Reevaluation of Programmed I/O with Write-Combining Buffers to Improve I/O Performance on Cluster Systems

Steen Larsen
Intel Corporation
22238 NW Beck
Portland, OR 97231
Steenx.k.larsen@intel.com

Ben Lee
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331-5501, USA
benl@eecs.orst.edu

Abstract—**Performance improvement of computer system I/O has been slower than CPU and memory technologies in terms of latency, bandwidth, and other factors. Based on this observation, how I/O is performed needs to be re-examined and explored for optimizations. To optimize the performance of computer system having multiple CPU cores and integrated memory controllers, this paper re-visits a CPU oriented I/O method where data movement is controlled directly by the CPU cores, instead of being indirectly handled by DMA engines using descriptors. This is achieved by leveraging the write-combining memory type and implementing the I/O interface as simple FIFOs. Our implementation and evaluation of the proposed method show that transmit latency and throughput significantly better for small and medium sized messages, and throughput for large messages is comparable to descriptor-based DMA approach.**

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

I. INTRODUCTION

I/O transactions are typically performed using *descriptor-based Direct Memory Access* (DMA), which decouples the software that produces data on a CPU core from the data departure from the system. This allows an I/O device to execute I/O transactions as fast as it can handle. Descriptor-based DMA transactions are appropriate for transferring large messages. However, the overhead of using descriptors diminishes the performance of transferring small and medium sized messages.

In order to better understand this overhead and thus the motivation for this paper, Fig. 1 shows the typical Ethernet transmit flow. After the kernel software constructs the outgoing packet and enqueues it in the transmit buffer within the system memory (1), the core sends a doorbell request to the NIC via the platform interconnect (e.g., PCIe) indicating that there is a pending packet transmission (2). The doorbell request triggers the NIC to initiate a DMA operation to read the descriptor containing the physical addresses of the transmit header and payload (3). The NIC parses the descriptor contents and then initiates a DMA operation to read the header information (e.g., IP addresses and the sequence number) and payload data of the packet (4). An Ethernet frame is constructed with the correct ordering for the bit-stream (5). The NIC will also signal the operating system (typically with an interrupt) that the transmit payload has been processed, which allows the transmit buffer of the packet data to be deallocated. Finally, the bit-stream is passed to a PHYsical (PHY) layer that properly conditions the signaling for transmission over the medium (6).

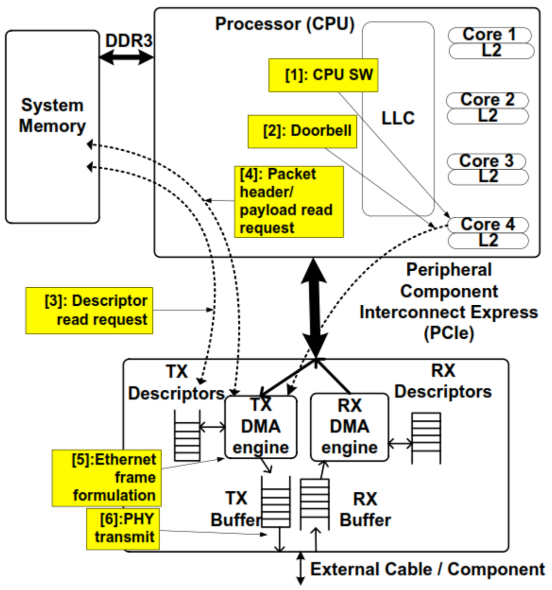


Figure 1. Typical Ethernet transmit flow

As can be seen, descriptor-based DMA operations require several round-trip latencies between the system memory and the I/O device. A simple and obvious approach to eliminating this overhead and improving the performance of small messages is to use *Programmed I/O* (PIO), where a CPU core writes directly to an I/O device [7]. However, PIO cannot fully utilize the available I/O bus bandwidth, and thus the throughput suffers. Therefore, this paper presents the use of *Kernel-protected Programmed I/O with Write-Combining buffers* (kPIO+WC) to improve I/O performance. The idea behind PIO with WC buffers is not new. Bhoedjang *et al.* first presented a study that showed that PIO with WC buffers outperformed DMA for messages less than 1 KB. However, there are two major reasons for reevalauting the potential of this concept. First, the original study on PIO with WC buffers was performed on older CPUs, which had the issue of quickly over-running the few available WC buffers and stalling the CPU during packet transmission. For example, their study showed that PIO with WC buffers for message sizes larger than 1 KB resulted in only about 70% of the traditional DMA throughput. In contrast, current CPUs have many WC buffers per core allowing pipelined write transactions to provide near DMA throughput. Second, with the proliferation of cores in CPUs, there are benefits of having these cores be more directly involved in I/O transactions to improve latency and throughput for small and medium-sized messages.

