly increased the speed of data movement. However, since the CPU instructions executed per-byte-accessed during I/O remains nearly constant [14], moving data at a higher rate consumes significantly more CPU resources. Moreover, cloud providers are evolving their data centers to be more disaggregated. With decoupled compute and data, resource disaggregation intensifies network communication, exacerbating the performance and cost challenges in cloud data systems.

A long line of work has aimed to address these cloud data processing challenges: hardware acceleration using domain-specific hardware (e.g., GPUs [9, 13, 24, 25, 32, 36] and FPGAs [12, 18, 33, 41]), OS kernel bypass with userspace I/O (e.g., RDMA [30, 38, 43] and SPDK [1, 14]), as well as caching [47] and compute push-down [42, 46] to minimize the impact of disaggregation. Each of these proposals has limitations of its own. Hardware acceleration demands deep hardware expertise and embeds domain-specific, non-portable characteristics into system designs; userspace I/O requires modification to applications for direct hardware access, making it impractical for existing large-scale software systems such as DBMSs to adopt; though effective in improving performance, techniques targeting resource disaggregation fall short to reduce cost. Overall, *there is a lack of holistic platforms that systematically combat cloud data processing performance and cost challenges.*

Data processing units (DPUs) [2–5, 26], the latest generation of programmable NICs (i.e., SmartNICs), emerge as a promising hardware platform. A DPU is a System-on-a-Chip (SoC) equipped with a collection of hardware resources optimized for data-path efficiency. This includes energy-efficient CPUs (e.g., Arm cores), hardware accelerators (e.g., compression and encryption ASICs), network processors, and a moderate amount of onboard memory. DPUs are positioned to overcome the limitations of existing proposals. As an SoC, a DPU runs independently of the host. It can, therefore, augment the overall system architecture without modifying host applications, facilitating portability and adoption. In addition, a DPU can offload host I/O processing at line-rate, reducing resource consumption on the host.

Effective utilization of DPUs for data processing systems has to address the following challenges.

Challenge #1: abstraction mismatch. DPUs are packet-oriented networking devices. Consequently, the programming interfaces exposed by DPUs are not intended for data system developers and operators. For instance, NVIDIA's DOCA [27] and Intel's IPDK [16] enable users to build in-network offloading pipelines. They provide libraries such as DPDK [22], OVS [34], and P4 [35], with which user programs operate on packets, flows, and raw bytes, rather than data objects (e.g., pages and records). Friendlier DPU interfaces and toolkits are thus needed for data systems.

Challenge #2: resource diversity. A DPU SoC consists of a spectrum of hardware resources, ranging from general CPU cores to specialized ASICs and peer device accessibility via PCIe. Orchestrating these processing units for various data tasks and scheduling tasks across them to reflect workload dynamics are non-trivial. [23].

Challenge #3: DPU heterogeneity. Many parties are producing their own DPUs, such as NVIDIA BlueField [4], Intel IPU [3], Microsoft Fungible [2], Alibaba CIPU [5], and AWS Nitro [6]. Running data systems on heterogeneous DPUs requires an infrastructure that facilitates portability. Even though DPUs share similar architectural characteristics (Section 3), they differ significantly in detailed hardware specifications. For example, NVIDIA BlueField-2 is equipped with a regular expression hardware accelerator, which is missing in Intel IPU; BlueField-3 supports generic code offloading to NIC cores, while most other DPUs only support match-action table style network offloading. Making things worse, DPU vendors often provide proprietary SDKs for programming the device. Without a portable framework, developers have to manually rewrite DPU-specific optimizations for all the different DPUs.

This paper proposes DPDPU, a holistic DPU-centric framework for cloud data processing. Our key insight is when the aforementioned impediments are tamed, data systems can efficiently exploit DPUs to optimize a wide spectrum of tasks, e.g., workloads that are compute-, network-, or storage-intensive.

DPDPU includes three components to judiciously harness the various DPU resources: *a compute engine* that runs on DPU CPU

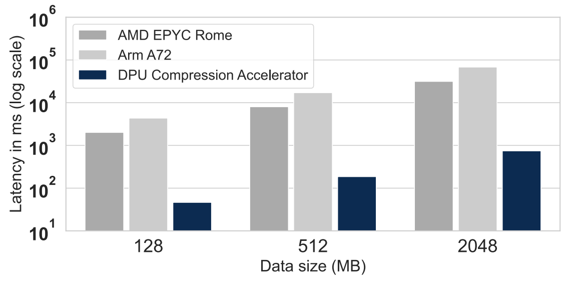

Figure 1: Compression performance on different hardware

cores and hardware accelerators for tasks such as expensive on-path data operations (e.g., compression and encryption) and pushdown database operators (e.g., predicates and aggregation); *a network engine* that offloads communication primitives from host CPUs to the DPU network interfaces; and finally, *a storage engine* that leverages direct storage device access to improve local and disaggregated storage performance while saving cost. DPDPU further schedules tasks across DPU hardware accelerators, DPU CPUs, and host CPUs based on task specifications and resource availability.

