

Understanding PCIe performance for end host networking

Paper #146, 13 pages

ABSTRACT

Recent years have been characterized by significant innovation around the implementation of network functions. With the rise of programmable NICs, end hosts have been increasingly considered as a natural enforcement point for the deployment of core functionalities in modern datacenters, e.g., load balancing and congestion control with minimal network support requirements. However, implementing custom accelerators in practice is not easy. Indeed, many potential bottlenecks can impede an high performance realization of such hardware/software co-designs.

This paper focus on two of them. The PCIe, the de-facto standard I/O for contemporary servers, and its performance implication when interacting with different operating system features. We present a theoretical model for the I/O to support the development of new smart NIC enabled applications and `pcie-bench`, an open-source test suite that allows developers to gain an accurate and deep understanding of the PCIe substrate operation on real systems. Those contributions allow to fine tune both host applications performance and the implementation of solutions based on emerging smart NICs. Using `pcie-bench` we present a fine-grained characterization of the PCIe subsystem in modern servers. We then highlight undesirable impact of popular PCIe implementation features such as IOMMU, and the practical limits for common network interface cards operating at 40Gb/s and beyond. Furthermore, with `pcie-bench`, we gained performance insights which guided software and future hardware architectures for both commercial and research oriented network cards and DMA engines.

1 INTRODUCTION

The idea of end hosts participating in the implementation of network functionality has been extensively explored in enterprise and datacenter networks [16, 17, 27, 45, 52, 54]. Moreover, the disruptive introduction into the market of programmable NICs, alongside the deployment in datacenters of hybrid Xeon and FPGA platforms [22], has boosted the demand for new refined solutions which combine software functions and hardware NIC acceleration to improve end host networking performances [49], flexibility [23], or a combination of both [31]. In this scenario, a number of research efforts have been proposed to improve datacenter scalability [20] or specific network functions such as load balancing, application level quality of service and congestion control [11], by leveraging end host hardware programmability.

In this paper, we show that PCIe, the standard I/O interconnect for contemporary servers, alongside its interaction with the operating system, can significantly impact the performance of network applications. Past research has touched on this, but mostly in the context of specific applications such as Remote DMA (RDMA) [26], GPU-accelerated packet processing [24], and optimised Key-Value-Store (KVS) applications [33, 35]. In contrast, we argue that a more generic approach to studying and characterizing PCIe is needed as it has become more essential to implement high-performance custom network accelerators to enable a wide range of improvements for

diverse workloads. In this scenario, considering the rapid evolution that the I/O interconnect operation is facing in recent years with new implementation features such as IOMMU, it is important to provide a solution that systematically evaluate the performance of PCIe in relation to different operating system features. It is in this context that we characterize current PCIe performances, developed a tool to model its behaviour and highlight the effect of the constant development of the PCIe subsystem. We introduce a methodology, describe its implementation, and present the results derived using our approach. This permits us to draw several specific conclusions about the way PCIe currently operates and the implications this has for operating systems and applications design. For example, we demonstrate how both NUMA for PCIe devices and IOMMUs can have a negative performance impact on application performances.

The contributions of this paper are:

- We introduce a PCIe theoretical model, particularly suited to quickly asses the impact of design alternative when implementing custom NIC functionality.
- We outline a methodology, `pcie-bench`: a combination of micro-benchmarks that measures aspects of performance for both NICs DMA engines and, more crucially, the PCIe root complex¹.
- We describe two implementations of `pcie-bench` permitting systematic, measurement-based, characterization of PCIe devices, their DMA engines, and the interaction with the root complex. Each is based on commodity PCIe peripherals: the first board from Netronome and the second, the latest NetFPGA PCIe board targeted toward researchers. The implementations are available as open source² to allow for the reproduction of our results and to enable other practitioners to apply to other systems and architectures.
- We present the first, publicly-disclosed, detailed measurements of PCIe performance across several generations of conventional host-systems; this permits us to trace and highlight the micro-architectural evolution of PCIe implementations within the host-context.
- We discuss the lesson we learned alongside a number of practical use-cases for `pcie-bench`. From the original need to characterize, debug, and improve implementations, to aiding and evaluating the growing area of research moving beyond the straightforward TCP offload with more complex solutions using programmable network cards, i.e., smart NICs.

The remainder of this paper is organized as follows: we first motivate further the need for `pcie-bench` (§2). We then provide the necessary background to understand PCIe performance bottlenecks and introduce a model for PCIe which allows designers of custom Smart NICs to asses their design choices (§3). In §4 we introduce our methodology to gain deep insides into the performance of PCIe and,

¹The PCIe root complex connects the processors and memory subsystem of a host system with the PCIe switch fabric and, subsequently, to individual PCIe devices. Its functionality is similar to the PCI host bridge in older systems.

²link anonymized.

in §5 provide details on implementations for both the Netronome and NetFPGA boards. An extensive set of results are reported in §6. We discuss the lessons we learned and how the insights we gained can be applied in §7. Finally, we cover related work (§8) and conclusions (§9).

2 MOTIVATION

Traditionally, I/O interconnects in commodity systems, such as PCIe, are not widely studied, in part, because they have worked effectively for many years and have been usefully treated as a black-box that *just works*. It is only as we have I/O devices that approach (and exceed) the capabilities offered by PCIe, e.g., dual-port 40Gb/s network adapters, that we have seen a significant hardware-software co-design with the constraints of PCIe in mind [23, 31, 32].

Such I/O-aware software design is hardly new. In the past, when faster ATM and 1Gb/s PCI bus adapters were introduced, many driver and kernel redesigns have been proposed to accommodate the PCI behavior [18, 46, 56]. Unfortunately, current PCIe network systems in the commercial domain often involve dedicated silicon and proprietary procedures to speeding up certain tasks, e.g., high-frequency-trading or cluster protocols such as MPI. This black-box approach has made it difficult for application developers and researchers to build solutions that properly capitalize on the PCIe features to improve I/O performances, pushing the research community to focus mainly on novel software frameworks [1, 8, 13, 50]. These frameworks have been widely adopted in order to dramatically improve network performance, overcoming the limitations of conventional network stacks. However, PCIe can significantly affect the performances in terms of throughput and latency experienced by applications, potentially undermining efforts made towards better software solutions. Most importantly, the PCIe root complex has seen a rapid evolution in recent years, introducing new features and properties that can be exploited when high performance packet I/O is a first-class requirement.

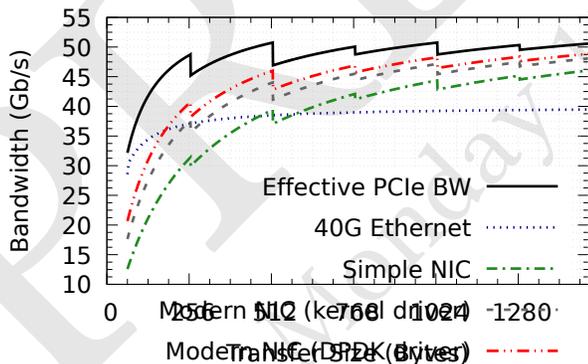


Figure 1: Modelled bidirectional bandwidth on a PCIe Gen 3 x8 link for a simple NIC model (descriptor transfers and register accesses for each packet) and a modern NIC (which batches descriptor transfers and other interactions) with a typical kernel driver and a DPDK poll mode driver.

PCIe impact on network application throughput. A PCIe Gen 3, typically used by 40 Gb/s NICs, has a throughput of 62.96 Gb/s at the physical layer. However, PCIe protocol overheads reduce the usable bandwidth to around 50 Gb/s, or significantly less, depending on the amount of data being transferred. Figure 1 shows the effective bi-directional bandwidth achievable for such a device (labeled Effective PCIe BW). We observe a saw-tooth pattern as the transfer size is increased, which can affect the achievable throughput by up to 10%. This is caused by the *packetized*, transactional structure of the PCIe protocol where the data to be transferred, e.g., a network packet, is broken up into smaller PCIe packets with their own PCIe level headers.

Apart from transferring the packet data itself, a NIC also has to read TX and freelist descriptors, write back RX (and sometimes TX) descriptors and generate interrupts. NIC device drivers have to read and update queue pointers on the device. All these interactions are PCIe transactions consuming additional PCIe bandwidth. If we model a very simple NIC, which DMAs individual descriptors for every packet and where the driver reads/writes the queue pointers for every packet, we can see a dramatic decrease in achievable bi-directional bandwidth (labeled Simple NIC in the figure). Such a device would only achieve 40 Gb/s linerate throughput for Ethernet frames larger than 512B, *purely based on PCIe bandwidth calculations*. Most modern NICs (and drivers) therefore implement a number of optimizations, such as en-queuing descriptors in batches, pre-fetching descriptors, interrupt moderation, etc. These optimizations increase the complexity of the NIC but are required to achieve acceptable throughput. Modern NIC (kernel driver) in the figure shows the throughput of such a moderately optimized NIC/driver combination with a typical Linux kernel driver. More modern software frameworks, such as DPDK [1], enable further optimizations at the driver level alone (no NIC hardware changes). By disabling interrupts and polling write-back descriptors in host memory instead of device registers, the number of PCIe transactions is further reduced and the achievable bandwidth is increased (labeled Modern NIC (DPDK driver)). We detail how these models are derived in Section 3. However, Figure 1 shows that when designing custom offloads to programmable NICs, developers and researchers have to be acutely aware of the overheads imposed by the PCIe transactions, caused both by the device and the device driver, as they might impact the throughput experienced by the application.