Low latency for small- and medium-sized messages would be beneficial in clustering applications, such as *Memcached*, as well as in the financial market where a microsecond can garner millions. *Memcached* is a distributed memory caching system used by companies such as Facebook to quickly access vast amounts of inter-related data [18]. Low latency stock prices allow for sophisticated high-frequency trading methods such as algorithmic and arbitrage trading [1]. One example is Xasax claiming 30 microseconds delay between NASDAQ quotes and trade execution [1].

The kPIO+WC approach was implemented as an I/O adapter using an FPGA and tested on a current high-performance system. Our study shows that the proposed kPIO+WC method reduces the latency by 86.7% for small messages compared to descriptor-based DMA operations. In addition, kPIO+WC provides similar throughput results as descriptor-based DMA operations.

II. Related Work

There have been specialized approaches to improve I/O performance. Mukherjee *et al.* proposed the use of coherent memory interfaces for I/O communications, bypassing DMA transfers completely [2]. Although this can reduce system-to-system latency and is applicable to top-tier HPC systems, our goal is to explore the general-purpose cluster interconnect for HPC and datacenters that is more cost sensitive and compatible with the existing ubiquitous PCIe-based Ethernet and InfiniBand interconnects.

Ethernet-based interconnects are widely used in HPC systems as shown by the Top500 supercomputers, where 41% of the systems are Ethernet-based [3]. InfiniBand, which also uses descriptor-based DMA, is also a common HPC interconnect [3]. However, InfiniBand is a complex I/O device that offloads the connection management task (essentially the TCP/IP stack) to the I/O device. This requires I/O device memory to support connection contexts and possible re-ordering of packets. There are also some supercomputers that use dedicated I/O processors to perform *I/O forwarding*, which minimizes operating system noise due to interrupts and context switches associated with I/O transactions [24].

The proposed kPIO+WC method is baselined to Ethernet interconnects since our motivation is to examine improvements to generic I/O devices. Since descriptor-based DMA is used in most high-performance I/O devices, Ethernet interconnects allow for a comparative study of both latency and throughput performance.

The closest related work on simplifying I/O transactions can be found in Programmed I/O (PIO) or memory-mapped I/O that allows direct access by an application to perform I/O transactions. A comparison of PIO and DMA showed that PIO has lower latency for messages less than 64B; however, DMA outperforms PIO for messages larger than 64B [4]. Part of the reason PIO performs poorly for large I/O transactions is that they are treated as uncacheable memory transactions. In addition, PIO transactions using the PCIe interface occur in maximum of 8-byte PCIe payload packets. Adding the 24-byte PCIe packet header effectively reduces PCIe bandwidth utilization to 25% of the peak PCIe bandwidth.

A variation of PIO is user-level I/O that avoids system calls to the operating system, which may include memory copies to kernel buffers impacting I/O transaction performance [7]. The primary issue with user-level I/O is sharing, or I/O virtualization, which causes possible contention between multiple writers and readers for the shared I/O queues.

Another variation of PIO is the *PIO with Write-Combining* (WC) buffer [5] (which Intel has recently renamed them as *Fill Buffers* [6]), which involves writing packets directly to the I/O device using the WC buffers. A WC buffer can be used to combine multiple 4-byte or 8-byte writes into a 64-byte data, which can then be written to an I/O device using a single PCIe transaction.

This approach is similar to our proposed method, but with the following differences. First, each Intel CPU core implements up to 10 WC buffers allowing for pipelined write transactions at near system memory bandwidths using PCIe memory writes [16]. For example, if a CPU core issues a 64-bit write operation each cycle, a 64-byte WC buffer would fill in 8 cycles and a 2.5 GHz CPU core could be writing at maximum data rate of 20 GB/s. This throughput is comparable to the 16 GB/s bandwidth of a single PCIe Gen3 ×16 slot. Although this throughput does not account for PCIe protocol overhead, a single I/O slot can almost sustain the 16 GBps peak throughput of a single PCIe Gen3 ×16 slot. As a result, an individual PCIe interface can now support an individual core throughput. Second, our method maintains the OS kernel protection by using a kernel module (or driver) to protect accesses between the I/O device and system memory. This approach allows multiple applications to access the I/O device without special access arbitration control such as virtualization managers or custom software.

III. Proposed Method

The structure of the NIC based on kPIO+WC is shown in Fig. 2. In addition to legacy components PCIe Host Interface, TX and RX DMAs, TX and RX queues, and the MAC and PHY layers, the proposed method requires kPIO+WC Queue, EOP Filter, and MUX. The *kPIO+WC Queue* act as a buffer for contents evicted from WC buffers. The *EOP Filter* module filters demarcation signatures required for WC buffer eviction (see Section III.A). Finally,

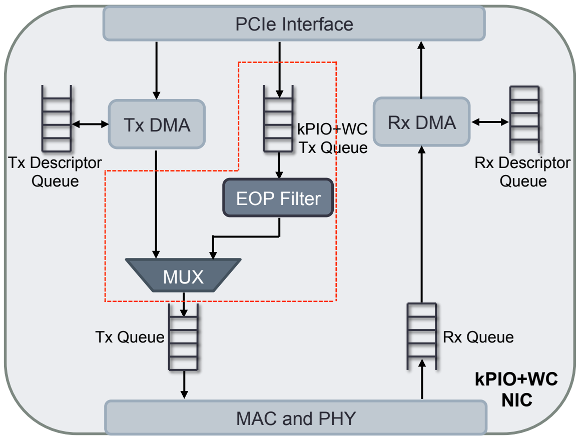


Figure 2. Structure of the proposed method

MUX is needed to support both legacy and kPIO+WC-based traffic.

