Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors

Citation	Larsen, S., & Lee, B. (2014). Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors. Advances in Computers, 92, 67-104. doi:10.1016/B978-0-12-420232-0.00002-7
DOI	10.1016/B978-0-12-420232-0.00002-7
Publisher	Elsevier
Version	Accepted Manuscript
Terms of Use	http://cdss.library.oregonstate.edu/sa-termsofuse

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
Corvallis, OR 97331
steen.knud.larsen@gmail.com, benl@eecs.orst.edu

[‡]Intel Corporation
124 NE Shannon St
Hillsboro OR 97124

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, descriptor-based DMA transactions have become more important in high performance systems to move data between I/O adapters and system memory buffers. DMA transactions are done with hardware engines below the software protocol abstraction layers in all systems other than rudimentary embedded controllers. CPUs can switch to other tasks by offloading hardware DMA transfers to the I/O adapters. Each I/O interface has one or more separately instantiated descriptor-based DMA engines optimized for a given I/O port. I/O transactions are optimized by accelerator functions to reduce latency, improve throughput and reduce CPU overhead. This chapter surveys the current state of high-performance I/O architecture advances and explores benefits and limitations. With the proliferation of CPU multi-cores within a system, multi-GB/s ports, and on-die integration of system functions, changes beyond the techniques surveyed may be needed for optimal I/O architecture performance.

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

I. INTRODUCTION

I/O is becoming a peer to processor core (or simply *core*) and memory in terms of latency, bandwidth, and power requirements. Historically, 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 the core/memory bandwidth, it makes sense to move data at a pace governed by the external device.

Typically, I/O 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 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 and Simultaneous Multi-Threading (SMTs), the I/O device capability can be scaled as the number of cores scale per Central Processing Unit (CPU). A CPU can consist of multiple cores and other related functions, but is unified on a single silicon die. Figure 1 shows how CPU scaling and the integration of the memory controller have exceeded I/O bandwidth gains during the period from 2004 to 2010 [2]. As can be seen, I/O bandwidth has improved at a much lower rate than CPU and memory performance capabilities. I/O also needs quality of service to provide low latency for network interfaces and graphics accelerators, and high bandwidth support for storage interfaces.

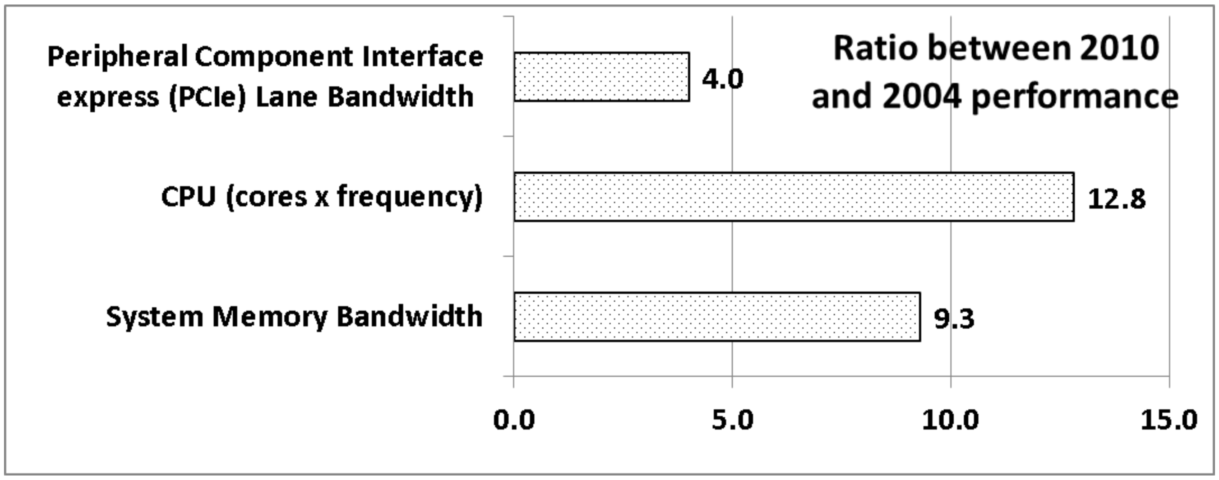

Figure 1: I/O Performance increase comparison (adapted from [2]).

The movement of data between CPUs and I/O devices is performed using variety methods, each often optimized based on the traffic type. For example, I/O devices such as storage disk drives typically move large blocks of data (> 4 Kilobyte) for throughput efficiency but result in poor latency performance. In contrast, low latency is crucial in a scenario, such as a cluster of inter-networked systems, where messages may be small (on the order of 64 bytes). Therefore, this paper provides a survey on existing methods and advances in utilizing the I/O performance available in current systems. Based on our measurements and analysis, we also show how I/O is impacted by latency and throughput constraints. Finally, we suggest an option to consider based on these measurements to improve I/O performance.

The paper is organized as follows: Section II provides a general background on I/O operation and how the DMA transactions typically occur between I/O adapters and higher-level software layers. This is followed by a detailed measurement and analysis of typical current high-performance I/O devices in

Section III. Section IV provides a survey on how various current systems perform I/O transactions. Finally, Section V suggests areas for improvement and optimization opportunities.

II. BACKGROUND AND GENERAL DISCUSSION

Current I/O devices, such as Network Interface Controllers (NICs), storage drives, and Universal Serial Bus (USB), are 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. CPUs with multiples cores 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.

Figure 2 shows a diagram of the internal system components of a current high-performance system based on the Intel 5520 chipset [3]. The 8-core system contains two quad-core processors, each with 8 MB L3 cache, memory interface and QuickPath Interconnect (QPI) coherent memory interface. The speed of each core (3.2 GHz) is not directly applicable to the discussion as we will see that I/O transaction efficiency is governed more directly by the I/O device controllers. The two dotted arrows between the I/O adapter and a CPU indicate the path and bandwidth available for the I/O adapter to read data to be transmitted from system memory and write received data to system memory.

The *I/O Hub* (IOH) interfaces between the QPI interface and multiple Peripheral Component Interconnect express (PCIe) interfaces. This flexible design allows 1 to 4 CPUs to be configured using 1 to 2 IOHs for a variety of I/O expansion capabilities. In addition, each IOH has Basic Input/Output System (BIOS) controlled registers to define the PCIe lane configuration allowing the system to have either multiple low bandwidth PCIe interfaces or fewer high bandwidth PCIe interfaces, such as graphics engines. In previous generations of Intel and AMD systems, the IOH was termed “Northbridge” and included a memory controller allowing the processor silicon to be dedicated to core and cache functions. Advances in silicon die technology have allowed the memory controller to be integrated on the same silicon die with the cores and cache for improved memory performance. This performance improvement is mainly due to the removal of the old “Northbridge” and the related protocol overhead.

In 2012, Intel launched products that integrate the IOH onto the same CPU. This integration reduces system power and form factor size, but not other factors such as latency and throughput. Our measurements show that it is the PCIe interface capabilities that define the latency between the system components and not whether or not the IOH is integrated. For this reason, our measurement analysis discussed in Section III is based on the more recent platform shown in Figure 2.

High performance I/O adapters connect directly to the IOH, while more basic I/O adapters are interfaced with the *I/O Controller Hub* (ICH). The ICH, which was often termed “Southbridge” in previous system generations, supports the interface hardware to BIOS boot flash memory, direct attached storage, USB, and system management modules that include temperature, voltage, and current sensors. Our focus is on I/O devices connected to the IOH interface and not ICH-connected devices.

An NIC is used as the baseline I/O device since it offers a wide variety of I/O performance factors to study. For storage, throughput is more important than latency, and several techniques can be incorporated into NICs to enhance storage over the network. For clusters and high-performance computing, latency is often a critical component, thus different techniques can be applied. Table 1 lists the performance focus and reference section.

Table 1: NIC performance accelerator examples

Performance focus	Techniques to improve performance	Chapter reference
Throughput	CPU DMA	IV.A.2
Throughput	Interrupt moderation	IV.B.2
Throughput	RSS, LRO, LSO	IV.C.1
Latency	LLI	IV.B.2
Latency	Infiniband	IV.D.3
Latency	User-based I/O	IV.B.1

Note that although power minimization is always a concern, it is common practice to disable power-saving states to maintain a low (and predictable) I/O latency. This baseline NIC for I/O transactions can be extended to other high-performance devices, such as disk storage controllers and graphics adapters. These devices all share the descriptor-based DMA transactions that will be discussed in detail in the rest of this section.

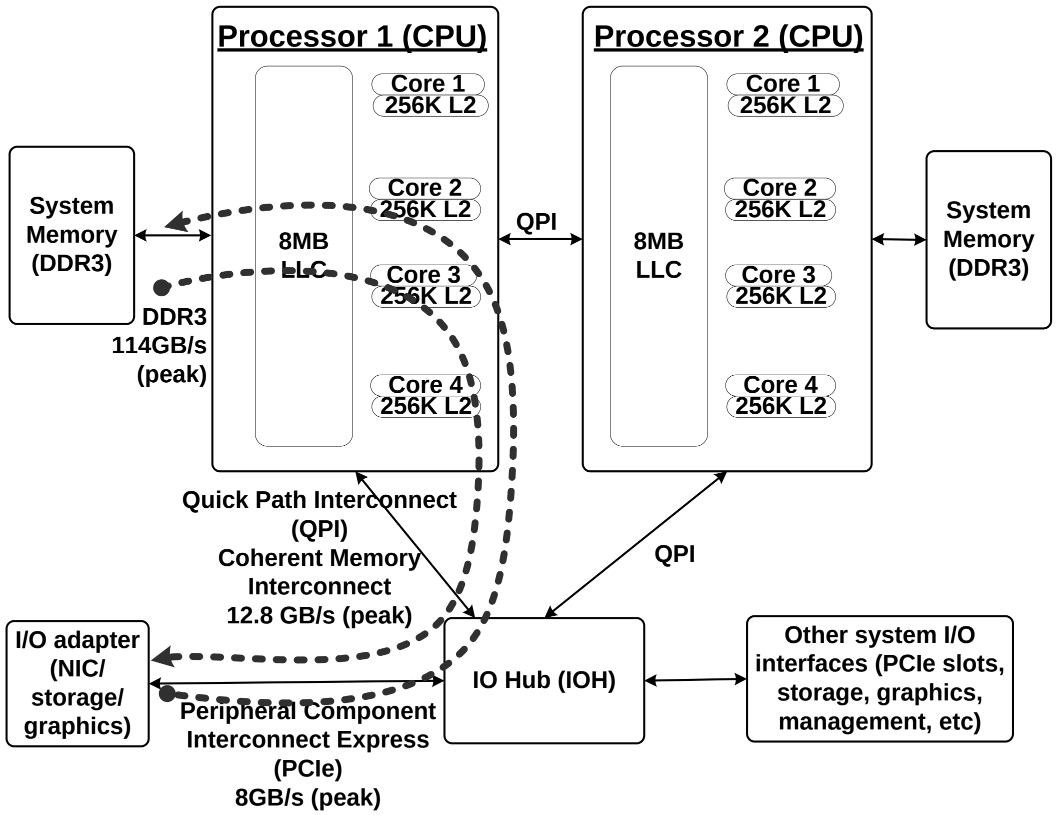
Processor 1 (CPU)
8MB LLC
Core 1 256K L2
Core 2 256K L2
Core 3 256K L2
Core 4 256K L2
Processor 2 (CPU)
8MB LLC
Core 1 256K L2
Core 2 256K L2
Core 3 256K L2
Core 4 256K L2
System Memory (DDR3)
System Memory (DDR3)
QPI
DDR3 114GB/s (peak)
Quick Path Interconnect (QPI) Coherent Memory Interconnect 12.8 GB/s (peak)
QPI
I/O adapter (NIC/ storage/ graphics)
IO Hub (IOH)
Other system I/O interfaces (PCIe slots, storage, graphics, management, etc)
Peripheral Component Interconnect Express (PCIe) 8GB/s (peak)