PCIe impact on network application latency. We used an ExaNIC³ to estimate the contribution of PCIe to the overall end-host latency experienced by a network application. We executed a loop-back test to measure the total NIC latency (from application to the wire and back) with various sized packets. The test writes a packet to the driver’s buffer and measures the latency between when the packet starts to be written to PCIe and when the packet returns. The test utilizes kernel bypass mode, so does not include any kernel overheads. We also modified the ExaNIC firmware, using Exablaze firmware development kit⁴, to measure the PCIe latency contribution.

Figure 2 shows the results from the test. On the ExaNIC, the PCIe subsystem contributes between 77% of the latency for 1500B

³<http://www.exablaze.com>

⁴<http://www.exablaze.com/docs/exanic/user-guide/fdk/fdk/>

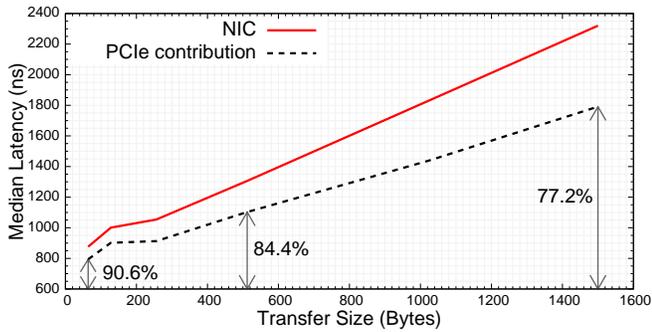


Figure 2: Comparison between the NIC latency and just the PCIe share.

packets, increasing to more than 90% for small packet sizes. In particular, the overall latency is well within the range which has been shown to have negative impact on performance for several common datacenter applications [59].

PCIe latency impact on NIC performance. Figure 2 illustrates that the round trip latency for a 128B payload is around 1000 ns with PCIe contributing around 900 ns. With 40 Gb/s Ethernet at line rate for 128B packets, a new packet needs to be received and sent around every 30ns. Assuming that the measured PCIe latencies are symmetric, this implies that the NIC has to handle at least 30 concurrent DMAs in each direction to cover for the PCIe latencies in order to achieve line rate for 128B packets. Given that a NIC also has to issue DMAs for descriptors, the number of in-flight DMAs a NIC has to handle is even higher. This not only introduces significant complexity when writing optimized software for programmable NICs but also imposes constraint on the device driver having to supply host buffers in large enough batches for the NIC to consume.

PCIe root complex is evolving. In modern, x86 based servers, the PCIe root complex has seen a rapid evolution in recent years. Specifically, together with the memory controller(s), it has been closely integrated with the CPU cores, enabling tighter integration of PCIe devices with a CPU’s caches, e.g., Intel’s Data Direct I/O (DDIO) [4]. For multi-socket systems, this integration also results in non-uniform memory access (NUMA [30]) for PCIe devices: DMAs may target memory local to the socket the PCIe device is attached while others need to traverse the CPU interconnect. . Finally, most modern systems have an IOMMU [5] (Input-Output Memory Management Unit) interposed in the data path between a PCIe device and the host. These recent technologies not only impact the overall latency (and bandwidth) of PCIe transactions, but they also introduce variance as transactions are now depending on the temporal state of caches, IOMMU Transaction Lookaside Buffers (TLBs) and the characteristics of the CPU interconnects. Furthermore, after years of a relative x86 mono-culture in the server market, designs around ARM64, Power, and SPARC processors have all received mainstream deployment. Each of these architectures have vastly different implementations of the I/O subsystem and most of the associated details are not publicly available, making it

hard for researcher to characterize and understand the differences among these server architectures.

The impact of PCIe on the experienced throughput when data needs to be transferred from hardware to software and vice versa, the predominant contribution of PCIe to the overall NIC latency, the introduction of new technologies in the PCIe root complex, and the mainstream deployment of new server architectures have motivated us to build `pcie-bench`: to get a better, more detailed and systematic understanding of the PCIe implementation in commodity systems and to help developers to fine tune applications to make more efficient use of the underlying system.

3 MODELLING PCIE

As part of `pcie-bench` we have developed a model of PCIe specification [43, 53]. The model allows us to: (1) validate bandwidth measurements from micro-benchmarks against expected bandwidth based on the specification, and (2) model the impact of more complex NIC/host interactions, such as shown in Figure 1.

PCIe uses a high-speed serial interconnect based on a point-to-point topology consisting of several serial links (or lanes) between endpoints. The coordination of the point-to-point topology forces a packetized, transactional structure: with headers, flow and congestion control along with acknowledgments onto PCIe. It is a protocol with three layers: physical, data link and transaction. While the data link layer (DLL) implements error correction and flow control, the transaction layer turns user application data, or completion data, into PCIe transactions using Transaction Layer Packets (TLPs).

Most of the single or dual port 40Gb/s NICs on the market today have a PCIe gen 3 interface with 8 lanes. Each lane offers 8 GT/s (Giga Transactions per second) using a 128b/130b encoding⁵, resulting in a usable bandwidth at the physical layer of $8 \times 7.87 \text{ Gb/s} = 62.96 \text{ Gb/s}$. The DLL adds around 8–10% of overheads due to flow control and acknowledgement messages, leaving around 57.88 Gb/s available at the TLP layer⁶.

At the transport layer, the standard defines a number of different TLP types. For the purpose of this paper, the relevant packet types are: Memory Read (MRd), Memory Write (MWr), and Completion with Data (CplD). TLPs have a common header, a type specific header, and an optional trailing digest. The common header contains information such as the type of TLP, the TLP length, various attributes and is 4B in size. The header of MWr and MRd TLPs is 12B long (assuming 64bit addressing) while for CplD TLPs it is 8B. The maximum amount of data carried in a single TLP (Maximum Payload Size, or MPS) is negotiated between the peers and the optional 4B digest contains an end-to-end CRC (ECRC). Typical values for the MPS are 256B or 512B.

Figure 3 illustrates how a Ethernet frame received from the network is transferred to a host buffer via PCIe DMAs. Most NICs are configured to strip the Ethernet FCS, so only the Ethernet header and the payload are DMAed. The frame is broken into several MPS sized MWr TLPs. Each TLP consists of 2B of framing (start and stop byte at the physical layer), 6B of DLL header, and $4B + 12B = 16B$

⁵Different generations of PCIe use different encoding, which are also available in the model we developed.

⁶These overhead depends on the transaction rate and implementation details. The value of 57.88 Gb/s was derived using recommended values from the PCIe specification [43].

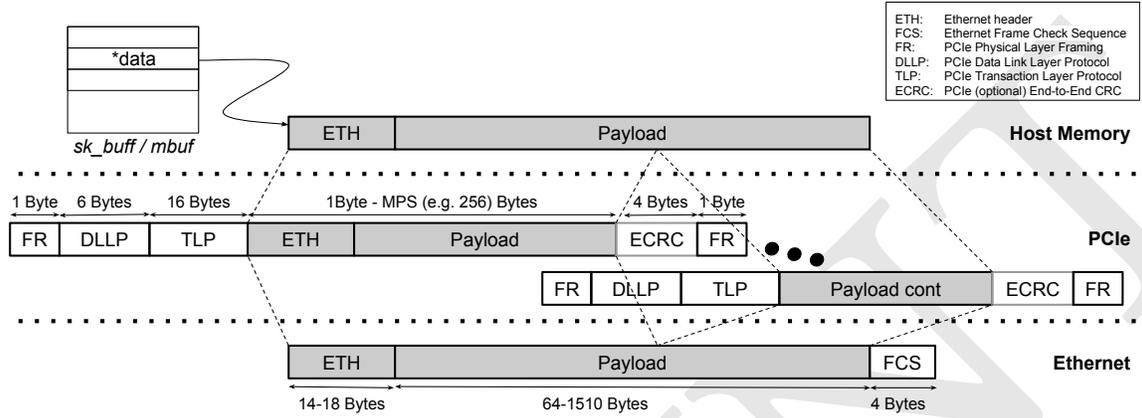


Figure 3: A structural illustration of PCIe transaction(s) and their relationship with Ethernet packets and the hosts' physical memory.

for the MWr TLP as well as the MPS sized payload. The addresses in the MWr TLP refer to the DMA address of the skbuf or mbuf in host memory.

