AN ABSTRACT OF THE DISSERTATION OF

Steen K Larsen for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on March 5, 2015.

Title: Offloading of I/O Transactions in Current CPU Architectures

Abstract approved: _____

Ben Lee

I/O transactions within a computer system have evolved along with other system components (i.e., CPU, memory, video) from programmed I/O (PIO). In current mainstream systems (spanning from HPC to mobile) the I/O transactions are CPU-centric descriptor-based DMA transactions. The key benefit is that slower I/O devices can DMA write system receive traffic to system memory and DMA read system transmit data at slower device throughput relative to the CPU. With the advent of more cores in a CPU, power restrictions and latency concerns, we show this approach has limitations and based on measurements we propose alternatives to descriptor-based DMA I/O transactions. We explore and quantify performance improvement in three options:
1) iDMA: Embedded smalller core to offload DMA descriptor processing from the larger application-oriented cores, reducing latency up to 16% and increasing bandwidth per pin up to 17%.
2) Hot-Potato: Where latency is a concern we re-visit using WC-buffers for direct I/O CPU transactions and avoiding CPU hardware changes. While keeping a specialized receive I/O device DMA engine, we reduce latency for small messages by 1.5 µs.

3) Device2Device: For applications moving data between devices, we propose how to bypass the CPU, improving latency, power, and CPU utilization.

©Copyright by Steen K Larsen
March 5, 2015
All Rights Reserved

Offloading of I/O Transactions in Current CPU Architectures

by

Steen K Larsen

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented March 5, 2015
Commencement June 2015

Doctor of Philosophy dissertation of Steen K Larsen presented on March 5, 2015.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

---

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Steen K Larsen, Author

ACKNOWLEDGEMENTS

To Denica, Hans, Heath, Hope and other lights in my life.

TABLE OF CONTENTS

Page

1 General Introduction 1

2 iDMA manuscript 6

2.1 INTRODUCTION . . . 7

2.2 BACKGROUND . . . 9

2.2.1 IO flow using descriptors . . . 12

2.3 The Proposed Method . . . 14

2.3.1 iDMA Transmit Flow . . . 17

2.3.2 iDMA Receive Flow . . . 19

2.4 Experiments and Analysis . . . 21

2.4.1 Latency . . . 21

2.4.2 Throughput/Bandwidth efficiency . . . 26

2.4.3 Other benefits . . . 28

2.5 RELATED WORK . . . 29

2.6 FUTURE WORK . . . 29

2.7 CONCLUSIONS . . . 30

2.8 ACKNOWLEDGMENTS . . . 30

3 I/O survey manuscript 32

3.1 I. INTRODUCTION . . . 37

3.2 II. BACKGROUND AND GENERAL DISCUSSION . . . 38

TABLE OF CONTENTS (Continued)

Page

3.3 III. SURVEY OF EXISTING METHODS AND TECHNIQUES 44

3.3.1 III.A Simple systems and basic DMA operations 44

3.3.2 III.B Generic workstations and servers 49

3.3.3 III.C Datacenters and HPC cluster systems 51

3.3.4 III.D System interconnects and networks 57

3.4 IV. MEASUREMENTS AND QUANTIFICATIONS 62

3.4.1 IV.A Latency 63

3.4.2 IV.B Throughput and Bandwidth Efficiency 65

3.4.3 IV.C Other Descriptor DMA Considerations 67

3.5 V. PROPOSED OPTIONS 68

3.6 VI. CONCLUSIONS AND FUTURE WORK 69

4 Hot-Potato manuscript 72

4.1 INTRODUCTION 73

4.2 BACKGROUND 75

4.3 RELATED WORK 79

4.4 PROPOSED METHOD 81

4.4.1 I/O Transmit operation 82

4.4.2 I/O Receive operation 86

4.4.3 I/O device and core mismatch issues 87

4.5 MEASUREMENTS AND ANALYSIS 88

TABLE OF CONTENTS (Continued)

Page

4.5.1 Micro-benchmark comparison . . . 90

4.5.2 Application benchmark comparison . . . 93

4.6 CONCLUSION AND FUTURE WORK . . . 96

4.7 ACKNOWLEDGMENTS . . . 96

5 D2D manuscript 99

5.1 Introduction . . . 100

5.2 Related Work . . . 102

5.3 D2D Transfer Protocol . . . 103

5.3.1 Configuration of D2D Stream Control Registers and Tx/Rx Queues 104

5.3.2 D2D Tx FSM . . . 108

5.3.3 D2D Rx FSM and UOE . . . 110

5.3.4 D2D Stream: NIC to SSD . . . 113

5.4 Proof-of-Concept Evaluation . . . 113

5.4.1 NetFPGA Implementation . . . 114

5.4.2 Evaluation of Methodology . . . 115

5.4.3 Performance Evaluation and Results . . . 117

5.5 Implementation and performance issues . . . 123

5.6 Conclusion and Future Work . . . 124

6 General Conclusion 125

TABLE OF CONTENTS (Continued)

Page

Bibliography 126

LIST OF FIGURES

Figure Page

2.1 Relative performance increases over time 9

2.2 Current DMA method for IO adapters 11

2.3 Typical Ethernet Transmit 13

2.4 iDMA internal blocks 15

2.5 iDMA transmit flow 18

2.6 iDMA receive flow 19

2.7 Critical path latency between two systems 23

2.8 NIC TX Latency breakdown 24

2.9 iDMA HW transmit breakdown 25

2.10 Raw PCIe link bandwidth utilization 26

3.1 I/O Performance increase comparison (adapted from[2]) 38

3.2 High-performance I/O system block diagram 40

3.3 Typical Ethernet Transmit 41

3.4 Typical Ethernet Receive 43

3.5 Cell processor 46

LIST OF FIGURES (Continued)

Figure Page

3.6 Speed Statistics for DMA 48

3.7 Basic TOE comparison to software-based implementation 54

3.8 I2O fabric diagram 55

3.9 NIC system memory apertures 56

3.10 Discrete vs Integrated NIC performance 57

3.11 SCC message data types 59

3.12 Infiniband communication stack 60

3.13 Communication balance ratio with Infiniband cluster 61

3.14 Critical path latency between two systems 64

3.15 NIC TX latency breakdown 65

3.16 Raw PCIe bandwidth not used for payload transfer 67

3.17 An option for MCP integrated I/O acceleration 69

4.1 Typical Ethernet transmit flow 76

4.2 High-performance I/O system block diagram 77

4.3 Typical Ethernet receive flow 79

LIST OF FIGURES (Continued)

Figure Page

4.4 Structure of the proposed method 82

4.5 Hot-potato TX FIFO 85

4.6 HW & SW co-utilization of interfaces 88

4.7 Hot-Potato ping loopback trace 89

4.8 Intel 82575EB 1GbE ping transmit trace 90

4.9 Throughput comparison 94

4.10 Transmit latency comparison 95

5.1 Conventional I/O Streaming 101

5.2 D2D-enabled SSD. 105

5.3 D2D-enabled NIC. 106

5.4 D2D Tx FSM. 109

5.5 D2D Rx FSM. 111

5.6 NetFPGA design hierarchy. 114

5.7 D2D prototype with emulated SSD. 116

5.8 Example 1500 byte packet latency in the D2D prototype 118

LIST OF FIGURES (Continued)

Figure Page

5.9 D2D *iperf* UDP throughput measurements 120

5.10 D2D CPU power benefit 120

5.11 D2D CPU utilization benefit 122

LIST OF TABLES

Table Page

2.1 Quantified benefits of iDMA 22

3.1 Quantified benefits of an MCP I/O engine 62

3.2 Potential benefits for non-descriptor DMA 70

5.1 D2D Stream Control Register Definitions. 105

5.2 D2D UOE Control Registers. 112

Chapter 1: General Introduction

Current computer systems must handle a variety of Input/Output (I/O) transactions in as optimal a manner as possible. These transactions range from communicating stock prices where low latency is crucial to media streaming that requires high throughput and low CPU overhead.

To understand the motivation of this dissertation, it is helpful to look at the history of I/O transactions. A few decades ago, *Programmed-I/O* (PIO), where the CPU software directly accesses each I/O device, e.g., Network Interface Controller (NIC), storage device, video interface, etc., was the method used to perform I/O transactions. Since I/O devices tend to be slower and have lower throughput than the CPU, this would cause CPU tasks to stall until the I/O transactions are completed. As the system performance improved and I/O devices became more complex, the natural evolution was to have CPU-controlled *Direct Memory Access* (DMA) engines in the I/O devices to access the faster System Memory using *descriptors*, which define where the data to be accessed are located in System Memory. This essentially offloads the data transfer operations from the CPU to I/O devices allowing the CPU core(s) to context-switch to other tasks.

However, CPU-controlled descriptor-based DMA has a number issues: First, as the number of I/O transactions per second increases in a system, the CPU load increases due to the overhead of processing DMA descriptors. Second, the ratio of CPU performance and I/O performance has been increasing [1] due to the increase in CPU core count, which clearly indicates the urgency to improve I/O performance relative to CPU performance. Third, systems with multiple CPUs are becoming more common, where each CPU has integrated I/O functions on-die. This together with the System Memory distributed across multiple CPUs, which is referred to as Non-Uniform Memory Architecture (NUMA), results in complex I/O transaction scenarios. These scenarios will involve even more cores and context switches per I/O transaction resulting in

higher CPU overhead and longer latency. Fourth, overall system power reduction is increasingly becoming an issue. If a CPU core is idle, it enters a sleep state with some combination of frequency reduction, voltage reduction or power gating. With CPU-controlled descriptor-based DMA, either polling or interrupt is used to determine when I/O transactions complete or need service. These interrupts will wake the CPU from power saving states increasing CPU power consumption. Finally, another issue related to power is that these I/O generated interrupts result in context switches among application tasks, referred to as *OS noise*, which reduces computational efficiency.

These aforementioned factors led to our motivation to develop better ways to process I/O transactions without relying on CPU-controlled descriptor-based DMA transactions. This has resulted in several papers that explore improvements as briefly summarized below:

- The first paper proposes the *integrated DMA* (iDMA) [2]. Basically, iDMA involves having a small dedicated controller in the CPU for descriptor processing and management. While the iDMA controller is reminiscent of older, but simpler CPU-controlled descriptor-based DMA engines, the prior implementations were limited to a small number of DMA channels (e.g., less than 8 DMA operations at a time). In contrast, iDMA is implemented using a processor, and thus it can scale to any number of DMA operations and work together with other CPU resources such as the CPU Power Control Unit (PCU) and caches. Therefore, the overall latency of I/O transactions is reduced since the I/O device is not processing the descriptors defined by the CPU. Our measurements show a 16% reduction in latency for small packets by implementing iDMA versus a conventional CPU-centric descriptor-based DMA system. In addition, 17% of the PCIe traffic due to the overhead needed support CPU-based descriptors is removed using iDMA.

  The iDMA controller also reduces OS noise by offloading the descriptor processing from the application-driven core. The processor can be very small since it's main responsibility is to manage the descriptor queues as a proxy for the larger CPU cores. As a result, the overall system cost in terms of silicon die area and power is reduced since DMA engines can be removed from I/O devices.

- The second paper is a survey on I/O transaction techniques [3], which was motivated by the fact that there are many methods available in the literature for accelerating I/O functions. This survey discusses how different systems handle I/O transactions and covers many specialized methods available for optimizing performance. However, it also became clear from this survey that there are no real viable alternatives to CPU-controlled descriptor-based DMA I/O transactions. This led to two research directions for improving I/O transaction system architecture, which are discussed in the third and fourth papers.

- The third paper investigates how CPU access to I/O devices can be simplified by removing DMA descriptors. The use of DMA descriptors result in long latency to perform an I/O transmit operation, which can be explained in terms of an NIC. The CPU software must first formulate the packet and the DMA descriptor and then notify the NIC. After the descriptor is fetched by the NIC with a round-trip PCIe interface read latency, it can be parsed and read by the NIC with another round-trip PCIe interface read latency. Although PIO remains the most direct way of accessing an I/O device, its key weakness is the accessing of the I/O device data across a relatively slow PCIe interface. Therefore, this paper proposes the use of *Write-Combining Buffers* to directly write to I/O devices while maintaining a device DMA engine for receive data. Write-Combining Buffers, which are currently 64 bytes in mainstream CPUs, are non-coherent and non-temporal CPU core buffers that combine multiple 64-bit words of data before performing the memory write operation.

  To test this idea on a current system, an FPGA on the PCIe interface was used to implement I/O device write operations using Write-Combining Buffers. PCIe throughput and latency were measured using a PCIe protocol analyzer to compare with the standard CPU-controlled descriptor-based DMA performance. Our measurements show that transmit latency is reduced by 1.5 μs as a result of using direct memory access with write combining compared to descriptor-based DMA.

- The fourth paper proposes the direct *Device-to-Device* (D2D) communication protocol. The primary example application and the prototype implemented is in video streaming from a Solid State Drive (SSD) to a 10GbE NIC, but D2D

can be applied to any general device-to-device communication. By removing the CPU as a mediator between devices, and having the devices directly transfer data between device memory-mapped I/O space, the CPU can execute other tasks or reduce its power.

To validate this idea, a prototype system consisting of two NetFPGA [4] 10GbE PCIe cards acting as a VoD server was implemented. One card emulated an SSD by receiving video streams from a stimulus system. This card then communicates directly with the second card acting as an NIC. Our implementation demonstrated how a VoD server can benefit from D2D resulting in 2x improvement in latency for small messages and power savings of greater then 32 W for multiple video streams. Our results also show that CPU utilization is significantly reduced, and almost an entire CPU core is allowed to be idle with several concurrent video streams.

There is some CPU overhead required to configure and manage a D2D stream between I/O devices. However, this overhead is negligible and a smaller CPU can be used in the system for I/O intensive applications using D2D. This results in overall system savings in power and cost.

These four manuscripts show that there is a wide area of improvements and benefits to I/O transaction optimizations. This will lead to interesting future research and product development.

Platform IO DMA Transaction Acceleration

Steen K Larsen, Ben Lee

First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems
http://www.mcs.anl.gov/events/workshops/ashes/2011/

Chapter 2: iDMA manuscript

Platform IO DMA Transaction Acceleration

Steen Larsen

Intel Labs

2111 NE 25th Ave

Hillsboro OR 97124

steen.larsen@intel.com

Ben Lee

School of Electrical Engineering and Computer Science

Oregon State University

3013 Kelley Engineering Center

Corvallis, OR 97331

benl@eecs.oregonstate.edu

ABSTRACT

Computer system IO (with accelerator, network, storage, graphics components) has been optimized to use descriptor-based direct memory access (DMA) operations to move data to and from relatively fast addressable system (or main) memory or cache structures. Traditionally, transactions between slower IO sub-systems and system memory have been done using a host bus/bridge adapter (HBA). Each IO interface

has one or more separately instantiated descriptor-based DMA engines optimized for a given IO port. As heterogeneous cores multiply in exascale systems, IO traffic can be expected to be more complex and will require more resources. This paper measures the descriptor overhead and analyzes its impact on latency and bandwidth. Based on quantifications of the latency and bandwidth overhead, we propose to improve IO performance using an integrated platform IO accelerator. This IO engine localizes IO transactions to the processor CPU, rather than offloading to various remote platform interface controllers. By simplifying hardware control of IO in complex systems that rely on a central system memory, we conclude there are quantifiable benefits of integrated platform IO transactions in terms of bandwidth-per-pin and latency, and other areas.

Categories and Subject Descriptors

B.4.1 [**Input/Output and Data Communications**]: Data Communications Devices – *processors, channels and controllers, memory.*

General Terms

Performance, Design, Measurement

Keywords

IO latency, memory, DMA, IO bandwidth

2.1 INTRODUCTION

With IO becoming a peer to processor core (or simply *core*) and memory in terms of bandwidth availability and power requirement, it is important to consider alternatives to existing methods of moving data within a platform. Historically, when a core was simpler and more directly user-focused, it was acceptable to "bit-bang" IO port operations using models such as the Soft-modem , where core instructions directly controlled the modem. However, with complex user interfaces and programs using multiple processes, the benefit of offloading data movement to an IO adapter became more appar-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CACHES '11, June 4th, 2011, Tucson, Arizona, USA Copyright ï¿½ 2011 ACM
978-1-4503-0760-4/11/06 $10.00

ent. In the case where IO speeds are moderate, it makes sense to move data at a pace governed by the external device, which is much slower than core/memory bandwidth. Data transfer is initiated using a descriptor containing the physical address and size of the data to be moved. This descriptor is then posted (i.e., sent) to the IO adapter, which then processes the direct memory access (DMA) read/write operations as fast as the core/memory bandwidth allows.

The descriptor-based DMA approach makes sense when IO bandwidth requirements are much lower than the core/memory bandwidth. However with increasing use of heterogeneous accelerators, such as PCIe graphics engines, their proximity to system memory leads to a memory latency and a throughput bottleneck. This is coupled with multi-core processors and integrated memory controllers driving higher core performance and memory bandwidth. As a result, IO bandwidth increase is slower than the core/memory bandwidth. Figure 1 shows this trend on typical server platforms .

This paper analyzes the performance impact of the current descriptor-based DMA method and proposes an alternative that integrates DMAs closer to the core/memory complex with a standard/universal interface for communication with memory buffers. The proposed *integrated DMA* (iDMA) removes the overhead of setting up and managing descriptors over a chip-to-chip interface, at a cost of a small incremental core package complexity, and improves bandwidth-per-pin, latency, power, Quality-of-Service (QoS), security, and cost (in terms of silicon die area).

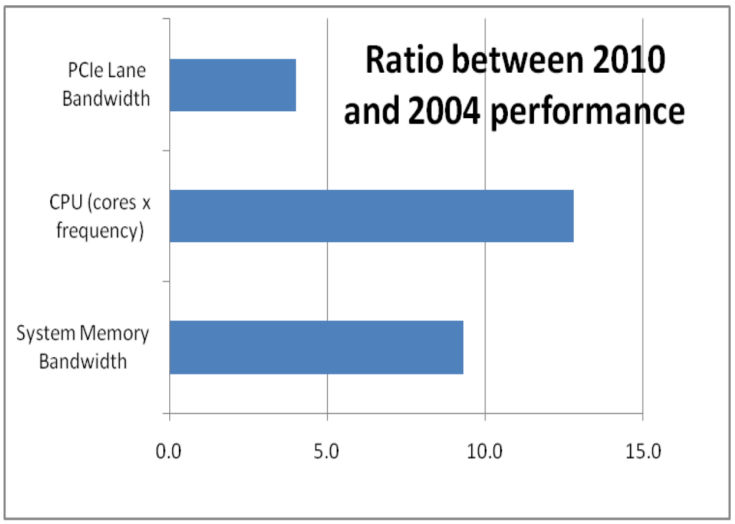

Figure 2.1: Relative performance increases over time

The expected benefits are:

- Reduced latency since there is no descriptor traffic across a chip-to-chip interface such as PCIe or system memory.
- Increased bandwidth-per-pin due to absence of descriptor-based traffic. This also implies more efficient and un-fragmented packet transfers to and from addressable memory.
- Other benefits that are less quantifiable are also explored.

In a sense, iDMA can be seen as re-visiting the Soft-modem implementation where the core plays a more active role in moving data within a platform. We expect the research discussed herein to clearly define and demonstrate the benefits of this approach.

2.2 BACKGROUND

Today's IO-related devices, such as Network Interface Controllers (NIC) (both wired

and wireless), disk drives (solid state and magnetic platter), and USB, have traditionally been orders of magnitude lower in bandwidth than the core-memory complex. For example, a modern 64-bit core running at 3.6 GHz compared to a 1.5 Mbps USB1.1 mouse has 153,600 (64×3600MHz/bit/1.5MHz/bit) times higher bandwidth. Multicores and Simultaneous MultiThreading (SMT) make this ratio even higher. Therefore, it makes sense to offload the cores by allowing the IO adapters some control over how input/output data is pushed/pulled to/from memory. This allows a core to switch to other tasks while the slower IO adapters operate as fast as they are capable.

GPU accelerators, when used as graphics output devices have been single direction data output. Although graphics require high bandwidth, the output-only characteristic of graphic displays allows for loosely ordered data transactions such as write-combining memory buffers. With more generic accelerators tasks such as sorting on Intel's Many Integrated Core (MIC) architecture , data access to system memory is done with descriptors.

Figure 2 shows how IO adapters can bypass core interaction (other than snoop transactions to maintain cache coherency) and directly access system memory. The system memory is defined as the entire coherent memory complex, including multi-core caches as well as system DRAM. The diagram is based on the current 5520 Intel server chipset and is similar for AMD interconnects as well. An IO Hub (IOH) is used to connect directly to the CPU sockets over a coherent memory interface using a QPI protocol that supports the MESI(F) coherency states for 64 Byte cache-line units. High-performance IO adapters connect directly to the IOH, while less critical IO adapters and protocol specific interfaces (SATA, USB, 1GbE, etc) connect using an IO Controller Hub (ICH). The ICH communicates to the IOH using an ESI interface that is similar to PCIe, allowing direct media access to BIOS boot flash. A typical Non-Uniform Memory Architecture (NUMA) maintains coherency either through directory tables or other means outside the scope of this discussion. Arbitrary processor scaling can be realized as Intel QPI links scale out, which is analogous to the AMD Hyper-Transport coherent memory interconnect. This 5520 chipset is the platform on which data for our analysis has been collected. Intel's current MIC accelerator implementation is a PCIe ×16 Gen2 interface providing 8GB/s bandwidth, but Section 2.1 demonstrates how any

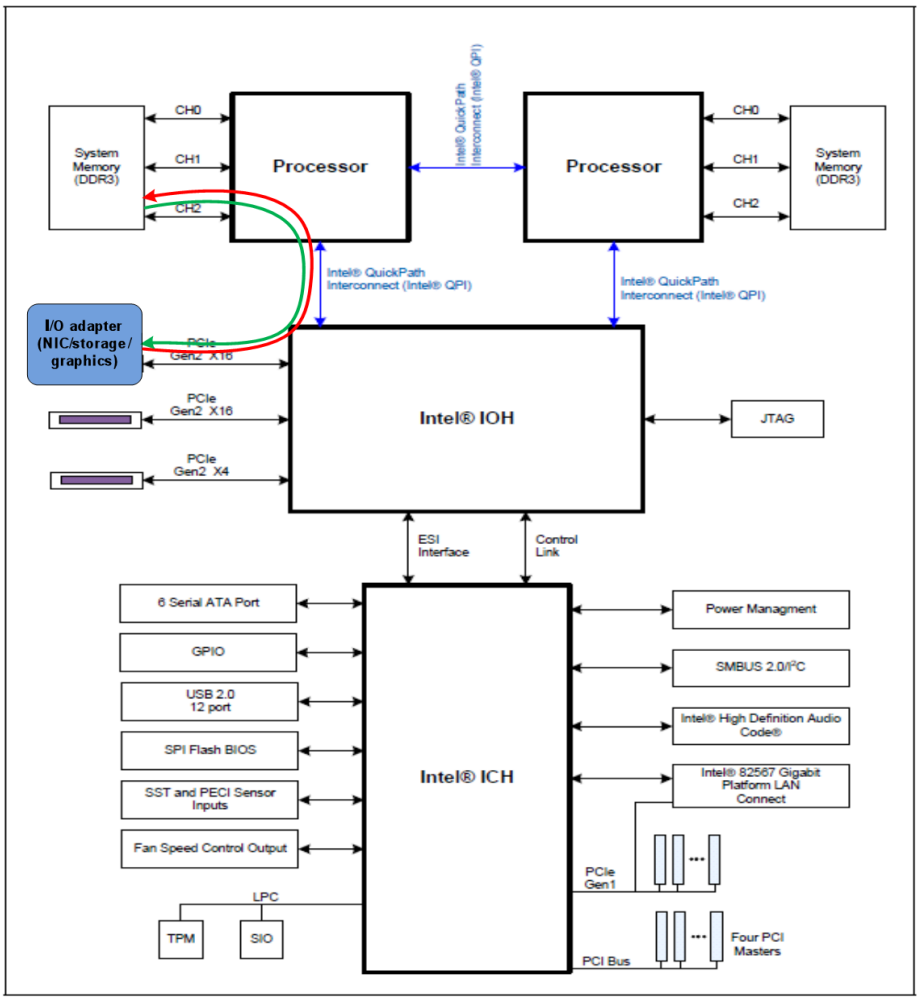

Figure 2.2: Current DMA method for IO adapters

descriptor-based memory coherency would not be a viable consideration since every data snoop would require a descriptor to define the memory location and snoop results.

An optimization to DMA is *Direct Cache Access* (DCA), where an IO device can write to a processor cache by either directly placing data in a cache level or hinting to a pre-fetcher to pull the data from system memory to a higher level. In general, once data is

written into system memory DRAM or processor cache, the data is in coherent memory space and the underlying coherency protocol ensures that reads and writes to the memory address space is handled properly. However, as mentioned by Leon *et al.* , there can be a pollution aspect to a cache hierarchy with DCA. This will trigger cache evictions and higher memory bandwidth utilization as asynchronous IO transactions occur at different times (also described as reuse distance) relative to CPU utilization. Cache pre-fetchers are also good at hiding the memory page misses of large receive buffers.

2.2.1 IO flow using descriptors

Figure 3 illustrates a typical IO transmission for an Ethernet NIC (either wired or wireless). Similar descriptor-based transactions are found on storage and graphics adapters. The following sequence of operations occurs for transmitting an Ethernet message or packet assuming a connection between two systems has already been established (i.e., kernel sockets have been established and opened):

(1) The kernel software constructs the outgoing message or packet in memory. This is required to support the protocol stack, such as TCP/IP, with proper headers, sequence numbers, checksums, etc.

(2) The core sends a write request on the platform interconnect (e.g., PCIe, but also applies to any chip-to-chip interconnect within a platform) to the NIC indicating that there is a pending packet transmission. Since this is a write operation to the memory space reserved by the IO adapter, the write is un-cacheable with implications that other related tasks that are potentially executing out-of-order must be serialized until the un-cacheable write completes. The core then assumes the packet will be transmitted, but will not release the memory buffers until confirmed by the NIC.

(3) The NIC state machine is triggered by this doorbell request, which initiates a DMA request to read the descriptor containing the physical address of the transmit payload. The descriptor is not encapsulated in the doorbell write request because there are two descriptors (frame and payload) in an Ethernet packet definition, and a

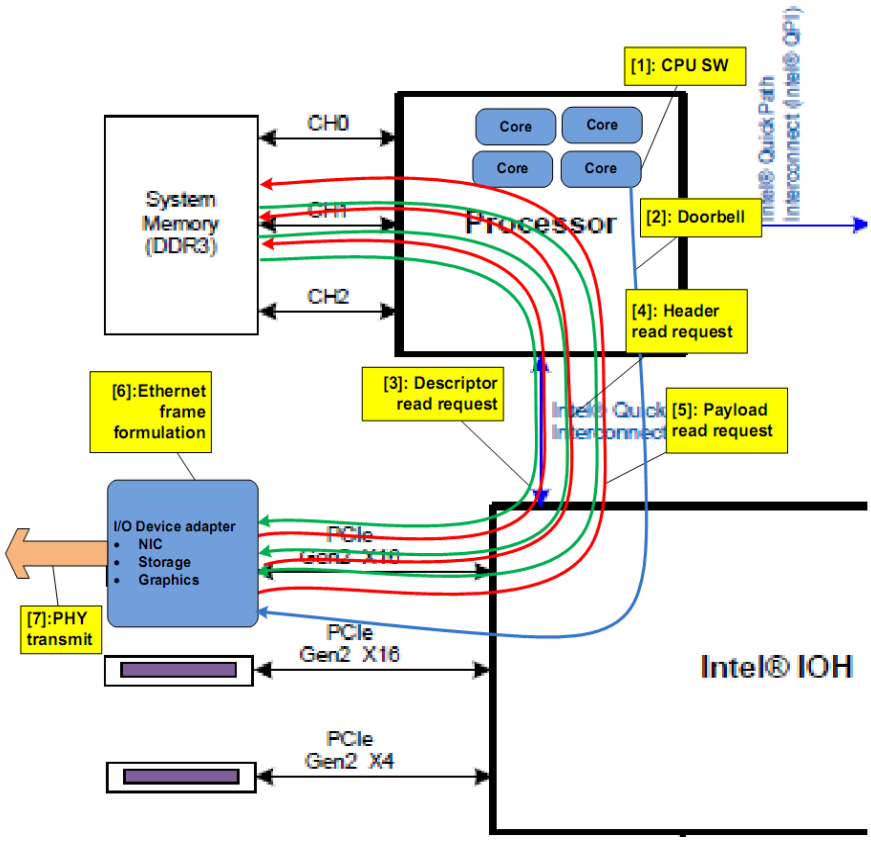

Figure 2.3: Typical Ethernet Transmit

larger network message will require more descriptors (maximum payload for Ethernet is 1460 bytes).