From a software perspective, kPIO+WC utilizes a kernel-based approach where the contents of WC buffers in the CPU core are directly written into the kPIO+WC TX Queue. This approach simplifies porting of higher level software stack protocols since the driver already has a network packet data structure with the proper frame headers and checksums. This also provides the device sharing protection that is currently supported by mainstream kernels and operating systems (such as Linux and Windows).

The following subsections describe the I/O transmit operation, as well as some implementation issues.

A. *I/O Transmit operation*

In order to transmit an I/O message, the driver first initializes the kPIO+WC TX Queue as a write combining memory type, which allows any writes to the kPIO+WC TX Queue to be performed using WC buffers instead of typical PIO. Then, the CPU core formulates the message and appropriate header information. The header information is the standard Ethernet header with source and destination MAC addresses as well as higher-level packet information, such as IP, TCP, ICMP, etc. The CPU core writes the entire Ethernet frame to the kPIO+WC TX Queue.

In our implementation, the driver is coded to align all transmit frames on a 64-byte WC buffer. This allows a CPU core to write a 64-byte data to the PCIe interface with a 24-byte PCIe frame overhead. This is significantly better than PIO that can only transfer an 8-byte data (in a 64-bit operating system) on each PCIe frame, which improves the PCIe write throughput efficiency from 25% (8/(8+24)) to 72.7% (64/(64+24)).

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

The WC buffers are not directly visible to the software and can be mapped to different memory regions allowing each WC buffer to operate independent of other WC buffers within a CPU core. In our implementation, there is a 4 KB address range allocated in the kernel memory for the kPIO+WC TX Queue. This means that when a transmit packet is written, a new WC buffer is requested and each 64-byte region is filled. A WC buffer gets evicted from the CPU core when it becomes full. In our case, the driver is executing in the kernel and the 4 KB transmit region is not being shared with any other cores, and thus it has full control of the order among WC buffers. Therefore, transmissions by multiple CPU cores would require either a locking mechanism or multiple kPIO+WC TX Queues, such as seen in Receive Side Scaling (RSS) [7].

One limitation of WC buffers is that the data writes needs to combined until a WC buffer becomes full, or some other event flushes the buffer, to optimally coalesce the write operations. The WC buffers can be flushed with x86 instruction `cflush`, `sfence`, or `mfence` [8], but each of these instructions is a costly operation of about 100 CPU core cycles [9]. In our implementation, the eviction of WC buffers is carefully controlled in the kernel driver code to avoid this explicit flush requirement as explained below.

The Intel specification for WC buffer eviction [8] notes an “option” to evict a given WC buffer, but the wording is in the context of cache coherency. This is a critical specification for proper memory coherence (which does not apply to our non-temporal data movement instructions) as well as the understanding that a partial fill of a WC buffer may not be evicted for a long period of time. Our interpretation is that as soon as a WC buffer is completely filled, its eviction is triggered. This assumption was verified by our measurements over multiple WC buffer writes as well as PCIe trace analysis, where 4 KB writes to WC buffers were measured and very little variability between PCIe write transactions were observed.

Since transmit frames may not align on 64-byte boundaries of WC buffers, the remaining bytes are “stuffed” with an 8-byte *End-Of-Packet* (EOP) signature similar to an Ethernet frame EOP. This stuffing serves the purpose of reliably flushing the WC buffer. This EOP field is not part of the network packet outside the system, and thus, it is an overhead only between the CPU and the I/O device. In our implementation, a special 64-bit code is used to implement EOP. There is some inefficiency due to this artificial stuffing of data, but the overhead is small when compared to the bandwidth inflation involved with doorbells and descriptors [10].

Fig. 3 shows how the PCIe write transactions are enqueued in the kPIO+WC TX queue of the I/O device. Each entry is 64 bytes and the figures shows four packets of different payload sizes. In our implementation, the EOP signature is included in the message passed between the CPU core and I/O device. The EOP Filter module then filters the EOP signatures before the message is sent out to the network. This allows compatibility between legacy devices and kPIO+WC-enabled devices. Packet *A* with 120-byte payload has a single 8-byte EOP signature. Packet *B* with

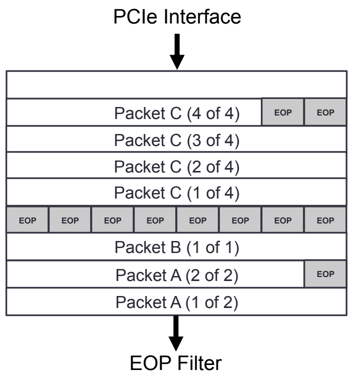


Figure 3. kPIO+WC-enabled TX Queue

64-byte payload requires an additional 64-byte of EOP stuffing to notify the I/O adapter that it is only 64 bytes. This is because it needs to signal an EOP, but there is no space within a single 64-bit WC buffer. Packet *C* contains 240 bytes with two EOP signatures.

