A New Server I/O Architecture for High Speed Networks

Guangdeng Liao[+], Xia Zhu[±], Laxmi Bhuyan[+]
[+]*University of California, Riverside*
[±]*Intel Labs*
{gliao, bhuyan}@cs.ucr.edu, xia.zhu@intel.com

Abstract

Traditional architectural designs are normally focused on CPUs and have been often decoupled from I/O considerations. They are inefficient for high-speed network processing with a bandwidth of 10Gbps and beyond. Long latency I/O interconnects on mainstream servers also substantially complicate the NIC designs. In this paper, we start with fine-grained driver and OS instrumentation to fully understand the network processing overhead over 10GbE on mainstream servers. We obtain several new findings: 1) besides data copy identified by previous works, the driver and buffer release are two unexpected major overheads (up to 54%); 2) the major source of the overheads is memory stalls and data relating to socket buffer (SKB) and page data structures are mainly responsible for the stalls; 3) prevailing platform optimizations like Direct Cache Access (DCA) are insufficient for addressing the network processing bottlenecks.

Motivated by the studies, we propose a new server I/O architecture where DMA descriptor management is shifted from NICs to an on-chip network engine (NEngine), and descriptors are extended with information about data incurring memory stalls. NEngine relies on data lookups and preloads data to eliminate the stalls during network processing. Moreover, NEngine implements efficient packet movement inside caches to address the remaining issues in data copy. The new architecture allows DMA engine to have very fast access to descriptors and keeps packets in CPU caches instead of NIC buffers, significantly simplifying NICs. Experimental results demonstrate that the new server I/O architecture improves the network processing efficiency by 47% and web server throughput by 14%, while substantially reducing the NIC hardware complexity.

1. Introduction

Ethernet continues to be the most widely used network architecture today for its low cost and backward compatibility with the existing Ethernet infrastructure. It dominates in data centers and is replacing specialized fabrics such as InfiniBand [12], Quadrics [32], Myrinet [4] and Fiber Channel [6] in high performance computers. Driven by increasing networking demands such as Internet search, web hosting, video on demand etc, network speed is rapidly migrating from 1Gbps to 10Gbps and beyond [7]. High speed networks require servers to provide efficient network processing with a low design complexity of network interfaces (NIC).

Traditional architectural designs of processors, cache hierarchies and system interconnect focus on CPU/memory-intensive applications, and are usually decoupled from I/O considerations. They are inefficient for network processing. It was reported that network processing in the receive side over 10Gbps Ethernet network (10GbE) easily saturates two cores of an Intel Xeon Quad-Core processor [19, 22]. Assuming ideal scalability over multiple cores, network processing over 40GbE and 100GbE will saturate 8 and 20 cores, respectively. In addition to the processing inefficiency, the increasing network speed also poses a big challenge to NIC designs. DMA descriptor fetches over long latency PCI-E bus heavily stress the DMA engine in NICs and need larger NIC buffers to temporarily keep packets. These requirements significantly increase NIC's design complexity and price [37]. For instance, the price of a 10GbE NIC can be up to $1.4K but a 1GbE NIC costs less than $40 [13, 14]. Our aim is to understand network processing efficiency over high speed networks and design a new server I/O architecture to tackle those challenges.

In the past decade, a wide spectrum of research has been done in network processing to understand and optimize the processing efficiency [1-3, 9, 16-22, 25, 27-28, 38]. Zhao *et al.* [38] used a cache simulator to study cache behavior of the TCP/IP protocol. They showed that packets show no temporal locality and proposed a copy engine to move packets in memory. To eliminate memory stalls to packets, Intel proposed DCA to route incoming network data to caches [9, 18-19]. Furthermore, Binkert *et al.* [2-3] integrated a simplified NIC into CPUs to naturally implement DCA and used Program I/O (PIO) to move data from NICs in software. We also did extensive performance evaluation of an Integrated NIC (INIC) architecture [20] and along with Intel researchers, suggested some

978-1-4244-9435-4/11/$26.00 ©2011 IEEE

techniques to improve the network processing performance [21-22]. Recently, Intel proposed on-die message engines to move data between I/O devices and CPUs to reduce the PCI-E traffic [17]. However, all these works did not study overheads at the operating system (OS) level (e.g. NIC driver, buffer management etc) and address those bottlenecks.

In this paper, we begin with per-packet processing overhead breakdown by running a network benchmark Iperf [10] over 10GbE on Intel Xeon Quad-Core processor based servers. We find that besides data copy, the driver and buffer release, unexpectedly take 46% of processing time for large I/O sizes and even 54% for small I/O sizes. To understand the overheads, we instrument the driver and OS kernel using hardware performance counters [11]. Unlike existing profiling tools attributing CPU cost, such as retired cycles or cache misses, to function level [29], our instrumentation is at very fine granularity and can pinpoint data incurring the cost. Through the above studies, we obtain several new findings: 1) the major network processing bottlenecks lie in the driver (>26%), data copy (up to 34% depending on I/O sizes) and buffer release (>20%), rather than the TCP/IP protocol itself; 2) in contrast to the generally accepted notion that long latency NIC register access results in the driver overhead [2-3], our results show that the overhead comes from memory stalls to network buffer data structures; 3) releasing network buffers in OS results in memory stalls to in-kernel page data structures, contributing to the buffer release overhead; 4) besides memory stalls to packets, data copy implemented as a series of load/store instructions, also has significant time on L1 cache misses and instruction execution. Prevailing optimizations for data copy like DCA are insufficient to address the copy issue.

Based on the studies, we discuss several intuitive solutions and find that a holistic I/O solution is needed for high speed networks. We propose a new server I/O architecture, where the responsibility for managing DMA descriptors is moved to an on-chip network engine (NEngine). The on-chip descriptor management exposes plenty of optimization opportunities like extending descriptors. We add information about data incurring memory stalls during network processing into descriptors. When the NIC receives a packet, it directly pushes the packet into NEngine without waiting for long latency descriptors fetches. NEngine reads extended descriptors to obtain packet destination location and information about data incurring memory stalls. Then, it moves the packet into the destination memory location and checks whether data incurring the stalls resides in caches. If not, NEngine sends data address to the hardware prefetching logic for loading the data. To address the data copy issue, NEngine moves payload inside last level cache (LLC) and invalidates source cache lines after the movement. The new I/O architecture allows DMA engine to have fast access to descriptors and keeps packets in CPU caches rather than in NIC buffers. These designs substantially reduce the burden on the DMA engine and avoid extensive NIC buffers in high speed networks. While NICs are decoupled from DMA engine, they still maintain other hardware features such as Receive Side Scaling (RSS) [33], and Interrupt Coalescing [7]. Different from previous research aiming at data copy, the new server I/O architecture ameliorates all major performance bottlenecks of network processing and simplifies NIC designs, making general purpose platforms well suited for high speed networks.

