IOctopus: Outsmarting Nonuniform DMA

Igor Smolyar
Technion & VMware Research

Alex Markuze
Technion

Boris Pismenny
Technion & Mellanox

Haggai Eran
Technion & Mellanox

Gerd Zellweger
VMware Research

Austin Bolen
Dell

Liran Liss
Mellanox

Adam Morrison
Tel Aviv University

Dan Tsafrir
Technion & VMware Research

Abstract

In a multi-CPU server, memory modules are local to the CPU to which they are connected, forming a nonuniform memory access (NUMA) architecture. Because non-local accesses are slower than local accesses, the NUMA architecture might degrade application performance. Similar slowdowns occur when an I/O device issues nonuniform DMA (NUDMA) operations, as the device is connected to memory via a single CPU. NUDMA effects therefore degrade application performance similarly to NUMA effects.

We observe that the similarity is not inherent but rather a product of disregarding the intrinsic differences between I/O and CPU memory accesses. Whereas NUMA effects are inevitable, we show that NUDMA effects can and should be eliminated. We present IOctopus, a device architecture that makes NUDMA impossible by unifying multiple physical PCIe functions—one per CPU—in manner that makes them appear as one, both to the system software and externally to the server. IOctopus requires only a modest change to the device driver and firmware. We implement it on existing hardware and demonstrate that it improves throughput and latency by as much as 2.7× and 1.28×, respectively, while ridding developers from the need to combat (what appeared to be) an unavoidable type of overhead.

CCS Concepts. • **Hardware → Communication hardware, interfaces and storage**; • **Software and its engineering → Operating systems**; **Input / output**.

Keywords. NUDMA; NUMA; OS I/O; DDIO; PCIe; bifurcation

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378509

ACM Reference Format:
Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. 2020. IOctopus: Outsmarting Nonuniform DMA. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378509

1 Introduction

In modern multi-CPU servers, each CPU is physically connected to its own memory module(s), forming a *node*, and can access remote memory of other nodes via a CPU interconnect [2, 32, 82, 94]. The resulting nonuniform memory access (NUMA) architecture can severely degrade application performance, due to the latency of remote memory accesses and the limited bandwidth of the interconnect [49].

NUMA effects are inevitable, because there are legitimate, canonical application behaviors that mandate CPU access to the memory of remote nodes—e.g., an application that requires more memory than is available on the local node. Therefore, despite extensive NUMA support in production system and many research efforts [11, 15, 18, 21, 29, 43, 48, 49, 54, 85, 86], a "silver bullet" solution to the problem seems unrealistic.

The NUMA topology is usually perceived as consisting of CPUs and memory modules only, but it actually includes I/O devices as well. CPUs are equipped with I/O controllers that mediate direct memory access (DMA) by the device to system memory. Consequently, device DMA to the memory of its node is faster and enjoys higher throughput than accesses to remote node memory. We refer to such DMA as nonuniform DMA (NUDMA).

Similarly to NUMA, NUDMA can degrade performance of I/O-intensive applications, and the many techniques proposed for addressing the problem [11, 13, 28, 31, 35, 74, 81, 91, 92] only alleviate its symptoms instead of solving it.

This paper presents *IOctopus*, a device architecture that makes NUDMA impossible once and for all. The observation underlying IOctopus is that the similarity between NUMA and NUDMA is not inherent. It is a product of disregarding the intrinsic differences between device and CPU memory accesses. I/O devices are *external* to the NUMA topology, gaining access to it through the PCIe fabric. It is therefore possible to *eliminate* NUDMA by connecting the device to every CPU, which allows it to steer each DMA request to the PCIe endpoint connected to the target node.

Crucially, the IOctopus architecture is not simply about device wiring. In fact, there exist commercially available NICs whose form-factor consists of two PCIe cards that can be connected to different CPUs [63]. There also exist "multi-host" NICs [16, 38, 62]—aimed at serving multiple servers in a rack [76]—that could be engineered to connect to multiple CPUs within one server.

However, these commercial NIC architectures still suffer from NUDMA effects, because they tacitly assume that a PCIe endpoint must correspond to a physical MAC address. MAC addresses are externally visible, which prompts the OS to associate the PCIe endpoints with separate logical entities such as network interfaces. The IOctopus insight is that decomposing one physical entity—the NIC—into multiple logical entities is the root cause of NUDMA. This decomposition forces a permanent association between a socket and the PCIe endpoint corresponding to the socket's interface, which, in turns, leads to NUDMA if the process using the socket migrates to a CPU remote from that PCIe endpoint.

Accordingly, IOctopus introduces a conceptually new device architecture, in which all of a device's PCIe endpoints are abstracted into a single entity, both physically and logically. The IOctopus model crystallizes that the PCIe endpoints are not independent entities. They are extensions of one entity—the limbs of an octopus.

We describe the design and implementation of octoNIC, an IOctopus-based 100 Gb/s NIC device prototype, and of its device driver. We show that the IOctopus design enables leveraging standard Linux networking APIs to completely eliminate NUDMA. We also report on initial work to apply IOctopus principles to NVMe storage media.

Our evaluation on standard networking benchmarks shows that, compared to a Mellanox 100 Gb/s NIC which suffers from NUDMA, the octoNIC prototype improves throughput by up to 2.7× and lowers network latencies by 1.28×.

2 Background and Motivation

Modern servers are often multisocket systems housing several multicore CPUs. Each CPU is physically connected to its own "local" memory modules, forming a node. CPU cores access "remote" memory of other nodes in a cache coherent manner via the CPU interconnect. (For x86, this interconnect is HyperTransport (HT) [2, 32] for AMD processors, or QuickPath Interconnect (QPI) [82, 94] and, more recently, UltraPath Interconnect (UPI) [5, 40] for Intel processors.) Remote accesses into a module M are satisfied by the memory controller of M's CPU. Node topology is such that some nodes might be connected to others indirectly via intermediate nodes, in which case remote accesses traverse through multiple memory controllers.

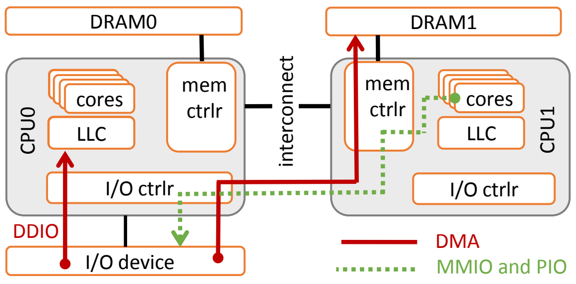

Figure 1. *I/O interactions might suffer from nonuniformity. There are four types of such interactions: DMAs and interrupts (initiated by I/O devices), and MMIO and PIO operations (initiated by CPUs).*

2.1 NUMA

The ability to access both local and remote modules creates a non-uniform memory access (NUMA) architecture that poses a serious challenge to operating system kernel designers. The challenge stems from the slower remote read/write operations as well as the limited bandwidth and asymmetric nature of the interconnect [49]. Together, these factors can severely degrade the performance of applications.

Addressing the NUMA challenge is nontrivial. It involves accounting for often conflicting considerations and goals, such as: (1) bringing applications closer to their memory and (2) co-locating them at the same node if they communicate via shared memory, while (3) avoiding overcrowding individual CPUs and preventing harmful competition over their resources (notably their cache and memory controller capacities); (4) deciding whether it is preferable to migrate applications closer to their memory pages or the other way around; (5) weighing the potential benefit of migrating application between nodes against the overhead of continuously monitoring their memory access patterns at runtime, which allows for (6) approximating an optimal node-to-application assignment at any given time in the face of changing workload conditions.

Due to the challenging nature and potential negative impact of NUMA, this issue serves as the focus of many research and development efforts [11, 15, 18, 21, 29, 43, 48, 49, 54, 85, 86]. Production operating system kernels and hypervisors—such as Linux/KVM, FreeBSD, and VMware ESXi—provide basic NUMA support: by satisfying application memory allocations from within the memory modules of the node that runs them [27, 31, 68, 88, 93]; by exposing the NUMA topology to applications [17]; by allowing applications to decide their node affinity [44]; and by automatically migrating virtual memory pages residing on remote nodes to the local node of the corresponding applications [20, 52, 80, 89].

2.2 The Problem of NUDMA – Nonuniform DMA

We usually perceive the NUMA topology as consisting of CPUs and memory modules only. However, the topology contains a third type of hardware—I/O devices—as illustrated in Figure 1. In addition to memory controllers, CPUs have I/O controllers, which mediate all memory interactions involving I/O devices. As each device is connected to a single I/O controller, I/O interactions are nonuniform as well. Namely, local interactions between the device and its node (CPU0 and DRAM0 in Figure 1) are speedier and enjoy a higher throughput as compared to remote interactions of the device (with CPU1 and DRAM1), because the latter must traverse through the CPU interconnect and therefore suffer from the same NUMA limitations.

Most of the traffic that flows through I/O controllers is typically associated with direct memory accesses (DMA) activity, which takes place when devices read from or write to memory while fulfilling I/O requests; we denote this activity as nonuniform DMA (NUDMA). There are other forms of nonuniform I/O: CPU cores communicate with I/O devices via memory-mapped I/O (MMIO) and port I/O (PIO), and devices communicate with cores via interrupts. These types of interactions are also depicted in Figure 1. However, for brevity, and since interrupts, MMIO, and PIO operations tend to be fewer as compared to DMA operations, we overload the term NUDMA to collectively refer to all types of nonuniform I/O activity.

