2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

DCS-ctrl: A Fast and Flexible Device-Control Mechanism for Device-Centric Server Architecture

Dongup Kwon*[1], Jaehyung Ahn†[1], Dongju Chae†, Mohammadamin Ajdari†,
Jaewon Lee*, Suheon Bae*, Youngsok Kim*, and Jangwoo Kim*
*Department of Electrical and Computer Engineering, Seoul National University
†Department of Computer Science and Engineering, POSTECH

***Abstract*—Modern high-performance servers leverage a large number of emerging peripheral devices (e.g., data processing accelerators, non-volatile memory storage, high-bandwidth network cards) to meet ever-increasing performance demands of server applications. However, as such servers experience severe kernel overhead due to frequently invoked device operations (e.g., buffer management and data copy), server architects have proposed various hardware and software approaches to enable direct communications among the devices. Unfortunately, existing direct device-to-device (D2D) communication schemes still suffer from low performance and the lack of flexibility. First, software-based schemes depend on complicated kernel routines and necessitate multiple hardware-software and user-kernel boundary crossings, which significantly limit the performance improvement opportunities from direct D2D communications. On the other hand, hardware-based schemes require tight integration and custom-built devices, preventing architects from flexibly adding off-the-shelf devices.**

In this paper, we propose DCS-ctrl, a novel Hardware-based Device-Control (HDC) mechanism for Device-Centric Server (DCS) architecture to provide fast and CPU-efficient direct D2D communications among a large number of off-the-shelf peripheral devices. The key idea of DCS-ctrl is to implement a low-cost and flexible device-control mechanism on an independent FPGA device called HDC Engine. As HDC Engine manages all data and control transfers among devices at the hardware level, the server achieves high performance, scalability, and flexibility. First, optimizing both data and control paths at the hardware level minimizes the latency of inter-device communications. Second, implementing FPGA-based reconfigurable device controllers enables direct D2D communications among commodity devices and thus improves per-device flexibility. Third, merging heterogeneous device operations with intermediate data processing supports creates more opportunities for direct inter-device communications in server applications. Our DCS-ctrl prototype reduces the latency of software-based direct D2D communications by 42% and the CPU utilization by 52%.

***Keywords*-Server Architecture; I/O Optimization; FPGA-based Accelerator; Device-to-Device Communication;**

I. Introduction

Modern servers are now being equipped with an increasing number of high-performance peripheral devices such as data processing accelerators [1]–[3], non-volatile memory (NVM) storage [4], [5], and high-bandwidth network interface cards [6], [7] through fast interconnect technologies (e.g., PCI Express (PCIe) and NVLink [8]). Incorporating such high-performance devices and interconnect technologies, modern servers offer unprecedented performance and keep up with the ever-increasing performance demands of server applications.

Intensive uses of high-performance peripheral devices, however, make host-side CPUs and memory extremely busy due to frequently invoked, complicated kernel routines for device operations. The overall server performance is consequently throttled by the latency of software components and the available bandwidth of CPUs and memory. Server architects, to reduce the kernel overhead, have simplified and optimized existing kernel stacks [9]–[13] or aggressively implemented user-level stacks [14], [15]. However, they focus on only single-device tasks (e.g., storage or network) and do not address the kernel overhead of multi-device tasks.

To further reduce the software overhead and latency (e.g., buffer management and data copy operations) of multi-device tasks, server architects have proposed various *direct device-to-device (D2D) communication* schemes: (1) peer-to-peer (P2P) communications and (2) device integration. However, they fail to achieve the expected performance of the high-performance devices and also limit types of devices eligible for D2D communications mainly due to their slow and expensive device-control mechanisms as follows.

First, existing PCIe P2P communication schemes [16]–[19] still rely on slow and CPU-inefficient software-based device-control mechanisms. Their device-control mechanisms responsible for initiating and terminating device operations must be executed in complicated software components and inevitably require frequent software/hardware and user/kernel boundary crossings. Thus, it is difficult for them to achieve the expected performance potential of direct D2D communications.

Second, existing hardware-based device-control schemes depend on custom-built devices tightly integrating heterogeneous devices (e.g., [20], [21]). Their consolidated devices can perform direct data and control transfers without software intervention. However, such custom-built and inte-

[1]These authors contributed equally to this work.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00048

	Host-centric architecture [9]–[15]	PCIe P2P communications [16]–[19]	Device integration [20], [21]	**DCS-ctrl**
Performance	**Slow** (Indirect data copy, SW-based control path)	**Slow** (Direct data copy, SW-based control path)	**Fast** (Direct data copy, HW-based control path)	**Fast** (Direct data copy, HW-based control path)
Scalability	**Not scalable** (CPU-centric)	**Scalable** (SW optimization)	**More scalable** (SW optimization, HW-based device control)	**More scalable** (SW optimization, HW-based device control)
Flexibility	**Flexible**	**Flexible** (Disaggregate implementation)	**Not flexible** (Aggregate implementation)	**Flexible** (Disaggregate implementation)

Table I: Comparison of existing inter-device communication schemes and DCS-ctrl

grated devices cannot achieve cost-effective device allocation nor flexibility as they are expensive and the supported device types are limited by their designs. Their architectural limitation will become increasingly critical when more diverse devices are introduced (e.g., [1], [2]) and server applications require a wide spectrum of device combinations with different target performance of each device.

Third, existing direct D2D communication schemes cannot perform direct inter-device data communications if intermediate data processing decouples device operations. For example, if an application performs data processing (e.g., hash, encryption, compression) between storage and network operations for more stable, secure, and cost-effective services [22]–[25], existing schemes must separate the device operations to perform the data processing using either a host-side CPU or an independent data processing device. Therefore, such decoupled device operations with possible intermediate data processing make it difficult to apply direct D2D communications to existing server applications.

In this paper, we propose *DCS-ctrl*, a novel hardware-based device-control (HDC) mechanism for device-centric server (DCS) architecture to significantly improve performance, flexibility, and applicability of the existing direct inter-device communication schemes. Table I summarizes the benefits provided by existing inter-device communication schemes and DCS-ctrl. The key idea of DCS-ctrl is to implement a low-cost and flexible device-control mechanism on an independent FPGA device called *HDC Engine*.

First, DCS-ctrl significantly reduces the latency of D2D communications and the utilization of host-side CPUs and memory by adopting a hardware-based device-control mechanism. DCS-ctrl performs device-control routines on HDC Engine, an FPGA-based device orchestrator. It consists of a scoreboard and standard device controllers for direct D2D communications without software intervention. The scoreboard schedules device operations involved in a multi-device task, and issues device commands to the corresponding device controllers. Device controllers interact with other peripheral devices through PCIe P2P communications.

Second, DCS-ctrl improves flexibility of direct D2D communications by implementing standard device controllers in HDC Engine. As HDC Engine's device controllers build standard device commands of other peripheral devices, and submit and complete them through PCIe P2P communications, DCS-ctrl does not require any tightly integrated or custom-built devices for fast control paths among devices. Thus, DCS-ctrl supports a large number of heterogeneous devices and enables per-device upgrades, which makes DCS-ctrl an extremely cost-effective and flexible D2D communication scheme.

Third, HDC Engine's reconfigurable near-device processing (NDP) unit can further increase the applicability of direct D2D communications. DCS-ctrl can effectively merge decoupled device operations including intermediate data processing in existing server applications to apply direct D2D communications more aggressively. Thus, by implementing NDP units in an FPGA, HDC Engine can further improve the performance of modern server applications.