To evaluate our designs, we enhanced the full system simulator Simics [24] with detailed timing models and implemented the new I/O architecture in the simulator. We developed a 10GbE NIC as a device module of the simulator and a corresponding driver with the support of Large Receive Offload (LRO) [8] in Linux. In the experiments, both the micro-benchmark Iperf and the macro-benchmark SPECWeb [34] are used. Experimental results demonstrate that the new I/O architecture improves the network processing efficiency by 47% and web server throughput by 14% while substantially reducing the NIC hardware complexity.

The remainder of this paper is organized as follows. We revisit the conventional I/O architecture in Section 2 and then present a detailed overhead analysis over 10GbE in Section 3. Section 4 elaborates the new server I/O architecture in detail, followed by performance evaluation in Section 5. Finally we discuss related work and conclude our paper in Section 6 and 7, respectively.

2. Revisiting I/O Architecture

2.1 Network Processing

Unlike CPU-intensive applications, network processing is I/O-intensive and involves several hardware components (e.g. NIC, PCI-E, I/O Hub, memory, CPU) and system components (e.g. NIC driver, TCP/IP). Network processing in the receive side has significant processing overheads, consuming thousands of CPU cycles for each packet. In this subsection, we revisit the network receiving process.

In the receive side, an incoming packet starts with the NIC/driver interaction. The RX descriptors (typically 16 bytes each), organized in circular rings, are used as a communication channel between the driver and the NIC. The driver tells the NIC through these descriptors, where in the memory to copy the incoming packets. To be able to receive a packet, a

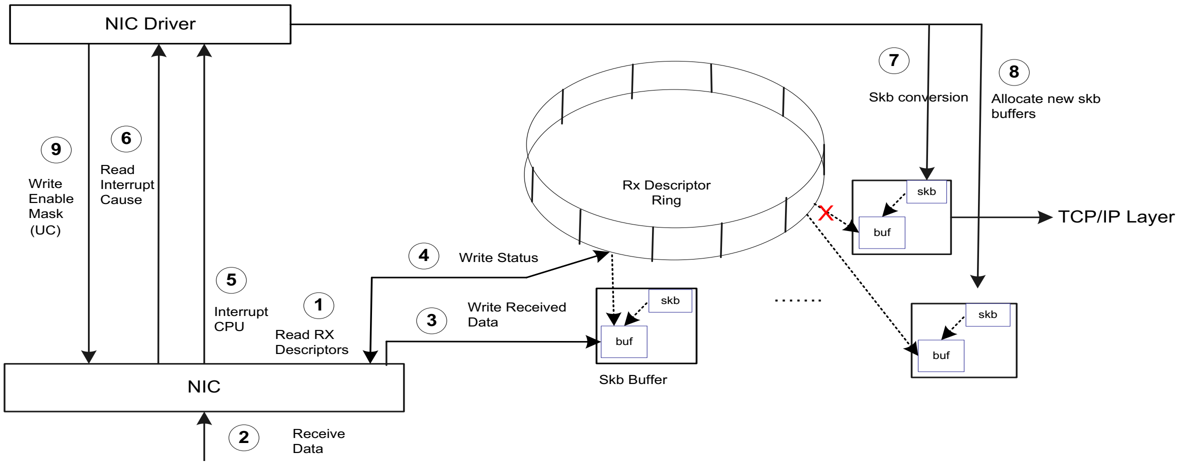

Figure 1. NIC/Driver interaction

descriptor should be in "ready" state, which means it has been initialized and pre-allocated with an empty packet buffer (SKB buffer in Linux) accessible by the NIC [36]. The SKB buffer is the in-kernel network buffer to hold any packet up to MTU (1.5 KB). It contains an SKB data structure of 240 bytes carrying packet metadata used by the TCP/IP protocol and a DMA buffer of 2 KB holding the packet itself.

The detailed interaction is illustrated in Figure 1. Before transferring the received packets, the NIC needs to read ready descriptors from memory over PCI-E bus to know the DMA buffer address (step 1). After receiving the Ethernet frames from the network (step 2), it transfers the received packets into those buffers (denoted as *buf* in Fig.1) using DMA engine (step 3). Once the data is placed in memory, the NIC updates descriptors with packet length and marks them as used (step 4). Then, the NIC generates an interrupt to kick off network processing in CPUs (step 5). Note that several interrupts can be coalesced into one interrupt to reduce overheads. In the CPU side, the interrupt handler in the driver reads the NIC register to check the cause of the interrupt (in step 6). If legal, the driver reads descriptors to obtain packet's address and length, and then maps the packet into SKB data structures (step 7). After the driver delivers SKB buffers to the protocol stack, it reinitializes and refills used descriptors with new allocated SKB buffers for future incoming packets (step 8). Finally, the driver re-enables the interrupt by setting the NIC register (step 9). After the driver completes its operations, SKB buffers are delivered up to the protocol stack. Once the protocol stack finishes processing, applications are scheduled to move packets to user buffers. Finally, the SKB buffers are reclaimed into OS [5, 36].

2.2 High Speed NIC Challenges

As shown in Fig.1 (step 1), the NIC must initiate DMA transfers over PCI-E bus to fetch descriptors from memory before it writes packets into memory (or reads packets from memory in the transmit side). Although PCI-E bus bandwidth has improved, its latency has worsened by up to 25X over earlier PCI-X incarnations. The round-trip traversal over PCI-E bus can take up to ~2200 *ns*, mostly due to complex PCI-E transaction layer protocol implementation [26]. The long latency traversal substantially increases the processing overhead of DMA engine. As network traffic becomes intense, the DMA engine is heavily stressed [37]. Long latency descriptor fetches also require large NIC buffers to temporarily store packets. Moreover, in order to leverage multiple cores in CMPs for packet processing, modern NICs typically introduce a large number of receive/transmit (RX/TX) queues and allow each core to have a dedicated RX/TX queue. For instance, an Intel 82599 10GbE NIC has 128 RX/TX queues per port, corresponding to 512KB and 160KB buffers [14]. All of these complicate NIC designs and pose big challenges.

