Doctoral Dissertation

An FPGA-based Device Control Mechanism for Fast Inter-device Communication

Jaehyung Ahn (안 재 형)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2019

디바이스 간 빠른 통신을 위한 FPGA 기반 디바이스 제어 기술

An FPGA-based Device Control Mechanism for Fast Inter-device Communication

An FPGA-based Device Control Mechanism for Fast Inter-device Communication

by

Jaehyung Ahn
Department of Computer Science and Engineering
Pohang University of Science and Technology

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Computer Science and Engineering

Pohang, Korea
12. 14. 2018
Approved by
Jong Kim (Signature)
Academic advisor

An FPGA-based Device Control Mechanism for Fast Inter-device Communication

Jaehyung Ahn

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

12. 14. 2018

Committee Chair	Jong Kim	(Seal)
Member	Jangwoo Kim	(Seal)
Member	Jae-Joon Kim	(Seal)
Member	Gwangsun Kim	(Seal)
Member	Jae Wook Lee	(Seal)

DCSE 안 재 형. Jaehyung Ahn
20131000
An FPGA-based Device Control Mechanism for Fast Inter-device Communication,
디바이스 간 빠른 통신을 위한 FPGA 기반 디바이스 제어 기술
Department of Computer Science and Engineering , 2019, 86p, Advisor : Jong Kim. Text in English.

ABSTRACT

Modern high-performance servers employ a large number of high-throughput peripheral devices to meet the demands of server applications, which process a large amount of data in a short time. However, as the number of peripheral devices increases, the host CPU and memory becomes extremely busy only for executing the complex kernel routines to control the devices and handle incoming interrupts from them. Some architects suggested bypassing the kernel routines to alleviate the control overheads. However, they failed to leverage the full performance potential of the peripheral devices. In such architectures, the host CPU is still controlling peripheral devices and the host memory bandwidth is consumed in handling data movements. Other architects exploit direct device-to-device (D2D) communication to reduce the overhead on both the host CPU and the host memory bandwidth. Unfortunately, existing D2D communications suffer

from low flexibility, because they can not perform the D2D communication if an intermediate data processing decouples device operations.

To address the issues, we propose a fast, scalable, and flexible FPGA-based device orchestration method. The key idea is to implement a low-overhead, but flexible device-control mechanism on an independent FPGA-based orchestrator. The proposed method achieves high performance, scalability and applicability as follows. First, the orchestrator has standard device interfaces to enable direct inter-device transfers between any commodity peripheral devices. Second, the orchestrator handles all device controls and data transfers within the server to achieve the performance and save host resources. Third, the orchestrator performs intermediate data processing on its FPGA block to more actively invoke D2D communications.

The evaluation results show that our scheme reduces the host overhead and improves warehouse scale workloads. Our scheme also shows higher scalability than host-centric server architecture and takes full advantage of emerging high-throughput peripheral devices.

Contents

List of Tables

List of Figures

I. Introduction

When a modern server deploys an increasing number of high-performance peripheral devices, the host CPU and memory become extremely busy only to execute complex kernel routines frequently invoked by device operations. For such server architectures, the overall server performance is throttled by the latency of device-handling software components and the available bandwidth of the CPU and the memory for the target operations. To mitigate the overhead, kernel routines have been optimized through generations, but the overhead still exists and is expected to get worse with the introduction of faster devices in the future.

To address the issue, previous studies proposed many solutions which can be categorized into three groups: software-based solutions, hardware-based solutions, and PCIe P2P communication-based solutions. Software-based solutions simplified and optimized existing kernel stacks or aggressively implemented user-level stacks. However, they focus on only single-device tasks and do not address the multi-device tasks. As a result, the host CPU and memory still suffer from the high overhead.

On the other hand, hardware-based solutions depend on custom-build devices tightly integrating heterogeneous devices. Their consolidated devices can perform direct data and control transfers without software intervention. Unfortunately, such custom-built and integrated devices can achieve neither cost-effective device allocation nor flexibility as they are expensive and their designs limit the

supported device types. Their architecture limitation will become increasingly critical when more diverse devices are introduced and server applications require a wide spectrum of device combinations.

Lastly, PCIe P2P communication-based solutions enable data transfers between devices bypassing the host CPU and the memory using the PCIe interconnect and reduce the host overhead. However, they 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 the direct D2D communications.

To overcome the problems of the previous studies and exploit the full potential of peripheral devices, we propose a fast, scalable and flexible architecture for device control using an FPGA-based orchestrator [1, 2]. First, the orchestrator has device interfaces and a scoreboard to performs device-control routines. The scoreboard schedules the device operations involved in a multi-device task, and issues each device command to the corresponding device interfaces. Device interfaces then interact with the other PCIe devices through PCIe P2P communications without any software interference. This hardware-based device control mechanism significantly reduces both the latency of D2D communications and the host-side CPU and memory utilization.

Second, the orchestrator is independent from the devices and enables D2D

communications among off-the-shelf devices using standard device interfaces. The orchestrator performs corresponding kernel-level device-control routines on the hardware. By implementing low-cost standard device interfaces, our proposed architecture supports a large number of heterogeneous devices and allows per-device upgrades as needed, which makes the orchestrator more flexible than integrated device schemes.

Third, the orchestrator can schedule and merge decoupled device operations in existing server applications so that it more aggressively applies direct D2D operations. With data processing units implemented in an FPGA, the orchestrator can further increase the applicability of direct D2D operations by performing intermediate data processing between decoupled device operations. As a result, the configurable near-device processing (NDP) capabilities further improve the performance of modern server applications.

The rest of the thesis is as follows. First, we show that the proposed idea reduces almost all the host-side overhead by controlling devices using an FPGA-based orchestrator. To focus on the performance improvement, we decided to implement the network interface on the orchestrator and extremely minimized the device control overhead. The evaluation result shows that our architecture reduces the execution time of warehouse-scale workloads by 27.52%.

Second, we improve the flexibility and applicability of the orchestrator. We upgrade the proposed orchestrator by including standard network device interfaces and the configurable near-device processing units. We compare the proposed architecture to existing the D2D schemes and show that the proposed architecture

achieves the performance, scalability, and flexibility at the same time. The proposed architecture can be applied to various applications and reduces the latency of D2D operations by 72% leveraging the near-data processing.

Lastly, we introduce the related works that optimize multi-device tasks and compare them to our scheme. Many server architects have proposed various D2D communication schemes, however, they failed to leverage full potential of the high-performance devices or suffer from lack of flexibility. On the other hand, our proposed scheme achieves a fast and scalable device control mechanism while providing flexibility and applicability at the same time.

II. Reducing the Host Overhead for Device Control Using Hardware-Based Solution

2.1 Introduction

A traditional server architecture employs fast CPUs to improve the performance of compute-intensive workloads and run an operating system (OS) to support relatively slow I/O devices using memory accesses and interrupts. The main goal of such *host-centric server* design is to make compute-intensive workloads use maximum CPU bandwidth and spend only a small amount of CPU bandwidth for I/O devices. The host-centric server design has been indeed effective to improve the server performance and manage conventional I/O devices such as hard disk drives and 1-Gbps NICs.

However, as emerging workloads (e.g., MapReduce, object storage, and secure big-data serving) become both data-intensive and compute-intensive, the performance of modern servers has become heavily dependent on the performance of I/O devices as well as CPUs. To improve the I/O performance, various emerging devices (e.g., NVM storage and high-bandwidth NICs) and interconnection technologies (e.g., PCIe) have been introduced and widely deployed in modern host-centric servers. In that way, modern host-centric servers have improved the performance of the emerging workloads.

However, such host-centric server architectures fail to exploit the full po-

tential of emerging device technologies. The fundamental problem is that the deployment of the emerging devices violates the main design assumption of the host-centric architecture, i.e., I/O devices are relatively slower compared to the host-side resources.

Since the deployment of emerging high-performance devices, the host-centric architectures come to suffer from the following limitations. First, the latency of software stacks of the OS becomes a major portion of the I/O request handling latency. Second, the frequently triggered I/O requests consume the correspondingly increasing bandwidth of CPU and memory, which can create severe host-side resource contentions. Due to these problems, the performance of host-centric servers is scalable with neither the performance of devices nor the number of devices.

To mitigate these problems, previous studies propose to optimize the legacy software stacks to enable faster data and packet processing [3, 4, 5] or deploy a custom hardware device to completely bypass the software stacks
[6, 7, 8]. However, the existing schemes cannot achieve the full potential of the devices because they rely on special hardware bond and lack flexibility in utilizing host-side resources and supporting multiple heterogeneous devices. Therefore, architects now require a new server architecture to fully exploit the potential of emerging devices for running emerging server workloads which are both compute- and data-intensive.

In order to achieve the goal, we propose a novel architecture using FPGA-based device orchestrator. Unlike typical host-centric architectures, the proposed

architecture is designed to efficiently work with emerging devices to optimize the performance of both compute- and data-intensive workloads, and nicely scales with the performance of devices as well as the number of devices.

The key idea of our scheme is to selectively perform direct communications between devices via standard I/O communication protocols, without disturbing the host-side resources. In this way, our scheme satisfies the performance, flexibility, and scalability design goals as follows. First, it achieves high performance by enabling fast device-to-device transfers, while making the host-side resources fully reserved for other compute-intensive tasks. Second, it achieves flexibility by still providing both compute- and data-intensive workloads with the OS's key features which are essential for the workloads. Third, it achieves scalability by addressing excessive host-side resource consumption and using a separate hardware block to integrate a number of heterogeneous devices.

With the proposed architecture, users can enjoy many interesting usage scenarios. For example, users can accelerate the performance of their storage servers by simply adding more SSDs and NICs. In addition, they can provide secure data serving by adding GPUs to perform compute-intensive SSL protocol operations. Furthermore, as all these benefits come without disturbing the host-side resources, they can use the saved host-side resources to accelerate compute-intensive tasks or run them slowly to reduce power consumption.

In this chapter, we implement the prototype of the proposed architecture on top of a conventional host-centric server having NVM Express (NVMe) SSDs, NetFPGA NICs, and NVIDIA GPUs connected via PCIe. We first implement

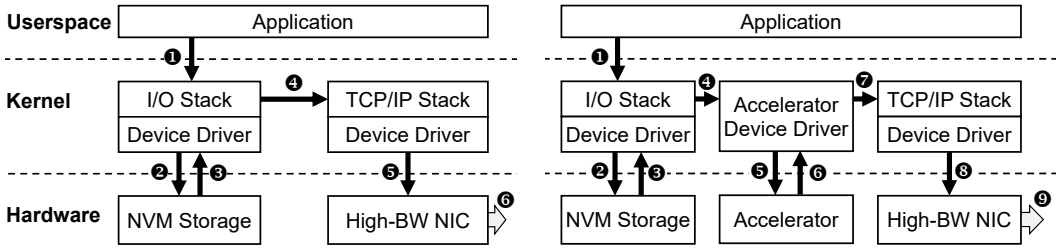
the architecture, a custom hardware device placed on each server to orchestrate other devices to perform direct device-to-device communications, on an FPGA chip embedded in the NetFPGA NICs. Next, we implement a supporting driver incorporating the protocol to make the devices work with the orchestrator properly. Finally, we implement user-level library which unmodified applications can exploit to take advantage of our scheme.

For evaluation, we run our prototype against microbenchmarks and full-scale emerging workloads. The results obtained from a real machine show that our scheme 1) reduces the device access latency by up to 25% and 2) saves the host-side resources up to 61% without degrading the performance of the microbenchmarks and the emerging workloads. Our projection further indicates that the performance benefit will continue to increase by improving the performance of the devices or adding more devices. As a result, our scheme significantly improves the performance of emerging workloads and it scales nicely with the performance of the devices.