(4) After the memory read request for the descriptor(s) returns with the physical addresses of the header and payload, the NIC initiates a request for the header information (i.e., IP addresses and the sequence number) of the packet.

In the case of an Ethernet descriptor, its size is 16 bytes and it is possible that multiple descriptors can be read from a single doorbell, (i.e., if the cache line size is 64B, 4 descriptors would be read at once). This ameliorates the overall transmit latency and PCIe and memory bandwidth utilization. This approach does not address the funda-

mental problem where a server may have thousands of connections over multiple descriptor queues that do not allow for descriptor bundling. Transmit descriptor aggregation is beneficial if large multi-framed messages are being processed. For example, a datacenter (where multiple systems are physically co-located) may simply choose to enable jumbo-frames, which would renew the transmit descriptor serialization describe above.

(5) The descriptor also defines the payload memory location, so with almost no additional latency other than the NIC state machine, a request is made to read the transmit payload.

(6) After the payload data returns from the system memory, the NIC state machine constructs an Ethernet frame sequence with the correct ordering for the bit-stream.

(7) Finally, the bit-stream is passed to a PHY (PHYsical layer), which properly conditions the signaling for transmission over the medium (copper, fiber, or radio).

The typical Ethernet receive flow is the reverse of the transmit flow. After the core prepares a descriptor, the NIC performs a DMA operation to transfer the received packet into the system main memory. Upon posting the transfer, the NIC can then interrupt the processor and update the descriptor.

The DMA sequence described above is typical of general IO adapters and makes sense when the IO is significantly slower than the cores. When the IO bandwidths are lower than the processor-system-memory interconnect, there is little conflict for available memory bandwidth.

2.3 The Proposed Method

The proposed method consists of an integrated DMA (iDMA) engine in the multi-core processor silicon package shown in Figure 5. The iDMA would be implemented as a simple micro-controller or enhanced state machine that manage the data flow between an arbitrary IO device and system memory. It would basically act as a hetero-

geneous or SoC core to the larger application generic cores. The red arrow shows the iDMA pulling receive traffic from an IO adapter into system memory. The green arrow shows the iDMA reading memory and pushing transmit traffic to the IO adapter. Since IO messages and cache-line memory accesses occur at different times and sizes, basic queue buffers are needed to support arbitration and fragmentation of off-chip interfaces.

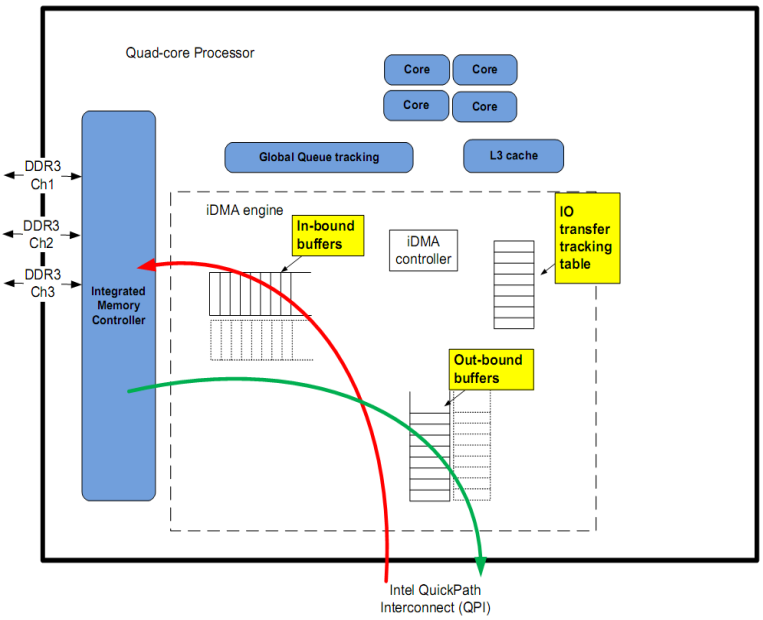

Figure 2.4: iDMA internal blocks

A driving factor for the iDMA engine proposal is the simple observation that inter-chip communication is complex. The life of an IO transmit activity will make about 20 inter-chip hops based on the typical descriptor-based DMA transfer in figure 3. Not only does this impact latency, but there is frame fragmentation to consider. The memory and QPI transfer sizes are cache-line sized (64 bytes) while the PCIe transactions are usually a maximum of 256 bytes and often bear no relevance to the higher level IO transaction for networking, storage, etc. Additionally off-chip transceivers consume significant energy when active to drive signals at high speed and properly frame the inter-chip communication. PCIe is particularly protocol heavy with a Transac-

tion Layer Protocol (TLP) encapsulated in a Data Layer Protocol (DLP) within a Link Layer Protocol (LLP).

IO offloading engines such as TCP Offload Engine (TOE), Infiniband and iWARP take the approach of supporting an IO connection context at the edge of the platform at the interface controller. This allows low latency in the task of frame formatting of header information that would normally be supported with a generic core. To maintain this connection context, side-RAM and controllers are needed on the IO adapter increasing cost and power consumption. Rather than repeating these customized offload engines for each IO type and instance in a system, we suggest to place a generic IO controller as close as possible to the cores processing the IO traffic

To support multiple types of IO (storage, networking, graphics, etc) a state machine may need to be replaced with a small micro-controller. Although the main function of the iDMA engine is to move data between memory and the IO interface, supporting multiple priorities and QoS is needed to prevent a large storage transaction from blocking a latency sensitive network operation. An Intel Atom core or ARM core would be appropriate to manage and track the iDMA engine queues over the various types of IO interfaces in the platform.

The size of these queue buffers depends on several factors, but is based on the largest IO size required and the "drain-rate" supported by the memory interface and the IO interface. Since the possibility of IO contention within the iDMA is highly platform dependent and increases as the number of IO devices increases, we do not make any specific claims on the buffers sizes at this time. NUMA architectures tie memory closely to related cores, so an optimal environment would have an iDMA per multi-core and memory pair. This silicon expense could be mitigated by having IO-oriented multi-core processors (each with iDMA) and less IO oriented multi-core processors (without iDMA) in the same NUMA platform.

For the IO adapter, we assume there are small transmit and receive queue buffers. This could be as small as two frames (e.g., 1,500 bytes for Ethernet and 8KB for SCSI) allowing a frame to transmit out of the platform or into the iDMA buffers while the second frame is filled either by receive PHY or the iDMA transmit. For the receive

flow described in Section 3.2, a small DMA engine is suggested. This would lead to a much smaller IO adapter implementation than current IO adapters (e.g. 25mm×25mm silicon dissipating 5.1W for a dual 10GbE).

There are several reasons that suggest system architecture should move away from current discrete DMA IO adapter engines and become a more centralized data mover engine. First, this leads to simpler IO adapters, which can be viewed as merely a FIFO and PHY since there is no need for a DMA engine with descriptor related logic in each IO adapter within the platform. There are also other advantages including latency, bandwidth-per-pin, QoS, power, security, silicon complexity, and system cost that are quantified in Section 4.

The following two subsections present transmit and receive flows to illustrate the advantages of iDMA.

2.3.1 iDMA Transmit Flow

Figure 6 illustrates the transmit sequence for iDMA, which consists of the following steps:

(1) The kernel software in a core constructs the transmit data structure (i.e., payload and header information) similar to the legacy method described in Section 2.

(2) The core writes a doorbell to the iDMA engine. The format of this doorbell is exactly the same as the legacy doorbell, but it does not traverse the inter-processor or PCIe interface.

(3) The doorbell triggers the iDMA engine to request the descriptor from the system memory. As in the previous step, since the iDMA engine is on the same silicon or package as the multi-core processor, latency is significantly reduced and no chip-to-chip bandwidth is utilized as compared to the descriptor fetch over PCIe.

(4) Based on the descriptor content, a read request is made by the iDMA for the header information. Note that the descriptor fetch and contents do not traversing the

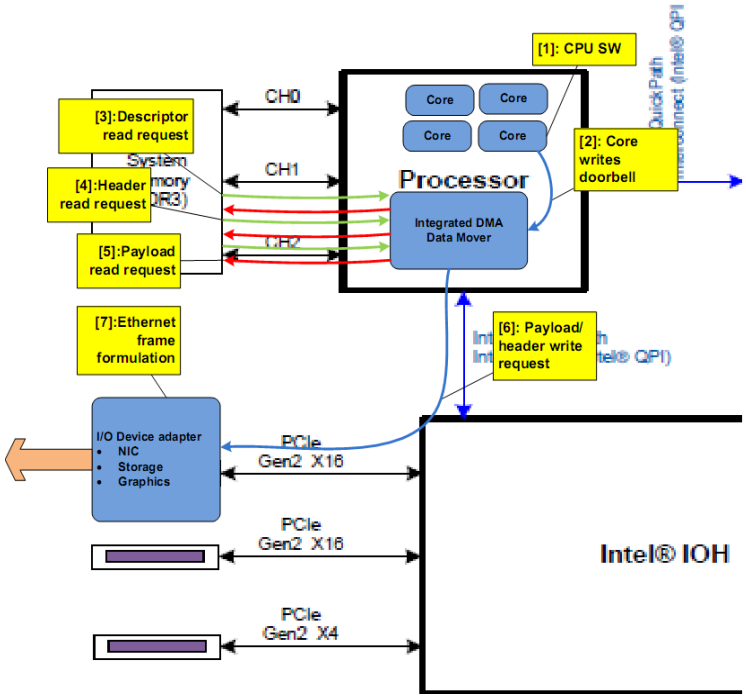

Figure 2.5: iDMA transmit flow

PCIe interface impacting latency and bandwidth-per-pin quantified in Section 4.

(5) A read request is made for the payload data with almost no additional latency.

(6) The iDMA sends a write request to the IO adapter. This is the only transmit payload related task on the PCIe interface, but relies on the underlying PCIe Data Layer Protocol (DLP) and Link Layer Protocol (LLP) for flow control credit and link integrity.

(7) The simplified IO adapter constructs the transmit packet and the PHY performs DSP signal conditioning for optical, copper or wireless media.

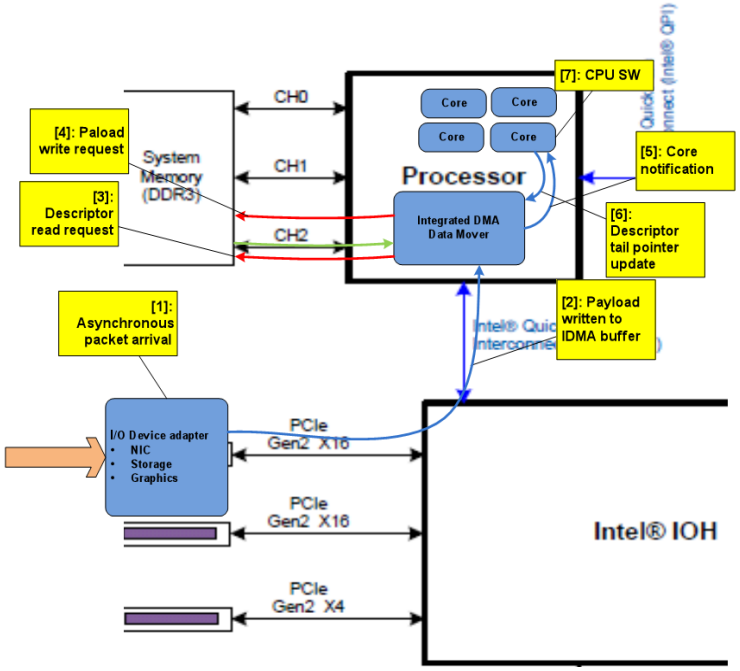

Figure 2.6: iDMA receive flow

2.3.2 iDMA Receive Flow

Figure 6 shows the proposed iDMA receive sequence consisting of the following steps:

(1) The PHY asynchronously receives the packet and some basic link level processing is done to verify that the packet is for the intended platform, such as MAC address filtering. The receive packet will possibly target a particular core, such as current RSS mapping of multiple connections to multiple cores.

(2) The iDMA engine could poll the IO adapter for receive traffic, but since memory bandwidth usually greatly exceeds a particular IO interface bandwidth, having a small buffer in the iDMA engine for receive traffic is suggested. Another argument is that the IOH currently has a write cache allowing buffering of in-bound IO related traffic as it enters memory coherency. The clear benefit is that as soon as the IO adapter receives a frame, it can be pushed into the iDMA receive buffer.

This task does need a simple DMA engine in the IO adapter to cause a PCIe write transaction, but does not need descriptors if the receive iDMA memory buffer is pinned to a particular location in physical memory. This physical memory address indicating the address to the available IO adapters can be initialized on kernel software boot. The iDMA receive control needs to properly manage and increment the receive queue as new receive messages arrive. Since the IOH write cache depth is 128 cache lines , an initial size of the iDMA receive buffer should be $128 \times 64 = 8$KB. This is also referred to as the *push-push* (or hot-potato) model.

(3) This triggers the iDMA engine to fetch the descriptor as in the legacy case. The descriptor defines how the iDMA should process the received packet and can be prefetched. Multiple receive descriptors can be fetched to optimize memory bandwidth utilization since 4 NIC descriptors can fit in a 64-Byte cache line. Again, it should be noted the descriptor fetch does not cross the PCIe interface, but only the memory interface.

(4) The iDMA engine processes the headers and writes to the system memory. This step does not preclude writing directly to a processor's cache. The simplest direct cache writing would be in the L3 cache that is shared by all cores in a multi-core processor on the socket at the cost of adding another write port to the physical cache structure. If the receive traffic can be targeted to a specific core, the iDMA could write directly into the L1 or L2 of the target core.

(5) The iDMA engine notifies a core based on the traffic classification and target core information found in the receive packet. This mapping is done effectively by a hash function, such as Toeplitz found in receive side scaling (RSS) . If none is specified, the iDMA engine can be configured to treat the packet as low priority and notify for example core0 or some preconfigured core.

It can be seen that descriptors still exist in the receive flow on the memory interface. There are three reasons why the use of descriptors on the memory interface should be preserved.

- There would be a significant software change requiring the device driver to han-

dle the control of the iDMA data movement. This has been explored with Intel's IOAT DMA engine , which resides on a PCIe interface and provides benefits only on larger (> 4KB) IO sizes.

- The IO adapter remains a simple FIFO + PHY structure. Such a simple topology saves silicon area and cost over the entire platform.
- Without descriptors, an application would need to manage the data movement directly at the cost of core cycles. It is unclear the cycle overhead would warrant this memory descriptor tradeoff.

The iDMA would need sufficient buffering to support the platform IO demands, but future work could show that buffer sharing for platform IO between networking, storage, and graphics could be an efficient use of CPU socket die area.

2.4 Experiments and Analysis

In order to test our hypothesis, a descriptor-DMA based network IO adapter is placed in an IOH slot of an existing platform based on the hardware configuration shown in Figure 2. This was to determine the potential impact of iDMA on an existing platform without actually building an entire system. By observing the PCIe transactions using a PCIe protocol analyzer, we could measure the impact descriptor-based IO adapter DMA engines had in terms of several factors. Two primary factors are discussed below and summarized in Table 1.

2.4.1 Latency

Latency is a critical aspect in network communication that is easily masked by the impact of distance. However, when inter-platform flight-time of messages is small, as in a datacenter of co-located systems, the impact of latency within a system is much more important. An example is seen with automated stock market transactions (arbitrage

Table 2.1: Quantified benefits of iDMA

Factor	Measurement unit	Descriptor DMA	iDMA	Estimated Improvement	Comment/justification	Section
Latency	microseconds to transmit a TCP/IP message between two systems	8.8	7.38	16%	Descriptors are no longer latency critical	4.1
Bandwidth-per-pin	Gbps per serial lane link	2.1	2.5	17%	Descriptors no longer consume chip-to-chip bandwidth	4.2

and speculation) as demonstrated by Xasax claiming 30µs latency to the NASDAQ trading floor. A second example is high performance computing (HPC) nodes where LinPack benchmark (used to define the Top500 supercomputers) share partial calculations of linear algebra matrix results among nodes. A third example is a Google search request, which may impact tens of systems between the page request and response as the results, advertisements, and traffic details are calculated on multiple networked or virtualized systems, in effect multiplying any system related latency. Certainly, intercontinental distances and multiple switches/routers will add millisecond-scale latencies, but for the scope below, let us consider a typical datacenter.

Figure 7 shows the typical 10Gb Ethernet (GbE) latency between a sender (TX) and a receiver (RX) in a datacenter type of environment where the fiber length is on the order of 3 meters. These results are based on PCIe traces of current 10GbE Intel 82598 NICs (code named Oplin) on PCIe ×8 Gen1 interfaces. The latency benchmark NetPIPE is used to correlate application latencies to latencies measured on the PCIe interface for 64-byte messages. The 64-byte size was used since it is small enough to demonstrate the critical path latencies, but also large enough to demonstrate a minimal message size that may be cache-line aligned.

End-to-end latency consists of both hardware and software delays, and depends on many aspects not directly addressed in this paper, such as core frequency, memory frequency and bandwidth, and cache structure. The software component is not related to

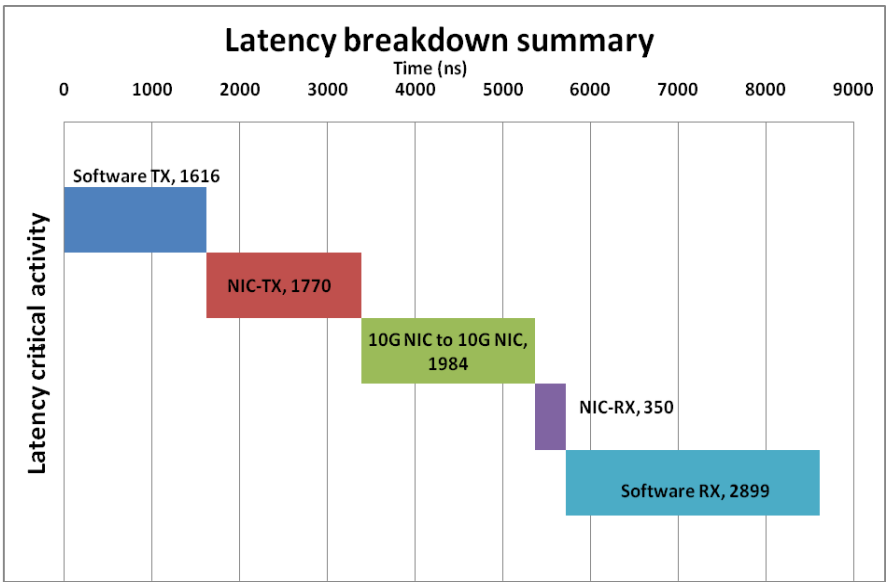

Figure 2.7: Critical path latency between two systems

iDMA since it relates to how the core processor core handles IO traffic only in terms of residency in main memory. Software latency is described in more detail by Larsen *et al.* . The hardware latency can be split into three portions. First, the TX NIC performs DMA reads (NIC-TX) to pull the data from the system memory to the TX NIC buffer. This is followed by the flight latency of the wire/fiber and the TX/RX NIC state machines (NIC to NIC). Finally, the RX NIC performs DMA writes (NIC-RX) to push the data from the RX NIC buffer into the system memory and interrupts a processor for software processing. The total latency can be expressed by the following equation:

$$Total\ critical\ path\ latency\ = Tx_{SW} + Tx_{NIC} + fiber + Rx_{NIC} + Rx_{SW}$$

The latency for the NIC-TX portion can be further broken down as shown in Figure 8 using PCIe traces. A passive PCIe interposer was placed between the platform PCIe slot and the Intel 82598 NIC. PCIe traces were taken from an idle platform and network environment. These latencies are averaged over multiple samples and show some variance, but is under 3% min to max. The variance is generated by a variety of factors such as software timers and PCIe transaction management. Based on a current 5500 In-

tel processor platform with 1066MB/s Double Data Rate (DDR3) memory, the doorbell write takes 230 ns, the NIC descriptor fetch takes 759 ns, and the 64B payload DMA read takes 781 ns. The CPU frequency is not relevant since the DMA transactions are mastered by the PCIe NIC adapter.

If DMA transfers were "pushed" by the iDMA engine close to the core and memory instead of being "pulled" by the slower NIC device, the transmit latency would be significantly reduced. This is summarized in Figure 9 with each bar described below:

(1) A core writes to the iDMA to trigger the transmit flow as in a legacy PCIe doorbell request. Since this is on-die propagation, this latency is well within 20 ns.

(2) The descriptor fetch would not be needed to determine physical addresses of the header and payload. Instead, the iDMA running at (or close to) core speeds in the 2-3 GHz range essentially reduces the descriptor fetch to a memory fetch of about 50 ns for current Intel 5500 systems.

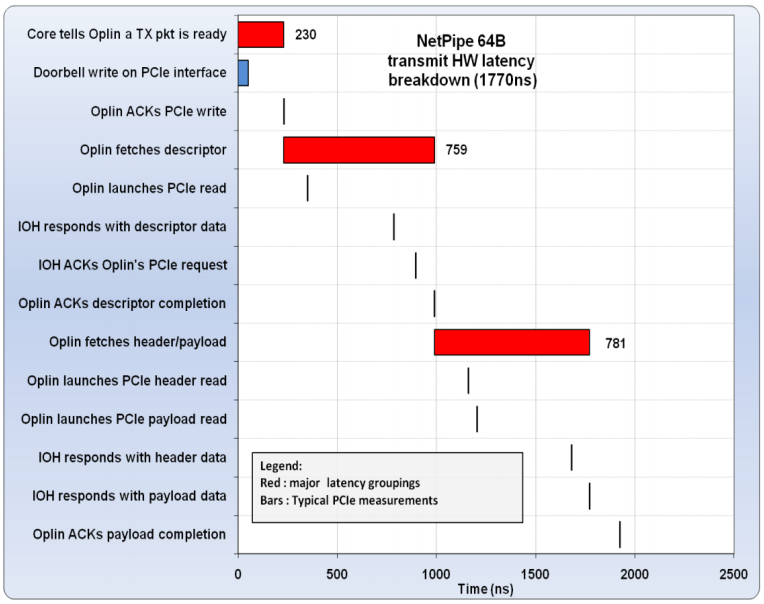

Figure 2.8: NIC TX Latency breakdown

(3) Similarly, this replaces the header/payload information fetch by the NIC adapter

with a memory read request of about 50 ns.

(4) The iDMA transfers the header/payload information to the IO adapter, which is expected to be similar to the legacy doorbell latency of 230 ns.

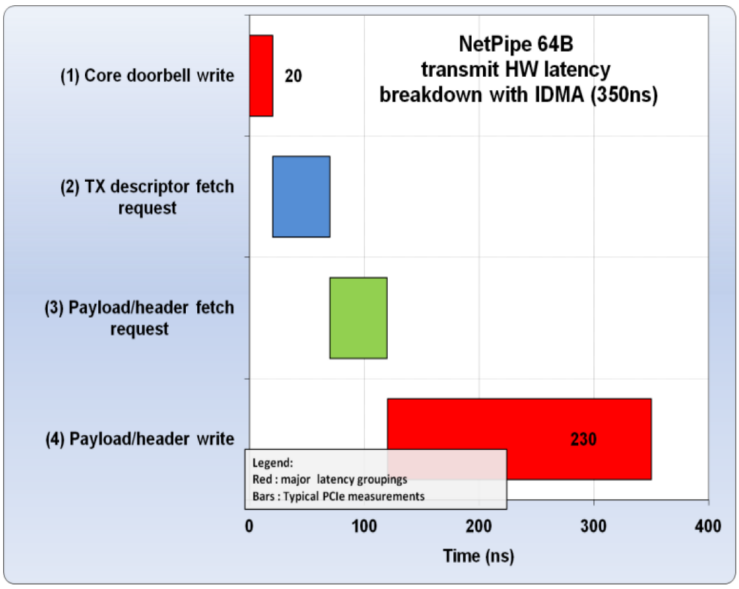

Figure 2.9: iDMA HW transmit breakdown

These improvements reduce the ideal NIC-TX transmit latency from 1,770 ns to 350 ns or 80%. While 80% latency reduction in moving a small transmit message to an IO adapter may be impressive, a more realistic benchmark is the end-to-end latency impact. The total end-to-end delay as seen by an application reduces from 8.8 µs to 7.38 µs, or 16% reduction in latency. 16% improvement by itself may not be impressive, but there are other advantages such as:

- Smaller bandwidth-delay product reduces buffering requirements at higher levels of hardware and software, e.g., 10Gbps × 8.8µs requires 11KB of buffering, while 10Gbps × 7.38 µs requires only 9KB. This bandwidth-delay product is per connection (or flow), so in the case of web servers the impact can be increased several thousand-fold.

- Less power consumed as the PCIe link is not as active and can transition to lower power states more quickly.

2.4.2 Throughput/Bandwidth efficiency

Figure 10 shows a breakdown of transaction utilization in receiving and transmitting data on a PCIe interface for a dual 10GbE NIC. The iperf bandwidth benchmark was used on a dual 10GbE Intel 82599 (codename Niantic) Ethernet adapter on an Intel 5500 server. The primary difference (in this context) to the earlier latency capture with the Intel 82598 is that PCIe bandwidth has doubled. The internal state-machine driven NIC and core frequency remains the same.

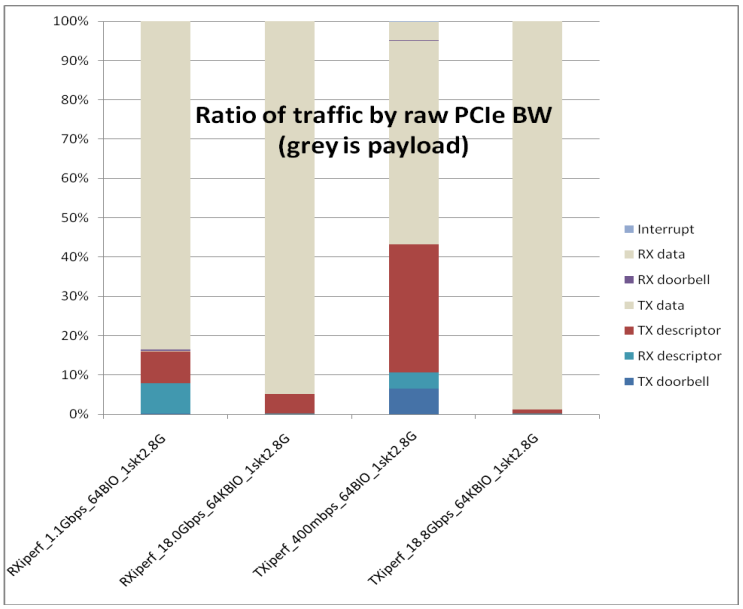

Figure 2.10: Raw PCIe link bandwidth utilization

PCIe captures consisted of more than 300,000 PCIe ×8 Gen2 packets, or 10 ms of real-time trace. This gives a statistically stable capture for analysis. The four bars in Figure 10 show extreme cases of TCP/IP receive traffic (`RXiperf_`) and transmit traffic (`TXiperf_`) for small (`_64BIO_`) and large (`_64KBIO_`) IO sizes. Receive traffic perfor-

mance is important for such applications as backup and routing traffic, while transmit traffic performance is important in serving files and streaming video. Small IO is representative of latency sensitive transactions and large IO is representative of storage types of transactions. TCP/IP is also a bi-directional protocol where an ACK message is required to complete any TCP communication. Descriptors and doorbell transactions consume a significant amount of bandwidth. The ratio of payload related transactions to non-payload transactions show a 50% inefficiency of PCIe transactions for small IO messages. This includes PCIe packet header and CRC data along with PCIe packet fragmentation. The bits-on-the-wire is shown in Figure 10, which shows that transmitting small payload sizes utilizes 43% of the PCIe bandwidth for descriptor and doorbell traffic that would otherwise be removed with iDMA.