In Intel systems, whenever possible, Data Direct I/O (DDIO) technology satisfies local DMAs using the last level cache (LLC), keeping the DRAM uninvolved [37] (bottom/left arrow in Figure 1). But DDIO technology only works locally; it does not work for remote DMA, thereby further exacerbating the problem of nonuniformity. The negative implications of the inability to leverage DDIO technology are more than just longer latency. With the ever increasing bandwidth of I/O devices, studies show that DRAM bandwidth is already becoming a bottleneck resource [3, 55]. This problem further increases the motivation to utilize DDIO, as serving DMA operations using the caches may substantially reduce the load that the DRAM modules experience [45].

We note that NUDMA activity frequently translates to "traditional" NUMA overheads. For example, if a device DMA-writes to some memory location that is currently cached by a CPU remotely to the device, then the corresponding cache line L is invalidated as a consequence, and the CPU has to fetch L from DRAM when subsequently accessing it.

No good solutions to the NUDMA problem exist, and so the relevant state-of-the-art is limited, consisting of recommending to users to manually pin I/O-intensive applications to the node that is connected to the corresponding device [13, 28, 31, 35, 81, 92], automatically doing such pinning [14, 30, 74, 77, 78, 87], and migrating some of the threads away from the said local node if it becomes overloaded [11]. Significant effort was invested in making OS schedulers NUDMA-aware [11, 74, 81, 91], which makes an already very sophisticated and sensitive sub-system even more complex and harder to maintain. All of these techniques clearly do not solve the NUDMA problem and only try to alleviate some of its symptoms if/when possible. It *seems* there is little else that can be done.

2.3 Multiple Device Queues Do Not Solve NUDMA

Modern high-throughput I/O devices—NICs in our context—support multiple per-device queues. Using these queues, the operating system and the device work in tandem to increase parallelism and improve memory locality. IOctopus uses device queues, but they alone are ineffective against NUDMA.

A queue is a cyclic array (known as a "ring buffer" or simply a "ring") in DRAM, which the OS accesses through load/store operations, and the device accesses using DMA. The queue consists of descriptors that encapsulate I/O requests, which are issued by the OS and are processed by the device. NICs offer two types of queues: transmit (Tx) queues for sending packets from DRAM to the outside world, and receive (Rx) queues for traffic in the opposite direction. Each such queue instance may be further subdivided to two rings, such that one is associated with the requests (that the CPU asks the device to process) and the other is associated with the responses (that the device issues after processing the corresponding requests).

When the device is local to the node, the OS carefully uses Tx queues to increase memory locality. Here, we outline how the Linux kernel accomplishes this goal with Transmit Packet Steering (XPS) [53]; other kernels use similar mechanisms [26, 67]. The Linux network stack maps each core C to a different Tx queue Q, such that Q's memory is allocated from C's node. Additionally, memory allocations of packets transmitted via Q are likewise fulfilled using the same node. Cores can then transmit simultaneously through their individual queues in an uncoordinated, NU(D)MA-friendly manner while avoiding synchronization overheads. When a thread T that executes on C issues a system call to open a socket file descriptor S, the network stack associates Q with S, saving Q's identifier in the socket data structure. After that, whenever T transmits through S, the network stack checks that T still runs on C. If it does not, the network stack updates S to point to the queue of T's new core. (The actual modification happens after Q is drained from any outstanding packets that originated from S, to avoid out-of-order transmissions.)

Assuming the device is local to the node, receiving packets with good memory locality is also possible, although it is somewhat more challenging than transmission and requires additional device support. Linux associates separate Rx queues with cores similarly to Tx queues, such that the associated ring buffers and packet buffers are allocated locally. The difference is that, when receiving, it is not the OS that steers the incoming packets to queues, but rather the NIC. Therefore, modern NICs support Accelerated Receive Flow Steering [53] (ARFS) by (1) providing the OS with an

API that allows it to associate networking flows[1] with Rx queues, and by (2) steering incoming packets accordingly. When the OS migrates T away from C, the OS updates the NIC regarding T's new queue using the ARFS API. Once again, the actual update is delayed until the original queue is drained from packets of S, to avoid out-of-order receives.

Together, XPS and ARFS improve memory locality, and they eliminate all NU(D)MA effects if the device is local to N—the node that executes T. However, both techniques are ineffective against remote devices. For example, assume that the NIC is remote to N, and that L is a line that is cached by the CPU of N. If L holds content of an Rx completion descriptor or packet buffer that will soon be DMA-written by the NIC on packet arrival, then L will have to be invalidated before the NIC is able to DMA-write it, as DDIO is not operational when the device is remote. When L is next read by T, its new content will have to be fetched from DRAM.

2.4 Remote DDIO Will Not Solve NUDMA

Even if hardware evolves and extends DDIO support to apply to remote devices, NU(D)MA effects nevertheless persist. Even if the NIC could write to a remote LLC, its accesses would suffer from increased latency on the critical data path, while contending over the bandwidth of the CPU interconnect (Figure 1). A less drastic remote DDIO design would allocate the line written by the NIC in the local LLC even if the target address belongs to another node. However, the remote CPU would still have to read the data from the NIC's node, resulting in cache lines ping-pongs between nodes and again increasing the critical path latency.

We empirically validate that the latter remote DDIO design does not alleviate NU(D)MA overheads in a significant way as follows. Remote DDIO already partially works for DMA-writes in cases where a response ring (containing I/O request completion notifications) is allocated locally to the device and remotely to the CPU. Let us denote the latter ring as R. After receiving a packet, the NIC DMA-writes to R the corresponding completion notification. In this case, the physical destination of the DMA is the LLC of the CPU that is local to the NIC, because device-to-memory write activity allocates cache lines in the LLC when the target memory is local to the device [37]—as is the case for R. In the pktgen benchmark experiment (described in detail in §5), which is dominated by memory accesses to rings, we find that allocating R remotely to pktgen and locally to the NIC yields only a marginal performance improvement of up to 2%.

2.5 Multiple Devices Do Not Solve NUDMA

NUDMA effects can be potentially alleviated by installing multiple identical I/O devices, one for each CPU, thus allowing all cores to enjoy their own local device [83, 87]. Let us assume that the system's owner is willing to tolerate the potentially wasted bandwidth and added hardware price associated with purchasing a different NIC for each CPU node in each server along with enough network switches with enough ports to connect all these NICs. This costlier setup can help to curb NU(D)MA effects, but only if the workload is inherently static enough to ensure that all threads remain in their original nodes throughout their lifetime. (And of course only if these threads are limited to exclusively using local devices.)

In contrast, dynamic workloads that require load balancing between CPUs will experience NU(D)MA overheads, because, technically, once a socket S is established, there is no generally applicable way to make the bytes that it streams flow through a different physical device. Therefore, using the above notation, if a thread T migrates from one CPU to another, its socket file descriptor S will still be served by the device at the original node, thereby incurring NU(D)MA overheads.

With Ethernet, for example, the inability to change the association between S and its original NIC stems from the fact that an IP address is associated with exactly one MAC. While it is possible to transition this IP address from one NIC (and MAC) to another, doing so would mean that all the other threads that use this IP address would either lose connectivity or have to change their association as well, potentially causing new NUDMA effects.

When a server is connected to a switch through multiple NICs, it may instruct the switch to treat these NICs as one channel (called "bonding" [51] or "teaming" [50]), if the switch supports EtherChannel [19] or 802.3ad IEEE link aggregation [33]. This approach does not eliminate NUDMA activity as well, because there is no way for the server to ask the switch to steer flows of some thread T to a specific NIC, and then to another NIC, based on the CPU where T is currently executing. Switches do not support, for example, a mechanism similar to the aforementioned ARFS (§2.3). (While SDN switches have similar capabilities [75], they typically do not provide individual hosts with the ability to steer between aggregated links.) It is possible to design switches that support ARFS-like functionality, but we will have to replace all the existing infrastructure to enjoy it.

2.6 Technology Trends: One Device May Be Enough

In addition to the fact that multiple I/O devices do not solve the NUDMA problem (§2.5), in the case of networking, we contend that technology trends suggest that the I/O capacity of a single device should typically be enough to satisfy the needs of all the CPUs in the server. Figure 2 depicts these trends by showing the past and predicted progression of the network bandwidth that a single NIC supports vs. the network bandwidth that a single CPU may consume. The two NIC lines shown correspond to the full-duplex throughput of a single- and a dual-port NIC, respectively.

[1]An IP flow is uniquely identified by its 5-tuple: source IP, source port, destination IP, destination port, and protocol ID.

