Direct Device-to-Device Transfer Protocol: A New Look at the Benefits of a Decentralized I/O Model

Steen Larsen*†, Ben Lee†, Jin-Hyuk Yoon‡, and Jae-Yeun Yun‡

*Intel Corp.
2238 NW Beck Rd, Portland OR 97231
Email: steenx.k.larsen@intel.com.

†School of Electrical Engineering and Computer Science
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97331-5501, USA
Email: benl@eesc.orst.edu

‡SK Telecom
Fusion Technology R&D Center
9-1 Sunae-dong, Pundang-gu, Sungnam
Kyunggi 463-784, Korea
Email: {jhyoon01, jaeyeun.yun}@sk.com

***Abstract*—Current I/O devices communicate based on the PCIe protocol, and by default, all the traffic passes through the CPU-memory complex. However, this approach causes bottleneck in system throughput, which increases latency and power as the CPU processes device specific protocols to move data between I/O devices. This paper examines the cost of this centralized I/O approach and proposes a new method to perform direct device-to-device I/O communication. Our proof-of-concept implementation using NetFPGA shows that latency can be reduced by more than 2x, CPU utilization can be reduced by up to 18%, and CPU power can be decreased by up to 31 W.**

***Keywords*-PCIe; I/O device; system architecture**

I. Introduction

I/O transactions are typically handled using a centralized approach where all I/O data passes through the CPU-memory complex. This approach has a cost in terms of performance as well as power consumption especially for media streaming. This is because typical I/O transactions involve device DMA access to system memory. In order to illustrate the aforementioned cost, Fig. 1 shows the conventional I/O streaming model where two I/O devices, i.e., Solid State Drive (SSD) and Network Interface Card (NIC), communicate with each other, e.g., in a VoD server environment. First, the transmitting device (i.e., SSD) buffers I/O data in the system memory using DMA write transactions, and then requests service from the CPU, usually via an interrupt. Second, the CPU copies the system memory buffer region to the NIC buffer region. Third, the CPU notifies the

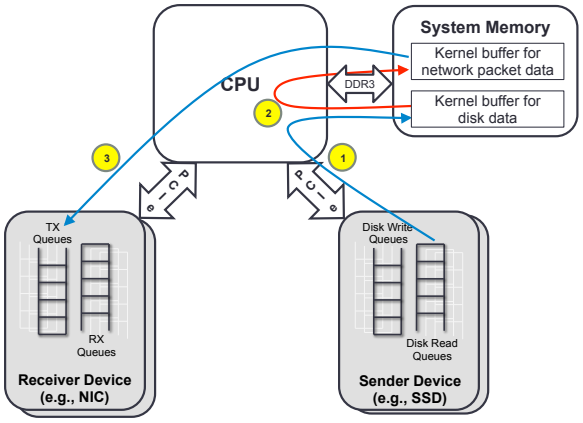

Figure 1. Conventional I/O Streaming

receiving device (i.e., NIC) to perform DMA read transactions. This DMA process is managed with descriptors, which increases overhead and thus latency. In addition, I/O transactions consume CPU resources in the form of cache lookups to maintain memory coherency among the caches and system memory as a part of the descriptor-based DMA processing, which detracts from non-I/O system loads. In terms of power, DMA operations are asynchronous to CPU power policies, and thus I/O transactions will wake the CPU out of its possible sleep states reducing the system power efficiency.

The primary advantage of a CPU-centric model for I/O

transactions is that the control of data (e.g., system security, coherency with other data, etc.) can be maintained by the software by only enabling the trusted DMA engines. Another advantage of the CPU-centric model is the flexibility software provides as protocols evolve (e.g., IDE to SATA and SCSI to SAS). However, the CPU-centric model taxes the CPU and system memory performance and power. Furthermore, as CPUs provide larger core count and greater feature integration, the cost of managing I/O transactions will increase in the form of power, CPU utilization, and latency.

This paper proposes a method called *Device-to-Device (D2D) transfer protocol* that allows I/O devices to communicate directly with each other using PCIe transactions without any CPU involvement. This allows the CPU to drop to a sleep state or execute non-I/O related tasks to save memory bandwidth while reducing latency and improving I/O throughput, particularly for streaming I/O workloads such as VoD servers. Moreover, as in the traditional descriptor-based DMA transfers, the control of data is maintained by having the software enable only the D2D streams that occur between trusted devices.

II. Related Work

The prior work related to D2D can be found in two general application areas: High-Performance Computing (HPC) and server systems.

Among the Top500 supercomputers, 84% of them use descriptor-based DMA I/O protocols, such as Ethernet and InfiniBand [11]. Since D2D eliminates the need for DMA descriptors, it is possible to replaces these traditional interfaces with D2D-enabled devices and thus remove CPU-centric descriptor-based DMA processing in the CPUs.

Among the top 10 supercomputers, eight of them use a variety of custom system interconnects [11]. Ali *et al.* proposed *I/O forwarding* to optimize communications between compute nodes and the interconnection network by using I/O nodes [4]. I/O forwarding is used as a hardware infrastructure for Message-Passing Interface (MPI) variants (e.g., OpenMPI and MPICH2) [6] and other file-based communication methods such as Buffered Message Interface [7] and ZOIDFS [8]. It is also commonly used in system architectures such as the IBM BlueGene/P to reduce OS interference for POSIX kernel I/O system calls [9]. I/O forwarding reduces OS noise in the compute nodes of the system (e.g., context switches, cache poisoning, and interrupts) by using I/O nodes to offload the I/O transaction overhead. Much of the OS noise is due to device-driven interrupts requesting CPU services that result in context switches to I/O software routines, which reduce CPU performance. The proposed D2D also addresses the OS noise, but by removing the descriptor-based DMA transactions that help induce the OS noise.