PCIe MWr transactions are simple *posted* transactions. In contrast, PCIe memory reads (MRd), e.g., a device reading data from the host, are *non-posted* transactions and require two TLP types: MRd TLPs and CplD TLPs. A MRd TLP is sent to request data from the peer, then the data is returned via one or more CplD TLPs. Thus, PCIe memory reads consume bandwidth in both directions. A MRd request may only request data up a certain amount (Maximum Read Request Size, or MRRS), negotiated between the peers. A typical value for the MRRS is 512B. The data returned via CplD TLPs is bounded by the MPS. With this data in hand we can build a model to calculate effective PCIe bandwidth for transferring data to and from the host from a PCIe device for different PCIe generations and configurations. The PCIe configuration, e.g., PCIe gen3 device with 8 lanes (x8), provides us with the available bandwidth and the values for MPS and MRRS. We can then calculate the number of bytes transferred for a DMA write of size sz with:

$$B_{tx} = \lceil sz/MPS \rceil \times MWr_Hdr + sz \quad (1)$$

where MWr_Hdr is 24B (2B framing, 6B DLL header, 4B TLP header, and 12B MWr header). A DMA read of size sz transfers data in both directions:

$$B_{tx} = \lceil sz/MRRS \rceil \times MRd_Hdr + sz \quad (2)$$

$$B_{rx} = \lceil sz/MPS \rceil \times CplD_Hdr + sz \quad (3)$$

with MRd_Hdr being 24B and $CplD_Hdr$ 20B.

The graph labelled *Effective PCIe BW* in Figure 1 was calculated using this model assuming a PCIe gen3 x8 device, $MPS = 256B$, and $MRRS = 512B$ using 64bit addressing. This configuration is typical for modern 40Gb/s NICs in Intel servers. The saw-tooth pattern shows the overhead of the additional DLL/TLP headers for every MPS bytes of data, with the overhead being higher for smaller transfer sizes. The graph also shows the impact of MRRS as the additional MRd TLPs consume bandwidth which otherwise would

be usable by MWr transactions. In addition, the model allows to calculate PCIe overheads for more complex device/host interactions by calculating the bytes transferred in each direction for multiple PCIe operations. For example, for the graph of the *Simple NIC* in Figure 1 we calculate the PCIe bandwidth used both for packet send (TX) and receive (RX) of a very simple NIC. For TX, we assume the following steps:

- (1) Host writes the TX queue tail pointer (4B PCIe write).
- (2) Device DMA's the TX descriptor from the host (16B PCIe read).
- (3) Device DMA's the packet buffer to the host (PCIe read).
- (4) Device generates an interrupt (4B PCIe write).
- (5) Host reads the TX queue head pointer (4B PCIe read).

Likewise for packet RX, we assume the following steps:

- (1) Host writes the RX Queue Tail Pointer to enqueue a buffer on the freelist (4B PCIe write).
- (2) Device DMA's freelist descriptor (16B PCIe read).
- (3) Device DMA's the packet to the host (PCIe write).
- (4) Device DMA's the RX descriptor to the host (16B PCIe write).
- (5) Device generates an interrupt (4B PCIe write).
- (6) Host reads RX queue head pointer (4B PCIe read).

By calculating the PCIe protocol overheads for each of these transactions and then add up the consumed bytes we can model the achievable throughput. The NIC model described above is very simplistic and all modern NICs deploy a variety of optimizations. For example, the Intel Niantic NIC [3] DMA's batches of up to 40 TX descriptors from the host and may write batches up to 8 TX descriptors back to the host after packets have been transmitted. The graph labelled *Optimized NIC (kernel driver)* in Figure 1 shows the performance impact of these optimizations over the simple NIC model. Throughput can be further improved by driver changes, as shown by *Optimized NIC (DPDK driver)* in the same figure. The Intel DPDK driver configures the device slightly different as no interrupts are generated and the driver does not read device registers to determine which packets have been received/transmitted⁷. The graphs for the optimized NIC show that even a moderate amount of

⁷The DPDK driver polls on descriptor queues in host memory instead, checking for the valid fields indicating that a descriptor has been updated by the device.

optimization on both the device and the driver side can significantly improve the achievable throughput of a NIC.

Our PCIe model is not limited to calculating achievable throughput for NICs. It can equally be used to model RDMA NICs or, in fact any PCIe device. Modelling a device typically requires careful study of the manufacturers data sheets and drivers to extract the PCIe transactions performed and is particularly suited to quickly assess the impact of design alternative when implementing custom NIC functionality.

The model has some limitations. Some of the lower level overheads, in particular for flow control messages are only estimated based on the PCIe specification and the model likely slightly overestimates their impact. The model also does not account for PCIe overheads of unaligned accesses, in particular for DMA reads. For these transactions the specification requires the first CplD to align the remaining CplDs to a advertised Read Completion Boundary (RCB), which is typically 64B. Thus, PCIe reads may generate an additional TLP depending on the address and the requested transfer size. Finally, the model only considers the PCIe overheads and does not take into account any limitations imposed by the host or the device’s DMA engines. We evaluate these overheads in practice in the next sections.

4 PCIe-BENCH: A METHODOLOGY FOR BENCHMARKING PCIe

While the PCIe protocol stack, described in the previous section, is relatively easy to understand and model, it is the implementation of the DMA engine and, in particular, the increased complexity of the PCIe root complex, discussed in §2, which make it hard to evaluate the end to end PCIe performance in real systems. Therefore, we designed a set of PCIe micro-benchmarks in the spirit of *lmbench* [40] and *hbench:OS* [15]. The main idea is to perform individual PCIe operations while carefully controlling the parameters which may affect performance.

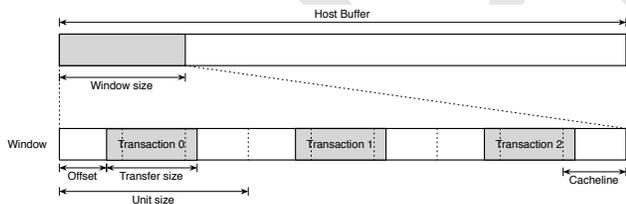


Figure 4: Host buffer layout and access parameters for `pcie-bench`

Figure 4 illustrates the host buffer setup and the parameters which determine its access from a PCIe device. First, we define a (logically) contiguous buffer on the host that is accessible from a PCIe device. The buffer may be contiguous in DMA address space or comprised by a number of smaller buffers. Moreover, it must be significantly larger than the size of the Last Level Cache (LLC), because, on some architectures, the PCIe root complex interfaces with the cache system of the CPUs. To measure cache effects, only a subset of the host buffer, the *window size*, is accessed repeatedly.

For a given micro-benchmark we repeatedly access the window size of the host buffer with multiple DMA requests and we keep the

amount of data transferred with each request fixed (*transfer size*). A DMA may start at an *offset* from a host cache line, allowing us to determine if there are any host side penalties for unaligned accesses and to evaluate the effectiveness of a PCIe devices’ DMA engine in handling unaligned DMAs. To ensure that each DMA request touches the same number of cache lines, the window is divided into equal sized units. Each unit is the sum of offset and transfer size, rounded up to the next cache line. A micro-benchmark can typically be configured to access the units sequentially or randomly (*access pattern*), though for most experiments in this paper we use random access.

For each benchmark run, the *state of the CPU cache* needs to be carefully controlled. By default, the cache is thrashed before every benchmark to ensure a *cold* cache. Optionally, we may attempt to warm the cache from the host, by writing to the window (*host warm*), or from the device. The latter is achieved by issuing a number of write DMAs to the window (*device warm*), before running tests. Understanding the impact of the host cache system is a fundamental aspect in NICs characterization. When transmitting packets from the host, typically, at least the packet header would be cache resident, and when receiving from the network, the packet data may get DMAed to non-cache resident buffers, depending on the overall pressure on the caches.

Finally, the *locality* of the host buffer needs to be controlled. In SMP systems with integrated PCIe and memory controller, the entire host buffer is either allocated local to the node the PCIe device is attached to, or to a different (non-local) node. Controlling the *locality* allows us to measure the interaction between NUMA architectures and PCIe transactions.

4.1 Latency benchmarks

The first set of the PCIe micro-benchmarks measures the latency of individual PCIe operations. Measuring the latency of a PCIe Memory Read (MRd) transactions (from the device) is relatively easy on most programmable hardware: take a timestamp before issuing a DMA Read and another timestamp when the completion of the DMA Read is signaled by the DMA engine, then logging the time-difference. DMA Read latency benchmarks are labeled `LAT_RD` throughout the paper.

The latency of DMA Memory Writes can not be easily measured directly as PCIe Memory Write (MWr) transactions are *posted*: they are sent without explicit acknowledgment of success, as no completion is signaled back to the device. Instead, we indirectly measure the latency of writes by issuing a DMA Write followed by a DMA Read from the same address. PCIe ordering ensures that the PCIe root complex handles the Read after the Write. Note, this approach does not allow us to compute the latency of DMA Writes alone as the latency of the DMA Read may be affected by the preceding DMA Write. The results from these benchmarks are labeled `LAT_WRRD`.

For latency benchmarks we record the latency for each individual transaction and then reduce the raw data by calculating the average and median latency across the data collected, as well as the min, max, 95th, and 99th percentile.