DPDPU offers high-level, hardware-neutral interfaces to ease programming and porting effort for DPU accelerated data systems. Specifically, users write stored procedures to express tasks in the compute engine. The network and storage engines expose a familiar asynchronous I/O abstraction, allowing existing data systems to adopt DPDPU with minimal change. We eschew vendor-specific features in the framework such that customized optimizations atop DPDPU are portable across different DPUs.

This paper makes the following contributions.

- We demonstrate the performance and cost challenges in cloud data processing (Section 2), and show the opportunities enabled by DPUs (Section 3).
- We present the overall vision of DPDPU (Section 4).
- We discuss challenges in each DPDPU component and propose the high-level design (Sections 5, 6, and 7).
- We survey related prior work (Section 8), report on current progress, and propose next steps for DPDPU (Section 9).

2 EMERGING CHALLENGES IN THE CLOUD

Compute inefficiency. It is well-known that CPU speed has been increasing rather slowly over the past decade. On the other hand, data systems frequently invoke compute-heavy subroutines. For instance, DBMSs often compress and encrypt data to handle privacy, security, and data size challenges. Can data systems still rely on CPUs to sustain good performance on these compute tasks?

To answer this question, we measured the performance of lossless compression (with the DEFLATE algorithm [29]) on natural language datasets of various sizes on an AMD EPYC CPU and an Arm CPU. Figure 1 shows that, while the EPYC CPU outperforms the Arm CPU, both suffer from high latencies which grow with data size. This shows that it is increasingly difficult for data systems to perform compute-intensive operations on large-scale data.

I/O cost. Performing high-bandwidth I/O is among the most common tasks in database systems. We next evaluate the CPU consumption of accessing 8 KB pages from Linux-managed SSDs.

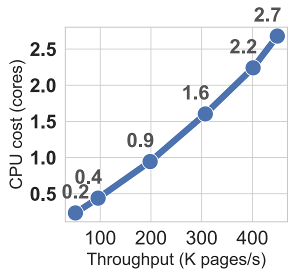

Figure 2: CPU consumption of storage access

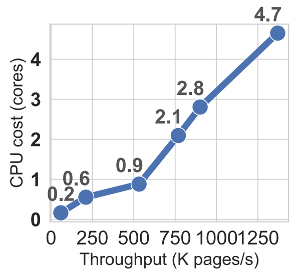

Figure 3: CPU consumption of network communication

As observed in Figure 2, the number of CPU cycles increases linearly with I/O throughput. When the throughput reaches the peak (450 K pages per second), the average CPU consumption is as high as 2.7 cores. We also tested Linux storage performance with the more recent `io_uring`, and observed similar CPU cost. The experiment highlights the growing hardware cost due to high I/O requirements of data systems.

Disaggregation overhead. Lastly, we assess the overhead of resource disaggregation. In particular, storage disaggregation, where compute and storage are hosted on different servers connected via the network, has been commonplace in today's cloud data centers. The architecture enables better flexibility in resource management but at the expense of additional network I/O for storage accesses. leading to higher access latency and even more CPU consumption.

To quantify the latency and CPU consumption overhead of disaggregation, we measure the cost of network communication via TCP/IP sockets for transferring 8 KB pages over a 100 Gbps network. As shown in Figure 3, the additional network I/O induced by disaggregation consumes significant CPU resources, particularly at higher bandwidth. This I/O processing compete with other compute tasks, such as those in Section 2, for CPUs.

3 DPU OPPORTUNITIES

DPUs are SoC-based SmartNICs[1]. Figure 4 shows the architecture of NVIDIA BlueField-2 (BF-2) [4], a popular DPU in mass production. Resources on a DPU can be categorized into five types: (1) energy-efficient CPU cores, (2) onboard memory, (3) hardware accelerators, (4) high-speed network interfaces, and (5) PCIe interface. BF-2 consists of 8 Arm A72 cores clocked at 2.5 GHz, 16 GB DDR4 memory, a set of accelerators including regular expression, compression, encryption and deduplication, ConnectX-6 NIC with 100 Gbps bandwidth, and a PCIe 4.0 switch that has access to host memory and other PCIe-connected devices such as SSDs and GPUs. Although the hardware details vary across vendors, the general capabilities of other DPUs are similar, e.g., Intel IPU [3] and Microsoft Fungible [2]. These resources, combined with DPU data-path optimizations, can be leveraged to address the above challenges.