Although detailed documentations on custom interconnects for the top supercomputers [11] are unavailable, they may well utilize the *PCIe Non-Transparent Bridges* (NTB) technology to communicate between PCIe-based systems [12]. NTB allows I/O transactions to be performed directly across the PCIe switch fabric. The PCIe NTB effectively allows nodes to share memory regions over the PCIe switch fabric without the traditional CPU-centric networking protocol of Ethernet or InfiniBand. This allows a given node to directly access memory of any other node using the CPU's load and store instructions. Although NTB is similar to D2D, its intended application space is the interconnection of multiple CPUs within an HPC, not between I/O devices. Moreover, while both NTB and D2D use the PCIe protocol, NTB relies on a hierarchical protocol to allow separate PCIe root complexes (e.g., systems) to communicate over direct PCIe links. In contrast, the proposed D2D transfer protocol focuses on allowing any PCIe device to communicate with any other PCIe device. D2D uses PCIe bridges to communicate between peer devices, but does not require CPU software to perform descriptor processing for the I/O transaction. In addition, D2D is an open architecture instead of a proprietary custom architecture, meaning an arbitrary D2D device can work with any other D2D device.

For typical server systems, where I/O expandability and types of I/O transactions supported are important, there have been proprietary proposals such as the Toshiba ExaEdge [10] for streaming storage data to a network device. However, ExaEdge is not expandable since it bonds a particular SSD to a particular NIC. This bonding is managed using a processor to move data between the SSD and NIC, which limits its scalability since the sub-processor may not be able to expand beyond the current storage or network configuration. In contrast, D2D is a scalable, open protocol allowing any D2D I/O device to communicate with other D2D I/O devices (e.g., multiple SSDs communicating with multiple NICs).The ExaEdge has been available for two years [**?**], and it has yet to establish a significant market share compared to mainstream server systems that are expandable and scalable. This slow adoption may be an indication that methods to address I/O transaction optimization may be more widely accepted by an open standard approach such as D2D.

III. D2D Transfer Protocol

The proposed D2D transfer mechanism relies on the existing PCIe protocol, which is ubiquitous across various segments of computer systems ranging from personal computers to supercomputers. Figs. 2 and 3 show how PCIe-based SSD and NIC devices, respectively, can be extended with the D2D capability to support the flow of traffic for video streaming (i.e., video servers), which is the application studied in our prototype (see Sec. IV).

The three major components required for D2D transfer from SSD to NIC are (1) *D2D Stream Control Registers* in

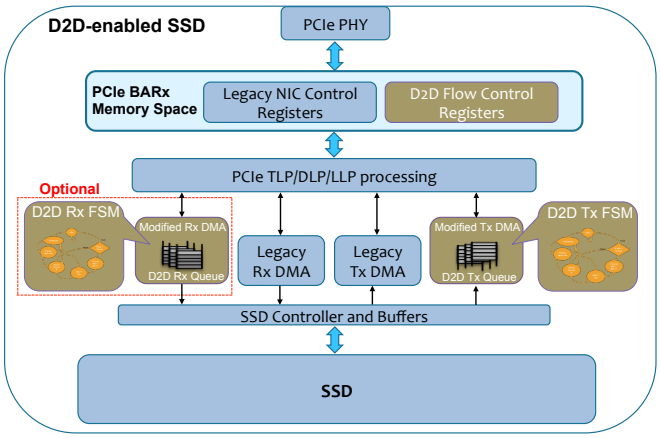

Figure 2. D2D-enabled SSD.

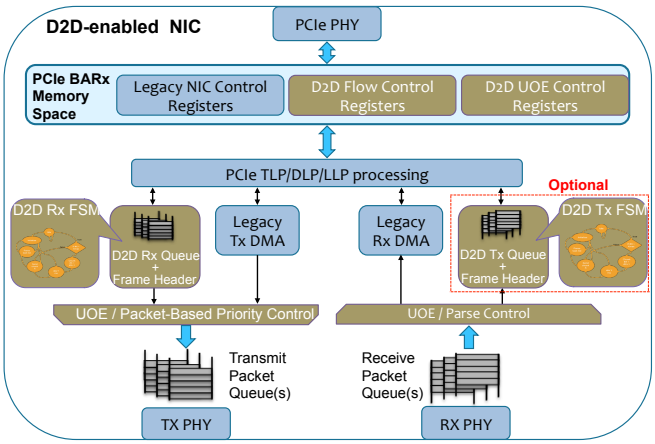

Figure 3. D2D-enabled NIC.

the SSD, (2) *D2D Tx Finite State Machine* (FSM) and *Tx Queue* in the SSD, and (3) *D2D Rx FSM* and *Rx Queue* in the NIC. In addition, the NIC requires *D2D UDP/IP Offloading Engine* (UOE) and *UOE Control Registers* for UDP-based video streaming. While our prototype only supports UOE, a TCP Offload Engine (TOE) can also be included to support TCP-based streaming.

Note that *D2D Rx FSM* and *Rx Queue* for SSD and *D2D Tx FSM* and *Tx Queue* for NIC are not required for a D2D stream from SSD to NIC. Other D2D-based devices can also be designed using the D2D transfer protocol as defined in this section. In addition to making I/O devices D2D capable, some changes are needed in the device driver software to enable D2D streaming. Finally, D2D-enabled devices are fully backward compatible, and thus the legacy functions will continue to operate with no changes to software or other hardware.

The following three subsections discuss these components in the context of a D2D stream from SSD to NIC. Afterwards, Sec. III-D discusses how the NIC and SSD can be configured to support the reverse stream, i.e., NIC to SSD.

A. Configuration of D2D Stream Control Registers and Tx/Rx Queues

Before D2D communication can occur, D2D Stream Control Registers for both the sender and receiver devices need to be properly configured. In addition to the standard NIC control registers, Table I defines all the D2D Stream Control Registers. These registers together with the standard PCIe control registers (not shown) facilitate both D2D and legacy I/O transactions.