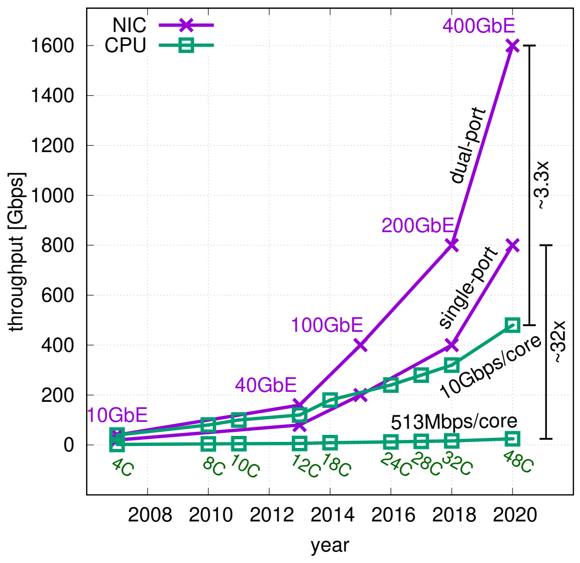

Figure 2. *The bandwidth of the NIC exceeds what a single CPU could use. Top labels show Ethernet generations. Bottom labels show the number of cores per CPU. (Data taken from various sources corresponding to Intel/AMD CPUs [8, 39, 70] and Mellanox and Intel NICs [6, 34, 39, 58, 59, 65].)*

The bottom-most CPU line assumes that *every* core in the CPU consumes 513 Mbps. This figure reflects an upper bound on the per-core TCP throughput that was reported for Amazon EC2 high-spec instances (4xlarge and up: 8xlarge, 12xlarge, etc.) with 8 and more cores when all cores concurrently engage in networking [7, 90]. An earlier report from 2014 shows that 8-core instances of four cloud providers (Amazon, Google, Rackspace, and Softlayer) consume at most 380 Mbps per core [71].

The upper CPU line assumes an unusually high per-core rate of 10 Gb/s TCP, which is about 50% of a core's cycles in a bare-metal setup when running the canonical netperf benchmark [42]; let us assume the other 50% is needed for computation, as netperf does not do anything useful. The number of cores shown reflects the highest per-CPU core-count available from Intel and AMD for the corresponding year. We multiply the assumed maximal per-core bandwidth with the highest core count and display the product as the maximal throughput that one CPU may produce/consume (optimistically assuming that OSes can provide linear scaling when all CPU cores simultaneously do I/O). The figure indicates that one NIC is capable of satisfying the needs of multiple CPUs, even in such a demanding scenario. Others have reached a similar conclusion [46].

3 Design

In this section, we describe the design of IOctopus, which consists of hardware and software components that together eliminate all NUDMA effects. We begin by observing the inherent differences between NUMA and NUDMA that make IOctopus possible (§3.1). We next describe the hardware/firmware support that IOctopus necessitates in wiring (§3.2) and networking (§3.3). We then describe the software, operating system aspect of IOctopus, which introduces a new type of I/O device that is local to all CPUs (§3.4). In the subsequent section, we describe how we implemented all of these components (§4).

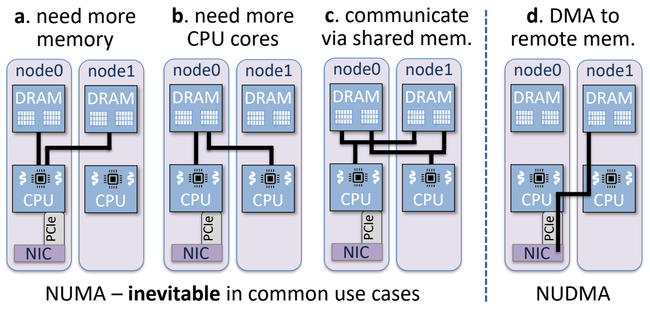

Figure 3. *NUMA effects are inevitable for some canonical algorithm classes, which dictate that CPU cores in one NUMA node must access the memory of another (a–c). NUDMA effects are likewise presently unavoidable (d), but not due to true node sharing.*

3.1 True and False Node Sharing

NUMA effects cannot be eliminated. This is true despite the extensive NUMA support provided by production system and all of the associated research efforts (§2.1). NUMA effects are inevitable because there are legitimate, canonical algorithm classes that *mandate* CPU cores to access the memory of remote NUMA nodes. Let us use the term "true node sharing" to denote such situations, where, algorithmically, it is impossible to avoid NUMA effects, as CPU cores are meant to access memory of remote nodes, by design.

True node sharing occurs, for example: when a single thread running on a single core solves a problem that requires more memory than is available on the local node (Figure 3a); or when the problem being solved requires relatively little memory and is housed by a single node, but additional cores—more than are available on the local CPU—can accelerate the solution considerably (Figure 3b); or when the problem is solved with a classically-structured parallel job, where each thread is assigned a core to execute a series of compute steps, separated by communication steps whereby all threads read from the memory of their peers in order to carry out the subsequent compute step (Figure 3c) [24].

The initial insight underlying the design of IOctopus is that NUDMA activity is *not* the result of true node sharing. This is so because, by definition (§2.2), NUDMA activity does not involve cores accessing *any* memory modules, neither local nor remote (Figure 3d). Instead, it is the device that accesses the memory.

More specifically, as its name suggests, the NUMA architecture intentionally makes memory accesses of CPU cores nonuniform. It employs a distributed memory controller that unifies the memory modules spread across all nodes into a single coherent address space. Memory access latencies experienced by cores are then determined by the internal topology of the distributed memory space. In contrast, I/O devices are entirely external to this topology, gaining access to it via a PCIe fabric. Thus, the specific connection point between the PCIe fabric and the NUMA memory space determine memory access latencies that devices experience. Namely, assuming it is possible to connect the NIC in Figure 3d via PCIe to both CPUs, then, in principle, it may be possible to eliminate NUDMA effects.

In light of the above, we denote NUDMA effects as happening due to "false node sharing." When restating our aforementioned insight using this terminology, we can say that the inherent difference between NUMA and NUDMA effects is that the former are the result of true node sharing, whereas the latter are the result of false node sharing. This articulation is helpful, because it highlights why, in principle, NUDMA effects may be avoidable.

3.2 Wiring Hardware Support

Connecting I/O devices via PCIe to only a single CPU is an old, standard practice, which is so pervasive that it appears as carved in stone. Consequently, one might easily mistakenly believe that there are sound technical reasons that prevent us from connecting a device to multiple CPUs. However, this is not the case. Such connectivity already exists in production, and we contend that its availability will become more and more prevalent in the future, as discussed below.

Before we conduct the discussion, however, it is essential to note that, by itself, connecting an I/O device to multiple CPUs does not eliminate NUDMA. Rather, such connectivity is equivalent to using multiple devices, as discussed in §2.5. Namely, for technical reasons explained later on, connecting a device to multiple CPUs translates to adding more PCIe endpoints to the PCIe fabric, such that each endpoint is local to its own CPU but remote to all the rest.

PCIe Bifurcation and Extenders Currently, probably the most straightforward approach that can be used to connect one I/O device to multiple CPUs is through PCIe bifurcation [41], which enables splitting a single PCIe link into several.[2] The vendor of the I/O device can implement different types of bifurcation, e.g., a 32-lanes PCIe link width could be split into 2 or 4 PCIe endpoints with a link width of 16- and 8-lanes, respectively. The additional endpoints that bifurcation creates could be connected to other CPUs.

In some bifurcation cases—e.g., splitting 16 lanes into two 8-lane endpoints connected to different CPUs—the resulting available bandwidth between the device and a single CPU may not be sufficient for certain workloads. To alleviate this problem, vendors can support extending, say, a 16-lane PCIe device with an additional 16-lane PCIe endpoint (provided that internally the device has 32-lanes; additional resources are required [66].

Attesting the architectural viability of PCIe bifurcation to connect a single I/O device to multiple targets is the fact that Broadcom, Intel, and Mellanox already produce "multi-host" NICs [16, 38, 62]. The goal of a multi-host NIC is to simultaneously serve 2–4 physical servers in a consolidated manner [76]. Given that such connectivity works for multiple servers in a rack, it stands to reason that it should also work for multiple CPUs within one server.

IOctopus is a joint project developed by several organizations, including Mellanox, which is a networking vendor. Mellanox already manufactures a NIC that employs bifurcation to be able to connect to multiple CPUs using standard PCIe extenders [64].

Motherboard Hard-Wiring The drawback of connecting one I/O device to multiple CPUs with PCIe extenders is the additional cabling, which takes up space within the server and complicates its packaging. An alternative that does not suffer from this drawback is for motherboard vendors to include this cabling directly, built into the motherboard. This approach, however, reduces flexibility, because hard-wired motherboard PCIe lanes cannot be rewired like PCIe extenders. Therefore, an improved alternative is to support motherboard riser cards that eliminate the need to use extender cabling but still provide some flexibility. Risers are expansion cards that host I/O devices and connect to the motherboard, such that different type of riser cards may provide different PCIe wiring. For example, whereas one riser card may employ bifurcation to split the available lanes and connect to all CPUs, another may connect all the lanes to a single CPU, in the old-fashioned way.

In addition to Mellanox, the organizations that develop IOctopus also include Dell, which is a server vendor. The next generation of Dell servers (available in 2020) provides riser cards that allow clients to connect a single I/O device to multiple CPUs [22]. The latest generation of Dell servers provides riser cards that allow clients to connect a single I/O device to multiple CPUs.

Programmable PCIe Switching The main drawback of extenders and motherboard hard-wiring is that they are static: lanes are partitioned in a certain way, and any change requires manual reconfiguration (such as switching riser cards). A more flexible solution is to use an onboard programmable PCIe switch that may connect all I/O devices to all CPUs. The benefits of this approach are: that it is dynamic and therefore eliminates the need for manual reconfiguration; that it requires no additional external physical hardware (PCIe extenders and riser cards); and that it additionally supports peer-to-peer DMA communications between different PCIe devices, which may be important for I/O intensive workloads. The drawbacks of this approach compared to

[2]The citation [41] refers to a bifurcating one CPU PCIe link into multiple links to the same CPU; we are presenting bifurcating to multiple CPUs here.

bifurcation or hard-wiring are that it is pricier, adds latency to individual operations, consumes more power and that it requires more lanes to support all switch configurations.