In the case of IO receive for small payload sizes, the benefit reduces to 16% since 16B descriptors can be pre-fetched in a 64B cache-line read request. For large message IO sizes, the available PCIe bandwidth is efficiently utilized with less than 5% of the bandwidth used for descriptors and doorbells.

In general, network traffic has bi-modal packet size with concentrations of small Ethernet frames and large Ethernet frames. Thus, we can average overall efficiency of TCP/IP traffic and see that without descriptors and associated interrupts and doorbell transactions on the PCIe interface, available bandwidth on the PCIe interface would increase by 17%.

There are other benefits beyond physical bandwidth-per-pin on the PCIe interface. The current 4-core Intel 5500 system arbitrates memory bandwidth between IO requests and multi-core processor requests. As a multi-core processor starts to contain more cores and SMT threads, memory bandwidth becomes more valuable. The memory subsystem is not capable of distinguishing priority of requests between a generic IO adapter and a core. For example, the current arbitration cannot distinguish between a lower latency IO priority (e.g. storage request) and a higher latency IO priority (e.g. network IO request). Instead, the bandwidth is allocated in a round-robin fashion that can be modified during system initialization. By having arbitration logic for memory bandwidth between cores and IO adapters in the processor socket, the level of conflicting priorities between cores and IO for system memory access is reduced with the iDMA.

2.4.3 Other benefits

Other benefits that were not quantified with measurements are described below:

- Reduced system software complexity in small accelerator cores, where the descriptor rings or queues need to be maintained on a per core basis.
- Arbitration between accelerator cores for DMA engine transfer resources becomes important as core count increases. The Intel MIC, with 32 cores can have up to 32 generators of DMA transaction requests. The Nvidia GTX 280, with 30 shared multiprocessors each with up to 32 separate threads of execution helps indicate how the arbitration of DMA transfers for proper service becomes important. Additionally, PCIe devices usually support only a certain number of outstanding transactions, so the overhead of descriptor processing can become significant.
- Increased system bandwidth scalability since IO interfaces are more efficient at moving data. Fewer PCIe lanes are required and smaller form-factors required for future designs.
- Power management control has better visibility of power state transitions to allow for better power reduction heuristics. Total system power is reduced by removing descriptor transactions on the IO adapter interface, not only from the point-of-view of a bits-on-the-wire, but also the associated framing overhead (e.g., TLP/DLP/LLP framing on PCIe packets).
- Improved QoS since various traffic flows can be moderated and controlled more carefully by the multi-core processor.
- Reduced cost and silicon area since the complex segmentation and reassembly tasks of IO operations can be centralized into a single hardware implementation.
- Increased security since arbitrary devices can be prevented from directly accessing physical memory.

2.5 RELATED WORK

Sun Niagara2 NIC integration showed a method of how descriptors can be removed from a PCIe interface, but this requires a complete integration of a fixed type of IO that is an inflexible approach to generic descriptor based IO. Schlansker *et al.* and others proposed bringing IO adapters into a memory coherent domain. This adds complexity to support the often slower IO adapters response to coherency protocol. Intel has implemented QuickData Technology focused on storage interfaces, but still needs descriptors on a PCIe interface to define the offloaded DMA transactions.

2.6 FUTURE WORK

We plan to explore further the memory and processor overhead of minimizing descriptor processing. Proper quantification of the core cycles and memory bandwidth is expected to further strengthen the iDMA proposal. The current results described in Section 4 are based primarily on measurements of the PCIe interface. We hope to confirm these findings using other descriptor-based interfaces, such as GPU accelerators, Light Peak , and SuperSpeed USB.

Reducing or removing descriptor-based traffic can lead to higher-level improvements in the area of sockets structure handling and driver implementation. Cases for improved QoS can be proposed with an iDMA engine that is peer to the application cores.

Further exploration of instantiation of the iDMA is needed. A state machine driven approach, such as Intel's IOAT , can be compared in terms of die area and power to a larger, but more adaptable small heterogeneous core.

IO offloading schemes such as TCP Offload Engines (TOE), iWARP and Infiniband reduce latency to some extent by formulating headers on the adapter, but still utilize descriptors to move payload larger than a cache-line size to and from system memory. We plan to characterize the benefit in typical offload mechanisms as well.

2.7 CONCLUSIONS

With accelerator data movement complexities, especially in a heterogeneous core environment, there is a need to address unscalable inefficiencies. As system core cycles become less valuable due to MCP and SMT, and IO requirements increase, we proposed an IO accelerator to consider in future platform architectures. The current IO approach of descriptor-based DMA engines to entirely off-load from the CPU can be considered "pull-push" for TX and RX transactions. In this paper, we explored the impact of the descriptor related overhead and proposed a "push-pull" method with a dedicated iDMA engine in the CPU. Adapting write-combining memory type to generic IO and offloading IO data movement to a generic platform requirement (networking, storage, and graphics) suggests quantifiable latency and bandwidth-per-pin advantages and potential benefits in power, QoS, silicon area, and security for future datacenter server architecture.

2.8 ACKNOWLEDGMENTS

Discussions with Steve McGowan on IOAT and USB descriptors and Justin Teller on MIC descriptor details.

Platform IO DMA Transaction AccelerationSurvey on system I/O Hardware Transactions and Impact on Latency, Throughput, and other Factors

Steen K Larsen, Ben Lee

Advances in Computers volume 92
https://www.elsevier.com/books/advances-in-computers/hurson/978-0-12-420232-0

Chapter 3: I/O survey manuscript

Survey on system I/O Hardware Transactions and Impact on Latency, Throughput, and other Factors

Steen Larsen and Ben Lee

(School of Electrical and Engineering Computer Science, Oregon State University)

Prepared: June 6 2012

Steen Larsen (corresponding author)

School of Electrical Engineering and Computer Science

124 NE Shannon St

Hillsboro OR 97124

steen.knud.larsen@gmail.com

Ben Lee

School of Electrical Engineering and Computer Science

Oregon State University

3013 Kelley Engineering Center

Corvallis, OR 97331

benl@eecs.oregonstate.edu

I. INTRODUCTION 5

II. BACKGROUND AND GENERAL DISCUSSION 6

III. SURVEY OF EXISTING METHODS AND TECHNIQUES 10

III.A Simple systems and basic DMA operations 10

III.A.1 Embedded systems 10

III.A.2 CPU DMA and DSP systems 10

III.A.3 General CPU controlled engines 11

III.A.4 PCIe Optimization 11

III.A.6 Other DMA Related proposals 12

III.B Generic workstations and servers 12

III.B.1 Operating System Virtualization Protection 13

III.B.2 Interrupt moderation 13

III.C Datacenters and HPC cluster systems 13

III.C.1 Receive Side Scaling (RSS) 14

III.C.2 Large Receive Offloading (LRO) 14

III.C.3 Large Segment Offload (LSO) 14

III.C.4 Descriptor packing 14

III.C.5 TCP/IP Offload Engine (TOE) and other offload engines 14

III.C.6 Coherent Network Interfaces (CNI) 16

III.C.7 Network Interface Integration 17

III.D System interconnects and networks 18

III.C.1 CPU Socket Interconnect 18

III.D.1 Messaging between cores 18

III.C.3 InfiniBand 19

III.C.4 SciCortex Fabric 21

IV. MEASUREMENTS AND QUANTIFICATIONS 21

IV.A Latency 22

IV.B Throughput and Bandwidth Efficiency 24

IV.C Other Descriptor DMA Considerations 25

V. PROPOSED OPTIONS 26

VI. CONCLUSIONS AND FUTURE WORK 27

REFERENCES 27

Abstract

Computer system I/O has evolved with processor and memory technologies in terms of reducing latency, increasing bandwidth and other factors. As requirements increase for I/O (such as networking, storage, and video), hardware DMA transactions have become more important in high performance systems to transfer data between IO adapters and system memory buffers. DMA transactions are done with hardware engines below the software protocol layers in all systems other than rudimentary controllers. By offloading hardware DMA transfers to the IO adapters, CPUs can switch to other tasks. Each I/O interface has one or more separately instantiated descriptor-based DMA engines optimized for a given I/O port. This offloading comes at a cost that is not found on smaller embedded systems where I/O performance may be as important as preserving controller core cycles. This paper reviews the current state of high-performance I/O architecture and suggests opportunities and options for improvement. As I/O complexity increases with GPUs and multi-GB/s ports there are contention issues with this distributed DMA approach for access to addressable memory with system CPUs. Possible integration options allows quantifiable benefits in terms of I/O latency, I/O bandwidth-per-pin, I/O right-sizing, power efficiency, QoS, cost, and security.

Keywords:input/output, processors, controllers, memory, DMA, latency, throughput, power

3.1 I. INTRODUCTION

I/O is becoming a peer to processor core (or simply core) and memory in terms of bandwidth and power requirements. Thus, it is important to consider alternatives to existing methods of moving data within a computer platform. Historically and with current embedded controllers when a core was simpler and more directly I/O focused, it was acceptable to "bit-bang" I/O port operations using port I/O or memory mapped I/O models [1]. However, with complex user interfaces and programs using multiple processes, the benefit of offloading data movement to an I/O adapter became more apparent. Since I/O devices are much slower than core/memory bandwidth, it makes sense to move data at a pace governed by the external device. Data transfer is initiated using a descriptor containing the physical address and size of the data to be moved. This descriptor is then posted (i.e., sent) to the I/O adapter, which then processes the direct memory access (DMA) read/write operations as fast as the core/memory bandwidth allows.

The disaggregated descriptor-based DMA approach makes sense when the I/O bandwidth requirements are much lower than the core/memory bandwidth. However, with the advent of multi-core processors (MCPs) and symmetric multi-threading (SMTs), the I/O device capability can be scaled as the number of cores scale per MCP. Figure 1 shows how MCP scaling and the integration of the memory controller have exceeded I/O bandwidth gains during the period from 2004 to 2010 [2]. I/O bandwidth has only gained by 4× over the same time period that MCPs and memory interfaces gained more than 8×. I/O also needs quality of service to provide low latency for network interfaces and graphics accelerators, and high bandwidth support for storage interfaces.

Sun Niagara2 network interface integration showed how descriptors can be removed from a PCI express (PCIe) interface [3], but integration of a fixed type of I/O is inflexible compared to generic software-based descriptor controlled I/O. Schlansker et al. [4] and others proposed bringing I/O adapters into a memory coherent domain. However, this adds complexity to support the slower I/O adapters response to coherency protocol. Intel has implemented QuickData Technology focused on storage interfaces [5], but it still needs descriptors on a PCIe interface to facilitate the offloaded DMA transactions. Liao et al. proposed an integrated network interface DMA controller [6], but they do

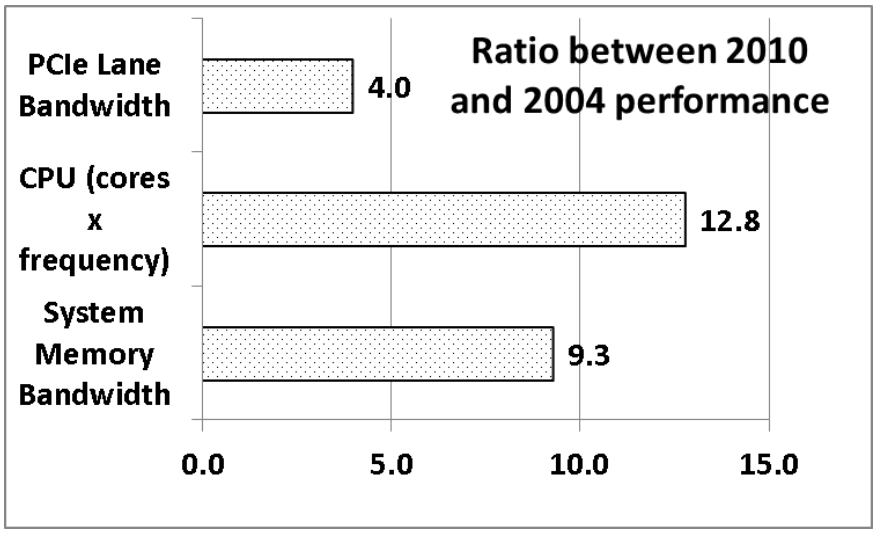

Figure 3.1: I/O Performance increase comparison (adapted from[2])

not consider using the DMA engine on non-networking types of I/O interfaces, such as heterogeneous Graphics Processing Units (GPUs) and storage devices.

This paper provides a general background and discussion in Section II, followed by a survey in section III on how various systems perform I/O based on the prior sections description. Section IV provides detailed measurements and analysis typical of current high-performance I/O devices, and Section V suggests areas for improvement and optimizations.

3.2 II. BACKGROUND AND GENERAL DISCUSSION

Today's I/O devices, such as Network Interface Controllers (NICs), storage drives, and Universal Serial Bus (USB) have traditionally been orders of magnitude lower in bandwidth than the core-memory complex. For example, a modern 64-bit core running at 3.6 GHz compared to a 1.5 Mbps USB1.1 mouse has 153,600 times higher bandwidth. MCPs and SMT make this ratio even higher. Therefore, it makes sense to offload the cores by allowing the I/O adapters some control over how input/output data is pushed/pulled

to/from memory. This allows a core to switch to other tasks while the slower I/O adapters operate as efficiently as they are capable.

A current high-performance system diagram using the Intel 5520 chipset [7] is shown in Figure 2 describing the internal system components. The two arrows between the IO adapter show the path and bandwidths available as the I/O adapter reads transmit data from system memory and writes received data to system memory. This system is used in the measurements and analysis discussed in section IV.

Each processor die consists of 4 cores each with 256KB L2 cache and a shared inclusive 8MB last level cache. The speed of each core (3.2GHz) is not directly applicable to the discussion as we will see the I/O transaction efficiency is governed more directly by the I/O device controllers. A tracking mechanism called a Global Observation Queue (GOQ) controls memory transaction coherency such that snoops by memory requests into on-die cores, or other MCPs are performed correctly before a transaction is completed. The GOQ also helps protect memory contention between I/O devices and the cores within a system.

The I/O Hub (IOH) interfaces between the Intel QPI coherent memory interface and multiple PCIe interfaces. This flexible IOH design allows 1 to 4 MCPs to be configured using 1 to 2 IOHs for a variety of I/O expansion capabilities. Each IOH has BIOS controlled registers to define the PCIe lane configuration allowing sytems to have either multiple lower bandwidth PCIe interfaces or fewer high bandwidth PCIe interfacs such as graphics engines.

High performance I/O adapters connect directly to the IOH while slower I/O interfaces are interfaced with the I/O Controller Hub (ICH). In our discussion this part of the system is not used.

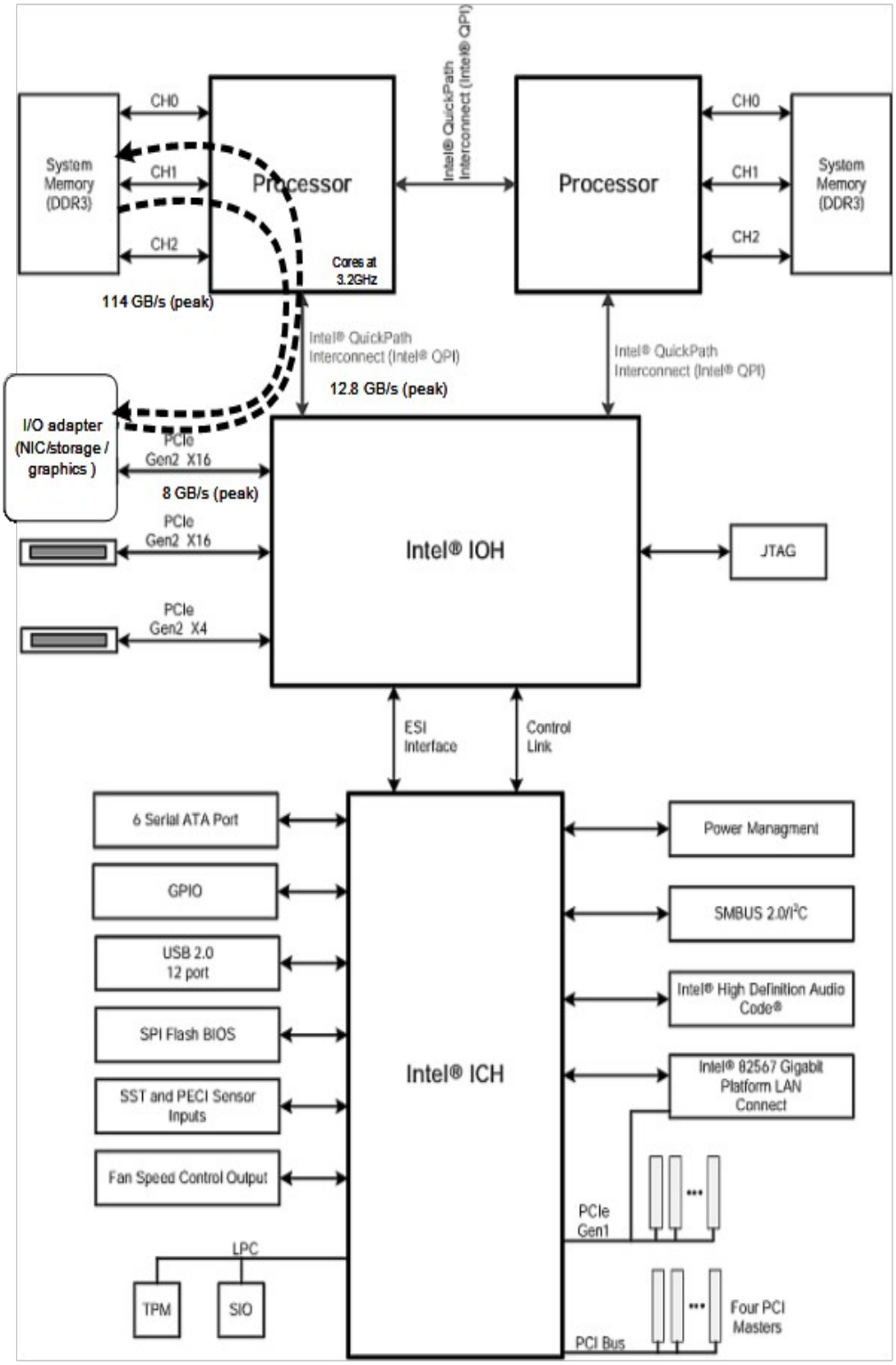

Figure 3.2: High-performance I/O system block diagram

Figure 3 illustrates a typical I/O transmission for an Ethernet NIC (either wired or wireless).

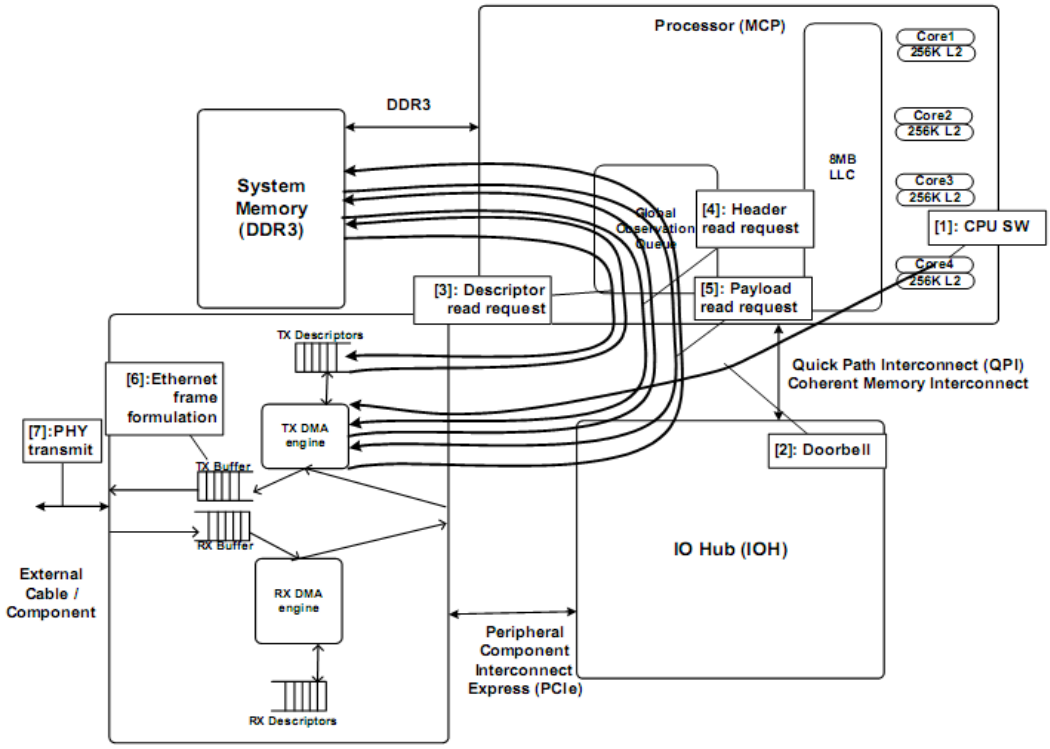

Figure 3.3: Typical Ethernet Transmit

Similar descriptor-based transactions are also found on storage and graphics adapters. The following sequence of operations occurs to transmit an Ethernet packet between two connected systems (i.e., kernel sockets have been established and opened):

(1) The kernel software constructs the outgoing packet in memory. This is required to support the protocol stack, such as TCP/IP, with proper headers, sequence numbers, checksums, etc.

(2) The core sends a doorbell request on the platform interconnect (e.g., PCIe, but this also applies to any chip-to-chip interconnect within a platform) to the NIC indicating

that there is a pending packet transmission. This is a write operation to the memory space reserved by the I/O adapter, which is un-cacheable with implications that other related tasks that are potentially executing out-of-order must be serialized until the un-cacheable write completes. The core then assumes the packet will be transmitted, but will not release the memory buffers until confirmed by the NIC that the packet has been transmitted.

(3) The doorbell request triggers the NIC to initiate a DMA request to read the descriptor containing the physical address of the transmit payload. The descriptor is not included in the doorbell write request because there are two descriptors for header and payload in an Ethernet packet definition, and a larger network message will require more descriptors (e.g., maximum payload for Ethernet is 1460 bytes).

(4) A memory read request for the descriptor(s) returns with the physical addresses of the header and payload. Then, the NIC initiates a request for the header information (e.g., IP addresses and the sequence number) of the packet.

(5) A request is made to read the transmit payload using the memory location of the payload in the descriptor with almost no additional latency other than the NIC state machine.

(6) When the payload data returns from the system memory, the NIC state machine constructs an Ethernet frame sequence with the correct ordering for the bit-stream.

(7) Finally, the bit-stream is passed to a PHYsical layer (PHY) that properly conditions the signaling for transmission over the medium (copper, fiber, or radio).

The typical Ethernet receive flow is the reverse of the transmit flow. After the core prepares a descriptor, the NIC performs a DMA operation to transfer the received packet into the system memory. Upon posting the transfer, the NIC interrupts the processor and updates the descriptor.

Figure 4 illustrates the typical Ethernet receive flow, which is similar to the transmit flow but in reverse: (1) The NIC pre-fetches a descriptor associated with the established connection, and matches an incoming packet with an available receive

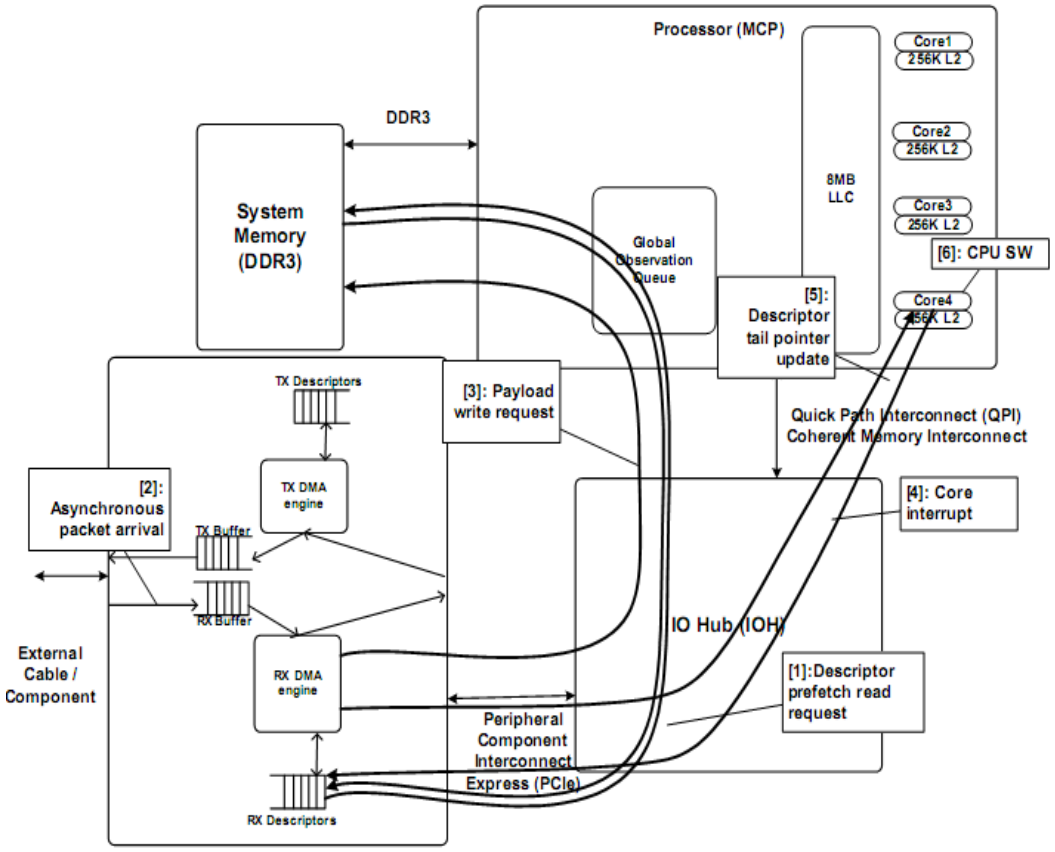

Figure 3.4: Typical Ethernet Receive

descriptor.

(2) The receive packet arrives asynchronously to the NIC adapter.

(3) The NIC performs a DMA write to transfer the packet contents into the memory space pointed by the receive descriptor in the system memory.

(4) After the memory write transaction completes, the NIC interrupts the core indicating a new packet has been received for further processing. The interrupt is usually moderated (also described as coalesced or aggregated) with other incoming packets to

avoid overloading the core with context switches on a per-packet basis.

(5) As part of the interrupt processing routine, a core write instruction is issued to synchronize the NIC adapter descriptor ring with the core descriptor ring

(6) Finally, the kernel software can process the receive packet residing in the system memory.

The DMA sequence described above is typical of I/O adapters including GPUs and storage adapters.