Specifically, to improve compute efficiency, data systems can utilize the hardware accelerators to execute compute-intensive operations on the data path, which are ASICs designed for specific compute tasks; improving power efficiency and performance compared to CPUs. Figure 1 shows that the compression accelerator on

[1]The other major category is FPGA-based SmartNICs. We focus on SoC-based SmartNICs for their easier programmability and development process.

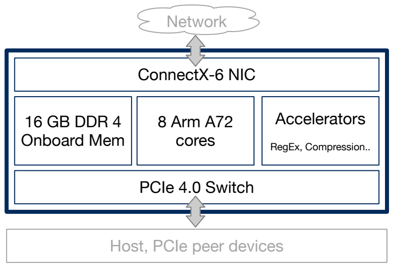

Figure 4: NVIDIA BlueField-2 DPU architecture

BF-2 outperforms CPUs by an order of magnitude. To reduce I/O cost, DPUs usually provide advanced userspace libraries to build efficient I/O pipelines. For instance, BF-2 offers SPDK and DPDK so direct access to storage devices and network interfaces is possible without host involvement. Together with the general-purpose CPU cores and a moderate amount of memory on board, users can build arbitrary, low-latency and high-bandwidth I/O services to free the host from expensive storage and network activities.

Despite the potential benefits, challenges around abstraction mismatch, resource diversity, and DPU heterogeneity must be addressed in order to better utilize DPUs for data processing systems.

4 THE DPDPU FRAMEWORK

We envision a DPU-powered software platform that exploits the opportunities in Section 3 to tackle the cloud data processing problems in Section 2. The platform, called DPDPU, does so by (1) bridging the semantic gap between raw DPU resources and cloud data processing tasks, (2) efficiently utilizing diverse hardware resources on different DPUs, and (3) decoupling hardware details of different DPUs from the optimizations at the data system layer. As shown in Figure 5, DPDPU consists of three modules that allow for optimizing compute-intensive and I/O-intensive operations.

Components and accessed resources. We now describe the DPU resources managed by each component and the interactions between components. The next sections discuss detailed designs of each component and the key challenges.

Compute Engine offers *efficient* and *versatile* computational power for data processing tasks. The engine carefully orchestrates these tasks across four types of compute resources: DPU onboard CPUs, DPU hardware accelerators, host CPUs, and other popular accelerators, e.g., GPUs and FPGAs, connected via PCIe. The working set of execution can be cached in both DPU and host memory.

Network Engine handles network I/O. It utilizes the advanced networking facilities built in DPUs (high-speed interfaces, match-action offloading, and user libraries) to improve network I/O efficiency. More specifically, the DPU DMA engine serves as an abstraction boundary to decouple popular networking APIs utilized by host applications from their protocol execution, offloading to the DPU using onboard memory, CPU, and the network interface.

Storage Engine improves storage path efficiency, including requests from both local applications and those from remote clients. For local applications, the engine offers a lightweight user library to forward storage requests from the client to the DPU, where it accesses SSDs via PCIe peer-to-peer communication. For requests from remote clients, it coordinates with the Network Engine to execute storage requests immediately on the DPU without involving the host.

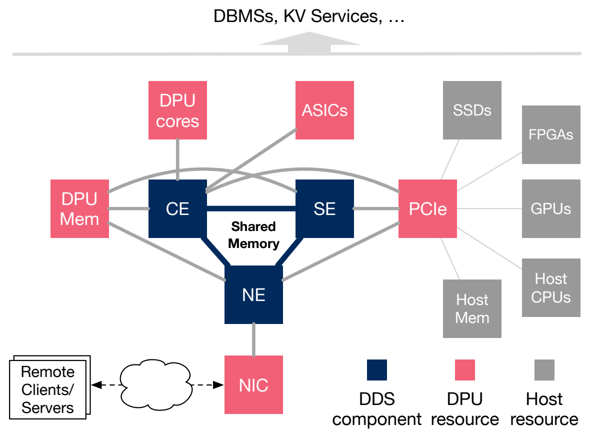

Figure 5: DPDPU components—Compute, Network, and Storage Engines (CE, NE, and SE)—and resources they access.

Interactions. DPDPU components can be *composed* to execute complex tasks. For instance, in response to a remote storage access request, a DPDPU program may first read the data from local SSDs using the Storage Engine. It then invokes the Compute Engine to compress the data in the DPU compression accelerator. Finally, the Network Engine delivers the result to the client.

Take predicate pushdown as another example. Leveraging DPDPU, the storage server first reads the database records from SSDs through the Storage Engine. It then directly applies predicates on these tuples using the Compute Engine, and sends only the qualified tuples back to the remote database server via the Network Engine.

DPDPU facilitates composability using two mechanisms. First, it enables shared states across the three engines via the DPU memory. The schema of the state and cached data are customizable by the application. Note that within each component, consistency is not guaranteed due to asynchronous accesses, e.g., the network, the hardware accelerators, and host resources via PCIe.