3.3 Networking Hardware Support

Simply connecting a NIC to multiple CPUs does not eliminate the NUDMA problem, because existing devices are designed tacitly assuming that a PCIe endpoint (also referred to as physical function or PF) must correspond to a physical MAC address. Consequently, the OS associates NIC PFs with separate *logical* entities such as network interfaces and IP addresses. The IOctopus insight is that this decomposition of one physical entity—the NIC—into multiple logical entities is the *root cause* of NUDMA. Forcing a socket's unique association with an interface to imply a unique association with a PF leads to NUDMA whenever the socket's owner is scheduled on a CPU remote from the PF.

To address this design problem, IOctopus introduces a conceptually new multi-PF device model. In IOctopus, all PFs are abstracted into a single entity, both physically and logically. An IOctopus NIC (octoNIC) has a single interface with the external world—a single physical port and MAC address. Similarly, the OS associates the octoNIC with a single interface and IP address (§3.4). The IOctopus model crystallizes that the PFs are extensions of one entity—the limbs of an octopus—and not independent entities.

With the IOctopus model, I/O traffic is no longer associated *a priori* with a PF. The OS needs to transmit data through the octoNIC PF local to the transmitting CPU (§3.4), and the octoNIC must steer incoming traffic to the PF local to the CPU on which the receiving process runs on.

To facilitate this steering, we propose a new NIC feature, *IOctoRFS*. With IOctoRFS, the octoNIC provides the OS with an API to associate a flow 5-tuple with the PF to which the flow's traffic should be steered. The OS updates IOctoRFS mappings exactly as it updates ARFS mappings (§2.3) today. Figure 4 depicts the overall design. IOctoRFS can be implemented with modest firmware changes by combining existing multi-PF NIC hardware features (§4).

To eliminate NUDMA, IOctopus must handle the corner case in which a single packet spans pages from different NUMA nodes. This scenario can only occur when the transmitted buffers are not allocated by the NIC's driver. (E.g., when using `sendfile()` to transmit data directly from the page cache, where buffers might span NUMA nodes.) In contrast, received packets are DMAed into buffers allocated by the NIC driver. The driver can guarantee that these buffers do not span NUMA nodes by allocating them appropriately.

IOctoRFS does not suffice to address packets whose data spans NUMA nodes, since no single PF can access the packet over PCIe without incurring NUDMA. We propose an *IOctoSG* (scatter-gather) feature that allows the driver to provide a hint in ring descriptors specifying which PF to use when accessing each fragment.

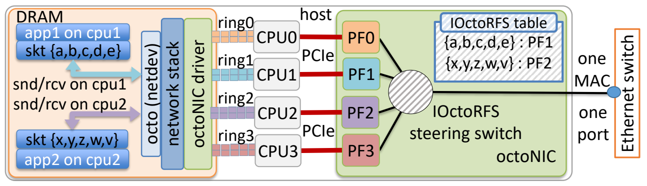

Figure 4. *OctoNIC design.*

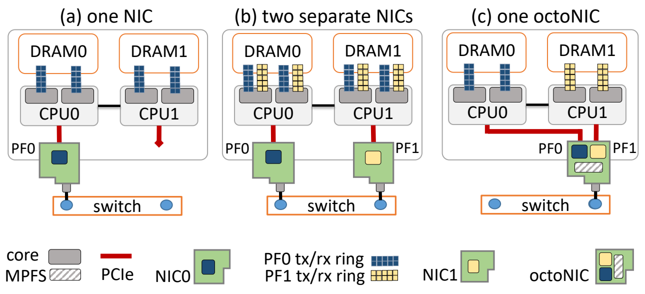

Figure 5. *Compare existing designs (a–b) with IOctopus (c).*

Memory ordering issues Whenever packet data is read or written by a single octoNIC PF, IOctopus does not require different OS handling (compared to a standard NIC) to maintain correct memory ordering of DMAed data—i.e., to guarantee that data DMAed to/from memory be correctly observed by the CPU/device. Consequently, the only possible problematic case could be with IOctoSG used to transmit data that spans NUMA nodes. However, the CPU memory ordering primitives used by the OS to guarantee that a device correctly observe transmitted data are agnostic to the device's PF. These primitives provide a more coarse-grained guarantee, namely, that the transmitted data is observable to *any* external devices.[3] Therefore, IOctopus does not require any special handling on the OS even for data transmitted using IOctoSG.

3.4 Software

Our design is grounded in the following principles:

Transparency OctoNIC's physical structure should be transparent to applications and the networking stack. Just as the octoNIC appears to the outside world as a single physical entity, so should it appear to the software stack—a NIC with one networking interface, which applications can use and experience no NUDMA activity.

The goal of transparency rules out alternative NUDMA solutions, such as using multiple NICs (Figure 5b) or Mellanox's existing solution [64]. These approaches represent each PF of the device as a separate logical entity, and thus, NUDMA cannot be avoided transparently. For instance, both these approaches require the OS to maintain separate Rx/Tx rings per-PF (Figure 5b), whereas IOctopus does not (Figure 5c).

[3] For example, Linux on ARM uses a `dmb(oshst)` instruction, which makes the data visible in the "outer shareable domain," which captures "other observes" such as a "GPU or DMA device." [9].

Locality IOctopus software should work alongside the octoNIC to guarantee that data flowing between the NIC and an application is dynamically steered to the PF that is local to the application's CPU. For instance, achieving locality would allow the OS scheduler to disregard NUDMA considerations in its scheduling decisions (§2.2).

In accordance with the above, IOctopus software is a device driver that presents the multi-PF octoNIC as a single logical device to the system. Moreover, the IOctopus driver piggybacks on *existing* ARFS and XPS kernel functionality to (1) make sure data is transmitted through the PF local to the sending CPU and (2) update IOctoRFS so that arriving data is steered to the PF local to the CPU of the receiving process.

4 Implementation

4.1 OctoNIC Prototype

We have implemented an octoNIC prototype by modifying the firmware of a Mellanox 100 Gb/s NIC with a bifurcated PCIe interface [63] The NIC's 16 PCIe lanes are bifurcated into two 8-lane buses, and we connect them to each CPU of a dual node system (Figure 5c).

The modified firmware implements IOctoRFS by composing two existing features of multi-PF NICs. The first is the mechanism used for ARFS (§2.3), which maps flows to Rx queues. The second is an integrated multi-PF Ethernet switch (MPFS), which steers incoming traffic to PFs based on their target MAC address [16, 36, 60].[4] In principle, IOctoRFS simply requires the NIC to store another mapping, from Rx queue id to the PF local to the queue's CPU. The IOctoRFS switch then steers an arriving packet to the right PF by mapping it to a queue id and then to a PF. The storage size required for the queue-to-PF table is negligible.

Our prototype takes a different approach, however, to avoid changing the NIC's processing logic—specifically, that the NIC maintains ARFS tables per-PF, not globally. To accommodate this property, we modify the MPFS to map packets to a PF based on their flow 5-tuple instead of the MAC address. With the PF determined, the NIC looks up the target Rx queue in the PF's ARFS table as usual.

Our prototype does not implement IOctoSG.

4.2 OctoNIC Driver

We considered two alternatives to present the octoNIC to the OS network stack as a single networking device (netdevice): (1) modify the vendor driver to create a single netdevice that hides the underlying octoNIC PCIe endpoints; or (2) keep the existing driver, which creates a netdevice for each PCIe endpoint, but use an additional software layer that aggregates IOctopus netdevices into a single virtual interface. While the first approach is transparent for the OS, it requires considerable changes to a complicated vendor device driver. Therefore, we chose to add a special IOctopus mode to Linux's existing *team driver* [23, 50], which allows teaming multiple network interfaces into a single logical interface.

Receive The octoNIC driver needs to maintain the NIC's MPFS tables so that arriving packets are handed to the CPU on which the receiving process runs. To handle process CPU migrations, we re-use the existing ARFS callback in the Linux kernel, which informs a networking driver when a process migrates between CPUs. (The OS takes care to invoke the callback only after the network stack has dequeued incoming packets on the old CPU's Rx queue, to avoid out-of-order packet delivery.)

The octoNIC driver maintains a mapping of flow 5-tuples to MPFS metadata rules. When the driver receives an ARFS callback, it determines if it should add a new entry to the MPFS table or an existing entry needs to be updated to steer the flow to a different NUMA node. The MPFS table is then updated asynchronously by a separate kernel worker thread. Similarly to Linux ARFS, we use a separate kernel thread to periodically search for expired rules and delete them from the driver tables and the device.

Transmit Generally, when handed an outgoing packet by the networking stack, the octoNIC driver transmits it through the octoNIC PCIe endpoint that is local to the current CPU, as the data is hot in its LLC. The only complication is avoiding out-of-order packet transmission after process CPU migration because there is no OS callback for when the previous Tx queue empties. However, Linux's XPS (§2.3) packet metadata has this information (a per-packet (*ooo_okay*) flag), so our driver obtains it from there.

Implementation effort IOctopus does not change any kernel APIs. The octoNIC team device driver consists of 463 lines of code (LOC), we added 6 LOC to the *libteam* library that configures team devices. We further added/changed 23 LOC in the kernel and 50 LOC in the Mellanox NIC driver.

5 Evaluation

We experimentally evaluate IOctopus to answer the following questions: By how much does eliminating NUDMA improve the throughput and latency of I/O traffic (§5.1)? What is the impact of NUDMA elimination on unrelated processes that share the CPU with I/O workloads (§5.2)? How effective is IOctopus in handling process migration between CPUs (§5.3)? And finally, what is the result of applying IOctopus principles and design to storage I/O (§5.4)?