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

A CPU core can quickly over-run the kPIO+WC TX Queue on an I/O device, and thus a larger buffer would be needed to account for increase in bandwidth-delay products. To address this issue, our implementation takes advantage of each CPU core in a typical system having between 6~10 WC buffers depending on the core architecture. Some tests have shown 2×~4× throughput improvement by pipelining writes across the available WC buffers using inline assembly instructions that bypass L1 and L2 lookups [8]. An example of such a device driver code that writes two 64-byte portions of a packet using in-line assembly is shown below where each SSE2 `movntdq` instruction writes 16 bytes to a WC buffer so that it can be filled with four instructions.

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

The above code shows that WC buffer `xmm0` is first written. As `xmm0` is flushed to the PCIe interface, the second half of this code increments the index to WC buffers and writes to `xmm1`. This code segment can be expanded with pointers to different WC buffers based on the number of WC buffers available to each core in the CPU. By efficiently using the available WC buffers on each CPU core there will be no throughput transmit bottleneck and the throughput will track closely with the available PCIe bandwidth.

Note that the `movntdq` instruction is a non-temporal move, meaning there are no cache lookups or coherency checks. Moreover, since the example code is part of the I/O device driver code, the operating system protects the device and only allows access with the normal function calls such as `send()`.

B. kPIO+WC Implementation Issues

There are two general implementation options for kPIO+WC. The simplest implementation is similar to I/O adapter accelerator functions, e.g., Large Receive Offloads (LRO) [11], which are enabled for all connections using the I/O adapter during driver initialization. In this case, all network connections are either descriptor DMA based or kPIO+WC generated transmissions. To enable kPIO+WC transmits, a control bit in the I/O adapter would be set via an operating system command such as *modprobe()*.

The second more complex implementation is to define each network connection to be either kPIO+WC or descriptor-based DMA. For example, kPIO+WC can be used for a certain range of TCP ports. It is also possible to control kPIO+WC versus descriptor-based DMA transmission on a per-packet basis, but the added overhead probably may not justify the flexibility.

IV. MEASUREMENTS AND ANALYSIS

Our baseline measurements and analysis are based on a 2.5 GHz Intel Sandy Bridge 4-core i5-2400S platform configured as shown in Fig 1. A Linux x64 kernel 2.6.35 is used with an Ubuntu distribution to support the custom network driver code. The proposed method is implemented using a PCIe-based Xilinx Virtex5 (XC5VTX240T) FPGA. The PCIe bandwidth is 8 Gbps simplex. Although PCIe Gen1 interface technology is used, the subsequent PCIe generations also follow the same protocol basically increasing lane speed and number of lanes. This allows extrapolation of our Gen1 data to the current Gen2 I/O devices, and future Gen3 devices. Each core in the CPU used has 10 WC buffers. The measurements are taken using a combination of Lecroy PCIe analyzer tracing and internal FPGA logic tracing. These hardware measurements are strictly passive and do not induce any latency or performance overhead. The software micro-benchmarks ICMP/IP ping and *iperf* are used for latency and bandwidth testing, respectively, to compare the proposed method versus the standard Ethernet.

Our test code is built on the example Ethernet driver code found in Linux Device Drivers [12], which loops back the subnet and IP addresses allowing experiments to be run without a real external network. This is done by instantiating two bi-directional Ethernet interfaces on two separate subnets. This allows us to isolate the system latencies for analysis without wire and fiber PHY latencies and their variations.

Note that it is also possible to utilize kPIO+WC on the receive path. However, existing I/O devices prefetch and coalesce Ethernet frame descriptors, and thus there is no significant latency improvement by having kPIO+WC for receive data. As a result, latency savings appear only in the transmit path, and thus our analysis is focused on the transmit path.

Fig. 4 shows how the kPIO+WC-based I/O adapter implemented in FPGA is interfaced to the host system. Our implementation only includes a single I/O adapter with a single kPIO+WC Queue since our interest is in how a single-core interacts with a single I/O adapter. This avoids any undesired PCIe traffic, such as TCP/IP ACK frames, and other multi-core and multi-interface traffic that occurs over a single PCIe device to skew the experiment. Therefore, the only traffic on the PCIe interface, marked by the green arrow, is transmitted from the chi0 interface and received by the chi1 interface. The reverse traffic (from chi1 to chi0 marked with a red arrow) occurs only in memory as the original driver is coded to avoid irrelevant PCIe traffic in the analysis. This reverse traffic is needed to support higher-level network protocols such as TCP, which assumes ACK packets to ensure a reliable connection.