2.2 Background & Motivation

2.2.1 The Host-Centric Architecture

Conventional servers employ the *host-centric architecture* which imposes the burden of device management on the host's OS. When we attach a new device, the host-centric architecture requires a device driver which implements hardware management routines. Interrupts are typically used for executing specific software routines of the device driver upon an event of the device. The device driver also



(a) Data serving

(b) Data serving with data-processing accelerators

Figure 2.1: Example datapaths on the conventional host-centric architecture.

implements interrupt handlers which process interrupts generated by the device. In addition to the device driver, some devices provide a user-level library (e.g., `libcudart.so` for NVIDIA CUDA) which implements device-specific application programming interface (API) functions.

Figure 2.1a illustrates how the command and data traverse between the system components (i.e., application, OS stacks, device drivers, and devices), for an example data serving application (e.g., FTP server) running on the host-centric architecture. After receiving a request from a remote client, the application loads the requested data from storage to CPU memory by executing relevant software routines (e.g., kernel's I/O stack and storage device driver). OS then generates packets with the data and sends them to a NIC using kernel's TCP/IP stack and the NIC device driver. Lastly, the NIC relays the packets to the remote client and notifies OS of their transmission. These steps are repeated until the requested data are fully sent to the remote client.

In case the data stored in the storage need to be processed (e.g., encryption) before being transferred to the remote client, high-performance accelerators can

be used (e.g., GPUs for SSLShader [9]). Figure 2.1b shows the datapath of the host-centric architecture with the accelerators. Similar to the non-accelerator case, the data are loaded to the CPU memory from the storage. Then, the host transfers data to the accelerator's memory using the accelerator device driver, commands the accelerator to process the data, and copies back the processed data. The processed data are then sent to the remote client via kernel's TCP/IP stack and the NIC.

2.2.2 Limitations

Unfortunately, as devices become faster, the host-centric architecture suffers from high inter-device communication overheads and host-side resource contentions (e.g., CPU and memory bandwidth) due to the following reasons. First, the latencies of kernel's software stacks, originally hidden by slow device access latencies, become the primary latency bottleneck. Second, the software stacks consume more CPU bandwidth due to the increased amount of device management overhead (e.g., more frequent interrupts and more device data to handle/process). Third, CPU memory being an intermediate buffer for all inter-device communications sacrifices a large amount of its bandwidth.

We profile an example data serving application to reveal the significance of the discussed limitations. We execute the example data-serving application on a server equipped with an NVMe SSD and a 10-Gbps NIC. The example application uses Linux's `sendfile()` system call to transfer 1-KB data to a remote client. We break down the datapath into three components: kernel's software stacks (I/O and TCP/IP stacks), storage access (storage-to-memory data transfer via

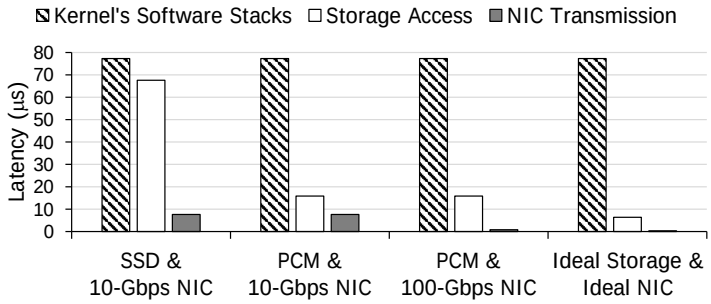
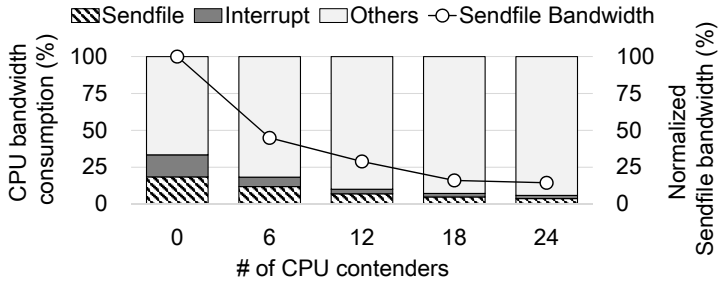


Figure 2.2: Breakdown of 1-KB data serving latency on the host-centric architecture

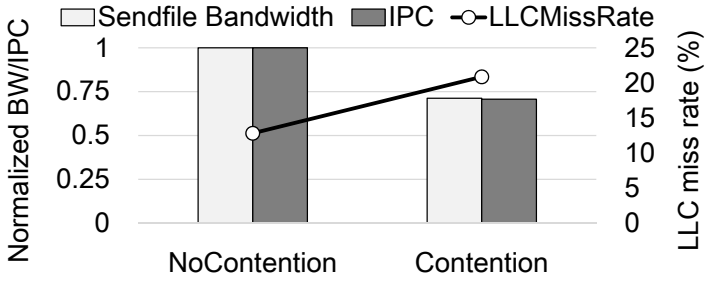
PCIe), and NIC transmission (memory-to-NIC packet transfer).

We observe that the I/O and TCP/IP stacks are already a primary performance bottleneck as they occupy 50.7% of the overall data serving latency (Figure 2.2). Furthermore, we evaluate how significant the overhead of the I/O and TCP/IP stacks becomes by simulating servers which employ emerging high-performance devices. Figure 2.2 shows how much of the overall data serving latency the I/O and TCP/IP stacks occupy as the performance of devices increases. We simulate a PCM storage device having a bandwidth of 3 GB/s (modeled after [6]), a 100-Gbps NIC, and ideal storage and NIC whose bandwidth fully saturates a PCIe Gen3 x32 link. We observe that the overhead of the I/O and TCP/IP stacks becomes exceptional; they can take up to 92.1% of the overall latency with ideal devices. We note that our simulation result is on a par with a previous study [7]; the study expects the kernel's software stacks will take up to 99.6% of the overall latency with emerging devices on a similar data serving scenario.

To evaluate how the host-side resource contention affects the data serving



(a) CPU bandwidth



(b) Memory bandwidth

Figure 2.3: Effects of resource contention on 1-GB data serving throughput

throughput, we prepare two microbenchmarks which stress different resources. One microbenchmark consumes CPU bandwidth by executing spin-loop code which does not incur any memory accesses. The other microbenchmark executes a set of write-after-read memory accesses with a stride size which matches the line size of the last-level cache (LLC). Executing the CPU-contending and memory-contending microbenchmarks will reduce the amount of CPU bandwidth and memory bandwidth available to the example data-serving application, respectively.

We observe that the data serving throughput clearly decreases as we run the resource-contending microbenchmarks concurrently with the example appli-

cation (the leftmost bars in Figure 2.3). As we increase the number of the CPU-contending microbenchmarks, a reduction in the throughput can become as large as 85.5% (Figure 2.3a). The CPU-contending microbenchmarks reduce the CPU bandwidth which `sendfile()` and interrupt handlers can consume, making it difficult to transfer data in a timely manner. Reducing the available memory bandwidth for the example application also leads to a 28.7% reduction in throughput (Figure 2.3b) because the memory-contending microbenchmark pollutes the LLC and thus reduces the data-serving application's LLC hit rate. The reduced LLC hit rate degrades the example application's instructions-per-cycle (IPC), resulting in the reduced data serving throughput. These results clearly show that the host-centric architecture suffers from both CPU and memory bandwidth contentions.

2.3 FPGA-based Device Control Mechanism

In this section, we implement a novel device control architecture using FPGA-based device orchestrator to overcome the inefficiencies of host-centric architecture and leverage the full performance of the emerging devices.

2.3.1 Key Idea and Benefits

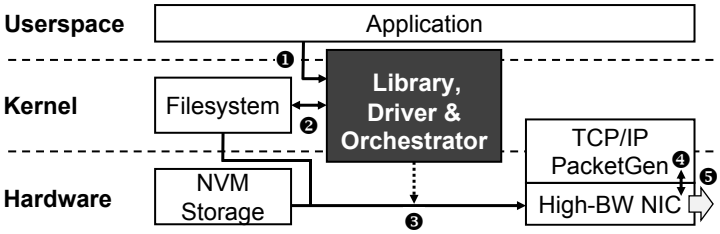
The key idea of the orchestrator is to allow multiple emerging devices to directly interact with each other in performing tasks, without host interventions. This optimization brings two significant benefits: 1) the host becomes free from the device management routines and their corresponding resource consumption, and 2) devices achieve shorter latency and higher throughput from the efficient

device management routines and the optimized data movement path.

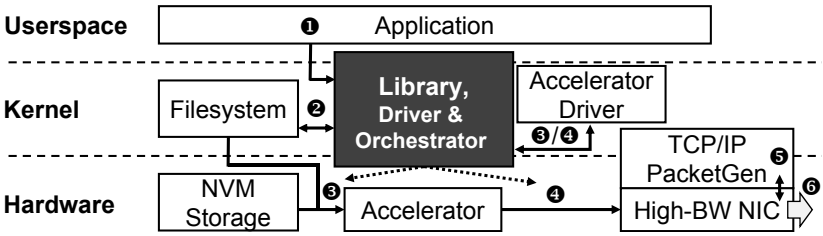
Figure 2.4 illustrates two cases of the direct device-to-device communication operation. In Figure 2.4a, the application sends the data in NVM storage to the others via NIC, and in Figure 2.4b, the application processes the data in NVM storage with the accelerator before sending it. The orchestrator implements the core functionality of our device-centric task processing.

In both cases, the orchestrator takes the initial request from the user applications and invokes the related devices in order to process the task. Contrary to the host-centric architecture (Figure 2.1), a device in the orchestrator passes its output directly to the next device to minimize the data transfer path as well as the corresponding resource consumption. As a result, the proposed scheme reduces the number of hardware-software or kernel-userspace boundary crossings for both storage-NIC and storage-accelerator-NIC cases.

Our architecture also consults the existing device supports (e.g., filesystem, accelerator driver, TCP/IP network stack) to provide full device functionality. We emphasize that such cooperation allows the proposed architecture to seamlessly integrate off-the-shelf devices and maximize compatibility, without restructuring the traditional architecture or adding expensive hardware customizations. Furthermore, the consulting overhead is minimal as the device supports usually incorporate communication of only a small amount of data (e.g., logical block addresses of a file).



(a) Storage and NIC



(b) Storage, accelerator, and NIC

Figure 2.4: The device management method using the FPGA-based orchestrator

2.3.2 Architecture and Operations

In this section, we explain the structure and the role of components in the proposed architecture (Figure 2.5).

Library

The library is the userspace component. The main role of the library is to capture a user application's device invocation request and forward it to driver that supports the orchestrator with the necessary information. Either userspace application programming interface (API) or modified dynamic-link library (DLL) which replaces an existing device invocation routine can perform such a task.

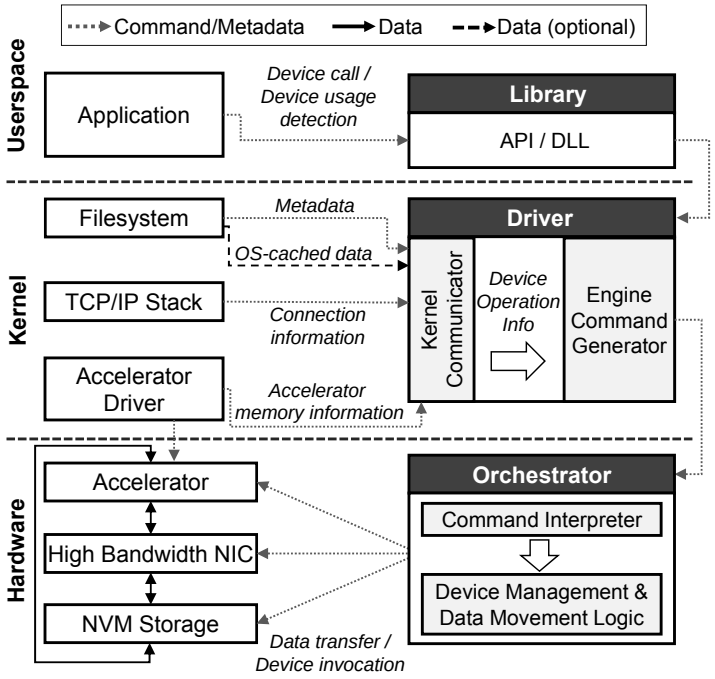


Figure 2.5: Proposed architecture

Driver

Upon receiving a device invocation request, driver consults the existing device supports to supplement the request. The kernel communicator extracts extra information via the existing interfaces of the device supports. Because such interfaces are usually generic and their implementations do not change frequently (e.g., filesystem, OS network stack), The driver is highly compatible (i.e., independent from device-specific details) and needs to be updated only when major changes occur to the traditional device supporting modules.