Latency measurements allow us to assess the cost of cache, IO-TLB misses, or the added latency for accessing remote memory

from a PCIe device, by comparing it across different system configurations. While we can not measure the latency of a DMA Write directly, the latency of a DMA Write followed by a DMA Read to the same address provides insights into the effect of architectural choices such as LLC sharing (Intel’s DDIO).

Measuring latency in this way include the device overheads, such as the cost of issuing DMA requests or receiving notifications of completions. Therefore, when looking at the impact of the PCIe root complex and its implementation, it is more insightful to study the relative change in latencies rather than the absolute latency values.

Latency measurements are of particular importance when writing software for programmable NICs or when implementing DMA engines in (re-configurable) logic. For 128 byte packets on 40 Gb/s Ethernet the frame rate is approximately 33.78 Million packets per second (MPPS), i.e., a packet needs to be DMA’d to and from the host every 30ns to sustain bidirectional line rate. Furthermore, a NIC needs to perform additional DMA requests, such as fetching and writing back descriptors, though these DMA requests can typically be accumulated across multiple packets. It is worth noting that if the latency of individual DMA requests exceed the inter-packet time, the NIC DMA engine and its driving logic must be able to handle multiple DMA requests in flight.

4.2 Bandwidth benchmarks

The second set of PCIe micro-benchmarks focuses on bandwidth. DMA bandwidth is measured by taking a timestamp at the start, issue a significant number of DMAs and take a timestamp at the end. The bandwidth can be calculated by the time difference and the amount of data transferred. We are interested in straight DMA Read and DMA Write bandwidth (labeled BW_RD and BW_WR respectively). To measure bi-directional bandwidth we issue alternating DMA Read and Write transactions (labeled BW_RDWR). This ensures that PCIe MRd TLPs compete with PCIe MWr TLPs for the bandwidth to the root complex.

Bandwidth benchmarks, especially with small transfer sizes and random access patterns, generate a significant load on the system. If the DMA engine can saturate the PCIe link with, for example, 64 byte transfers, this would generate around 69.5 million transactions per second in each direction; this implies that the root complex would have to handle a transaction every 5ns. Therefore, bandwidth micro-benchmarks can expose limitations in the root complex implementation as well as stressing the implementation of a device’s DMA engines for small transfer sizes.

4.3 Summary

This section presented a generic methodology for PCIe micro-benchmarks, which can be implemented on a variety of programmable NICs or in re-configurable hardware. Table 1 summarizes the micro-benchmarks, while Table 2 shows the input parameters for each test.

This suite of micro-benchmarks cover the full range of typical PCIe transactions, as performed by a PCIe device, on the performance critical data-path. A selected subset of combinations of these

Name	Description
LAT_RD	Latency of a DMA Read
LAT_WRRD	Latency of a DMA Write followed by a Read
BW_RD	DMA Read bandwidth
BW_WR	DMA Write bandwidth
BW_RDWR	Alternating DMA Read & Write bandwidth

Table 1: List of micro-benchmarks.

Name	Description
Transfer size	# bytes transferred by each request
Offset	Offset from cache line start
Access pattern	Random or Sequential
Window size	Size of host buffer accessed
Cache state	Cold or warm
NUMA locality	NUMA node the host buffer is located

Table 2: Parameters for each micro-benchmark.

parameters is suitable, and sufficient, to highlight the characteristics of both the implementation of PCIe and its integration with host specific features.

5 IMPLEMENTATIONS

The pcie-bench methodology requires programmatic and fine grained control over PCIe device’s DMA engines. We have implemented the methodology on both commodity and research oriented boards. Two independent implementations of the same methodology validate our performance measurements against different host architectures and provides direct insights into the two different PCIe implementations. This section provides an overview of the implementation choices made for the Netronome and NetFPGA boards.

5.1 Netronome NFP implementations

Netronome offers a number of highly programmable NICs based on the NFP-6000 and NFP-4000 Ethernet controllers [7, 10, 55]. These Ethernet controllers feature up to 120 eight-way multi-threaded Flow Processing Cores (FPCs), a hierarchical memory subsystem with integrated custom engines, and fully programmable PCIe interfaces, all interconnected via a distributed switch fabric. The PCIe subsystem exposes a number of interfaces to the FPCs: A *command* interface allowing the core to issue small PCIe read and write transactions directly from registers and a bulk *DMA* interface allowing FPCs to en-queue large transfers between host memory and memory on the NFP.

The PCIe micro-benchmarks are implemented as firmware on the FPCs utilizing both interfaces exposed by the NFP PCIe subsystem. The micro-benchmarks work both on NFP-4000 and NFP-6000 based controllers. A variant of the micro-benchmarks for the older NFP-32xx generation of controllers [9] is also available.

5.1.1 Firmware. The full PCIe micro-benchmark suite is implemented in a single firmware image. It provides a simple control

interface, allowing a user space program on the host to select which type of benchmark to run and provide the necessary configuration parameters, such as the host buffer location, transfer size etc. Benchmark results are written to NFP memory where they can be easily read back from the host.

Latency tests are run on a single thread in one of the FPCs. This thread simply calculates the next host address to use and prepares a DMA descriptor. The thread then reads the current value of the internal timestamp counter and enqueues the DMA descriptor to the DMA engine. For DMA Reads, the DMA descriptor is configured so that the DMA engine signals the thread once the DMA completed. On reception of the signal, the thread takes another timestamp and journals the difference to memory on the NFP. A variant of the latency benchmarks uses the direct PCIe command interface, which is suitable for small transfer (up to 64 or 128 bytes). With this variant, instead of building and enqueueing a DMA descriptor, the thread can directly issue the PCIe read or write command and gets signaled on completion.

The timestamp counter increments every 16 clock cycles, which, on a 1.2 GHz NFP, provides a resolution of 19.2ns. The journal used for the latency numbers is configured to hold up to 16 million values, but typically we run the latency benchmarks with 2 million transactions.

For DMA bandwidth tests, the challenge is to ensure that DMA descriptors are enqueued to the DMA engine at a higher rate than it drains the queue. To achieve this, even for high transaction rates required when small transfers are happening, we use up to 12 cores with 8 threads as DMA workers. In a loop, each DMA worker thread calculates a host address, prepares and enqueues a DMA descriptor and yields until the completion is signaled. An atomic counter, shared between all worker threads, is decremented by each worker before issuing a new DMA request. If the counter value goes below zero, the worker stops issuing DMA requests. For BW_RDWR tests, each worker thread issues a DMA Read if the counter is even and a DMA Write when then counter is uneven.

A control thread is used to start all the worker threads and it waits for them to complete. The control thread reads the timestamp counter before starting the workers and when they finish. The elapsed time together with the transfer size and the number of transactions is used to calculate the bandwidth. For bandwidth tests 8 million DMA requests are performed.

For both DMA latency and bandwidth tests the DMA descriptors are configured to target an internal 256 KB SRAM, known as the Cluster Target Memory. This memory has an access latency of 50-100 cycles. Enqueueing DMA descriptors incurs a similar latency. These latencies add a fixed cost to each DMA transfer, and while they are specific to the NFP, we expect other programmable NICs to have similar overheads.

The firmware is implemented in Micro-C, a NFP specific extension to C. The micro-benchmark suite is implemented with around 1500 lines of code, with the core requiring around 500 lines of code. The code can be compiled, without external dependencies, to run on NFP6000, NFP4000, and NFP3200 based networking cards.

5.2 NetFPGA

NetFPGA is an open-source community platform [6]. It is supported by cutting-edge reconfigurable boards, reference software and hardware designs, as well as the infrastructure for design development and verification. The latest generation of NetFPGA board, NetFPGA-SUME, is a PCIe host adapter card [58] capable of 100Gb/s application speeds. The board utilizes a large Xilinx Virtex-7 FPGA device incorporating two PCIe Gen 3 *hard-blocks*, along with resources such as QDRII+ and DDR3 memory types.

5.2.1 The hardware. The PCIe micro-benchmark suite is implemented directly on the FPGA. It enhances the DMA engine described in [57] with `pcie-bench` capabilities. The system provides a simple control interface, allowing the host to select which type of micro-benchmark to run along with its parameters. In particular, the software can configure the operation to perform (e.g., read, write), the base address of the host buffer to access, the transfer size, access pattern (i.e., sequential or random addresses) and define the number of repetitions for it. The system keeps track of time through a free-running counter, operating at PCIe core frequency (250MHz), thus providing a resolution of 4ns.

Every time the software triggers a new micro-benchmark, a finite state machine coded into the hardware, calculates the host addresses and generates the associated memory read or write request. The design does not use any kind of FIFO to enqueue DMA requests, instead the DMA requests are directly passed to the DMA engine. All the memory requests are generated on-the-fly as the hardware design allows transmission of a new request every clock cycle.