An optimization to I/O DMA is Direct Cache Access (DCA), where an I/O device can write to a processor cache by either directly placing data in a cache or hinting to a pre-fetcher to pull the data from the system memory to a cache. However, this method still requires a descriptor fetch for the I/O device to determine where to place the data in physical memory.

3.3 III. SURVEY OF EXISTING METHODS AND TECHNIQUES

Computer systems utilize a broad range of I/O methods depending on the external usage requirements and the internal system requirements. We segment the survey based on scale, or complexity, of the system since the fundamental method of moving data to and from an external destination to the system is more effective than different types of I/O (network, storage, video, etc).

3.3.1 III.A Simple systems and basic DMA operations

Systems that do not support multiple users and hardware virtualization are considered simple systems.

3.3.1.1 III.A.1 Embedded systems

The simplest method of system I/O and usually not considered fast system is found in embedded controllers where there may be dedicated signals or addressable memory locations for input and/or output data. An example would be a clock radio where outputs are LCD segment signals and inputs are buttons with dedicated signals that may be polled or interrupt a basic software routine. Example controllers used for such functions include device families around the Intel 8051, Atmel AVR, ARM, and Microchip PIC where no operating system is used to virtualize hardware.

This direct access of I/O by a controller can support I/O protocols at low bandwidths such as the WinModem mentioned in Section I by “bit-banging"

3.3.1.2 III.A.2 CPU DMA and DSP systems

An extension of the direct I/O control is to have a DMA engine configured and controlled by the CPU. This is commonly found in digital signal processing (DSP) systems where large amounts of “signal” need to be moved between I/O, signal processing function blocks and memory. The basic algorithm is:

1. CPU software determines via interrupt or other control a need to move N bytes from location X to location Y, often in a global memory mapped address space.
2. The CPU configures the DMA engine with the instructions to move N bytes starting at address X to address Y.
3. The CPU either polls or waits for DMA interrupt for completion.

Normally there is no operating system, again for the purpose of removing any virtualization complexities.

An example is seen with the IBM/TI Cell processor shown in Figure 5 as an architecture with its Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPE) engines (or cores) [8, 9].

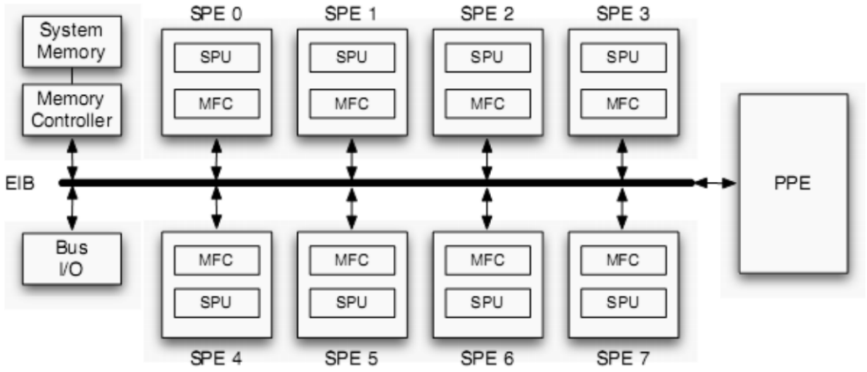

Figure 3.5: Cell processor

Each SPE has 256KB of local on-die memory. If non-local memory access is required, SPE can either request the PPE for a kernel/OS service, or DMA from the system memory for I/O transactions. To reduce the SPE processing overhead Ionkov et al. proposed using co-routines to schedule memory accesses to the local memory [8]. There is no memory coherency structure as found in x86 MCP processors, reducing inter-core communication requirements. The Cell processor architecture affirms that memory movement via core controlled DMAs is effective in graphics processing workloads in current workloads.

Two papers [10, 11] cover typical embedded DMA processing and Katz and Gentile [12] provide a similar description of DMA on a Texas Instruments Blackfin digital signal processing processor. DMA descriptors are used to define physical payload status and memory locations similar to legacy Ethernet I/O processing described in Section II.

3.3.1.3 III.A.3 General CPU controlled engines

An execption to the simple system case is where Intel has implemented a DMA offload

engine called QuickData Technology in a PCIe accessible device as a method to improve I/O performance for storage and RAID [5]. This allows in-flight XOR calculation for efficient RAID and is beneficial for large fixed-size blocks of data. Currently, it is implemented in the IOH shown in Figure 2, and with a large and cumbersome 64-byte descriptor. Obviously there is inefficiency for asynchronous variable sized data such as networking, especially if the network packets are smaller than the descriptor that needs to be setup before any DMA copies can be made between I/O adapter and memory. As a result, the DMA offload is often used for memory-to-memory copies, such as between kernel space and user space. In networking, this shows little if any benefit to CPU utilization [13], in part since memory accesses can be pipelined and hide a software core-controlled memory-to-memory copy. Possibly the reason for a DMA engine in the IOH rather than CPU socket is to conserve MCP silicon for core and caching resources.

3.3.1.4 III.A.4 PCIe Optimization

Yu et al. described how existing DMA transfer speed can be improved by increasing buffer efficiencies for large block DMA transfers on a particular PCIe implementation (PEX8311) [14]. Figure 6 shows the speedup, but this result is relevant mainly for large size DMAs and not for a broad spectrum of I/O sizes.

An additional detriment to Yu's approach and in general is that it is common practice in PCIe devices to support a maximum PCIe frame size of 256 bytes. The reasoning is that PCIe device buffering needs to increase to support larger PCIe frame sizes at the cost of silicon area. Any PCIe enhancement needs to address the ubiquitous fragmentation of large I/O frames into 256 byte (or smaller) PCIe transactions. This fragmentation of large I/O messages into PCIe transaction comes at a cost to total bandwidth discussed further in section IV.B. The PCIe protocol [15] specifies three headers for each transaction: Transaction Layer Protocol (TLP), Data Layer Protocol (DLP) and Link Layer Protocol (LLP) which combined add 24 bytes to each PCIe transaction.

In [16] Tumeo et al. provide details in optimizing for double buffering, and how to use

this technique to optimize latency in a multi-core FPGA with each core having a DMA engine to move data between system memory to core memory.

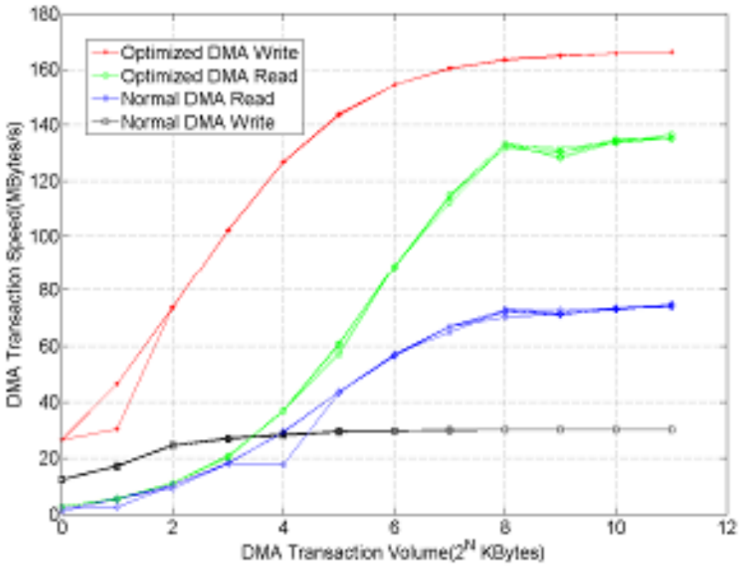

Figure 3.6: Speed Statistics for DMA

3.3.1.5

III.A.5 Predictability and Real-Time Operating Systems (RTOS)

In RTOS, I/O latency prediction is an important factor to guarantee predictable operations. Several papers [17-21] focusing on real time systems present models on how to accurately predict and bound the DMA latency when controlled by the IO adapter. Worst Case Execution Time (WCET) is the most critical parameter to consider. If the DMA state machine is in the core or processor socket, the predictions can be more accurate and latencies can be reduced. This is particularly true since the DMA engine in the I/O adapter often runs a much slower clock frequency than the core.

However, I/O adapter DMA engines remain common in RTOS systems since standard

"bit-banging" by the core results in poor I/O performance. Salah and El-Badawi in particular compared programmed I/O (PIO) to DMA, where PIO results in only 10% throughput compared to DMA throughput measurements. [17].

3.3.1.6 III.A.6 Other DMA Related proposals

The dynamic compression of data for memory accesses method discussed in [22] is an option to reduce I/O latency and reduce the chip-to-chip bandwidth, but the added logic complexity required of compressing and decompressing payload data does not make sense when many I/O adapters, such as the Intel 82599 NIC, are clocked at slow speeds (155 MHz) to reduce power and technology requirements.

Two papers [23, 24] deal with the challenges of asynchronous DMA where it can yield lower latencies across a DMA interface, which typically requires scheduling by a DMA controller. Their results show that asynchronous DMA is more appropriately targeted for a heterogeneous clock domain, which is fundamentally already done in the Intel 5500 system where the NIC is running at 155 MHz, the PCIe Gen2 interface at 2.5 GHz (double-clocked or 5GT/s), and core at 2.6 GHz. At most a few clock cycles are saved with an asynchronous DMA implementation.

3.3.2 III.B Generic workstations and servers

Systems that run standard multi-user and multi-processing operating systems such as Unix and Windows use more complex I/O structures. Since the application software is usually an abstraction of hardware capability to virtualize the I/O ports, the CPU can be utilized very effectively during I/O transactions on other tasks. This has led to the distributed I/O model where each I/O port may have a DMA engine to move incoming system data to system memory and outgoing data from system memory. Offloading the actual I/O transaction to a non-CPU-based DMA engine is very effective since often the I/O device is a 100MHz state machine while the CPU may be multiple

GHz, allowing the I/O to proceed as fast as the I/O device can transfer data (network, storage or video).

Since CPU and memory performance capabilities have increased faster than I/O performance, the descriptor-based DMA mechanism described in Section II is used for a variety of devices in the system. These include not just add-in cards, but also the on-board NIC, USB, and storage controllers.

3.3.2.1 III.B.1 Operating System Virtualization Protection

Although the mechanism to place a kernel barrier between a user application program and hardware makes sense when there are multiple potential processes requesting I/O services, there is a penalty in performance. I/O latency increases since the application software needs to first request kernel services to perform any I/O transaction. Throughput may decrease since buffers need to be prepared and managed using system calls which often include CPU context switches.

An alternative to always having kernel interaction with system I/O is to have carefully controlled user-mode DMA I/O. The two proposals discussed in [25, 26] describe how to bring kernel based system calls for moving data into user space that can be accessed by normal applications. Significant performance improvement can be obtained for small I/O messages by having a user application directly control data movement rather than using a system call that require a context switch. However, there are serious security risks in that any user application can access physical memory. The risk grows when systems are interconnected, potentially allowing physical memory access not just from other processes within a system, but processes on other systems.

3.3.2.2 III.B.2 Interrupt moderation

Interrupt moderation is also described as interrupt coalescing or interrupt aggregation.

The purpose is to interrupt the CPU on received data with relatively low latency and reduce the number of context switches a CPU requires for received network traffic. Common methods are discussed in [27] where two timers can be used. An absolute timer with a default, yet configurable, interval of 125us to interrupt after any received data arrival, and a per packet timer that can expire based on each received packet arrival.

Since network traffic can be both latency sensitive and throughput sensitive, adaptive interrupt moderation has been an ongoing area of research and implementations. For example Intel's 82599 NIC [28] allows interrupt filtering control based on frame size, protocol, IP address and other parameters. This Low Latency Interrupt (LLI) moderation is based on credits that still allows interrupt moderation, but based on event control rather than on timer control.

3.3.3 III.C Datacenters and HPC cluster systems

Systems that are used in datacenters and HPC clusters have requirements beyond generally stand-alone workstations and servers. Details on the interconnects are in section III.D, but there are internal system details that are important to consider discussed in the sections below. Often these systems are composed of high-end servers using multiple 10GbE interconnects , 15,000 RPM disks and SSD clusters. This drives the internal system requirements to use high performance scalable protocols such as Serial Attached SCSI (SAS) for storage and low latency Ethernet or Infiniband for internode communications.

Since Ethernet is ubiquitous and flexible as both a network and storage interface, we consider some of the high-end capabilities found in datacenters as link speeds increase to 10GbE and beyond.

Some HPCs will actually have front-end CPUs to prepare the I/O in memory for supercomputer execution. This places a software aspect that from an I/O performance perspective is not interesting for hardware I/O transactions.

3.3.3.1 III.C.1 Receive Side Scaling (RSS)

RSS [29]is the ability to demultiplex Ethernet traffic and spread effectively the flow across multiple available cores within a system. An example would be a web server supporting thousands of simultaneous TCP sessions. Each session is hashed using a Toeplitz hash table to properly direct the receive traffic to a core that presumably maintains context for the related TCP session.

Current RSS implementations are not effective at power management, so it is conceivable that a 16 core system, servicing only 16 low-bandwidth TCP sessions would utilize all 16 cores to service the incoming traffic, regardless of the bandwidth requirements.

3.3.3.2 III.C.2 Large Receive Offloading (LRO)

LRO [30] is also called Receive Side Coalescing (RSC) by Intel [28] as a receive mechanism to clump sequenced frames together, presenting them to the operating system as a single receive frame. Effectively this allows creation of a jumbo frame avoiding the higher level software layer to patch the discrete smaller Ethernet frames (that typically has a maximum of 1500 bytes) together.

3.3.3.3 III.C.3 Large Segment Offload (LSO)

LSO [31] is also called TCP Segmentation Offload (TSO) or Generic Segmentation Offload (GSO) allowing an I/O device to be given a pointer to a segment of data much larger than a single frame (e.g. 64KB) and allow the I/O device to stream the data for transmits. At 64KB sized messages LSO can improve performance by up to 50%.

3.3.3.4 III.C.4 Descriptor packing

An Ethernet descriptor size is typically 16 bytes and thus multiple descriptors can be read from a single doorbell [32] (e.g., if the cache line size is 64B, 4 descriptors would be read at once). This works well when the NIC is prefetching descriptors to DMA future receive packets.

In transmitting packets, descriptor coalescing can also be used and reduces the overall transmit latency. However, a server supporting web or database transactions for thousands of sessions over multiple descriptor queues may not allow for descriptor bundling. Transmit descriptor coalescing is beneficial if large multi-framed messages are being processed. For example, a datacenter where multiple systems are physically co-located may simply choose to enable jumbo-frames, which again may cause the transmit descriptor serialization described above

3.3.3.5 III.C.5 TCP/IP Offload Engine (TOE) and other offload engines

Offloading the entire TCP/IP stack onto the I/O adapter can be done to reduce the stack protocol overhead on the CPU [33] as shown in Figure 7.

This approach makes sense for large blocks of I/O, such as storage with average block size greater than 4KB. The downside of this approach is that HPC and datacenter servers are processing small messages as well. For example, virtualizing multiple operating systems within a single system is common due to increased capabilities of processors and memory. This leads to a single I/O adapter processing small packets over thousands of TCP connections. The TOE is also processing the frames at slower frequencies than a modern CPU. This adds significant amount logic complexity and connection context memory requirements to the less flexible TOE IO adapter [33].

In general, any higher level software I/O protocol may be implemented in an I/O adapter. For instance the Intel 82599 [28] does not offload the TCP stack, but does offload IPsec

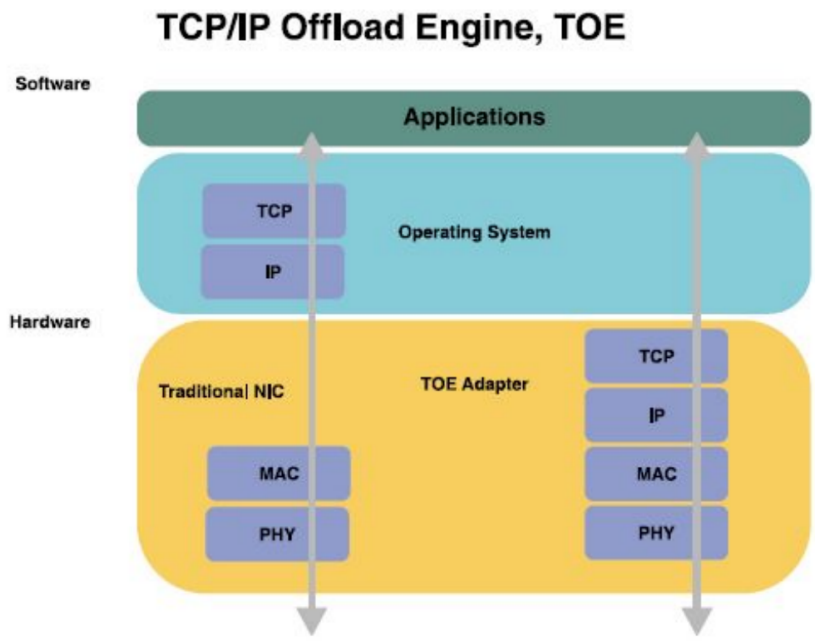

Figure 3.7: Basic TOE comparison to software-based implementation

and Fibre Channel over Ethernet protocols.

Myrinet is another type of TOE engine that provides a commonly used HPC interconnect using low cost Ethernet over a non-TCP/IP proprietary fabric [34]. Although Ethernet protocol is used, the higher layers need to be routed using Myrinet routing devices.

I2O [12] is used in some HPC environments, such as the Large Hadron Collider and is also an offload protocol mechanism. In order to reduce the protocol processing overhead, the I2O device driver is split into two sections: the traditional application and operating system space, and the Hardware Device Model (HDM). HDM is an IO processor to handle device specific tasks. Figure 8 shows a typical I2O configuration.

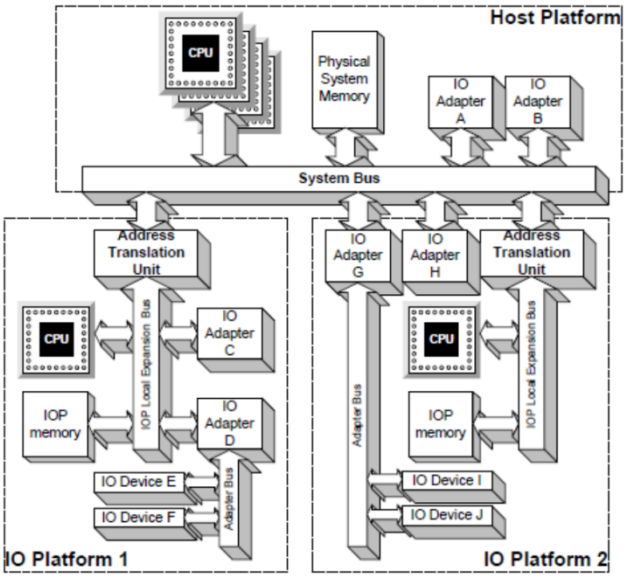

Figure 3.8: I2O fabric diagram

3.3.3.6 III.C.6 Coherent Network Interfaces (CNI)

Mukherjee et. al. proposed a coherent NIC interface [35] which was followed by an implementation on the Front-Side Bus (FSB) architecture by Schlansker et al.[4] The block diagram of the coherent memory implementation is shown in Figure 9.

The key detriment to this implementation proved to be the coherency traffic as the 1GbE coherent NIC interface performed snoops and write-back operations between cores and NIC buffers. This additional FSB traffic conflicted with CPU related traffic between cores on the FSB. [36]

Work such as Dan et. al. [37] explores the idea of having a separate I/O cache structure, allowing DMA traffic to be treated with a modified MOESI coherence protocol. However, it is unclear if the silicon area required in supporting such a cache partitioning and coherence trades off the performance improvements.

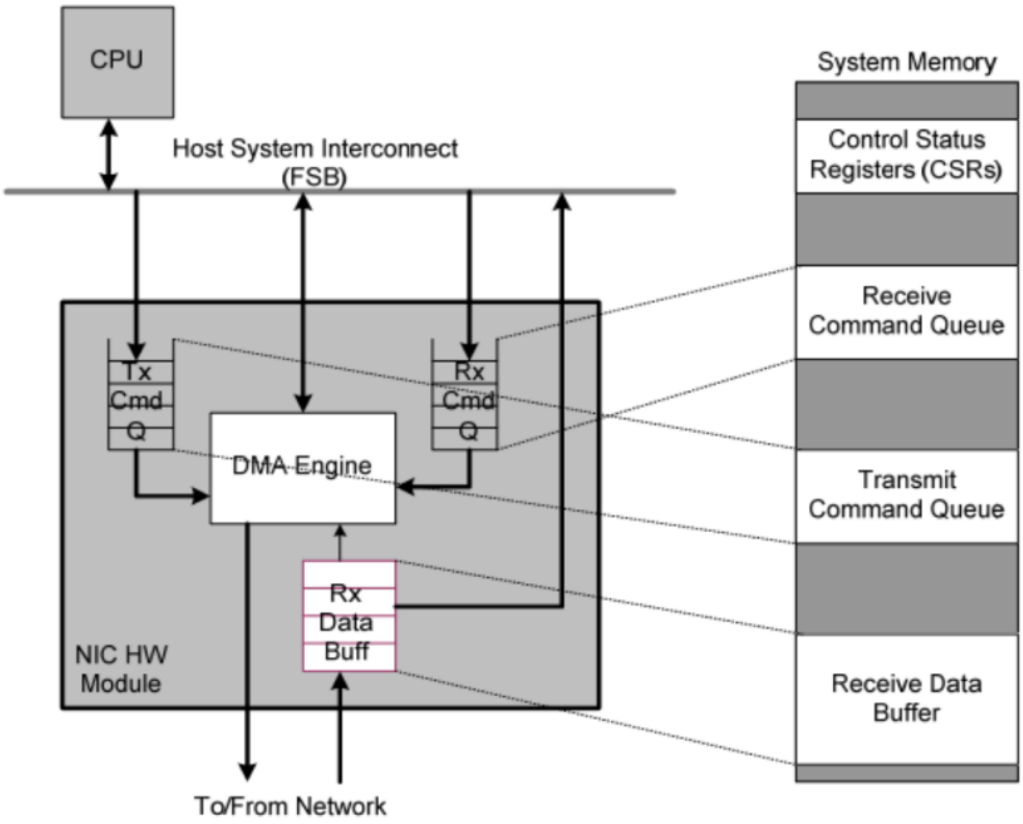

Figure 3.9: NIC system memory apertures

3.3.3.7 III.C.7 Network Interface Integration

One approach, which is basically taken for granted, is that integration of I/O adapters closer to the processor will make I/O performances improve. The design of Sun's Niagara2 processor with integrated 10GbE showed that the internal architecture needs to be carefully considered to match the desired network performance [3]. Simply gluing I/O adapters to processors can often be a waste of die area, power, and development time with little performance improvement. Figure 10 shows how discrete NIC adapters (DNIC) perform relative to integrated NIC adapters (INIC) on the Sun Niagara2 integration [3].

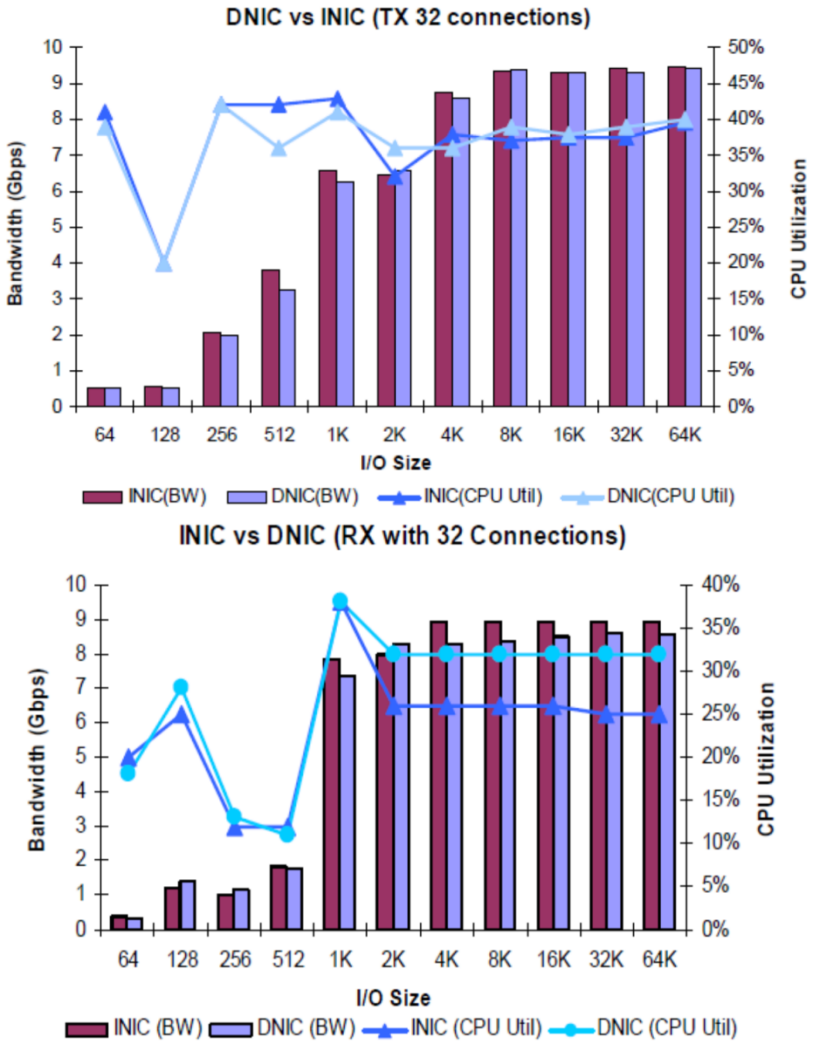

Figure 3.10: Discrete vs Integrated NIC performance

3.3.4 III.D System interconnects and networks

Since we are examining how data can be moved optimally within a system, we need also consider how interconnect architectures generically move data within a system and between systems. A particular emphasis is given to High Performance Computing (HPC) environments since this area bears more importance on latency and bandwidth

performance characteristics.

3.3.4.1 III.C.1 CPU Socket Interconnect

AMD Hyper-Transport [38] and Intel QPI [5] CPU interconnects are two widely accepted and competitive implementations of data movement between multiple CPU sockets. Communication between cores is usually done using the memory coherency specified by the MESI(F) [39] protocol. They both use snoops and coherence directories to determine a common state of all memory locations.

While this interconnect could be used for system I/O transactions, there has not been any wide-spread demand for it, and a coherent I/O interface would need to address the coherency overhead discussed in section III.B.6

3.3.4.2 III.D.1 Messaging between cores

Intel has explored I/O messaging with the Single-chip Cloud Computer (SCC) [40]. This defines a new non-coherent memory type for I/O where software controls what is written and read from a message passing buffer shown in Figure 11. This allows for efficient movement of data between cores while avoiding cache coherency overhead in the form of snoops and data write-backs. This becomes particularly important as the core count increases to 48, as is the case for SCC, and beyond. Extending this example to generic I/O, it can be seen that an MCP IO engine to address non-coherent memory, or a variant of the write-combining (WC) memory type is feasible and architecturally beneficial.