Depending on the devices involved in the request, the driver extracts different types of information. For example, if a storage device is involved, the driver

obtains filesystem metadata such as logical block address. In the case where the requested data is in the OS buffer cache, the driver manages data consistency by redirecting the requests to the storage device to the buffer cache. For other types of devices, the driver extracts the corresponding information (e.g., connection information for NIC, memory maps for accelerator). Once the driver supplements the original request, it passes the command to the orchestrator to initiate the device operation.

Orchestrator

The orchestrator resides in the hardware layer and intervenes between devices to manage their interactions. It interprets the command from the driver and translates a high-level task to the corresponding device operations to be performed. For each operation, the orchestrator sets up the corresponding device (i.e., provide necessary metadata), loads the data to device memory if necessary, and invokes the device function. Once all operations complete, the resulting data (or signal) can be handled either by the orchestrator or the interrupt handler of a device involved in direct communications.

The Proposed Architecture vs. Software-Only Approaches

Although we use a hardware-assisted design, software-only approaches are functionally possible if all components of the orchestrator are implemented in software. However, software-only approaches are sub-optimal in that they cannot meet the design goals.

First, they still suffer from CPU bandwidth consumption as they rely on

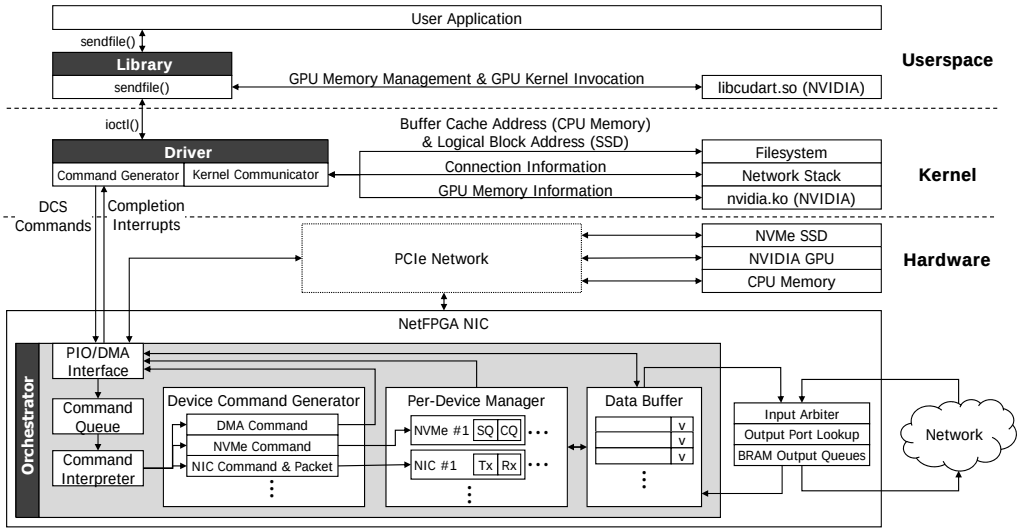


Figure 2.6: Implementation details of our prototype

CPUs to generate per-device commands (e.g., NVMe read/write commands). The CPU overhead would be nontrivial if the devices require frequent managements. Instead, our scheme offloads such CPU overheads to the orchestrator to minimize as much CPU bandwidth consumption as possible. Second, they may not work well with the off-the-shelf devices. Suppose a storage-to-NIC communication where the storage has plaintext data and the NIC lacks a TCP offload engine. In this case, software-only implementations must convert data into TCP/IP packets for the NIC which goes through the process similar to host-centric architecture. By employing a piece of hardware (i.e., the orchestrator), the proposed architecture resolves the mentioned problems and can easily support TOE to enable direct storage-to-NIC communications for NICs without TOE.

2.4 Implementation

In this section, we describe the implementation details of our example architecture prototype with Figure 2.6. The devices included in our example prototype are NVMe SSDs, NICs, and NVIDIA GPUs; however, any devices can be easily supported by implementing modest extensions to the components.

2.4.1 Library

Our library intercepts the commonly used `sendfile()` system call and replaces it with a substitute `sendfile()` implementation performing device-centric operations. As a result, applications using `sendfile()` automatically benefit from the device-centric operations.

The substitute `sendfile()` invokes custom system calls defined by the driver using `ioctl()`. The the driver's custom system call then initiates a set of device operations accordingly. As `sendfile()` receives the source and the destination file descriptors, the driver automatically retrieves information needed to perform the given task by communicating with corresponding kernel components. In addition to the file descriptors, users only need to forward device-specific information (e.g., GPU kernel codes) to the library which is already available by applications.

2.4.2 Driver

The driver consists of a kernel communicator and a command generator. The kernel communicator is responsible for retrieving information necessary to perform the given device-centric task. The command generator generates commands corresponding to the given device-centric task and sends them to the orchestrator.

For storage related operations, the kernel communicator interacts with kernel's virtual file system (VFS) and ext4 filesystem. In order to prevent stale data stored in NVMe SSDs from being transferred instead of the latest data in OS buffer cache, the kernel communicator first consults VFS and checks whether the latest data is cached in CPU memory. If OS buffer cache has the requested data, the kernel communicator informs the command generator to change the data source (or destination) from the storage device to OS buffer cache. Also, to ensure correct operations, the requested data chunk should reside in the cache until the task finishes; we use an existing page locking mechanism (i.e., increasing the reference count via `page_cache_get()`) to guarantee the correctness.

If the data resides in the storage device, the kernel communicator consults ext4 filesystem to obtain the physical data location, i.e., logical block address (LBA). While other filesystems may adopt different LBA management schemes, they expose similar interfaces to those we use for ext4 filesystem to communicate with generic layers such as VFS. The driver can therefore support other filesystems by adding pointers to those interfaces. With the LBA available, the storage device can directly transfer the requested data from storage to any other device via PCIe.

For network related operations, the kernel communicator interacts with the kernel's network stack. Using the socket file descriptor provided by `sendfile()`, the kernel communicator extracts the current connection status such as host address, client address, and sequence and acknowledgement numbers for TCP/IP connections. The information is passed on to the underlying network device

through the orchestrator.

For interactions between a GPU and other devices, it should pin its memory to PCIe address space. To create such a space, the kernel communicator asks the kernel-level GPU driver to provide the physical address of the source (or destination) GPU memory reserved via the userspace API functions (e.g., `cudaMalloc()`). The kernel-level GPU driver then obtains the physical address of the memory space and pins the memory space to make it visible through a base address register of the GPU.

After receiving such information from the kernel communicator, the command generator creates the commands and sends them to the orchestrator. Each command includes the data source device, metadata regarding the data source (e.g., address and size), the target device, and extra information to invoke target device function (e.g., IP address for NIC). In order to invoke a device-centric task, the driver simply writes the commands to the memory of the orchestrator.

2.4.3 Orchestrator

Our orchestrator prototype consists of six major modules: PIO & DMA Interface, Command Queue, Command Interpreter, Device Command Generator, Per-Device Manager, and Data Buffer. Currently, the prototype resides on NetFPGA for the ease of implementation; we expect it to be a stand-alone PCIe device in future implementations.

PIO & DMA Interface acts as the entry point of the orchestrator. It is responsible for all interactions with the driver's command generator and other PCIe devices. When it receives commands from the driver, it forwards them

to Command Queue. PIO & DMA Interface serves as a communication path between the orchestrator and other devices. It is also responsible for generating interrupts to CPUs so that the driver can acknowledge the completion of device-centric tasks.

Command Interpreter decodes the commands queued in Command Queue. Then, for each decoded command, Command Interpreter signals and forwards necessary information to per-device command generation logics inside Device Command Generator. Our current prototype implements the command generation logics for memory, NVMe, and NIC. First, the memory logic generates PCIe transactions which read from or write to CPU memory or GPU memory via base address registers. Second, the NVMe logic generates NVMe read and write commands which follow the NVMe specification [10]. Third, the NIC logic generates TCP/IP packets and their descriptors (i.e., payload address and length).

After creating the per-device commands, Device Command Generator sends the commands to either PIO & DMA Interface (for memory) or Per-Device Manager (for NVMe and NIC). Memory commands go to PIO & DMA Interface as they are standard PCIe memory access commands. On the other hand, NVMe and NIC commands are inserted to their corresponding command queue pair inside Per-Device Manager.

Per-Device Manager tracks and issues per-device commands by providing per-device interfaces. Our prototype implements NVMe and NIC interfaces. NVMe interface provides a pair of submission queue (SQ) and completion queue (CQ) for each NVMe SSD. NIC interface also provides a transmission queue

Virtex 5 Resource Usage	
LUTs	24506 / 149760 (16%)
Registers	27790 / 149760 (19%)
BRAM Blocks	75 / 324 (23%)
Performance Values	
Internal Throughput	9.54 Gbps (160 MHz, data width: 64 bit)
Power	5.87 Watts

Table 2.1: Prototyping results

(Tx) and reception queue (Rx) pair for each NIC. Per-Device Manager periodically checks whether a command is available in the interfaces. When Per-Device Manager finds an available command, it sends the command to the target device.

Data Buffer acts as an intermediate storage for inter-device transfers. Each buffer entry is associated with a ready bit to indicate the validity of its data. Upon reading data from a device (e.g., SSD-to-orchestrator), Per-Device Manager allocates buffer entries and marks their ready bits as invalid. After Data Buffer receives the data, the ready bits become valid. Then, pending per-device commands due to data dependency (e.g., orchestrator-to-NIC) can proceed by consuming the dependent data. When the data are fully consumed, Per-Device Manager deallocates the buffer entries for other commands.

The design of the orchestrator is both scalable and flexible. First, it provides scalable throughput with an increasing number of inter-device communications as its modules are pipelined and per-device commands are generated and issued in parallel. Second, it can support a scalable number of devices by increasing the size of the buffers of Per-Device Manager and Data Buffer. Third, it can support

any off-the-shelf device which implements the standard protocols (e.g., NVMe) regardless of its internal implementations. If the orchestrator needs to implement a new protocol, it only needs modest extensions to Command Interpreter (e.g., new commands) and a command generation logic to Device Command Generator. Table 2.1 shows that a typical orchestrator implementation consumes minimal logic resources while achieving high throughput. As can be seen, there is enough space to add 1) additional interfaces and buffers for a larger number of devices, and 2) new command generation logics to Device Command Generator to support a wider range of devices.

2.4.4 Design Considerations

Concurrent Inter-Device Communications

The proposed architecture seamlessly supports concurrent inter-device communications by providing the consistency of the orchestrator's Command Queue and device memories. First, the driver must acquire the lock associated with Command Queue when writing commands to Command Queue. Requiring lock acquisition ensures that no race condition occurs for Command Queue. Second, to avoid data races in device memories, the orchestrator processes the commands stored in Command Queue in a first-in, first-out manner. Thus, the commands sent by the driver do not get reordered, preventing any violation of device memory consistency due to command reordering.

Device Pair	Master	Slave
NVMe SSD ⇄ NIC	NVMe SSD	NIC
NVMe SSD ⇄ GPU	NVMe SSD	GPU
NIC ⇄ GPU	NIC	GPU

* When issueing per-device commands,
The orchestrator always becomes the master.