Figure 2: High-performance I/O system block diagram (adapted from [3])

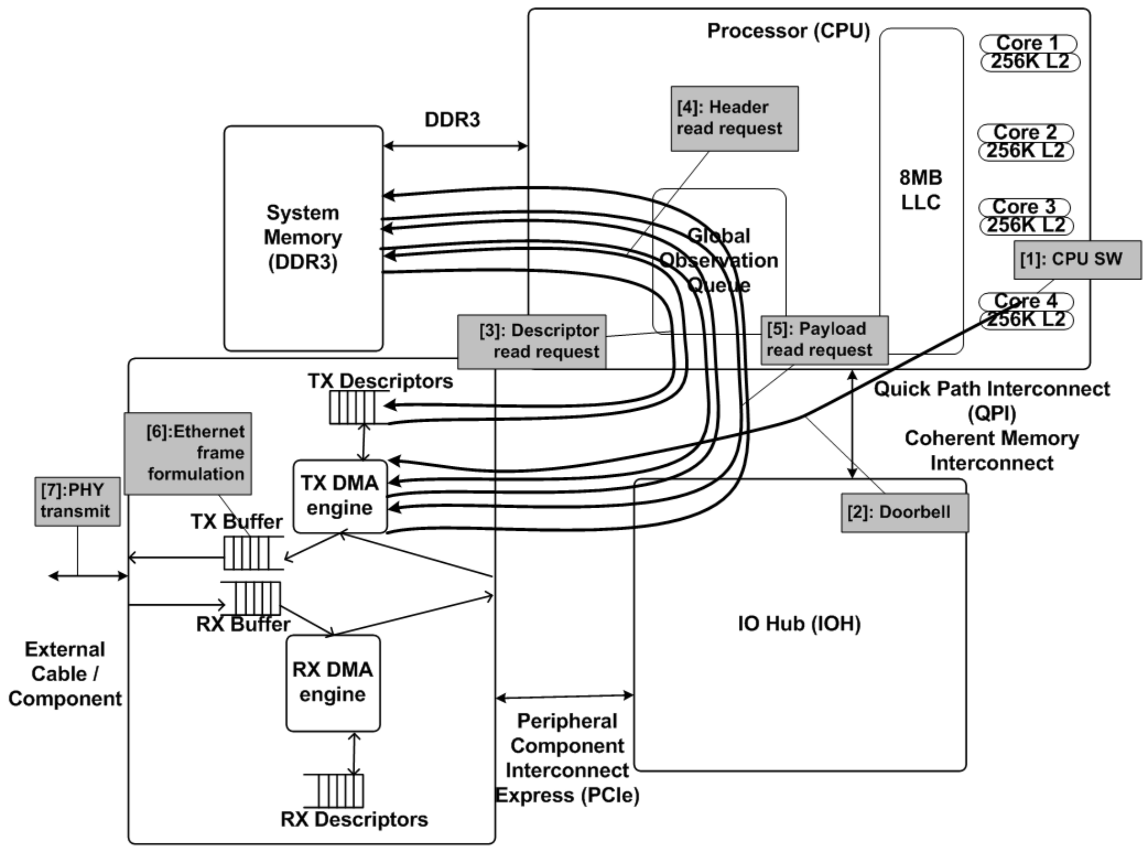

Figure 3: Typical Ethernet transmit flow.

Figure 3 illustrates a typical I/O transmission for an Ethernet NIC (either wired or wireless). 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 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* 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 by a core 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 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 for the payload is not included in the doorbell request because there are two separate 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). A tracking mechanism called a Global Observation Queue (GOQ) in the CPU controls memory transaction coherency such that on-die cores or other CPUs correctly snoops the (system) bus for memory requests. The GOQ also helps avoid 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. Then, 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 address 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 (PHY) layer 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 and is shown in Figure 4. After a core prepares a descriptor, the NIC performs a DMA operation to transfer the received packet into the system memory. After the transfer completes, the NIC interrupts the processor and updates the descriptor.

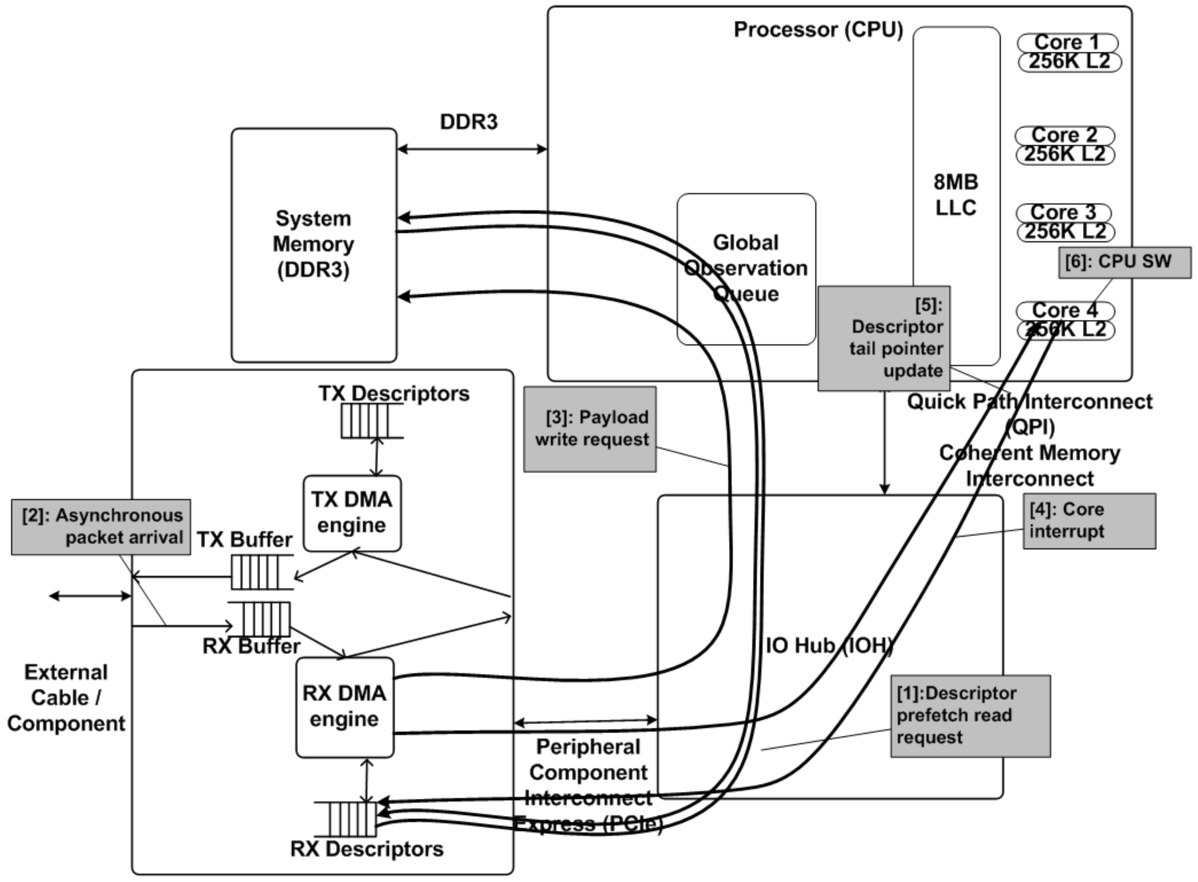
Processor (CPU)
Core 1
256K L2
Core 2
256K L2
Core 3
256K L2
Core 4
256K L2
8MB LLC
DDR3
System Memory (DDR3)
Global Observation Queue
[6]: CPU SW
[5]: Descriptor tail pointer update
Quick Path Interconnect (QPI) Coherent Memory Interconnect
[3]: Payload write request
[4]: Core interrupt
TX Descriptors
TX DMA engine
[2]: Asynchronous packet arrival
TX Buffer
RX Buffer
RX DMA engine
External Cable / Component
IO Hub (IOH)
[1]:Descriptor prefetch read request
Peripheral Component Interconnect Express (PCIe)
RX Descriptors

Figure 4: 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 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 system memory space pointed to by the receive descriptor.
(4) After the memory write transaction completes, the NIC interrupts the core indicating a new packet has been received for further processing.
(5) As part of the interrupt processing routine, the core driver software issues a write to the NIC to synchronize the NIC adapter 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 in the system memory.

III. MEASUREMENTS AND QUANTIFICATIONS

In order to quantify various I/O design aspects of current servers and workstations and explore potential changes, a conventional NIC was placed in an IOH slot of a platform described in Section II. A PCIe protocol analyzer was used to observe the PCIe transactions, which are summarized in Table 2. Using measurements on a real (and current) system offers validity in extrapolations and conclusions that are less certain in simulated environments.

Table 2: Quantified metrics of current descriptor-based DMA transactions.

	Factor	
	Latency	**Bandwidth-per-pin**
Description	Latency to transmit a TCP/IP message between two systems	Gbps per serial link
Measured value	8.6 µs	2.1 Gbps/link
Descriptor related overhead	18%	17%
See Section	III.A	III.B

The following subsections discuss the measurements and analysis in more detail. By observing the latency breakdown and bandwidth-per-pin utilization, we can explore requirements and inefficiencies in the current model.

III.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 [4]. Another example is in High Performance Computing (HPC) nodes where LinPack benchmark (used to define the Top500 supercomputers) share partial calculations of linear algebra matrix results among nodes [5].

Figure 5 shows a typical 10 Gigabit-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 10 GbE Intel 82598 NICs (code named Oplin) on PCIe ×8 Gen1 interfaces [6]. 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.

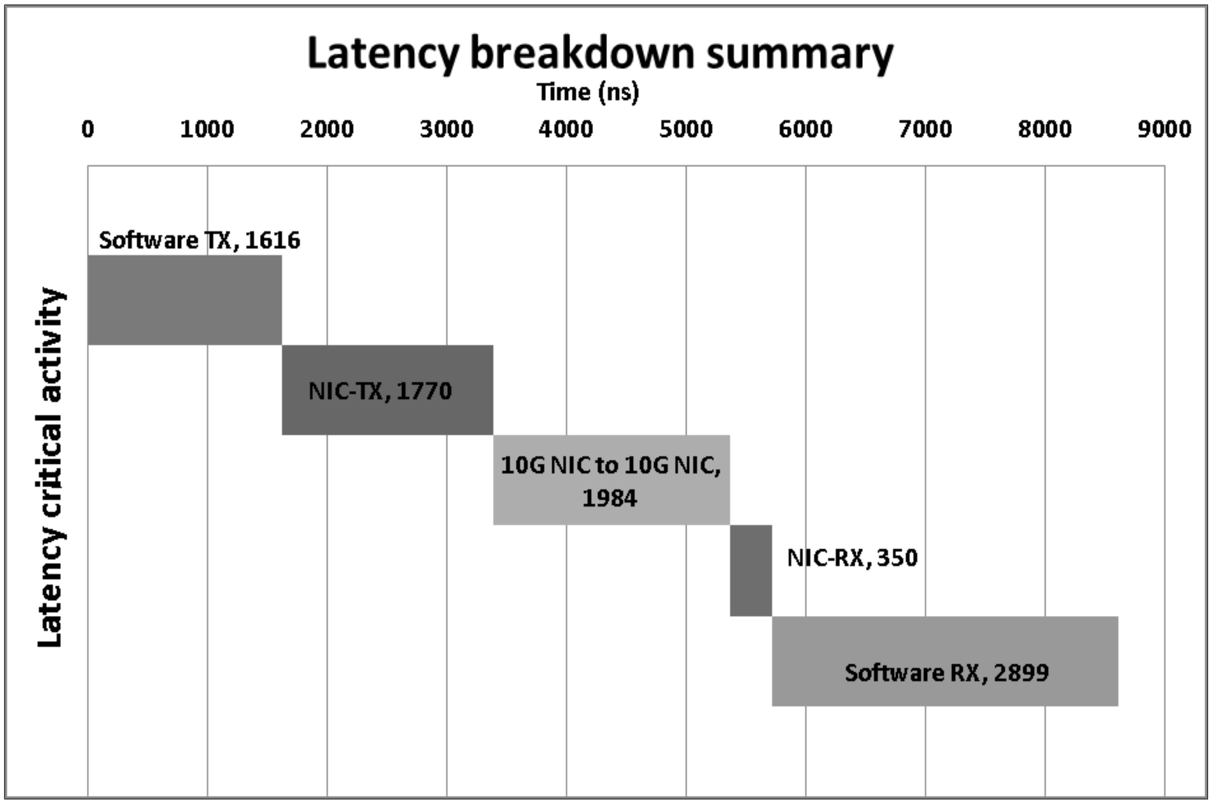

Figure 5: GbE critical path latency between two systems.