Experimental setup We use two Dell PowerEdge R730 machines, a client and a server. Each machine has two 14-core 2.0 GHz Intel Xeon E5-2660 v4 (Broadwell) CPUs, connected via two 9.6 GT/s QPI links. Each machine has 128 GB of memory (4x16 GB DIMMs per socket). Both machines run Ubuntu

[4]The MPFS exists to support configurable MAC addresses and SR-IOV.

16.04 with Linux kernel 4.14.110, and have hyperthreading and Intel Turbo Boost (dynamic clock rate control) disabled.

The client is equipped with a 100 Gb/s Mellanox ConnectX-4 NIC [61]. The server has a Mellanox 100 Gb/s NIC with a bifurcated PCIe interface [63]. The client is connected back-to-back to one of the server NIC's ports. NIC drivers are configured to use a descriptor ring per core with even distribution of interrupts between cores. Linux adaptive interrupt coalescing is enabled.

Evaluated configurations By default, the server's NIC appears to the OS as two NICs, each connected to a different CPU. By loading our IOctopus firmware, we can turn the server's NIC into an octoNIC. Our experiments compare three server configurations: (1) *local* and (2) *remote*, which use the standard firmware and where the utilized NIC is local or remote, respectively, to the socket on which the workload (including interrupt handling) runs and to which allocated memory belongs; and (3) *IOctopus*, in which the NIC acts as an octoNIC. The *remote* configuration triggers NUDMA activity. The *IOctopus* and *local* results are usually identical, in which case we report them as a single *ioct/local* configuration in the figures. The client-side of the workload uses the socket local to its NIC and so incurs no NU(D)MA effects.

5.1 Impact on I/O Traffic

5.1.1 Throughput Impact

We evaluate TCP throughput using the TCP-STREAM test of the netperf [42] benchmark. In these tests, the process repeatedly receives (or transmits) a fixed-size buffer from (or to) a TCP socket. We run the benchmark for 60 seconds and report averages of 11 runs. We perform single-core experiments, in which both process and OS networking activity (such as interrupt processing) run on a single core, and multi-core experiments.

Single-core receive (Rx) Figure 6 shows the throughput, memory bandwidth used, and CPU utilization as we vary the netperf buffer size; numbers above the *ioct/local* throughput curve report its throughput relative to the *remote* CPU. Both configurations are bottlenecked by the CPU. The *ioct/local* throughput is always higher than that of *remote*, with the relative advantage depending on the amount of data transferred per packet. When the netperf buffer size exceeds the 1500-byte MTU, and all received packets become MTU-sized, *ioct/local* outperforms *remote* by about 1.25×. This throughput gap is due to *ioct/local* benefiting from DDIO, which allows the CPU to read received data from the LLC rather than memory. The lack of DDIO in *remote* results in a memory bandwidth use of 3× the network throughput.

Single-core transmit (Tx) Figure 7 shows the results of the Tx workload. The Tx path uses the NIC's TCP Segmentation Offload (TSO) functionality, which allows the kernel to aggregate sent data into 64 KB TCP segments before handing it to the NIC. As a result, both configurations more than double their throughput compared to the Rx workload. Unlike the Rx workload, however, both configurations obtain comparable throughput.

The reason for this behavior is that in *both* configurations, the CPU writes to its working set—which is hot in the LLC—without incurring cache misses. This behavior is expected for *ioct/local* (due to DDIO), but may be surprising for *remote*. We believe that to guarantee DMA coherency, remote DMA reads are satisfied by probing the LLC and DRAM in parallel. If the target line is in the LLC, the DMA is serviced from there, without invalidating the line; otherwise, the line is read from DRAM. The fact that *remote*'s memory bandwidth consumption is *equal* to its obtained throughput supports this hypothesis: If DMA reads were satisfied by evicting the line in order to read it from memory, memory bandwidth usage would have been double the throughput. [5]

Multi-core throughput We evaluate multi-core performance by running a netperf instance on each core of the machine. Having multiple cores driving the workload shifts the bottleneck from the CPU to the network, and both configurations are able to sustain line rate. However, *ioct/local* incurs memory traffic, unlike the single-core workloads. The reason is that the combined working set of all the cores exceeds the LLC size. Due to space constraints, we omit the figures.

Single-core packet throughput Raw packet transmission rates are important for packet-based network functions, such as gateways, routers, load balancers, etc. To evaluate packet throughput, we use pktgen [73], an in-kernel tool for generating packets at high speeds. Figure 8 shows the resulting throughput (Gb/s) and memory bandwidth for various packet sizes. (In all experiments, CPU utilization is 100%.) We observe a striking difference from the TCP Tx throughput experiment. Whereas with TCP Tx the throughput of the configurations is comparable, here *ioct/local* consistently obtains 1.3× the throughput of *remote*.

This throughput difference is due to the orders-of-magnitude higher rate of packet transmissions (i.e., packets handed to the NIC), which causes per-packet NUDMA effects to become significant. In the TCP Tx experiment, CPU work dominates by copying the sent data from the process to the kernel, and—due to TSO—the CPU hands the NIC 64 KB segments. Thus, the highest TCP Tx packet rate is 91 K (= 47 Gb/s/64 KB segments) packet per seconds (PPS). In a TCP Tx experiment without TSO (not shown), the packet rate increases to $\approx$ 500 KPPS (but throughput is much lower). In contrast, pktgen repeatedly transmits the same IP packet without touching any data. Consequently, *ioct/local* is able to transmit 4.1 MPPS, and *remote* transmits 3.08 MPPS. At these

[5] This observation implies that Intel's statement that "DDIO technology only works locally" (§2.2) refers only to DMA writes. Indeed, Intel's documents do not precisely define what "works" means.

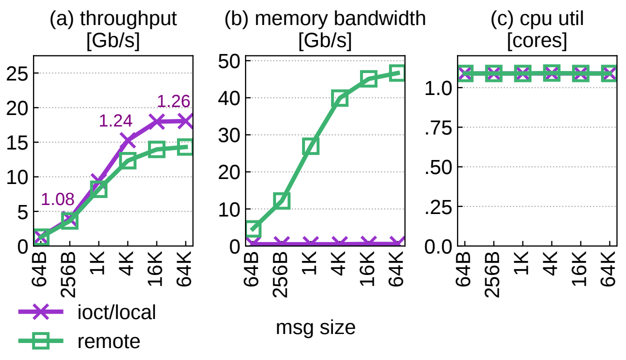

Figure 6. *Single-core TCP stream receive.*

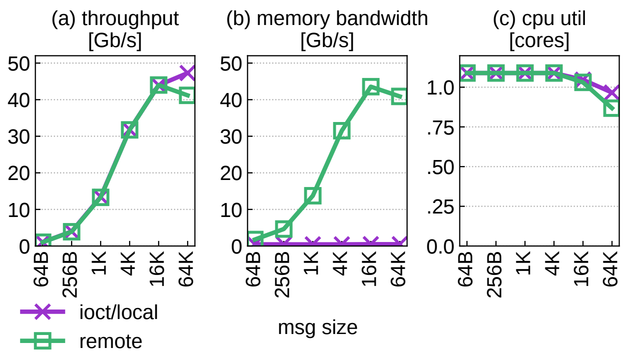

Figure 7. *Single-core TCP stream transmission (TSO enabled).*

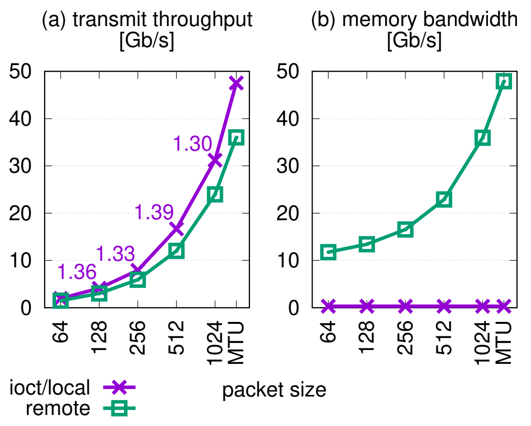

Figure 8. *(a) Network throughput and (b) memory bandwidth utilization when using the pktgen benchmark with a single core.*

rates, per-packet transmission cost determines the throughput. Due to DDIO, *ioct/local* does not experience LLC misses, whereas *remote* incurs one LLC miss per packet. This miss results from reading the *completion entry* that the NIC writes after transmitting a packet. Reading this entry from memory costs about 80 ns, which is essentially the delta between the per-packet costs of *ioct/local* and *remote*.

5.1.2 Latency Impact

To measure TCP latency, we use a single-core netperf request/response benchmark (TCP RR). This benchmark measures the latency of sending a TCP message of a certain size from the server machine to the client machine and receiving a response of the same size. We run the benchmark for six minutes and report the average round-trip time. (We verify that this average is stable across runs.) To minimize latency, we disable adaptive interrupt coalescing.

We compare configurations in which both server and client utilize the NIC local or remote, respectively, to their CPUs. These are indicated as *ll* and *rr*, respectively. Results when the server's NIC acts as an octoNIC are identical to the results obtained with a local NIC, and so are not reported separately.