Tx and *Rx Addresses* represent the transmitter and receiver addresses, respectively. The device driver writes Rx Address of the D2D receiver device into the Tx Address register of the D2D transmitter device. This address is defined by the BIOS on system bootup. As described by the PCIe specification [1], the device after reset specifies the memory

D2D Stream Control Register Definition	
Registers [64b]	Description
Tx Address	The D2D-enabled sender device performs PCIe writes to this address for the D2D stream (bits [63:2]). Writing a one to bit 0 resets the D2D stream. Writing a one to bit 1 clears D2D Tx/Rx Queues. Bit 2 is reserved.
Rx Address	The address of the D2D-enabled receiver device for the D2D stream (bits [63:2]). This address is used by the D2D driver software to configure the Tx Address register of the D2D-enabled sender device. Bits [2:0] are reserved
D2D Data Rate (B/s) & Granularity (s)	The streaming data rate in bytes per second and the chunk size in sec.
Tx and Rx Base Credits	Credits supported by the I/O device for transmit and receive transactions.
D2D Transmit Byte Count	A counter initialized with the number of bytes that are to be streamed by D2D.
Tx Credit Update Address	The address to write new credit grants. D2D Rx FSM can write a new credit grant to this address when there is space available in D2D Rx Queue.

Table I
D2D STREAM CONTROL REGISTER DEFINITIONS.

region size required for each device to function. The BIOS then maps this device address range into the overall system address space. The location of the Tx Address register is simply an offset into the PCIe BAR address space, and it varies depending the system configuration and operating system boot sequence. D2D uses Tx and Rx Queues that are instantiated as flat memories in the I/O device. As such, Tx and Rx Addresses do not change once a D2D session has been initialized.

Occasionally, the stream may need to be reset, stopped, or reinitialized. The lower 3 bits of the 64-bit address field are used for this purpose. When the 0^{th} bit of Tx Address is set to one, the D2D device resets the D2D stream. When the 1^{st} bit of Tx Address is set to one, the Tx and Rx Queues will be emptied. The 2^{nd} bit is reserved for pausing the stream.

The *Data Rate* for a stream can be defined in terms of bytes per second. Moreover, its *Granularity* can be defined in terms of time-slot per stream chunk. For example, a 20 Mbps

stream with 4 KB chunks has a time interval of 1.64 ms, thus SSD would write 4 KB of VoD data to NIC every 1.64 ms. As the chunk size decreases, the time interval decreases and the D2D write frequency increases. This mechanism is discussed further in the PCIe specification [1].

Tx and *Rx Base Credits* define the flow control of a stream. Upon reset, Rx Base Credit is initialized based on the size of D2D Rx Queue of the receiver device (i.e., D2D-enabled NIC). These credits are read and programmed by the device driver to prevent buffer overflow/underflow. For example, D2D-enabled NIC may initialize its Rx Base Credit at a quantity of 32 credits, where each credit is equivalent to 4 KB of data. The D2D-enabled SSD device would then have its Tx Base Credit set to 32, which corresponds to D2D Tx Queue size of 128 KB. Afterwards, each 4 KB of data sent by SSD to NIC reduces the credit counter by one. When the credit counter reaches zero, D2D transfer stalls until Tx Base Credit in D2D-enabled SSD is updated with a positive value.

There are two approaches to updating Tx Base Credit in a D2D-enabled SSD. The simplest and most flexible method is to interrupt the CPU and rely on the D2D driver software to appropriately update Tx Base Credit. The other method is to have D2D-enabled NIC directly update Tx Base Credit as space becomes available in D2D Rx Queue. This is achieved by having D2D Rx FSM perform a PCIe write to the address defined in the *Tx Credit Update Address* register with an appropriate credit value. The frequency of Tx Base Credit updates depends on various factors, such as the type of data being passed between D2D-enabled devices and the D2D Rx Queue size. While this approach adds no CPU load throughout the stream, there may be significant number of PCIe transactions for credit exchange if the D2D Rx Queue is small. As a result, the D2D Rx Queue should be at least several kilobytes in size. The exact Rx Queue size is determined by a tradeoff among the D2D Rx Queue silicon footprint, power, and available PCIe bandwidth.

Finally, *D2D Transmit Byte Count* defines the total number of bytes to be transferred. This is needed by D2D Tx FSM to determine when the D2D stream is finished.

Note that the D2D Stream Control Registers described above are for a single streaming session. The set of control registers will need to be replicated to support multiple concurrent streaming sessions. The amount of register space supported is defined by the BAR register mapping and depends on the space available on the FPGA and the external memory. In our implementation, each streaming session requires only 48 bytes of register space.

The memory-mapped D2D Rx Queue is referenced by a single address. D2D-enabled sender devices (and the CPU) write to this address to queue D2D transactions. In the case of D2D-enabled NIC, the D2D Rx FSM is triggered when there is enough data in D2D Rx Queue for an Ethernet packet (typically 1500 bytes). Then, the D2D Rx FSM stores the data in a temporary frame buffer, which is used by UOE to perform frame related calculations such as checksum.

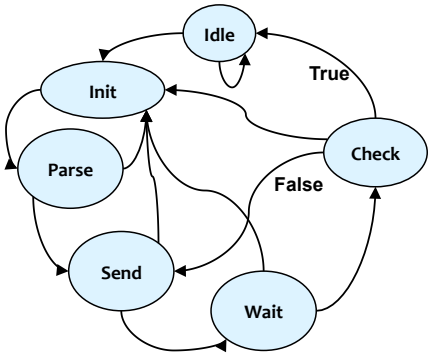

Figure 4. D2D Tx FSM.

D2D Tx Queue is basically a buffer for data to be transmitted by the D2D Tx FSM. In the case of a D2D-enabled SSD, D2D Tx Queue will hold the sequence of blocks that the D2D-enabled SSD device will send to a D2D-enabled NIC. The D2D Tx FSM specified in Sec. III-B parses the job requests from D2D Tx Queue and then fetches data from the SSD. Since a D2D stream provides chunks of data in SSD block sizes, the SSD's D2D Tx FSM writes data directly from the SSD controller to the PCIe interface based on the address of a D2D-enabled NIC (i.e., NIC's) Rx Address.

B. D2D Tx FSM

The D2D-enabled sender device (i.e., SSD) needs to support D2D Tx Queue, and D2D Tx FSM. Note that D2D Tx FSM will be slightly different for different types of D2D-enabled sender devices. Our discussion is based on D2D-enabled SSD.