In order to compare the CPU transmit overhead, similar tests are performed on a descriptor-based Intel 10GbE NIC. Using the Linux *perf* performance tool show that up to 2% of CPU overhead was due to transmit descriptor related operations in the *ixgbe_xmit_frame()* function.

If the PCIe bandwidth cannot sustain the CPU core throughput, meaning the 10 WC buffers (640 bytes) are not drained, the transmitting core will stall and CPU transmit overhead will increase. The risk of stalls is highly workload dependent and requires further explorations [10].

A. Latency Results

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

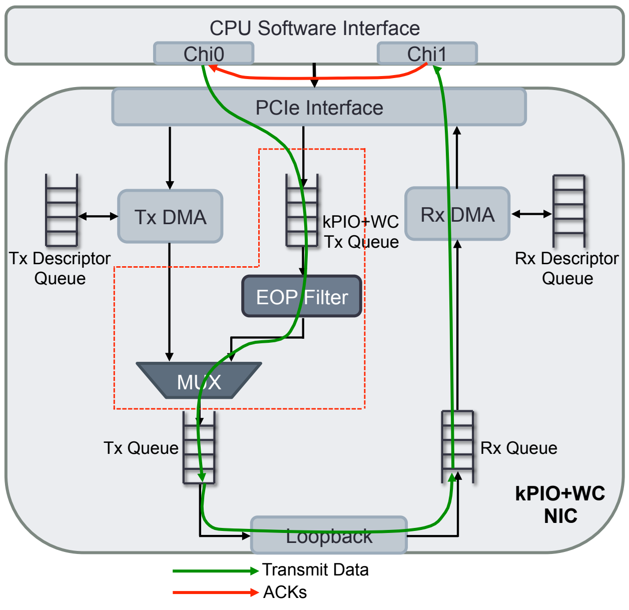
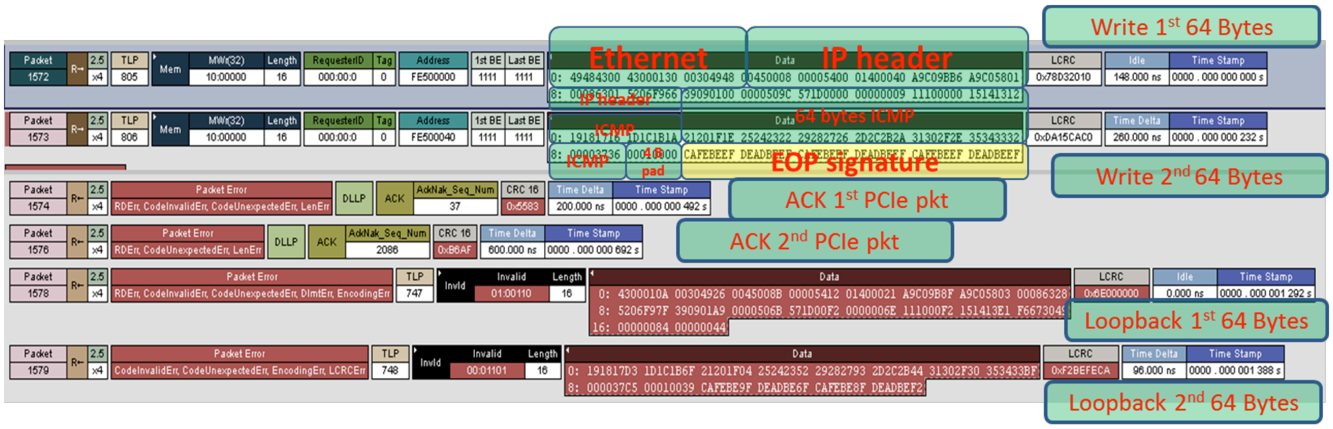


Figure 4. HW & SW co-utilization of interfaces

Fig. 5 shows the loopback trace for the proposed method where each PCIe packet is shown as a separate line and enumerated in the field marked “Packet”. The two PCIe write transactions for the ICMP message are indicated by PCIe packets #1572 and #1573. The temporal reference point at the beginning of packet #1572 is T_0. These two packets are acknowledged with packets #1574 and #1576 by the FPGA I/O device at T_0 + 492 ns and T_0 + 692, respectively. Note that there are CRC failures in the upstream PCIe frames due to a PCIe analyzer failure, but the software verified that the expected loopback data was properly written into the pinned system memory buffers.