Moreover, DPDPU enables efficient, streamlined data communication across engine boundaries. To do so, the API and the execution model of the engines facilitate pipelined data processing—one engine's output can be streamed to another engine without waiting for the completion of work-in-progress. This allows for constructing efficient asynchronous pipelines that overlap I/O and computation.

5 COMPUTING

We design the Compute Engine (CE) with the following goals.

(1) **Efficient.** As the primary motivation for CE is to address compute inefficiency, we aim to maximize the efficiency of compute tasks that are offloaded to CE.
(2) **General-purpose.** To benefit various cloud data processing systems, the CE should handle a wide spectrum of tasks, from data-path primitives to relational operator pushdown.
(3) **Easy to program.** A major difficulty of programming DPUs is the low-level interfaces across different hardware designs. CE offers APIs familiar to data system developers.
(4) **Portable.** In addition to portability across DPUs, CE must also account for the diverse compute resources on the DPU and the host when executing the same user tasks.

Interface. We provide stored procedures (sprocs) for users to express their compute tasks. Previous work [44] has explored using sprocs as a general programming abstraction to offload computation for data processing systems. Despite their benefits, sprocs are primarily designed for CPU execution, and lacks native support for hardware acceleration. To overcome this, we introduce *DP kernels*, an extensible set of specialized functions built in DPDPU that optimizes sproc execution efficiency. The user can query which DP kernels are available in the CE and select the ones that match the application's need. However, they do not concern developers with hardware details. Sprocs with DP kernels naturally satisfy CE's general-purpose and easy to program goals. We next discuss execution details and explain how efficiency and portability is achieved.

Execution. A sproc is first registered with the CE, which precompiles it into a shared library, which the user program loads to run on a DPU core. DP kernels, on the other hand, represent compute-heavy tasks and thus are prioritized for hardware acceleration to maximize compute efficiency. However, DPUs are heterogeneous, and specific accelerators are not universally supported. E.g., BlueField-2 has a RegEx engine, which is not available on Intel IPU. To ensure the same user code can run on different DPUs, DP kernels must be portable and backward- and forward-compatible.

To that end, we allow that each DP kernel can be executed on *any compute hardware*, e.g., CPUs, ASICs, FPGAs, or GPUs. The actual execution during runtime depends on hardware availability. Users can specify where a DP kernel is executed (specified execution); alternatively, the CE can construct a schedule for all the DP kernels (scheduled execution). Scheduled execution enables the CE to optimize the overall performance of a sproc given hardware constraints of the target DPU platform.

Example. An example of a sproc with a DP kernel is shown in Figure 6. The sproc serves a request from a remote client that reads a set of pages, compresses them, and sends the compressed pages back to the client. Since compression is the most compute-intensive task in this sproc, we accelerate it using the compression kernel (`dpk_compress`). Here, the user first specifies the kernel to be executed on the compression accelerator (line 20). If the accelerator is currently unavailable on the DPU, the user moves the computation to a DPU CPU core (line 24).

Alternatively, the implementation can leave target device unspecified in `dpk_compress`. The kernel will then be scheduled by CE, and the call always returns a valid work item in progress. The main benefit of specified execution is predictable program behavior; it, however, leaves the burden of optimizing sproc performance to the user.

Open challenges. Developing the CE must address several technical challenges. First, since a sproc may be invoked in parallel at a high rate, e.g., per received packet, *proper scheduling is critical to the overall performance*. Prior work adopted various scheduling disciplines to achieve high NIC offloading performance. For instance, iPipe [23] utilizes a first-come-first-served queue and a deficit round robin queue to schedule tasks with low and high variance respectively across DPU and host CPU cores. The CE also needs to schedule DP kernels across all computing units. Hardware accelerators exhibit vendor-specific characteristics, (e.g., high

```
1  import dpdpu.compute_engine as ce
2  import dpdpu.network_engine as ne
3  import dpdpu.storage_engine as se
4
5  read_compress_send_pages(req):
6    page_read_list = {}
7    page_comp_list = {}
8    page_send_list = {}
9    dpk_compress = ce.get_dpk("compress")
10
11   for net_req in req.pages:
12     # async read
13     read_req = se.read(net_req.file_id,
14       net_req.addr, PAGE_SIZE)
15     page_read_list.add(read_req)
16
17   for read_req in page_read_list:
18     wait(read_req)
19     # async compression (fast)
20     comp_req = dpk_compress(read_req.data,
21       "dpu_asic")
22     if comp_req is None:
23       # async compression (slow)
24       comp_req = dpk_compress(read_req.data,
25         "dpu_cpu")
26     page_comp_list.add(comp_req)
27
28   for comp_req in page_comp_list:
29     wait(comp_req):
30     # async send with TCP
31     send_req = ne.tcp.send(req.client,
32       comp_req.data)
33     page_send_list.add(send_req)
34
35   for send_req in page_send_list:
36 wait(send_req)
```

Figure 6: An example of sproc with DP kernels where page compression is accelerated (specified execution). Different modes of execution facilitate portability.

throughput with high latency) that are distinct from CPUs. Consequently, the problem space for scheduling in DPDPU is expanded: How to schedule DP kernels on the same accelerator? How to co-schedule sprocs and DP kernels? How to cater for performance targets from different applications?