3. Understanding Processing Overheads

We conducted extensive experiments to understand network processing overheads over 10GbE across a range of I/O sizes. Both SUT (System under Test) and stress machines are Intel servers, which contain two Quad-Core Intel Xeon 5355 processors [11]. Each core is running at 2.66GHz frequency and each processor has 2 LLC of 4MB each shared by 2 cores. The servers are connected by two Intel 10Gbps 82598 server adapters sitting in PCI-E X8 slots [13]. They ran Linux kernel 2.6.21 and Intel 10GbE NIC driver IXGBE version 1.3.31. We retain default settings of the Linux network subsystem and the NIC driver, unless stated otherwise. Note that LRO [8], a technique to amortize the per-packet processing overhead by combining multiple in-order packets into one single packet, is enabled in the NIC driver. Popular stream hardware

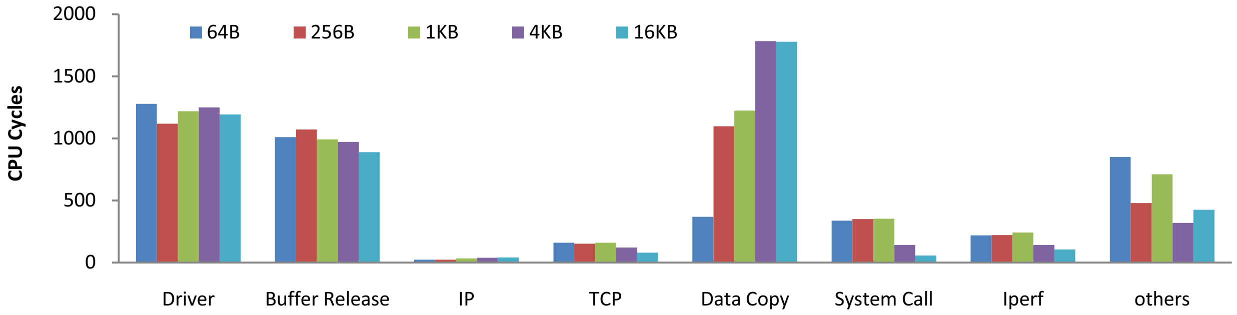

Figure 2. Per-packet processing overhead breakdown

prefetcher employing a memory access stride based predictive algorithm is configured in the servers [11]. In our experiments, the micro-benchmark Iperf with 8 TCP connections is run to generate network traffic between servers (SUT is a receiver). We find that one core with 4MB LLC achieves ~5.6Gbps throughput and two cores with 8MB LLC are saturated in order to obtain line rate throughput. The high processing overhead motivates us to further breakdown the per-packet processing overhead.

3.1. Packet Processing Overhead Breakdown

We used Oprofile [29] to collect system-wide function overheads while Iperf is running over 10GbE. We group all functions into components along the network processing path: the driver, IP, TCP, data copy, buffer release, system call and Iperf. All other supportive kernel functions such as scheduling, context switches etc. are categorized as others. Per-packet processing time breakdown under various I/O sizes is calculated and illustrated in Figure 2. Note that I/O size is used by TCP/IP stack and I/Os larger than MTU are segmented into several Ethernet packets (<=MTU).

We observe the following from Fig.2: 1) the overhead in data copy increases as the I/O size grows and becomes a major bottleneck with large I/Os (>=256 bytes); 2) the driver and buffer release consume ~1200 cycles and ~1100 cycles per packet, respectively, regardless of I/O sizes. They correspond to ~26% and 20% of processing time for large I/Os and even higher for small I/Os; 3) the TCP/IP protocol processing overhead is substantially reduced because LRO coalesces multiple packets into one large packet to amortize the processing overhead. Thus Fig.2 reveals that, besides data copy, high speed network processing over mainstream servers has another two unexpected major bottlenecks: the driver and buffer release.

3.2. Fine-Grained Instrumentation

Oprofile in Subsection 3.1 does profiling at the coarse-grained level and attributes CPU cost, such as retired cycles and cache misses, to functions. It is unable to identify data or macro incurring the cost, so we had to manually instrument inside functions. Table 1 shows one such example of instrumentation in the driver. We first measured the function's cost and then did fine-grained instrumentation for every code segment if the function has high cost. We continue to instrument each code segment until we locate the bottlenecks in all functions along the processing path. Most events are collected including CPU cycles, instruction and data cache misses, LLC misses, ITLB misses and DTLB misses etc. Since large I/Os include all three major overheads, this subsection presents the detailed analysis only for the 16KB I/O.

Table1. Fine-grained instrumentation

Coarse-grain	Fine-grain

INSTRUMENT(Counter1)
Ixgbe_clean_rx_irq()
INSTRUMENT(Counter2)
	Ixgbe_clean_rx_irq()
	{
	INSTRUMENT(Counter3)
	Code segment 1
	INSTRUMENT(Counter4)
	Prefetch(skb->data-NET_IP_ALIGN);
	INSTRUMENT(Counter5)
	....
	INSTRUMENT(Counter6)

3.2.1. Driver

The driver comprises of three main components: NIC register access (step 6 and 9), SKB conversion (step 7) and SKB buffer allocation (step 8), as shown in Fig.1. Existing studies [2-3] claimed that NIC register access contributes to the driver overhead due to long latency traversal over PCI-E bus, and then proposed NIC integration to reduce the overhead. In this subsection, we architecturally breakdown the driver overhead for each packet and present results in Figure 3. In contrast to the generally accepted notion that the long latency NIC register access results in the overhead [3], the breakdown reveals that the overhead comes from SKB conversion and buffer allocation. Although NIC register access takes ~2500 CPU cycles on mainstream servers, ~60 packets are processed per interrupt over 10GbE (~7 packets/interrupt over 1GbE) substantially amortizing the overhead. In addition, Fig.3 also reveals that L2 cache misses mainly result in the SKB conversion overhead, and long instruction path is the largest contributor of the SKB buffer allocation overhead.

