NetDIMM: Low-Latency Near-Memory Network Interface Architecture

Mohammad Alian
University of Illinois Urbana-Champaign

Nam Sung Kim*
University of Illinois Urbana-Champaign

ABSTRACT

Optimizing bandwidth was the main focus of designing scale-out networks for several decades and this optimization trend has served well the traditional Internet applications. However, the emergence of datacenters as single computer entities has made latency as important as bandwidth in designing datacenter networks. PCIe interconnect is known to be latency bottleneck in communication networks as its latency overhead can contribute to up to ~90% of the overall communication latency. Despite its overheads, PCIe is the de facto interconnect standard in servers as it has been well established and maintained for more than two decades. In addition to PCIe overhead, data movements in network software stack consume thousands of processor cycles and make ultra-low latency networking more challenging. Tackling PCIe and data movement overheads, we architect NetDIMM, a near-memory network interface card capable of in-memory buffer cloning. NetDIMM places a network interface card chip into the buffer device of a dual in-line memory module and leverages the asynchronous memory access capability of DDR5 to share the memory modules between the host processor and near-memory NIC. Our evaluation shows NetDIMM, on average, improves per packet latency by 49.9% compared with a baseline network deploying PCIe NICs.

CCS CONCEPTS

• **Hardware → Networking hardware**; *Dynamic memory*; • **Computer systems organization** → *Client-server architectures.*

KEYWORDS

network architecture, near-memory computing

ACM Reference Format:
Mohammad Alian and Nam Sung Kim. 2019. NetDIMM: Low-Latency Near-Memory Network Interface Architecture. In *MICRO '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358278

*Nam Sung Kim's current affiliation is Samsung Electronics but this work has been done while he was at UIUC.

MICRO '52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358278

1 INTRODUCTION

Traditionally, the main design requirement for scale-out networks was high bandwidth. To ensure fairness and avoid congestion, network transport protocols such as TCP [41] have thrived. For the past three decades, such network architecture has served well throughput oriented Internet applications such as file and email servers. Even for interactive web applications, such as web search, that are sensitive to the per packet delivery time, a response time of several hundreds of milliseconds is considered acceptable as long as it can satisfy a service level objective, often defined as 99^{th} percentile response time. This throughput oriented network design has driven the development of high bandwidth network devices such as 100Gb+ Ethernet network interface cards (NIC).

The proliferation of datacenters and emerging applications over the past few years has changed network design requirements. In addition to high bandwidth, low-latency communication has become a primary metric for evaluating the next generation of scale-out networks. Ultra-low latency applications such as in-memory caching, high-performance computing, and financial trading [18, 24, 49] benefit from even sub microsecond latency improvements in the network hardware and software stack.

Ethernet, as the backbone of datacenter networking technology, is tightly coupled with the TCP/IP protocol to ensure reliable and fair communication between nodes in a datacenter. The deployment of TCP offload engines [11, 31, 43, 69] along with more efficient implementation of the software stack [5, 12, 14, 21, 22, 36, 44] has significantly reduced the computational overhead in the software stack of Ethernet networks. For instance, RDMA over converged Ethernet (RoCE) protocol technically offloads the whole network software stack to the Ethernet NIC device by implementing a priority flow control inside the NIC to make the Ethernet lossless [23]. A RoCE network can achieve node to node latency as low as ~1.3μs [4] by minimizing the software stack overhead. These technological advancements have made it possible to achieve end to end network latency that is close to hardware limits.

PCIe is a widely used and well-established server I/O interconnect technology. PCIe is used to connect off-chip storage, network, and accelerator devices to the processor chip. A bleeding edge ×16 PCIe Gen 4.0 provides a theoretical bandwidth of 31.51GBps. PCIe has a layered architecture and the protocol overhead at each layer reduces the usable bandwidth and adds to the latency overhead [28]. Therefore, PCIe interconnect is known to be the bottleneck especially in low-latency communication networks [34, 45, 50, 52, 59]. Frequent transactions over PCIe interconnect are the main contributor to the end to end network latency of software-stack optimized networks. For example, the PCIe subsystem contributes to 77.2~90.6% of the overall network latency for sending packets of various size over an ExaNIC 10Gbps NIC [59]. Besides the PCIe overhead, data copying from DMA buffers to application memory

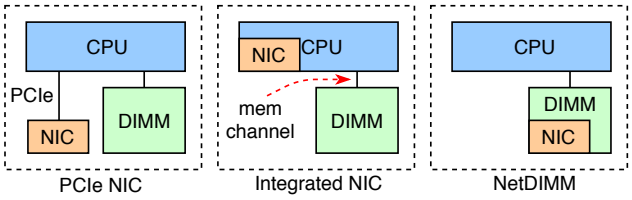

Figure 1: State of the art network interface architectures vs. NetDIMM

space is a major bottleneck in network subsystem that can constitute 18~92% of the per-byte operation overhead for different network protocols [27, 29].