End-to-end latency consists of both hardware and software delays, and depends on many aspects not directly addressed in this article, such as core and memory clock frequencies, bandwidth, and cache structure. The critical path latency of the software stack is around 1.61 µs and 2.9 µs for send and receive, respectively, and is not related to descriptor-based I/O communication since it is only associated with how a core handles I/O traffic data that is already in system memory. Software latency in terms of the core cycles required to formulate TCP/IP frames for transmit and processing received TCP/IP frames is described in more detail in [7]. On the other hand, 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, which is around 1.77 µs. This is followed by a flight latency of 1.98 µs for the wire/fiber and the TX/RX NIC state machines (NIC to NIC). Finally, the RX NIC requires 0.35 µs to perform DMA writes (NIC-RX) to push the data from the RX NIC buffer into the system memory and interrupt a core for software processing. The total latency $Latency_{Total}$ is 8.6 µs and can be expressed by the following equation:

$$Latency_{Total} = Tx_{SW} + Tx_{NIC} + fiber + Rx_{NIC} + Rx_{SW}$$

The latency for the Tx_{NIC} portion can be further broken down using PCIe traces as shown in Figure 6. 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. Note that 43% and 44% of the transmit HW latency (Tx_{NIC}) are used by the descriptor fetch (and decode) and payload read, respectively. This is important in the scope of getting a packet from memory to the wire, and assuming the core could write the payload directly to the NIC, 1770 ns could be nearly reduced to 230 ns. This results in about 18% reduction in total end-to-end latency as shown in Table 2.

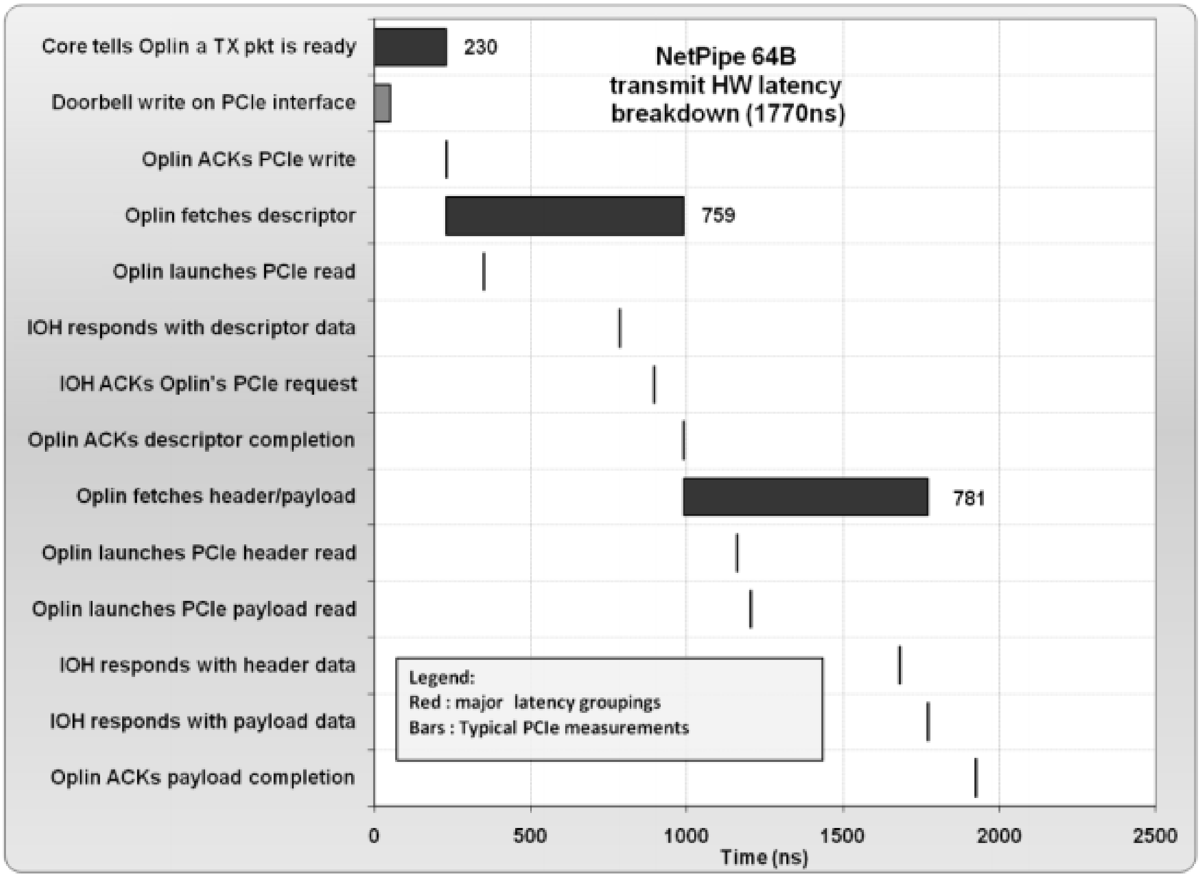

Figure 6: NIC TX latency breakdown.

III.B Throughput and Bandwidth Efficiency

The iperf bandwidth benchmark was used on a dual 10 GbE Intel 82599 Ethernet adapter (codename Niantic) [8] on an Intel 5500 server. The primary difference between the 82598 and the 82599 Intel NIC is the increase in PCIe bandwidth. This does not impact the validity of the previous latency discussion, but allows throughput tests up to the theoretical 2×10GbE maximum. PCIe captures consisted of more than 300,000 PCIe ×8 Gen2 packets, or 10 ms of real-time trace, which gives a statistically stable data for analysis.

Figure 7 shows a breakdown of transaction utilization in receiving and transmitting data on a PCIe interface for a dual 10 GbE NIC. The four stacked-bars show extreme cases of TCP/IP receive (RX_) and transmit (TX_) traffic for small (_64B_) and large (_64KB_) I/O message sizes. Since throughput is lower than the link rate for small message sizes, we also show the aggregate throughput (1Gbps, 18Gbps, 400Mbps, 19Gbps) across both 10GbE ports when using the iperf benchmark. The traffic is normalized to 100% to illustrate the proportion of non-payload related traffic across the PCIe interface.

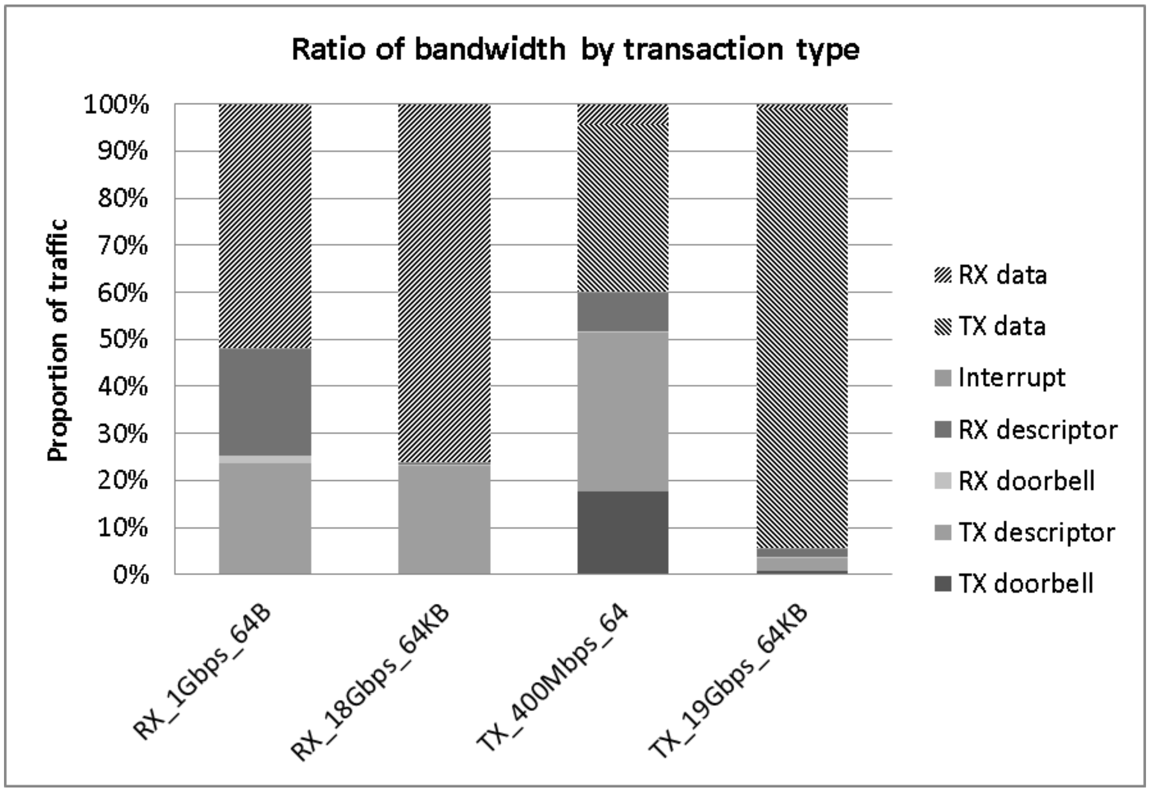

Figure 7: Proportions of PCIe transaction bandwidths.

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 operation on small messages is representative of latency sensitive transactions while large I/O is representative of storage types of transactions. Figure 8 highlights the impact of non-payload related PCIe bandwidth when the PCIe frame overhead is factored in to account for the actual bandwidth required for each frame. This figure shows that descriptors and doorbell transactions for small messages represent a significant portion of the total PCIe bandwidth utilized in this measurement. This includes PCIe packet header and Cyclic Redundancy Check (CRC) data along with PCIe packet fragmentation. If descriptor and doorbell overhead were to be removed, the bandwidth could be improved by up to 43% as indicated by the TX_400Mbps_64 case. In the case of I/O receive for small payload sizes, the inefficiency due to descriptors and doorbells is only 16% since 16-byte descriptors can be pre-fetched in a 64-byte 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.