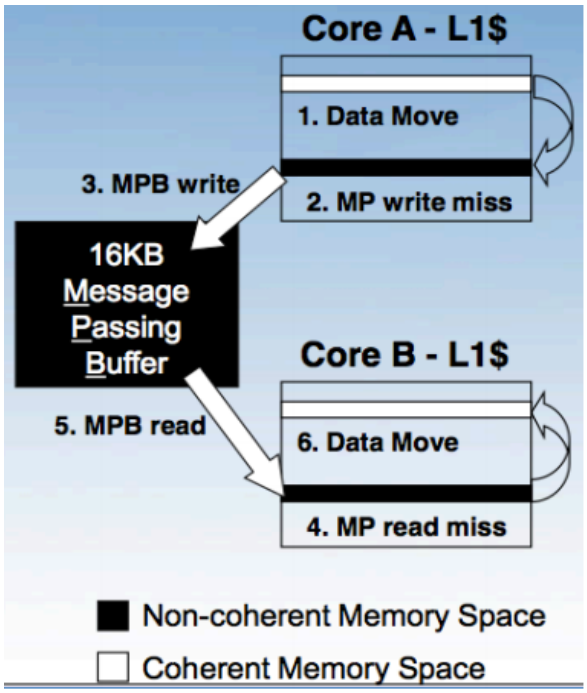

Figure 3.11: SCC message data types

3.3.4.3 III.C.3 InfiniBand

InfiniBand is considered a high bandwidth and low latency system interconnect [28]. It is the 1999 merger of two competing specifications from Future IO and Next Gen IO [41]. InfiniBand lacks broad popularity due to the high porting cost both in hardware and software when compared to the ubiquitous Ethernet. It shares many similarities, as shown in Figure 12, with TOE/Myrinet/I2O in that transmit and receive queues are processed without the OS involvement. However, this adds hardware complexity to the IO adapter to track the connection context.

Infiniband is a performance reference not only in latency but throughput as well. In a comparison of Infiniband Double Data Rate (DDR at 8Gbps) and Quad Data Rate (QDR at 16Gbps) I/O adapters on Intel platforms [42], the inter-node bandwidth requirements start to exceed the intra-node bandwidth available. In particular, DDR

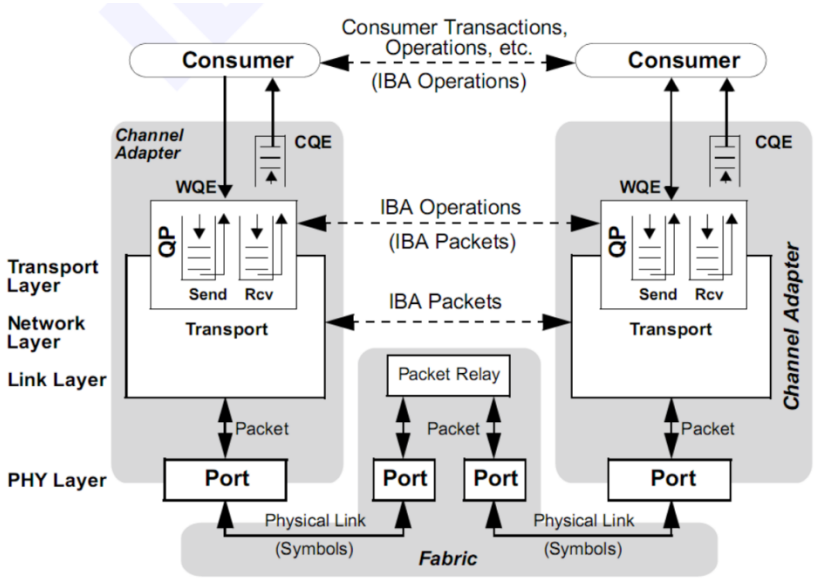

Figure 3.12: Infiniband communication stack

inter-system bandwidth exceeded the PCIe Gen1 host interface of 16Gbps in bandwidth traffic of small (less than 1KB) messages. This is an example where optimizing the chip-to-chip communication with an MCP I/O engine would have an immediate impact on current inter-node design.

The survey paper [42] introduces a communication balance ratio defined as the intra-system bandwidth divided by inter-system bandwidth of a network of systems. This ratio is desirable to have in a balanced system, but it needs to be scalable and capable of internal system bandwidth above the external network demand.

An example is an Infiniband adapter with a PCIe adapter bandwidth of 16 Gbps and an external system bandwidth of 8Gbps suggests for this particular case a 50% de-rating of inter-system bandwidth be done for external communications. In other words, an inter-system communication bandwidth should be multiplied by 0.5 to determine a balanced configuration both between systems and within systems. The de-rating is required due to the descriptor and doorbell mechanisms needed when there is a complex IO adapter directly addressing memory.

Figure 13 shows the intra-system bandwidth supported by the Clovertown DDR configuration (in yellow) is almost equal to inter-system bandwidth support over a range of 1-8KB of message sizes.

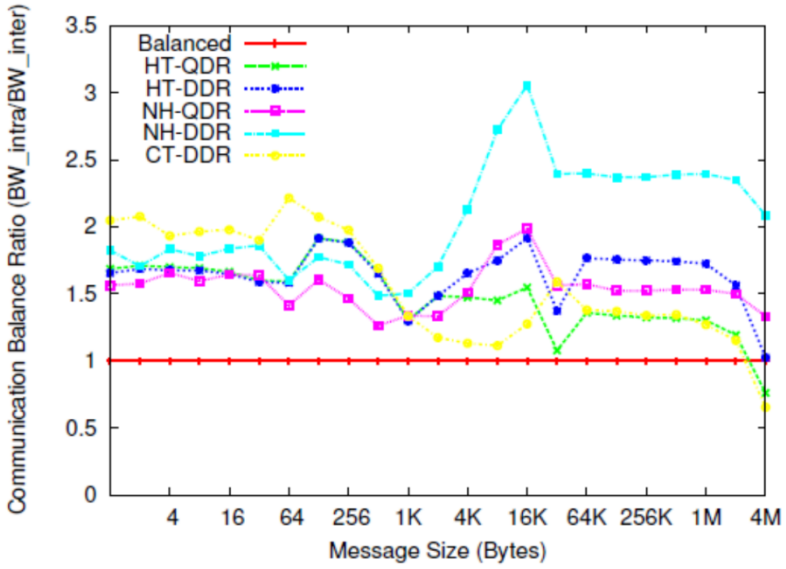

Figure 3.13: Communication balance ratio with Infiniband cluster

The communication balance proposal does not only hold true for Infiniband adapters, but also Ethernet adapters which normally provide about 40-50% more bandwidth on the PCIe interface than on the Ethernet interface to support the descriptor, doorbell, interrupt overhead traffic.

3.3.4.4 III.C.4 SciCortex Fabric

SciCortex uses a custom fabric using a Kautz graph to connect 5,832 cores on 972 nodes of 6 cores each. An important point of this architecture is that relatively low performance cores were used and the network hops between cores was optimized. Based on their HPC benchmark results [43], a bimodal message size pattern of 128 bytes and

100 KB was found. In this case, PIO would be better for small messages with an MCP I/O engine support for large messages.

Due to the importance of reducing latency, a mesochronous clocking structure was architected with a single master clock that was then PLL locked to higher inter-node frequencies. This is basically the same method used in PCIe/memory/core clocking on a single system. A combination of power supply current and this single clock can cause inter-node sync loss. With the proposed MCP I/O engine, the primary logic will be running on the same PLL as the core/memory interface.

3.4 IV. MEASUREMENTS AND QUANTIFICATIONS

In order to quantify aspects of current HPC I/O design and explore potential changes, a conventional NIC was placed in an I/O Hub (IOH) slot of an existing platform. A PCIe protocol analyzer was used to observe the PCIe transactions and the impact of descriptor-based I/O adapter DMA engine was measured in terms of several factors, which are summarized in Table 1. Using measurements on a real (and current) system offers validity in extrapolations and conclusions that are less certain in simulated environments.

Factor	Measurement unit	Descriptor DMA	Estimated Potential Improvement	Comment/justification	Section
Latency	microseconds to transmit a TCP/IP message between two systems	8.8	16%	Descriptors are no longer latency critical	IV.A
Bandwidth-per-pin	Gbps per serial lane link	2.1	17%	Descriptors no longer consume chip-to-chip bandwidth	IV.B

Table 3.1: Quantified benefits of an MCP I/O engine

3.4.1 IV.A Latency

Latency is a critical aspect in network communication that is easily masked by the impact of distance. However, when inter-platform flight-time of messages is small, the impact of latency within a system is much more important. One such example is automated stock market transactions (arbitrage and speculation) as demonstrated by Xasax claiming 30 µs latency to the NASDAQ trading floor [44]. Another example is High Performance Computing (HPC) nodes where LinPack benchmark (used to define the Top500 supercomputers) share partial calculations of linear algebra matrix results among nodes.

Figure 14 shows the typical 10Gb Ethernet (GbE) latency between a sender (TX) and a receiver (RX) in a datacenter environment where the fiber length is on the order of 3 meters. These results are based on PCIe traces of current 10GbE Intel 82598 NICs (code named Oplin) on PCIe ×8 Gen1 interfaces [45]. The latency benchmark NetPIPE is used to correlate application latencies to latencies measured on the PCIe interface for 64-byte messages. The 64-byte size was used since it is small enough to demonstrate the critical path latencies, but also large enough to represent a minimal message size that can be cache-line aligned.

End-to-end latency consists of both hardware and software delays, and depends on many aspects not directly addressed in this paper, such as core and memory clock frequencies, bandwidth, and cache structure. The critical path of the software stack is not related to non-descriptor-based IO communication since it is only associated with how a core handles I/O traffic when I/O data is already in main memory. Software latency in terms of the core cycles required to formulate TCP/IP frames in memory for transmit and processing received TCP/IP frames residing in memory is described in more detail in [27]. The hardware latency can be split into three portions. First, the TX NIC performs DMA reads (NIC-TX) to pull the data from the system memory to the TX NIC buffer. This is followed by the flight latency of the wire/fiber and the TX/RX NIC state machines (NIC to NIC). Finally, the RX NIC performs DMA writes (NIC-RX) to push the data from the RX NIC buffer into the system memory and interrupts a core for software processing. The total latency $\text{Latency}_{\text{Total}}$ can be

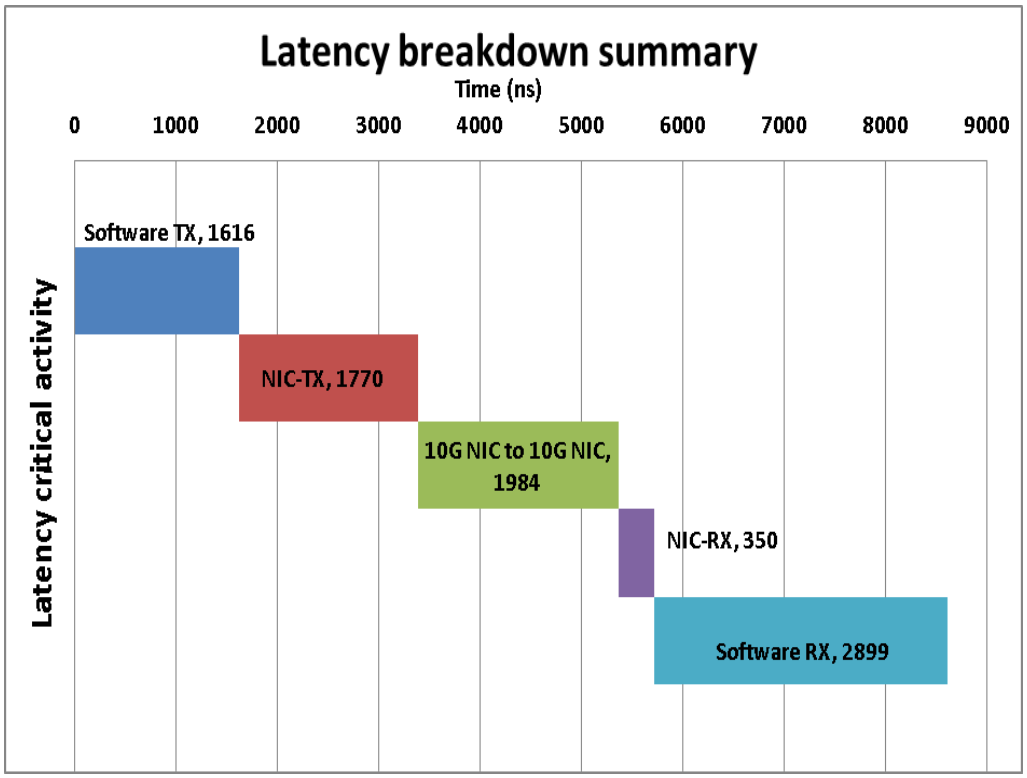

Figure 3.14: Critical path latency between two systems

expressed by the following equation:

$$\text{Latency}_{\text{Total}} = \text{Tx}_{\text{SW}} + \text{Tx}_{\text{NIC}} + \text{fiber} + \text{Rx}_{\text{NIC}} + \text{Rx}_{\text{SW}}$$

The latency for the NIC-TX portion can be further broken down using PCIe traces as shown in Figure 15. A passive PCIe interposer was placed between the platform PCIe slot and the Intel 82598 NIC. PCIe traces were taken from an idle platform and network environment. These latencies are averaged over multiple samples and show some variance, but it is under 3%. The variance is due to a variety of factors, such as software timers and PCIe transaction management. Based on a current 5500 Intel processor platform with 1066MB/s Double Data Rate (DDR3) memory, the doorbell write takes 230 ns, the NIC descriptor fetch takes 759 ns, and the 64B payload DMA read takes 781 ns. The core frequency is not relevant since the PCIe NIC adapter controls

the DMA transactions.

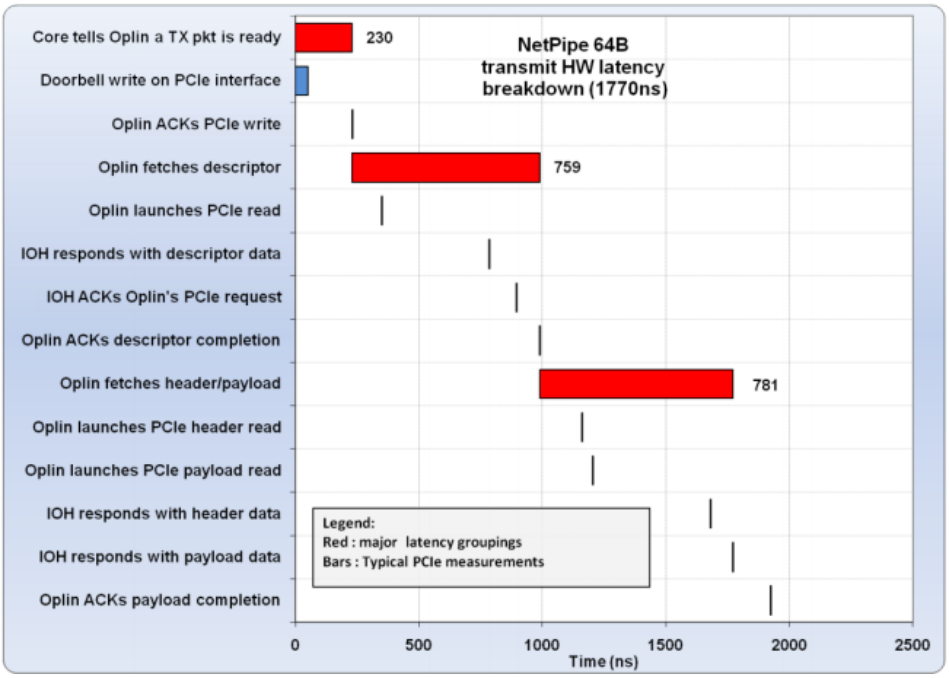

Figure 3.15: NIC TX latency breakdown

Latency can be reduced by not relying on an IO adapter's DMA engine and descriptor handling. If the DMA engine of the IO device can be integrated on the CPU, it is possible to achieve a 16% reduction of the 8.8 μs end-to-end latency and also reduce the size of the IO device memory buffer.

3.4.2 IV.B Throughput and Bandwidth Efficiency

The iperf bandwidth benchmark was used on a dual 10GbE Intel 82599 Ethernet adapter (codename Niantic) [28] on an Intel 5500 server. The primary difference between this and the previously described experiment is that PCIe bandwidth has doubled. The internal state-machine driven NIC and core frequency remains the same. PCIe captures

consisted of more than 300,000 PCIe ×8 Gen2 packets, or 10 ms of real-time trace, which gives a statistically stable capture for analysis.

Figure 16 shows a breakdown of transaction utilization in receiving and transmitting data on a PCIe interface for a dual 10GbE NIC. The four bars show extreme cases of TCP/IP receive traffic (RX_) and transmit traffic (TX_) for small (_64B_) and large (_64KB_) I/O sizes. Receive traffic performance is important for applications such as backup and routing traffic, while transmit traffic performance is important in serving files and streaming video. I/O operations on small messages is representative of latency sensitive transactions and large I/O is representative of storage types of transactions. As can be seen by the figure, descriptors and doorbell transactions for small messages consume a significant amount of bandwidth. This includes PCIe packet header and CRC data along with PCIe packet fragmentation. The proportional PCIe bandwidth used for transmitting small payload sizes utilizes up to 43% of the PCIe bandwidth for descriptor and doorbell traffic that could be removed by non-descriptor DMA methods. In the case of I/O receive for small payload sizes, the benefit reduces to 16% since 16B descriptors can be pre-fetched in a 64B cache-line read request. For large I/O message sizes, the available PCIe bandwidth is efficiently utilized with less than 5% of the bandwidth used for descriptors and doorbells.

In general, network traffic has a bi-modal packet size with concentrations of small and large Ethernet frames. Some applications will have higher proportions of small frames while other applications will have large frames, but we could project that on average small and large impacts can be averaged. Therefore, based on the measurements shown in Figure 5 we can take the range of bandwidth benefits of the four columns (1% to 43%). A basic summary of the benefit could be an average across these extremes with 17% of bandwidth optimization due to descriptor, doorbell, and interrupt removal from the PCIe interface.

Further measurements are done on a SPECweb 2009 workload where a web server is being requested a range of file sizes emulating a typical web-based file server. The average 18% of non-payload related PCIe traffic provides another datapoint that a significant amount of non-payload bandwidth is required to move network I/O related traffic

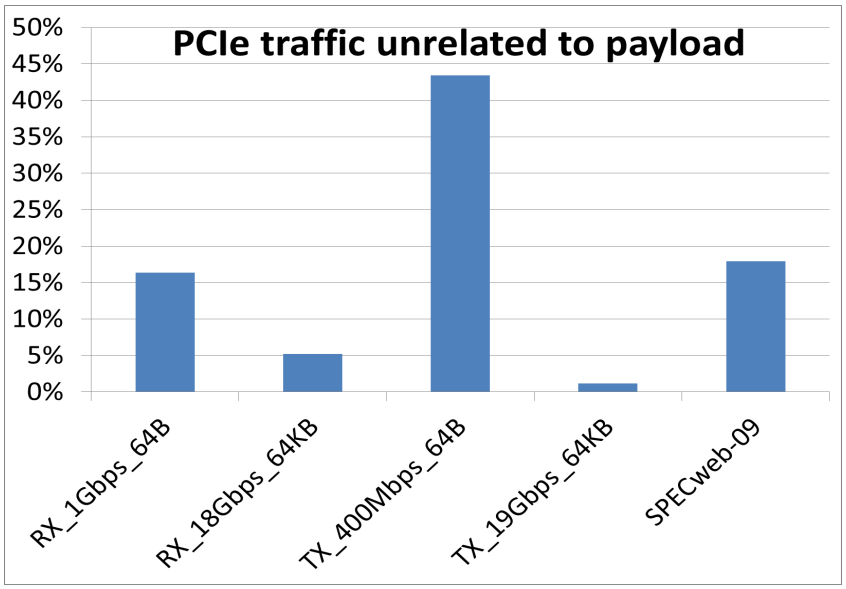

Figure 3.16: Raw PCIe bandwidth not used for payload transfer

across the PCIe interface.

3.4.3 IV.C Other Descriptor DMA Considerations

There are other less quantifiable aspects of descriptor-based DMA transactions.

1. In current systems there is little priority in matching I/O bandwidth and core and memory bandwidths. Although there are a variety of good reasons, often systems can be found with too little I/O capability or inflexible in expanding I/O bandwidth and latency capabilities. This is exacerbated by having multiple MCP devices with possibly integrated IOHs.

2. Power management efficiency where mechanisms like RSS can spread I/O traffic across all available cores, thereby wasting power that could be localized to a subset of cores.

3. Quality-of-Service in that as I/O increases in bandwidth and complexity, having

independent DMA engines contending for memory bandwidth reduces quality control.

4. Silicon cost and complexity is increased by having independent DMA engines servicing I/O transactions.

5. Security is a concern in that a card slot can have a DMA agent read and write to physical memory.

3.5 V. PROPOSED OPTIONS

The prior sections have shown how I/O uses DMA engines on the I/O adapters to move network, storage and video within a system. As CPU and memory performance increases faster than I/O requirements, it is important to reconsider the premise that I/O tasks should always be off-loaded to I/O devices in the descriptor-based DMA method. To this end, we propose an MCP-based iDMA engine.

The iDMA would be implemented as a simple micro-controller or enhanced state machine that manages the data flow between an arbitrary I/O device and system memory. It would basically act as a heterogeneous or SoC core to the larger application generic cores. The arrow from the I/O adapter shows the iDMA pulling receive traffic from an I/O adapter into system memory. The arrow from system memory shows the iDMA reading memory and pushing transmit traffic to the I/O adapter. Since I/O messages and cache-line memory accesses occur at different times and with different sizes, basic buffers are needed to support arbitration and fragmentation of off-chip interfaces.

More details are discussed by Larsen and Lee [46] as how Table 2 results in performance improvements of I/O transactions.

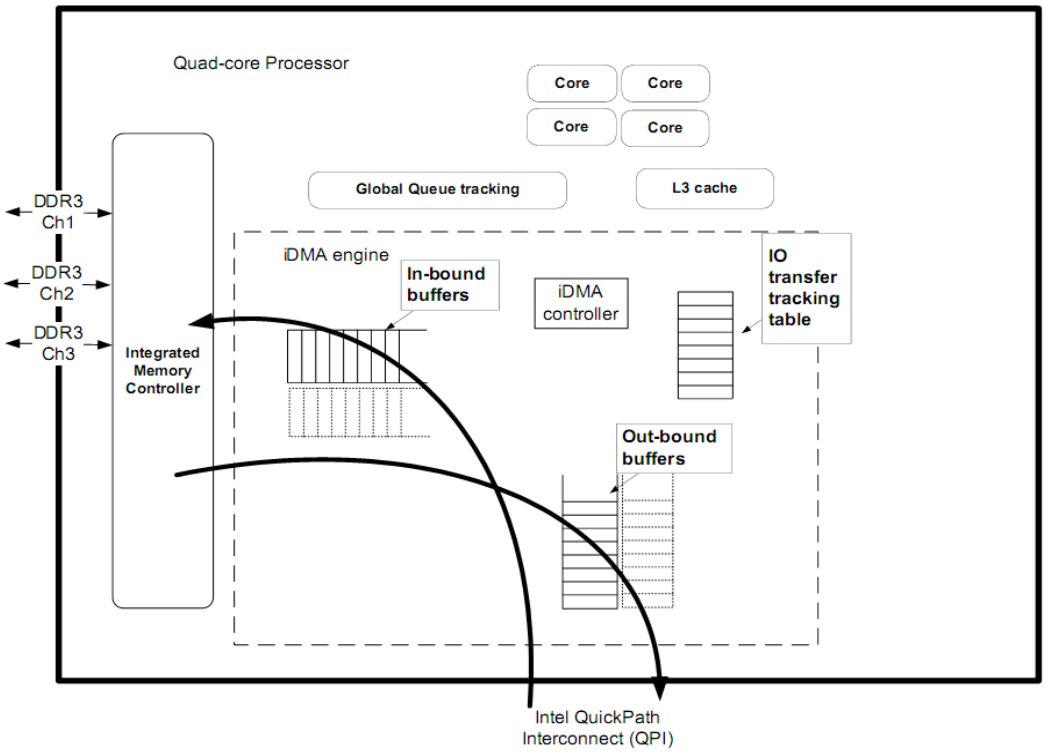

Figure 3.17: An option for MCP integrated I/O acceleration

3.6 VI. CONCLUSIONS AND FUTURE WORK

This survey of I/O methods and existing optimizations point to a potential benefit of having a CPU-based I/O engine. The initial approach of questioning and simplifying I/O structures as core cycles become less valuable due to MCP and SMT and increasing I/O requirements suggests an I/O accelerator to consider in future platform architectures.

Future work planned is adapting the write-combining memory type to generic I/O and offloading I/O data movement to a generic platform requirement (networking, storage, and graphics) suggesting quantifiable latency and bandwidth-per-pin advantages and potential benefits in power, QoS, silicon area, and security for future datacenter server

Factor	Measurement unit	Descriptor DMA	Potential for non-descriptor	Comment/justification
Latency	microseconds to transmit a TCP/IP message between two systems	8.8	16%	Descriptors are no longer latency critical
Bandwidth-per-pin	Gbps per serial lane link	2.1	17%	Descriptors no longer consume chip-to-chip bandwidth
Bandwidth right-sizing	Not quantifiable	na	na	Reduced platform silicon area and power
Power efficiency	Normalized core power (maximum)	100%	71%	Power reduction due to more efficient core allocation of IO
Quality of service	Nanoseconds to control connection priority from software perspective	600	92%	Round trip latency to queuing control reduced from PCIe to system
Multiple IO complexity	Die cost reduction	100%	>50%	Silicon, power regulation and cooling cost reduction of multiple IO controllers into a single
Security		na	na	not quantifiable

Table 3.2: Potential benefits for non-descriptor DMA

architecture.

Hot-Potato I/O: Improving and simplifying I/O DMA transactions on cluster systems

Steen K Larsen, Ben Lee

Submitted to : 10th IEEE International Conference on Networking, Architecture, and Storage
http://www.nas-conference.org/

Chapter 4: Hot-Potato manuscript

Hot-Potato I/O: Improving and simplifying I/O DMA transactions on cluster systems

Steen Larsen

Intel Corp

22238 NW Beck

Portland OR 97231

(503) 846-1937

steen.knud.larsen@gmail.com

Ben Lee

School of Electrical Engineering and Computer Science

Oregon State University

1148 Kelley Engineering Center

Corvallis, OR 97331-5501, USA

(541) 737-3148

benl@eecs.orst.edu

ABSTRACT

Performance improvement of computer system I/O has been slower than processor and memory technologies in terms of reducing latency, increasing bandwidth, and other factors. Based on this observation, how I/O is performed needs to be re-examined and explored for optimizations. To rebalance the computer system having multiple cores and multiple CPUs with integrated memory controllers, this paper proposes a new system oriented I/O method where data movement is controlled more directly by the CPU. This is achieved by treating the ownership of payload data movement as a "Hot-Potato" and the I/O interfaces as simple FIFOs. Leveraging the write-combining memory type, our experiments with the proposed method show that transmit latency for small Ethernet frames reduces from 1,673 ns to 108 ns, and *Memcached* application testing shows 8% latency reduction.

Categories and Subject Descriptors

B.4.1 [**Input/Output and Data Communications**]: Data Communications Devices – *processors, channels and controllers, memory.*

General Terms

Performance, Design, Measurement

Keywords

I/O latency, memory, DMA, I/O bandwidth, communication

4.1 INTRODUCTION

The data movement within a computer system becomes more complex as the system components become more complex. This is due to the proliferation of cores, Graphics Processing Units (GPUs), Non-Uniform Memory Access (NUMA) systems, deeper caching levels as well as the increase in communication bandwidth among these components.

I/O transactions in typical systems are not directly managed by the CPU, and thus,

there is no quality control between CPU core memory transactions and I/O DMA device engines that may be several silicon packages away from memory. Currently, the link between data producer software running on a core and the data departure from the system is decoupled, and the receiving data consumer software is decoupled from the data arrival to the system. This decoupling is done using *descriptor-based DMA data transfers* that allow an I/O device to execute I/O transactions as fast as it can handle.