Second, a server equipped with a DPU can run multiple applications. *The CE should provide fairness and performance isolation in a multi-tenant setting.* A naive approach can use containers to slice CPUs and memory on both the DPU and host. A complete solution, however, must also consider hardware accelerators. Compared to CPUs, accelerator capacity (i.e., the number of concurrent tasks) varies greatly across hardware. Accelerators also lack virtualization support. Hence, multiplexing resources and isolating DP kernel execution on accelerators present a challenge.

Finally, CE can be extended to PCIe-connected accelerators such as FPGAs and GPUs. We need to map DP kernels to these devices and find efficient data paths based on how a sproc and its DP kernels span across different locations. Since such accelerators have

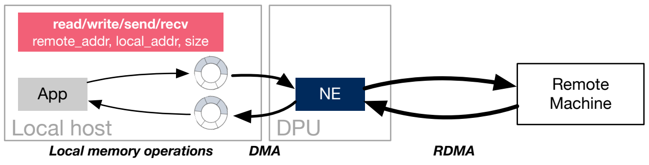

Figure 7: DPU-optimized RDMA

more cores and memory than DPU accelerators, fusing multiple DP kernels inside an accelerator may improve performance.

6 NETWORKING

Our primary goal for the Network Engine (NE) is to lower communication overhead while maintaining high performance for popular transport protocols, e.g., TCP and, more recently, RDMA. The principle in designing NE is to offload CPU consuming network activities to the DPU, while leaving only lightweight front-end libraries that emulate existing communication frameworks' APIs. This is enabled by DPU's DMA and packet generation capabilities.

Optimizing TCP. TCP/IP remains the most popular protocol for network communication in data processing systems, which consumes substantial host CPU cycles (§2). Recent proposals mitigated TCP cost with DPUs. For instance, IO-TCP [20] divides TCP into a control plane (e.g., connection management) that runs on a single core on the host and a data plane (data transmission) that runs on the DPU. These solutions, however, target specific applications (e.g., streaming media files) and require application modifications.

To support general communication for distributed and disaggregated data processing, we propose to move the TCP/IP stack to the DPU and provide a POSIX-like socket API for host applications through a user library. Doing so requires tackling two challenges. First, as the DPU's CPU is significantly weaker than the host's, the DPU's TCP/IP stack must be carefully optimized to avoid performance degradation. Second, as network messages are eventually processed on the host, flow control now spans the host and DPU. We must co-design TCP on the DPU and host-DPU communication to reflect signals from host applications in the flow control protocol.

Optimizing RDMA. Remote Direct Memory Access (RDMA) has emerged as a promising data center networking technology for achieving high-speed network communication in data processing systems [30, 43]. RDMA runs in userspace and can completely bypass OS overhead. It also eliminates remote CPU involvement via DMA from NIC hardware. To best utilize RDMA for database systems, DFI [38] layers a data flow interface atop the transport to provide pipelined, thread-centric flow execution. It achieves communication performance that is close to the raw RDMA network.

Despite its performance benefits over traditional networking stacks, issuing RDMA operations is still CPU costly. For instance, accessing the send/receive queues in an RDMA queue pair requires spinlocks and memory fences to ensure queue ordering. CPU stalls can also happen when ringing the doorbell register of the RDMA NIC. These overheads have been confirmed by recent work [11].

Figure 7 depicts our proposal for optimizing RDMA communication. The design offloads the heavy issuing-side RDMA handling to the DPU. We first replace the RDMA queues with lock-free ring buffers to accept user requests. These buffers are DMA-accessible such that NE on the DPU can poll user requests using the DPU

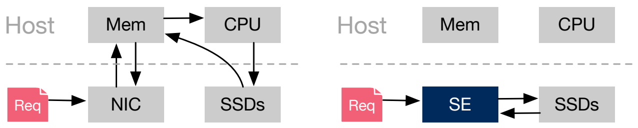

Figure 8: Round trips from NIC to host in today's disaggregated storage (left) can be saved with DPDPU SE (right).

DMA engine. Upon receiving requests, NE issues corresponding RDMA read/write or send/receive to access memory on the remote machine. This asynchronous execution of RDMA must be served together with the non-blocking interface on the host, such that applications only spend minimal resources polling responses.