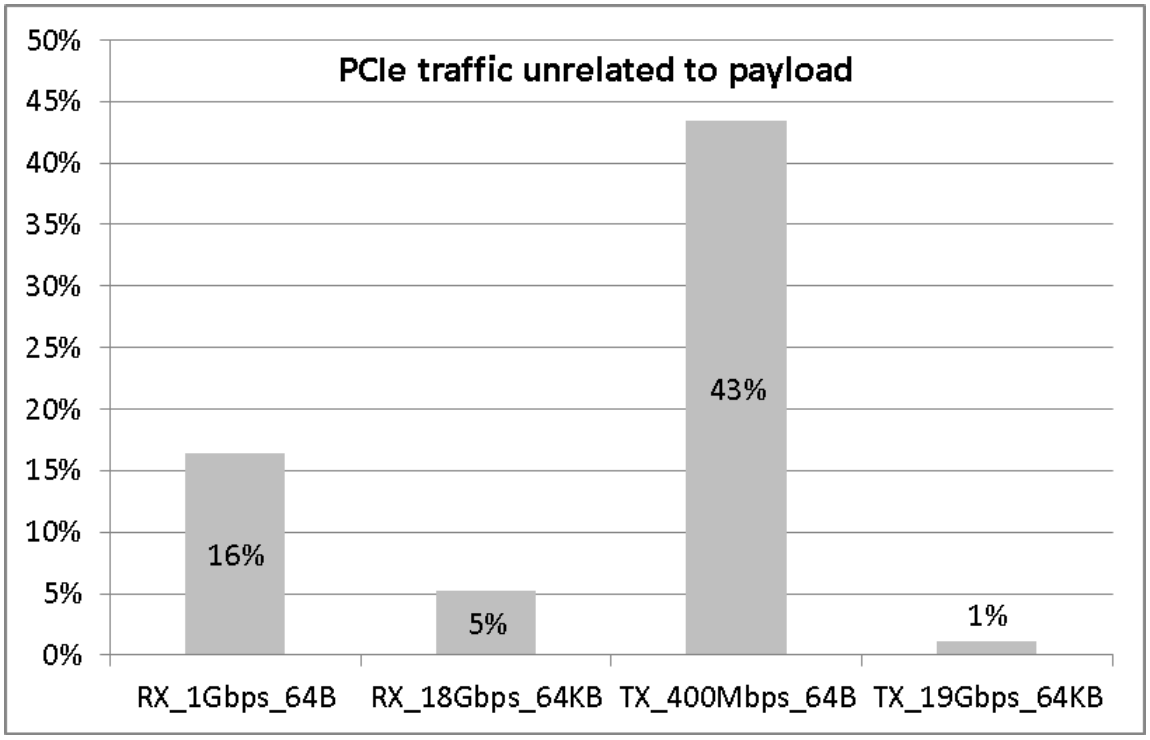

Figure 8: PCIe bandwidth utilized for non-payload vs. payload bandwidth.

IV. 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. Since the scope of this article is on the hardware I/O transactions that are essentially common regardless of application, the survey is structured based on system complexity for the following four categories: (1) simple systems, (2) workstations and servers, (3) datacenters and High-Performance Computing (HPC) clusters, and (4) system interconnects and networks. Note that an alternative I/O survey organization could be based on different I/O usages (such as networking, storage and video applications). However, this would add an orthogonal dimension to the survey, and thus is not considered.

IV.A Simple Systems - Direct I/O access and basic DMA operations

This subsection discusses I/O for systems that access I/O directly by the CPU without DMA, sometimes termed *Programmed I/O* (PIO) as well as systems that have DMA engines directly associated with CPUs. These include embedded systems, systems with Digital Signal Processing (DSP) and graphics subsystems. These systems may either access I/O directly using the CPU instruction set or set up a DMA operation that is associated directly with a CPU. Real-Time Operating Systems (RTOS) is part of this classification since the I/O performance characterization is a critical part of system performance.

IV.A.1 Embedded systems

The simplest method of system I/O, which is usually found in slower systems with a single dedicated function, is found in embedded controllers with dedicated memory locations for I/O data. An example would be a clock radio where outputs are LCD segment signals and inputs are buttons with dedicated signals that can 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 microcontroller can support I/O protocols at very low bandwidths such as "bit-banging" mentioned in Section I as implemented with SoftModem [1]

IV.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 common in Digital Signal Processing (DSP) systems where large amounts of data need to be moved between I/O, signal processing function blocks, and memory. The basic operations consist of:

1. CPU 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. CPU configures the DMA engine to perform the data transfer.
3. CPU either polls or waits for DMA interrupt for completion.

An example of such as system is the IBM/TI Cell processor shown in Figure 9, which consists of Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPE) or cores [9, 10]. Each SPE has a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC), which is used to handle the SPE DMA transactions.

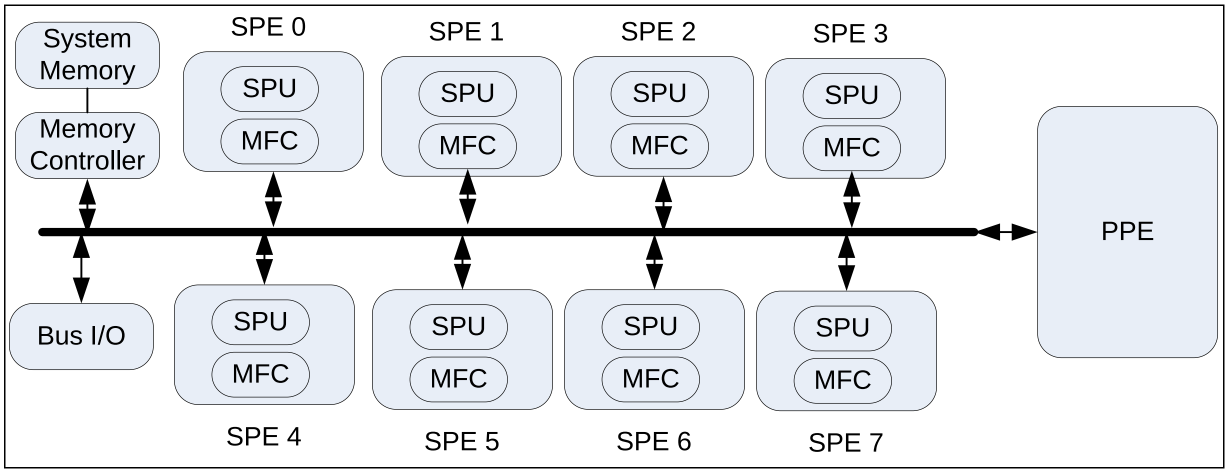

Figure 9: Cell processor.

Each SPE has 256 KB of local on-die memory. If a SPE requires non-local memory access, it can either request the PPE for a kernel/OS service or configure DMA to perform I/O transaction from the System Memory to and from the SPE. To reduce the SPE DMA programming overhead, Ionkov *et al.* [9] proposed using co-routines to manage I/O transactions between SPEs and System Memory. This removes the requirement to program each SPE DMA transaction, but adds a software layer that requires tracking co-routine state that impacts the number of co-routines that can be supported and the switch time between co-routines. There is no memory coherency structure, as found in x86 CPUs, reducing inter-core communication requirements. The Cell processor architecture shows how data movement via core-controlled DMAs is effective for graphics processing in current workloads.

Further examples of core-based DMA control are presented in [11, 12] covering typical embedded DMA processing, Katz and Gentile [13] provide a similar description of DMA on a Texas Instruments Blackfin digital signal processor. These systems use DMA descriptors to define physical payload status and memory locations similar to legacy Ethernet I/O processing described in Section II.

IV.A.3 General CPU controlled DMA engines

An alternative to the simple embedded controller system cases discussed thus far is Intel's DMA offload engine in server CPUs called QuickData Technology, which a method to improve I/O performance for storage and Redundant Array of Independent Disks (RAID) [14]. Certain RAID configurations will use XOR bit-level calculations to regenerate data on failed disk drives [15]. Often an expensive RAID storage controller will execute the XOR functions, but the Intel QuickData Technology allows standard disks to be attached in a RAID configuration and support in-flight XOR calculations. Currently, it is implemented in the IOH shown in Figure 2, and with a large and cumbersome 64-byte descriptor. Obviously, there is inefficiency in handling asynchronous variable sized data common in networking that needs to be setup before any DMA copies can be made between I/O adapter and memory. As a result, the DMA engine 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 [16], in part because memory accesses can be pipelined and thus hide a software core-controlled memory-to-memory copy.

IV.A.4 PCIe bandwidth optimization

Yu *et al.* [17]described how existing DMA transfer speed can be improved by increasing buffer efficiencies for large block DMA transfers on a particular PCIe implementation (PEX8311). Figure 10 shows the speedup, which is achieved primarily by expanding the PCIe packet frame size. An analogy could be made to the use of jumbo frames on Ethernet that exceed the default 1500 byte maximum transfer unit (MTU). PCIe transactions normally have a maximum frame payload of 256 bytes because larger maximum frame payloads would require more silicon on both ends of the PCIe interface. The selection of 256 bytes is an industry norm since often each end of the PCIe interface can be populated by silicon from various companies. The PCIe protocol specifies three headers for each transaction [18]: Transaction Layer Protocol (TLP), Data Layer Protocol (DLP) and Link Layer Protocol (LLP), which combined add 24 bytes to each PCIe transaction reducing the effective bandwidth.

Tumeo *et al.* [19]provide details on optimizing for *double buffering*, and how this technique can be used to optimize latency in a multi-core FPGA with each core having a DMA engine to move data between system memory and core memory. The basic idea is to pipeline multiple DMA transactions such that the PCIe interface is optimally utilized. While this allows DMA transactions to be setup, executed and terminated in parallel, it can be detrimental to latency and predictability of latency as discussed in the next section.

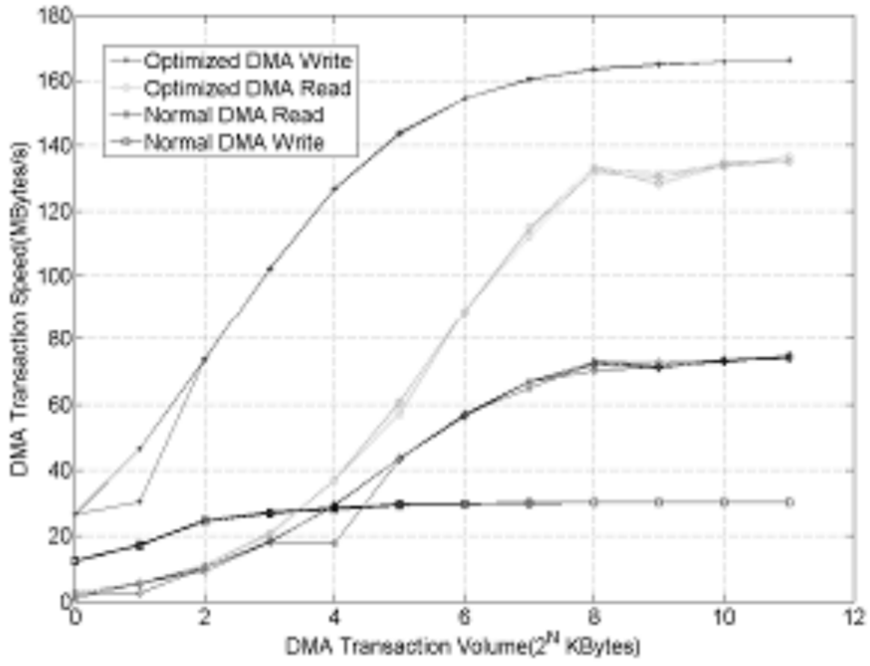

Figure 10: Performance comparison between normal and optimized DMA

IV.A.5 Predictability and Real-Time Operating Systems

In RTOS, *I/O latency prediction* is an important factor to guarantee predictable operations. If an I/O transaction cannot be predicted to occur within a certain time interval, the degree of deterministic behavior by the operating system cannot be defined. The smaller variability in latency prediction of an I/O transaction results in better overall RTOS performance. Several papers present models on how to accurately predict and bound the DMA latency when controlled by the I/O adapter [20-24]. Worst Case Execution Time (WCET) is the most critical parameter to consider. Current I/O transactions use multiple DMA engines in different clock domains throughout the system and each engine may have different I/O transaction queuing and quality of service characteristics. Based on this, the predictability of I/O transactions is generally inversely proportional to system complexity.

A more predicable I/O transaction can involve the core directly accessing the I/O device using *program I/O* (PIO). However, I/O adapter DMA engines remain common in systems with RTOS since standard "bit-banging" by the core results in poor I/O performance. For example, Salah and El-Badawi compared PIO to DMA, where PIO resulted in only 10% throughput compared to DMA throughput measurements [20].