This type of non-managed I/O transaction control is appropriate in a system where core cycles are valuable as I/O is typically much slower than higher level core software processing and power per I/O transaction is low. Today's I/O devices, such as Network Interface Controllers (NICs), storage drives, and Universal Serial Bus (USB), have traditionally been orders of magnitude lower in bandwidth than the core-memory complex. For example, a modern 64-bit core running at 3.6 GHz compared to a 1.5 Mbps USB1.1 mouse has 153,600 times higher bandwidth. Therefore, it makes sense to offload the cores by allowing the I/O adapters some control over how I/O data is pushed/pulled to/from memory. This allows a core to context switch to other tasks while a slower I/O adapter performs reads/writes from/to memory.

However, as CPUs with multiples cores and Simultaneous Multi-Threading (SMT) make this bandwidth ratio even higher, I/O transaction latency and throughput performance need to be balanced with the improvement in CPU performance. Therefore, there are benefits to having a more directly involved CPU interaction with I/O transactions, similar to how SoftModems [1] simplified modem I/O by bit-banging a protocol over a controller-less device.

As such, this paper proposes the "Hot-Potato" approach to improving I/O performance. The implication of this name is that I/O transaction should be treated as a hot-potato between producer and consumer endpoints. That is, if the CPU generates the data, it is responsible for directly writing (or pushing) the data to I/O transmit buffers, while during receive operations the I/O adapter is responsible for performing DMA to write data to the system memory. This is in contrast to the current descriptor-based method where during I/O transmit the CPU writes the packet payload to system memory and then the I/O adapter performs DMA to move data from system memory to I/O adapter

buffers. The descriptor-based approach requires several I/O device read operations, i.e., round trips to memory, while Hot-Potato implements a more direct I/O approach using *Write-Combining* (WC) buffers. Therefore, our proposed method in essence simplifies I/O traffic between I/O interface and system memory, and thus leads to overall system goodness.

The proposed Hot-Potato approach was implemented as an I/O adapter FPGA and tested on a current high-performance system. Our study shows that the proposed method reduces latency by 1.6 μs, or about 16% system-to-system latency reduction, in a datacenter or a compute-cluster environment primarily by removing the descriptor and transmit DMA operations.

4.2 BACKGROUND

In order to better understand the motivation for the proposed method, this section discusses typical transmit and receive operations within a system. Fig. 1 shows a diagram of the internal system components of a current high-performance system based on the Intel 2nd Generation Core 2400S processor with integrated memory and PCIe controllers [2]. The two dotted arrows between the I/O adapter and the CPU indicate the paths and peak bandwidths available as the I/O adapter reads data to be transmitted from system memory and writes received data to system memory.

The system contains a single quad-core SMT processor with 6 MB L3 cache. The speed of each core (2.5 GHz) is not significantly applicable to the discussion at hand as it will be seen that the I/O transaction efficiency is governed more directly by the I/O device controllers. The processor integrates the PCIe interface directly onto the silicon, so there is no need for a QuickPath Interconnect (QPI) coherent memory interface between devices.

Fig 2 shows the typical Ethernet transmit flow, which involves the following sequence of operations:

(1) The kernel software constructs the outgoing packet in transmit buffers within

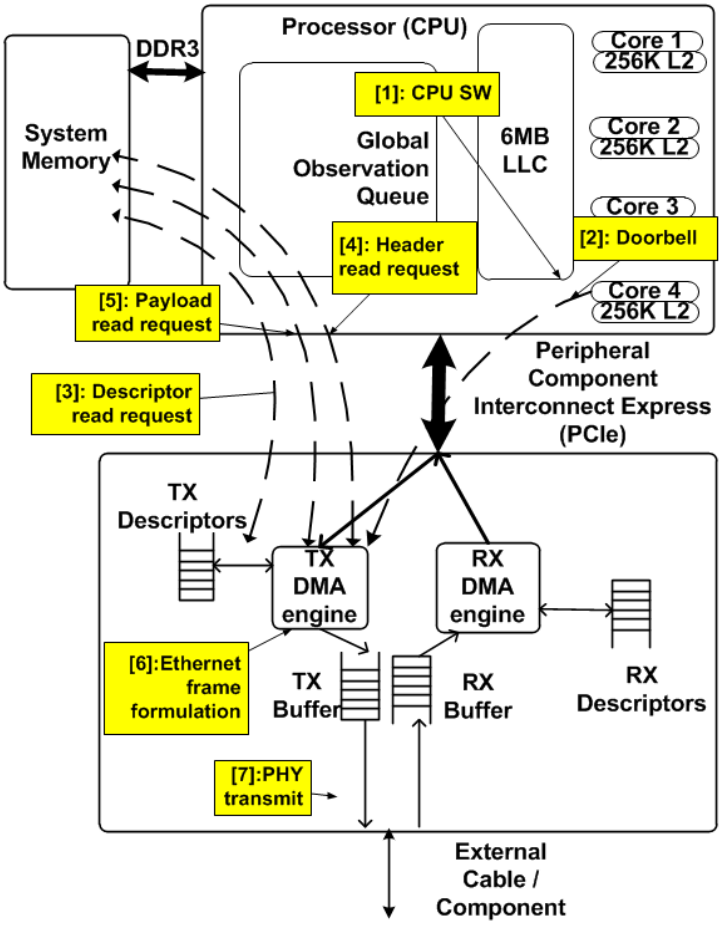

Figure 4.1: Typical Ethernet transmit flow

the system memory. This is required to support the protocol stack, such as TCP/IP, with proper headers, sequence numbers, checksums, etc.

(2) A core sends a *doorbell request* to the NIC via the platform interconnect (e.g., PCIe, but this also applies to any chip-to-chip interconnect within a platform) indicating that there is a pending packet transmission. This is a write operation to the memory space reserved for the I/O adapter, which is un-cacheable with implications that other related tasks that are potentially executing out-of-order must be serialized until the un-cacheable write completes. The core then assumes the packet will be transmitted, but

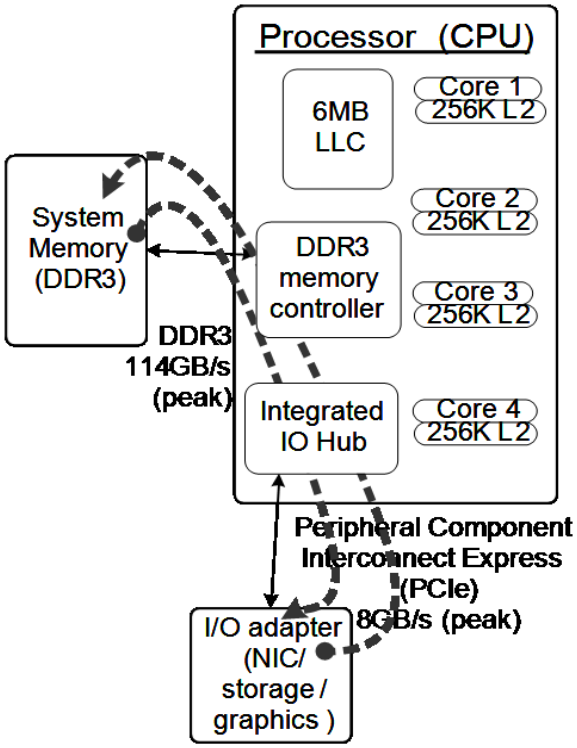

Figure 4.2: High-performance I/O system block diagram

the memory buffers will not be released until confirmed by the NIC that the packet has been transmitted.

(3) The doorbell request triggers the NIC to initiate a DMA request to read the descriptor containing the physical address of the transmit payload. The descriptor is not included in the doorbell write request because there are two descriptors for header and payload in an Ethernet packet definition, and a larger network message will require more descriptors (e.g., typical maximum payload for Ethernet is 1460 bytes). A tracking mechanism called a Global Observation Queue (GOQ) in the CPU controls memory transaction coherency such that snoops by memory requests from on-die cores or other CPUs are performed correctly before a transaction is completed. The GOQ also protects memory contention between I/O devices and the cores within the system.

(4) A memory read request for the descriptor(s) returns with the physical addresses

of the header and payload. After parsing the descriptor contents, the NIC initiates a request for the header information (e.g., IP addresses and the sequence number) of the packet.

(5) After the header information becomes available, a request is made to read the transmit payload using the memory location of the payload in the descriptor with almost no additional latency other than the NIC state machine.

(6) After the payload data is transferred from the system memory to the NIC TX buffer, the NIC state machine constructs an Ethernet frame sequence with the correct ordering for the bit-stream. The NIC will also signal the operating system (typically with an interrupt) that the transmit payload has been processed. The signal allows the transmit buffer of the packet data to be deallocated, and this operation is not in the transmit latency critical path.

(7) Finally, the bit-stream is passed to a PHYsical (PHY) layer that properly conditions the signaling for transmission over the medium (copper, fiber, or radio).

Fig. 3 shows the Ethernet receive flow, which is the reverse of the Ethernet transmit flow. The DMA sequences described below are typical of any I/O adapters including GPUs and storage adapters.

(1) The NIC pre-fetches a descriptor associated with the established connection, and matches an incoming packet with an available receive descriptor.

(2) The receive packet arrives asynchronously to the NIC.

(3) The NIC performs a DMA write to transfer the packet contents into the receive buffer in the system memory referenced by the receive descriptor.

(4) After the memory write transaction completes, the NIC interrupts the core indicating an arrival of a new packet for further processing. Interrupt throttling or coalescing can be used to reduce interrupt service time and is usually based on NIC timers.

(5) As part of the interrupt processing routine, the core driver software issues a

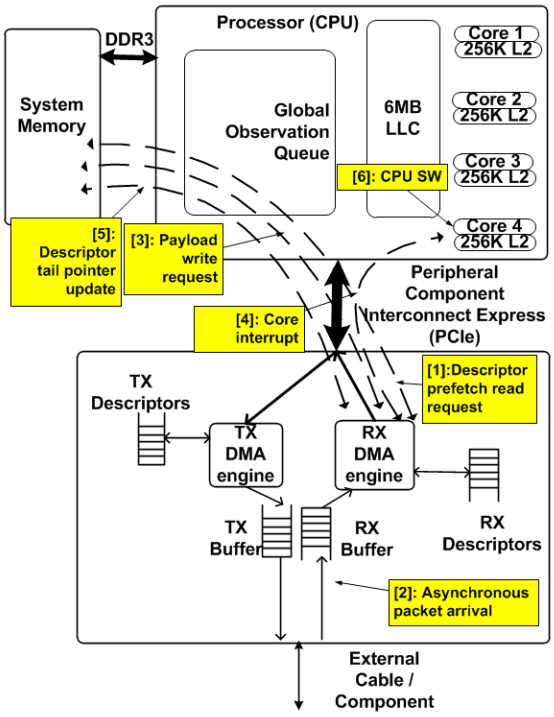

Figure 4.3: Typical Ethernet receive flow

write to the NIC to synchronize the NIC descriptor ring with the core descriptor ring. This also acts as a confirmation that the NIC packet has been successfully moved from the I/O adapter to system memory.

(6) Finally, the kernel software processes the receive packet residing in the system memory.

4.3 RELATED WORK

The most fundamental related work on simplifying I/O transactions can be found in programmed I/O (PIO) or user I/O that allows direct access by an application to perform I/O transactions. This approach avoids system calls to the operating system, which

may include memory copies to kernel buffers impacting I/O transaction performance. The primary issue with PIO is sharing, or I/O virtualization, which causes possible contention between multiple writers and readers for the shared I/O queues. While this can be addressed with virtualization software, (i.e., VMware and Xen hypervisors) this additional layer duplicates some legacy I/O transaction control that is commonly implemented by the kernel.

Legacy I/O has always been protected from user applications by the kernel. This security barrier is also a performance barrier and some work has been done to allow user-mode access to I/O adapters. The work in [3, 4] describe how kernel-based system calls can be used to move data more directly into user space, but requires user-space overhead to control hardware resource allocation. Significant latency improvement can be obtained for small I/O messages by having a user application directly control data movement rather than using a kernel system call that requires a context switch. A comparison of PIO, PIO with WC buffer [5] (which Intel has recently renamed them as fill buffers [6]), and DMA shows that PIO has lower latency for messages less than 64B. However, PIO+WC outperformed DMA for messages less than 1KB [7]. The PIO+WC transfer involves writing descriptors using PIO and network packets using the WC buffers. This approach is similar to the Hot-Potato proposal, but our approach does not use descriptors and is based entirely on WC buffers allowing for near system memory bandwidths.

Part of the reason PIO performs poorly for large I/O transactions is that they are treated as uncacheable memory transactions. As a result, reads are not allowed to pre-fetch and writes are not allowed to post multiple outstanding transactions. In addition, PIO transactions using the PCIe interface occur in maximum of 8-byte PCIe payload packets. The 24-byte PCIe packet header effectively reduces PCIe bandwidth utilization to 25% of the peak PCIe bandwidth. There are also serious security risks that allow other user applications to access the physical memory. Our method maintains the OS kernel protection, by using a kernel module (or driver).

Specialized approaches such as using coherent memory interfaces for communications have also been proposed. While this can reduce system-to-system latency to 970 ns [8], and is applicable to top-tier HPC systems, our goal is to explore the general-purpose clus-

ter interconnect for HPC and datacenters that is more cost sensitive. The Ethernet-based interconnects are ubiquitous for systems in this second tier as shown by the Top500 interconnect family breakdown, where 41.4% of the systems are Ethernet-based [2]. While InfiniBand, which also uses descriptor-based DMA, is also a common HPC interconnect [2], our experiments are baselined to Ethernet interconnects since we want to also examine I/O device simplifications. InfiniBand is a complex I/O device that offloads the connection management task (essentially the TCP/IP stack) to the I/O device, which requires a large device memory to support connection queues and re-ordering of packets.

Although Hot-Potato is based on the concepts of memory-mapped I/O (MMIO) for message transmit, WC buffers in the CPU core are used optimize this approach by buffering 64 bytes in each WC buffer. For example, each Intel core implements separate 6˜10 WC buffers allowing for pipelined write transactions at near system memory bandwidths when targeting PCIe memory writes [16]. Thus, if a core issues a 64-bit write each core cycle, a WC buffer would fill in 8 cycles and a 2.5 GHz core could be writing at maximum data rate of 2.5×10^9 cycles/second $\times$ 8 bits = 10 GBps. Assuming no core or CPU contention, a quad-core system would be theoretically capable of 40 GBps.

4.4 PROPOSED METHOD

The objective of the proposed Hot-Potato method is to allow I/O data to be serviced (or queued) immediately, instead of being indirectly handled using DMA descriptors. The structure of the proposed method is shown in Fig. 4, which consists of PCIe Host Interface, RX DMA, TX and RX queues, and the MAC and PHY layers.

From a software perspective, the Hot Potato method maintains a kernel-based approach where the I/O device is mapped to the kernel space (as opposed to the user space). This approach simplifies porting of higher level software stack protocols since the driver already has a network packet data structure with the proper frame headers and checksums. In addition to minimizing the software shift to support Hot Potato, this approach also provides the device sharing protection that is currently supported by mainstream kernels and operating systems (such as Linux and Windows).

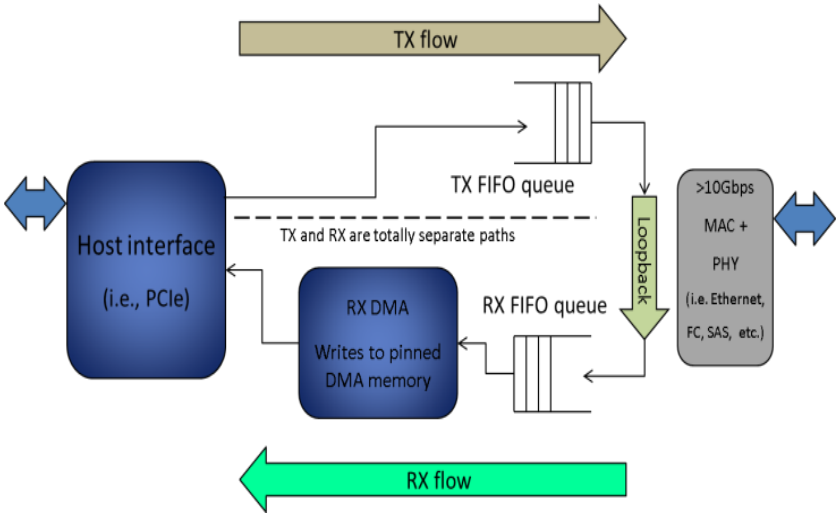

Figure 4.4: Structure of the proposed method

The following subsections describe I/O transmit and receive operations, as well as a couple of implementation issues.

4.4.1 I/O Transmit operation

In order to transmit an I/O message, the driver first initializes the TX FIFO as a *WC memory type*, which allows any writes to the TX FIFO to be performed using WC buffers instead of typical PIO. Then, the core formulates the message and appropriate header information. The header information is the standard Ethernet header with source and destination addresses and appropriate higher-level packet information, such as IP, TCP, ICMP, etc. The core writes the entire Ethernet frame to the I/O adapter TX FIFO queue. This is fundamentally the same as PIO access in the transmit direction (see Section 3), but WC buffers are used to increase the PCIe bandwidth and latency efficiency.

In our implementation, the driver is coded to align all transmit frames on a 64-byte WC buffer. This allows a core to write a 64-byte data to the PCIe interface with a 24-byte PCIe frame overhead. This is significantly better than the PIO method that

sends an 8-byte data (in a 64-bit operating system) on each PCIe frame. This improves the PCIe write throughput efficiency from 25% (8/(8+24)) to 72.7% (64/(64+24)) on the raw PCIe bandwidth.

The current Intel WC buffer implementation does not guarantee that the writes will occur in the correct order as issued by the core, and this is often referred to as weakly-ordered memory. This restriction is not a limitation in the Hot-Potato method since the Ethernet driver uses the WC buffer such that packets are sent out of the system as *non-temporal memory writes*, i.e., these packets will not be cached, and thus do not require coherency checks

The WC buffers are not directly visible to software and they can be mapped to different memory regions allowing each WC buffer to operate independent of other WC buffer address regions. In our implementation, there is a 4 KB address range allocated in the kernel memory for the transmit FIFO. Thus, when the driver writes to this region, there are no potential conflicts among different WC buffers mapped to different memory regions. This means that when a transmit packet is written, a new WC buffer is requested and each 64-byte region is filled. A WC buffer gets evicted from the CPU when it becomes full. In our case, the driver is executing in the kernel and the 4 KB transmit region is not being shared, and thus there is full control of the order between among WC buffers. Note that transmissions by multiple CPU cores would require either a locking mechanism or multiple TX FIFOs, such as seen in Receive Side Scaling (RSS) [9].

A limitation of WC buffers is that the data needs to combined until a WC buffer become full, or some other event flushes the buffer, to optimally coalesce the write operations. The WC buffers can be flushed with a cflush, sfence, or mfence x86 instruction [10], but each of these instructions is a costly operation of about 100 core cycles [11]. In our implementation, the eviction of WC buffers is carefully controlled in the kernel driver code to avoid this explicit flush requirement as explained below.

The Intel specification for WC buffer eviction [10] notes an "option" to evict a given WC buffer, but the wording is in the context of cache coherency. This is a critical specification for proper memory coherence (which does not apply to our non-temporal

data movement instructions) as well as the understanding that a partial fill of a WC buffer may not be evicted for long periods of time. Our interpretation is that as soon as a WC buffer is completely filled, its eviction is triggered. This assumption was verified by our measurements over multiple WC buffer writes as well as PCIe trace analysis where 4 KB writes to WC buffers were measured and very little variability between PCIe write transactions were observed.

Since most network frames do not align on 64-byte boundaries, the remaining bytes are "stuffed" with an 8-byte end-of-packet (EOP) signature similar to an Ethernet frame EOP. This stuffing serves the purpose of reliably flushing the WC buffer. This EOP field is not part of the network packet outside the system and is only an overhead between CPU and I/O device. In our experimental setup, a special 64-bit code is used. There is some inefficiency due to this artificial stuffing of data, but the overhead is small when compared to the bandwidth inflation involved with descriptors and doorbells [12].

Fig. 5 illustrates what the TX FIFO queue would contain with four packets of different payload sizes. Packet A with 120-byte payload has a single 8-byte EOP alignment signature. Packet B with 64-byte payload requires an additional full 64-byte of EOP stuffing to notify the I/O adapter that it is only 64 bytes. This is because it needs to signal an EOP, but there is no space within a single 64-bit slot in the WC buffer. Packet C contains 240 bytes with two EOP alignment signatures.

Note that there is a potential of having a code generate a false EOP signal. However, the probability of this is extremely low (~$1/2^{64}$ or 5.4×10^{-20}), and when such an event occurs the false transmit payload will effectively be treated as a dropped packet. Thus, higher protocol levels will be relied on to recover the packet via re-transmit requests.

A core can quickly over-run the TX FIFO buffer on an I/O device interface, and thus a larger FIFO would be needed to account for increase in bandwidth-delay products beyond the requirements of a datacenter. To address this issue, we take advantage of each core in a typical system having between 6~10 WC buffers per core depending on the core architecture. Some memory tests have shown 2×~4× throughput improvement by pipelining writes across the available WC buffers using inline assembly instructions that

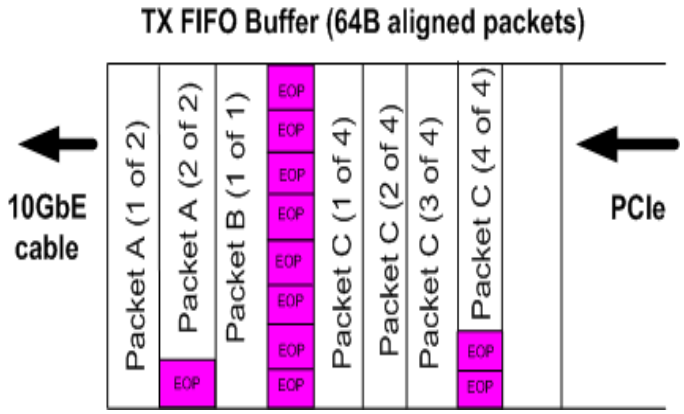

Figure 4.5: Hot-potato TX FIFO

bypass L1 and L2 lookups [10]. An example of such a code is shown below:

```
__asm__ __volatile__ (
  "  movntdq %%xmm0, (%0)\n"
  "  movntdq %%xmm0, 16(%0)\n"
  "  movntdq %%xmm0, 32(%0)\n"
  "  movntdq %%xmm0, 48(%0)\n"
  : : "r" (chimera_tx) : "memory");
  __asm__ __volatile__ (
  "  movntdq %%xmm1, (%0)\n"
  "  movntdq %%xmm1, 16(%0)\n"
  "  movntdq %%xmm1, 32(%0)\n"
  "  movntdq %%xmm1, 48(%0)\n"
  : : "r" (chimera_tx + 64) : "memory");
  chimera_tx+=64;
```

These SSE2 movntdq instructions write 16 bytes per instruction to the WC buffers so that each WC buffer can be filled with four instructions. The "nt" in the instruction defines a non-temporal move, so there are no cache lookups or coherency checks, and the "dq" specifies a double quad-word or 16 bytes.

By efficiently using the available WC buffers on each core, such as interleaving writes to different WC buffers as shown in the code above, there will be no throughput transmit bottleneck and throughput will track closely with available PCIe bandwidth.

4.4.2 I/O Receive operation

On the receive flow, the interface driver allocates memory upon device initialization (i.e., the driver is loaded into kernel space). This pins memory into a non-swappable region such that the RX DMA writes will always have a valid target address. This approach is similar to the descriptor-based DMA approach, which also pins memory – usually one page (4 KB) per packet. The descriptor-based DMA approach allows for a scatter-gather approach for receive traffic usually using a page per packet at the cost of memory fragmentation and pointer references. In contrast, our approach allocates a single receive memory block for all device receive traffic.

As part of the driver initialization, the base address of this buffer is programmed into the RX DMA engine as the space to place received packets. This address initialization eliminates the need for the per-packet receive descriptor that adds to the system bandwidth overhead. Thus, when an I/O adapter asynchronously receives a message in the RX FIFO queue, the receive data is written into the system memory space specified by the base address pointer. After the DMA transaction, the RX DMA engine increments the base address pointer by the size of the received packet so that the next Ethernet frame has the correct base address in system memory. The receive driver then parses the packet based on the header information such as the length field of an Ethernet frame. While there is a rare race condition between the NIC writing a partial packet and the driver reading the packet, the driver can be coded to retry the receive buffer after a short interval upon a failed Ethernet CRC value.

A benefit to this approach that the NIC interrupt logic is no longer needed to predict when to generate an interrupt. The CPU software algorithm can appropriately determine when to service incoming traffic. Furthermore, the network receive processing does not need to gather the packet page references, and instead simply looks for EOP frame

delimiters.

In our measurements, each RX DMA transaction is 64 bytes to match the I/O transmit writes. Larger PCIe DMA transactions could be used for receive I/O transactions, but this increase in bandwidth efficiency (due to PCIe frame overhead of 24B per PCIe frame) would need to be traded-off for latency improvement benefits.

In our implementation, a receive interrupt is generated after the DMA operation completes. As in most device drivers, the receive interrupt routine consists of an upper half [13] that pre-empts the currently scheduled OS task, which then calls the lower half of the driver to schedule more complex tasks (such as TCP protocol processing) to process the received packet.

4.4.3 I/O device and core mismatch issues

For transmit operations, there are two general implementation options. The simplest implementation is similar to I/O adapter accelerator functions, e.g., Large Receive Offloads (LRO) [14], which are enabled for all connections using the I/O adapter. In this case, all network connections are either descriptor DMA based or Hot-Potato generated transmission. To enable Hot-Potato transmits, a control bit in the I/O adapter would need to be set via an operating system command such as modprobe(). The second more complex implementation is to define each network connection to be either Hot-Potato or descriptor-based DMA.

In both cases, the I/O adapter hardware needs to support descriptor-based DMA and Hot-Potato. Although this adds hardware complexity relative to only implementing Hot-Potato, the risk of failure in the field is reduced as various operating systems need to adapt to support Hot-Potato. This could be done by having separate I/O buffers for Hot-Potato and descriptor-based DMA. This additional hardware complexity will consume some power, but is on the order of 10 mW based on our FPGA projections. Note that as soon as the transmit message leaves the I/O adapter, there is no distinction between a descriptor DMA generated transmit and a Hot-Potato generated transmit.

4.5 MEASUREMENTS AND ANALYSIS

Our baseline measurements are based on a 2.5 GHz Intel Sandy Bridge 4-core i5-2400S platform configured as shown in Fig 1. A Linux x64 kernel 2.6.35 is used with an Ubuntu distribution to support the custom network driver code. The proposed Hot-Potato-based I/O adapter is implemented using a PCIe-based Xilinx Virtex5 (XC5VTX240T) FPGA. The PCIe bandwidth is 8 Gbps simplex. Although PCIe Gen1 interface technology is used, the subsequent PCIe generations also follow the same protocol basically increasing lane speed and number of lanes. This allows extrapolation of our Gen1 data to the current Gen2 I/O devices, and future Gen3 devices. The measurements are taken using a combination of Lecroy PCIe analyzer tracing and internal FPGA logic tracing. These hardware measurements are strictly passive and do not induce any latency or performance overhead. The software micro-benchmarks of ICMP/IP ping and *iperf* are used for latency and bandwidth testing, respectively, to compare Hot-Potato versus the standard Ethernet. In addition, the latency sensitive *Memcached* [15] application, where a cluster of systems share memory references in key/value pairs, was analyzed to complement the latency and throughput micro-benchmarks.

Our test code is built on the example Ethernet driver code found in Linux Device Drivers [13], which loops back the subnet and IP addresses allowing experiments to be run without a real external network. This is done by instantiating two bi-directional Ethernet interfaces on two separate subnets. This allows us to isolate the system latencies for analysis without wire and fiber PHY latencies and their variations.