Cowbird [11] proposes an asynchronous I/O abstraction for disaggregated memory; it offloads RDMA to programmable switches and harvested VMs. NE can be viewed as an extension to Cowbird targeting general network communication, supporting both one- and two-sided RDMA. The key challenge is co-designing the interface and the execution of the full set of RDMA operations while consuming minimal resources on both the host and the DPU.

7 STORAGE

Offloading file-related operations onto a DPU can free significant host resources (§2). In addition, a DPU sits on the data path to serve requests for disaggregated storage (Figure 8): when a remote storage request arrives, the DPU can service the request immediately by accessing PCIe-connected SSDs. In comparison, existing disaggregated storage must process the request using host CPUs, incurring additional PCIe, OS, and storage stack overheads.

Offloading file execution. We first propose a DPU-backed storage framework that offers a POSIX-like file system API for host applications to manage files and perform file I/O. The processing of file requests is offloaded to the DPU, where we build a *file service* leveraging userspace storage solutions, e.g., SPDK, to optimize file I/O efficiency. Similar to NE, the contention between application threads is minimized with lock-free ring buffers in the user library, and the requests are lazily DMA'ed by the DPU. Our design requires delegating the management of SSDs from host servers to DPUs, which is a popular trend of adopting DPUs [28].

Offloading remote requests. To fully exploit DPU for disaggregated storage, we propose an offload-engine in the SE that allows users to directly process remote storage requests on the DPU. A supplied UDF parses network messages to identify requests that can be offloaded, and translates them into file operations. Since the DPU maintains the mapping between user files and physical blocks on the SSDs (the file mapping) in its aforementioned file service, SE can directly execute the file operation without host involvement.

The key challenge in realizing this design is the limited resources on DPUs. For instance, transaction updates in cloud-native DBMSs are reflected on disaggregated storage servers with log replay [7, 40], which can consume up to 100s GB memory caching hot pages to prevent write amplification. This memory footprint is an order of magnitude larger than the DPU's memory capacity (e.g., 16 GB). Hence, storage requests unsuited for DPU offloading must still be forwarded to the host. This partial offloading raises several technical questions: which requests should be offloaded? What should the offloading API look like to reflect the division? How do we split network traffic without violating transport protocol semantics?

8 RELATED WORK

SmartNICs have been explored for distributed systems and computer networking. Below we summarize this line of work.

LineFS [19] improves the efficiency of distributed file system by offloading CPU-intensive tasks to the DPU and use pipeline parallelism to improve performance. Xenic [31] caches data on DPUs to accelerate distributed transactions. hKVS [10] is a KV store that uses DPU memory to cache hot records and meticulously synchronizes updates to the host. iPipe [23] proposes an actor-based execution framework that utilizes DPUs for distributed applications, enabling scheduling and flexible load migration between the DPU and the host. More recently, IO-TCP [20] proposes a host-DPU codesigned TCP that leverages the data-path efficiency of DPUs to offload the streaming of media files. Lovelock [28] is a DPU-based cluster manager that eliminates the need for host servers, e.g., for hardware accelerators and storage devices.

DPUs are increasingly attracting attention in the database community. Thostrup et al. [37] evaluate the performance benefits of a specific DPU (NVIDIA BlueField-2) for two specific DBMS components (a B-tree index and a sequencer). Their results are aligned with our DPU characterization. SmartShuffle [21] offloads to DPUs networking components as well as DBMS tasks in data shuffling.

Our work differs from others in the generality of our proposal. It systematically exploits the capabilities of DPU SoCs to tackle a spectrum of challenges in cloud data processing. The three complementary engines in DPDPU present an easily-utilized and portable offering of DPU resources for data system optimizations.

More generally, offloading operations to programmable network and storage devices has been shown to be an effective approach to improving database systems efficiency [8, 15, 17, 39]. Our proposal seeks to support relevant data-path operations in a unified platform.

9 PROGRESS AND NEXT STEPS

Our first step towards realizing DPDPU is developing DDS [45], a DPU-optimized disaggregated storage server architecture as part of the Storage Engine. Recall from Section 7 that a DPU is inappropriate for fully offloading disaggregated storage requests. Hence, the design of DDS is centered around partial offloading, i.e., remote storage requests are split between the DPU and the host. We need to address three key questions: (1) *how to access files on SSDs directly from the DPU?* (2) *how to direct traffic between the DPU and the host?* and (3) *how to enable general and efficient DPU offloading?*

For (1), we developed a unified file system that directs file operations on the host to the DPU. Doing so allows the DPU to own the file mapping for serving a remote request. Question (2) is handled by a traffic director that determines whether each packet should be forwarded to the DPU or the host. It accomplishes the task without breaking end-to-end transport semantics. Finally, for (3) we introduce a high-level API in the offload engine for users to implement the UDF in Section 7, and extensively employ zero-copy to maximize the efficiency of request execution. We integrated DDS with FASTER (a KV store) and Azure SQL Hyperscale (a cloud-native DBMS), two production systems at Microsoft. Empirical studies show that DDS can save up to 10s of CPU cores per storage server.