The D2D Tx FSM is shown in Fig. 4. In the *Init* state, the driver reads the Rx Address register from the NIC and writes it to the Tx Address register of the SSD. Since SSD is addressed in block format, the necessary support for translation or memory-mapped access will be needed. The list of the data blocks also needs to be converted into a task list so that D2D Tx FSM can fetch data from the SSD for transmission.

The number of bytes to be sent is written to the D2D Transmit Byte Count register in the SSD. Based on the VoD streaming requirements, such as latency sensitivity and D2D Rx Queue size, the Data Rate and Granularity parameters are calculated and programmed into the Stream Control Registers of the SSD. The initialization of Tx and Rx Base Credits is also done by setting the registers to zeros. Afterwards, the D2D stream is started by writing a positive value to the Tx Base Credit register and setting the 0^{th} bit of the Tx Address register.

After the initialization, D2D Tx FSM transitions to the *Parse* state where the data to be transmitted during the

streaming session is defined. Since the SSD block addresses from the VoD streaming application are non-contiguous, a method is needed to reference the SSD data for transmission. This is accomplished by storing the SSD block addresses in the D2D Tx Queue. Since logical to physical address translation is needed, the *Parse* state maps the operating system block addresses to the SSD physical sector addresses. The *Parse* state is performed at the beginning of the D2D stream to avoid any CPU involvement during the stream session.

The specific architecture of the SSD or other storage devices will define the exact lower level SSD read operations. Generally, the *Send* state will fetch a block of SSD data based on the address at the head of D2D Tx Queue and forward it to the PCIe interface. For example, if the SSD block size is 4 KB, the *Send* state would read the address of the 4 KB block from the head of D2D Tx Queue and generate the PCIe memory write transactions for the address defined in the Tx Address register. The size of each PCIe memory write operation is determined by the system architecture, which is usually either 128 or 256 bytes per PCIe frame. Moreover, these write bursts will be based on the value defined in the D2D Granularity field. For example, with a default chunk size of 4 KB and PCIe frame MTU of 256 bytes, there will be bursts of 16 PCIe memory writes with the frequency defined by D2D Data Rate.

After a chunk is sent, D2D Tx FSM transitions into the *Wait* state until the next chunk needs to be sent. After waiting some predefined time of *Granularity* in μs, D2D Tx FSM transitions to the *Check* state, where the total number of bytes written to the NIC is compared with the value set in the D2D Transmit Byte Count register. If they are equal, D2D Tx FSM stops transmitting and transitions to the *Idle* state; otherwise, a transition is made to the *Send* state to send another chunk. The *Granularity* parameter modulates the higher PCIe bandwidth to the bandwidth delivered by the stream session.

At any point in D2D Tx FSM, events such as pause or replay will trigger the state machine to revert to the *Init* state. When this occurs, the application may re-program the D2D Stream Control Registers to start a new D2D transaction.

C. D2D Rx FSM and UOE

The D2D-enabled receiver device (i.e., NIC) needs to support D2D Rx Queue, D2D Rx FSM, and UOE. Again, the design of D2D Rx FSM will depend on the type of D2D-enabled receiver devices. Our discussion is based on D2D-enabled NIC with UOE. Moreover, UOE is an integral part of the D2D Rx FSM, thus their operations will be explained together.

D2D Rx FSM is shown in Fig. 5. In the *Init* state, the driver initializes *D2D UOE Control Registers* so that the UDP header information can be properly attached as data is packetized for network transmission. These fields

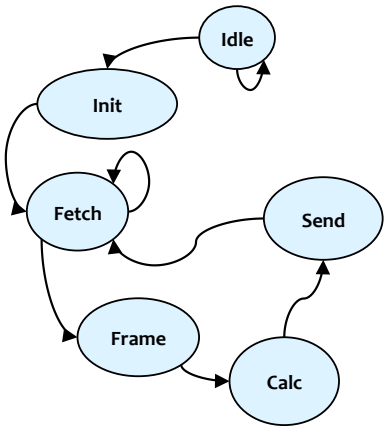

Figure 5. D2D Rx FSM.

are defined in Table II. The *MAC Source address* is the MAC address of the transmitter device (i.e., SSD). The *MAC Destination address* is the MAC address of the receiver device (i.e., NIC), which is made available using the TCP session that initiated the streaming session. Similarly, the *Source IP* address is the IP address of the transmitter device and the *Destination IP* address is the IP address of the receiver device. The *UDP Destination Port* is an agreed-upon port based on the TCP session that initializes the streaming session. The other fields are self-explanatory and static throughout the streaming session. The exceptions are Ethernet length, Ethernet CRC, IP length, IP checksum, UDP length, UDP checksum, RTP timestamp, and RTP sequence number. Each of these fields are calculated on a per-packet basis and written by the D2D Rx FSM.

Once these registers are configured, D2D Rx FSM enters the *Fetch* state where D2D Rx Queue is monitored for received data. When there is sufficient amount of data in D2D Rx Queue to a transmit packet (typically 1500 bytes), D2D Rx FSM transitions into the *Calc* state where packet-specific details are assigned to the packet header, such as MAC and IP addresses. The *Calc* state is part of the UOE logic that calculates the Ethernet length, Ethernet CRC, IP length, IP checksum, UDP length, UDP checksum, RTP timestamp, and RTP sequence number. The explanation of the UOE operation will be explained shortly. After the packet has been fully formed, it is queued in the legacy *Transmit Packet Queue* for network transmission. Afterwards, D2D Rx FSM transitions to the *Check* state, where the packet is dequeued and sent to the MAC layer for final framing. The Rx FSM will wait in the *Fetch* state for data until the stream is reset.