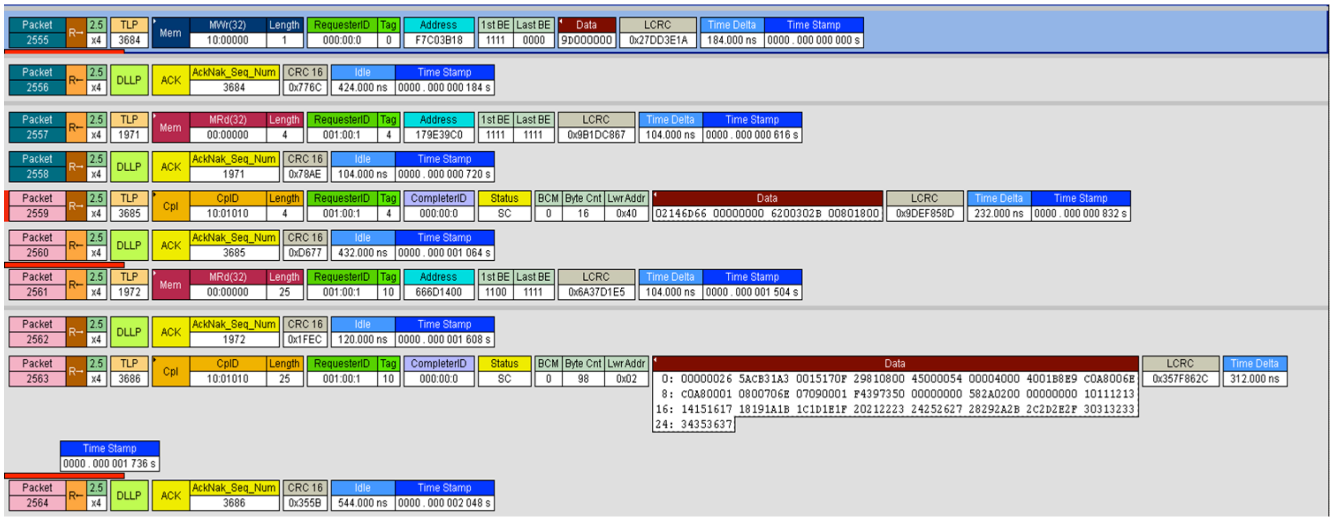
The I/O device initiates the DMA write to the system memory for the looped back packet starting with packet #1578. This transaction is seen on the PCIe interface at T_0+1,292 ns. The second 64-byte PCIe packet containing EOP signatures is written to the system memory with packet #1579 at time T_0 + 1,388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

Fig. 6 shows the latency breakdown of the 64-byte ICMP message using a standard 1GbE NIC [13, 14]. Since the measurement was between two different systems, Fig. 6 only shows the transmit operation. Again, T_0 is used as the initial observance of PCIe traffic in the transmit direction of the doorbell write in packet #2555, which is acknowledged with packet #2556 at T_0+184 ns in. The NIC responds with the transmit descriptor fetch request in packet #2557. The read request is completed with data in packet #2559 and acknowledged at T_0+1,064 ns in packet #2560. After parsing the descriptor, the NIC requests the payload data in packet #2561, which completes with data in packet #2563 and is acknowledged by the NIC at T_0+2,048 ns.



```
Packet 1572 | R→ | 2.5 x4 | TLP 805 | Mem | MWr(32) 10:00000 | Length 16 | RequesterID 000:00:0 | Tag 0 | Address FE500000 | 1st BE 1111 | Last BE 1111 | Data
  0: 49484300 43000130 00304948 00450008 00005400 01400040 A9C09BB6 A9C05801
  8: 00086301 5206F966 39090100 0000509C 571D0000 00000009 11100000 15141312
  | LCRC 0x78D32010 | Idle 148.000 ns | Time Stamp 0000 . 000 000 000 s
Packet 1573 | R→ | 2.5 x4 | TLP 806 | Mem | MWr(32) 10:00000 | Length 16 | RequesterID 000:00:0 | Tag 0 | Address FE500040 | 1st BE 1111 | Last BE 1111 | Data
  0: 19181716 1D1C1B1A 21201F1E 25242322 29282726 2D2C2B2A 31302F2E 35343332
  8: 00003736 000[illegible]00 CAFEBEEF DEADBEEF CAFEBEEF DEADBEEF CAFEBEEF DEADBEEF
  | LCRC 0xDA15CAC0 | Time Delta 260.000 ns | Time Stamp 0000 . 000 000 232 s
Packet 1574 | R← | 2.5 x4 | Packet Error RDErr, CodeInvalidErr, CodeUnexpectedErr, LenErr | DLLP | ACK | AckNak_Seq_Num 37 | CRC 16 0x5583 | Time Delta 200.000 ns | Time Stamp 0000 . 000 000 492 s
Packet 1576 | R← | 2.5 x4 | Packet Error RDErr, CodeUnexpectedErr, LenErr | DLLP | ACK | AckNak_Seq_Num 2086 | CRC 16 0xB6AF | Time Delta 600.000 ns | Time Stamp 0000 . 000 000 692 s
Packet 1578 | R← | 2.5 x4 | Packet Error RDErr, CodeInvalidErr, CodeUnexpectedErr, DlmtErr, EncodingErr | TLP 747 | Invld | Invalid 01:00110 | Length 16 | Data
  0: 4300010A 00304926 0045008B 00005412 01400021 A9C09B8F A9C05803 00086328
  8: 5206F97F 390901A9 0000506B 571D00F2 0000006E 111000F2 151413E1 F6673049
 16: 00000084 00000044
  | LCRC 0x6E000000 | Idle 0.000 ns | Time Stamp 0000 . 000 001 292 s
Packet 1579 | R← | 2.5 x4 | Packet Error CodeInvalidErr, CodeUnexpectedErr, EncodingErr, LCRCErr | TLP 748 | Invld | Invalid 00:01101 | Length 16 | Data
  0: 191817D3 1D1C1B6F 21201F04 25242352 29282793 2D2C2B44 31302F30 353433BF
  8: 000037C5 00010039 CAFEBE9F DEADBE6F CAFEBE8F DEADBEF2
  | LCRC 0xF2BEFECA | Time Delta 96.000 ns | Time Stamp 0000 . 000 001 388 s
```