IV.A.6 Other DMA related proposals

The *dynamic compression of data* for memory accesses discussed in [25] is an option to reduce I/O latency and reduce the chip-to-chip bandwidth. However, there is added logic complexity required to compress and decompress payload data. In addition, the I/O device is often clocked at a much slower rate than the memory interface, such as the Intel 82599 NIC internal frequency of 155 MHz [8].

One method to overcome the problem with a common clock DMA engine to transfer data between memory devices is to use *asynchronous DMA.* With asynchronous DMA the transfer occurs as soon as the producer has the data to transfer and requires the consumer to be available without a defined clock boundary. The challenges of asynchronous DMA are discussed in [26, 27], 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 would reduce latency by a few clock cycles in the system shown in Figure 2.

Table 3 summarizes how simple systems implement I/O transactions, their primary benefits and costs, and example implementations.

Table 3: Simple System I/O transactions

Simple systems	Key Benefits	Key Costs	Example implementation
Embedded Systems	No DMA – allows direct interaction between controller and I/O interface	Core overhead to read and write directly to I/O interface	Clock radio
CPU DMA and DSP	Each core has a dedicated DMA engine close to core to service data movement saving core cycles and power	Per core silicon area and power	Cell processor and 1980s personal computers
General CPU DMA engine	Shared DMA engine between cores and CPUs saves silicon area and power	DMA transaction complexity and increased latency	Intel QuickData technology
PCIe Bandwidth Optimization	Reducing PCIe protocol overhead allows lower I/O latency	Requires larger PCIe buffers that consume more silicon area and power	Research proposals [17] [19]
Predictability and RTOS	More predictable latency bounds and throughput minimums by removing DMA variability	Overall lower system throughput	RTOS research to optimize Worst Case Execution Time (WCET)
Other DMA proposals	I/O data compression and asynchronous DMA may allow lower latencies	Silicon complexity and power	Research proposals [25-27]

IV.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 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/outgoing data to/from system memory. Offloading I/O transactions to a DMA engine is very effective since the I/O device is typically a 100 MHz state machine while the CPU operates in multi-GHz range allowing the I/O to proceed as fast as it 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 on current desktop and workstation systems. This section discusses the I/O issues of such systems and how they are addressed.

IV.B.1 Operating system virtualization protection

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; however, there is a penalty in performance. Latency increases because the application software needs to first request kernel services to perform I/O transactions. Throughput may also 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 carefully controlling *user-mode DMA I/O.* The two proposals discussed in [28, 29] describe how to bring kernel-based system calls for moving data into user space so that it can be accessed by user applications. Significant performance improvement can be obtained for small I/O messages by having a user application directly control data movement rather than using system calls. 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 also processes on other systems.

IV.B.2 Interrupts and moderation

Interrupt methods have advanced over the last several years. Traditionally, an interrupt signal or connection was asserted upon requiring CPU software services. This requires a core context switch and parsing through the available interrupt sources within a system. Since other interrupt sources could be I/O devices on slow interfaces, polling for interrupt would require many core cycles. In current systems, Message Signaled Interrupts (MSI) uses a message on the PCIe interface rather than a separate signal for a device interrupt, which saves circuit board resource and allows up to 32 interrupt sources to be defined in the message [18]. MSI was extended to MSI-X that allows up to 2048 sources since there can often be more than 32 interrupt sources in a system. As a result, a core being interrupted can directly proceed to the needed interrupt vector software.

Interrupt moderation, which is also referred to as interrupt coalescing or interrupt aggregation, allows multiple incoming data (e.g., Ethernet packets) to be processed with a single interrupt. This results in relatively low latency and reduces the number of context switches required for received network traffic. Common methods are discussed in [7] where two timers can be used: One absolute timer with a default, yet configurable, interval of 125 μs to interrupt after any received data arrival, and another per packet timer that can expire based on each received packet arrival.

Since network traffic can be both latency and throughput sensitive, adaptive interrupt moderation has been an ongoing area of research and implementations. For example Intel's 82599 NIC allows interrupt filtering control based on frame size, protocol, IP address, and other parameters [8]. This Low Latency Interrupt (LLI) moderation uses credits based on received packet event control rather than on timer control.

Table 4 summarizes how generic workstations and servers implement I/O transactions, their primary benefits and costs, and example implementations

Table 4: Generic Workstation and Server I/O transactions

Generic Workstations and Servers	Key Benefits	Key Costs	Example Implementation
Operating system virtualization protection	Protection from I/O generated access to system resources	Latency and CPU overhead to authorize user access to operating system controlled transactions	Linux and Windows OS protection layers
Interrupts and moderation	Reduced CPU overhead to manage I/O transactions	Added latency per I/O transaction	10GbE network interface controllers

IV.C Datacenters and HPC cluster systems

Systems that are used in datacenters and High-Performance Computing (HPC) clusters have requirements beyond general stand-alone workstations and servers. Often these systems are composed of high-end servers using multiple 10 Gigabit Ethernet (GbE) interconnects, 15,000 RPM disks, multiple Graphic Processing Units (GPUs), and Solid-State Drive (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 [30] for internode communications. Some HPCs will have front-end CPUs to prepare the I/O in memory for high-speed processing. An example would be an environmental simulation that loads the working set into memory and after simulation outputs a completed working set.

Since Ethernet is ubiquitous and flexible as a network and storage interface, this subsection considers some of the high-end capabilities found in datacenters as link speeds increase to 10 GbE and beyond. We first review I/O receive optimizations followed by I/O transmit optimizations, ending with more complex bi-directional improvements.

IV.C.1 Device I/O Optimizations

Received Side Scaling (RSS) is the ability to de-multiplex Ethernet traffic and effectively spread the traffic flow across multiple available cores within a system [31]. 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 interrupt and traffic to a core that presumably maintains context for the related TCP session. An additional benefit of RSS is that a given TCP session will often be serviced by a persistent core that may well have the connection context in cache.

Large Receive Offloading (LRO), also called Receive Side Coalescing (RSC) by Intel [8], is a receive mechanism to clump sequenced frames together, presenting them to the operating system as a single receive frame [32]. This effectively allows creation of a jumbo frame avoiding the higher-level software layer to patch the discrete smaller Ethernet frames (typically 1500 bytes) together.

Large Segment Offload (LSO), also called TCP Segmentation Offload (TSO) or Generic Segmentation Offload (GSO), allows an I/O device to be given a pointer to a segment of data much larger than a single frame (e.g., 64 KB) and stream the data for transmits [33]. With 64 KB sized messages, LSO can improve performance by up to 50%.

IV.C.2 Descriptor packing

Descriptor coalescing can be used to reduce the overall receive latency. For example, an Ethernet descriptor size is typically 16 bytes and thus multiple descriptors can be read from a single doorbell [34] (e.g., four descriptors would be read at once for a cache line size of 64 bytes). This works well when the NIC is prefetching descriptors to DMA packets that will be received in the future.

However, a server supporting web or database transactions for thousands of sessions over multiple descriptor queues may not allow for transmit descriptor bundling. This is because organizing the transmit descriptors for packing over thousands of sessions may not be productive since transmit messages need to be formulated in system memory before the descriptor can be defined. In contrast, transmit descriptor coalescing may be beneficial when 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 leading to longer latencies.

IV.C.3 TCP/IP Offload Engine (TOE) and other offload engines

Offloading the entire TCP/IP stack onto an I/O adapter can reduce the stack protocol overhead on the CPU [35]. Figure 11 shows an example of *TCP/IP Offload Engine* (TOE).

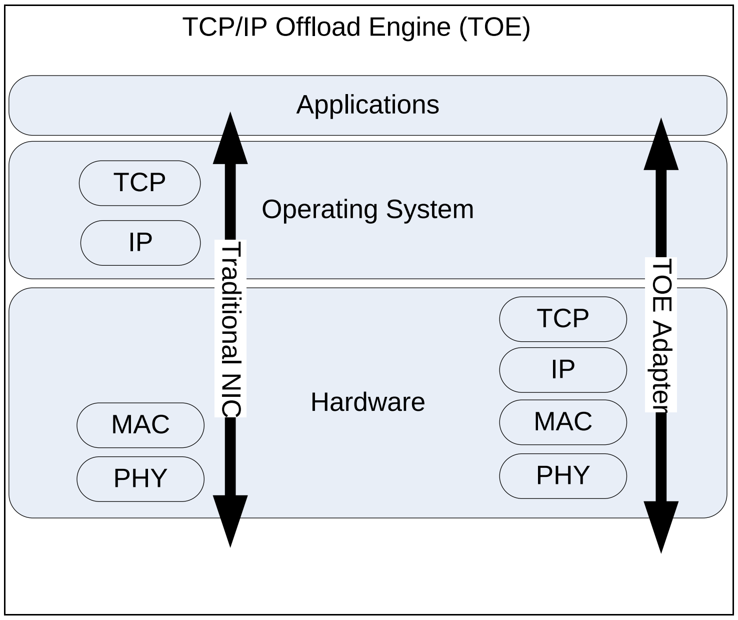

Figure 11: Basic TOE comparison to software-based implementation.

This approach is beneficial for large blocks of I/O, such as storage with average block size greater than 4 KB. The downside of this approach is that HPC and datacenter servers have to also process small messages. For example, virtualizing multiple operating systems within a single system is common due to increased capabilities of CPUs and memory. This leads to a single I/O adapter processing small packets over thousands of TCP connections. The TOE also processes frames at slower frequencies (typically in the order of 100 MHz) than a modern CPU with multi-GHz frequencies. This adds significant amount logic complexity and connection context memory requirements to the less flexible TOE I/O adapter [35].

In general, any higher level software I/O protocol may be implemented in an I/O adapter. For instance, the Intel 82599 does not offload the TCP stack, but does offload IPsec and Fibre Channel over Ethernet protocols [8]. 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 [36]. Since a proprietary protocol is used, Myrinet-specific routing devices between nodes are required.

IV.C.4 Coherent Network Interface

Mukherjee *et al.* proposed a *coherent NIC interface* (CNI) [37], which was followed by an implementation on the Front-Side Bus (FSB) architecture by Schlansker *et al.* [38]. A block diagram of the coherent memory implementation is shown in Figure 12, which consists of Transmit Command Queue (TCQ), Receive Command Queue (RCQ), and Rx Data Buffer that are all memory-mapped to coherent memory address space. This approach makes the NIC a peer to core memory communication by exposing some of the NIC buffers as coherent system memory. The implementation in coherent memory removes the need for DMA transactions since the core directly reads or writes to the NIC pointers and buffers. They implemented a prototype design and tested on a CPU socket using an FPGA.

The TCQ maintains the transmit descriptor information such that as soon as the CPU software writes the location of the transmit packet in system memory, the DMA engine fetches the packet for transmission. This removes the requirement to fetch a descriptor for transmitting a network packet.

The RCQ is updated as soon as a receive packet is placed in the Rx Data Buffer. Both regions are in coherent memory allowing the CPU software to access the CNI receive packets without the traditional DMA operation into system memory. When the CPU, either through polling or interrupt, is signaled with a receive operation, the Rx Data Buffer points to the appropriate memory location for the received packet.

The key detriment to this implementation is the additional coherency traffic since CNI has to perform snoops and write-back operations between cores and NIC buffers. This additional traffic conflicts with CPU related traffic between cores on the FSB [39].