The D2D-enabled NIC also has a module to differentiate D2D packets from legacy packets. During the *Send* state operations, the *Packet-Based Priority Control* module arbitrates between D2D and legacy packets with priority given to the latter. This is because there is stream control information that are carried over legacy packets that have higher priority

D2D NIC UOE Configuration Registers	
Registers [64b]	Description
MAC Dst [6B]	6-octet destination MAC address for the stream.
MAC Src [6B]	6-octet source MAC address for the stream.
802.1Q [4B] & Length [2B]	Optional setting for 802.1Q tag. The length is calculated by the NIC based on data size read from D2D RX Queue
IP Header1	First 8 bytes of IP header including Version, IHL, DSCP, ECN, length, ID, and flags. Only the length [2B] is calculated by NIC based on the size of the D2D RX Queue entry. All other fields are set during D2D initialization
IP Header 2	TTL, protocol, header checksum, and source IP address. Only header checksum [2B] is calculated by NIC, and all others are static based on D2D stream initialization.
IP Header 3	Destination IP address and UDP option fields. Static based on D2D stream initialization values except for the UDP length [2B] field that is calculated by the NIC based on the length of the UDP header and data.
UDP Header	UDP source and destination port, length and checksum. UDP length [2B] is calculated by NIC as the sum of UDP header and data. UDP checksum [2B] is calculated by NIC for both header and data.
RTP Header 1	Version and bit definitions, sequence number, and timestamp. Timestamp is tracked by D2D-enabled NIC and written based on when the RTP packet leaves the NIC buffer if needed. Sequence number is set on initialization and incremented per RTP packet.
RTP Header 2	SSRC and CSRC identifiers. Specified by the driver during initialization.

Table II
D2D UOE CONTROL REGISTERS.

than D2D packets.

Finally, the NIC UOE engine takes data from the head of the Rx Queue and applies proper framing protocol to generate an Ethernet packet based on the UOE control registers defined in Table II. This packet is then sent to the NIC PHY for transmission.

D. D2D Stream: NIC to SSD

The previous subsections discussed the D2D streaming of data from an SSD to NIC. This subsection discusses the reverse stream from NIC to SSD to illustrate the flexibility of the D2D transfer mechanism for other applications. A typical example could be network storage services, where the D2D-enabled NIC would utilize its D2D Tx FSM and Tx Queue (D2D Rx FSM and Rx Queue would be optional). In addition, the UOE and *Parse Control* module are needed filter and process the received packets and enqueue them in the D2D Tx Queue. On the other hand, the SSD would require the D2D Rx FSM and Rx Queue (D2D Tx FSM and Tx Queue would be optional).

For a D2D-enabled NIC, UOE appropriately parses the headers of incoming network packets and enqueues them in D2D Tx Queue based on the packet header information. This means that the incoming network packets need to be filtered before any D2D or legacy DMA transactions can occur. This is accomplished by the Parse Control module that distinguishes between received legacy network and D2D packets based on the D2D UOE Control Registers in

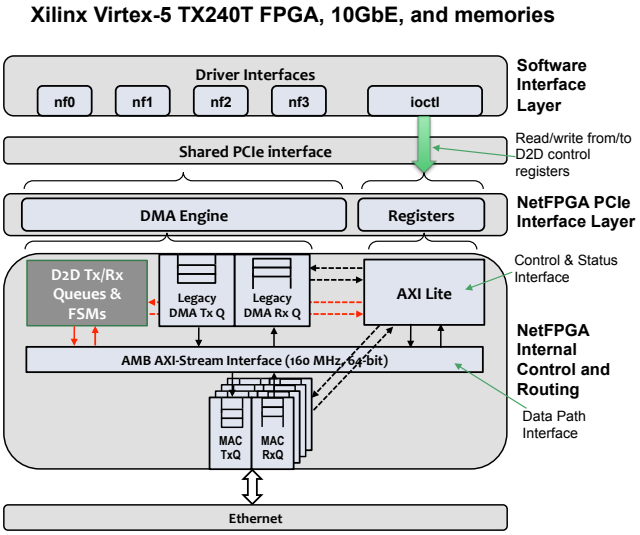

Figure 6. NetFPGA design hierarchy.

Table II. If an incoming packet is determined to be a valid D2D stream packet, the legacy DMA is bypassed and the packet is written into D2D Tx Queue. For a D2D-enabled NIC, the D2D stream rate is determined by the rate at which the network traffic is received. Therefore, D2D Tx FSM can immediately send the received network packets to other D2D-enabled devices.

A special consideration is needed for IP layer packet fragmentation. If IP fragmentation is used and the IP fragments are not reassembled properly by the receiving system, video streaming applications such as VLC cannot properly display the video stream. As a result, UOE on the D2D-enabled NIC has to parse the incoming packets for proper reassembly of the fragmented packets.

IV. PROOF-OF-CONCEPT EVALUATION

This section discusses the implementation and evaluation the proposed D2D transfer protocol discussed in Sec. III.

A. NetFPGA Implementation

The D2D transfer protocol was implemented using a NetFPGA board [14], which has Xilinx Virtex-5 TX240T FPGA, 4×10GbE, and memories. A block diagram of a D2D-enabled NIC design using NetFPGA is shown in Fig. 6. The *AMB AXI-Stream Interface* is used to interface and route the four 10GbE MACs with a single DMA engine. The DMA engine controls all the PCIe traffic to and from the host system. The driver software interfaces to the nf0, nf1, nf2, nf3 ports to allow for processing of the higher layers of the networking protocol. A basic direct PCIe register interface, which does not use DMA to transfer data, is also available in the NetFPGA architecture to access the D2D configuration registers defined in Tables I and II using the common ioctl interface. *AXI Lite* is the interface for the microcontroller

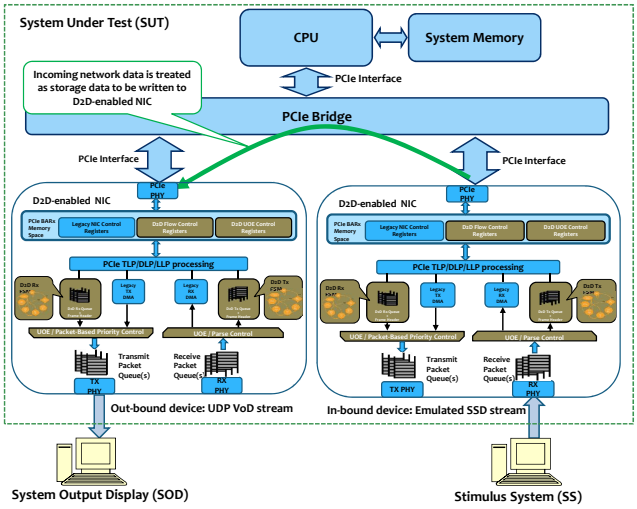

Figure 7. D2D prototype with emulated SSD.

and other internal components on the FPGA, and is used to initialize the MAC configuration.