Table 2.2: Master/slave relationships between devices

Closed Hardware

Ideally, the orchestrator's Command Generator should be able to generate any per-device command. However, if the device-specific commands (e.g., memory allocation, memory pinning, and kernel execution for GPUs) are not available to public, our scheme must depend on the vendor-provided software as a workaround. For example, as NVIDIA does not disclose such information, our prototype relies on CUDA runtime and GPUDirect RDMA APIs at the expense of extra CPU bandwidth consumption. Therefore, to exploit the full potential of the proposed architecture, it is recommended for hardware vendors to make such information open to the public.

The closed hardware also affects master/slave relationships between devices. For a given device pair, the master device issues commands to access the slave device's memory. To serve the commands, the slave device must expose its memory to the master device.

Table 2.2 lists the master/slave relationships which our prototype uses for different device pairs. As we don't have any information on GPU command sets, GPUs always work as slaves for our prototype. On the other hand, the

orchestrator always works as a master device when it sends per-device commands to the attached devices.

Further Optimizations

While our main goal is to use a number of devices in a more efficient manner, previous studies [11, 6, 5] propose various schemes to address the limitations of the traditional single emerging-device management routines. The proposed architecture can directly adopt any of these ideas to further boost the performance of many devices, because such single device optimizations are orthogonal to our idea.

In addition, our current prototype implements only a critical set of TCP/IP protocol. Therefore, to support the entire TCP/IP protocol, the prototype relies on the kernel's network stack at the expense of extra CPU bandwidth consumption. For future implementations, we plan to enhance the prototype to support the entire protocol.

2.5 Evaluation

In this section, we present our experimental setup and evaluation results of the proposed architecture. First, we show that the proposed architecture achieves low-latency, high-throughput inter-device communications without consuming the host-side resources using microbenchmarks. Then, we present the performance improvements of the proposed architecture over the host-centric server architecture with real-world data-centric workloads. In addition, we show the high scalability of the proposed architecture by increasing the number of devices

Host	Description
CPU	Intel Core 2 Duo (2.00 GHz, LLC: 2 MB, FSB: 800 MHz)
RAM	2 GB, DDR2
OS	CentOS 6.5
Device	**Description**
NVM Storage	Samsung XS1715 SSD
NIC (orchestrator)	NetFPGA [12] with Xilinx Virtex 5
NIC (Profiling)	Intel 82599 10 Gbps Controller
Accelerator	NVIDIA Tesla K20m
Interconnect	Cyclone Microsystems PCIe2-2707 (PCIe Gen2 switch, # of slots = 5, switch bandwidth = 80 Gbps)

Table 2.3: Details of our experimental setup

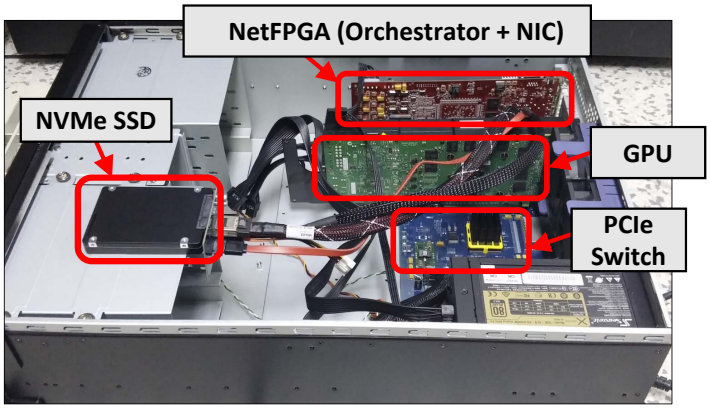


Figure 2.7: Our prototype (single-node view)

attached to a server.

2.5.1 Experimental Setup

Table 3.5 summarizes the system setup we use to evaluate the proposed architecture. We use a server with moderate processing power and memory band-

width. Such a server not only models recently-proposed energy-efficient servers (e.g., HP Moonshot System [13]), but also emphasizes the performance impacts of the host-side resource contentions. For emerging devices, we use off-the-shelf products and connect them with a high-bandwidth PCIe switch. Figure 2.7 shows the devices and the PCIe switch. The PCIe switch is connected to the host via a PCIe link.

We use the host-centric architecture discussed in Section 2.2.1 as our baseline system. The baseline system has the identical hardware and software setup to those of our scheme. The existing OS's kernel software stacks (SSD and NIC) and vendor-provided proprietary software stacks (GPU) manage the devices.

Currently, our NetFPGA NIC supports up to a single Gbps of network bandwidth. To match the network bandwidth with other emerging devices, we use a commercial 10-Gbps NIC to profile the bottlenecks of real-world workloads and derive the potential performance gains of the proposed architecture. The other system components are identical to those of the aforementioned system setup.

2.5.2 Evaluation with Microbenchmarks

Multi-Device Latency

This microbenchmark tests the latency of transferring a small amount of data between multiple devices in a given order. The goal of this benchmark is to emulate a case where an application issues small latency-sensitive requests which are served by traversing multiple devices.

In the benchmark, we measure the latency of reading a 4-KB data chunk from NVMe SSD and sending the packetized data chunk to the network via NIC.

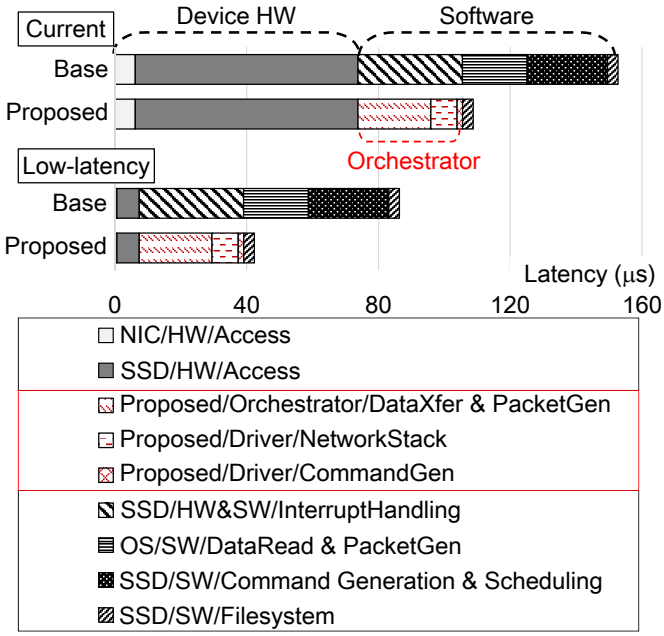
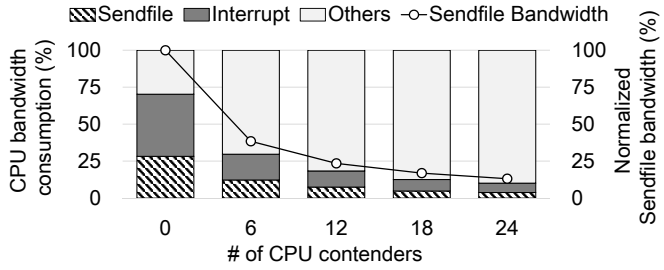


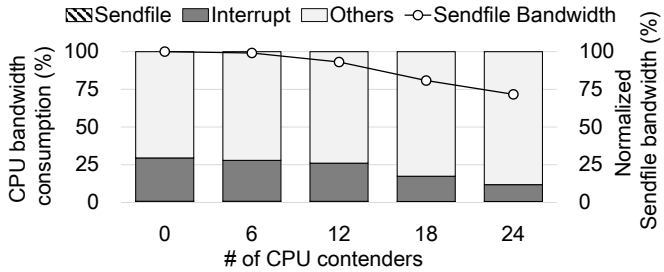
Figure 2.8: NVMe SSD & NIC latency on the baseline (Base) and the proposed architecture

Figure 2.8-Current shows the latency decomposition of the baseline system and the proposed architecture. For the baseline's NVMe SSD, we measure the access latency and consider the manufacturer specified hardware latency to deduce the interrupt handling overhead. For the orchestrator, we measure the latency from reading the data chunk from NVMe SSD to send the packetized packets to NIC. Then, we apply the same NVMe SSD and NIC latencies to those of the baseline system.

As expected, the baseline system suffers from going through multiple software stacks to manage devices. The proposed architecture replaces them with the operations within the hardware layer and significantly reduces the latency. The



(a) Baseline

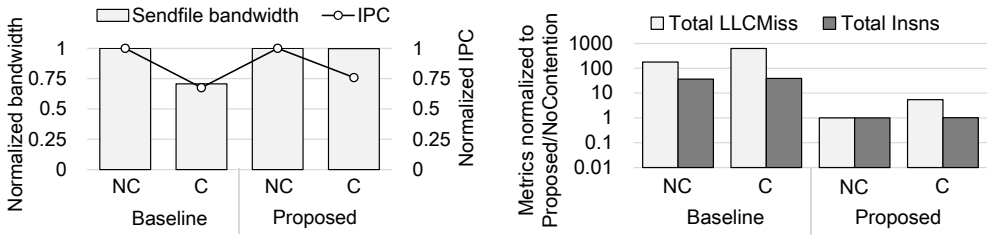


(b) The proposed architcure

Figure 2.9: Performance impacts of CPU cycle contention on the baseline and the proposed architecture

major benefit comes from integrating two completely separated NVMe SSD data fetch and packet generation process to a single operation.

Figure 2.8-Low-latency shows the expected latency for emerging high-performance devices. We reduce the latencies of NVMe SSD and NIC hardware by 90% while preserving the device management overheads. The projection illustrates that the inefficiency of the host-centric server becomes intolerable due to the growing software stack overhead, while the our architecture advantages grow as it highly optimizes the management operations.



(a) Sendfile bandwidth and IPC

(b) LLC misses and the number of executed instructions

Figure 2.10: Performance impacts of LLC and memory contention on the baseline and the proposed architecture (NC: no contention, C: contention)

Resource Contention and Device Performance

We now model a situation where an application attempts to use the devices while the host is busy with other tasks. In real datacenters, such contentions are common as a server typically runs multiple tasks to maximize the utilization as well as efficiency.

In particular, we consider two different types of host resource contentions: *CPU cycle* contention and *LLC capacity and memory bandwidth* contention.

CPU cycle contention. To emulate a server with CPU contentions, we run an increasing number of *spinlock-like* CPU cycle contenders along with the device-using application. The application performs `sendfile()` operations to send the data in the SSD to a remote host.

Figure 2.9 illustrates the performance of baseline and the proposed architecture with the CPU cycle contention. Without any contention, both servers achieve the full file transfer bandwidth (i.e., 1 Gbps). The baseline consumes more than twice larger CPU cycles than the proposed architecture due to the

software-managed devices (i.e., sendfile). The interrupt overhead of the two systems represents the RX packet handling of the NetFPGA driver. The overhead is large in the current implementation due to the unoptimized vendor-provided driver; the overhead would be much smaller with well-refined drivers, and the proposed architecture will consume almost no CPU cycles in such cases.

When we increase the number of CPU cycle contenders, the performance of baseline rapidly decreases. This is because the OS allocates fewer CPU cycles to the application to invoke and manage the devices. As a result, the data transfer bandwidth (i.e., device utilization) decreases with the increasing contention and the corresponding starvation of CPU cycles. In contrast, the proposed architecture is immune to such CPU contentions because the orchestrator manages the devices without host interventions. The slight performance drop at extreme contention cases (i.e., 18 to 24 contenders) is due to the current suboptimal NetFPGA driver implementation and the corresponding CPU bandwidth consumption. DCS will reduce such CPU bandwidth consumption to a minimal level with better driver implementations.

LLC and memory bandwidth contention. In the second case, we launch an LLC-stealing process along with `sendfile()` to cause memory contention. The LLC-stealing process strides a large memory region to incur LLC misses and the corresponding memory accesses. We ensure that CPU cycle does not become the bottleneck by pinning each process on each core.