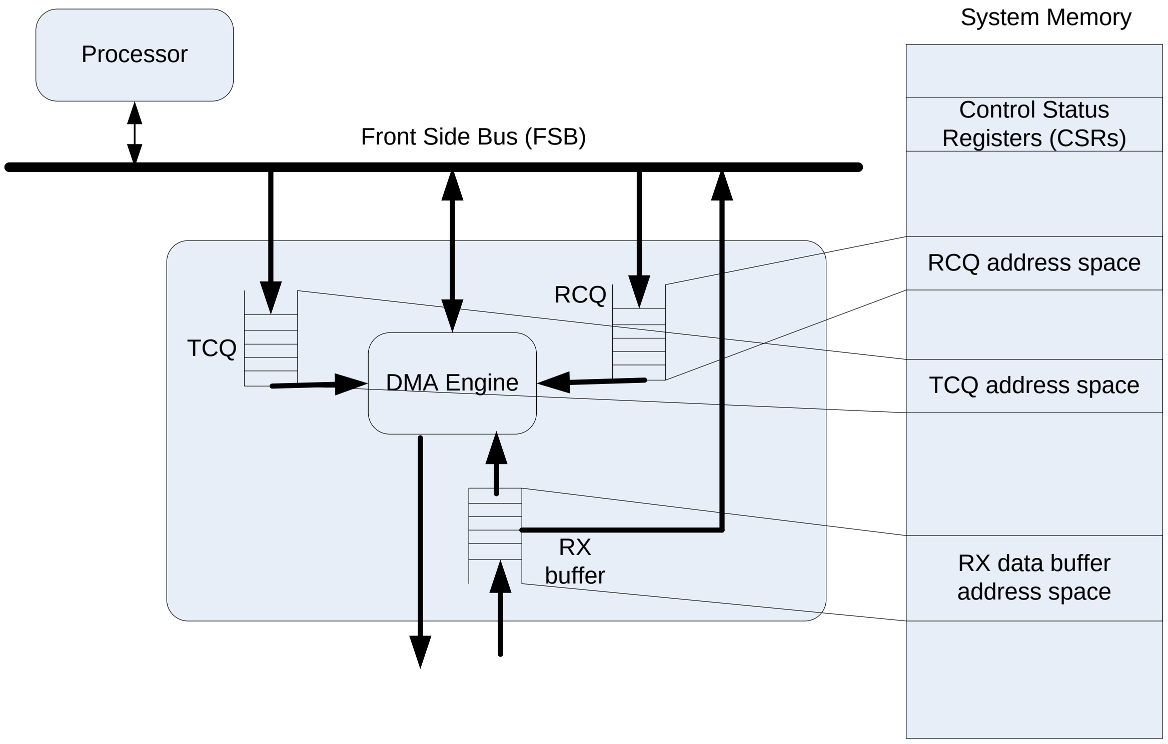

Figure 12: NIC system memory apertures.

IV.C.5 CPU caching optimizations

One optimization to I/O DMA is *Direct Cache Access* (DCA), where an I/O device can write to a processor's Last Level Cache (LLC) 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 [40]. However, this method still requires a descriptor fetch for the I/O device to determine where to place the data in physical memory.

Work such as Dan *et al.* explores the idea of having a separate I/O DMA cache structure, allowing DMA traffic to be cached separately from non-I/O related data [41]. Similar to DCA, this allows a CPU to have access to receive traffic with an on-die cache and to transmit without requiring the system memory transactions. A criticism of DCA is the risk of cache pollution where the cache becomes a dumping site for I/O evicting more critical data. This risk of pollution can be lowered by having a dedicated I/O cache or partitioning the cache structure into I/O cache and general cache. Nevertheless, a dedicated I/O DMA cache would require more CPU silicon resources and benefits would be small for systems with small I/O.

IV.C.6 CPU network interface integration

One approach that is assumed to improve performance is the integration of I/O adapters closer to the CPU. By integrating I/O adapters on the same silicon as the CPU, there are no chip-to-chip requirements such as the PCIe frame protocol that can impact latency and throughput. The design of Sun's Niagara2 processor with integrated 10 GbE showed that the internal architecture needs to be carefully considered to match the desired network performance [42].   Figures 13 and 14 compare the transmit and receive performance of traditional *discrete NIC* (DNIC) and *integrated NIC* (INIC) on Sun Niagara2 [42]. Both figures show that, as the I/O size increases, INIC results in significantly lower processor utilization compared to DNIC since the CPU generally has faster I/O responses. Figure 13 shows no discernable benefit in terms of bandwidth for transmit traffic. However, Figure 14 shows a slight improvement in terms of CPU utilization for large I/O receive traffic. As a result, simply gluing I/O adapters to processors can often be a waste of die area, power, and development time with little performance improvement.

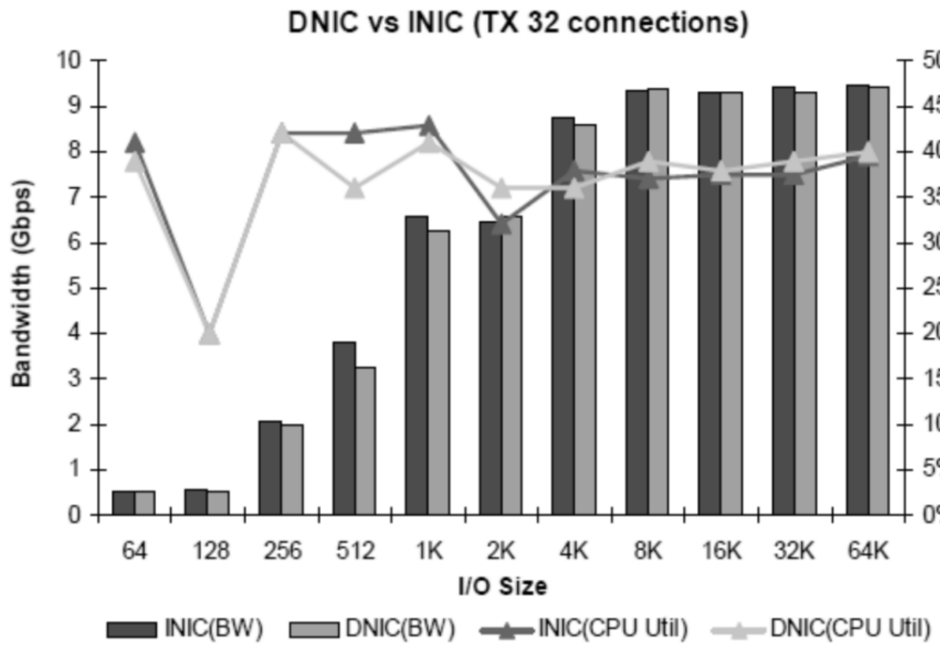

Figure 13: Discrete vs. Integrated NIC transmit performance.

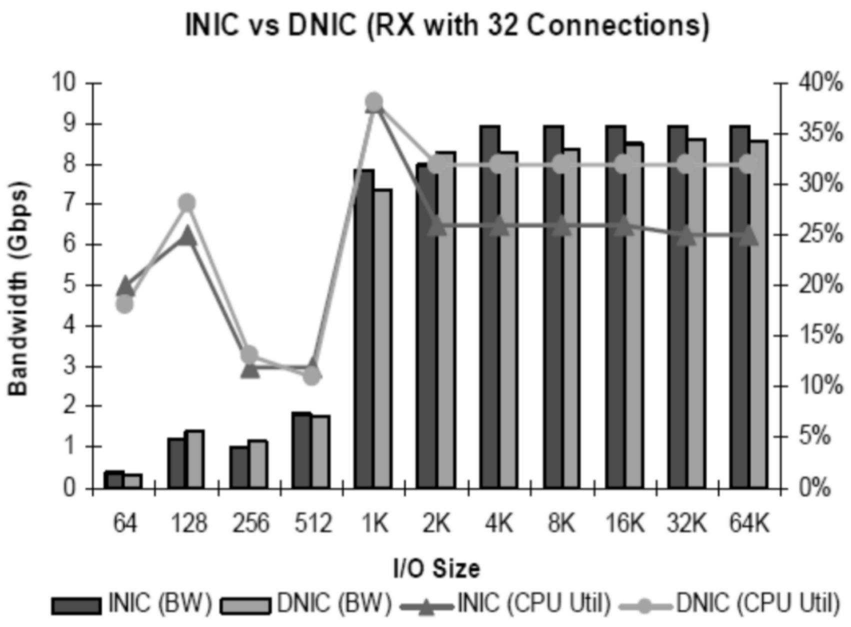

Figure 14: Discrete vs. Integrated NIC receive performance.

Table 5 summarizes how datacenter and HPC cluster systems implement I/O transactions, their primary benefits and costs, and example implementations

Table 5: Datacenter and HPC cluster I/O transactions

Datacenters and HPC clusters	Key benefits	Key costs	Example implementation
Device I/O optimizations	Higher I/O throughput by adding offload accelerators	I/O device silicon area and power for large I/O transactions	RSS, LRO, LSO on Intel and other NIC devices
Descriptor packing	Increased throughput based on lower PCIe interface overhead	Increased latency	Receive descriptor prefetching on Intel 10GbE NIC devices
TOE and other offloads	Offload stack protocol processing to I/O device saving CPU cycles	Increased latency, silicon area and power	Broadcom and Chelsio 10GbE
Coherent Network Interfaces	Reduced latency and higher throughput	Increased silicon area and power with non-standard I/O interfaces	Research proposals [37, 38]
CPU caching optimizations	Lower latency and higher throughput	Probable cache inefficiencies and more coherency transactions	Intel direct cache access
CPU network interface integration	Lower latency and higher throughput	Architectural risk of fixing a CPU with a lower volume I/O device	Oracle/Sun Niagara2

IV.D System interconnects and networks

Since we are examining how data can be moved optimally within a system, this section considers the important role interconnect architectures play in moving data within a larger, more monolithic system (as opposed to a cluster of systems) and between systems. A particular emphasis is given to HPC environment since this area bears more importance on latency and bandwidth performance characteristics. HPCs prioritize CPU-to-CPU communications rather than moving data into and out of a system, so we discuss the system I/O oriented communication aspects.

IV.D.1 CPU socket interconnect

AMD Hyper-Transport [43] and Intel QPI [14] CPU interconnects are two widely accepted and competitive implementations of data movement between multiple CPUs. These interconnects enable the increasing number of cores in a system to interface with each other, memory, and I/O. Not only are there multiple cores, but memory is also distributed using non-uniform memory access (NUMA) architectures. For example, Figure 2 shows only two NUMA nodes (current platforms can have many more) for a total of 8 cores (where each can operate as two logical cores to the operating system with SMT). Communication between cores is usually done using the memory coherency specified by the MESI(F) protocol [44]. Both Hyper-Transport and QPI use snoops and coherence directories to determine common states of all memory locations. While they both address similar architecture concerns, there are differences that make them incompatible.

While these interconnects could be used for system I/O transactions, there has not been any widespread demand for using them as I/O interface. Presumably, part of the reason is that a coherent I/O interface would need to address the coherency overhead discussed in Section IV.C.4.

IV.D.2 Messaging between cores

Intel has explored *I/O messaging* with the Single-chip Cloud Computer (SCC) [45]. 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 15. This allows for efficient movement of data between cores while avoiding cache coherency overhead in the form of snoops and data write-backs. By using the message memory type, the coherency issues with write-back and write-through memory are avoided and the core interfaces are not impacted with irrelevant inter-core traffic. In SCC, this is accomplished by reserving space in the level 1 (L1) cache as non-coherent memory. For core *A* to pass a message to core *B*, the L1 cache line has to be first moved to a non-coherent space, after which the message can be moved to a message-passing buffer to be moved into core *B*'s non-coherent L1 cache space. This becomes particularly important as the core count increases to 48, as is the case for SCC, and beyond. While this is similar in architecture to the well-established Message Passing Interface (MPI) standard [46], MPI defines only the higher software layer that uses TCP/IP or other interconnect fabrics and does not define any of the hardware details.