For evaluation, we implement our HDC Engine prototype on a Xilinx Virtex7 VC707 board, and build DCS-ctrl server architecture with an Intel NVM Express (NVMe) SSD, Broadcom 10-Gbps NIC, and NVIDIA GPU. We also develop a Linux kernel module and user-level library to apply DCS-ctrl to existing scale-out storage applications.

We first use microbenchmarks to validate the functionality and show the performance impact of our approach. At this step, we show that DCS-ctrl seamlessly performs direct inter-device communications among off-the-shelf SSDs, NICs, and GPUs, while executing all device control routines on HDC Engine. The results show that DCS-ctrl's hardware-based device-control mechanism reduces the latency of software-based D2D operations by 42% (without NDP) and by 72% (with NDP).

Next, we evaluate our prototype with real-world scale-out storage applications including OpenStack Swift [23] and Hadoop distributed file system (HDFS) [22]. As it is impossible to profile the latency of each D2D communication for large-scale server applications, we measure the host-side CPU utilization reduction and the corresponding throughput improvement of the applications. The results show that DCS-ctrl reduces the utilization of host-side CPUs by 52% or improves the throughput by roughly 2× for the same CPU utilization. For future servers equipped with faster devices, our projection expects DCS-ctrl to further

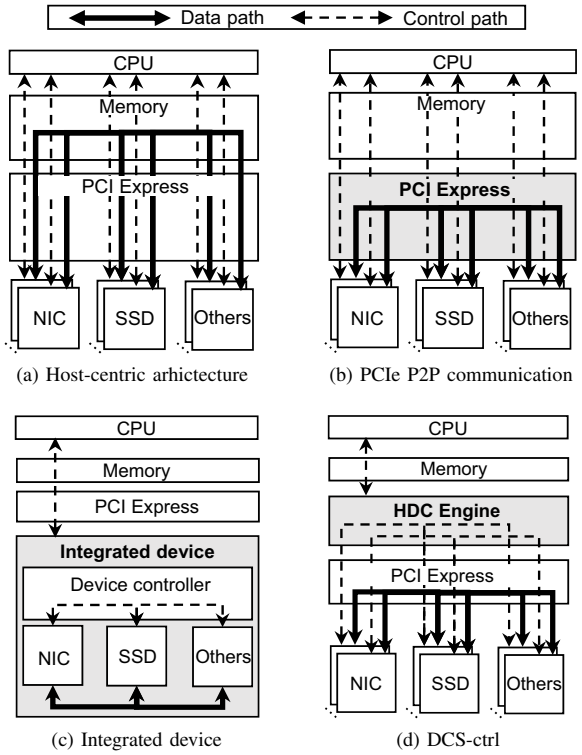


Figure 1: Existing inter-device communication schemes and DCS-ctrl

outperform existing D2D communication schemes.

In summary, our work has the following contributions:

- **Novel architecture**. DCS-ctrl, a fast and flexible hardware-based device control mechanism, significantly improves the performance and flexibility of direct D2D communications.
- **Performance**. FPGA-based HDC Engine orchestrates devices at the hardware level and thus significantly reduces the both D2D communication latency and CPU utilization.
- **Flexibility**. HDC Engine's low-cost and flexible device controllers enable architects to add any types of off-the-shelf devices for direct D2D communications.
- **Applicability**. HDC's NDP unit significantly increases the opportunity of applying direct D2D communications to existing server applications.
- **Prototyping and release**. We release our FPGA IPs and software tools to the community.

II. Background and Motivation

A. Background

Conventional server architectures are *host-centric* as they rely on host-side CPUs and memory for device operations. Figure 1a shows inefficient control and data paths in host-centric server architecture in which CPUs manage all device-control routines (control path), and devices can transfer data to others only through host memory (data path). As complicated kernel-level device-control routines and indirect data copies can delay inter-device communications, their performance significantly relies on available host-side CPUs and memory. Such host-side overhead has rapidly become the dominant performance bottleneck of modern servers equipped with high-performance devices [9], [16]–[21].

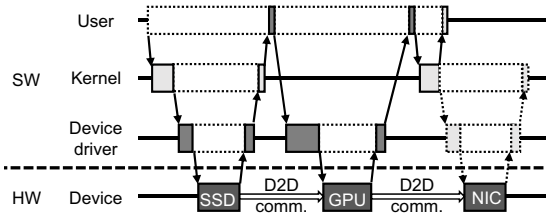


Figure 2: Timeline of a software-based device-control mechanism (each block does not exactly match time length.)

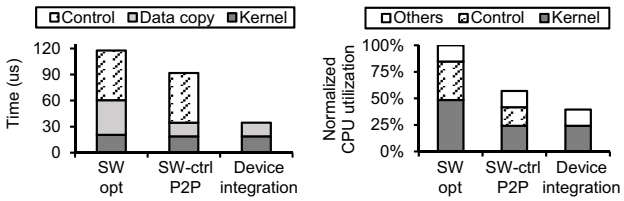
To reduce the host-side overhead of multi-device tasks, server architects have proposed direct D2D communication schemes: (1) *P2P communications* and (2) *device integration*. As shown in Figure 1b, P2P communication schemes allow a device to directly transfer data to another device's internal memory through modern interconnect technologies (e.g., PCIe, NVLink). They enable fast data transfers among peripheral devices, and significantly reduce host-side CPU and memory utilization to execute a multi-device task. For instance, [16]–[19] optimize various modern server applications exploiting PCIe P2P communications among off-the-shelf peripheral devices.

Integrated devices, on the other hand, tightly consolidate distinct devices into a single custom-built device as shown in Figure 1c. They tightly integrate different types of devices through internal interconnections. They can further optimize inter-device communications by introducing fast control paths with an internal device controller. For instance, QuickSAN [20] integrates storage controllers and network interfaces into a single device functioning as a fast storage area network. As another example, BlueDBM [21] deploys an FPGA board integrating storage controllers, network interfaces, and data processing units to accelerate SSD-based data analytics.

Unlike the existing D2D communication schemes, DCS-ctrl introduces a flexible and low-cost device-control mechanism implemented on an *independent* FPGA board. As shown in Figure 1d, DCS-ctrl enables fast, efficient, and flexible direct inter-device communications among off-the-shelf peripheral devices.

B. Motivation

1) Software-based Device Control - Performance Overhead: Existing PCIe P2P communication schemes still rely



(a) Latency (b) CPU utilization

Figure 3: Software overheads of multi-device communication

Application	Category	Intermediate processing
HDFS [22]	Data integrity	CRC32
	Compression	GZIP
	Encryption	AES256
Swift [23]	Data integrity	MD5
	Encryption	AES256
Amazon S3 [24]	Data integrity	MD5
	Compression	GZIP
	Encryption	AES256
Azure Blob [25]	Data integrity	MD5
	Encryption	AES256

Table II: Intermediate data processing for scale-out storage applications

on slow and CPU-inefficient software-based device-control mechanisms. As shown in Figure 2, complicated software components (user, kernel, device driver) initiate and terminate all device operations involved in the multi-device task, and they inevitably make the control paths to cross software-hardware and user-kernel boundaries frequently. With such slow and CPU-inefficient device control paths, software-based device control mechanisms suffer from the following performance overhead.