Each 10GbE MAC has an outgoing master AXI-Stream interface and an incoming slave AXI-Stream interface. The AXI-stream interface is 64-bit wide and operates at 160 MHz allowing for full 10GbE throughput. The UOE must therefore process both transmit and receive network packets at 160 MHz. Since the Xilinx PCIe interface used by the DMA engine is 64 bits, both the master AXI-Stream interface and the D2D Rx Queue entries are 64 bits.

The NetFPGA PCIe interface was modified to support the D2D transfer protocol by adding the D2D Tx and Rx FSM logic and queues and the UOE logic. The rest of the NetFPGA design shown in Fig. 6 is left unchanged.

B. Evaluation of Methodology

The evaluation of the D2D transfer protocol was performed using two NetFPGA cards connected to a host system as shown in Fig. 7. The *System Output Display* (SOD) representing a client requests a VoD stream from the *System Under Test* (SUT), which represents a VoD server. This results in video stream being sent from *Stimulus System* (SS) via the SUT to the SOD. In this setup, the SS together with the in-bound D2D-enabled NIC emulate an SSD. The SS will have a preloaded video file for streaming over to the SUT for D2D performance measurements.

The host system contains an Intel 2500K quad-core CPU running at 3.1 GHz. The streamed video has a resolution of 720×480 @29 fps with an average bit rate of 7,820 kbps. The audio stream is also part of the video stream requiring 320 kbps of bandwidth. The VLC application is running in server mode on the SS and uses MPEG-2 packets [15], which is the only UDP streaming configuration currently supported by VLC.

The SOD initiates a VoD stream by requesting service from the CPU in the SUT on a specific UDP port using a TCP connection. In our prototype, this is executed as an *ssh* Linux operating system command. The CPU in the SUT responds by requesting the UDP video stream from the SS using a TCP connection. The SS then responds by running a VLC application to start an MPEG-2 video stream on the specified UDP address of the SUT. We have successfully performed video streaming on the testbed shown in Fig. 7, and an example run can be seen at https://www.youtube.com/watch?v=qgGCPHDMbK0.

The SUT receives UDP Ethernet frames from the SS on the in-bound D2D-enabled NIC and strips the transport, Internet, and link layer header information. This is done by the UOE based on the D2D Tx FSM (see Sec. III-B) and the D2D Stream Control Register parameters (see Sec. III-A). After extracting the MPEG TS segments, the D2D emulated SSD performs PCIe memory writes to move the TS data to the memory address space of the out-bound D2D-enabled NIC as specified by the Tx Address (see Table I). The out-bound D2D-enabled NIC executes the D2D Rx FSM and stores the received PCIe data into the D2D Rx queue. When there is a sufficient amount of data for a network packet, the D2D-enabled NIC Rx FSM forms a TS-aligned Ethernet frame. This frame has header information defined by the UOE of the D2D-enabled NIC, which is based on the D2D UOE configuration parameters (see Table II). After the UOE task is complete, the entire frame is transmitted to the SOD. Finally, the SOD receives the Ethernet packet stream and displays it using a VLC client application.

In our prototype, the D2D stream configuration parameters are hard coded, but ultimately these parameters would be programmed into D2D-enabled devices in the SUT using device driver calls.

C. Performance Evaluation and Results

1) Latency: In order to compare the latency of D2D transactions with conventional descriptor-based DMA transactions, the measurements obtained from our prototype is compared with the results obtained by Larsen *et al.* [2]. This study showed that on two recent servers, each with a 10GbE NIC connected back-to-back using the well known *netpipe* latency test, the descriptor-based DMA latency required for a 64-byte packet to be received, processed by the CPU, and transmitted is 11,906 ns. A large part of this delay can be removed with D2D since the emulated SSD in our prototype performs PCIe memory write transactions directly to the 10GbE NIC.

In order to analyze the latency of the D2D prototype, the delay between when the in-bound NIC (i.e., the emulated SSD) receives a 1500 byte VoD packet from the SS and when it is processed by the out-bound D2D NIC was measured. Fig. 8 shows the various latency components based on the measurements obtained using Xilinx Chipscope.