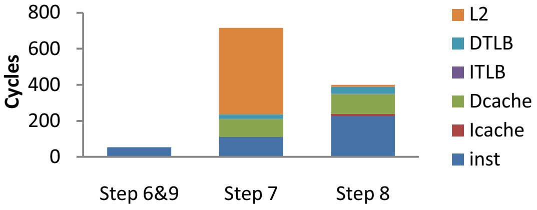

Figure 3. Architectural breakdown

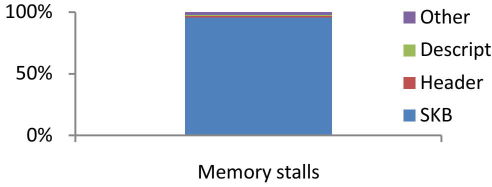

Figure 4. L2 cache misses breakdown for step 7

Since L2 cache misses in the SKB conversion constitute ~50% of the driver overhead, we did further instrumentation to identify the data incurring those misses. We group data in the driver into various data types (SKB, descriptors, packet headers and other local variables) and measure their percentage of misses. The result, presented in Figure 4, reveals that SKB is the major source of the memory stalls (~1.5 L2 misses/packet on SKB). Different from the results in prior studies [2-3], we find that the memory stalls to packet headers are hidden and overlapped with computation. It is because the new drivers use software prefetch instructions to preload headers before being accessed. Unfortunately, SKB access occurs at the very beginning of the driver and software prefetch instructions cannot help. Although DMA invalidates descriptors to maintain cache coherence, the memory stalls to descriptors are negligible (~0.04 L2 misses/packet). That is because each 64 bytes cache line can host 4 descriptors of 16 bytes each, and the hardware prefetchers preload several consecutive descriptors with a cache miss. To understand the SKB misses, we also instrumented the kernel to study its reuse distance over 10GbE. It is observed that SKB has long reuse distance (~240K L2 access), which explains the miss behavior.

3.2.2. Data Copy

After protocol processing, user applications are scheduled to copy packets from SKB buffers to user buffers. Data copy incurs mandatory cache misses on payload because DMA triggers cache invalidation to maintain cache coherence. We study the architectural overhead breakdown of data copy and show the results in Figure 5. 16KB I/O is segmented into small packets of MTU each in the sender and they are sent to the receiver. Fig.5 shows that L2 cache misses are the major overhead (~50%, ~3.5 L2 misses/packet), followed by data cache misses (~27%, ~50 misses/packet) and instruction execution (~20%). Although DCA, implemented in recent Intel platforms avoids L2 cache misses, it is unable to reduce overheads of L1 cache misses and a series of load/store instruction executions (total ~47%). Also, routing network data into L1 caches would pollute caches and degrade performance because of small L1 cache size [19, 35]. Since packets become dead after data copy [36], loading them into L1 caches may evict other valuable data. Hence, more optimizations are needed to fully address the data copy issue.

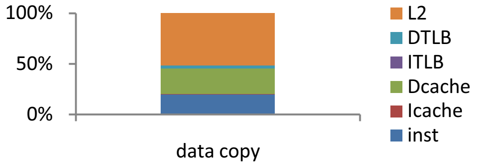

Figure 5. Data copy overhead breakdown

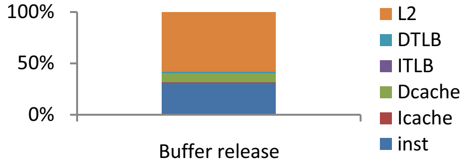

Figure 6. Buffer release overhead breakdown

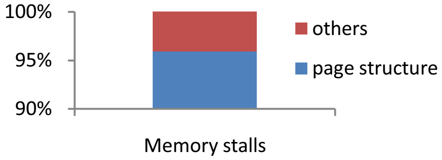

Figure 7. L2 cache misses breakdown

3.2.3. Buffer Release

SKB buffers need to be reclaimed after packets are copied to user applications. SKB buffer allocation and release are managed by slab allocator [5]. The basis for this allocator is retaining an allocated memory that used to contain a data object of certain type and reusing that memory for the next allocations for another object of the same type. Buffer release consists of two phases: looking up an object buffer controller in OS and releasing the object into the controller. In the implementation of slab allocator, the page data structure is used to keep buffer controller information and read during the object lookup. This technique is widely used by mainstream OS such as FreeBSD, Solaris and Linux etc.

Figure 6 shows the architectural overhead breakdown of buffer release. We observe from Fig.6

that L2 cache misses are the single largest contributor to the overhead (~1.6 L2 cache misses/ packet). Similarly, we analyze data sources of L2 cache misses and present results in Figure 7. The figure reveals that L2 cache misses are from 128 bytes in-kernel page data structures. The structure reuse distance analysis shows that it is reused after ~255K L2 cache access, which results in the large cache misses.

The above studies reveal that besides memory stalls to itself, each packet incurs several cache misses on corresponding data (skb buffer and page data structures) and has considerable data copy overhead. Some intuitive solutions like having larger LLC (>8MB for 10GbE) or extending the optimization DCA might help to some extent. Our simulation results show that, without considering application memory footprint, 16MB LLC is needed to avoid those cache misses for packet processing over 10GbE. When network jumps to 40GbE and beyond, increasing LLC becomes an ineffective solution. More importantly, it is unable to address NIC challenges and the data copy issue. Extending DCA to deliver both packets and those missed data from NICs into caches is more efficient in avoiding memory stalls. Unfortunately, it stresses NICs more heavily and degrades PCI-E efficiency of packet transfers [30-31], and does not consider the data copy issue as well. In order to attack all challenges from increasing network speed, a new holistic I/O solution is needed.

4. Proposed Server I/O Architecture

In this section, we propose a new server I/O architecture for high speed networks. The overview of architecture is illustrated in Figure 8. Essentially, we move the DMA descriptor management from NICs to an added on-chip network engine (NEngine) close to LLC. The on-chip descriptor management enables us to easily extend descriptors with information about data incurring memory stalls. Similar to the memory controller, the NEngine connects to I/O Hub (IOH) for parsing PCI-E transactions. It communicates with faster cache hierarchy for DMA descriptor fetches/writes and packet movement, alleviating the processing burden on the DMA engine. The NEngine has low communication cost with LLC due to its close proximity.

When NEngine receives a packet, it reads descriptors from cache hierarchy. Then it moves the packet into corresponding cache location and preloads those data incurring memory stalls. The new architecture exploits LLC to keep packets instead of multiple RX/TX queues in NICs. Modern high speed NICs have one dedicated RX/TX queue for each core, thus increasing their cost and impeding scalability over a large number of cores. Moreover, NEngine also implements efficient payload movement inside LLC and proactively purges dead packet data after data copy is finished to address the data copy issue. The new I/O architecture fundamentally reduces all three major performance bottlenecks of network processing while effectively simplifying NICs. The proposed designs are elaborated in the following subsections.