First, their CPU-dependent device-control routines and frequent software/hardware and user/kernel boundary crossings increase the overall latency of inter-device communications, and thus cannot fully exploit the performance potentials of high-performance devices. We measure and break down the software-side latency of a multi-device microbenchmark (SSD→GPU→NIC; sending data to network with hash computation on a GPU) into three main software components (Figure 3a). We use optimized I/O kernel stacks introduced in [9], [16], [17], [19], [21], [26] as our baseline to minimize the overestimation of the device-control overhead. Compared to the baseline (SW opt), software-controlled P2P communications (SW-ctrl P2P) avoid indirect data copies and thus reduce the data-copy latency. However, they still suffer from slow device-control routines (e.g., submitting and completing device commands, and boundary crossings).

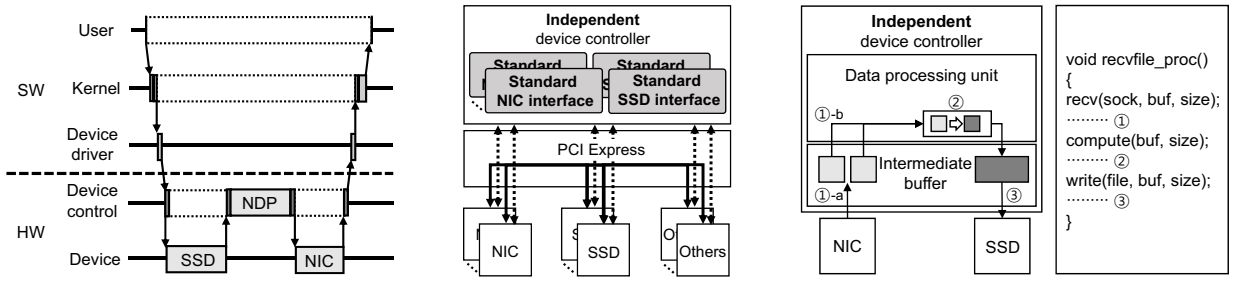
Second, frequently invoked device-control routines consume a significant portion of CPU resources, which incurs severe host-side resource contentions. We measure and break down the normalized CPU utilization of the same microbenchmark into software components (Figure 3b). Software optimizations and P2P communications seem to reduce host-side resource overheads by removing the indirect data copies and bypassing the complicated kernel routines (e.g., page cache and buffer management). However, as device-control routines relying on host-side CPUs still consume the non-trivial amount of CPU bandwidth, they prevent server applications from achieving scalable throughput with emerging devices.

2) Device Integration - Limited Flexibility: While integrated devices can achieve the optimal performance of inter-device communications (device integration in Figure 3), they have limited flexibility due to their aggregate integration. Such tight device integration makes it challenging to merge any commodity devices into a single device, while supporting direct control and data transfers among all consolidated devices. Their architectural limitation becomes increasingly critical as a wide spectrum of devices are introduced (e.g., [1], [2]). Furthermore, their inflexible integration easily leads to under-provisioning or over-provisioning problems in modern servers.

3) Intermediate Data Processing - Limited Applicability: Intermediate data processing is an essential application-level or kernel-level routine used to perform D2D communications. Such intermediate data processing between device operations prevents the server architects from aggressively applying direct D2D communications to existing server applications.

Many off-the-shelf devices which require device-specific intermediate processing cannot be used for direct data communications. For example, most network devices require packet header processing and scatter-gather (SG) processing as the part of direct inter-device communications. Network header information (e.g., flow identification, protocol, length) must be provided to network devices along with data to identify a remote client. To directly transfer received data from NICs to other devices, split data have to be gathered and placed in contiguous memory space.

As another example, application-level intermediate data processing also prevents server applications from adopting direct D2D communications. Table II shows application-level intermediate data processing in scale-out storage applications (e.g., data integrity, encryption, compression) for more stable, secure, and cost-effective services. As such intermediate data processing disables direct inter-device communication by forcing to use host-side CPUs and memory for data processing, existing D2D communication schemes cannot improve the overall performance of the server applications.



(a) Fast and CPU-efficient device control (b) Flexible device support (c) Intermediate data processing

Figure 4: Design goals of direct inter-device communications

C. Design Goals

Motivated by the limitations of existing direct D2D communications, we claim that inter-device communication schemes should satisfy the following design goals.

- **Fast and CPU-efficient device-control mechanism.** It should achieve fast and CPU-efficient D2D communications by performing device-control routines at the hardware level with minimum software intervention (Figure 4a).
- **High flexibility.** It should be flexible and cost-effective to support a large number of off-the-shelf devices by providing device-independent, standard device controllers (Figure 4b).
- **High applicability.** It should enable inter-device communications even with intermediate data processing required at application and kernel levels. For example, as shown in Figure 4c, it should merge network (①) and storage (③) operations and seamlessly perform intermediate data processing (②) between them.

III. DCS-ctrl: A Fast and Flexible Device-Control Mechanism

A. DCS-ctrl Architecture

We propose *DCS-ctrl*, a hardware-based device-control (HDC) mechanism for device-centric server (DCS) architecture, to achieve high performance, flexibility, and applicability of direct inter-device communications with off-the-shelf devices. The key idea of DCS-ctrl is to implement a hardware-based device-control mechanism on an independent and FPGA-based *HDC Engine*. As HDC Engine orchestrates all the control and data transfers among high-performance devices at the hardware level, DCS-ctrl architecture achieves the design goals as follows. First, optimizing both control and data paths at the hardware level and minimizing software intervention significantly reduce the latency of inter-device communications and the host-side CPU utilization. Second, implementing an independent and disaggregate device orchestrator with standard device controllers enables cost-effective and flexible D2D communications with a wide spectrum of off-the-shelf devices.

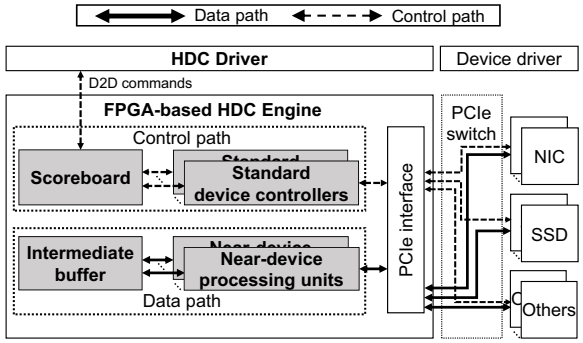


Figure 5: DCS-ctrl architecture

Third, adding a reconfigurable near-device data processing (NDP) unit provides more opportunities for direct D2D communications in existing server applications.

Figure 5 shows how DCS-ctrl architecture performs all control and data transfers among high-performance peripheral devices. *HDC Driver* is an optimized kernel module providing low-overhead software stacks for multi-device tasks. HDC Driver retrieves necessary metadata (e.g., source and destination devices, memory address, length) from OS kernel and forwards them to HDC Engine with unique D2D command IDs. After that, a *scoreboard* in HDC Engine keeps track of all user-requested D2D commands with the metadata, issues them to target device controllers, and then updates the states of the commands. *Standard device controllers* provide hardware interfaces between HDC Engine and other peripheral devices, control the target peripheral devices directly, and make the target devices transfer data to others through the PCIe switch. Moreover, when an application requires intermediate data processing between device operations, DCS-ctrl merges the decoupled device operations and offloads the intermediate data processing routines to NDP units in HDC Engine. *NDP units* then perform the required data processing seamlessly utilizing *intermediate buffers* in HDC Engine.