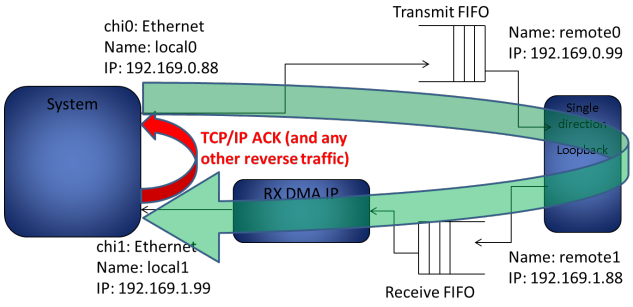

Figure 4.6: HW & SW co-utilization of interfaces

Note that since existing I/O devices prefetch and coalesce Ethernet frame descriptors, there is no significant latency improvement by having the Hot-Potato receive data stream into a system memory buffer. As a result, our latency savings appear only in the transmit path, so we focus our findings in the Hot-Potato transmit path.

Fig. 6 shows how the proposed Hot-Potato based I/O adapter implemented in FPGA is interfaced to the system. The FPGA only implements a single pair of transmit and receive interfaces since our interest is in how a single I/O adapter device works. This avoids having undesired PCIe traffic, such as TCP/IP ACK frames, and other multi-interface traffic that occurs over a single PCIe device to skew the experiment. Therefore, the only traffic on the PCIe interface, marked by the green arrow, is transmitted from the chi0 interface and received by the chi1 interface. The reverse traffic (from chi1 to chi0, and marked with a red arrow) occurs in memory as the original driver is coded to avoid irrelevant PCIe traffic in the analysis. This reverse traffic is needed to support higher-level network protocols such as TCP, which assumes ACK packets to ensure a reliable connection. Our driver sends traffic to the FPGA (via chi0), while the receive traffic (via chi1) is immediately looped back to the RX FIFO.

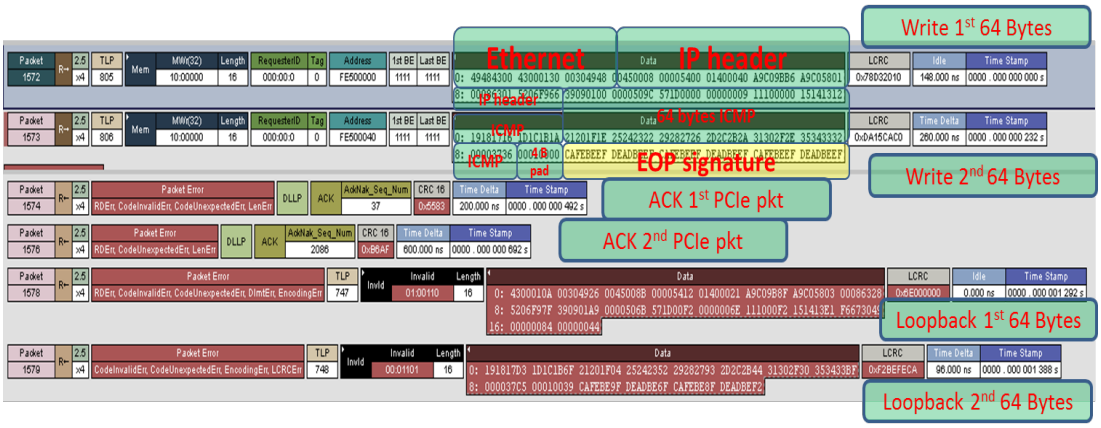

Figure 4.7: Hot-Potato ping loopback trace

For transmit CPU overhead, we performed similar iperf tests on a descriptor-based Intel 10GbE. Using the Linux *perf* performance tool, we observed that up to 2% of CPU overhead is due to transmit descriptor related overhead in the ixgbe_xmit_frame() function.

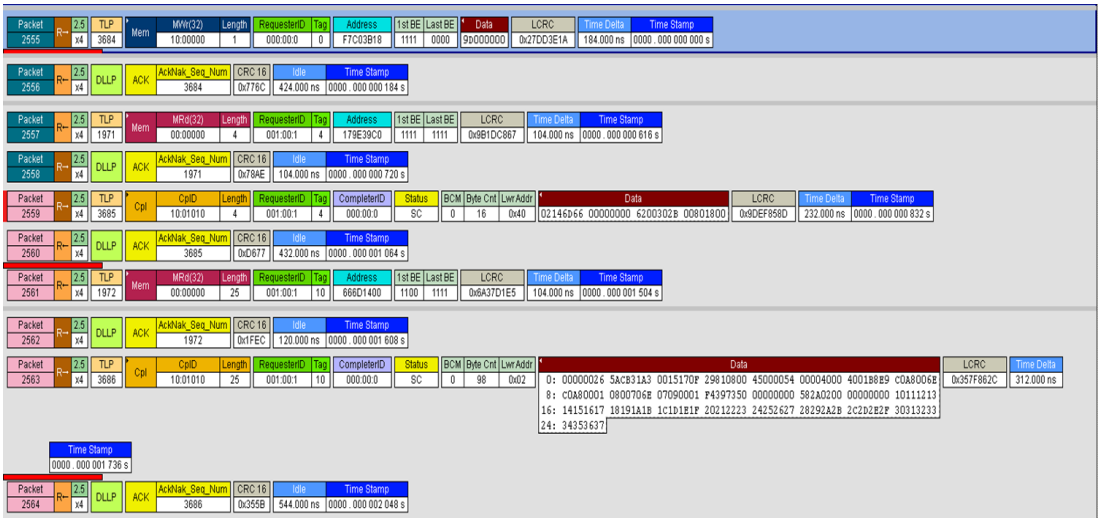

Figure 4.8: Intel 82575EB 1GbE ping transmit trace

Similar CPU savings is expected with Hot-Potato transmit CPU overhead for higher bandwidth traffic. If the PCIe bandwidth cannot sustain the CPU core throughput, meaning the 6˜10 WC buffers (384-640 bytes) are not drained, the transmit core may stall, which will also increase transmit CPU overhead. The stall risk is highly workload dependent and requires further explorations [12].

4.5.1 Micro-benchmark comparison

The micro-benchmark performance is evaluated by sending a single Ethernet ICMP ping packet, which consists of 64 bytes along with the required IP and Ethernet header information. The IP layer requires 24 bytes and the Ethernet frame requires 12 bytes for a total message size of 100 bytes. Since an 8-byte EOP signature is used, a packet needs to be aligned to 8 bytes. Therefore, four more bytes are needed for a total Hot-Potato payload of 104 bytes. The 104-byte payload requires three 8-byte EOP signatures to align the 64-byte ICMP message across two WC buffers.

Fig. 7 shows the Hot-Potato loopback trace where each PCIe packet is shown as

a separate line and enumerated in the field marked "Packet". The two PCIe write transactions for the ICMP message are indicated by PCIe packets #1572 and #1573. The temporal reference point at the beginning of packet #1572 is T_0. These two packets are acknowledged with PCIe packets #1574 and #1576 by the FPGA I/O device at T_0 + 492 ns and T_0 + 692, respectively. Note that there are CRC failures due to a PCIe analyzer failure, in the upstream PCIe frames, but the software verified that the expected loopback data was properly written into the pinned system memory buffers.

The Hot-Potato-based I/O device, upon sensing the EOP indicator, initiates the DMA write back to the system memory starting with packet #1578. This transaction is seen on the PCIe interface at T_0 + 1,292 ns. The second 64-byte PCIe packet containing EOP signatures is written to system memory with packet #1579 at time T_0 + 1,388 ns.

Fig. 8 shows the latency breakdown of a 64-byte IP packet using a standard 1GbE NIC [16, 17]. Since the measurement was between two different systems, Fig. 8 only shows the transmit operations to avoid comparing loopback versus system-to-system latencies. Again, T_0 is used as the initial observance of PCIe traffic in the transmit direction of the doorbell write in packet #2555, which is acknowledged at T_0 + 184 ns in packet #2556. The NIC responds with a transmit descriptor fetch request in packet #2557. The read request is completed with data in packet #2559 and acknowledged at T_0 +1,064 ns in packet #2560. After parsing the descriptor, the NIC requests the payload data in packet #2561, which completes in packet #2563 with data and is acknowledged by the NIC at T_0 + 2,048 ns.

Table 1: Latency breakdown comparison

Latency critical path for 64B message	Hot-Potato I/O Interface experimental setup (Fig.7)	Standard 1GbE Intel 82575EB (Fig. 8)
Doorbell to PCIe	0	T_0
Descriptor fetch	0	T_0 + 1,064 ns
Payload (DMA fetch or core write)	T_0 + 232 ns	T_0 + 1,736 ns
PCIe NIC to fiber	NA (equivalent)	NA (equivalent)
Fiber delay	NA (equivalent)	NA (equivalent)
Fiber to PCIe	NA (equivalent)	NA (equivalent)
PCIe to system memory	NA (DMA operations are similar)	NA (DMA operations are similar)
Total latency	**232 ns**	**1,736 ns**

Since both Hot-Potato and the 1GbE scenarios use DMA receive transactions, there is little latency difference in the I/O receive path. Table 1 compares the latencies in these two example traces for a 64-byte message between two systems. Table 1 shows that the Hot-Potato-based I/O reduces the latency in transmitting a message by (1,736 ns – 232 ns) = 1,504 ns.

Note that our proposed approach can reduce latency even further when the message is within a single WC buffer instead of the two WC buffers shown in Fig 7. Accounting for the header and EOP requirement, only a single WC buffer is needed if the message is less than 20 bytes, which is applicable in the financial trading market. Based on multiple back-to-back WC buffer writes, there is on average 108 ns delay between two consecutive WC buffer writes. Therefore, the minimum latency to send a message out of a system in our implementation is 108 ns. In contrast, the minimum 1GbE latency to send a message out of a system in a descriptor-based adapter, including the frame header, is 1,673 ns.

Fig. 9 shows DMA latency comparison using the WC buffers, which is bounded by the 8 Gbps PCIe bandwidth limitation of our test environment. The DMA transmit latency curve is smoother than the Hot-Potato curve since the latter uses 64-byte alignment, while the former uses byte-level alignment.

Fig. 10 compares throughput for the two methods, which shows that the proposed method outperforms descriptor-based DMA for small messages and the performance converges with descriptor-based DMA for large messages. The abrupt degradation for the Hot-Potato method is again due to the 64-byte WC buffer alignment.

Bandwidth utilization is another area where Hot-Potato provides improvement. This is because the descriptor traffic overhead is removed from both the PCIe interface as well as the memory interface. This benefit is difficult to quantify, but since I/O DMA transactions will contend with CPU core access to memory, there will be a benefit to overall system performance. While there is a small overhead in EOP traffic on the PCIe interface, this is embedded in the PCIe frames and is relatively small when compared with the descriptor-DMA overhead.

Our analysis of the *iperf* microbenchmark throughput results (sampled for $>$ 100 ms) on the PCIe interface using a dual 10GbE Intel 82599 NIC shows that the percentage of payload overhead for receive throughputs are taxed with 18% for small messages and 5% for larger messages. For transmit overhead, up to 43% of the traffic on the PCIe interface is used for descriptors or doorbells for small messages. The Hot-Potato approach removes this PCIe bandwidth overhead. This throughput overhead ratio is expected to be matched on the memory interface (with the exception of core generated doorbells), but was not directly measured.

4.5.2 Application benchmark comparison

Memcached is used by social networking sites such as Facebook, which in 2008 had 800 Memcached servers with up to 28 Terabytes of cached data [18]. *Memcache* has also been extended to support distributed file systems [19]. *Memcached* is based on a website service that either SETs a value or GETs a value. In our benchmark, the default size of 256 bytes of data was used. The system setup consists of 2.5 GHz Intel Sandy Bridge quad-core i5-2400S connected via 10GbE 82599 interfaces back-to-back, where one of them is setup as the Apache server with Linux kernel revision 3.6.6. The *Memcached* client runs a PHP-based script to generate the SET and GET requests of the server

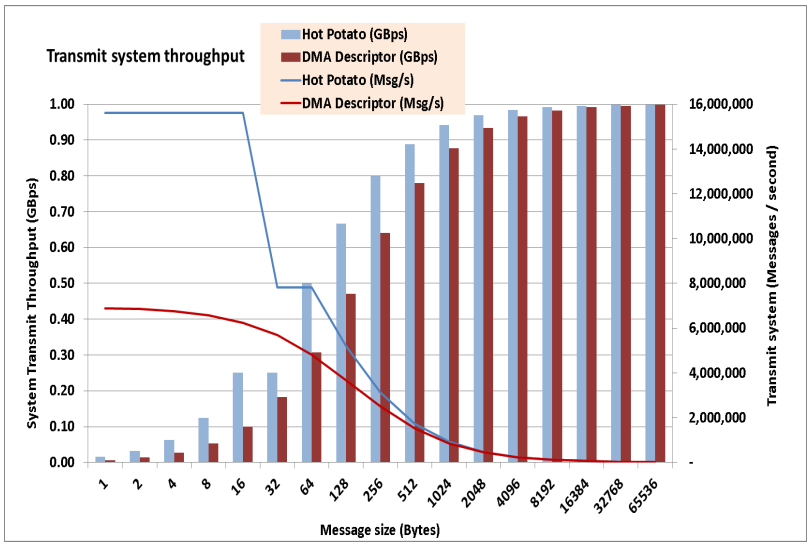

Figure 4.9: Throughput comparison

[20]. To ensure power states do not impede performance results, all cores were forced to operate in an active state so that no CPU resources would enter into a sleep mode, and NIC interrupt throttling is disabled.

The default configuration has 90% of transactions as GETs with the remainder being SETs. The baseline measurement showed average GET latency of 47 µs with 16,900 GET operations per second.

To understand the sensitivity of *Memcached* transactions to latency, the open-source 10GbE driver was modified to incur a controlled latency when the transmit doorbell is issued. The transmit doorbell was selected since Hot-Potato bypasses doorbell/descriptor operations. This latency was set and tested over ten additional latencies ranging from 1 μs to 1 ms. The result shows a linear decrease in GET latency as the doorbell latency decreases resulting in higher GET operations per second. For every 1us device driver latency increase, we saw an average 1.2us increased GET latency.

Since in our two system *memcached* test configuration, each Ethernet device could be implemented with Hot-Potato, we calculate that the 1.5us latency savings on each

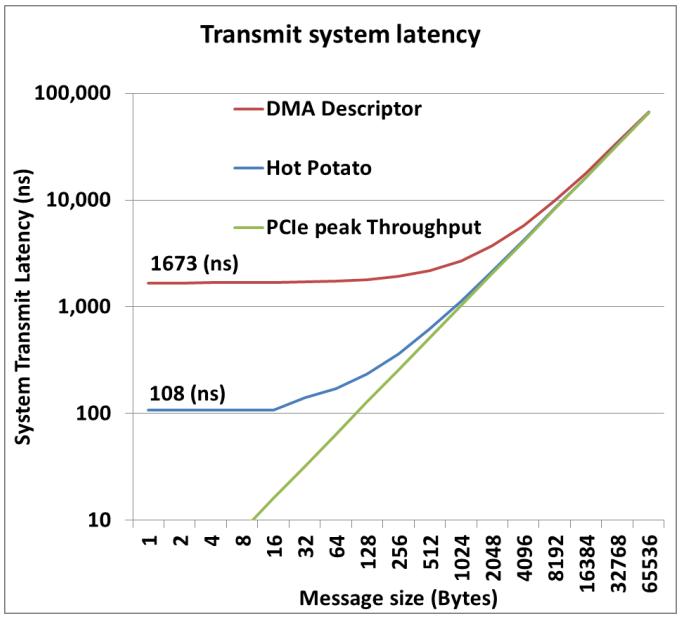

Figure 4.10: Transmit latency comparison

Hot-Potato device results in a total of 3.0us for our test environment. Factoring the linear relationship to memcached latency impact, by saving 3.0us, we expect a reduction of 3.6us to the baseline 47us GET latency discussed above. This results in 7.7% overall performance improvement.

Memcached transactions were not implemented directly using the Hot-Potato FPGA due to technical issues interfacing to the 10GbE port. The simple GET and SET transactions to manage large database queries in memory (as opposed to slower disk-based) allow the correlation that saving 3.6us in a *memcached* query results in a 7.7% *memcached* performance improvement. Our future work will include more complex multi-system memcached scenarios.

4.6 CONCLUSION AND FUTURE WORK

This paper proposed the Hot-Potato I/O mechanism, which leverages existing methods of WC buffer access to memory and direct CPU access to I/O devices. Our results show that Hot-Potato provides significant latency improvement (8% latency reduction in a *Memcached* application configuration) on current systems. Low latency small-messages would be beneficial in clustering applications, such as *Memcached*, as well as in the financial market where a microsecond can garner millions [21]. Although, some changes are required in both the hardware implementation and software driver interface, they are mainly a simplification of the current descriptor-based distributed DMA engine approach.

Other less quantifiable benefits of the Hot-Potato approach include a core directly controlling the I/O transmit transactions to allow system power algorithms involving the on-die Power Control Unit (PCU) [22] to react more effectively than sending slow control messages to a PCIe attached I/O DMA engine. I/O transaction Quality-of-Service (QoS) also improves since a core can control (or filter) I/O transactions based on priority. In addition, system memory bandwidth utilization and memory latency improve by not having I/O DMA transactions between multiple I/O devices contending with core related memory transactions.

As future work, we plan to explore other benefits and tradeoffs that can improve the performance of the Hot-Potato approach. Improvements in WC buffers, such as increased size and control, would benefit I/O in general. In addition, this Hot-Potato approach to moving data can be used in PCIe fabric interconnects where instead of the Ethernet protocol (or other network protocol) the FIFO structures directly communicate across a PCIe switch fabric.

4.7 ACKNOWLEDGMENTS

The authors would like to thanks Guandeng Liao, Jesse Brandeburg, and Steve McGowan at Intel for providing helpful comments and clarifications. This research was

supported in part by Ministry of Education Science and Technology (MEST) and the Korean Federation of Science and Technology Societies (KOFST).

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

Steen K Larsen, Ben Lee

Submitted to : 10th IEEE International Conference on Networking, Architecture, and Storage
http://www.nas-conference.org/

Chapter 5: D2D manuscript

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

Steen Larsen

Intel Corp

22238 NW Beck

Portland OR 97231

(503) 846-1937

steen.knud.larsen@gmail.com

Ben Lee

School of Electrical Engineering and Computer Science

Oregon State University

1148 Kelley Engineering Center

Corvallis, OR 97331-5501, USA

(541) 737-3148

benl@eecs.orst.edu

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.

PCIe; I/O device; system architecture

5.1 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. 5.1 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 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.

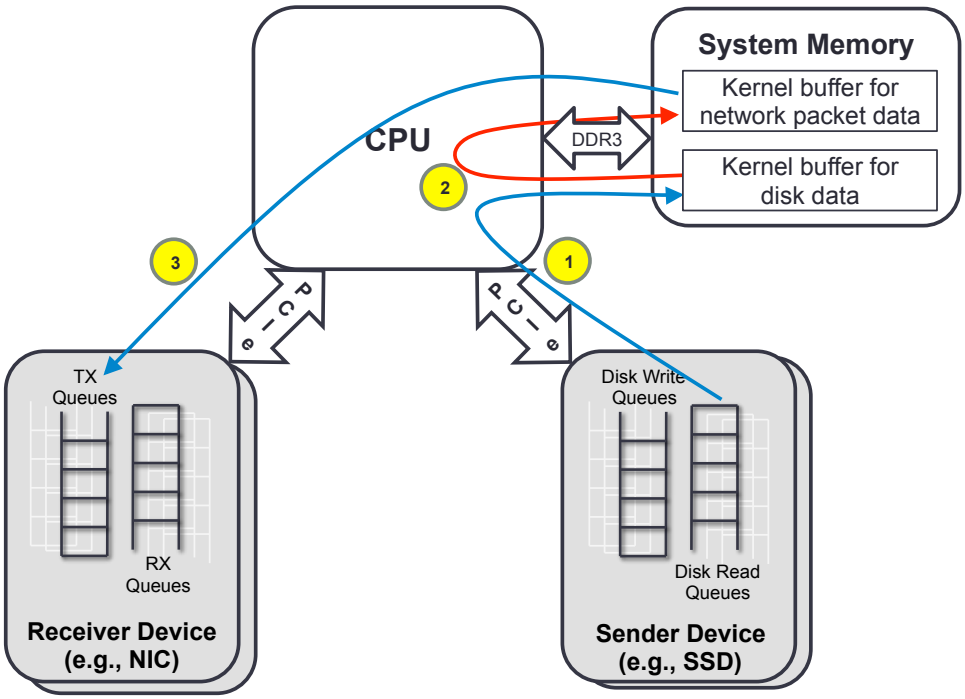

Figure 5.1: Conventional I/O Streaming

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.

5.2 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.

5.3 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. 5.2 and 5.3 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. 5.4).

The three major components required for D2D transfer from SSD to NIC are (1) *D2D Stream Control Registers* in 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. 5.3.4 discusses how the NIC and SSD can be configured to support the reverse stream, i.e., NIC to SSD.

5.3.1 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 5.1 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

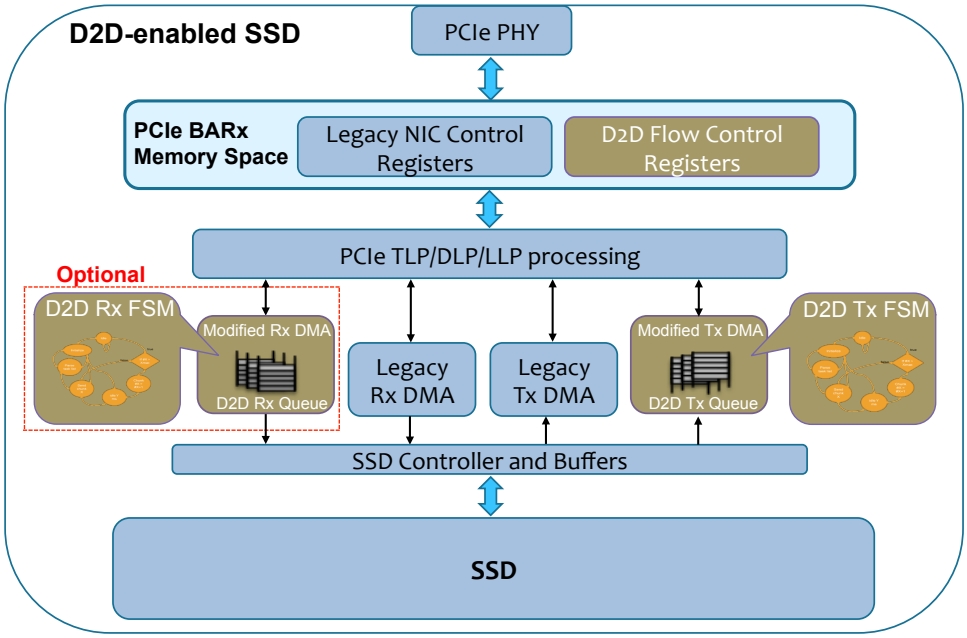

Figure 5.2: D2D-enabled SSD.

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 5.1: D2D Stream Control Register Definitions.

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

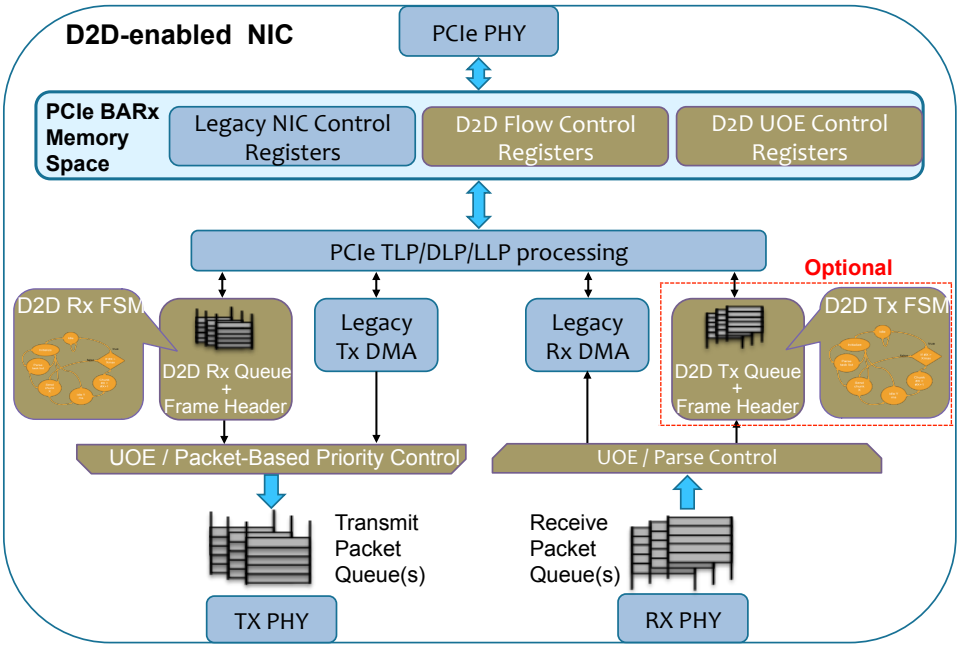

Figure 5.3: D2D-enabled NIC.

memory 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.

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. 5.3.2 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.

5.3.2 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. 5.4. 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

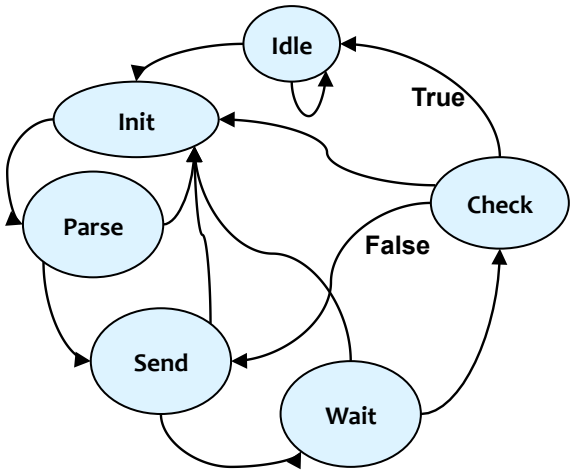

Figure 5.4: D2D Tx FSM.

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.

5.3.3 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. 5.5. 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 are defined in Table 5.2. 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*

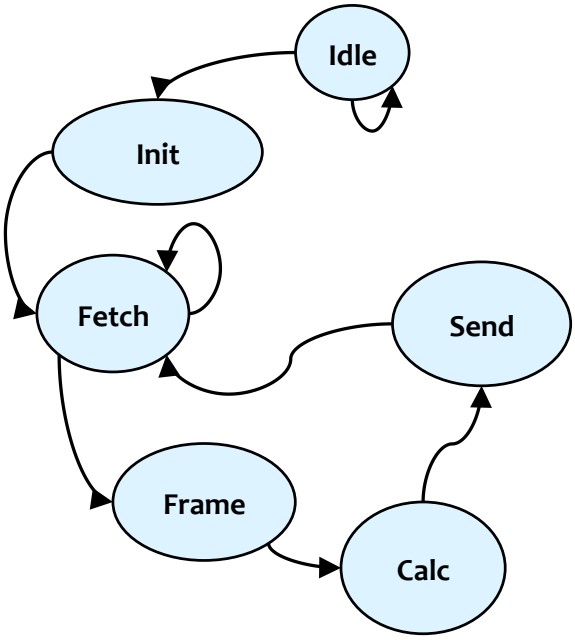

Figure 5.5: D2D Rx FSM.

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

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 5.2: D2D UOE Control Registers.

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 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 5.2. This packet is then sent to the NIC PHY for transmission.

5.3.4 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 Table 5.2. 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.

5.4 Proof-of-Concept Evaluation

This section discusses the implementation and evaluation the proposed D2D transfer protocol discussed in Sec. 5.3.