To confront the PCIe and memory movement bottlenecks, previous works have proposed several solutions: (S0) reducing the number of PCIe transactions needed for packet transmission and reception [34, 50, 52], (S1) integrating the network interface card to the processor chip [26, 53], (S2) integrating a large memory buffer to the NIC [35, 47], (S3) adding processing units to the NIC and offloading the software to the NIC logic [33, 46], and (S4) developing zero copy networking [29]. Although these techniques can alleviate some of the network overhead, each has several drawbacks. S0 proposals mostly minimize the number of PCIe transactions for small packets. Moreover, the NIC is still connected to the processor through a PCIe interconnect and at least one round-trip over PCIe is needed for sending or receiving a packet. S1 designs are costly due to the area and power overhead for the processor chip. Furthermore, NIC and processor chips are often manufactured by different vendors and it is not practical to integrate them into one chip. Lastly, integrated NIC can pollute on-chip CPU resources when receiving large packet sizes (Sec. 5.3). Even though S2 and S3 can accelerate some applications, these techniques cannot benefit general purpose applications and are often hard to manage/program. Moreover, such NIC architectures still suffer from PCIe overhead. Regarding S4, although zero copy networking eliminates the data copying from DMA to application buffers, it introduces several problems including security breaches, main memory exhaustion, and extra virtual memory operation overheads that can nullify its benefits [27, 63].

In this work, we propose **Net**work attached **DIMM** (NetDIMM), a novel near-memory network interface card that utilizes a high speed DDR5 channel to interconnect a near-memory NIC to the processor. NetDIMM integrates a NIC into the buffer device of a dual inline memory module (DIMM) and uses the low-latency, high-bandwidth memory channel to communicate with the processor. NetDIMM leverages the asynchronous memory access support of DDR5 specification to seamlessly expose its local memory capacity to the host processor as if it is part of the host processor address space. Furthermore, NetDIMM supports in-memory buffer cloning that provides the performance of zero copy networking without its drawbacks. More specifically, NetDIMM makes the following contributions:

- *Eliminate the PCIe bottleneck in the network subsystem.* NetDIMM uses the memory channel instead of PCIe link to interconnect a NIC to the processor.
- *In-memory acceleration of network stack data movements.* NetDIMM accelerates the DMA between NIC and DRAM by placing the NIC close to the DRAM modules. Furthermore, NetDIMM performs in-memory buffer cloning to accelerate data movements in the network stack.
- *Application-transparent network stack acceleration.* NetDIMM runs the kernel software stack with minimal modification in the Linux kernel. Therefore NetDIMM can run unmodified userspace applications.
- *Reducing memory interference from the network traffic.* NetDIMM reduces the host memory channel utilization by using the local memory channels of NetDIMM for transferring packets between the memory and NIC. NetDIMM also split header and payload of packets that reduces on-chip resource pollution.

Fig.1 compares NetDIMM with the state of the art NIC architectures. NetDIMM significantly improves the communication latency by eliminating costly PCIe transactions and leveraging the physical proximity of NIC and DRAM for data movement. Based on our evaluation results, across various packet sizes, NetDIMM on average reduces the one-way network latency between two servers by 49.9% and 25.9% compared with servers employing PCIe and integrated NICs, respectively. We also replay traces from three Facebook production clusters and observe 25.3~40.6% average per packet latency reduction when replacing PCIe NICs with NetDIMM across different clusters. Lastly, we show that depending on the network application running on a server, co-running applications that use the same memory channel as NetDIMM can experience up to 30.9% lower memory access latency while in worst case experiencing 15.4% higher memory access latency compared with running the workloads on a system with an integrated NIC.

The rest of this paper is as follows. We first start with background information on the conventional network architecture, DDR5 support for asynchronous memory access, and memory management in Linux kernel. Sec. 3 motivates NetDIMM design. Sec. 4 explains the NetDIMM architecture. Sec. 5 includes the evaluation results. We talk about related works in Sec. 6. Sec. 7 is conclusion.