Figure 2.10 shows the impact of memory contention on the device-using application. For the baseline, even a single memory contender significantly drops

the core efficiency (i.e., IPC in Figure 2.10a) and the data transfer performance because the system heavily relies on CPU to make progress. On the other hand, the performance of the proposed architecture is independent from the core execution efficiency.

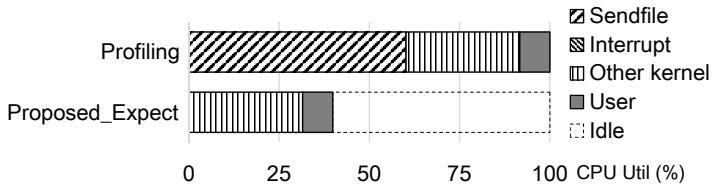
Figure 2.10b illustrates why the proposed architecture is free from host-side resource contentions. First, it delegates the device managements to the hardware and executes an order of magnitude fewer instructions (vs. baseline) to perform the same task. This in fact can benefit the contending workload by allowing the others to consume more CPU cycles. Second, memory requirements of the proposed architecture are orders of magnitude smaller than that of the baseline. While the baseline should read all of the device data to host and therefore demands a lot of memory capacity as well as bandwidth, The proposed architecture allows direct device-to-device communications and is less sensitive to memory availability.

In summary, the microbenchmark results prove that the proposed architecture offers significantly better performance and robustness compared to the traditional host-centric architecture. The proposed architecture also saves a large amount of host-side resources to let the others leverage it or let the host scale down to achieve better efficiency.

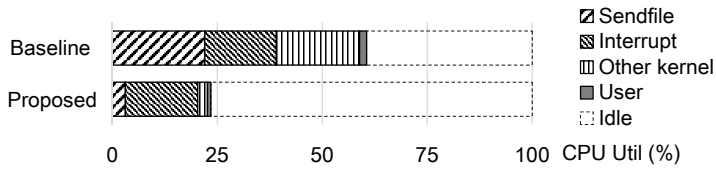
2.5.3 Evaluation with Data-Centric Workloads

Object Storage

The first data-centric workload we consider is an object storage system [14, 15], which aims to address scalability, high performance, and reliability in storing



(a) Bottleneck analysis and expectation (10 Gbps)



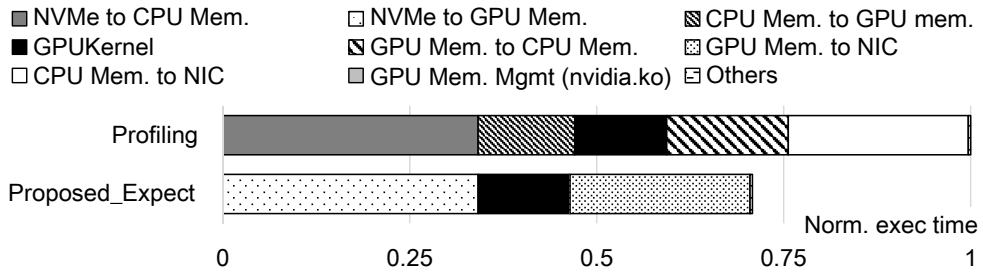
(b) Real-world evaluation (1 Gbps)

Figure 2.11: CPU bandwidth consumption of object storage on the baseline and DCS

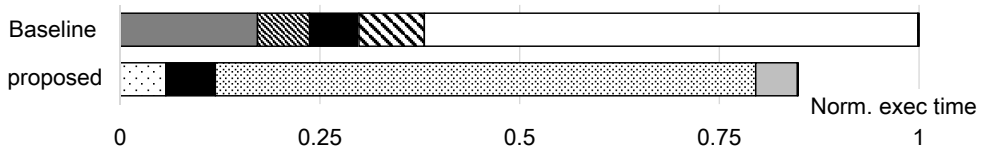
and serving big data. We use OpenStack Swift [16]; the data-serving storage backend, which sends data from an NVMe SSD to clients via NIC, leverages the proposed architecture. To stress the system, the 10 emulated clients concurrently issue download requests for a 1-GB object to the system. We use the latency it takes to serve all the client requests as a performance measure.

Figure 2.11a-Profiling shows the bottleneck analysis of the object storage workload on the profiling system. To serve the data with high-bandwidth (10 Gbps), `sendfile()` and device management routines (i.e., other kernel functions) fully utilize CPU. This shows that even storage servers can become host-bottlenecked due to the high data serving rate of emerging storage devices and interconnects.

We observe that the proposed architecture improves the performance by 17.5% (from 94 seconds to 80 seconds) by increasing the object serving through-



(a) Bottleneck analysis and expectation (10 Gbps)



(b) Real-world evaluation (1 Gbps)

Figure 2.12: Execution time of encrypted data serving on the baseline and the proposed architecture

put from 0.85 Gbps to 1 Gbps. Figure 2.11b shows the breakdown of CPU bandwidth consumption of the baseline and the proposed architecture. Although neither system is fully saturated due to the lower bandwidth (1 Gbps), sendfile and NetFPGA driver's interrupt handlers consume a significant amount of CPU bandwidth. In addition, the baseline's interrupt handling overhead from the two emerging devices cause the interrupts to stack up and consume extra host resource (i.e., CPU and memory bandwidth). The proposed architecture totally eliminates the sendfile overhead and the additional kernel overhead. For a high-bandwidth system (Figure 2.11a-Proposed_Expect), by conservatively applying the impact of removing sendfile overhead, we can expect the proposed architecture to reduce the CPU bandwidth consumption by more than 50%.

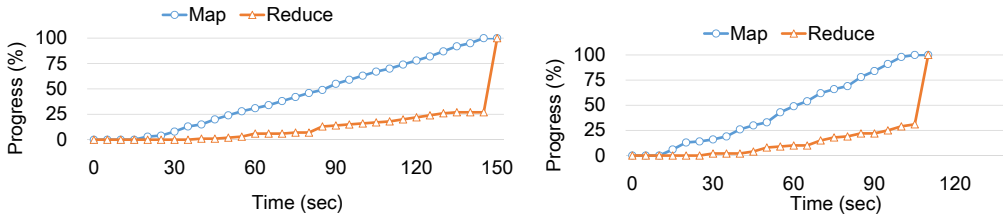
Encrypted Data Serving

Secure data transfer has become increasingly important with the growth of data volume. The main concern is that modern data objects are too large to rapidly encrypt and transfer. We therefore consider an encrypted data serving application which uses a GPU to quickly encrypt the data stored in an NVMe SSD and send it via a high-bandwidth NIC. In our experiment, we encrypt and send a 512-MB object with SSLShader [9].

Figure 2.12a-Profiling shows the bottleneck analysis on the profiling system. We note that the data always goes through the host due to the closed nature of NVIDIA GPUs; this consumes unnecessary host-side resources as well as elongates the data path and processing time. The baseline system (Figure 2.12b) shows a similar pattern except that the CPU memory to NIC transfer takes longer due to lower NIC bandwidth. As the proposed architecture eliminates all indirect transfers and reduces the processing time (shown in Figure 2.12b), it will also optimize the datapath for a high-bandwidth system (Figure 2.12a-Proposed_Expect) and bring larger benefits.

MapReduce

MapReduce [17], a popular programming framework to process a huge amount of data using data-parallel tasks, also significantly benefits from the proposed architecture. MapReduce requires both fast inter-device data transfers (to serve data to mappers and reducers) and a large amount of CPU bandwidth (to execute mappers and reducers). We examine how MapReduce benefits from the



(a) Baseline

(b) The proposed architecture

Figure 2.13: Execution time of 10-GB *grep* on Hadoop

proposed architecture by executing a Hadoop-based, 10-GB *grep* task on a two-node setup. One node is a data-serving node which stores all the data needed by the *grep* task's mappers and reducers. Therefore, the mappers and reducers running on the other node (i.e., computation node) must receive data from the data-serving node. Both of the nodes execute the *grep* task's mappers and reducers using the available CPU bandwidth.

Figure 2.13 shows how the *grep* task progresses when using the baseline (Figure 2.13a) or the proposed architecture (Figure 2.13b) as the data-serving node. The proposed architecture reduces the execution time of the *grep* task by 27.52% (from 149 seconds to 108 seconds) due to the following reasons. First, mappers begin earlier as the computation node can retrieve data with a bandwidth of 713 Mbps from the data-serving node using the proposed architecture, which is 80.96% faster than the data-serving node using the baseline (394 Mbps). Second, the execution latency of mappers and reducers running on the data-serving node become much shorter as the proposed architecture allows the reclaimed CPU bandwidth to be consumed by mappers and reducers.

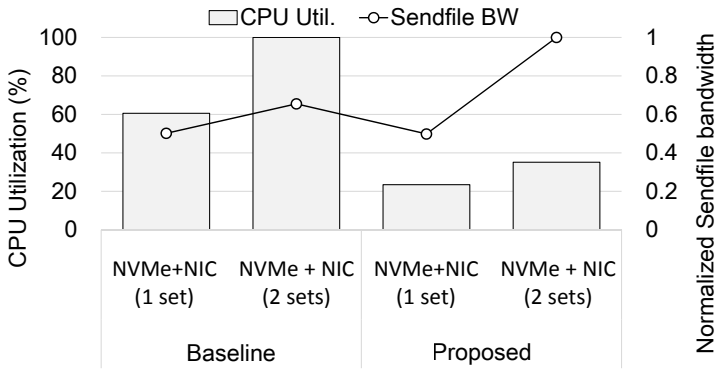


Figure 2.14: CPU bandwidth consumption and `sendfile()` bandwidth on the proposed architecture

2.5.4 Scalability

To test the scalability of the proposed architecture, we increase the number of devices for both the baseline and the proposed architecture. For the workload, we use `sendfile()` to utilize NVMe SSDs and NICs.

Figure 2.14 shows the workload performance and the CPU utilization for the scalability test. When we increase the number of devices to two sets (i.e., two NVMe SSDs and two NICs), the baseline suffers from high CPU utilization and therefore fails to scale. In contrast, data transfer bandwidth of the proposed architecture linearly scales with the number of devices because it is independent from the host. We note that the increase in CPU utilization for the proposed architecture comes from an unoptimized NetFPGA NIC driver; for optimized drivers, increasing the number of devices in the proposed architecture would have a negligible impact on host-side resource utilization.

III. Implementing FPGA-Based Orchestrator for Flexible and Highly Applicable Device-Control Mechanism

3.1 Introduction

In the previous chapter, we discussed the performance problem of conventional host-centric architecture servers and proposed an FPGA-based device orchestrating method to reduce the software overhead. The results show that the proposed idea achieves the full potential of emerging high-performance devices, and it scales nicely with the performance and the number of devices.

For further optimization 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 [1, 22, 23, 24] 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,

	Host-centric architecture [11, 18, 19, 20, 21, 6, 5]	PCIe P2P communications [1, 22, 23, 24]	Device integration [7, 25]	**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 3.1: Comparison of existing inter-device communication schemes and DCS-ctrl

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.,[7, 25]). Their consolidated devices can perform direct data and control transfers without software intervention. However, such custom-built and integrated 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.,[26, 27]) 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 de-

vice 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 [28, 16, 29, 30], 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 chapter, we enhance the flexibility and applicability of the orchestrator which proposed in the previous section and prototype the upgraded version of the orchestrator, named HDC Engine. we introduce *DCS-ctrl*, which leverages the HDC Engine, 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. Contrast to the orchestrator introduced in the previous section, HDC Engine achieves high applicability and flexibility as follows.