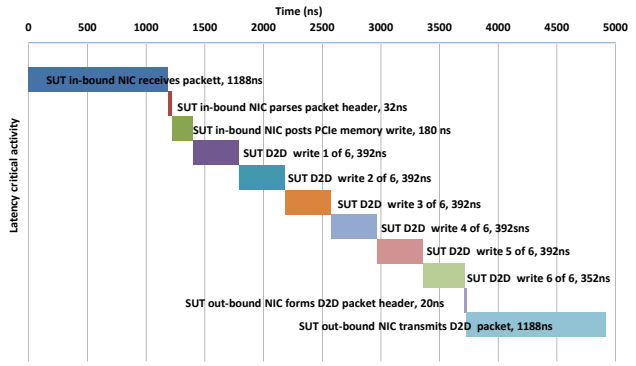

Figure 8. Example 1500 byte packet latency in the D2D prototype

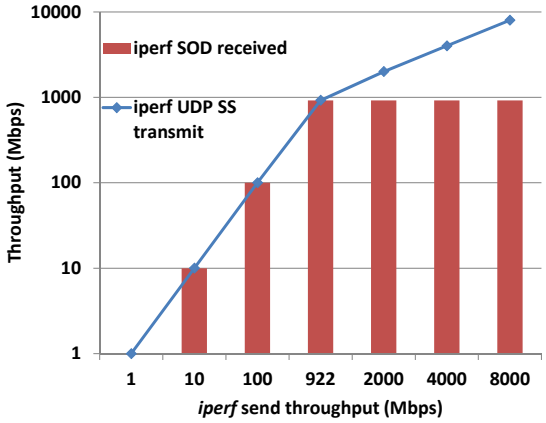

Figure 9. D2D *iperf* UDP throughput measurements

The NetFPGA AXI Stream Interface is 8 bytes wide and operates at 160 MHz, so a 1500-byte packet with a 20-byte Ethernet header requires 1,188 ns to move between the 10GbE PHY and the PCIe DMA engine. Once the packet is in the DMA engine, it is in the FPGA internal memory and it can be determined if it is a D2D packet for D2D transfer or a legacy packet for CPU-centric descriptor-based DMA processing. The PCIe interface operates at 125 MHz with the maximum PCIe packet frame size of 256 bytes. Our Chipscope measurements show that each 256 byte PCIe frame write operation, including the PCIe protocol header information, requires 392 ns. Note that the last PCIe write is a fragment, and thus it only takes 352 ns. After the Ethernet and UDP/IP protocol information is removed, the 1500-byte packet requires six PCIe frames for a total PCIe latency of 2,352 ns. Therefore, the total latency to transfer across both PCIe devices in our prototype is 4,959 ns. Comparing this latency to the 11,906 ns latency discussed in [2], which is for a smaller 64-byte packet, D2D provides more than 2x latency improvement. Note that using D2D for a smaller 64-byte packet requires only one PCIe write operation, which reduces latency by up to 9x. Also note that the PCIe protocol implementation on the FPGA is likely not as optimized as an ASIC implementation, which would have better latency characteristics.

2) Throughput: The *iperf* bandwidth test tool [16] was used to stream UDP packets from the SS to the SOD using D2D on the SUT. VoD streaming was not used because the inter-packet delay of the video streams cannot be controlled under VLC. As a result, some clusters of video packets with small inter-packet delays may exceed the throughput limitation of our prototype (see the discussion in Sec. V) causing video packets to be dropped.

The resulting measurements are shown in Fig. 9, where the red bars shows the data throughput received by the SOD and the blue curve shows the data throughput sent by the SS. As can be seen, the two results match until the throughput reaches 922 Mbps. UDP data rates higher than 922 Mbps resulted in dropped packets, which is due to the throughput limitation of the NetFPGA board used for our implementation (see Sec. V). Repeating the tests with the SUT in a router configuration using standard off-the-shelf Intel 82599 10GbE NICs shows that the received throughput for the SOD matches with the transmitted throughput of SS until 9.8 Gbps, which is the maximum throughput that includes the Ethernet protocol overhead. Although the throughput results for D2D were low due to the limitation of the NetFPGA board, it is important to emphasize that there were no CPU cycles used during this test.

3) Power: To measure CPU power, an ammeter was placed across the 12 V power rail that powers the CPU and the voltage regulator. The CPU voltage regulator itself consumes power to convert the 12 V power rail to about 1.1 V for the CPU power pins. Thus, the CPU voltage regulator power increases as the CPU power increases. To compare D2D power dissipation against a reference, the SUT was converted to act as a Linux-based IP router between the two IP subnets containing the SS and the SOD. In other words, the emulated SSD was essentially converted to an NIC receiving VoD streams and the SUT is used to route the video stream to the SOD. In this router configuration, the CPU will consume power as it processes the TCP/IP protocol and data is passed from the in-bound NIC to the out-bound NIC using descriptor-based DMA transfers.

Fig. 10 shows the CPU and CPU regulator power dissipation as a function of number of streams, where each point is an average of 10 measurements. The SS transmits multiple 7.82 Mbps video stream files to the SUT using *iperf*. Note that the maximum of 120 video streams corresponds to the maximum throughput in our test environment. The average CPU power dissipation for D2D is only 4.32 W since there is no network traffic is being processed by the CPU. In contrast, the average CPU power for the SUT router configurations is 32 W. The SUT router configurations was also tested up to 9.8 Gbps, and it showed a similar trend of

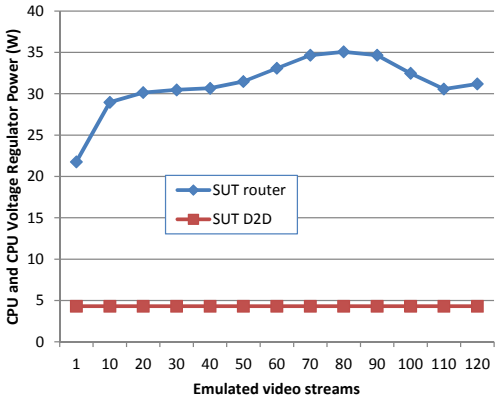

Figure 10. D2D CPU power benefit

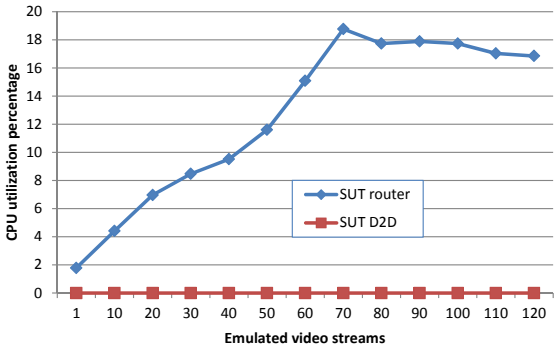

Figure 11. D2D CPU utilization benefit

around 32 W for CPU power. Although D2D was not able to handle more than 120 video streams, its power would expect to continue the sub-5W trend.

These results show that there is almost no CPU power increase from the idle state since most of the time the cores, caches and even the entire CPU are in sleep states during D2D streaming. The amount of time a task sleeps is usually represented as a ratio of time when the given CPU core is active and when it is in one of the possible sleep states. A high ratio indicates that the CPU is sleeping most of the time, and thus it correlates with lower power. D2D allows for a high ratio of sleep states to be maintained, since there is no CPU activity required during the VoD session. As soon as system memory DMA operations and interrupts are required for network routing, the percentage of the time the CPU can be put into sleep is reduced resulting in higher CPU power dissipation.