In the following sections, we introduce key architectural components of HDC Engine: a scoreboard, standard device

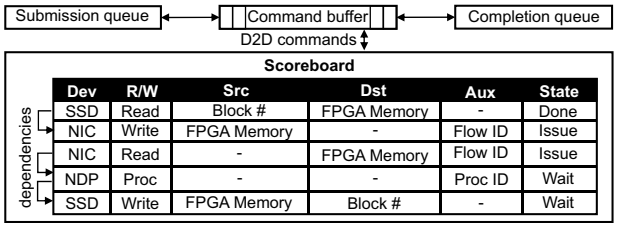
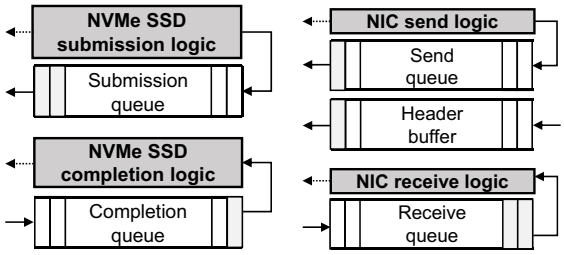


Figure 6: A scoreboard in HDC Engine to orchestrate peripheral devices for multi-device tasks



(a) SSD controller (b) NIC controller

Figure 7: Standard device controllers for (a) SSDs and (b) NICs. Each device controller includes a queue pair and device-control hardware logic.

controllers, and NDP units. We also explain how DCS-ctrl works with the existing software optimizations in Section III-E.

B. Scoreboard

To correctly perform direct control and data transfers of multi-device tasks without software intervention, HDC Engine handles all user-requested D2D commands using the scoreboard as shown in Figure 6. After fetching a user-requested D2D command from the command buffer, the scoreboard splits the D2D command into multiple device commands to control each target peripheral device, and stores them in scoreboard entries with necessary information. Each scoreboard entry holds a device ID (dev), read/write (r/w), source memory/block address (src), destination memory/block address (dst), auxiliary data (aux), and command state (state).

The scoreboard monitors current states of all fetched device commands and dynamically schedules them for user-requested multi-device tasks. When it determines there are no conflicts with incomplete device commands and target device controllers are ready, the scoreboard issues device commands to the corresponding device controllers and updates their states (i.e., ready→issue). If it finds that there are conflicts or target device controllers are not available (i.e., wait), it delays issuing those device commands until all dependent device operations are done. For instance, when HDC Engine performs SSD→NIC communication, the scoreboard does not issue the second NIC command until the first NVMe command is completed (i.e., issue→done). When it notices that all user-requested device commands are completed, the scoreboard delivers their unique IDs to the completion queue to interrupt HDC Driver.

C. FPGA-based and Disaggregate Device Controllers

To overcome the architectural limitation of integrated devices introduced in Section II-B2, DCS-ctrl exploits standard device controllers on independent and FPGA-based HDC Engine. They provide hardware interfaces between HDC Engine and other peripheral devices. Each device controller builds standard device commands of a target peripheral device, and submits and completes them leveraging PCIe P2P communications. In contrast to existing integrated devices, HDC Engine needs neither expensive device integration nor customized internal interconnection for fast hardware-based device-control mechanisms. Furthermore, due to FPGA's reconfigurability, device controllers in HDC Engine can easily support emerging devices and a large number of devices with low cost and hardware complexity.

In this paper, we implement device controllers for NVMe SSDs and 10-GbE NICs on HDC Engine to directly control them through PCIe. The NVMe SSD controller handles SSD-involved D2D communications, and directly submits and completes NVMe SSD commands leveraging PCIe P2P communications. As shown in Figure 7a, the NVMe SSD controller allocates HDC Engine memory for a submission and completion queue pair, and it implements hardware logic to build NVMe commands and to handle completion messages from the devices. In addition, it rings doorbell registers located in NVMe SSD devices to notify the number of newly submitted or completed commands.

Similarly, the 10-GbE NIC controller handles NIC-involved D2D communications, and directly submits and completes Broadcom 10-GbE NIC commands (Figure 7b). When HDC Engine needs to deliver data to a NIC device, the NIC controller generates TCP/IP packet headers and stores them in the header buffer. It also builds NIC commands, puts them in a send queue, and rings the registers allocated in the network device to notify the packet transmission. When the NIC controller receives a packet through the network, it parses the received packet headers and messages to identify a target connection and destination location (e.g., page cache address or block address) requested by applications.

D. Near-Device Processing Units

In contrast to the prior work [16] which does not consider device-specific nor application-level intermediate data processing, DCS-ctrl maximizes the opportunities and benefits of D2D communications with HDC Engine's NDP units.

We find that intermediate processing in scale-out storage applications can be categorized into three groups: data

Processing units	LUTs	Registers	Maximum clock freq.	Throughput per unit
MD5 [33]	3.0% (8970)	0.69% (4180)	130MHz	0.97Gbps
SHA1 [34]	3.49% (10760)	1.13% (6848)	235MHz	1.10Gbps
SHA256 [35]	4.28% (13090)	1.23% (7480)	130MHz	0.80Gbps
AES256 [36]	3.52% (10689)	0.99% (6000)	>250MHz	40.90Gbps
CRC32 [37]	0.03% (93)	0.01% (53)	>250MHz	10Gbps
GZIP [38]	5.36% (16273)	2.09% (12718)	178MHz	100Gbps

Table III: Estimated Virtex 7 FPGA resource utilization and maximum operating clock frequency for commonly required intermediate processing units

integrity check, data encryption, and compression. (Table II). Such intermediate processing can be easily offloaded to an FPGA and achieves the high performance with the low FPGA resource overhead. To check the FPGA resource requirement for NDPs, we implement and synthesize them on Xilinx Virtex 7 FPGA using Xilinx Vivado Design Suite [27]. Also, we measure the highest clock frequency that passes the timing analysis.[1] We then calculate the IP core maximum throughput based on its bus width and the number of IP cores needed to reach 10 Gbps data processing throughput. As Table III shows, to achieve the 10-Gbps throughput, on average, only 3.28% slice LUT and 1.02% slice register of a Virtex 7 FPGA are required.[2]

To support intermediate processing on HDC Engine, NDP units utilize FPGA on-board memory as intermediate buffers and executes the data processing between device operations. For example, to provide secure services in scale-out storage applications, NDP units execute encryption routines after receiving data from storage devices. Then HDC Engine performs D2D communications between a network device and HDC Engine to move processed data. Furthermore, NDP units perform device-specific intermediate processing to support a wide spectrum of off-the-shelf devices. For example, NDP units parse packet headers and gather packet payloads to place them into consecutive memory space for direct NIC↔SSD communications.

E. Software Optimization

DCS-ctrl adopts existing software optimizations (e.g., [9], [16], [17], [19], [21], [26]) in HDC Driver to fully utilize high-performance devices and further reduce host-side overheads. For instance, DCS-ctrl bypasses page cache and I/O buffer management operations of inter-device communications (e.g., Linux direct I/O). Likewise, it bypasses socket buffer management operations as it reuses dedicated packet send and receive buffers in HDC Engine. Figure 8 shows the kernel-side CPU utilization of Linux and DCS-ctrl in simple direct communications between a SSD and a NIC. The result indicates DCS-ctrl significantly reduces kernel-side CPU utilization as much as other existing software optimization approaches do.