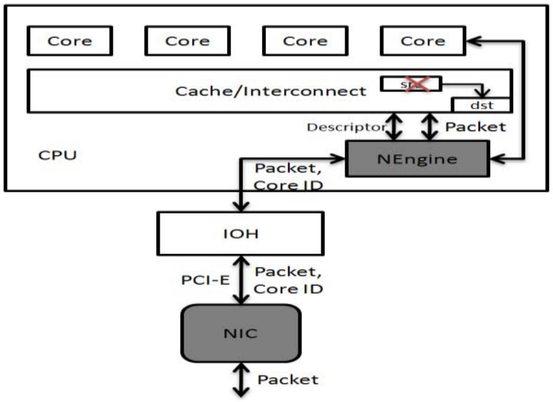

Figure 8. New server I/O architecture

4.1. NEngine

During network processing, CPUs and NICs communicate through DMA descriptors, as described in detail in Section 2. Descriptors are organized as a circular ring. Each descriptor is 16 bytes long and includes packet metadata such as packet length, memory address and status etc. In the contemporary I/O architecture, NICs fetch or write descriptors via PCI-E bus before or after packet movement. The descriptor fetches/writes have long latency stressing DMA engine [37] and also waste a large number of PCI-E transactions degrading PCI-E payload efficiency [30-31]. Our on-chip descriptors management scheme avoids these issues and more importantly, enables us to easily extend the descriptors due to faster communication with the cache hierarchy. By exploiting this design, we extend RX descriptors with information about data incurring memory stalls: SKB and page data structures, as pinpointed in Subsection 3.2. The extended descriptors are illustrated in Figure 9. Besides original 16 bytes, each new descriptor includes 4 bytes physical address of SKB and page data structures each. Two hardware registers in NEngine are dedicated to storing data structure length in terms of the number of cache lines. For example in Linux, SKB is 240 bytes and page structure is 128 bytes, corresponding to four and two cache lines of 64 bytes each. For a typical ring buffer size of 1024 entries in 10GbE NICs, the new ring buffer size only increases by 8KB.

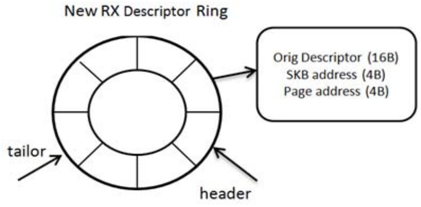

Figure 9. Extended descriptors

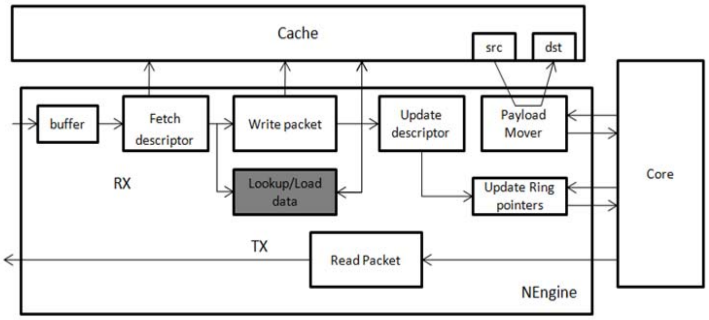

Figure 10. Basic blocks of NEngine

The block diagram of NEngine is illustrated in Figure 10 with the new descriptors. Besides major components shown in Fig. 10, NEngine also offers dedicated registers to keep ring buffer base address and ring pointer information as traditional NICs do. When a packet arrives at the NIC, without fetching DMA descriptors to know memory location for the packet, the NIC calculates core ID for packet processing using RSS hardware unit (RSS hardware distributes packets among cores by hashing packet's 4-tuple) and sends the packet with core ID into a small buffer in NEngine. Fetch descriptor unit identifies the corresponding descriptor address according to the ring base address of the core ID and ring buffer pointers, and then sends a cache read request to get the descriptor. Sec.3 shows that mainstream severs exhibit extremely high descriptor cache hit ratios even with DMA invalidation (96%). The on-chip descriptor management avoids DMA invalidation and has a higher descriptor cache hit ratio. Thus, the fetch descriptor unit can access the descriptors very fast and is much simpler than the original DMA engine. With the knowledge of memory location and data incurring memory stalls, the write packet unit moves the packet into caches. Meanwhile, the lookup/load unit lookups and preloads those data. To facilitate the unit, we extend the cache architecture with a new cache operation: *lookup*. The new operation *lookup* returns whether data is in caches, other than data themselves. The lookup/load unit sends *lookup* operations to lookup those data and generates prefetch commands to the hardware prefetching logic if they are not in caches. After the packet is moved into cache hierarchy, NEngine updates the descriptor status field and ring buffer pointers similar to the traditional NICs.

In addition, NEngine is capable of moving payload inside LLC. Since the source data becomes dead after data copy [36], NEngine invalidates source cache lines to purge the data. To support efficient movement, we extend the cache architecture with a new cache operation: *read_invalidate*, which reads cache lines and then does cache invalidation. During data copy, TCP/IP protocol breaks discontinuous physical address ranges into a set of consecutive physical ranges and programs NEngine via three hardware registers: src, dst, len. Then, NEngine breaks continuous physical address ranges into a set of chunks at the cache line granularity and generates new *read_invalidate* operations to read and invalidate cache lines. Finally, it writes those data into destination cache lines. Our payload movement differs from prior copy engines [1, 16, 38] as follows: 1) payload movement is done inside caches as opposed to memory in previous cases, and payload in caches is invalidated after movement. The invalidation avoids unnecessary memory write-backs of dirty data, reducing memory traffic and improving performance [15, 23]; 2) the virtual-to-physical address translation overhead is negligible because data copy is done in the OS context. In Linux, less than 10 cycles are needed for the address translation.

When we come to the transmit side, NEngine reads transmitted packets from cache hierarchy and transfers them into the NIC over PCI-E bus. Once the NIC receives the transmitted packets from NEngine, the MAC processing units automatically sends them over Ethernet links. Besides efficient network processing, our designs simplify NIC designs in terms of buffer resource and DMA engine and also reduce PCI-E traffic used for descriptor fetches/writes.

4.2. NIC

In the new architecture, NICs are simplified with less hardware resource. Figure 11 illustrates a traditional NIC in the left box and the new NIC in the right box. In the traditional NIC, the MAC processing unit receives packets from Ethernet and does RSS to load balance incoming packets among cores at the connection level. The packets are stored in corresponding RX queues. DMA engine uses PCI-E transactions to fetch descriptors from memory and to move data from RX queues to memory. Interrupt coalescing unit will send interrupts to cores when the number of transferred packets reach up to a threshold set by the driver or a preprogrammed timer expires. Similarly, in order to transmit packets, the NIC fetches

TX descriptors to know packet memory location and moves packets into corresponding TX queues. Then, packets are sent over Ethernet and interrupts are sent to cores. In the new NIC, we remove large multiple hardware queues and DMA engine marked as grey in the left box. When RSS receives a packet from the MAC processing unit, it calculates the core assigned to packet processing. Then, the NIC directly sends the packet with core ID to NEngine. Similar to the receive side, when the NIC receives a transmitted packet, the MAC processing unit directly takes over the packet for transmission. RSS and Interrupt coalescing units behave the same as the traditional NICs.