Figure 5. Ping loopback trace on the PCIe interface for kPIO+WC.



```
Packet 2555 | R→ | 2.5 x4 | TLP 3684 | Mem | MWr(32) 10:00000 | Length 1 | RequesterID 000:00:0 | Tag 0 | Address F7C03B18 | 1st BE 1111 | Last BE 0000 | Data 9D000000 | LCRC 0x27DD3E1A | Time Delta 184.000 ns | Time Stamp 0000 . 000 000 000 s
Packet 2556 | R← | 2.5 x4 | DLLP | ACK | AckNak_Seq_Num 3684 | CRC 16 0x776C | Idle 424.000 ns | Time Stamp 0000 . 000 000 184 s
Packet 2557 | R← | 2.5 x4 | TLP 1971 | Mem | MRd(32) 00:00000 | Length 4 | RequesterID 001:00:1 | Tag 4 | Address 179E39C0 | 1st BE 1111 | Last BE 1111 | LCRC 0x9B1DC867 | Time Delta 104.000 ns | Time Stamp 0000 . 000 000 616 s
Packet 2558 | R→ | 2.5 x4 | DLLP | ACK | AckNak_Seq_Num 1971 | CRC 16 0x78AE | Idle 104.000 ns | Time Stamp 0000 . 000 000 720 s
Packet 2559 | R→ | 2.5 x4 | TLP 3685 | Cpl | CplD 10:01010 | Length 4 | RequesterID 001:00:1 | Tag 4 | CompleterID 000:00:0 | Status SC | BCM 0 | Byte Cnt 16 | LwrAddr 0x40 | Data 02146D66 00000000 6200302B 00801800 | LCRC 0x9DEF858D | Time Delta 232.000 ns | Time Stamp 0000 . 000 000 832 s
Packet 2560 | R← | 2.5 x4 | DLLP | ACK | AckNak_Seq_Num 3685 | CRC 16 0xD677 | Idle 432.000 ns | Time Stamp 0000 . 000 001 064 s
Packet 2561 | R← | 2.5 x4 | TLP 1972 | Mem | MRd(32) 00:00000 | Length 25 | RequesterID 001:00:1 | Tag 10 | Address 666D1400 | 1st BE 1100 | Last BE 1111 | LCRC 0x6A37D1E5 | Time Delta 104.000 ns | Time Stamp 0000 . 000 001 504 s
Packet 2562 | R→ | 2.5 x4 | DLLP | ACK | AckNak_Seq_Num 1972 | CRC 16 0x1FEC | Idle 120.000 ns | Time Stamp 0000 . 000 001 608 s
Packet 2563 | R→ | 2.5 x4 | TLP 3686 | Cpl | CplD 10:01010 | Length 25 | RequesterID 001:00:1 | Tag 10 | CompleterID 000:00:0 | Status SC | BCM 0 | Byte Cnt 98 | LwrAddr 0x02 | Data
  0: 00000026 5ACB31A3 0015170F 29810800 45000054 00004000 4001B8E9 C0A8006E
  8: C0A80001 0800706E 07090001 F4397350 00000000 582A0200 00000000 10111213
 16: 14151617 18191A1B 1C1D1E1F 20212223 24252627 28292A2B 2C2D2E2F 30313233
 24: 34353637
  | LCRC 0x357F862C | Time Delta 312.000 ns | Time Stamp 0000 . 000 001 736 s
Packet 2564 | R← | 2.5 x4 | DLLP | ACK | AckNak_Seq_Num 3686 | CRC 16 0x355B | Idle 544.000 ns | Time Stamp 0000 . 000 002 048 s
```

Figure 6. Ping loopback trace on the PCIe interface for Intel 82575EB 1GbE

Since both the kPIO+WC method and the standard 1GbE use DMA for receive transactions, there is little latency difference in the I/O receive path. Table 1 compares the latencies in these two example traces for a 64-byte ICMP message between two systems. Table 1 shows that kPIO+WC reduces the latency by 1,504 ns.