For latency tests, the system is configured so that for DMA Read operations, two timestamps are taken: when the operation is requested, and when the acknowledgment signal is received. The system records up to 1000 transaction latency values. For bandwidth tests the challenge is to stress the DMA engine as much as possible. In this scenario, 1 million transactions are performed, and the system takes the total time: the time that has passed from the first request until the last completion (in read or write/read mode) or the last request (in write mode). Benchmark results are written to NetFPGA memory after a benchmark run, where they can be easily read back from the host.

The FPGA design is written in Verilog and System Verilog. The micro-benchmark suite is implemented with around 1200 lines of code. It can be compiled directly on NetFPGA-SUME and Xilinx VC709 boards.

5.3 Kernel drivers

Both the NFP and the NetFPGA implementations use a kernel driver to initialise the hardware, allocate host memory for DMAs and provide access to user space programs to control the execution of the benchmarks and to collect the results.

The NFP `pcie-bench` driver uses the standard NFP kernel driver⁸. It allocates the host side DMA buffers in chunks of 4MB as this the maximum size which can be allocated physically contiguous on most Linux kernel versions. The NetFPGA driver allocates memory either from `hugetlbfs` (with 1GB pages) or standard 4KB system pages. `hugetlbfs` is the default option as it allows for the easy

⁸<https://github.com/Netronome/nfp-driv-kmods>

allocation of large, physically contiguous areas of memory. Both drivers provide control over which NUMA node memory is allocated from and export an interface allowing a user space program to warm the caches with the controlled portions of the host DMA buffers.

The NFP `pcie-bench` kernel module is implemented in around 400 lines of code while the NetFPGA driver is implemented in approximately 800 lines of code.

5.4 Control programs

The execution of benchmarks, gathering of the data and post-processing of the results is performed by user space programs in both implementations. The NFP implementation is using a Python program with a small utility, written in C, to handle cache warming. The Python program can be used to run individual tests or a full suite of tests (with combinations of parameters from Tables 1 and 2). A complete run takes about 4 hours and executes around 2500 individual tests. For latency benchmarks, the control program reads the timing data of the individual transactions and calculates various metrics, such as the average, median, min, max and 95th and 99th percentile. Optionally, it generates CDFs, histograms and time series of the data. For bandwidth tests, the control program calculates the bandwidth and transaction rate. It is implemented in 1600 lines of Python and 120 lines of C code.

The NetFPGA control program is written in approximately 600 lines of C code and provides a command line interface to the control the individual test parameters. The raw test results are written to a file for further processing. The control program is easily scripted to provide results for combinations of parameters.

6 EXPERIMENTAL EVALUATION

This section reports the `pcie-bench` suite in action. Table 3 lists the details of the systems used for the evaluation results presented in this paper. We focus on systems built around Intel’s Xeon E5 processors as this is the most common configuration used in today’s datacenter servers [51]. We also discuss results obtained comparing several generations of processors.

The `pcie-bench` methodology is equally applicable to evaluate earlier generations of PCIe; we focus upon PCIe gen 3 as, at time of writing, the most widely deployed version. The next generation PCIe gen 4 is in the final stages of ratification and we expect that `pcie-bench` can be applied there once hardware is available.

6.1 Baseline bandwidth and latency

As the first set of results we present the baseline throughput of the NFP-6000 and NetFPGA implementations in Figure 5. We compare them to the required bandwidth for 40Gb/s Ethernet as well as a simplified model of achievable throughput for a PCIe gen 3 device. The model accurately calculates the overheads of the physical, data link, and transaction layer of PCIe but only estimates the overhead of flow control messages. We measure the throughput of PCIe read, write, and alternating read/write transfers for different transfer sizes to a fixed, 8KB host buffer, which is warmed before each test to eliminate any caching effects. We vary the transfer size from 64B to 2048B in rough increments of powers of 2 but taking additional measurements with $-1/+1B$ around important transfer sizes, such

as some cache line or TLP size boundaries. All DMA start addresses are cache line aligned and all tests were performed on the same Xeon E5 2637v3 system to eliminate any differences in system configuration. Figure 5 shows the results.

In all three data sets, the NetFPGA implementation of `pcie-bench` closely follows the PCIe bandwidth calculated with our model. For PCIe writes the NetFPGA implementation achieves a slightly higher throughput though. This is because the model assumes a fixed overhead for flow control messages which, for uni-directional traffic, would not impact traffic throughput. The NFP implementation of `pcie-bench` generally achieves slightly lower throughput than the NetFPGA implementation (but typically achieves throughput sufficient to support 40Gb/s Ethernet rates). The main reason is that the NetFPGA implementation directly drives the DMA engines from the internal FPGA memory and does not perform any additional processing. In the NFP implementation, the DMA engines must also transfer data to/from the host into internal SRAM, and then transfer it further into NFP internal memory where it can be accessed by the FPCs of the NFP card. These additional overheads, as well as the required buffer management and signaling, introduce a slight performance degradation. Finally, it is worth noting that each graph shows that neither implementation is able to achieve a read throughput required to transfer 40Gb/s Ethernet at linerate for small packet sizes.

Next, we look at the latency of individual DMA transactions varying the transfer size (Figure 6). The setup is the same as for the bandwidth results described above.

Overall, the latency numbers for both the NFP-6000 and NetFPGA are of the same order of magnitude, indicating that the bulk of the latency can be attributed to general PCIe and overall host overheads. It is very similar across the four generations of Intel processors we measured. The latency is also significantly higher than the inter-frame arrival rate on 40Gb/s Ethernet. For example, the median latency for a 128B DMA Read is 626ns with the NFP-6000 and 529ns with the NetFPGA, while a new 128B frame would have to be DMAed from the host around every 30ns to saturate the link. This shows that the respective DMA engines have to handle at least around 20 DMA Read transactions in flight to mask the latency.

The latency for DMA requests is higher on the NFP-6000 with an initial fixed offset of about 100ns for smaller transfers, and the gap increasing for larger transfers. The reasons are twofold. The fixed offset can be explained with the overhead of enqueueing DMA descriptors to the DMA engines, which is avoided in the NetFPGA implementation. When using the NFP’s direct PCIe command interface, designed for smaller transfers, the NFP-6000 achieves the same latency as the NetFPGA, further indicating that the majority of the latency can be attributed to host system. The widening of the gap as the transfer size increases can be explained by considering the internal architecture of the NFP (§ 5.1). Every DMA transfer to/from the host causes an additional transfer from the internal SRAM to NFP internal memory. This additional transfer increases the latency depending on the transfer size, but it is useful for decoupling the DMA request from the (often variable) packet processing times performed by the FPCs.

Name	CPU	NUMA	Architecture	Memory	OS/Kernel	Network Adapter
NFP6000-BDW	Intel Xeon E5-2630v4 2.2GHz	2-way	Broadwell	128GB	Ubuntu 3.19.0-69	NFP6000 1.2GHz
NetFPGA-HSW	Intel Xeon E5-2637v3 3.5GHz	no	Haswell	64GB	Ubuntu 3.19.0-43	NetFPGA-SUME
NFP6000-HSW						NFP6000 1.2GHz
NFP6000-IB	Intel Xeon E5-2620v2 2.1GHz	2-way	Ivy Bridge	32GB	Ubuntu 3.19.0-30	NFP6000 1.2GHz
NFP6000-SNB	Intel Xeon E5-2630 2.3GHz	no	Sandy Bridge	16GB	Ubuntu 3.19.0-30	NFP6000 1.2GHz

Table 3: System configurations. All systems have 15MB of LLC, except NFP6000-BDW, which has a 25MB LLC.

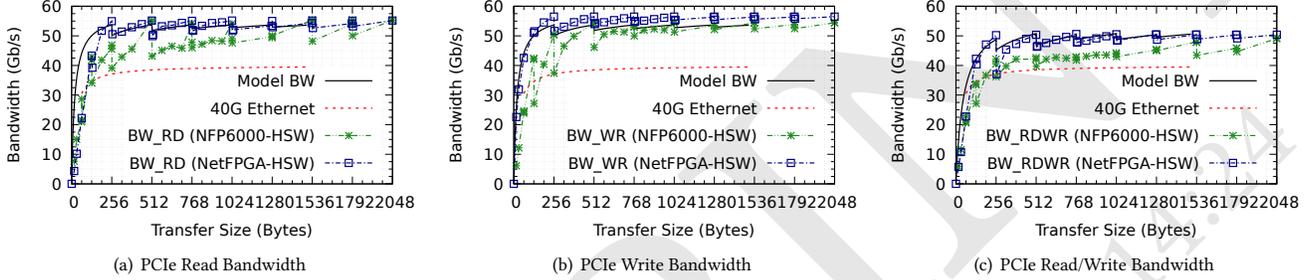


Figure 5: Baseline PCIe DMA bandwidth for NFP6000-HSW and NetFPGA-HSW with warm caches.