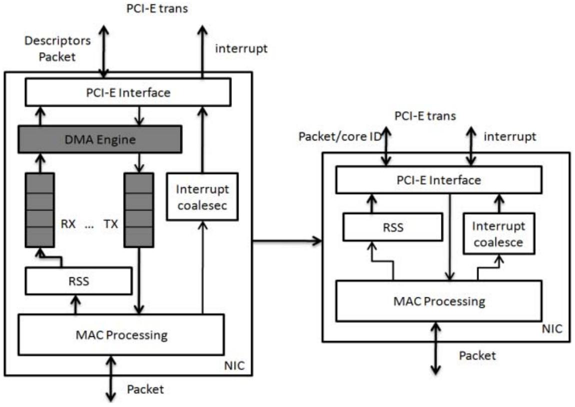

Figure 11. Simplification of the NIC

4.3. Software Support

The new server I/O architecture inherits the descriptor-based software/hardware interface, and only needs some modest support from the device driver and the data copy components. When new SKB buffers are allocated to refill RX descriptors, the driver sets starting address of SKB and page data structures in descriptors in addition to DMA buffer address. When packets finish protocol processing, the data copy component programs NEngine to move payload and waits until NEngine finishes the movement. There is no need to modify TCP/IP protocol stack, system call and user applications.

5. Evaluation

We choose the full system simulator Simics to evaluate our designs by enhancing it with detailed cache, I/O timing models and effects of network DMA. We extend the DEC 21140A Ethernet device with the support of interrupt coalescing using Device Modeling language to simulate a 10GbE Ethernet NIC. The device itself is connected to a lossless, full-duplex link of configurable bandwidth. The latency of a packet traversing the link is simply fixed to 1 *us*. Two systems (client and server) running Linux 2.6.16 are simulated and interconnected with 10GbE. Since the stream hardware prefetcher is the most popular prefetcher in servers, we employ it in the simulator to speed up the memory access of network data.

We implemented the new I/O architecture and developed a NIC driver in Linux. LRO was implemented in the driver. To understand performance impacts of our designs on network processing, we first used the micro-benchmark Iperf. Then, we study how much benefit web servers achieve by running the SPECWeb benchmark. In each case, only one system is of interest, while the other merely serves as a stressor. SUT is configured with detailed timing models and the stressor runs with the fast functional mode and is not a bottleneck. The parameters we used in modeling the configuration are listed in Table 2. We are more interested in the relative behavior of these systems than their absolute performance, so some of these parameters are approximate.

Table 2. System configurations

Processor	Quad-Core, 3GHz, two-issue, in-order
ICache/DCache	32KB 2-way, 3 cycles hit, 64 bytes cache line
L2 Cache	8M, 16-way, 14 cycles hit, 64 bytes cache line, shared by all cores
Main memory	400 cycles
Prefetcher	Stream prefetch with degree 4
I/O Register	1600 cycles
Interrupt rate	64 packets per interrupt
NEngine	10 cycles to L2 cache
Ring buffer	1024 entries/ring

5.1. Network Performance

First, we looked at network performance in the receive side by running Iperf under various configurations: the original system (*orig*), DCA routing data to L1 caches (*DCA-L1*), DCA routing data into L2 caches (*DCA-L2*), and the new server I/O architecture (*new*). LRO is included in all configurations. Since large I/Os have all three major overheads, we present large I/O results in this subsection.

Figure 12 illustrates network throughput achieved by various configurations. We also present corresponding core utilization and utilization breakdown in Figure 13. As shown in the figures, *orig* can achieve only ~8 Gbps throughput by consuming ~225% core utilization in the SUT with four cores. Memory subsystem is the potential bottleneck of achieving line rate throughput and an increase in CPU performance could not further improve throughput. We observe from Fig. 13 that data copy, the NIC driver and buffer release are three major overheads. By injecting network data into L1 caches, *DCA-L1* eliminates the memory stalls to packets and obtains line rate throughput using ~200% core utilization. Utilization breakdown reveals that the higher network processing efficiency or throughput/core is from CPU cycle savings in data copy. Instead of L1 caches, *DCA-L2* routes network data into a larger L2 cache. It achieves line rate throughput and consumes fewer CPU cycles than *DCA-L1*. That is because *DCA-L1* delivers ~64 packets or ~96KB data for each interrupt into small L1 caches of 32 KB each, incurring cache pollution. With high speed networks like 10GbE and beyond, *DCA-L2* is a more practical approach.

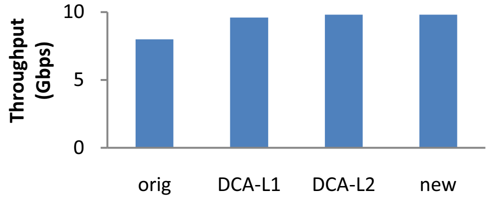

Figure 12. Network throughput

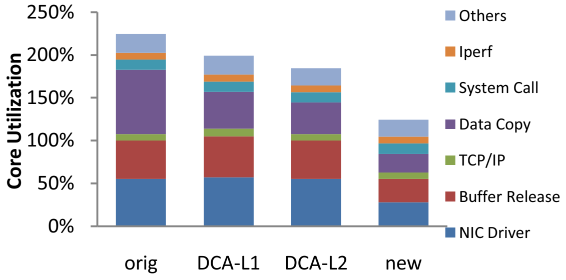

Figure 13. Utilization breakdown

Although DCA is able to reduce the data copy overhead, it is unable to resolve the performance issues in other components such as the driver and buffer release. The new I/O architecture not only avoids memory stalls in the driver and buffer release, but also further improves data copy performance. Fig. 12 and Fig.13 show that it obtains line rate throughput but substantially reduces core utilization to ~125%. The utilization breakdown confirms that the reduction is from the driver, buffer release and data copy. Compared to *DCA-L2* which is employed in recent commercial servers, the new I/O architecture reduces core utilization by 33%, corresponding to 47% network processing efficiency improvement. The reduced core utilization or higher processing efficiency means that the cores can be better used for application processing instead of network processing.