DPDPU opens a broad space of systems and optimization research for cloud data processing. Our next steps are as follows.

Caching in DPU-backed file system. DDS currently achieves minimal memory footprint and has no support for caching on either the host or the DPU in the file system. With more memory, we can cache hot data to further improve file performance. How to cache, however, is non-trivial because of separate sources of access: caching in host memory favors host applications, while DPU memory favors remote requests that can be offloaded. A key challenge is sizing the cache at the right granularity on the DPU and host based on workload characteristics.

Faster persistence. Techniques such as caching and DDS have been adopted to improve the read query performance for many data systems. Although in cloud data systems writes are less prevalent than reads, optimizing persistent updates, particularly their end-to-end latency, is meaningful to mission critical applications and presents unique challenges. Persistent operations often traverse deeper storage stacks than reads; the backing store typically runs on slow hard drives, many located in disaggregated storage. DPDPU offers opportunities to accelerate persistence performance. By directly connecting DPUs with fast persistent storage (e.g., NVMe SSDs) through PCIe P2P, we can persist a write request to storage devices or DPU's onboard fast storage and immediately acknowledge the request before forwarding the operation to the host. We plan to design a generic DPU fast-persistence interface that allows existing data systems to benefit from fast persistence with minimum code modifications. We also need to address the challenge of coordinated recovery in this new model, as well as consistency issues arising from concurrent reads, including both reads forwarded to the host and those offloaded to the DPU, and fast persistent writes.

Implementing DP kernels. DP kernels are at the core of DPDPU's Compute Engine for harvesting the compute efficiency of various DPU processing units. As detailed in Section 5, designing and implementing these primitives is challenging. As DP kernels are portable across DPUs, we need to investigate a collection of vendor-provided DPU SDKs, seeking plans that avoid excessive engineering effort. Scheduling DP kernels (and co-scheduling them with sprocs) based on app-specific performance requirements is another critical task.

Database communication optimization. In addition to the design challenges in Section 6, developing the Network Engine requires mapping out the details of target networking protocols (i.e., TCP and RDMA) and constructing cross-host-DPU operations that decouple interface and protocol execution.

In our experience, the internal networking stack of cloud-native production DBMSs is a primary source of I/O overhead. We thus plan to dissect the networking stacks of open-source systems to search for a common set of DBMS-specific communication tasks suitable for DPU offloading.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. We also thank Badrish Chandramouli and Bingsheng He for their valuable comments on this work at different stages. This work was supported in part by an NSERC Discovery grant.

REFERENCES

[1] Storage performance development kit. https://spdk.io/.

[2] The fungible dpu. https://www.hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Fungible_v04.pdf, 2020.

[3] Intel infrastructure processing unit (intel ipu). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2023.

[4] Transform the data center with nvidia dpus. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2023.

[5] Alibaba Cloud Community. A detailed explanation about alibaba cloud cipu. https://www.alibabacloud.com/blog/599183, 2022.

[6] All Things Distributed. Reinventing virtualization with the aws nitro system. https://www.allthingsdistributed.com/2020/09/reinventing-virtualization-with-nitro.html, 2020.

[7] P. Antonopoulos, A. Budovski, C. Diaconu, A. H. Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade. Socrates: The new SQL server in the cloud. In *SIGMOD 2019*. ACM.

[8] A. Barbalace and J. Do. Computational storage: Where are we today? In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[9] N. Boeschen and C. Binnig. Gacco - A gpu-accelerated OLTP DBMS. In *SIGMOD '22*. ACM, 2022.

[10] H. Chen, C. Chang, and S. Hung. hkvs: a framework for designing a high throughput heterogeneous key-value store with smartnic and RDMA. In *RACS 2022*. ACM.

[11] X. Chen, L. Yu, V. Liu, and Q. Zhang. Cowbird: Freeing cpus to compute by offloading the disaggregation of memory. In *SIGCOMM '23*. ACM, 2023.

[12] M. Chiosa, F. Maschi, I. Müller, G. Alonso, and N. May. Hardware acceleration of compression and encryption in SAP HANA. *Proc. VLDB Endow.*, 2022.

[13] H. L. et al. Ghive: A demonstration of gpu-accelerated query processing in apache hive. In *SIGMOD '22*. ACM.

[14] G. Haas, M. Haubenschild, and V. Leis. Exploiting directly-attached nvme arrays in DBMS. In *CIDR 2020*, 2020.