The SUT router configuration results in Fig. 10 also show that the CPU power dissipation is relatively independent of video stream count over 10 streams because the CPU is held in an active state, i.e., a very low sleep ratio. This is because the parts of the CPU that are monitored by PCU are not idle for sufficient periods of time to be put to sleep resulting in a relatively constant CPU power dissipation for more than 10 video streams

The SUT D2D results show that D2D scales very well as the number of video streams increases since the only CPU interaction is the session control (i.e., setting up the D2D streams and side-band control such as stopping or forwarding a video stream). The SUT router also scales well, but as the stream count increases at some point need more CPUs will be needed.

Although D2D could support thousands of streaming sessions with a single CPU, the actual number of sessions that can be supported will depend on how frequently sessions are started and stopped. For instance, D2D can service long video streams more effectively than short video streams. This is because the CPU-centric descriptor-based model needs to perform both the stream management as well as move the stream data, while D2D only requires the stream management allowing the CPU context switch to other tasks or to save power.

4) Utilization: Fig. 11 shows the CPU Utilization as a function of number streams, which was analyzed using the Linux process monitoring program *top* [17]. Again, these results are based on the average of 10 measurements. As expected, the CPU utilization for D2D is close to 0%. For the SUT router configuration, there is correlation between CPU utilization and network throughput (for 1 and 80 video streams), since the amount of descriptor processing required increases as traffic increases. With network traffic higher than 70 video streams, CPU utilization becomes relatively constant possibly due to NIC default acceleration functions (e.g., LRO, TSO, NAPI, etc.). These results show that up to 18% utilization of the 4-core CPU is saved using D2D.

V. Implementation and Performance Issues

Our proof-of-concept implementation of the D2D transfer protocol had some limitations. The biggest issue is the PCIe throughput limitation of the NetFPGA board. Although there have been reports of up to 1.5 Gbps throughput from the NetFPGA board [14], the maximum verifiable throughput of the D2D prototype with two NetFPGA cards was only 922 Mbps. Most of the prior studies using NetFPGA have been on network routing related research. These kinds of studies do not require high throughput on the PCIe interface since the routing among the four 10GbE Ethernet ports is performed on the NetFPGA. As a result, the throughput measured on our D2D prototype is not an issue for most NetFPGA users. Although no root cause has been reported by the NetFPGA community for the low PCIe throughput and no solutions have been proposed to solve this problem [14], we feel the issue is due to a variety of factors including:

- The Xilinx Virtex 5 silicon is 8 years old, and there may be some inherent limitations on its hard IP implementation of the PCIe interface. Our Xilinx Chipscope debug

traces shows that there is a clock timing issue (i.e., a setup-hold violation within the FPGA logic) at data rates higher than 922 Mbps, where one or two bytes are sometimes corrupted on the 64-bit data interface of the hard IP implementation of the PCIe protocol. We do not suspect that there is a problem with the D2D logic implementation since this issue is not seen on the entire 64-bit data interface and does not occur for lower data rates.

- The Xilinx PCIe endpoint logic supports only a single outstanding PCIe transaction. Other PCIe devices, such as Intel 10GbE, support 24 outstanding PCIe transactions allowing for pipelining of multiple DMA tasks.
- The NetFPGA DMA logic supports only a single network queue, while other NICs, such as Intel 10GbE, support 64 or more queues allowing multiple IP sessions to occur concurrently.

VI. Conclusion and Future Work

The open and scalable D2D transfer protocol offers significant benefits compared to the CPU-centric descriptor-based DMA operations used in current server and HPC environments. Our results show three primary areas of I/O transaction performance improvement. First, the latency between I/O devices in a server is reduced by at least 2x. Second, CPU power is reduced by up to 31 W. Third, up to 18% of a 4-core CPU cycles are made available for servicing other tasks. The D2D transfer protocol is based on the standard PCIe specification, and thus, it is scalable to many devices and application models, such as HPC, servers, and mobile devices.

Our future plan is to expand D2D to support different devices and more complex networking scenarios.

Acknowledgment

This research was supported in part by SK Telecom, Fusion Technology R&D Center, Korea.

References

[1] PCIe Base Spec 3.0. [Online] Available: http://pciesig.com

[2] S. Larsen, P. Sarangam, R. Huggahalli, S. Kulkarni, "Architectural Breakdown of End-to-End Latency in a TCP/IP Network," in *International Journal of Parallel Programming,* Dec. 2009, Vol. 37, Issue 6, pp. 556–571.

[3] C. Maciocco, "Power breakdown analysis of iPad operations," Intel Labs study 2012 unpublished

[4] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward. "Scalable i/o forwarding framework for high-performance computing systems," in *CLUSTER*, 2009

[5] "Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing" [Online] Available: http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing

[6] "MPI Forum, MPI-2: Extensions to the Message-Passing Interface", [Online] Available: http://www.mpi-forum.org/docs/docs.html, 1997.

[7] P. H. Carns, W. B. Ligon III, R. Ross, P. Wyckoff, "BMI: A network abstraction layer for parallel I/O," in *IEEE International Parallel and Distributed Processing Symposium, Workshop on Communication Architecture for Clusters,* Denver, CO, Apr. 2005.

[8] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* Salt Lake City, UT, 2008, pp. 153-162.

[9] "IBM, Overview of the IBM Blue Gene/P project," in *IBM Journal of Research and Development,* vol. 52, no. 1/2, pp. 199-220, 2008.

[10] White Paper, "ExaEdge: Advanced Distribution for Rapid HTTP Streaming Delivery." [Online] Available: https://www.webcom.toshiba.co.jp/snis/en/whitepaper.php

[11] Top500. 2012; [Online] Available: http://i.top500.org/stats

[12] J Regula, "Using Non-transparent Bridging in PCI Express Systems." [Online] Available: http://www.plxtech.com/files/pdf/technical/expresslane/NontransparentBridging.pdf

[13] PCIe Specification, rev. 3.0, Appendix A Isochronous Applications; [Online] Available: http://www.pcisig.org

[14] NetFPGA. [Online] Available: http://netfpga.org

[15] Video LAN client and server Available: http://www.videolan.org/

[16] Iperf network throughput utility Available: http://en.wikipedia.org/wiki/Iperf

[17] Task manager for system resources Available: http://en.wikipedia.org/wiki/Top_(software)