Additionally, we also investigate cache behavior of high speed network processing under various configurations in Figure 14. We observe that *orig* only achieves a 92% L2 cache hit ratio. By avoiding the memory stalls to packets, both *DCA-L1* and *DCA-L2* increase L2 cache hit ratios to 96%. The new architecture almost avoids memory stalls during network processing and escalates the L2 cache hit ratio to 99%. The higher L2 cache hit ratio explains the benefits of core utilization shown in Fig.13. All configurations achieve similar hit ratios in L1 cache except *DCA-L1* and *new*. Due to small cache sizes, *DCA-L1* results in L1 cache pollution and decreases the L1 cache hit ratio. *New* bypasses L1 caches during data copy and has a higher L1 cache hit ratio. We do not present results for the sender side because performance is not significantly improved.

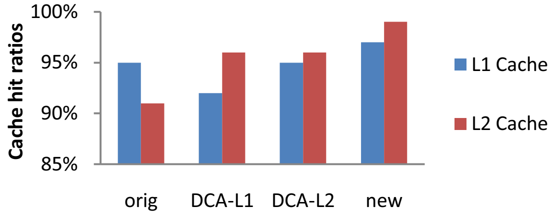

Figure 14. Cache hit ratios

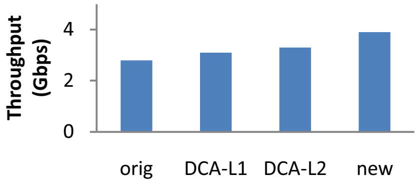

Figure 15. Web server throughput

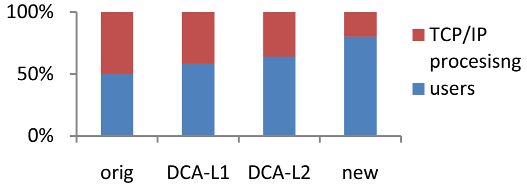

Figure 16. Utilization breakdown

5.2. Web Server Performance

We also studied web server performance by running the web server benchmark SPECweb99 over 10GbE. The same configurations as subsection 5.1 were used. Web server throughput with various configurations is illustrated in Figure 15, where the server achieves ~2.8Gbps, ~3.1Gbps and ~3.3Gbps throughput in *orig*, *DCA-L1* and *DCA-L2*. CPU utilization breakdown in Figure 16 reveals that throughput increases are from the CPU cycle savings in network processing. In the new architecture, the network processing overhead is further reduced due to the elimination of the memory stalls and more efficient data copy. The improved network processing translates up to ~3.8Gbps server throughput, 14% better than *DCA-L2*.

5.3. NIC Design Benefits

Besides having efficient network processing, the new server I/O architecture also simplifies NIC hardware designs by lessening pressure on DMA engine and avoiding extensive NIC buffers. We measure round-trip time over PCI-E bus on mainstream servers and assume that each PCI-E transaction (typically, 256 bytes transaction size) transfers 16 descriptors. We obtain average per packet time for descriptor read/write by amortizing the round-trip time over the number of descriptors per transfer. Packets themselves can be transferred in a pipelined way and do not stress DMA engine. Assuming DMA engine runs at 200MHz, time of a MTU packet spent on DMA engine is illustrated in Figure 17. It shows that the new architecture substantially ameliorates DMA engine pressure. Although results for DCA configurations are not shown, they do not avoid long latency descriptor fetches/writes and behave the same as *orig*. In addition to the benefits from DMA engine, the new I/O architecture also reduces NIC buffers. Our experiment results show that it only needs 8KB buffer (4KB buffer in the NEngine and 4KB buffer in the NIC) for the 10Gbps network, but more than 512KB NIC buffer is needed in traditional I/O architectures. With 40Gbps and 100Gbps networks, the new I/O architecture will achieve much higher benefits. In the new architecture, NEngine essentially behaves similarly to DMA engine but simplifies designs of DMA engine and reduces NIC buffers. Therefore, it saves overall hardware cost (CPU+NIC) and offers a promising I/O solution for high speed networks.

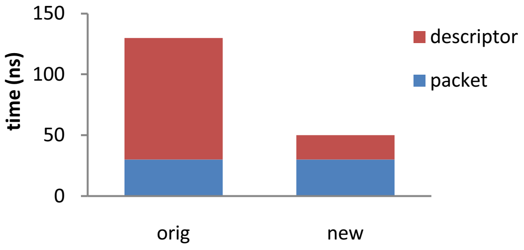

Figure 17. Per packet time on DMA Engine

6. Related Work

A wide spectrum of research has been done to understand the network processing overhead [18-22, 25, 28, 38]. Nahum *et al.* [28] used a cache simulator to study cache behavior of the TCP/IP protocol and showed that instruction cache has the greatest effect on network performance. Similarly, Zhao *et al.* [38] revealed that packets and DMA descriptors exhibit no temporal locality. Makineni *et al.* [25] conducted architectural characterization of TCP/IP processing on Pentium M microprocessors and concluded that the receive side is more memory-intensive than the send side. Unfortunately, they did not conduct a system-wide architectural analysis for high speed network processing on mainstream servers. Moreover, due to a lack of fine-grained instrumentation, none of them located the performance bottlenecks.

In addition, researchers have proposed several architectural schemes to optimize the processing efficiency [1-3, 9, 27, 35, 38]. Most of them aimed to reducing the data copy overhead. Mukerjee *et al.* [27] put a NIC in coherent memory to improve the performance by facilitating transfers of whole cache blocks and reducing control overheads. Zhao *et al.* [38] designed an off-chip DMA engine close to memory to move data inside memory. The similar idea has been implemented in the Intel I/OAT technique [1], but has little performance improvement because memory stalls to packets are still incurred. To eliminate the memory stalls, Intel proposed DCA to route network data into caches [9], and implemented it in Intel 10 GbE NICs and server chipsets. Its performance evaluation on real servers has demonstrated overhead reduction in data copy [18-19]. Recently, Tang *et al.* [35] claimed that DCA might incur cache pollution on small LLC and introduced two cache designs (a dedicated DMA cache or limited ways of LLC) to keep packets. Similar to our work, Binkert *et al.* [2-3] integrated a redesigned NIC to reduce the processing overhead by implementing zero-copy and reducing access latency to NIC registers. The major difference between the integrated NIC and our designs lies in as follows: 1) our architecture only integrates DMA descriptor management onto CPU rather than the whole NIC, leveraging existing NIC designs and reducing CPU die area; 2) the integrated NIC targets at data copy and uses PIO to move data from NICs lacking scalability over a large number of cores. Our architecture enhances the legacy DMA mechanism and uses efficient on-chip data movement to attack all three major performance challenges; 3) instead of multiple queues in NICs, our designs leverage caches for keeping packets, thus further saving NIC cost and achieving better NIC scalability over cores.