[15] R. Hussein, A. Lerner, A. Ryser, L. D. Bürgi, A. Blarer, and P. Cudré-Mauroux. Graphinc: Graph pattern mining at network speed. *Proc. ACM Manag. Data*, 1(2):184:1–184:28, 2023.

[16] IPDK. Infrastructure programmer development kit. https://ipdk.io/, 2023.

[17] T. Jepsen, A. Lerner, F. Pedone, R. Soulé, and P. Cudré-Mauroux. In-network support for transaction triaging. *Proc. VLDB Endow.*, 14(9):1626–1639, 2021.

[18] M. Kiefer, I. Poulakis, E. T. Zacharatou, and V. Markl. Optimistic data parallelism for fpga-accelerated sketching. *Proc. VLDB Endow.*, 16(5), 2023.

[19] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostic, Y. Kwon, S. Peter, and E. Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *SOSP '21*. ACM, 2021.

[20] T. Kim, D. M. Ng, J. Gong, Y. Kwon, M. Yu, and K. Park. Rearchitecting the TCP stack for i/o-offloaded content delivery. In *NSDI 2023*. USENIX Association.

[21] J. Lin, T. Ji, X. Hao, H. Cha, Y. Le, X. Yu, and A. Akella. Towards accelerating data intensive application's shuffle process using smartnics. In *SIGMETRICS '23*.

[22] Linux Foundation. Data plane development kit. https://www.dpdk.org/, 2023.

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

[24] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *SIGMOD '22*. ACM.

[25] T. Maltenberger, I. Ilic, I. Tolovski, and T. Rabl. Evaluating multi-gpu sorting with modern interconnects. In *SIGMOD '22*. ACM.

[26] Marvell. Marvell liquidio iii: An inline dpu based smartnic for cloud network and security acceleration. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, 2020.

[27] NVIDIA Developer. Nvidia doca software framework. https://developer.nvidia.com/networking/doca, 2024.

[28] S. J. Park, R. Govindan, K. Shen, D. E. Culler, F. Özcan, G. Kim, and H. Levy. Lovelock: Towards smart nic-hosted clusters. *CoRR*, 2023.

[29] Peter Deutsch. Deflate compressed data format specification version 1.3. https://www.w3.org/Graphics/PNG/RFC-1951, 2024.

[30] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4), 2015.

[31] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *SOSP '21*. ACM, 2021.

[32] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden. Tile-based lightweight integer compression in GPU. In *SIGMOD '22*. ACM.

[33] H. Tan, X. Chen, Y. Chen, B. He, and W. Wong. Lightrw: FPGA accelerated graph dynamic random walks. *Proc. ACM Manag. Data*, 1(1), 2023.

[34] The Linux Foundation. Open vswitch. https://www.openvswitch.org, 2024.

[35] The Linux Foundation. P4 - language consortium. https://p4.org, 2024.

[36] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig. Distributed GPU joins on fast rdma-capable networks. *Proc. ACM Manag. Data*, 1(1).

[37] L. Thostrup, D. Failing, T. Ziegler, and C. Binnig. A dbms-centric evaluation of bluefield dpus on fast networks. In *ADMS@VLDB 2022*, 2022.

[38] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig. DFI: the data flow interface for high-speed networks. In *SIGMOD '21*. ACM, 2021.

[39] M. Tirmazi, R. B. Basat, J. Gao, and M. Yu. Cheetah: Accelerating database queries with switch pruning. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2407–2422. ACM, 2020.

[40] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017*. ACM, 2017.

[41] M. Xekalaki, J. Fumero, A. Stratikopoulos, K. Doka, C. Katsakioris, C. Bitsakos, N. Koziris, and C. Kotselidis. Enabling transparent acceleration of big data frameworks using heterogeneous hardware. *Proc. VLDB Endow.*, 15(13), 2022.

[42] X. Yu, M. Youill, M. E. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. Pushdowndb: Accelerating a DBMS using S3 computation. In *ICDE 2020*. IEEE, 2020.

[43] Q. Zhang, P. A. Bernstein, D. S. Berger, and B. Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 2021.

[44] Q. Zhang, P. A. Bernstein, D. S. Berger, B. Chandramouli, V. Liu, and B. T. Loo. Compucache: Remote computable caching using spot vms. In *CIDR 2022*.

[45] Q. Zhang, P. A. Bernstein, B. Chandramouli, J. Hu, and Y. Zheng. DDS: DPU-optimized Disaggregated Storage. *Proc. VLDB Endow.*, 17(11), 2024.

[46] Q. Zhang, X. Chen, S. Sankhe, Z. Zheng, K. Zhong, S. Angel, A. Chen, V. Liu, and B. T. Loo. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In *SIGMOD '22*. ACM, 2022.

[47] Y. Zhang, C. Ruan, C. Li, J. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo, and C. Bi. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.*, 2021.