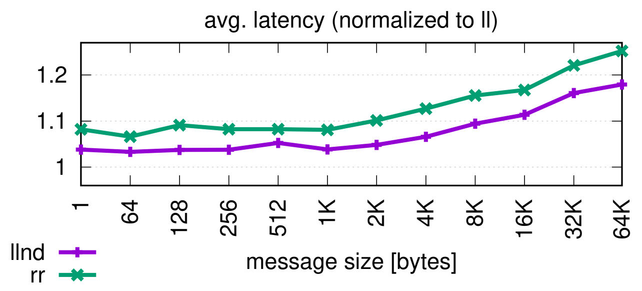

Figure 9. *Latency of netperf TCP RR with NUDMA effects* (rr) *normalized to latency without NUDMA effects* (ll)*.* rr/ll *indicates whether both server and client use local or remote NICs, respectively. An* nd *suffix indicates DDIO is disabled on both server and client.*

Figure 9 shows the latency obtained for various message sizes, normalized to the *ll* configuration. For a given message size, having NUDMA operations on the critical path adds an overhead of 10%–25% over *ll*. The 90th and 99th percentile latency (not shown) behaves similarly. To tease apart the overhead of QPI interconnect crossings, we further measure an *llnd* configuration, which is *ll* but with both sides having DDIO disabled in hardware [69, 84]. In both *llnd* and *rr*, the CPUs access DRAM to interact with the NIC and any latency difference is due to the QPI. We find that crossing the QPI imposes a latency overhead of 5%–15%. *The takeaway here is that even if DDIO worked for remote NICs, IOctopus would still eliminate substantial QPI latency overhead.*

5.1.3 Key-Value Store

To evaluate the benefit of IOctopus on a real-world application, we measure the aggregated throughput of a single memcached [25] key-value store accessed by 14 memslap [4] instances running on one client CPU. We use keys and values of 256 bytes and 512 KB, respectively, which reflect recent reports of key/value size in production workloads [1]. Here, *local* and *remote* refer to configurations in which memcached server and clients use the NIC local or remote, respectively, to their CPU. As before, when the server NIC's acts as an octoNIC, the results are identical to *local*, and are shown as *ioct/local*.

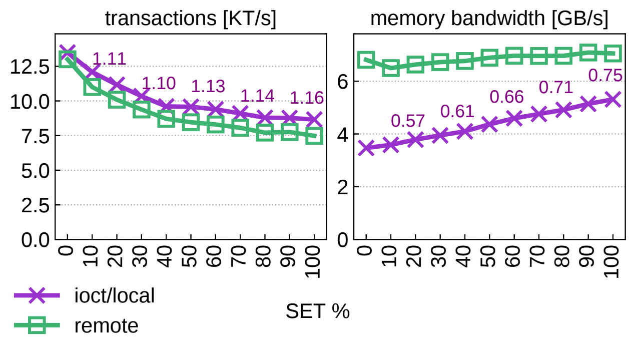

Figure 10. *Memcached throughput and server memory bandwidth, as the ratio of SET operations grows from 0% to 100%.*

Figure 10 shows the results as we vary the ratio of SET operations. The advantage of *ioct/local* over *remote* grows up to 16% with the ratio of SETs because these operations cause TCP Rx traffic that suffers from NUDMA effects, as discussed previously. The working set here is larger than in the netperf TCP Rx experiments (indicated by the fact that *ioct/local* has memory activity), and thus the NUDMA effects are less pronounced.

5.2 Co-Location

To achieve high utilization, data-center operators place multiple workloads on the same physical machine. For example, a single server may host a latency sensitive workload as well as a low priority batch computation workload. Here, we evaluate the effect NUDMA has on such configurations.

QPI congestion. We measure the effect that QPI load has on single-core TCP Rx throughput (netperf) and 64-byte UDP message latency (using sockperf [57]). To load the QPI, we occupy the other server cores with pairs of the STREAM [56] memory bandwidth benchmark. Both STREAM instances in each pair target memory remote to their CPU, one reading and the other writing. Figure 11 shows the throughput results. Both *ioct/local* and *remote* suffer as STREAM activity increases, but *ioct/local* obtains 1.82×–2.67× higher throughput than *remote*. The latency results (Figure 12) are similar, with *ioct/local* obtaining 10%–22% lower latency than *remote*. Since the latency benchmark is not data-intensive, the latency of *ioct/local* is not affected by the number of STREAMs, whereas the *remote* latency grows as the QPI becomes congested.

Macro benchmarks. We evaluate how NUDMA traffic affects a co-located programs. To serve as such a *victim*, we use a 16-thread parallel PageRank (PR) benchmark [12]), with 8 threads pinned to each CPU. We measure the effect of dedicating the remaining six cores on each CPU to instances of (1) memcached (256 KB values) or (2) netperf TCP Rx benchmarks. Figure 13 shows what effect placing the I/O workloads in *ioct/local* vs. *remote* configurations has on the running time of the victims as well as on the throughput of the I/O workloads. In both cases, PR slows down due to the co-located workloads. The PR run time is 12% higher when netperf is *remote* than when it is *ioct/local*. For memcached, the difference is 4%. However, memcached's throughput suffers more when it shares the QPI with PR, whereas netperf's throughput is comparable in both *remote* and *ioct/local* configurations.

5.3 IOctopus Steering Switch

To evaluate IOctopus' handling of thread migration, we run the TCP Rx netperf workload (64 KB buffers) and migrate the process to the other socket after approximately 4.5 seconds using the `sched_setaffinity` system call. Throughout the experiment, we sample the throughput of the NIC's two PFs every 50 msec. Figure 14 shows the results. When the NIC acts as an octoNIC, the octoNIC detects process migration and steers incoming traffic to its CPU. Consequently, traffic smoothly moves to the PF local to the process. (We observe no lost or out-of-order packets during the test.) In contrast, with the NIC's standard firmware and driver, the process keeps using the same PF after migrating, resulting in a throughput drop from *ioct/local*-level to *remote*-level.

5.4 IOctopus on NVMe

The IOctopus principles are relevant to any I/O device. Here, we consider NVMe controllers. The recent NVMe specification [72] supports multi-PF controllers that can be used to implement multi-path I/O systems. Such *dual-port* NVMe SSDs are already available on the market [47].

We customize a Dell server back-plane to allow connecting a dual-port drive to different CPUs and verify that it appears as two NVMe drives and that its internal storage is accessible from both. We leave creating an *OctoSSD*, with the entailed firmware and software support, to future work. Instead, we next evaluate the severeness of NUDMA effects on NVMe workloads, which an OctoSSD would address.

Experimental setup We use a Dell server with a standard back-plane. The server has two 24-core Intel Xeon Platinum 8160 (Skylake) CPUs, connected via two 10.4 GT/s UPI links. The machine has 96 GB of memory (6x8 GB DIMMs per socket). We use four Samsung PM1725a NVMe SSDs [79].

NVMe NUDMA impact We evaluate the sensitivity to the interconnect load exhibited by an NVMe I/O workload accessing a remote SSD. For the I/O workload, we use the fio benchmark [10] (v3.3) to generate NVMe I/O traffic. We run 8 fio threads that each perform asynchronous direct reads, thereby bypassing the page cache and interacting directly with the SSD. Each thread continuously submits 32 read requests for 128 KB blocks. The fio jobs interact with an SSD *remote* from their CPU. To load the interconnect, we run instances of the STREAM benchmark that target memory of the fio node but run on the SSD's node.

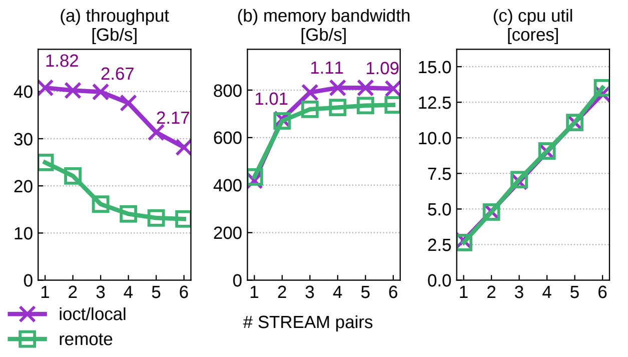

Figure 11. *TCP Rx throughput benchmark co-located with STREAM benchmarks.*

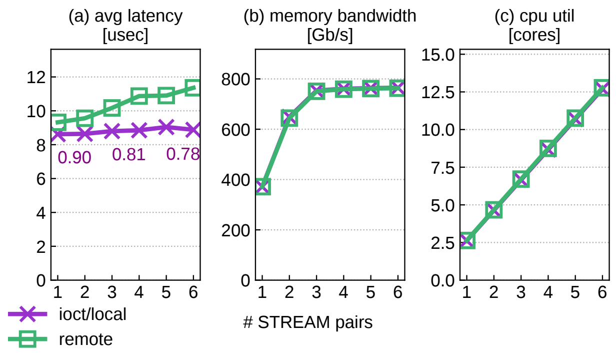

Figure 12. *Network latency benchmark (64 byte messages) co-located with STREAM benchmarks.*

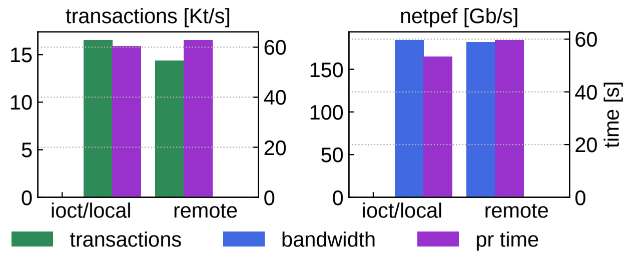

Figure 13. *The effect that co-locating a multithreaded PageRank (PR) benchmark with multiple memcached or netperf instances has on PR run time and I/O benchmark throughput.*