[1]For realistic throughput estimation, we do not use clock frequencies higher than 250 MHz even if the IP core passes the timing analysis.

[2]Resource utilization (i.e., LUTs and Registers) belongs to multiple instances of non-pipelined IP cores or a single instance of fully pipelined IP cores to achieve the 10-Gbps throughput.

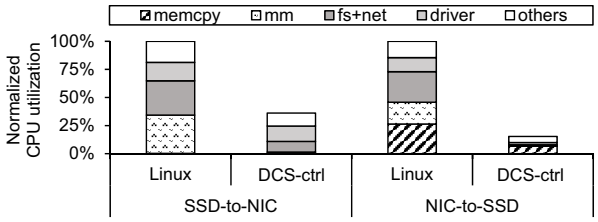


Figure 8: DCS-ctrl software optimization

IV. Implementation

We implement HDC Engine prototype on a Xilinx Virtex-7 FPGA VC707 board [28], and it is connected with off-the-shelf Intel NVMe SSDs [29], Broadcom 10-Gbps Ethernet cards [30], and NVIDIA GPUs [31] through a PCIe switch [32]. We also develop a Linux kernel module and user-level library to apply DCS-ctrl to existing scale-out storage applications. In the following sections, we explain the implementation details of DCS-ctrl (Figure 9).

A. HDC Library

HDC Library provides Linux's `sendfile`-like APIs to allow applications to exploit various scenarios of direct D2D communications. We replace D2D-eligible routines including intermediate data processing with a single API call. These APIs receive file descriptors of the D2D-involved devices as arguments and require function identifications and auxiliary data for intermediate processing. Each API defined in HDC Library internally invokes `ioctl` to initiate HDC Driver routines for direct D2D communications.

HDC Library implementation has two major benefits by using file descriptors. First, it helps DCS-ctrl to cooperate with the existing Linux kernel. For example, DCS-ctrl uses file and socket descriptors and existing kernel APIs when it needs to retrieve block addresses and TCP/IP connection information. Second, DCS-ctrl leverages existing security models by checking the permission of file and socket descriptors before direct D2D communications. Thus unpermitted storage or network devices cannot be involved in direct inter-device communications.

B. HDC Driver

We develop HDC Driver to communicate with HDC Engine and provide optimized software stacks. HDC Driver

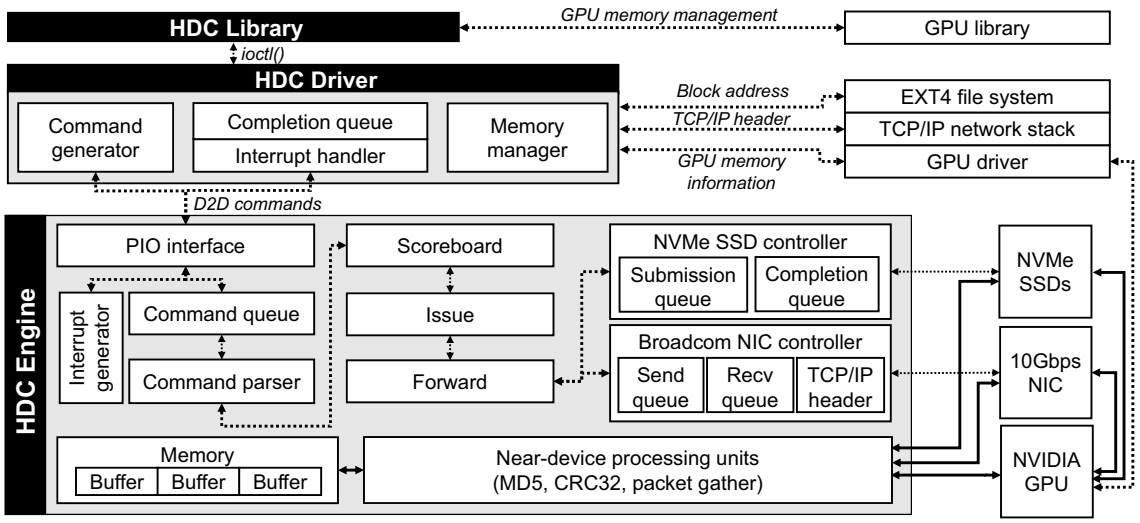


Figure 9: DCS-ctrl implementation

Virtex 7 FPGA Resource Utilization	
LUTs	116344 / 303600 (38%)
Registers	91005 / 607200 (15%)
BRAMs	442 / 1030 (43%)
Power	5.57 Watts

Table IV: HDC Engine's device controllers on Virtex-7 resource utilization

interacts with the existing kernel file system and TCP/IP network stacks to find necessary metadata such as block addresses and TCP/IP connection information to properly orchestrate D2D-involved devices. It also generates and forwards D2D commands, and handles interrupts from HDC Engine. In addition, to reduce kernel-side overhead, we apply existing software optimizations to our HDC Driver implementation. Especially, it bypasses host-side management routines for page caches and socket buffers. However, simply bypassing page caches violates the data consistency when the latest data are located in page caches. Thus, for the data consistency, HDC Driver identifies the address of latest data by interacting with the kernel virtual file system (VFS), and modifies the D2D commands with the appropriate location.

Moreover, we extend existing Linux generic NVMe and Broadcom NIC device drivers to dedicate device queue pairs (e.g., NVMe submission/completion queues and NIC send/recv queues) in HDC Engine and to minimize host-side memory accesses from those devices.

C. HDC Engine

We implement Intel NVMe SSD and Broadcom 10-GbE NIC controllers on a Xilinx Virtex-7 VC707 board [28] using Xilinx Vivado Design Suite [27]. Table IV shows the FPGA resource utilization of the current implementation. Note that the FPGA has enough remaining resources to add NDP units for intermediate data processing (Table III).

In addition to the key architectural components introduced in section III, we implement PCIe and host interfaces on HDC Engine to interact with HDC Driver. The host interface includes the 64-entry command queue (4KB) and the command parser to receive D2D commands from HDC Driver and deliver them to the scoreboard. When HDC Engine finds that all user-requested D2D commands are completed, it interrupts HDC Driver through the interrupt generator. Also, for the simple implementation, HDC Engine issues D2D commands in a requested order and notifies HDC Driver of their completions in the same order.

Standard device controllers for the NVMe SSD and the Broadcom 10-GbE NIC are implemented on the FPGA board as well. We allocate FPGA on-chip BRAMs for the device queue pairs to enable fast access of the peripheral devices. Even we find that transferring 4KB for every NVMe and NIC command is enough to fully utilize the off-the-shelf NVMe SSDs and 10-GbE NICs, we exploit bulk-transfer mechanisms of the existing devices to further improve the throughput of direct D2D communications. For instance, we utilize the large send offload (LSO) features commonly supported by modern network cards, and use a PRP list [40] to transfer multiple blocks with a single NVMe command.

Also, to support D2D communications whose source is a NIC device, HDC Engine gathers split packets as NIC-specific intermediate processing. Packet-gathering hardware logic removes the packet headers and put the split data into the continuous memory space for following D2D communications and intermediate processing. However, such NIC-specific intermediate processing can be offloaded to other network devices which support header-split features [39].