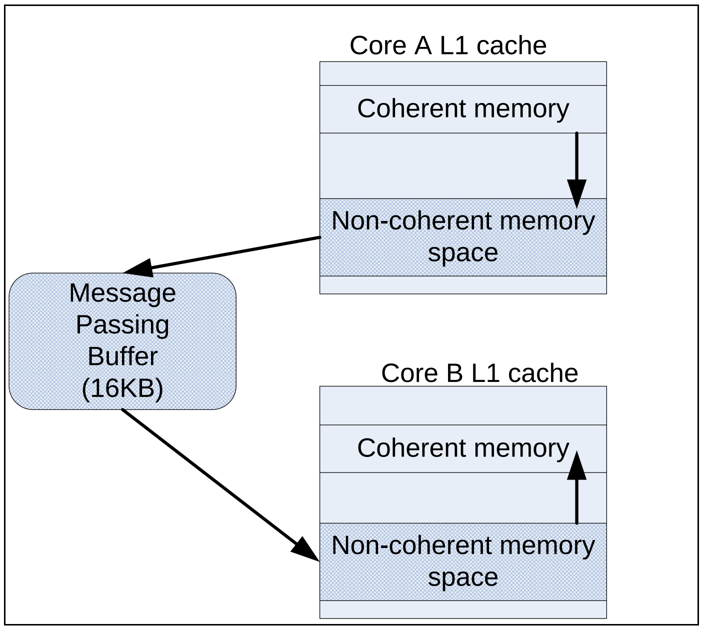
Core A L1 cache
Coherent memory
Non-coherent memory space
Message Passing Buffer (16KB)
Core B L1 cache
Coherent memory
Non-coherent memory space

Figure 15: SCC message data types.

IV.D.3 Remote Direct Memory Access: InfiniBand and iWARP

Remote Direct Memory Access (RDMA) is a method to allow remote access directly into a system memory without involving OS overhead. This reduces latency and increases throughput that is important particularly in datacenters and HPC clusters. In the two systems shown in Figure 16, once a connection has been established with proper authentications, a user application can access the remote system's user application space with no requirement for system calls to the OS on either system. This is basically an offload-engine where the I/O adapter grants physical access to system memory. Protection is ensured by the hardware interfaces and the software protocol.

InfiniBand and Internet Wide Area RDMA Protocol (iWARP) are both implementations of RDMA. iWARP implements RDMA over the standard Ethernet protocol, which allows iWARP clusters to utilize standard Ethernet routers and interconnect frameworks. In contrast, InfiniBand uses a re-designed network that is not compatible with Ethernet.

InfiniBand is considered a premier high bandwidth and low latency system interconnect [8]. The non-Ethernet protocol allows latencies as low as 1 µs for Mellanox ConnectX, which is the 1999 merger of two competing specifications from Future IO and Next Gen IO [47]. It shares many similarities with TOE/Myrinet/I2O in that transmit and receive queues are processed without the OS involvement in that the session connection is offloaded to the I/O device adapter. Figure 16 shows how two I/O devices communicate across an RDMA fabric with each adapter having a Work Queue Entry (WQE) for each communication session. To provide session reliability, the Completion Queue Entry (CQE) supports transaction acknowledgements. Thus, each RDMA session is basically represented by two queues where the WQE queue generates the outgoing work requests and the CQE queue tracks completion of outstanding WQE transactions. These two queues are usually termed a Queue Pair (QP) and are repeated for the sessions used by the higher-level application software.

InfiniBand is a performance reference not only in latency but throughput as well. In a comparison of InfiniBand Double Data Rate (DDR at 8 Gbps) and Quad Data Rate (QDR at 16 Gbps) I/O adapters on Intel platforms [48], the inter-node bandwidth requirements start to exceed the available intra-node bandwidth. In particular, DDR inter-system bandwidth exceeded the PCIe Gen1 host interface of 16 Gbps for traffic of small (less than 1KB) messages.

InfiniBand adds hardware complexity to the I/O adapter to track the connection context, and lacks broad popularity due to the high porting cost both in hardware and software when compared to the ubiquitous Ethernet protocol. This can be seen in the Top-500 supercomputer interconnect listings where 44% are Ethernet-based and only 42% are InfiniBand-based [5].

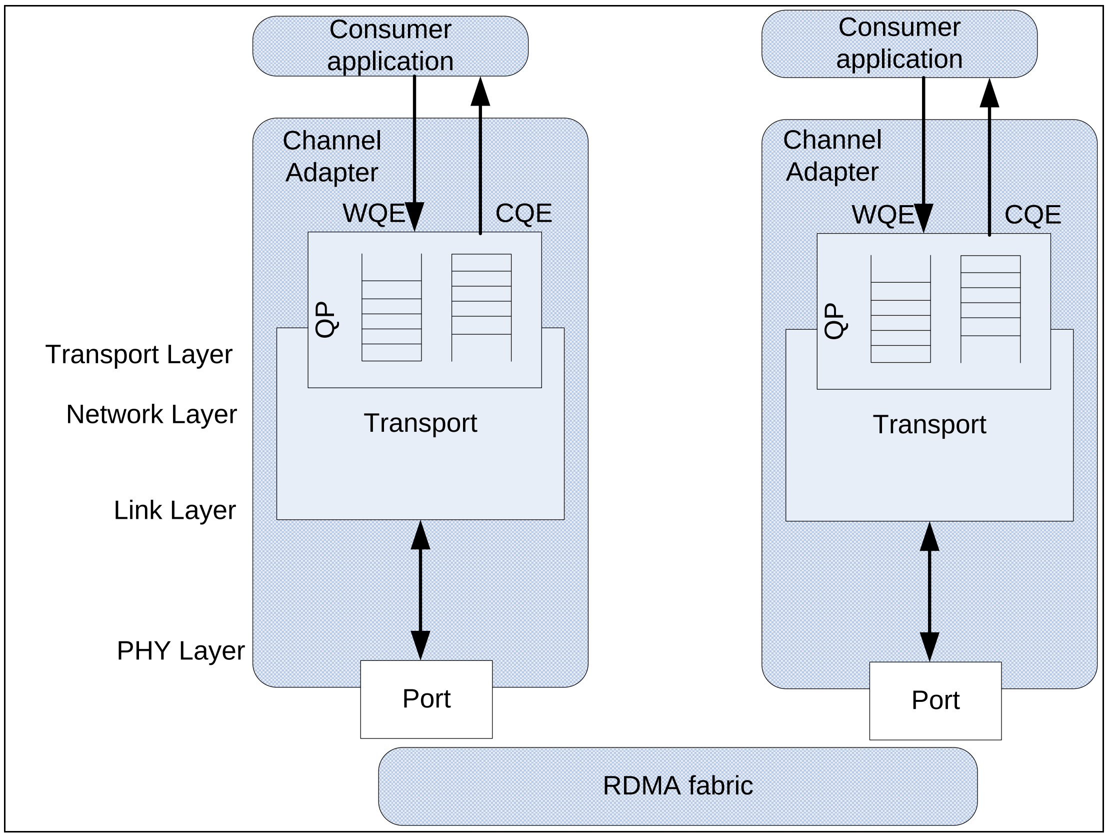

Figure 16: InfiniBand communication stack.

IV.D.4 SciCortex and other fabrics

SciCortex uses a custom fabric using a Kautz graph to connect 5,832 cores on 972 nodes of 6 cores each [49]. An important point of this architecture is that relatively low performance cores were used and the network hops between cores were optimized. Based on their HPC benchmark results, bimodal message size patterns of 128 bytes and 100 KB were found. In this case, PIO would be better for small messages with a CPU I/O engine support for large messages.

A popular fabric is the 3-dimension torus in the IBM Blue Gene/P to interconnect 36,864 nodes, each with 4 cores [50]. Each node has a DMA engine to communicate with other nodes, which is similar to the engine discussed in Section IV.A.3. The Blue Gene/P DMA engine services the core-to-core communications instead of serving general purpose I/O.

Table 6 summarizes how system interconnects and network I/O transactions can impact I/O transactions.

Table 6: System interconnect and network I/O transactions

System interconnects and networks	**Key benefits**	**Key costs**	**Example implementation**
CPU socket interconnect	Higher I/O throughput	I/O device complexity and memory coherency transaction overhead	Intel QPI and AMD Hyper-Transport
Messaging between cores	Higher I/O throughput and lower latency	Dedicated core silicon area and power	Intel SCC
RDMA InfiniBand and iWARP	Lower latency and reduced CPU overhead	Increased I/O device silicon area and power	Mellanox Infiniband I/O adapters
SciCortex fabric	Network optimized communication between nodes	Customized I/O interfaces with low volume	SciCortex clusters

V. POTENTIAL AREA OF PERFORMANCE IMPROVEMENT

The prior sections have shown that most I/O implementations rely on DMA engines on the I/O adapters to move network, storage and video data 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 as in the descriptor-based DMA method. As shown in Section III, the primary advantage of not relying on an I/O adapter's DMA engine and removing descriptor handling is that latency can be reduced. If the DMA engine of the I/O device can be integrated on the CPU, it is possible to reduce end-to-end latency by 18% and also reduce the size of the I/O device memory buffer.

Typical network traffic exhibits a bi-modal packet size with concentrations of small and large Ethernet frames [51]. Based on this assumption and projecting that small and large impacts can be averaged, our measurements summarized in Figure 7 show the potential benefits of removing descriptor related PCIe traffic. The 17% bandwidth reduction due to removal of descriptor, doorbell, and interrupt from the PCIe interface would allow for more payload transfers across the interface. Therefore, this section suggests a CPU-based *integrated DMA* (iDMA) engine and presents several benefits of this design.

The iDMA can 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, and would basically act as a heterogeneous or SoC core to the larger application generic cores. Figure 17 shows the placement of iDMA within a CPU. The arrow from the I/O adapter shows the iDMA pulling receive traffic from an I/O adapter into system memory. The arrows from system memory show the iDMA reading memory and pushing transmit traffic to the I/O adapter, and vice versa. 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.

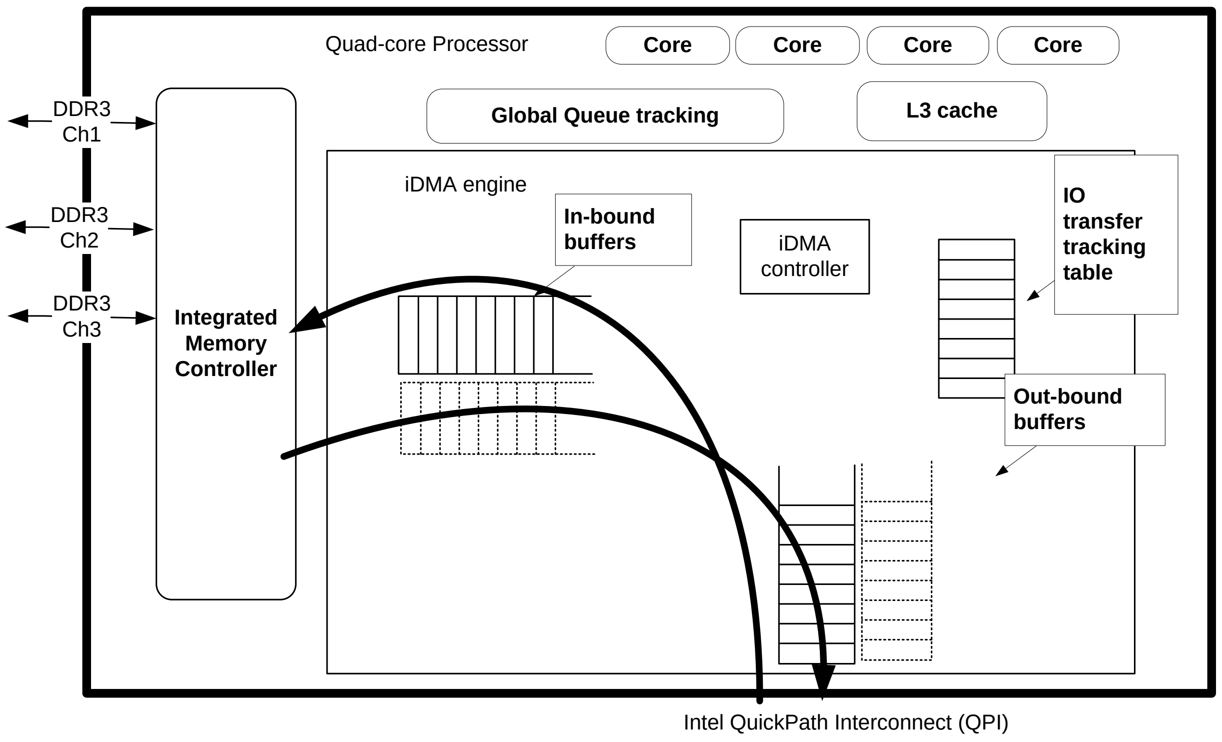

Figure17: An option for CPU integrated DMA.