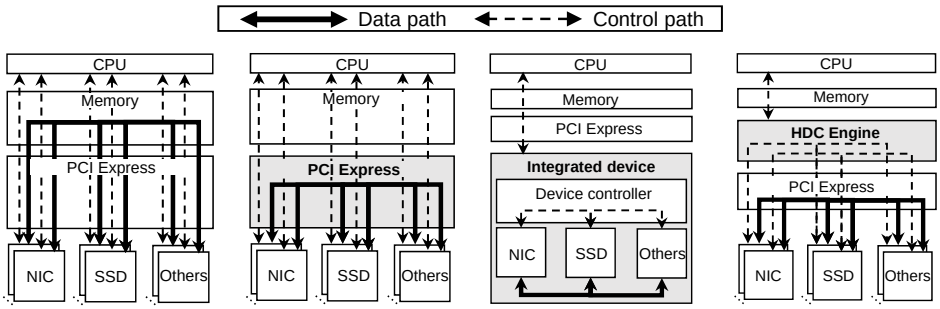
First, DCS-ctrl improves flexibility of direct D2D communications by implementing standard device controllers in HDC Engine. As HDC Engine's device controllers build standard device commands of other peripheral devices, and submit and complete them through PCIe P2P communications, DCS-ctrl does not require 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.

Second, 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 [16] and Hadoop distributed file system (HDFS) [28]. 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



(a) Host-centric arhitecture (b) PCIe P2P communication (c) Integrated device (d) DCS-ctrl

Figure 3.1: Existing inter-device communication schemes and DCS-ctrl

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 outperform existing D2D communication schemes.

3.2 Background and Motivation

3.2.1 Background

Conventional server architectures are *host-centric* as they rely on host-side CPUs and memory for device operations. Figure 3.1a 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). architectures incur the significant host-side overhead as modern devices become faster. 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 [11, 7, 1, 22, 25, 23, 24].

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 3.1b, 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, [1, 22, 23, 24] 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 3.1c. 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 [7] integrates storage controllers and network interfaces into a single device functioning as a fast storage area network. As another example, BlueDBM [25] 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 3.1d, DCS-ctrl enables fast, efficient, and flex-

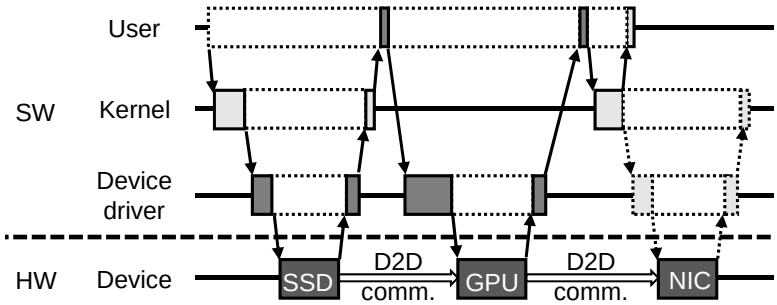


Figure 3.2: Timeline of a software-based device-control mechanism (each block does not exactly match time length.)

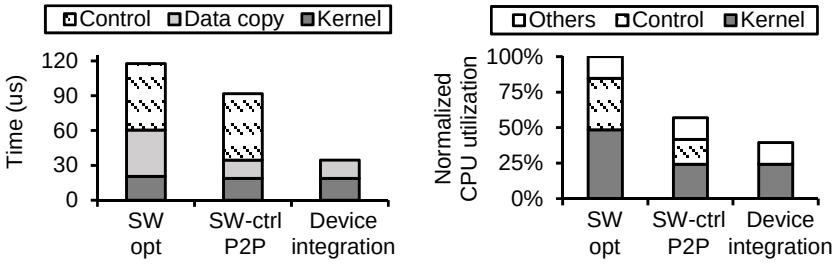
ible direct inter-device communications among off-the-shelf peripheral devices.

3.2.2 Motivation

Software-based Device Control - Performance Overhead

Existing PCIe P2P communication schemes still rely on slow and CPU-inefficient software-based device-control mechanisms. As shown in Figure 3.2, 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



(a) Latency (b) CPU utilization

Figure 3.3: Software overheads of multi-device communication

with hash computation on a GPU) into three main software components (Figure 3.3a). We use optimized I/O kernel stacks introduced in [11, 1, 22, 25, 24, 31] 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 3.3b). 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

Application	Category	Intermediate processing
HDFS [28]	Data integrity	CRC32
	Compression	GZIP
	Encryption	AES256
Swift [16]	Data integrity	MD5
	Encryption	AES256
Amazon S3 [29]	Data integrity	MD5
	Compression	GZIP
	Encryption	AES256
Azure Blob [30]	Data integrity	MD5
	Encryption	AES256

Table 3.2: Intermediate data processing for scale-out storage applications

devices.

Device Integration - Limited Flexibility

While integrated devices can achieve the optimal performance of inter-device communications (device integration in Figure 3.3), 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., [26, 27]). Furthermore, their inflexible integration easily leads to under-provisioning or over-provisioning problems in modern servers.

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 process-

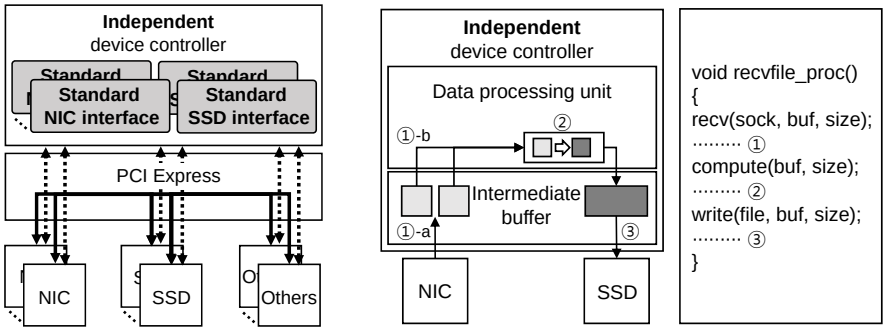
ing between device operations prevents the server architects from aggressively applying direct D2D communications to existing server applications.

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 3.2 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.

3.2.3 Design Goals

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



(a) Flexible device support (b) Intermediate data processing

Figure 3.4: Design goals of direct inter-device communications

- **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 3.4a).

- **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 3.4b, it should merge network (①) and storage (③) operations and seamlessly perform intermediate data processing (②) between them.

3.3 DCS-ctrl: A Fast and Flexible Device-Control Mechanism

3.3.1 DCS-ctrl Architecture

We propose *DCS-ctrl*, a hardware-based device-control (HDC) mechanism for device-centric server (DCS) architecture, to achieve high performance, flexi-

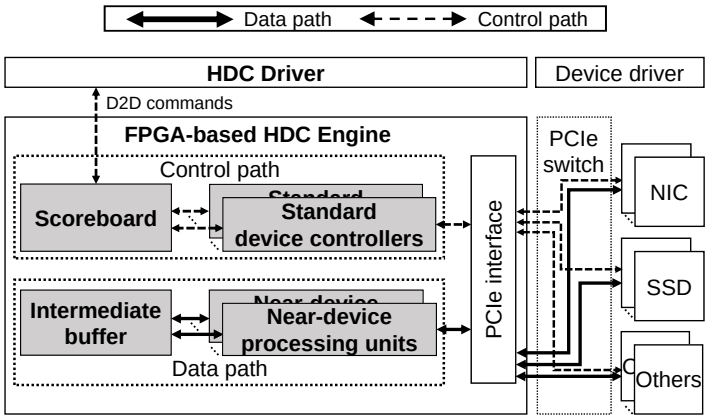


Figure 3.5: DCS-ctrl architecture

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

Figure 3.5 shows how DCS-ctrl architecture performs all control and data transfers among high-performance peripheral devices. *HDC Driver* is an op-

timized 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 their states. *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 controllers, and NDP units. We also explain how DCS-ctrl works with the existing software optimizations in Section 3.3.5.

3.3.2 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 3.6. 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

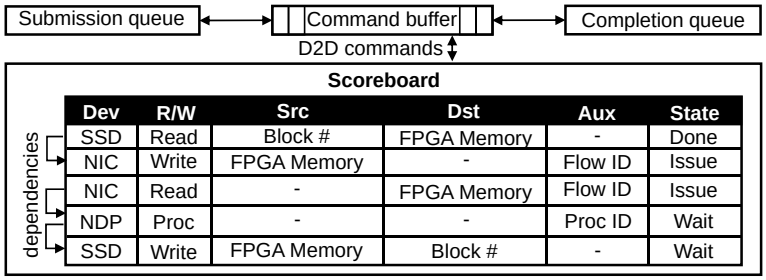
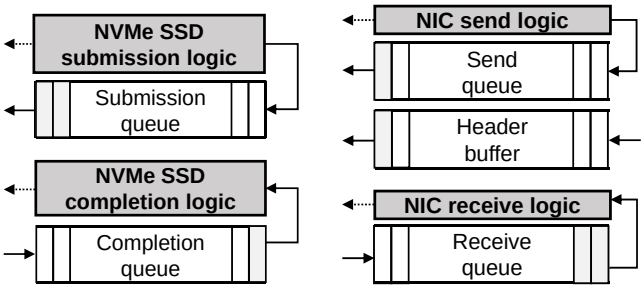


Figure 3.6: A scoreboard in HDC Engine to orchestrate peripheral devices for multi-device tasks

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 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.



(a) SSD controller (b) NIC controller

Figure 3.7: Standard device controllers for (a) SSDs and (b) NICs. Each device controller includes a queue pair and device-control hardware logic.

3.3.3 FPGA-based and Disaggregate Device Controllers

To overcome the architectural limitation of integrated devices introduced in Section 3.2.2, DCS-ctrl exploits standard and generic 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 or a large number of devices with low cost and hardware complexity.

In this thesis, 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 3.7a, 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 3.7b). 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.

3.3.4 Near-Device Processing Units

In contrast to the prior work [1] 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 integrity check, data encryption, and compression. (Table 3.2). Such intermediate processing can be easily offloaded

Processing units	LUTs	Registers	Maximum clock freq.	Throughput per unit
MD5 [32]	3.0% (8970)	0.69% (4180)	130MHz	0.97Gbps
SHA1 [33]	3.49% (10760)	1.13% (6848)	235MHz	1.10Gbps
SHA256 [34]	4.28% (13090)	1.23% (7480)	130MHz	0.80Gbps
AES256 [35]	3.52% (10689)	0.99% (6000)	>250MHz	40.90Gbps
CRC32 [36]	0.03% (93)	0.01% (53)	>250MHz	10Gbps
GZIP [37]	5.36% (16273)	2.09% (12718)	178MHz	100Gbps

Table 3.3: Estimated Virtex 7 FPGA resource utilization and maximum operating clock frequency for commonly required intermediate processing units

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 [38]. 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 3.3 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

[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.

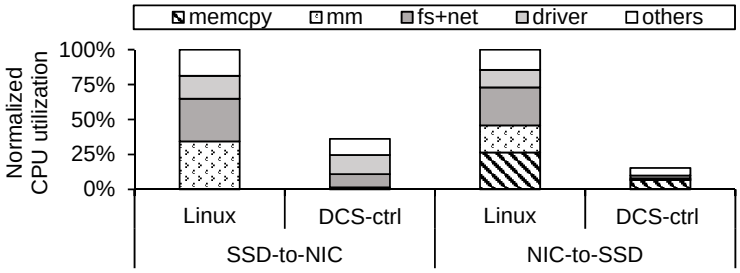


Figure 3.8: DCS-ctrl software optimization

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.

3.3.5 Software Optimization

DCS-ctrl exploits existing software optimizations (e.g., [11, 1, 22, 25, 24, 31]) 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 3.8 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 like other existing software optimization approaches.