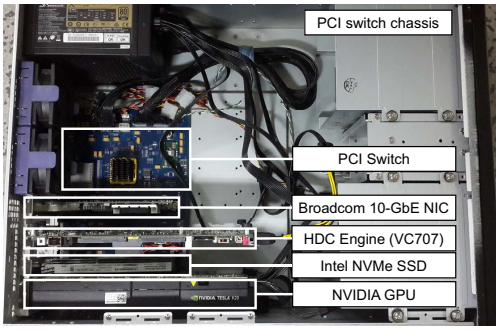


Figure 10: DCS-ctrl prototype (showing a single node)

Lastly, we utilize on-board 1GB DDR3 DRAMs as intermediate buffers for intermediate processing and packet recv buffers for NIC devices. To easily manage large memory space, the intermediate buffers and packet recv buffers are chunked into multiple fixed-size blocks (64KB).

V. Evaluation

In this section, we describe our experimental setup and evaluate DCS-ctrl. We first present that DCS-ctrl minimizes the latency of inter-device communications through the hardware level control path optimization. Then, we show that DCS-ctrl reduces the CPU utilization for inter-device communication and achieves higher scalability with scale-out storage workloads.

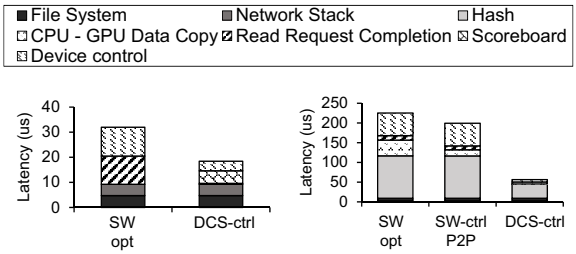
A. Experimental Setup

Table V summarizes the system setup used to evaluate DCS-ctrl. We use high-performance I/O devices (i.e., an NVMe SSD and a 10-Gbps NIC[3]) and connect them with a high-bandwidth PCIe switch. Figure 10 shows our prototype with the devices and the PCIe switch. We use a two-node setup equipped with the DCS-ctrl prototype for evaluations.

To present the effect of each optimization (i.e., kernel optimization, direct inter-device communication, control path optimization), we compare DCS-ctrl with two baseline designs: *Software optimization*, and *Software-controlled P2P*. Software optimization is the baseline system which uses the optimized software to minimize latency and CPU utilization, but all data transfer go through CPU memory. Software-controlled P2P uses optimized software and leverages direct inter-device communication. However, its control path is not optimized and a CPU still controls all device operations. Note that the direct communication between the NVMe SSD and the NIC is not possible. Both devices do not allow other devices to access their internal memory, so it is impossible to transfer data directly [40], [41]. In that case, software-controlled P2P cannot perform D2D communications.

Host	
CPU	Intel Xeon CPU E5-2630 (2.30 GHz, LLC 15 MB)
OS	Centos 6.5 (kernel version: 2.6.32-504.el6)
Device	
SSD	Intel 750 Series SSD 400GB (read: 17.2 Gbps, write: 7.2 Gbps)
NIC	Broadcom Corporation NetXtreme II BCM57711 10-Gbps NIC
GPU	NVIDA Tesla K20m
PCIe Switch	Cyclone Microsystems PCIe2-2707 (PCIe Gen2 switch, # of slots = 5, switch bandwidth = 80 Gbps)
HDC Engine	Xilinx Virtex 7 VC707 board

Table V: Details of our experimental setup



(a) SSD→NIC (b) SSD→Processing→NIC

Figure 11: Latency breakdown of inter-device communications

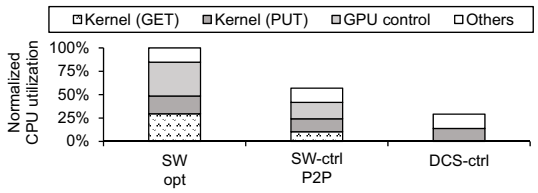
B. Inter-device Communication Latency

We run two microbenchmarks to measure the latency of inter-device communications. The first microbenchmark, SSD→NIC, reads data from an NVMe SSD and sends it to a NIC. The second microbenchmark, SSD→Processing→NIC, processes data before sending it to the NIC. The microbenchmark performs an MD5 checksum for the intermediate processing to check the data integrity. The baseline designs use GPUs to accelerate calculating the checksum and the checksum result is fetched into the CPU memory. Note that using a CPU to calculate the checksum might avoid the access to GPU, but it decreases the server throughput due to the increased CPU utilization.

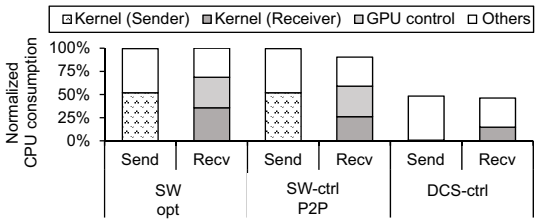
Figure 11a shows the inter-device communication latency decomposition of DCS-ctrl and the baseline designs for the SSD→NIC benchmark. Software optimization design has slow device controls due to its frequent crossings of user/kernel and software/hardware boundaries. On the other hand, DCS-ctrl optimizes the control paths and nearly eliminates such overheads (i.e., read request completion) with the minimal scoreboard overhead.

The result of the SSD→Processing→NIC microbenchmark (Figure 11b) presents the effect of direct inter-device

[3]Note that the effective bandwidth of the NIC is around 9 Gbps due to packet overheads.



(a) Swift



(b) HDFS

Figure 12: CPU utilization breakdown of scale-out storage applications

communication and the full control path optimization with NDP. Compared to the SSD→NIC microbenchmark, software optimization shows long latency due to the GPU control (e.g., launch a GPU kernel) and the data transfer between the GPU and CPU memory. Software-controlled P2P optimizes data path and shortens the latency of data copy between CPU and GPU memory. However, it still suffers from the long latency of the unoptimized control path. DCS-ctrl uses NDP units instead of GPUs to minimize the overall latency by avoiding unnecessary CPU intervention of controlling GPUs. As a result, it reduces the software latency by 72% compared to software-controlled P2P.

In summary, the microbenchmark results prove that DCS-ctrl offers a significant latency reduction over software-controlled P2P which controls all devices using CPUs. Also, NDP brings additional benefits when intermediate processing is required.

C. Scale-out Storage Workloads

As scale-out workloads, we use OpenStack Swift [23], an open-source object storage system, and Hadoop Distributed File System (HDFS) [22], a distributed file system. They perform compute-intensive data integrity checks (i.e., MD5 in Swift and CRC32 in HDFS) during data communications.

1) Object Storage - Swift: To evaluate the benefit of DCS-ctrl with Swift, we measure the CPU utilization on DCS-ctrl and the baseline designs when a client sends REST requests such as `PUT` and `GET`. The `PUT` and `GET` requests eventually make the storage server to send and receive files with the intermediate MD5 processing. DCS-ctrl and baseline designs accelerate the intermediate processing using NDP units and GPUs, respectively. To model a realistic user behavior, we generate user requests with the parameters (e.g., `PUT`/`GET` ratio, file size distribution) in [42] obtained from the real-world data-serving service. We also use the Poisson process to model request arrivals, and carefully scale the arrival rate until it saturates the bandwidth of target servers.