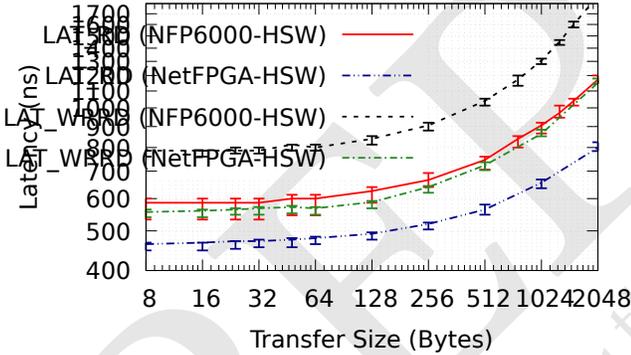


Figure 6: Median, minimum, and 95th percentile of DMA latency for NFP-6000 and NetFPGA.

6.2 Caching and DDIO

We now examine the impact of caching and DDIO on PCIe transactions. For these experiments we keep all parameters constant, except for the window size and the state of the LLC. The window size was varied from 4KB to 64MB, exceeding the 15MB or 25MB size of the LLC on all our systems.

Firstly, we look at PCIe transaction latency. To measure the latency we use the NFP’s PCIe command interface with 8 Byte transactions, as it is the lowest latency operation we have available. We ensure that each subsequent transaction touches a different cache line as described in Section 4.1. The results are shown in Figure 7(a).

For PCIe reads (LAT_RD) with a cold cache we see no changes in latency as we change the window size: All PCIe reads are serviced

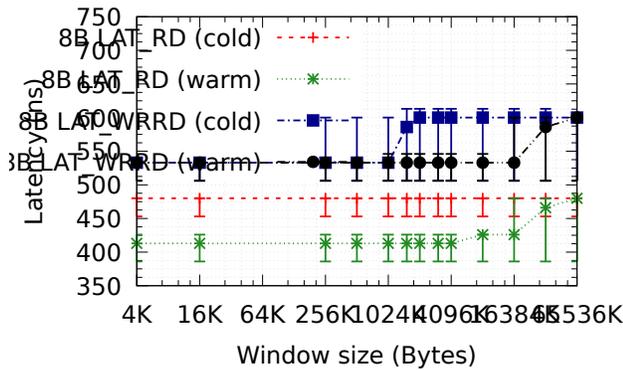
from memory. When the cache is warm, the read latency is around 70ns lower but increases once the window size exceeds the size of the LLC. This confirms that PCIe reads are serviced from the LLC if the data is resident in the cache.

With a warm cache, the latency of a posted PCIe write followed by a read (LAT_WRRD) follows roughly the same pattern: writes and reads hit the LLC and, once the window size exceeds the LLC size, the latency increases by around 70ns. For a cold cache the LAT_WRRD results illustrate the effect of DDIO: For small window sizes new cache lines get allocated and writes (and subsequent reads) are performed to/from the cache. Once the window size exceeds the 10% of the LLC reserved for DDIO, dirty cache lines have to be flushed out to memory before the write can succeed, causing a 70ns delay for most writes. For larger transfer sizes, the differences between hitting the cache or not is reduced significantly. For example, for 64B LAT_WRRD tests the difference is around 10ns.

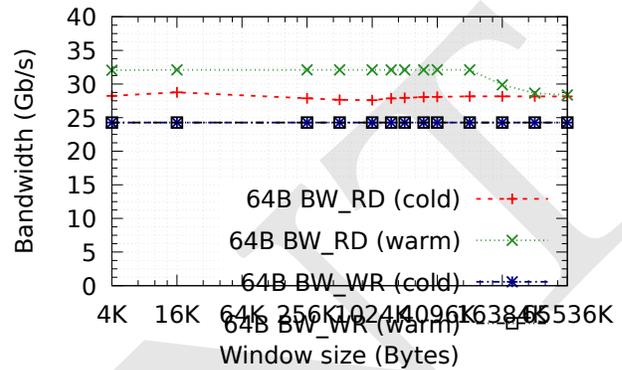
The bandwidth data, presented in Figure 7(b), suggest a similar story. For 64B DMA Reads (BW_RD), there is a measurable benefit if the data is already resident in the LLC. For larger transfer sizes (not shown) the benefit is smaller, and from 512B DMA Reads onwards, there is no measurable difference. For DMA Writes (BW_WR), there is no benefit if the data is resident in the cache or not. There is also not measurable benefit on keeping the windows size below the 10% of the LLC. We suspect that the DDIO portion of the cache is cleaned quick enough, so that all DMA Writes either hit the main LLC or the DDIO portion. Since DDIO can not be disabled and there don’t appear to be dedicated performance counters it is not possible to validate this hypothesis.

6.3 NUMA impact

With the PCIe root complex and memory controllers integrated in each CPU node, a PCIe device may access memory local or remote



(a) Median latency (with minimum and 95th percentile)



(b) Bandwidth

Figure 7: Cache effects on Latency and Bandwidth (NFP6000-SNB)

to the node to which it is attached. In this section we evaluate the performance impact of this non-uniform memory access. The host buffer is allocated either on the node the PCIe device is attached to (local) or on the other node in a 2-way NUMA system (remote). The cache is warmed or thrashed on the node the buffer is allocated from.

the conclusion in Section 6.2, and in contrast with current Intel specification [4], we believe that all DMA Writes may be initially handled by the local DDIO cache.

For completeness, the latency numbers do not differ much from the local case presented in Figures 6 and 7 except that remote accesses add a constant 100ns of latency to our systems. Overall, the results from our two dual socket systems (NFP6000_BDW and NFP6000_IB) are the same, indicating there has been little change in the two generations separating them.

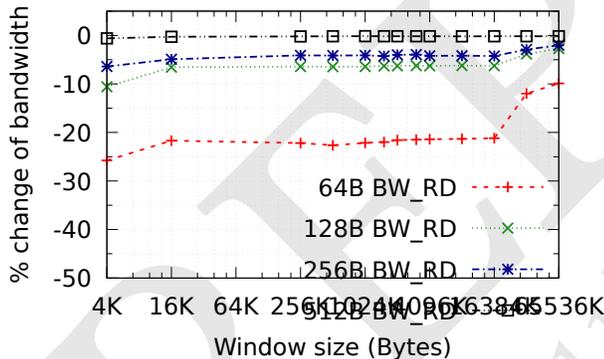


Figure 8: Difference between local and remote DMA Reads of different sizes with warm cache (NFP6000-BDW).

Figure 8 shows the percentage change for DMA read bandwidth of local memory versus remote memory for different transfer sizes across different window sizes with a warm cache. 64B DMA reads experience a 20% drop in throughput (from ~32Gb/s to ~25Gb/s). The difference drops to around 10% once the DMAs are not serviced from the local cache anymore. With 128 and 256B DMA reads the penalty for accessing remote memory drops to 5-7% (e.g., from ~44Gb/s to ~41Gb/s for 128B). There is no noticeable penalty for 512B DMA reads. The data is confirmed by cold cache DMA read throughput (not shown) where remote 64B reads experience a constant 10% penalty (~5% for 128B and 256B reads).

The throughput of DMA Writes does not seem to be affected by the locality of the host buffer nor by the size of the host buffer. Like

6.4 The IOMMU effect

Finally, we look at the impact of an IOMMU interposed in the datapath between PCIe devices and the host. The IOMMU performs address-translation for addresses present in TLPs and utilizes a TLB as a cache for translated addresses. On a TLB miss, the IOMMU must perform a full page table walk, which may delay the TLP and thus may increase latency and impact throughput.

To measure the impact of the IOMMU, we enable it on the Linux kernel command line with `intel_iommu=on`. IOMMU implementations in Ivy Bridge and newer architectures support super-pages to reduce the number of page table entries, and thus reduce the number of TLB entries. For our experiments we disable this behavior (by also specifying `sp_off` on the kernel command line). This forces the use of 4KB page table entries and allows us to use a relatively small 64MB host buffer for our experiments.

Figure 9 shows the result for different transfer sizes. Like with the data presented in Section 6.3 the data is represented as a percentage change to the same experiment run without IOMMU. For small window sizes, there is no measurable differences across the range of transfer sizes. However, once the window size is increased beyond 256KB, the throughput drops dramatically. For 64B DMA Reads it drops by almost 70% and even for 256B DMAs the drop is still a significant 30%. There is no change for transfer sizes for 512B and above. For DMA Writes (not shown), the drop is not quite as dramatic (55% for 64B writes), but is still very significant.

From this data we conclude that on that particular system the IO-TLB has 64 entries (256KB/4KB), a number not made publicly available by Intel. The latency of 64B Reads increases from around

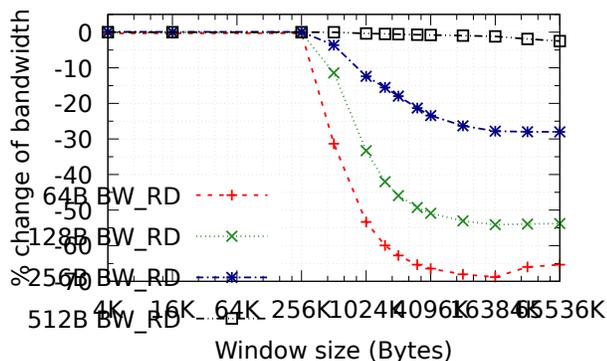


Figure 9: Impact of IOMMUs on DMA reads of different transfer sizes with warm caches (NFP6000-BDW).