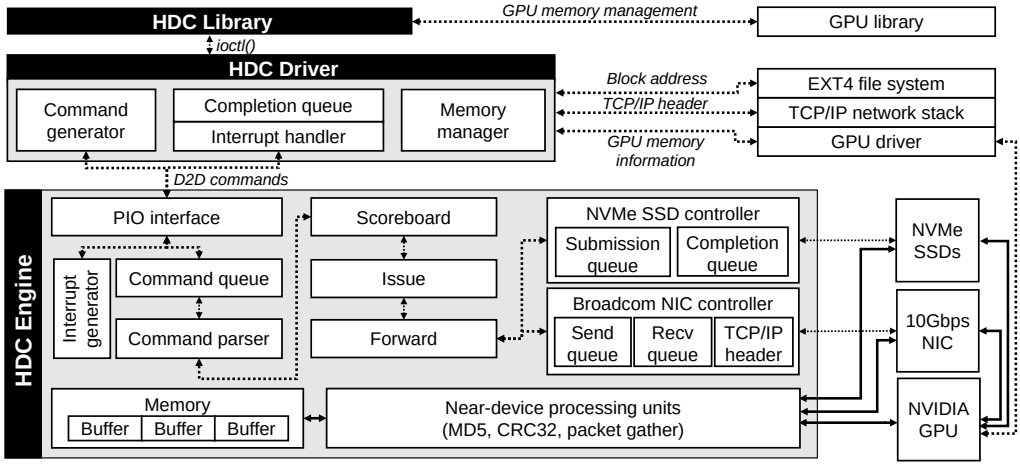


Figure 3.9: HDC Engine implementation

3.4 Implementation

We implement HDC Engine prototype on a Xilinx Virtex-7 FPGA VC707 board [39], and it is connected with off-the-shelf Intel NVMe SSDs [40], Broadcom 10-Gbps Ethernet cards [41], and NVIDIA GPUs [42] through a PCIe switch [43]. 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 3.9).

3.4.1 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.

3.4.2 HDC Driver

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

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

Table 3.4: HDC Engine's device controllers on Virtex-7 resource utilization

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.

3.4.3 HDC Engine

We implement Intel NVMe SSD and Broadcom 10-GbE NIC controllers on a Xilinx Virtex-7 VC707 board [39] using Xilinx Vivado Design Suite [38]. Table 3.4 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 3.3).

In addition to the key architectural components introduced in section 3.3, 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 PCIe interface. 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 [44] 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 [45].

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).

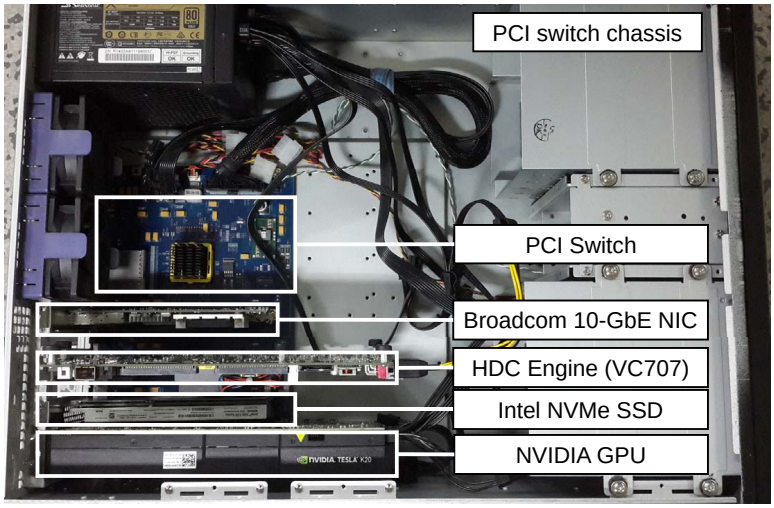


Figure 3.10: DCS-ctrl prototype (showing a single node)

3.5 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.

3.5.1 Experimental Setup

Table 3.5 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 3.10 shows our prototype with the devices and the PCIe switch. We use a two-node setup equipped with

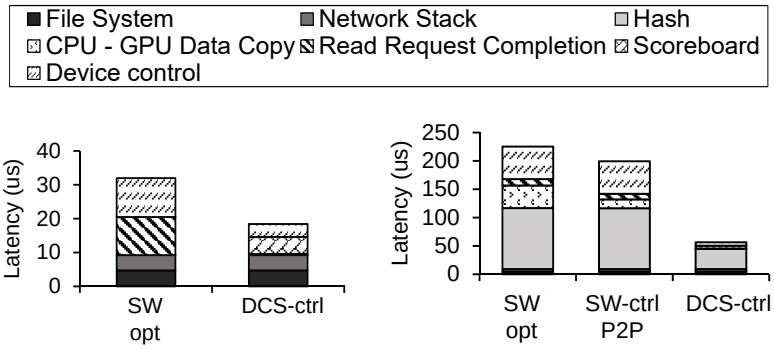
[3]Note that the effective bandwidth of the NIC is around 9 Gbps due to packet overheads.

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 3.5: Details of our experimental setup

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 [44, 46]. In that case, software-controlled P2P cannot perform D2D communications.



(a) SSD→NIC (b) SSD→Processing→NIC

Figure 3.11: Latency breakdown of inter-device communications

3.5.2 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 3.11a 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 re-

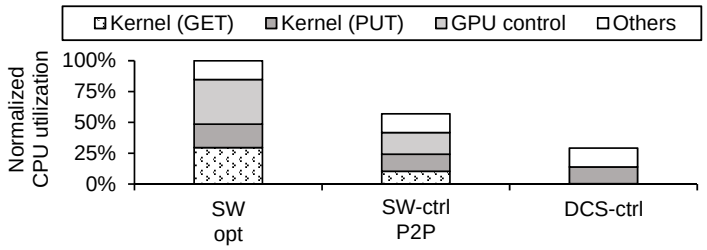
quest completion) with the minimal scoreboard overhead.

The result of the SSD→Processing→NIC microbenchmark (Figure 3.11b) presents the effect of direct inter-device 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.

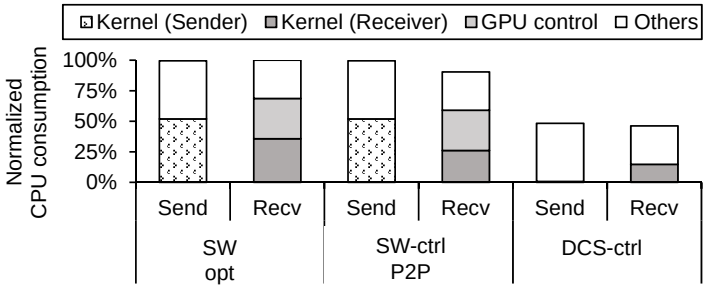
In summary, the microbenchmark results prove that DCS-ctrl offers a significant latency reduction over software-controlled P2P which controls all devices using CPUs. Also, NDP brings additional benefits when intermediate processing is required.

3.5.3 Scale-out Storage Workloads

As scale-out workloads, we use OpenStack Swift [16], an open-source object storage system, and Hadoop Distributed File System (HDFS) [28], a distributed file system. They perform compute-intensive data integrity checks (i.e., MD5 in Swift and CRC32 in HDFS) during data communications.



(a) Swift



(b) HDFS

Figure 3.12: CPU utilization breakdown of scale-out storage applications

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 [47] 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

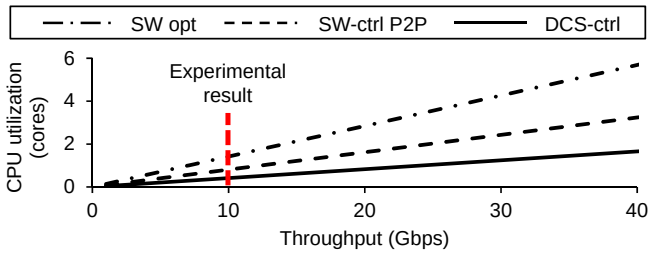
bandwidth of target servers.

We measure and break down CPU utilization of DCS-ctrl and baseline designs with the same throughput (Figure 3.12a). 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.

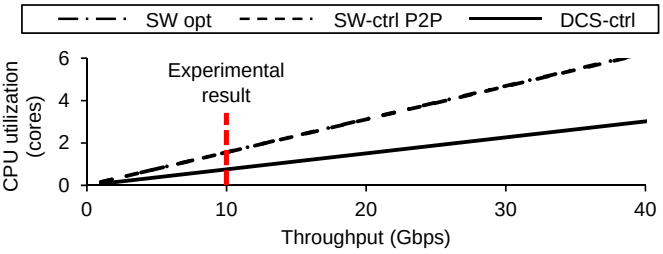
Distributed File System - HDFS

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 3.12b shows the CPU utilization breakdown of HDFS on DCS-ctrl and the baseline designs at the same bandwidth. There is



(a) Swift



(b) HDFS

Figure 3.13: Estimated CPU utilization with high-performance devices

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 3.12b 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.

Scalability

Figure 3.13 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 3.13a 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 3.13b). Software-controlled P2P shows little performance improvement because of the lack of opportunity for direct inter-device communication. Since DCS-ctrl requires only three 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.

IV. Related Work

In this chapter, we discuss other research shows goal is reducing the host-side overhead and take full advantage of emerging high-throughput devices. Due to the importance of the high-throughput devices, researchers proposed diverse architectures that can be categorized into three groups: software optimization, device integration, and PCIe P2P communication.

4.1 Software Optimization

Software optimization schemes modify the kernel, which is complex and the primary source of the device access overhead, and minimize the host-side overhead. Since they usually do not require any special hardware for the optimization, server architects can easily adopt them at low cost. Moreover, the software optimization schemes keep legacy API interfaces for the accessing devices. This enables old applications to still run on the servers after adopting the optimization schemes.

However, Software optimization schemes still make devices always send their data to the host memory even during the device-to-device communications. As a result, at device-centric servers, where inter-device communication happens frequently, it is hard for software optimization schemes to take full advantage of high-throughput devices. Also, since these schemes rely on host CPUs for controlling devices, the host suffers from high device control overhead with many

devices. This problem results in the contention between compute-intensive workloads and data-intensive workloads and the low utilization of devices. Therefore, software optimization methods show lower scalability than device integration and PCIe P2P communication methods.

Examples. Moneta-D [6] and NVMeDirect [48] use kernel bypassing when accessing storage devices. Similarly, mTCP [5] 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 [49] exploits the hardware virtualization support to partially bypass kernel. FLASH [50] suggests a custom node controller to achieve high scalability and low I/O overhead. However, both do not consider D2D communication.

Moneta [11] introduces various optimizations on the kernel software stack. Recent studies like SPIN [22] and ReFlex [51] 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 [22]. Also, Falcon [52] 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.

4.2 Device Integration

Device integration methods integrate two or more devices into a single device and make the data transfer between them faster. Because they are usually

specialized for some applications, the devices also incorporate processors to run the applications' code. As a result, they do not need help from CPU for processing the applications and it leads to a significant reduction of the host overhead. The methods also reduce latency dramatically since almost all communications happen in a single device.

However, since the devices are specialized for few applications, other applications cannot take advantage of the device integration. Also, the performance benefit only comes from the communications between integrated devices. This feature limits the applicability because adding a new or an additional device is unavailable.

Examples. QuickSAN [7] integrates SSDs and NICs to minimize host-side resource overheads and access to remote storage faster. Similarly, BlueDBM [25] 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 [53] enables a configurable SSD interface, and Biscuit [54] 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 [31] 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 [55] joins a GPU with an SSD in order to give a large memory for the GPU. In addition, Mellanox's BlueField [56] puts a NIC, a PCIe switch and small processor cores on one chip. It supports NVMe over fabric [57] and Mellanox OFED GPUDirect RDMA [58] 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.* [59] modify the device drivers and the libraries of GPUs and Infiniband network cards to allow GPUs to control network devices.

4.3 PCIe P2P Communication

PCIe P2P communication schemes leverage PCIe for direct communication between devices. When a device transfers its data to another device, the data only go through PCIe interconnect bypassing the host memory. Since they do not need to integrate devices for direct communication, they can add more devices and make a new data path. This means that any applications and devices can take advantage of PCIe P2P communication.