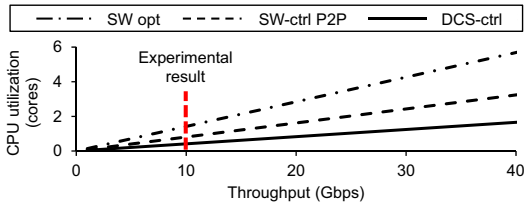
We measure and break down CPU utilization of DCS-ctrl and baseline designs with the same throughput (Figure 12a). Kernel (GET) and Kernel (PUT) refer to CPU usage when the kernel and HDC drivers handle GET and PUT requests, respectively. Software-controlled P2P alleviates GPU data copy overheads by eliminating redundant data transfers between CPU memory and the GPU when serving to GET operations. Also, direct inter-device communication reduces the kernel overheads because data copies between user and kernel memory do not occur. For PUT operations, software-controlled P2P cannot remove the GPU control overheads due to the unavoidable data gathering process. DCS-ctrl entirely removes the accelerator control overhead by using NDP units and further reduces the kernel overhead by optimizing the control path.

2) Distributed File System - HDFS: To evaluate designs with HDFS, we run *HDFS balancer* as a load generator. HDFS balancer distributes skewed data across nodes in HDFS. During the distribution, a *sender* reads data from an NVMe SSD and sends it to a *receiver* without the integrity check. On the opposite side, the receiver receives the data and computes a CRC32 checksum of the data. DCS-ctrl and baseline designs accelerate the checksum calculation using NDP unit and GPUs, respectively. After the receiver checks the checksum, it stores the data into an NVMe SSD.

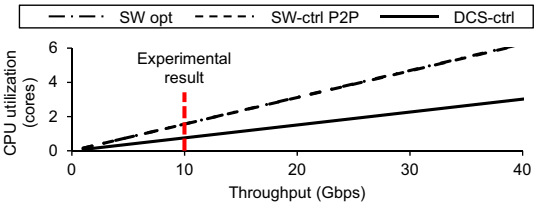
While HDFS balancer is running, we estimate the CPU utilization of data sender and data receiver. Figure 12b shows the CPU utilization breakdown of HDFS on DCS-ctrl and the baseline designs at the same bandwidth. There is little opportunity to benefit from software-controlled P2P because senders do not use GPUs and receivers suffer from the data gathering problem for NIC→GPU communication. It is observed in Figure 12b that software-controlled P2P cannot improve the performance of HDFS. On the other hand, DCS-ctrl improves the performance of HDFS by reducing the CPU utilization of the sender and enabling direct inter-device communication when receiving data.

3) Scalability: Figure 13 shows our estimation of the maximum throughput. We measure the throughput and CPU utilization using a 10 Gbps NIC (a red vertical line) and calculate the required number of cores based on the measured result. For the estimation, we assume a 40-Gbps NIC, six NVMe SSDs, and a single 6-core Intel Xeon CPU.

For Swift, Figure 13a shows that DCS-ctrl provides the same throughput to baseline designs with lower CPU utilization. In the HDFS case, baseline designs cannot serve 40 Gbps of throughput using one CPU (Figure 13b). Software-controlled P2P shows little performance improvement because of the lack of opportunity for direct inter-device communication. Since DCS-ctrl requires only three



(a) Swift



(b) HDFS

Figure 13: Estimated CPU utilization with high-performance devices

or fewer cores to fully utilize the 40-Gbps NIC with HDFS and Swift, it has chances to deliver higher throughput as more I/O devices are mounted on the server. Assuming the lack of CPU resources, DCS-ctrl provides 1.95× and 2.06× higher throughput compared to software-controlled P2P optimization with Swift and HDFS, respectively. In summary, DCS-ctrl not only shortens the latency of inter-device communication but also minimizes the host-side overhead, achieving high throughput and scalability.

VI. Related Work

Software Optimization. Software optimizations can help minimize device access overheads. For instance, Moneta-D [14] and NVMeDirect [43] use kernel bypassing when accessing storage devices. Similarly, mTCP [15] offers user-level I/O stack to bypass kernel when accessing NICs. Although they reduce access overheads, they are only applicable to a specific type of devices which limits their applicability. On the other hand, Arrakis [44] exploits the hardware virtualization support to partially bypass kernel. FLASH [45] suggests a custom node controller to achieve high scalability and low I/O overhead. However, both do not consider D2D communication.

Moneta [9] introduces various optimizations on the kernel software stack. Recent studies like SPIN [17] and ReFlex [46] integrate kernel I/O stack of different devices such as SSD-GPU or SSD-NIC. These ideas are especially beneficial in case of certain file access patterns such as short sequential reads because they can take advantage of the OS read-ahead mechanism [17]. Also, Falcon [47] brings in the idea of I/O batching and per-drive I/O processing into the kernel for fast random accesses in a multi-SSD environment. These optimizations can be adopted to HDC Driver as they are orthogonal to our research.

Device Integration. Integrating peripheral devices into a single device enables high-bandwidth and low-latency data transfers within the integrated devices. QuickSAN [20] integrates SSDs and NICs to minimize host-side resource overheads and access to remote storage faster. Similarly, BlueDBM [21] integrates NAND flash and network interfaces into an FPGA-based system, achieving high-bandwidth data transfer between flash devices and fast data processing using an accelerator close to the flash devices. Unfortunately, such work which tightly consolidates devices limits its applicability because they provide benefits only when the processed data strictly follow the pre-defined data path. Besides, it is challenging to add new devices.

Adding a processor to a peripheral device allows intermediate processing. Willow [48] enables a configurable SSD interface, and Biscuit [49] provides a programming framework that allows users to run applications on SSDs. They both allow developers to program data-intensive applications with a minimal data transfer between SSDs and the host, and provide high performance using intermediate processing. FlexNIC [26] moves some of the packet processing into a NIC. However, these works do not consider D2D communication and only are applicable to a certain device type.

Commercial products are also coming up with the idea of device integration. For example, AMD RADEON PRO SSG [50] joins a GPU with an SSD in order to give a large memory for the GPU. In addition, Mellanox's BlueField [51] puts a NIC, a PCIe switch and small processor cores on one chip. It supports NVMe over fabric [52] and Mellanox OFED GPUDirect RDMA [53] so that it can perform D2D communication between NIC and GPU, or NIC and NVMe. Nevertheless, they do not support every possible D2D scenarios, and their control path is not optimized due to a CPU involvement. Oden *et al.* [54] modify the device drivers and the libraries of GPUs and Infiniband network cards to allow GPUs to control network devices. Their work shows that offloading the network stack into GPUs is inefficient due to the low single-thread performance of GPUs. On the other hand, we employ an FPGA for fast device control and show significant performance improvement compared to a host-centric server.

PCIe P2P Communication. PCIe is a widely used interconnection technology to connect peripheral devices to each other and to the host. Neuwirth *et al.* [55] propose EXTOLL interconnect using PCIe to directly connect EXTOLL NICs and Xeon Phi coprocessors. Morpheus [18] also utilizes PCIe to connect NVMe SSDs and other PCIe devices. Especially, Morpheus can perform in-flight data processing on the data path by using embedded cores on an NVMe SSD. However, they do not provide flexible data path because they only support a specific pair of devices.

Exposing the memory of a PCIe device also enables

direct inter-device communication. NVIDIA GPUDirect RDMA [56] and AMD DirectGMA [57] provide a set of functions to expose GPU memory to PCIe address space. As one example, GPUnet [58] proposes the network programming model on GPU using Mellanox OFED GPUDirect RDMA [53]. In this way, other devices can directly access the GPU memory and perform direct inter-device communication. However, these work are not beneficial when intermediate processing is required.