430ns to 760ns, putting the cost of a IO-TLB miss and subsequent page table walk at around 330ns. For smaller transfer sizes this penalty is relatively high compared to the time it takes to transfer the data, and thus the impact on the throughput is higher still. The data is surprisingly consistent across all 4 generations of Intel micro-architectures where we ran the experiments, and we observe the same effects with our NetFPGA `pcie-bench` implementation. With such consistent results across the 4 micro-architecture generations we must conclude this aspect has undergone little development since its first implementation.

7 LESSON LEARNED AND USE CASES

The results obtained from `pcie-bench` can (and have) been used to characterize and tune system software for high performance I/O. There are a plethora of sophisticated tools to analyze operating system and application performance, e.g., [21, 25], as well as tools to understand the performance impact of host CPU architectures and OS primitives, e.g., [15, 40]. However, none of these tools provide detailed insights into the performance of the I/O subsystem. `pcie-bench` provides the necessary additional data to fine-tune specialized network stacks [38, 39], and optimize kernel IOMMU handling [44].

Table 4 reports the notable findings from this paper which have been derived experimentally with `pcie-bench`. Specifically, based on the IOMMU data, we strongly recommend using super-pages and trying to force the IO buffers used for DMA to be co-located in as few super-pages as possible. This may be possible in carefully controlled environments, such as virtualized network appliances. However, in multi-tenant Virtual Machine (VM) environments offering assigned PCIe devices to VMs, it is currently not possible to isolate the IO performance of VMs sufficiently with Intel’s IOMMUs.

For NUMA architectures, we found that small DMA reads from the remote cache are significantly more expensive than reads from the local cache. Accessing data on a remote node also adds around 100ns of latency. While it may not be feasible to “pin” small network packets to the local node, it would certainly be useful to locate data structures, such as descriptor rings, on the local node. Our data suggests that, for larger packet sizes, the locality of the packet buffer

is not critical and it is recommended to allocate data on the nodes where the processing happens.

Finally, our measurements confirm the documented operation of DDIO and show that, for small transfers, accesses are around 70ns faster for cache resident data. We see two areas where this can be beneficial for network applications. Firstly, access to descriptor rings is lower latency, and therefore incur less overhead. Secondly, the integration with the caches should benefit small packet receive, in particular for packet sizes which are not multiples of a cacheline (e.g., 64B Ethernet frames with the 4B FCS stripped). Since the data is directly DMAed into the cache, dirty cachelines do not need to be written back to memory before the DMA can succeed.

While the above lessons focus on optimizing host system software, the insights gained with `pcie-bench` have also implications for the growing area of research moving beyond straightforward TCP offload to offloading more complex processing to programmable and reconfigurable network cards, e.g., [12, 28, 47, 48]. In order to implement the firmware or reconfigurable logic for these application-offloads, it is important to have a detailed characterization of the DMA engines and their interaction with the host system. The data provided by `pcie-bench` is ideal to guide such implementation choices. For example, understanding the latency of transactions for the Netronome boards has heavily influenced the design of multiple firmware implementations that offer different offloads of network processing. The latency data determines how many in-flight DMAs the firmware has to handle (for both packet data DMA and descriptor DMAs) to sustain line rate. In turn, this determines the sizing of I/O structures such as rings and other storage for in-flight DMAs along with specifying the appropriate number of Flow Processing Cores and threads. The latency and bandwidth data also determines the amount of batching performed both in the firmware and the corresponding device driver in the operating system. For example, on the NFP6000-HSW system, it takes between 560-666ns to transfer 128B of data from the host to the device. At 40Gb/s linerate for 128B packets, a new packet needs to be transmitted every 29.6ns. Given the latency, this means that the firmware (and the DMA engines) need to handle 30 transactions in flight. These calculations can be extended to take into account the latency for descriptor transfers and to work out the cycle budget for each DMA.

Finally, the `pcie-bench` methodology and measurements can be used by silicon engineers to evaluate existing designs and inform the objectives of future architecture of DMA engines. For example, the PCIe-latency measurements were used to assess each iteration of a suite of NetFPGA DMA-engine designs. Being able to compare the current design with data from other implementations helps to determine if a performance bottleneck is due to artifacts in the host architecture or to a design limitation. More importantly, the methodology was also extensively used for validation during chip bring-up, and to guide architectural decisions in future silicon by a well-known network hardware manufacturer. Using micro-benchmarks, as provided by `pcie-bench`, are ideal as they provided detailed and controlled data which is not obtainable by other means.

Name	Notable observed effects	Evaluation-directed Recommendation
DDIO (Sec.6.2)	For small transfers, accesses are around 70ns faster when data is cache resident	DDIO improves descriptor ring access delay and improves performance for small packet receive
NUMA (Sec.6.3) (small transactions)	The (predictably) high cost of small DMA reads from remote cache being more expensive than from local cache	Place descriptor rings on the local node
NUMA (Sec.6.3) (large transactions)	No significant difference between remote and local cache performance	Place packet buffer on the node where processing happen
IOMMU (Sec.6.4)	Significant throughput drops as working-set size increases	Co-locate IO buffers into fewer superpages

Table 4: Notable findings from this paper, derived experimentally with pcie-bench.

8 RELATED WORK

Performance has been a driving force for change, evolution, and improvement within the interconnect standards with particular contributions forth coming from the hardware manufacturers [2, 4].

There has been little micro-benchmark work on PCIe beyond Miller *et al.* [41] who describes an approach that measures PCI and first generation PCIe interconnect latency through the differential analysis of transaction delays. Moll *et al.* [42] demonstrated the use of simple FPGA based hardware for the profiling of software applications including an analysis of PCI bus performance, while Endo *et al.* [19] demonstrate the importance of latency in interactive and transactional application software. In addition, they introduce a new set of micro-benchmarks to assess the performance against latency and encourage further, application-specific study of PCI interconnect latency impact.

Several papers have observed PCIe and its interactions with host architecture in the context of higher performance network applications. Kalia *et al.* [26] provide a low level evaluation and recommendation for RDMA primitives and how they interact with PCIe. Li *et al.* [33] discuss the benefits of DDIO on the performance of modern Key-Value-Store (KVS) application. The impact of NUMA on device DMA was discussed by Lim *et al.* [35] in relation to high performance KVS applications and by Han *et al.* [24] for GPU accelerated packet processing. All these works touch on some aspects covered in this paper but in the context of some higher level application. In contrast, pcie-bench allows for a systematic study of PCIe performance characteristics. The results obtained can then be used to explain the higher level application performance artefacts in more details and may guide future application level performance improvements more accurately.

Examining specialist cases of PCIe, Lostrie *et al.* [37] presents a setup for benchmarking the performance of the full PCIe-communication path between two FPGA-boards with microTCA. While, Koop *et al.* [29] evaluates InfiniBand on a PCIe 2.0 system. The authors study the benefits of PCIe 2.0 on both DDR and QDR data rates on the Mellanox ConnectX and also investigate the general trend of additional interconnect bandwidth upon application-performance on multi-core machines. Earlier still, Liu *et al.* [36] evaluates the third generation InfiniBand HCAs from Mellanox, which supports PCIe interfaces. They compare the performance of these with HCAs using PCI-X interfaces. The evaluation consists of a set of micro-benchmarks at the interconnect level, including latency, bandwidth,

and bidirectional bandwidth experiments. They show that InfiniBand HCAs with PCI Express interfaces deliver excellent performance. However, the methodology is only superficially similar to our-own with a focus firmly on InfiniBand performance.

Finally, Braithwaite [14] and Li *et al.* [34] investigate I/O bandwidth in NUMA architectures. The former presents a method for analyzing main memory and PCIe data-access characteristics of modern AMD and Intel NUMA architectures. They also present the synthesis of data-access performance models designed to quantify the effects of these architectural characteristics on bandwidth. Li *et al.* further contributes with a characterization of the state-of-the-art NUMA hosts, and propose a methodology to simulate I/O operations using memory semantics, and in-turn model the I/O bandwidth performance. In both cases, these efforts provide little insight into the explicit and evolving relationship between PCIe and the modern NUMA architecture.

9 CONCLUSION

The PCIe subsystem behavior can enable, or impede, high-performance networking. Although a number of tools exist to analyze the interactions between applications and operating systems, the I/O subsystem largely lacks such a detailed insight. Specifically, it has seen limited study of latency and throughput when integrated within complex modern host architectures.

This paper contributes a methodology permitting systematic evaluation and characterization of PCIe devices, including their DMA engines, and their critical interaction with the host side PCIe root complex and memory subsystem. Furthermore, we describe two open source implementations of such a methodology: one on a commodity Netronome board, and the second on the research-oriented NetFPGA board. We also make the resulting hardware and software tools, pcie-bench, available as open-source to the wider community.