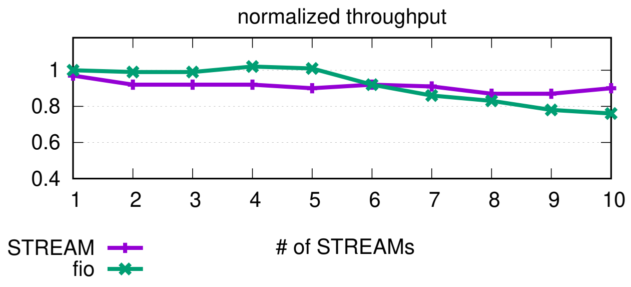

Figure 15. *Throughput of four NVMe devices while an increasing amount of STREAM instances generate interconnect traffic. In each configuration throughput is normalized to the results obtained without running the antagonist.*

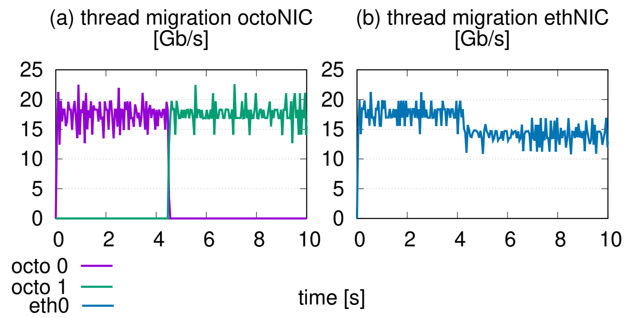

Figure 14. *Throughput on NIC physical functions when a netperf TCP Rx process migrates from CPU 0 to CPU 1.*

Figure 15 shows the throughput obtained by each benchmark, normalized to its throughput when running alone, as we vary the number of STREAM instances. The throughput of fio degrades by up to 24% after five instances of STREAM, as a result of UPI saturation (verified with performance counters). When the UPI is unloaded, fio's throughput is limited by the SSDs. We further validated that fio's throughput is not affected by UPI traffic if fio runs on the node local to the SSD (graphs omitted). The takeaway is that NUDMA also affects modern high-speed NVMe storage devices. Accessing high-speed I/O devices over the CPU interconnect is suboptimal, and can be avoided using IOctopus.

6 Conclusions

IOctopus is an idea whose time has come. It is based (1) on the observation that NUDMA overheads are inherently different than NUMA overheads in that the former are avoidable whereas the latter are inevitable; and (2) on the idea that multiple physical PCIe functions may serve as internal logical entities within a single device, in a manner that makes them transparent both to the external world and to system software layers higher in the I/O stack than the IOctopus device driver. By design, IOctopus eliminates all NUDMA effects and makes all node-device interactions local.

Acknowledgments

We thank Nadav Amit, Ilya Lesokhin, Natan Oppenheimer, Tzahi Oved, Guolin Yang, Craige Jiang, Ben Pfaff, Stephen Rousset, John Beckett, Gil Shomron, and the anonymous reviewers for their valuable feedback. We thank Mellanox and Dell for their assistance and provided hardware. This research was funded in part by the Israel Science Foundation (grant 2005/17).

References

[1] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. Fast key-value stores: An idea whose time has come and gone (HotOS'19 talk slides). https://ai.google/research/pubs/pub48030. (Accessed: Aug 2019).

[2] Ardsher Ahmed, Pat Conway, Bill Hughes, and Fred Weber. AMD Opteron shared memory MP systems. In *Hot Chips*, 2002. http://www.hotchips.org/wp-content/uploads/hc_archives/hc14/3_Tue/28_AMD_Hammer_MP_HC_v8.pdf (Accessed: Jan 2017).

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015. https://doi.org/10.1145/2872887.2750386.

[4] Brian Aker. Memslap - load testing and benchmarking a server. http://docs.libmemcached.org/bin/memslap.html. Accessed: August 2016.

[5] Paul Alcorn. Intel Xeon Platinum 8176 scalable processor review – the mesh topology & UPI. Tom's Hardware, https://www.tomshardware.com/reviews/intel-xeon-platinum-8176-scalable-cpu,5120-4.html, Jul 2017. (Accessed: Jan 2019).

[6] Ethernet Alliance. The 2018 Ethernet Roadmap. https://ethernetalliance.org/the-2018-ethernet-roadmap/, 2018. Accessed: January 2019.

[7] Amazon. Physical cores by Amazon EC2 and RDS DB instance type. https://aws.amazon.com/ec2/physicalcores/, 2019. Accessed: January 2019.

[8] AMD. AMD EPYC 7000 Series: Product Specifications. https://www.amd.com/en/products/epyc-7000-series, 2019. Accessed: January 2020.

[9] ARM. ARM Cortex-A Series Programmer's Guide for ARMv8-A: Cacheable and shareable memory attributes. https://developer.arm.com/docs/den0024/latest/memory-ordering/memory-attributes/cacheable-and-shareable-memory-attributes. (Accessed: Jan 2020).

[10] Jens Axboe. fio - Flexible IO Tester. http://git.kernel.dk/cgit/fio/, 2019. Accessed: August, 2019.

[11] Amitabha Banerjee, Rishi Mehta, and Zach Shen. NUMA aware I/O in virtualized systems. In *High-Performance Interconnects (HOTI)*. IEEE, 2015. https://doi.org/10.1109/HOTI.2015.17.

[12] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv e-prints*, 2015. http://arxiv.org/abs/1508.03619.

[13] John Beckett. NUMA best practices for Dell PowerEdge 12th generation servers: Tuning the Linux OS for optimal performance with NUMA systems. http://en.community.dell.com/techcenter/extras/m/white_papers/20266946, 2012. Accessed: January 2019.

[14] Daniel Berrangé. Openstack performance optimization: NUMA, large pages & CPU pinning. KVM Forum, 2014. Accessed: January 2017.

[15] Timothy Brecht. On the importance of parallel application placement in NUMA multiprocessors. In *USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS)*, 1993. https://www.usenix.org/legacy/publications/library/proceedings/sedms4/full_papers/brecht.txt.

[16] Broadcom. M150PM - 1 x 50gbe OCP 2.0 Multi-Host Adapter. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/50gb-nic-ocp/m150pm, 2018. Accessed: January, 2020.

[17] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010. http://doi.org//10.1109/PDP.2010.67.

[18] Georgios Chatzopoulos, Rachid Guerraoui, Tim Harris, and Vasileios Trigonakis. Abstracting multi-core topologies with MCTOP. In *European Conference on Computer Systems (EuroSys)*, 2017. https://doi.org/10.1145/3064176.3064194.

[19] Cisco Systems, Inc. Understanding EtherChannel load balancing and redundancy on catalyst switches. http://www.cisco.com/c/dam/en/us/support/docs/lan-switching/etherchannel/12023-4-01.pdf, 2007. Accessed: January 2019.

[20] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. *http://lwn.net/Articles/488709*, 2012.

[21] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. http://dx.doi.org/10.1145/2451116.2451157.

[22] Dell Engineering. Personal email communication, 2020.

[23] Linux network teaming driver. http://libteam.org/files/teamdev.pp.pdf, 2013. Accessed: April 2019.

[24] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 1–34, 1997. https://doi.org/10.1007/3-540-63574-2_14.

[25] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, Aug 2004. http://dl.acm.org/citation.cfm?id=1012889.1012894.

[26] Network RSS. https://wiki.freebsd.org/NetworkRSS. Accessed: January 2017.

[27] NUMA. https://wiki.freebsd.org/NUMA. Accessed: January 2019.

[28] ioMemory VSL: peak perforamnce guide. https://support.fusionio.com/load/-media-/2fk40u/docsConfluence/ioMemory_VSL_Peak_Performance_Guide_2013-08-20.pdf, 2013. Accessed: January 2019.

[29] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA systems. *Communications of the ACM*, 58(12):59–66, 2015. https://doi.org/10.1145/2814328.

[30] Brice Goglin and Stéphanie Moreaud. Dodging non-uniform I/O access in hierarchical collective operations for multicore clusters. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. IEEE, 2011. https://doi.org/10.1109/IPDPS.2011.222.

[31] Jiri Herrmann, Yehuda Zimmerman, Parker Parker, and Scott Radvan. Virtualization tuning and performance guide. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Virtualization_Tuning_and_Optimization_Guide/, 2016. Accessed: January 2017.

[32] HyperTransport Consortium. http://www.hypertransport.org (Accessed: Jan 2017).

[33] IEEE Std 802.3ad-2000: Amendment to carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications-aggregation of multiple link segments, 2000. https://doi.org/10.1109/IEEESTD.2000.91610.

[34] Creahan Research Inc. 400GbE to drive the majority of data center ethernet switch bandwidth within five years. http://www.crehanresearch.com/wp-content/uploads/2018/01/CREHAN-Data-Center-Networking-January-2018.pdf, 2018. Accessed: January 2019.

[35] Intel. DPDK: Data plane development kit. http://dpdk.org. (Accessed: May 2016).

[36] Intel. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html, Jan 2011.

[37] Intel. Intel data direct I/O technology (Intel DDIO): A primer. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012. Accessed: January 2019.

[38] Intel. Intel Ethernet Controller XL710. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf, 2016. Accessed: January, 2019.