To achieve high flexibility and support diverse peripheral devices, DCS [16] orchestrates peripheral devices to transfer data directly to each other. DCS enables partially control-optimized direct data transfer between SSDs and NICs, and allows a NIC to be either a DMA master or a DMA slave depending on the opponent device. GPUDirect Async [59] enables a direct data transfer between GPUs and NICs with partial control optimization which frees CPU from the control path between GPUs and NICs. Unfortunately, both have limited coverage of scenarios in D2D communications. Furthermore, DCS has limited applicability as it neither provides near-device processing nor supports bi-directional data transfer.

VII. Conclusion

In this paper, we proposed DCS-ctrl, a hardware-based device-control mechanism to significantly improve the performance of existing D2D communication schemes. The key idea of DCS-ctrl is to implement a low-cost and flexible device orchestrator, HDC Engine, while maximizing the opportunity and applicability of direct inter-device data communication using near-data processing at the same time. For evaluation, we implemented our DCS-ctrl prototype on a real machine, and our results show that DCS-ctrl significantly outperforms the state-of-art D2D communications schemes in performance and flexibility. To the best of our knowledge, DCS-ctrl is the first approach to optimize both control and data paths using an independent and FPGA-based device orchestrator with near-data processing supports.

Acknowledgment

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1503-05.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proc. 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *Proc. 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[3] M. Harris, "NVIDIA DGX-1: The Fastest Deep Learning System," https://devblogs.nvidia.com/parallelforall/dgx-1-fastest-deep-learning-system/.

[4] "Intel™ SSD Data Center Family for PCIe," https://www.intel.com/content/www/us/en/solid-state-drives/intel-ssd-dc-family-for-pcie.html.

[5] "Samsung NVMe SSD 960 PRO/EVO," http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960.html.

[6] "Mellanox Ethernet Adapters," http://www.mellanox.com/page/ethernet_cards_overview.

[7] "Mellanox InfiniBand Adapters," http://www.mellanox.com/page/infiniband_cards_overview.

[8] "NVIDIA Launches World's First High-Speed GPU Interconnect, Helping Pave the Way to Exascale Computing," http://nvidianews.nvidia.com/news/nvidia-launches-world-s-first-high-speed-gpu-interconnect-helping-pave-the-way-to-exascale-computing, 2014.

[9] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, "Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2010.

[10] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A New Programming Interface for Scalable Network I/O," in *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving Network Connection Locality on Multicore Systems," in *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[12] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls," in *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[13] J. Yang, D. B. Minturn, and F. Hady, "When Poll is Better than Interrupt," in *Proc. 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[14] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing Safe, User Space Access to Fast, Solid State Disks," in *Proc. 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[15] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2014.

[16] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim, "DCS: A Fast and Scalable Device-Centric Server Architecture," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2015.

[17] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs," in *USENIX Annual Technical Conference (ATC)*, July 2017.

[18] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: creating application objects efficiently for heterogeneous computing," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

[19] "Donard: A pcie peer-2-peer kernel patch and library that builds on top of nvm. express," https://github.com/sbates130272/donard.

[20] A. M. Caulfield and S. Swanson, "QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2013.

[21] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: an appliance for big data analytics," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2015.

[22] "Apache Hadoop," http://hadoop.apache.org/.

[23] "OpenStack Swift," https://docs.openstack.org/swift.

[24] "Amazon S3," http://docs.aws.amazon.com/AmazonS3/latest/dev.

[25] "Microsoft Azure Storage," https://docs.microsoft.com/en-us/azure/storage/.

[26] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2016.

[27] Xilinx, Inc., "Vivado Design Suite," https://www.xilinx.com/products/design-tools/vivado.html.

[28] "Xilinx VC707 Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html.

[29] "Intel SSD 750 Series," http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-750-series.html.

[30] "Broadcom NetXtreme II BCM57711 Dual-Port Direct Attach 10 GbE PCI Express Network Interface Card with TOE and iSCSI Offload," http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/broadcom-netxtreme-57711-spec-sheet.pdf.

[31] "TESLA K20 GPU ACCELERATOR," https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf.

[32] "PCIe2-2707 PCIe Gen2 Five Slot Expansion System," http://cyclone.com/products/expansion_systems/600-2707.php.

[33] "Open-source MD5 hash HDL code," https://github.com/stass/md5_core.

[34] "Open-source SHA1 hash HDL code," https://github.com/secworks/sha1.

[35] "Open-source SHA256 hash HDL code," http://opencores.org/project,sha256_hash_core.

[36] "Open-source AES encryption HDL code," http://opencores.org/project,tiny_aes.

[37] "Open-source CRC hash HDL code," http://opencores.org/project,ultimate_crc.

[38] "GZIP data compression core," https://www.xilinx.com/products/intellectual-property/1-7aisy9.html#productspecs.

[39] "Header-Data Split Architecture," https://docs.microsoft.com/en-us/windows-hardware/drivers/network/header-data-split-architecture.

[40] "NVM Express Specification," http://www.nvmexpress.org/wp-content/uploads/NVM_Express_1_2_1_Gold_20160603.pdf.

[41] Broadcom Corporation, "Highly Integrated Media Access Controller Programmer's Guide," https://docs.broadcom.com/docs/1211168564430?eula=true.

[42] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 481–494.

[43] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds." in *HotStorage*, 2016.

[44] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *ACM Transactions on Computer Systems (TOCS)*, 2016.

[45] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz *et al.*, "The stanford flash multiprocessor," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 302–313.

[46] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash? local flash," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 345–359.

[47] P. Kumar and H. H. Huang, "Falcon: Scaling io performance in multissd volumes," in *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association*, 2017, pp. 41–53.

[48] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A User-Programmable SSD," in *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[49] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *Proc. 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

[50] Advanced Micro Devices, Inc., "AMD RADEON PRO SSG," https://pro.radeon.com/en/product/pro-series/radeon-pro-ssg/.

[51] Mellanox Technologies, "BlueField™ Smart NIC," http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[52] Advanced Micro Devices, Inc., "NVMe over fabric," http://www.mellanox.com/blog/2016/06/nvme-over-fabrics-standard-is-released/.

[53] Mellanox Technologies, "Mellanox OFED GPUDirect RDMA," http://www.mellanox.com/related-docs/prod_software/PB_GPUDirect_RDMA.PDF.

[54] L. Oden and H. Fröning, "Infiniband verbs on gpu: a case study of controlling an infiniband network device from the gpu," *The International Journal of High Performance Computing Applications*, vol. 31, no. 4, pp. 274–284, 2017.

[55] S. Neuwirth, D. Frey, M. Nuessle, and U. Bruening, "Scalable Communication Architecture for Network-Attached Accelerators," in *Proc. 21st IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[56] Mellanox Technologies, "NVIDIA GPUDirect™ Technology – Accelerating GPU-based Systems," http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf.

[57] Advanced Micro Devices, Inc., "DirectGMA on AMD's FirePro GPUs," https://www.amd.com/Documents/SDI-tech-brief.pdf.

[58] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[59] E. Agostini, D. Rossetti, and S. Potluri, "Offloading communication control logic in gpu accelerated applications," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 248–257.