Note that our proposed approach can reduce latency even further when the message is within a single WC buffer instead of the two WC buffers shown in Fig. 5. Accounting for the header and EOP requirement, only a single WC buffer is needed if the message is less than 20 bytes, which is applicable in the financial trading market. Based on multiple back-to-back WC buffer writes, there is on average 108 ns delay between two consecutive WC buffer writes. Therefore, the minimum latency to send a message out of a system in our implementation is 108 ns. In contrast, the minimum latency to send a 64-byte message out of a system in a descriptor-based 1GbE NIC, including the frame header, is 1,736 ns.

Fig. 7 shows the transmit latency as a function of message size for kPIO+WC and the descriptor-based DMA operation. The figure also shows the 8 Gbps PCIe theoretical bandwidth limitation of our test environment, which is the limit of our ×4 Gen1 configurations and provides the asymptote that the latencies for both kPIO+WC and descriptor-based DMA approach. The descriptor-based DMA transmit latency curve is smoother than the latency for the proposed method since the latter uses 64-byte alignment while the former uses byte-level alignment.

B. *Throughput Results*

Fig. 8 compares throughput as a function of message size for the two methods, which shows that the proposed method outperforms descriptor-based DMA for small messages and the throughput converges with descriptor-based DMA as message size increases. The abrupt degradation for kPIO+WC is again due to the 64-byte WC buffer alignment.

Our analysis of the *iperf* microbenchmark throughput results (sampled for > 100 ms) on the PCIe interface using a dual 10GbE Intel 82599 NIC shows that for transmit overhead, up to 43% of the traffic on the PCIe interface is used for descriptors or doorbells for small 64-byte messages. For larger messages, e.g., 64 KB, the overhead is less than 5%. The proposed approach removes this PCIe bandwidth overhead.

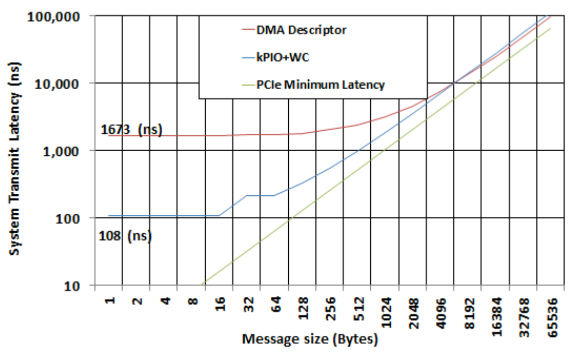


Figure 7. Transmit latency comparison

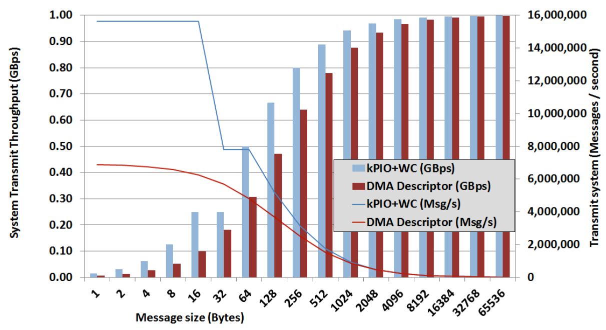


Figure 8. Transmit throughput comparison

Table 1: 64-byte Latency Breakdown Comparison

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

V. CONCLUSION AND FUTURE WORK

This paper evaluated the performance of PIO with WC buffers. Our results show that the proposed method provides significant latency improvement on current systems. Although some changes are required in both the hardware implementation and software driver interface, the implementation costs are small relative to the benefits gained in HPC applications where latency and throughput performance is crucial.

Other less quantifiable benefits of the kPIO+WC approach include a core directly controlling the I/O transmit transactions to allow system power algorithms involving the on-die Power Control Unit (PCU) [15] to react more effectively than sending slow control messages to a PCIe attached I/O DMA engine. I/O transaction Quality-of-Service (QoS) also improves since a core can control (or filter) I/O transactions based on priority. In addition, system memory bandwidth utilization and memory latency improve by not having I/O DMA transactions between multiple I/O devices contending with core related memory transactions.

As future work, we plan to explore other improvements to the kPIO+WC approach. For example, increasing the WC buffers would benefit I/O performance in general. We also want to further explore and quantify the receive flow benefits with and without descriptor-based DMA. Finally, we plan to study how kPIO+WC can be used to moving data across a PCIe switch fabric.

VI. ACKNOWLEDGMENTS

The authors would like to thanks Guandeng Liao, Jesse Brandeburg, and Steve McGowan at Intel for providing helpful comments and clarifications. This research was supported in part by Ministry of Education Science and Technology (MEST) and the Korean Federation of Science and Technology Societies (KOFST).

REFERENCES

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

[2] Mukherjee, S.S., et al. *Coherent network interfaces for fine-grain communication*. 1996: IEEE.

[3] Top500. 2013; Available from: http://i.top500.org/stats.

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

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

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

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

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

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] Wikipedia, *Haswell PCU*. 2012.