Other less quantifiable improvements of removing descriptor-based DMA transactions include the following and are discussed in more detail by Larsen and Lee [52]. Potential performance improvements are tabulated in Table 7.

1. *Matched I/O and core/memory bandwidths* - In current systems, I/O bandwidth and core and memory bandwidths are not matched, and often systems can be found with too little I/O capability or inflexible in expanding I/O bandwidth and latency capabilities. Although there are a variety of reasons for the I/O mismatch to core and memory service capabilities, this problem is exacerbated by having multiple CPUs with possibly integrated IOHs. With iDMA, a dual-core processor could be matched with a lower bandwidth iDMA engine than a 4- or 8-core processor. Similarly, a platform that may need higher processor performance relative to I/O performance may need many cores with a lower capacity iDMA engine.
2. *Power management efficiency* - Mechanisms like RSS can spread I/O traffic across all available cores, thereby wasting system power that could be localized to a subset of cores when I/O traffic is low.
3. *System-level quality-of-service guarantees* - As I/O increases in bandwidth and complexity, having independent DMA engines contending for memory bandwidth reduces quality control. By having a central observation and control point for I/O transactions, the system can prioritize and deliver transactions more appropriately.
4. *Silicon cost and complexity* - Not having independent DMA engines servicing I/O transactions can reduce silicon cost and complexity.
5. *Security* – Security can be improved since iDMA would provide more control over which I/O device reads/writes from/to physical memory.

Table 7: Potential benefits for non-descriptor DMA.

Factor	Description	Measured Value	Descriptor Related Overhead	Comments/Justification
Latency	Latency to transmit a TCP/IP message between two systems	8.6 µs	18%	Descriptors are no longer latency critical
BW-per-pin	Gbps per serial link	2.1 Gbps/link	17%	Descriptors no longer consume chip-to-chip bandwidth
BW-right-sizing	Not quantifiable	N/A	N/A	Reduced platform silicon area and power
Power Efficiency	Normalized core power relative to maximum power	100%	71%	Power reduction due to more efficient core allocation for I/O
Quality of Service	Time required to control connection priority from software perspective	600 ns	92%	Round trip latency to queuing control reduced between PCIe and system memory
Multiple I/O Complexity	Die cost reduction	100%	> 50%	Silicon, power regulation and cooling cost reduction of multiple I/O controllers into a single iDMA instance
Security	Not quantifiable	N/A	N/A	Not quantifiable

VI. CONCLUSIONS

This survey of I/O methods and existing optimizations illustrates the historical development of I/O transactions for variety of computing systems. In simple systems, I/O is directly addressed by the processor either using PIO or DMA engines associated with the cores, which eliminates the need to set up descriptors to post DMA requests. However, to reduce the CPU management overhead, I/O transaction distribution using DMA engines on the I/O devices became the optimal system design. This DMA off-loading approach has continued as I/O transfer optimizations techniques surveyed continue to drive for I/O performance improvements. Based on our measurements and analysis of latency and throughput efficiency, there is an argument to re-visit processor-based I/O transactions suggesting quantifiable latency and bandwidth-per-pin advantages and potential benefits in power, QoS, silicon area, and security for future datacenter server architecture.

Table of Abbreviations

Abbreviation	Explanation
ARM	Acorn RISC Machine
BIOS	Basic Input Output System – allows access by the operating system to low-level hardware.
BW	BandWidth supported by an interface, usually synonymous with throughput capability.
CNI	Coherent Network Interface
CPU	Central Processing Unit – consisting of potentially multiple cores, each with one or more hardware threads of execution.
CRC	Cyclical Redundancy Check
CQE	Completion Queue Entry – used in RDMA to track transaction completions
DCA	Direct Cache Access
DDR	Double Data Rate – allows a slower clock to transmit twice the data per cycle. Usually based on both the rising and falling edge of a clock signal.
DDR3	3rd generation memory DDR interface

DLP	Data Layer Protocol in PCIe, which is similar to networking IP layer.
DMA	Direct Memory Access – allows read or write transactions with system memory.
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
FSB	Front Side Bus – a processor interface protocol that is replaced by Intel QPI and AMD Hyper-Transport
GbE	Gigabit Ethernet
GBps	Gigabytes per second
Gbps	Gigabits per second, (GBps x 8)
GHz	GigaHertz
GPU	Graphic Processing Unit
GOQ	Global Observation Queue
HPC	High Performance Computing – usually implies a high-speed interconnection of high-performance systems.
HW	HardWare
ICH	Intel I/O Controller Hub – interfaced to the IOH to support slower system protocols, such as USB and BIOS memory.
I/O	Input/Output
IOH	Intel I/O hub – interfaces between QPI and PCIe interfaces.
iWARP	Internet Wide Area RDMA Protocol – an RDMA protocol that supports lower level Ethernet protocol transactions.
KB	KiloByte, 1024 bytes. Sometimes reduced to “K” based on context.
L1 cache	Level 1 cache
L2 cache	Level 2 cache
LLC	Last Level cache - Level 3 cache
LCD	Liquid Crystal Display
LLI	Low Latency Interrupt
LLP	Link Layer Protocol – used PCIe
LRO	Large Receive Offloading
LSO	Large Segment Offload
MB	MegaBytes
MESI(F)	Modified, Exclusive, Shared, Invalid and optionally Forward – Protocol to maintain memory coherency between different CPUs in a system.
MFC	Memory Flow Controller – used to manage SPU DMA transactions.
MMIO	Memory Mapped I/O
MPI	Message Passing Interface – a protocol to pass messages between systems often used in HPC.
MTU	Maximum Transmission Unit
MSI	Message Signaled Interrupt – used in PCIe to interrupt a core.
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Architecture – allows multiple pools of memory to be shared between CPUs with a coherency protocol.
PCIe	Peripheral Component Interface express – defined at www.pcisig.com. Multiple lanes (1-16) of serial I/O traffic reaching 16Gbps per lane. Multiple generations of PCIe exist, represented by Gen1, Gen2, Gen3 and Gen4. PCIe protocol levels have similarities with networking ISO stack.
PHY	PHYsical interface defining the cable (fiber/copper) interfacing protocol.
PIO	Programmed I/O – often synonymous with MMIO
QDR	Quad Data Rate – allows four times the data rate based on a slower clock frequency.
QoS	Quality of Service – A metric to define guaranteed minimums of service quality.
QP	Queue Pair – transmit queue and receive queue structure in RDMA to allow interfacing between two or more systems.
QPI	QuickPath Interconnect – Intel’s proprietary CPU interface supporting MESI(F) memory coherence protocol.
RAID	Redundant Array of Inexpensive Disks

RDMA	Remote Direct Memory Access – used to access memory between two or more systems.
RSS	Receive Side Scaling
RTOS	Real Time Operating System
RX	Reception from a network to a system
SAS	Storage Array System
SCC	Single Chip Cloud
SCSI	Small Computer System Interface
SMT	Surface Mount Technology
SPE	Synergistic Processing Element in the Cell processor
SPU	Synergistic Processing Unit in Cell SPE.
SSD	Solid Stated Disk
SW	SoftWare
TCP/IP	Transmission Control Protocol and Internet Protocol networking stack.
TLP	Transaction Layer Protocol of PCIe stack
TOE	TCP/IP Offload Engine
TX	Transmission from a system to a network
USB	Universal Serial Bus
WQE	Work Queue Entry – used in RDMA to track transaction parameters.

ACKNOWLEDGEMENT

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

REFERENCES

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. *Intel 5520 chip-set datasheet*. 2010; Available from: http://www.intel.com/assets/pdf/datasheet/321328.pdf.
4. Harris, D. *Banks and Outsourcing: Just Say 'Latency'*. 2008; Available from: http://www.hpcwire.com/features/Banks_and_Outsourcing_Just_Say_Latency_HPCwire.html.
5. Top500. 2012; Available from: http://i.top500.org/stats.
6. *Intel 82598 10GbE NIC*. Available from: http://www.intel.com/assets/pdf/prodbrief/317796.pdf.
7. 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.
8. *Intel 82599 10GbE NIC*. Available from: http://download.intel.com/design/network/prodbrf/321731.pdf.
9. 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.
10. 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.
11. 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.
12. 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.
13. *Using DMA effectively*. Available from: http://www.embedded.com/columns/technicalinsights/196802092fire-questid=228429
14. *Intel QuickPath Interconnect*. Available from: http://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect.
15. Wikipedia. *RAID*. 2013; Available from: http://en.wikipedia.org/wiki/RAID.

16. *IOAT performance*. Available from: http://www.linuxfoundation.org/collaborate/workgroups/networking/i/oat.
17. 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.
18. *PCIe Base 3.0 Specification*. Available from: http://www.pcisig.com/specifications/pciexpress/base3/.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. Bardsley, A. and D. Edwards, *Synthesising an asynchronous DMA controller with balsa.* J. Syst. Archit., 2000. **46**(14): p. 1309-1319.
27. Aghdasi, F. and A. Bhasin. *DMA controller design using self-clocked methodology*. in *AFRICON, 2004. 7th AFRICON Conference in Africa*. 2004.
28. Markatos, E. and M. Katevenis. *User-level DMA without operating system kernel modification*. in *Third International Symposium on HighPerformance Computer Architecture*. 1997.
29. Blumrich, M.A., et al. *Protected, userlevel DMA for the SHRIMP network interface*. in *Second International Symposium on High-Performance Computer Architecture Proceedings*. 1996.
30. *Infiniband Architecture Specification Version 1.2.1*. 2008 2010]; Available from: http://members.infinibandta.org/kws/spec/.
31. Corporation, M. *Introduction to Receive Side Scaling*. 2012 [cited June 2012; Available from: http://msdn.microsoft.com/en-us/library/ff556942.aspx.
32. 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.
33. Regnier, G., et al., *TCP onloading for data center servers.* Computer, 2004. **37**(11): p. 48-58.
34. Ross, M., et al. *FX1000: a high performance single chip Gigabit Ethernet NIC*. 1997: IEEE.
35. *TCP Offload Engine*. Available from: http://en.wikipedia.org/wiki/TCP_Offioad_Engine
36. Bhoedjang, R.A.F., T. Ruhl, and H.E. Bal, *User-level network interface protocols.* Computer, 1998. **31**(11): p. 53-60.
37. Hill, M.D., et al. *Coherent network interfaces for fine-grain communication*. 1996: IEEE.
38. 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.
39. Chitlur, N., S. Larsen, Editor. 2011.
40. *IOAT description*. Available from: http://www.intel.com/network/connectivity/vtc_ioat.htm.
41. 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.
42. 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.
43. *HypterTransport IO Link Specification Rev3.10*. 2008; Available from: http://www.hypertransport.org/docs/twgdocs/HTC20051222-00046-0028.pdf.
44. *MESI protocol*. Available from: http://en.wikipedia.org/wiki/MESI_protocol

45. 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.
46. *Message Passing Interface*. [cited 2012 July 29 2012]; Available from: http://en.wikipedia.org/wiki/Message_Passing_Interface.
47. *Infiniband overview*. Available from: http://en.wikipedia.org/wiki/InfiniBand.
48. 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.
49. 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.
50. Barney, B. *Using the Dawn BP/P System*. 2012; Available from: https://computing.llnl.gov/tutorials/bgp/.
51. Hurtig, P.J., Wolfgang; Brunstrom, Anna, *Recent Trends in TCP Packet-Level Characteristics*, in *ICNS 2011*. 2011.
52. 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.