However, since the data are transferred through PCIe interconnect without any processing, applications cannot use the schemes when intermediate process-

ing is required. As Table 3.2 shows, many applications need intermediate data processing and they prevent the applications from leveraging PCIe P2P communication.

Examples. Neuwirth *et al.* [60] propose EXTOLL interconnect using PCIe to directly connect EXTOLL NICs and Xeon Phi coprocessors. Morpheus [23] 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 [61] and AMD DirectGMA [62] provide a set of functions to expose GPU memory to PCIe address space. As one example, GPUnet [63] proposes the network programming model on GPU using Mellanox OFED GPUDirect RDMA [58]. 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 [1] 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 [64] 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.

V. Conclusion

In this thesis, we proposed a fast, scalable and flexible FPGA-based device orchestration method to resolve the inefficiency of conventional host-centric architectures. The key idea was to implement a low-cost and flexible device orchestrator while maximizing the opportunity and applicability of the direct inter-device data communication using near-data processing at the same time.

First, we evaluated the performance improvement of the proposed scheme. We implemented an FPGA-based orchestrator that control peripheral devices on behalf of the host CPU and memory. The results show that the orchestrator enormously reduces the CPU bandwidth utilization for controlling peripheral devices. At the same time, it also reduces the control latency by bypassing the complex OS kernel. We expect that the performance gain of the idea will increase as the peripheral devices are getting faster.

Second, we showed that our idea maximizes the opportunity and applicability of direct inter-device data communication when the orchestrator uses near-data processing unit and modular device controllers. To evaluate the efficiency of the idea, we prototyped the new orchestrator, named HDC Engine. It is highly flexible and maximizes the opportunity of D2D communications and improves the performance of various applications aggressively.

요 약 문

현대의 고성능 서버는 짧은 시간에 대용량의 데이터를 처리하기 위하여 많은 수의 고대역폭 주변장치(예: NVMe SSD, GPU)를 사용한다. 하지만 사용하는 주변장치의 수가 늘어나면서 서버의 CPU와 메모리는 주변장치 제어를 하는 것 만으로도 많은 수의 사이클을 소모하게 된다. 이는 주변장치를 제어하는 커널의 코드가 복잡하기 때문이다. 이 문제를 해결하기 위하여 몇몇 연구에서는 커널 코드를 우회하는 방법들을 제안하였지만 주변장치의 최대 성능을 끌어내기에는 부족하였다. 다른 연구에서는 주변장치 간 직접 통신을 활용하여 CPU와 메모리의 부담을 줄여보려고 시도하였다. 하지만 주변장치 사이에 데이터를 전달 중에 CPU가 해당 데이터에 접근하게 되는 경우에는 적용할 수 없어서 활용성이 떨어지는 문제가 있었다.

이 문제들을 해결하기 위하여, 본 논문에서는 디바이스 간 빠른 데이터 통신을 위한 FPGA 기반 디바이스 제어 기술을 제안한다. 제안된 방법에서는 고성능과 높은 확장성, 높은 활용성을 달성하기 위하여 다음과 같은 방법들을 사용한다. 우선 주변장치 제어를 위한 표준 주변장치 제어 모듈을 FPGA에 구현하여 다양한 주변장치 간의 직접 통신을 가능하게 한다. 두 번째로는 주변장치 제어를 FPGA에게 전담시킴으로써 CPU와 메모리의 부담을 최소화한다. 마지막으로 FPGA 위에서 CPU 대신 연산을 수행할 수 있도록하여 활용성을 극대화한다.

본 논문에서는 제안된 기술을 실제 시스템 위에서 구현함으로써 제안된 기술이 CPU와 메모리의 부담을 줄이고 클라우드 컴퓨팅 워크로드의 성능을 향상시킴을 확인하였다.

References

[1] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. DCS: A Fast and Scalable Device-Centric Server Architecture. In *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2015.

[2] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. Dcs-ctrl: a fast and flexible device-control mechanism for device-centric server architecture. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 491–504. IEEE Press, 2018.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[4] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[5] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A Highly Scalable

User-level TCP Stack for Multicore Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2014.

[6] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven 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.

[7] Adrian M. Caulfield and Steven Swanson. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2013.

[8] NVIDIA Corporation. NVIDIA GPUDirect. `https://developer.nvidia.com/gpudirect`.

[9] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[10] NVM Express, Inc. NVM Express. `http://www.nvmexpress.org/`.

[11] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2010.

[12] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *Proc. 2007 IEEE International Conference on Microelectronic Systems Education (MSE)*, 2007.

[13] Hp moonshot system. `http://www8.hp.com/us/en/products/servers/moonshot/index.html?jumpid=reg_r1002_usen_c-001_title_r0001`.

[14] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. *Local to Global Data Interoperability-Challenges and Technologies, IEEE*, 2005.

[15] M. Mesnier, G.R. Ganger, and E. Riedel. Object-based storage. *Communications Magazine, IEEE*, 41(8), 2003.

[16] OpenStack Swift. `https://docs.openstack.org/swift`.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symposium on Operating Systems Design & Implementations (OSDI)*, 2004.

[18] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[19] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, 2012.

[20] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[21] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better than Interrupt. In *Proc. 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[22] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *USENIX Annual Technical Conference (ATC)*, July 2017.

[23] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: creating application objects efficiently for heterogeneous computing. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

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

[25] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: an appliance for big data analytics. In *Intl. Symp. on Computer Architecture (ISCA)*, June 2015.

[26] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[27] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug

Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proc. 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[28] Apache Hadoop. `http://hadoop.apache.org/`.

[29] Amazon S3. `http://docs.aws.amazon.com/AmazonS3/latest/dev`.

[30] Microsoft Azure Storage. `https://docs.microsoft.com/en-us/azure/storage/`.

[31] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2016.

[32] Open-source MD5 hash HDL code. `https://github.com/stass/md5_core`.

[33] Open-source SHA1 hash HDL code. `https://github.com/secworks/sha1`.

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

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

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

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

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

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

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

[41] 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.

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

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

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

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

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

[47] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[48] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds. In *HotStorage*, 2016.

[49] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *ACM Transactions on Computer Systems (TOCS)*, 2016.

[50] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The stanford flash multiprocessor. In *ACM SIGARCH Computer Architecture News*, volume 22, pages 302–313. IEEE Computer Society Press, 1994.

[51] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash? local flash. In *Proceedings of the Twenty-Second International Conference on*

Architectural Support for Programming Languages and Operating Systems, pages 345–359. ACM, 2017.

[52] Pradeep Kumar and H Howie Huang. Falcon: Scaling io performance in multissd volumes. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association*, pages 41–53, 2017.

[53] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[54] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoo Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proc. 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2016.

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

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

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

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

[59] Lena Oden and Holger 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*, 31(4):274–284, 2017.

[60] Sarah Neuwirth, Dirk Frey, Mondrian Nuessle, and Ulrich Bruening. Scalable Communication Architecture for Network-Attached Accelerators. In *Proc. 21st IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2015.

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

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

[63] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[64] Elena Agostini, Davide Rossetti, and Sreeram Potluri. Offloading communication control logic in gpu accelerated applications. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 248–257. IEEE Press, 2017.

Acknowledgements

우선 지난 6 년 동안 부족한 저를 항상 연구 지도하여 주시고 제가 연구에만 집중할 수 있도록 여러 방면에서 도와주신 김장우 교수님께 감사의 말씀을 드립니다. 비싼 실험 장비들뿐만 아니라 많은 시간과 연구실의 우수한 인력들이 필요한 저의 연구에서 교수님의 전폭적인 지원과 지도가 있었기에 제가 연구를 진행하고 무사히 박사 과정을 마칠 수 있었습니다. 또한 항상 인자하신 모습으로 편안한 연구 환경을 만들어주신 김종 교수님께 감사의 말씀 드립니다. 저의 졸업 논문 제안과 발표 심사를 맡아주시고 조언을 아끼지 않으신 김한준 교수님과 유민수 교수님, 김재준 교수님, 그리고 이재욱 교수님과 김광선 교수님께도 감사의 말씀을 드립니다.

저의 대학원 생활 동안 연구실 동료들이 있었기에 더욱 훌륭한 연구를 하고 좀 더 나은 연구자가 될 수 있었던 것 같습니다. 저와 항상 함께 연구하며 같이 고민하고 서로 부족한 점을 채워준 동업이와, 부족한 저의 연구 진행 및 논문 작성을 도와주신 영석이형과 재원이형, 그리고 저의 연구를 다방면에서 도와준 아민과 동주형, 수헌이 모두 감사드립니다. 본인의 연구에 저를 참여시켜주어 제가 다양한 연구 경험을 쌓는데 도움을 준 준성이형과 평수, 그리고 저의 연구에 항상 관심을 가져주며 연구실에서 함께한 한휘형, 재언이형, 규현, 은진, 헌준, 동문, 일권, 성화, 광무, 다열 모두 감사드립니다. 또한 제가 포항에서 연구를 할 때에 함께하며 좋은 연구 환경을 만들어준 다른 분(상호형, 태호형, 종혁이형, 지훈이형, 승민이형, 형섭이형, 범진이형, 영웅이형, 재혁이형, 상학이형, 나경누나, 하영누나, 상우, 하영, 지원, 영주, 원업, 세영, 윤지, Osama, Xinyu, Hiep, Thinh, Hoa, Binh)들께도 감사드립니다.

행정적인 부분에서 저를 도와주신 이정민 선생님과 포항공과대학교와 서울대학

교의 학과 사무실 선생님들, 그리고 학생 지원팀 선생님들에게도 감사의 말씀 드립니다. 복잡했던 전문연구요원과 기타 여러 행정 업무들에서 선생님들께서 도와주셨기에 큰 문제없이 박사 과정을 마칠 수 있었습니다.

저의 학부 및 대학원 생활 동안 저와 연락을 하며 함께 시간을 나누어준 여러 친구분들께도 감사드립니다. 특히 저와 많은 시간을 함께하며 함께 공부하고 때때로 대회도 함께 출전한 보안 동아리 동기 및 선후배 분들께도 깊은 감사의 말씀 드립니다. 동아리 분들은 저의 학부 시절을 한가득 채워주셨고 저의 졸업 이후에도 큰 힘이 되어주셨습니다. 그들과 함께한 소중한 추억과 경험들은 제 인생에 소중한 보물로 남을 것입니다.

무엇보다 항상 저를 염려해주시고 응원하시고 때때로 직접 찾아와주시기도 하셨던 저의 부모님과 누나에게 깊은 감사의 말씀을 드립니다. 지나왔고, 또 앞으로 있을 제 인생에서 가장 고마운 분들이며 언제나 사랑한다는 말씀을 드리고 싶습니다.

Curriculum Vitae

Name : Jaehyung Ahn

Education

2013 – 2019 **Pohang University of Science and Technology**
Ph.D. in Computer Science and Engineering

2009 – 2013 **Pohang University of Science and Technology**
B.S. in Computer Science and Engineering

Publications

1. Joonsung Kim, Pyeongsu Park, **Jaehyung Ahn**, Jihun Kim, Jong Kim, and Jangwoo Kim "SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-Box SSDs *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2018

2. *Dongup Kwon, ***Jaehyung Ahn**, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim, "DCS-ctrl: A

Fast and Flexible Device-Control Mechanism for Device-Centric Server Architecture *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018. ** both authors equally contributed this work.*

3. Jaewon Lee, **Jaehyung Ahn**, Choongul Park, and Jangwoo Kim, "DTStorage: Dynamic Tape-based Storage for Cost-effective and Highly-available Streaming Service", *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2016.

4. ***Jaehyung Ahn**, *Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim, "DCS: A Fast and Scalable Device-Centric Server Architecture", *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015. ** both authors equally contributed this work.*


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교