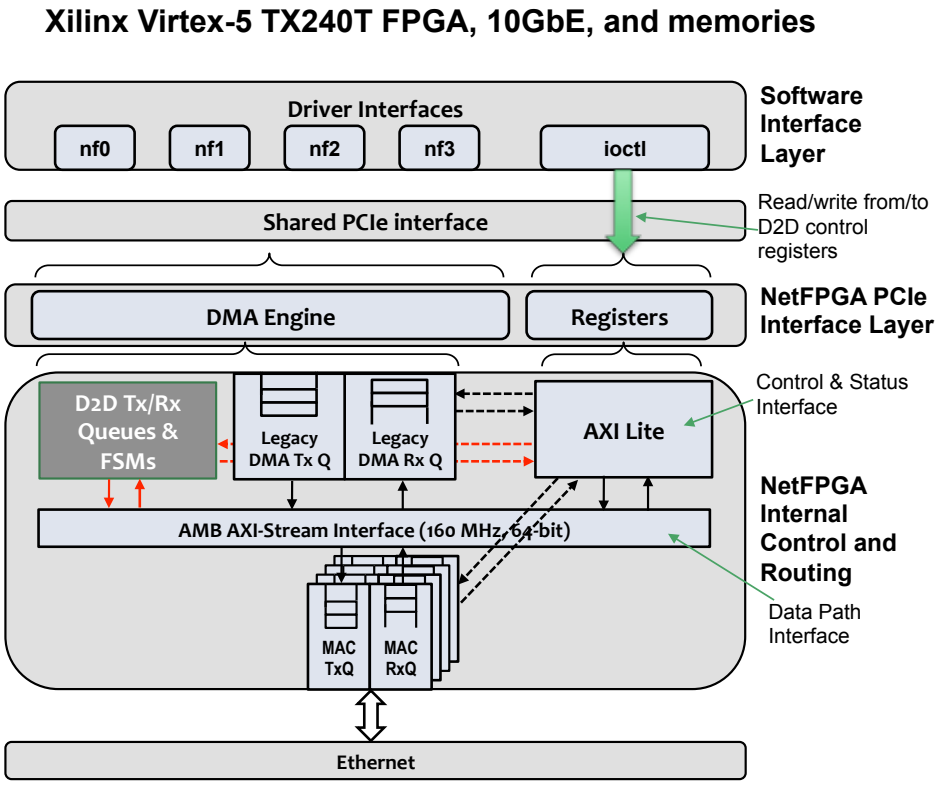

Figure 5.6: NetFPGA design hierarchy.

5.4.1 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. 5.6. 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 5.1 and 5.2 using the common `ioctl` interface. *AXI Lite* is the interface for the microcontroller 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. 5.6 is left unchanged.

5.4.2 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. 5.7. 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 SUT responds by requesting the UDP video stream from the SS using a TCP connection. The SS then responds by running a VLC

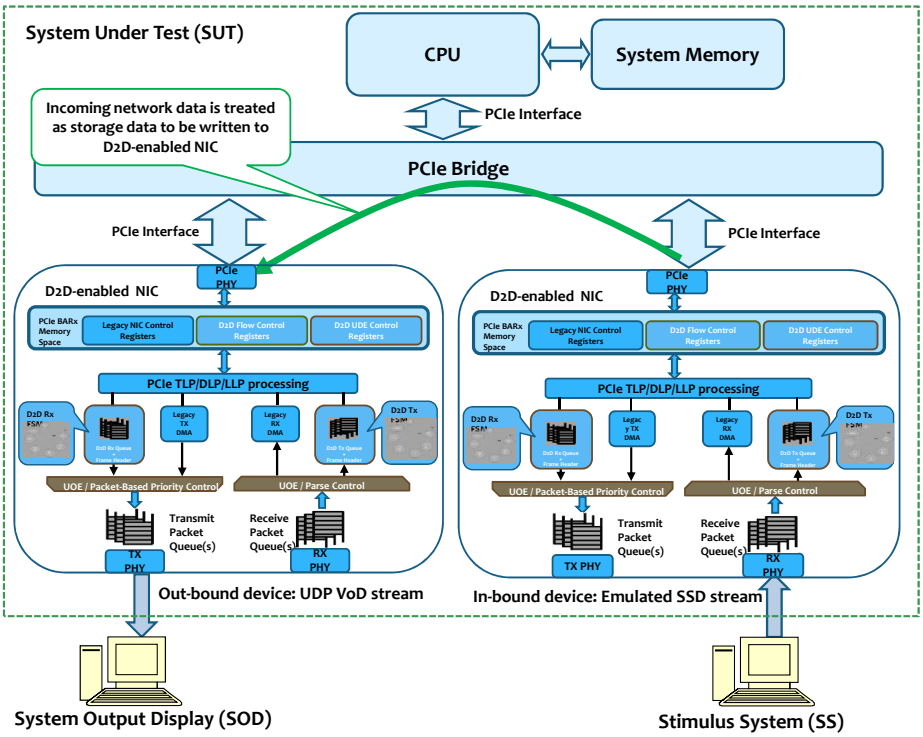

Figure 5.7: D2D prototype with emulated SSD.

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. 5.7, 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. 5.3.2) and the D2D Stream Control Register parameters (see Sec. 5.3.1). 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 5.1). 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 5.2). 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.

5.4.3 Performance Evaluation and Results

5.4.3.1 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. 5.8 shows the various latency components based on the measurements obtained using Xilinx Chipscope.

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.

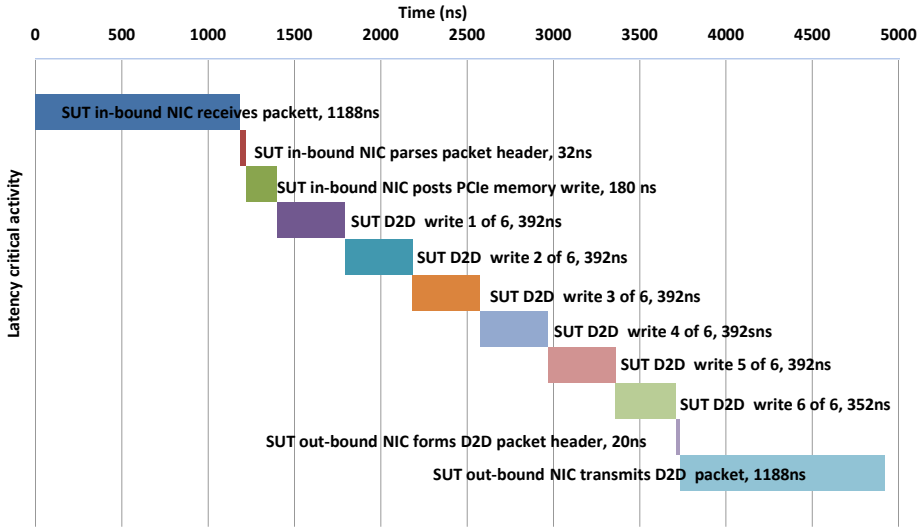

Figure 5.8: Example 1500 byte packet latency in the D2D prototype

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 memory write is a fragment, and thus 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. Summing the latency from the ingress of first byte of the packet into the SUT to the egress of the last byte of the packet from the SUT, the total latency is 4,959 ns to transfer across both PCIe devices in our prototype. 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, only one PCIe write packet is needed, reducing 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.

5.4.3.2 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. 5.5) causing video packets to be dropped.

The resulting measurements are shown in Fig. 5.9. In the graph, the red bars shows the data throughput received by the SOD, while 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. 5.5). 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.

5.4.3.3 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.

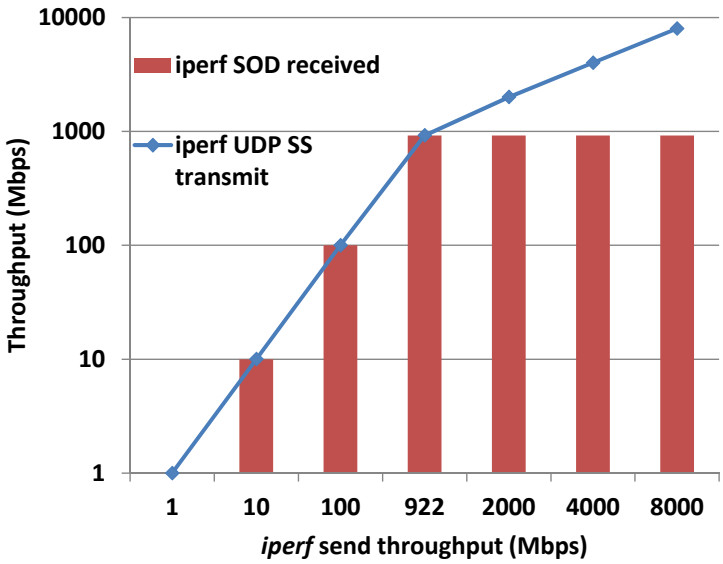

Figure 5.9: D2D *iperf* UDP throughput measurements

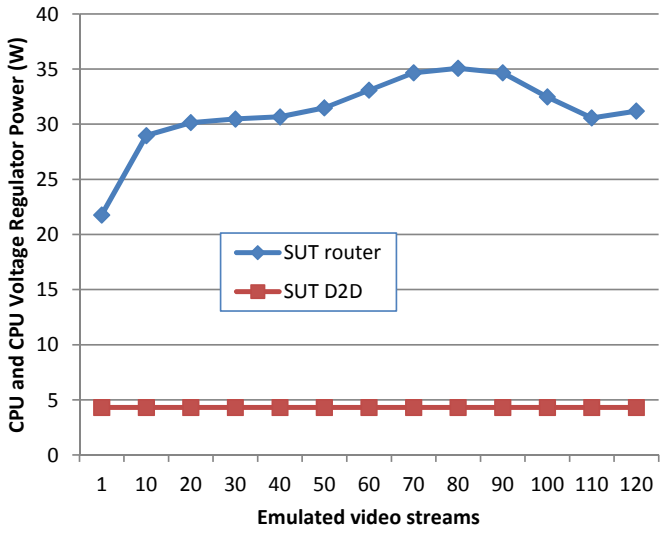

Figure 5.10: D2D CPU power benefit

Fig. 5.10 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. Testing the SUT router beyond the D2D limitations of up to 9.8 Gbps shows a similar trend of about 32 W for CPU power, and D2D power would expect to continue the sub-5 W power trend. Note that the average CPU power dissipation is only 4.32 W when no network traffic is being processed by the CPU.

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. 5.10 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

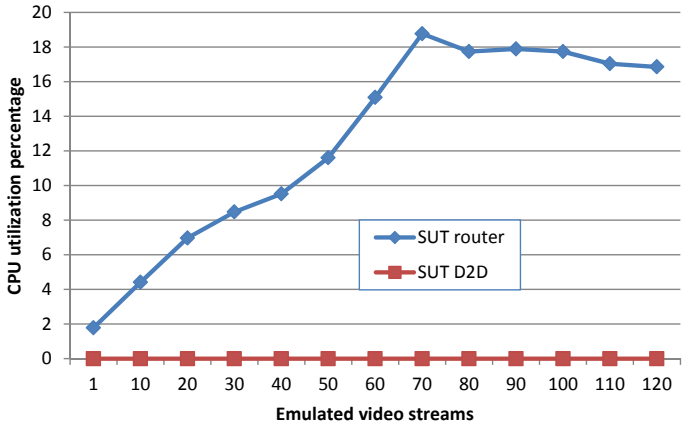

Figure 5.11: D2D CPU utilization benefit

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.

5.4.3.4 Utilization

Fig. 5.11 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.

5.5 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. 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 922 Mbps peak 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, such as the time required to generate and validate PCIe packet CRC32 values.
- 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.
- While there are reports of up to 1.5 Gbps throughput from the NetFPGA board [14], the observed maximum throughput of the D2D prototype with two NetFPGA cards was lower. Our Xilinx Chipscope debug traces shows that there may be a clock timing issue (i.e., a setup-hold violation within the FPGA logic) since only one or two bytes are sometimes corrupted on the 64-bit data interface at data rates higher than 922 Mbps. Since this issue is not periodic, or seen on the entire 64-bit data interface we do not suspect the D2D logic implementation.

5.6 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 services 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 support more complex networking scenarios.

Chapter 6: General Conclusion

In conclusion, it is clear there is no simple and elegant method to significantly improve I/O transactions without imposing hardware and software changes in how the computer system handles I/O data movement. Generally, the three proposals that have been made (iDMA, Hot-Potato, and D2D) have strengths in certain areas, but do not demonstrate compelling benefits across the range of applications in general purpose systems. As a result, it becomes an engineering tradeoff between cost vs benefit-gained. The cost in this case is the hardware and software requirements needed to implement the proposals in the majority of systems such that by default CPUs and I/O devices utilize one or all of the proposals to improve system performance.

iDMA shows there can be significant benefits to removing CPU-based descriptor processing across the PCIe links in terms of latency (reduced up to 16%) and bandwidth per pin (increased up to 17%). Current typical server CPUs have small controllers that are not directly OS-controlled (i.e. power control unit and security monitor control unit), so adding an iDMA control unit to manage descriptors is not without reason in the future.

Hot-Potato does not require a CPU hardware modification, and only small changes to the I/O device to bypass the current I/O device TX DMA and modify the RX DMA engine. This allows latency reduction (reduced 1.5µs) and reduction in OS noise.

As systems increase in complexity, more I/O devices need to move data between devices. In cases such as VoD, the CPU can setup a D2D flow enabling SSDs to communicate directly to NICs or any other generic I/O device. This is performed with CPU cycles required only at starting or interrupting the D2D flow, and not requiring CPU involvement for every smaller descriptor.

While the next few years will likely not see implementations of these proposals, we can make some general speculative extrapolations.

Assuming CPUs stay at a relatively fixed frequency multiplying core count and integrating further features, a variation on iDMA would make a lot of sense. Even if it means sequestering a core in the CPU for IO transactions, the rest of the CPU can execute without the OS noise that is an issue with HPC systems. As in HPC systems, the offloading of I/O transactions using I/O forwarding to an I/O processor would then be done by a core within the CPU.

As CPUs integrate more functions, including I/O devices such as NICs and storage interfaces, there is less need for descriptors since the I/O device is on-die instead of inches of lossy PCB traces away. In these integrated scenarios, a variation on Hot-Potato would be feasible.

As Internet traffic increases worldwide, much of the content is streamed as video or served as webpages between server systems and client systems. With the CPU becoming mainly a traffic management agent in these I/O streams, D2D becomes more appealing with its scalability and other benefits. For certain applications a power-hungry expensive Intel Xeon core could be replaced with a small Atom or ARM CPU.

These observations on the trajectory of computer system design indicate that this research of I/O transaction optimizations will continue to develop over several years. The common thread of the three proposals has been to address issues in current CPU-centric descriptor-based I/O transactions. Just as there are three proposals, there is likely a variety of final solutions that will be implemented in improving I/O transactions in the foreseeable future.

Bibliography

Chapter 1 : Introduction

[1] 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. CLUSTER, 2009

[2] Larsen, S. and B. Lee, *Platform IO DMA Transaction Acceleration.* International

Conference on Supercomputing (ICS) Workshop on Characterizing Applications for Heterogeneous Exascale Systems (CACHES), 2011. Available at: http://www.mcs.anl.gov/events/workshops/ashes/2011/program.html

[3] Steen Larsen and Ben Lee Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors https://www.elsevier.com/books/advances-in-computers/hurson/978-0-12-420232-0

[4] NetFPGA. Available at http://netfpga.org

Chapter 2 : iDMA manuscript

[1] Softmodem description, [cited 2010]; Available from: http://en.wikipedia.org/wiki/Softmodem.

[2] I/O Bottlenecks: Biggest Threat to Data Storage, [cited 2010]; Available from: http: //www.enterprisestorageforum.com/technology/features/article.php/3856121

[3] Fast Sort on CPUs, GPUs, and Intel MIC Architectures Available from: http: //techresearch.intel.com/spaw2/uploads/files/FASTsort_CPUsGPUs_IntelMICarchitectures.pdf.

[4] Intel 5520 chip-set datasheet, [cited 2010]; Available from: http://www.intel.com/assets/pdf/datasheet/321328.pdf.

[5] HypterTransport IO Link Specification Rev3.10, [cited 2010]; Available from: http://www.hypertransport.org/docs/twgdocs/HTC20051222-00046-0028.pdf.

[6] Leon, E.A., Ferreira, K.B., and Maccabe, A.B. 2007. Reducing the Impact of the MemoryWall for I/O Using Cache Injection. In High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on.22-24 Aug. 2007.

[7] Dan, T., Yungang, B., Weiwu, H., and Mingyu, C. 2010. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on.9-14 Jan. 2010.

[8] Intel 82599 10GbE NIC; Available from:
http://download.intel.com/design/network/prodbrf/321731.pdf.

[9] Introduction to Receive Side Scaling, [cited 2010]; Available from:
http://msdn.microsoft.com/en-us/library/ff556942.aspx.

[10] IOAT description; Available from:
http://www.intel.com/network/connectivity/vtc_ioat.htm.

[11] Banks and Outsourcing: Just Say 'Latency'; Available from:
http://www.hpcwire.com/features/Banks_and_Outsourcing_Just_Say_Latency_HPCwire.html.

[12] Intel 82598 10GbE NIC; Available from:
http://www.intel.com/assets/pdf/prodbrief/317796.pdf.

[13] Larsen, S., Sarangam, P., and Huggahalli, R. 2007. Architectural Breakdown of End-to-End Latency in a TCP/IP Network. In Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on.24-27 Oct. 2007.

[14] PCIe Base 2.1 Specification; Available from:
http://www.pcisig.com/specifications/pciexpress/base2/.

[15] Guangdeng, L. and Bhuyan, L. 2009. Performance Measurement of an Integrated NIC Architecture with 10GbE. In High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on.25-27 Aug. 2009.

[16] Schlansker, M., Chitlur, N., Oertli, E., Stillwell, P.M., Rankin, L., Bradford, D., Carter, R.J., Mudigonda, J., Binkert, N., and Jouppi, N.P. 2007. High-performance ethernet-based communications for future multi-core processors. In Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on.10-16 Nov. 2007.

[17] Intel QuickPath Interconnect; Available from:
http://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect.

[18] [cited 2010]; Available from: http://www.intel.com/go/lightpeak.

Chapter 3: I/O survey manuscript

1. Softmodem description. Available from: http://en.wikipedia.org/wiki/Softmodem.

2. Newman, H. I/O Bottlenecks: Biggest Threat to Data Storage. Available from: http://www.enterprisestorageforum.com/technology/features/article.php/3856121

3. Guangdeng, L. and L. Bhuyan. Performance Measurement of an Integrated NIC Architecture with 10GbE. in High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on. 2009.

4. Schlansker, M., et al. High-performance ethernet-based communications for future multi-core processors. in Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on. 2007.

5. Intel QuickPath Interconnect. Available from: http://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect.

6. Guangdeng, L., Z. Xia, and L. Bnuyan. A new server I/O architecture for high speed networks. in High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on. 2011.

7. Intel 5520 chip-set datasheet. 2010; Available from: http://www.intel.com/assets/pdf/datasheet/321328.pdf.

8. Ionkov, L., A. Nyrhinen, and A. Mirtchovski, CellFS: Taking the "DMA: out of Cell programming, in Proceedings of the 2009 IEEE International Symposium on Parallel\&Distributed Processing. 2009, IEEE Computer Society. p. 1-8.

9. Khunjush, F. and N. Dimopoulos. Extended characterization of DMA transfers on the Cell BE processor. in IEEE International Symposium on Parallel and Distributed Processing (IPDPS). 2008.

10. Dittia, Z.D., G.M. Parulkar, and J.R. Cox, Jr. The APIC approach to high performance network interface design: protected DMA and other techniques. in INFOCOM

'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE. 1997.

11. Yuan, L., H. Li, and C. Duan. The Design and Implementation of MPI Based on Link DMA. in Embedded Computing, 2008. SEC '08. Fifth IEEE International Symposium on. 2008.

12. Using DMA effectively. Available from: http://www.embedded.com/columns/technicalinsights/196802092fire-questid=228429

13. IOAT performance. Available from: http://www.linuxfoundation.org/collaborate/workgroups/networking/i/oat.

14. Yu, P., et al. A High Speed DMA Transaction Method for PCI Express Devices Testing and Diagnosis. in IEEE Circuits and Systems International Conference on ICTD. 2009.

15. PCIe Base 2.1 Specification. Available from: http://www.pcisig.com/specifications/pciexpress/base2/.

16. Tumeo, A., et al. Lightweight DMA management mechanisms for multiprocessors on FPGA. in Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on. 2008.

17. Salah, K. and K. El-Badawi. Throughput-delay analysis of interrupt driven kernels with DMA enabled and disabled in high-speed networks. in Journal of High Speed Networks. 2006.

18. Hahn, J., et al., Analysis of Worst Case DMA Response Time in a Fixed-Priority Bus Arbitration Protocol. Real-Time Syst., 2002. **23**(3): p. 209-238.

19. Huang, T.-Y., C.-C. Chou, and P.-Y. Chen, Bounding the Execution Times of DMA I/O Tasks on Hard-Real-Time Embedded Systems, in Real-Time and Embedded Computing Systems and Applications, J. Chen and S. Hong, Editors. 2004, Springer Berlin / Heidelberg. p. 499-512.

20. Huang, T.-Y., J.W.-S. Liu, and D. Hull, A Method for Bounding the Effect of DMA I/O Interference on Program Execution Time, in Proceedings of the 17th IEEE Real-Time Systems Symposium. 1996, IEEE Computer Society. p. 275.

21. Pitter, C. and M. Schoeberl. Time Predictable CPU and DMA Shared Memory Access. in Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on. 2007.

22. Rogers, B., et al. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. in ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture. 2009.

23. Bardsley, A. and D. Edwards, Synthesising an asynchronous DMA controller with balsa. J. Syst. Archit., 2000. **46**(14): p. 1309-1319.

24. Aghdasi, F. and A. Bhasin. DMA controller design using self-clocked methodology. in AFRICON, 2004. 7th AFRICON Conference in Africa. 2004.

25. Markatos, E. and M. Katevenis. User-level DMA without operating system kernel modification. in Third International Symposium on HighPerformance Computer Architecture. 1997.

26. Blumrich, M.A., et al. Protected, userlevel DMA for the SHRIMP network interface. in Second International Symposium on High-Performance Computer Architecture Proceedings. 1996.

27. Larsen, S., et al., Architectural Breakdown of End-to-End Latency in a TCP/IP Network. International Journal of Parallel Programming, 2009. **37**(6): p. 556-571.

28. Intel 82599 10GbE NIC. Available from: http://download.intel.com/design/network/prodbrf/321731.pdf.

29. Corporation, M. Introduction to Receive Side Scaling. 2012 [cited June 2012; Available from: http://msdn.microsoft.com/en-us/library/ff556942.aspx.

30. Hatori, T. and H. Oi, Implementation and Analysis of Large Receive Offload in

a Virtualized System. Proceedings of the Virtualization Performance: Analysis, Characterization, and Tools (VPACT'08), 2008.

31. Regnier, G., et al., TCP onloading for data center servers. Computer, 2004. **37**(11): p. 48-58.

32. Ross, M., et al. FX1000: a high performance single chip Gigabit Ethernet NIC. 1997: IEEE.

33. TCP Offload Engine. Available from: http://en.wikipedia.org/wiki/TCP_Offioad_Engine

34. Bhoedjang, R.A.F., T. Ruhl, and H.E. Bal, User-level network interface protocols. Computer, 1998. **31**(11): p. 53-60.

35. Hill, M.D., et al. Coherent network interfaces for fine-grain communication. 1996: IEEE.

36. Chitlur, N., S. Larsen, Editor. 2011.

37. Dan, T., et al. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. in High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on. 2010.

38. HypterTransport IO Link Specification Rev3.10. 2008; Available from: http://www.hypertransport.org/docs/twgdocs/HTC20051222-00046-0028.pdf.

39. MESI protocol. Available from: http://en.wikipedia.org/wiki/MESI_protocol

40. Mellor-Crummey, J. Intel Single Chip Cloud Computer. 2010; Available from: http://www.cs.rice.edu/~johnmc/comp522/lecture-notes/COMP522-2010-Lecture5-SCC.pdf.

41. Infiniband overview. Available from: http://en.wikipedia.org/wiki/InfiniBand.

42. Subramoni, H., M. Koop, and D.K. Panda. Designing Next Generation Clusters:

Evaluation of InfiniBand DDR/QDR on Intel Computing Platforms. in High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on. 2009.

43. Godiwala, N., J. Leonard, and M. Reilly. A Network Fabric for Scalable Multiprocessor Systems. in High Performance Interconnects, 2008. HOTI '08. 16th IEEE Symposium on. 2008.

44. Harris, D. Banks and Outsourcing: Just Say 'Latency'. 2008; Available from: http://www.hpcwire.com/features/Banks_and_Outsourcing_Just_Say_Latency_HPCwire.html.

45. Intel 82598 10GbE NIC. Available from: http://www.intel.com/assets/pdf/prodbrief/317796.pdf.

46. Larsen, S. and B. Lee, Platform IO DMA Transaction Acceleration. 2011.

Chapter 4: Hot-Potato manuscript

1. *Softmodem description*. Available from: http://en.wikipedia.org/wiki/Softmodem.

2. Top500. 2012; Available from: http://i.top500.org/stats.

3. Markatos, E. and M. Katevenis. *User-level DMA without operating system kernel modification*. in *Third International Symposium on HighPerformance Computer Architecture*. 1997.

4. Blumrich, M.A., et al. *Protected, userlevel DMA for the SHRIMP network interface*. in *Second International Symposium on High-Performance Computer Architecture Proceedings*. 1996.

5. Wikipedia. *Write-combining*. 2012; Available from: http://en.wikipedia.org/wiki/Write-combining.

6. Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2012.

7. Bhoedjang, R.A.F., T. Ruhl, and H.E. Bal, *User-level network interface protocols*. Computer, 1998. **31**(11): p. 53-60.

8. Litz, H., et al. *VELO: A novel communication engine for ultra-low latency message transfers.* 2008: IEEE.

9. Corporation, M. *Introduction to Receive Side Scaling.* 2012 [cited June 2012; Available from: http://msdn.microsoft.com/en-us/library/ff556942.aspx.

10. Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Vol1-3.* 2013.

11. Milewski, B. *Memory fences on x86.* 2008; Available from: http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/.

12. Larsen, S. and B. Lee, *Platform IO DMA Transaction Acceleration.* International Conference on Supercomputing (ICS) Workshop on Characterizing Applications for Heterogeneous Exascale Systems (CACHES), 2011.

13. Jonathan Corbet, A.R., Greg Kroah-Hartman, *Linux Device Drivers 3rd Edition.* 2005.

14. Hatori, T. and H. Oi, *Implementation and Analysis of Large Receive Offload in a Virtualized System.* Proceedings of the Virtualization Performance: Analysis, Characterization, and Tools (VPACT'08), 2008.

15. *Memcached.* 2012; Available from: http://memcached.org/.

16. *Intel 82599 10GbE NIC.* Available from: http://download.intel.com/design/network/prodbrf/321731.pdf.

17. Larsen, S., et al., *Architectural Breakdown of End-to-End Latency in a TCP/IP Network.* International Journal of Parallel Programming, 2009. **37**(6): p. 556-571.

18. Saab, P. *Scaling memcached at Facebook.* 2008; Available from: http://www.facebook.com/note.php?note_id=39391378919.

19. *memcachefs: a memcache filesystem using FUSE.* Available from: http://memcachefs.sourceforge.net/.

20. *Brutis for memcached.* 2010; Available from: http://code.google.com/p/

brutis/.

21. Goldstein, J. *The Million Dollar Microsecond.* 2010; Available from: http://www.npr.org/blogs/money/2010/06/08/127563433/the-tuesday-podcast-the-million-dollar-mi

22. Wikipedia, *Haswell PCU.* 2012.

Chapter 5: D2D manuscript

Bibliography

[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)