[39] Intel. Intel ARK: Product Specifications. http://ark.intel.com/, 2017. Accessed: January 2019.

[40] Intel. Intel Xeon processor scalable family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, Sep 2017. (Accessed: Jan 2019).

[41] Intel. BIOS setup utility user guide for the Intel server board S2600 family supporting the Intel Xeon processor scalable family. https://www.intel.com/content/www/us/en/support/articles/000025892/server-products.html, Aug 2018. Article ID 000025892. Accessed: Jan, 2020.

[42] Rick A. Jones. Netperf: A network performance benchmark (Revision 2.0). http://www.netperf.org/netperf/training/Netperf.html, 1995. Accessed: August, 2016.

[43] Patryk Kaminski. NUMA aware heap memory manager. *AMD Developer Central*, page 46, 2009. https://developer.amd.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf (Accessed: Jan 2019).

[44] Andi Kleen. A NUMA API for Linux. *Novel Inc*, 2005.

[45] Maciek Konstantynowicz, Patrick Lu, and Shrikant M. Shah. Benchmarking and analysis of software data planes. https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf, Dec 2017. Whilte paper from FD.io – The Fast Data I/O Project.

[46] Fritz Kruger. CPU bandwidth - the worrisome 2020 trend. https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend, 2016. Accessed: January 2017.

[47] Andrey Kudryavtsev. An Introduction to Dual-Port NVMe SSD. https://itpeernetwork.intel.com/an-introduction-to-dual-port-nvme-ssd, 2016. Accessed: August, 2019.

[48] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. MemProf: A memory profiler for NUMA multicore systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[49] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[50] Linux Team infrastructure specification. https://github.com/jpirko/libteam/wiki/Infrastructure-Specification. Accessed: August 2019.

[51] Linux Ethernet bonding driver HOWTO. https://www.kernel.org/doc/Documentation/networking/bonding.txt. Accessed: April 2019.

[52] Page migration. https://www.kernel.org/doc/Documentation/vm/page_migration. Accessed: January 2017.

[53] Scaling in the Linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. Accessed: January 2017.

[54] Zoltan Majo and Thomas R. Gross. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. In *International Symposium on Memory Management (ISMM)*. ACM, 2011. https://doi.org/10.1145/2076022.1993481.

[55] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk|Crypt|Net: Rethinking the stack for high-performance video streaming. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 211–224, 2017. https://doi.org/10.1145/3098822.3098844.

[56] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/, 1991-2007. Accessed: August 2019.

[57] Mellanox. Sockperf Network Benchmarking Utility. https://github.com/Mellanox/sockperf. Accessed: August, 2019.

[58] Mellanox. Press release: Introduction of ConnectX-3 40GbE. http://www.mellanox.com/page/press_release_item?id=1009, 2013. Accessed: January 2019.

[59] Mellanox. Press release: Introduction of ConnectX-4 100GbE. http://www.mellanox.com/page/press_release_item?id=1416, 2014. Accessed: January 2019.

[60] Mellanox. Introducing ConnectX-4 Ethernet SRIOV. https://lwn.net/Articles/666180/, 2015. Accessed: August, 2019.

[61] Mellanox. Mellanox ConnectX-4 VPI Adapter. http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-4_VPI_IC.pdf, 2016. Accessed: January, 2017.

[62] Mellanox. Mellanox multi-host evaluation kit. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Multi-Host_EVB_Kit.pdf, 2016. Accessed: January, 2017.

[63] Mellanox. Mellanox ConnectX-5 VPI Socket Direct Adapter. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_VPI_Card_SocketDirect.pdf, 2018. Accessed: January, 2019.

[64] Mellanox. Mellanox Socket Direct Adapters. http://www.mellanox.com/page/products_dyn?product_family=285&mtag=socketdc, 2018. Accessed: January, 2019.

[65] Mellanox. Product brief: ConnectX-6 200Gb/s. www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf, 2018. Accessed: January 2019.

[66] Mellanox. Mellanox ConnectX-6 EN Card. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf, 2019. Accessed: August, 2019.

[67] MS TechNet: receive side scaling. https://technet.microsoft.com/en-us/library/hh997036.aspx. Accessed: January 2017.

[68] MSDN: NUMA support. https://msdn.microsoft.com/en-us/library/windows/desktop/aa363804(v=vs.85).aspx. Accessed: January 2019.

[69] Tomer Y Morad, Gil Shomron, Mattan Erez, Avinoam Kolodny, and Uri C Weiser. Optimizing Read-Once Data Flow in Big-Data Applications. *IEEE Computer Architecture Letters*, 2016. https://doi.org/10.1109/LCA.2016.2520927.

[70] Timothy Prickett Morgan. Intel to challenge AMD with 48 core "Cascade Lake" Xeon AP. https://www.nextplatform.com/2018/11/05/intel-to-challenge-amd-with-48-core-cascade-lake-xeon-ap/, 2018. Accessed: January 2019.

[71] David Mytton. Network performance at AWS, Google, Rackspace and Softlayer. https://blog.serverdensity.com/network-performance-aws-google-rackspace-softlayer, Apr 2014. (Accessed: Jan 2019).

[72] NVM Express Workgroup. NVM Express (NVMe) specification – Revision 1.2. http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf, Nov 2014. Accessed: Jan 2015.

[73] Robert Olsson. Pktgen the Linux packet generator. In *Ottawa Linux Symposium (OLS)*, pages 19–32, 2005.

[74] I/O (PCIe) based NUMA scheduling. https://specs.openstack.org/openstack/nova-specs/specs/kilo/implemented/input-output-based-numa-scheduling.html, 2015. Accessed: January 2017.

[75] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vSwitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[76] Vijay Rao and Edwin Smith. Facebook's new front-end server design delivers on performance without sucking up power. https://engineering.fb.com/data-center-engineering/facebook-s-new-front-end-server-design-delivers-on-performance-without-sucking-up-power, 2016. Accessed: August, 2019.

[77] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi. Design, implementation, and evaluation of a NUMA-aware cache for iSCSI storage servers. *IEEE Transactions on Parallel and Distributed Systems*, 2015. https://doi.org/10.1109/TPDS.2014.2311817.

[78] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *Proceedings of the VLDB Endowment*, 9(4), 2015. https://doi.org/10.14778/2856318.2856319.

[79] Samsung. PM1725a NVMe SSD. https://www.samsung.com/semiconductor/global.semi.static/Samsung_PM1725a_NVMe_SSD-0.pdf, 2017. Accessed: August, 2019.

[80] Lee T Schermerhorn. Automatic page migration for Linux [a matter of hygiene]. In *linux.conf.au*, 2007.

[81] Lance Shelton. High performance I/O with NUMA systems in Linux. *Linux Foundation Event*, 2013.

[82] Ronak Singhal. Inside Intel next generation Nehalem microarchitecture. In *Hot Chips*, 2008. http://www.hotchips.org/wp-content/uploads/hc_archives/hc20/3_Tues/HC20.26.630.pdf (Accessed: Jan 2017).

[83] Jeff Squyres. Process and memory affinity: why do you care? *High Performance Computing Networking—Cisco Blog*, 2013. http://blogs.cisco.com/performance/process-and-memory-affinity-why-do-you-care (Accessed: Jan 2017).

[84] Roman Sudarikov and Patrick Lu. Hardware-Level Performance Analysis of Platform I/O. https://static.sched.com/hosted_files/dpdkprcsummit2018/f6/Roman%20Sudarikov%20-%20DPDK_PRC_Summit_Sudarikov.pptx, 2018. Accessed: August, 2019.

[85] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *European Conference on Computer Systems (EuroSys)*, 2007. https://doi.org/10.1145/1272998.1273004.

[86] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013. https://doi.org/10.1109/HPCA.2013.6522318.

[87] VMware Technical Publications. Tuning vCloud NFV for data plane intensive workloads, Open Stack Edition. https://docs.vmware.com/en/VMware-vCloud-NFV-OpenStack-Edition/3.0/vmwa-vcloud-nfv30-performance-tunning.pdf. White Paper. Accessed: January 2019.

[88] VMware Technical Publications. VMware ESX Server 2 NUMA support. http://www.vmware.com/pdf/esx2_NUMA.pdf. White Paper. Accessed: January 2019.

[89] VMware Technical Publications. vSphere Resource Management: How ESXi NUMA Scheduling Works. https://docs.vmware.com/en/VMware-vSphere/6.7/vsphere-esxi-vcenter-server-671-resource-management-guide.pdf. White Paper. Accessed: January 2019.

[90] Andreas Wittig. EC2 network performance cheat sheet. https://cloudonaut.io/ec2-network-performance-cheat-sheet/, 2018. Accessed: January 2019.

[91] Bruce Worthington. NUMA I/O optimizations. Windows Hardware Engineering Conference (WinHEC), 2007. Accessed: January 2017.

[92] Xen NUMA roadmap: IONUMA support. https://wiki.xen.org/wiki/Xen_on_NUMA_Machines. Accessed: January 2017.

[93] Xen on NUMA machines. https://wiki.xen.org/wiki/Xen_NUMA_Roadmap. Accessed: January 2017.

[94] Dimitrios Ziakas, Allen Baum, Robert A. Maddox, and Robert J. Safranek. Intel QuickPath Interconnect architectural features supporting scalable system architectures. In *IEEE Symposium on High Performance Interconnects (HOTI)*, pages 1–6, 2010. http://dx.doi.org/10.1109/HOTI.2010.24.