7. Conclusion

As network speed continues to grow, it becomes critical to understand and address challenges of network processing in servers. In this paper, we first studied the per-packet processing overhead on servers with 10GbE and pinpointed three bottlenecks: data copy, the driver and buffer release. Then, we instrumented the driver and OS to do a system-wide architectural analysis. Unlike existing tools attributing CPU cost at the function level, our instrumentation was done with fine granularity to reveal exact bottlenecks.

Motivated by the studies, we proposed a new server I/O architecture that addresses all three performance challenges by using extended on-chip DMA descriptors and efficient payload movement. It allows DMA engine to have very fast access to descriptors alleviating burden on the DMA engine and keep packets in CPU caches avoiding extensive NIC buffers. Evaluation results show that the new architecture significantly improves network processing efficiency and achieves better web server performance while reducing the NIC hardware complexity. Given the trend towards rapid evolution of network speed in future, we view the new architecture as a promising I/O solution.

Acknowledgements

The research was supported by NSF grants CCF-0811834, CSR-0912850, and a grant from Intel Corporation.

References

[1] "Accelerating High-Speed Networking with Intel I/O Acceleration Technology", http://download.intel.com/support/network/sb/98856.pdf

[2] N. L. Binkert, A. G. Saidi, S. K. Reinhardt, "Integrated Network Interfaces for High-Bandwidth TCP/IP", *ASPLOS*, 2006.

[3] N. L. Binkert, L. R. Hsu, A. G. Saidi et al., "Performance Analysis of System Overheads in TCP/IP Workloads", *PACT*, 2004.

[4] N. J. Boden, D. Cohen, R. E. Felderman et al., "Myrinet: A Gigabit-per-Second Local Area Network", *IEEE MICRO*, 1995.

[5] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator", *USENIX Technical Conference*, 1994.

[6] L. Cherkasova, V. Kotov, T. Rokichi et al., "Fiber Channel Fabrics: Evaluation and Design", *HICSS*, 1996.

[7] S. GadelRab, "10-Gigabit Ethernet Connectivity for Computer Servers", Vol.27, Issue 3, *IEEE Micro*, 2007.

[8] L. Grossman, "Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver", *OLS*, 2005.

[9] R. Huggahalli, R. Iyer, S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O", *ISCA*, 2005.

[10] Iperf, http://sourceforge.net/projects/iperf/.

[11] "Inside Intel Core Micro-architecture: Setting New Standards for Energy-Efficient Performance", http://www.intel.com/technology/architecture-silicon/core.

[12] Infiniband Trade Association. http://www.infinibandta.org.

[13] Intel 82598 http://www.intel.com/assets/pdf/prodbrief/317796.pdf.

[14] Intel 82599 http://download.intel.com/design/network/prodbrf/321731.pdf.

[15] X. Jiang, N. Madan, L. Zhao et al., "CHOP: Adaptive Filter-based DRAM Caching for CMP Server Platforms", *HPCA*, 2010.

[16] X. Jiang, Y. Solihin, L. Zhao et al., “Architecture Support for Improving Bulk Memory Copying and Initialization Performance”, *PACT*, 2009.

[17] S. King, R. Huggahalli, X. Zhu, M. Memon, F. Berry, N. Bhardwaj, A. Kumar, T. Willke, "Message Communication Techniques", US Patent Application Publication, 2010/0169501.

[18] A. Kumar, R. Huggahalli, "Impact of Cache Coherence Protocols on the Processing of Network Traffic", *MICRO*, 2007.

[19] A. Kumar, R. Huggahalli, S. Makineni, "Characterization of Direct Cache Access on Multi-core Systems and 10GbE", *HPCA*, 2009.

[20] G. Liao, L. Bhuyan, "Performance Measurement of an Integrated NIC Architecture with 10GbE", *HotI,* 2009.

[21] G. Liao, L. Bhuyan, D. Guo, S. King, "EINIC: An Architecture for High Bandwidth Network I/O on Multi-Core Processors", *ANCS*, 2009.

[22] G. Liao, L. Bhuyan, W. Wu, H. Yu, S. King "A New TCB Cache to Efficiently Manage TCP Sessions for Web Servers", *ANCS*, 2010.

[23] F. Liu, X. Jiang, Y. Solihin, "Understanding How Off-chip Memory Bandwidth Partitioning in Chip-Multiprocessors Affects System Performance", *HPCA*, 2010.

[24] P. S. Magnusson, M. Christensson, J. Eskilson et al., "Simics: A Full System Simulation Platform", *IEEE Computer*, February 2002.

[25] S. Makineni, R. Iyer, "Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor", *HPCA*, 2004.

[26] D. J. Miller, P. M. Watts, A. W. Moore, "Motivating Future Interconnects: A Differential Measurement Analysis of PCI Latency", *ANCS*, 2009.

[27] S. S. Mukherjee, B. Falsafi, M. D. Hill et al., "A Coherent Network Interfaces for Fine-Grain Communication", *ISCA*, 1996.

[28] E. Nahum, D. Yates, D. Towsley et al., "Cache Behavior of Network Protocols", *SIGMETRICS,* 1997.

[29] Oprofile, http://oprofile.sourceforge.net/news/.

[30] PCI-E Performance Measurement, http://cp.literature.agilent.com/litweb/pdf/5989-4076EN.pdf.

[31] PCI-E Specificiation, http://www.pcisig.com/specifications/pciexpress/base2/.

[32] F. Petrini, W. Feng, A. Hoisie et al., "The Quadrics Network (QsNet): High-Performance Clustering Technology", *HotI,* 2001.

[33] "Scalable Networking: Eliminating the Receive Processing Bottleneck", Microsoft *WinHEC* April 2004.

[34] Standard Performance Evaluation Corporation. SPECweb benchmark. http://www.spec.org.

[35] D. Tang, Y. Bao, W. Hu et al., "DMA Cache: Using On-chip Storage to Architecturally Separate I/O Data from CPU Data for Improving I/O Performance", *HPCA*, 2010.

[36] Understanding the Linux Kernel, Third Edition, O' Reilly Media.

[37] P. Willmann, H. Kim, S. Rixner et al., "An Efficient Programmable 10 Gigabit Ethernet Network Interface Card", *HPCA*, 2005.

[38] L. Zhao, L. Bhuyan, R. Iyer et al., "Hardware Support for Accelerating Data Movement in Server platform", *IEEE Transactions On Computer*, Vol 56, No. 6, 2007.