Beyond the pcie-bench design and implementation, we discuss our own characterization results. The highlights include reporting the impact of recent four Intel generations over nine years, architectural innovations of host CPU and supporting chipsets on PCIe performance. We share lessons learned with implications for current software systems and future hardware design. Our study allows exploration of the impact of new PCIe and host architecture features such as DDIO and IOMMUs. Specifically, we demonstrate that the PCIe integration with caches of DDIO works well but, also characterize the significant, and sometimes undesirable, impact

that Intel IOMMUs and NUMA may have when high DMA rates are necessary.

REFERENCES

- [1] DPDK official website. (Jan 2018). <http://www.dpdk.org>.
- [2] Hardware level I/O benchmarking of PCI Express. (Jan 2018). <https://www.yumpu.com/en/document/view/7051564/hardware-level-io-benchmarking-of-pci-express-intel>.
- [3] Intel 82599 10 GbE Controller Datasheet. (Jan 2018). <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf>.
- [4] Intel Data Direct I/O technology (Intel DDIO): a primer. (Jan 2018). <http://www.intel.co.uk/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf>.
- [5] Intel virtualization technology for directed I/O. (Jan 2018). <http://www.intel.co.uk/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf>.
- [6] NetFPGA official website. (Jan 2018). <https://www.netfpga.org>.
- [7] NFP-4000 theory of operation. (Jan 2018). https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf.
- [8] PF_RING repository. (Jan 2018). https://github.com/ntop/PF_RING.git.
- [9] Product Brief: Netronome NFP-32xxx. (Jan 2018). https://www.netronome.com/static/app/img/products/silicon-solutions/PB_NFP-32xx_IC.pdf.
- [10] Product Brief: NFP-6000 intelligent Ethernet controller family. (Jan 2018). https://www.netronome.com/static/app/img/products/silicon-solutions/PB_NFP6000.pdf.
- [11] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling End-Host Network Functions. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
- [12] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling End-Host Network Functions. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
- [13] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. 2012. On multi-gigabit packet capturing with multi-core commodity hardware. In *Passive and Active Measurement (PAM)*. Springer.
- [14] Ryan Karl Braithwaite. 2013. *NUMA data-access bandwidth characterization and modeling*. Master's thesis. Virginia Polytechnic Institute and State University, US.
- [15] Aaron B. Brown and Margo I. Seltzer. 1997. Operating system benchmarking in the wake of Lmbench: a case study of the performance of NetBSD on the Intel x86 architecture. In *Special Interest Group for the Computer Systems Performance Evaluation Community (SIGMETRICS)*. ACM.
- [16] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. 2012. Fabric: A Retrospective on Evolving SDN. In *Hot Topics in Software Defined Networks (HotSDN)*. ACM.
- [17] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. 2011. ETM: A Scalable Fault Tolerant Network Manager. In *Networked Systems Design and Implementation (NSDI)*. USENIX.
- [18] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. 1994. Experiences with a High-speed Network Adaptor: A Software Perspective. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*. ACM.
- [19] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. 1996. Using latency to evaluate interactive system performance. In *Symposium on Operating Systems Design and Implementation (OSDI)*. ACM.
- [20] Daniel Firestone. Building Hardware-Accelerated Networks at Scale in the Cloud. <https://conferences.sigcomm.org/sigcomm/2017/files/program-kbnets/keynote-2.pdf>. ([n. d.]). [Keynote at ACM SIGCOMM 2017 Workshop on Kernel-Bypass Networks; Online; accessed 03-Sep-2016].
- [21] Brandon Greg. Linux Performance. (Jan 2018). <http://www.brendangregg.com/linuxperf.html>.
- [22] Prabhat K. Gupta. Xeon+FPGA Platform for the Data Center. (Jan 2018). <https://www.ece.cmu.edu/~calcm/car/lib/exe/fetch.php?media=car15-gupta.pdf>.
- [23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A Software NIC to Augment Hardware. In *Technical Report No. UCB/EECS-2015-155*. University of California at Berkeley.
- [24] Sangjin Han, Keon Jang, Kyoungsoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated Software Router. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
- [25] Intel. Intel VTune Amplifier 2017. (Jan 2018). <https://software.intel.com/en-us/intel-vtune-amplifier-xe>.
- [26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Annual Technical Conference (ATC)*. USENIX.
- [27] Thomas Karagiannis, Richard Mortier, and Antony Rowstron. 2008. Network Exception Handlers: Host-network Control in Enterprise Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
- [28] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.
- [29] Matthew J. Koop, Wei Huang, Karthik Gopalakrishnan, and Dhableswar K. Panda. 2008. Performance analysis and evaluation of PCIe 2.0 and quad-data rate InfiniBand. In *Symposium on High Performance Interconnects (HOTI)*. IEEE.
- [30] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): an overview. In *acmqueue*. ACM.
- [31] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael Swift, and T.V. Lakshman. 2017. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Symposium on Cloud Computing (SoCC)*. ACM.
- [32] Ki Suh Lee, Han Wang, and Hakim Weatherspoon. 2013. SoNIC: Precise Realtime Software Access and Control of Wired Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.
- [33] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *International Symposium on Computer Architecture (ISCA)*. ACM.
- [34] Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, and Thomas Robertazzi. 2013. Characterization of Input/Output bandwidth performance models in NUMA architecture for data intensive applications. In *International Conference on Parallel Processing (ICPP)*. IEEE.
- [35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Networked Systems Design and Implementation (NSDI)*. USENIX.
- [36] Jiuxing Liu, Amith Mamidala, Abhinav Vishnn, and Dhableswar K. Panda. 2004. Performance evaluation of InfiniBand with PCI Express. In *Symposium on High Performance Interconnects (HOTI)*. IEEE.
- [37] K. Lostrie, P. De Meulenaere, M. Temmerman, N. Van Remortel, and W. Beaumont. 2013. Benchmarking of PCIe-performance in microTCA-equipment. In *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. IEEE.
- [38] Ilias Marinos, Robert Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
- [39] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk/Crypt[Net]: Rethinking the Stack for High-performance Video Streaming. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
- [40] Larry McVoy and Carl Staelin. 1996. Lmbench: portable tools for performance analysis. In *Annual Technical Conference (ATC)*. USENIX.
- [41] David J. Miller, Philip M. Watts, and Andrew W. Moore. 2009. Motivating future interconnects: a differential measurement analysis of PCI latency. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM.
- [42] L. Moll and M. Shand. 1997. Systems performance measurement on PCI pamette. In *Symposium on FPGA-Based Custom Computing Machines (FCCM)*. IEEE.
- [43] PCI-SIG. PCI Express Base Specification Revision 3.1. (2014).
- [44] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafir. 2015. Utilizing the IOMMU Scalably. In *Annual Technical Conference (ATC)*. USENIX.
- [45] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. 2009. Extending Networking into the Virtualization Layer. In *Hot Topics in Networks (HotNets)*. ACM.
- [46] Ian Pratt and Keir Fraser. 2001. Arsenic: A user-accessible gigabit ethernet interface. In *International Conference on Computer Communications (INFOCOM)*. IEEE.
- [47] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE.
- [48] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. 2014. SENIC: Scalable NIC for End-host Rate Limiting. In *Conference on Networked Systems Design and Implementation (NSDI)*. USENIX.
- [49] Kaushik Kumar Ram, Jayaram Mudigonda, Alan L. Cox, Scott Rixner, Parthasarathy Ranganathan, and Jose Renato Santos. 2010. sNIC: Efficient Last Hop Networking in the Data Center. In *Architectures for Networking and Communications Systems (ANCS)*. ACM.
- [50] Luigi Rizzo. 2012. netmap: a novel framework for fast packet I/O. In *Annual Technical Conference (ATC)*. USENIX.
- [51] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

- [52] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. 2010. SideCar: Building Programmable Datacenter Networks Without Programmable Switches. In *Hot Topics in Networks (HotNets)*. ACM.
- [53] Edward Solari and Brad Congdon. The Complete PCI Express Reference. (2003).
- [54] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. 2013. Managing the Network with Merlin. In *Hot Topics in Networks (HotNets)*. ACM.
- [55] Gavin Stark and Sakir Sezer. 2013. NFP-6xxx – A 22nm high-performance network flow processor for 200Gb/s Software Defined Networking. In *Symposium on High Performance Chips (Hot Chips)*. IEEE/ACM.
- [56] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Symposium on Operating Systems Principles (SOSP)*. ACM.
- [57] Jose Fernando Zazo, Sergio Lopez-Buedo, Yury Audzevich, and Andrew W. Moore. 2015. A PCIe DMA engine to support the virtualization of 40 Gbps FPGA-accelerated network appliances. In *International Conference on Reconfigurable Computing and FPGAs (RECONFIG)*. IEEE.
- [58] Noa Zilberman, Yury Audzevich, Adam G. Covington, and Andrew W. Moore. 2014. NetFPGA SUME: toward 100 Gbps as research commodity. In *Micro*. IEEE.
- [59] Noa Zilberman, Matthew Grosvenor, Diana-Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wojcik, and Andrew W. Moore. 2017. Where Has My Time Gone?. In *Passive and Active Measurement (PAM)*. Springer.