2 BACKGROUND

2.1 Network Architecture

Despite a large body of research, the innovations in Internet network architecture have been limited to incremental updates and its architecture has remained more or less the same since the creation of the Internet. The main reason for this resistance to changes is the multi-provider nature of the network echosystem that any change in the existing architecture needs a consensus among several stakeholders. Moreover, this network architecture has been reliably working for several decades and radical changes in it have become increasingly difficult.

Fig. 2 shows the overall network hardware architecture of a server. A NIC is connected to a processor over a PCIe link. Modern NICs use the Data Direct I/O (DDIO) technology [9, 38] to reduce memory bandwidth utilization when sending and receiving network packets. That is, when a packet is received at a NIC, a DMA engine transfers the packet to a buffer inside processor's last level cache (LLC) instead of moving it all the way to DRAM. When transmitting a packet with a DDIO enabled NIC, the packet buffer is allocated in LLC and DMA engine reads the packet from LLC.

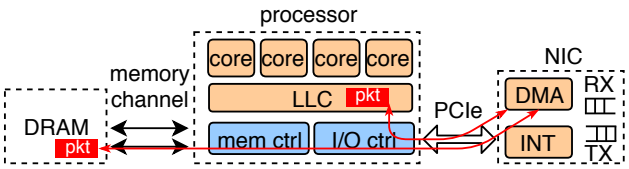

Figure 2: Server network architecture.

However, the DDIO share is usually 10% of the LLC capacity [9] (*i.e.* a few megabytes) and often this space is exhausted by a NIC at high RX/TX rates. Moreover, sharing the DDIO space between several network functions can result in a phenomenon known as DMA leakage [68]. The DDIO can cause cache pollution for other applications if there is no upper limit for its LLC share [67].

An Ethernet NIC employs a circular ring buffer (*i.e.* descriptor ring) inside the main memory to let the processor and NIC produce and consume packets at different rates. Because interrupt handling and interrupt moderation can delay the packet processing for several microseconds, ultra-low latency networks are usually deployed in (adaptive) polling mode [5, 51]. Here we explain NIC, CPU, and memory interactions when transmitting (TX) and receiving (RX) a packet using an Ethernet NIC with a polling driver. Before any transmission or reception (*i.e.* during the system boot up), the NIC driver allocates RX and TX descriptor rings, initializes them and sends their information to the NIC. (T1 - @Driver) The transmit function of the driver is called and the driver checks the status of the NIC. (T2 - @Driver) The driver sets up a DMA transfer by writing into a NIC configuration register. (T3 - @NIC) The DMA device fetches the next available TX descriptor from DRAM (or LLC if the DDIO is enabled) and then performs another DMA to transfer the packet to the NIC. (T4 - @NIC) The packet is transmitted over the Ethernet link and the TX ring tail pointer is updated. (R0 - @NIC) The packet is received at the destination NIC. (R1 - @NIC) The next available RX descriptor is fetched from DRAM or LLC (R2 - @NIC) The packet is DMAed to the RX descriptor buffer. (R3 - @NIC) The RX descriptor ring information is updated. (R4 - @Driver) Polling driver is notified of a new packet reception. (R5 - @Driver) A new socket buffer (*i.e.* SKB) is created and initialized with the data in the RX ring buffer. The Ethernet header is removed, and the rest of the packet is sent to an upper network layer.

2.2 Asynchronous Memory Access

In this subsection, we discuss nonvolatile dual-inline memory module (NVDIMM) protocols. We specifically talk about NVDIMM-P and how DDR5 specification manages to interact with such memory technology. The NVDIMM technology offers persistence and high memory capacity while using the memory channel, that is the fastest interconnect in the system, to interface with the processor. Based on JEDEC standard, there are three types of NVDIMMs:

(1) NVDIMM-N consists of byte-addressable DRAM modules and a backup NAND flash device. In NVDIMM-N, the host DDR memory controller only addresses the DRAM part of the NVDIMM-N. NVDIMM-N has the access time of a regular DDR DIMM from the host perspective.

(2) NVDIMM-F directly exposes the NAND flash storage to the processor and removes the DRAM devices. NVDIMM-F cannot be accessed with regular DDR timing and the memory channel has to slow down to meet the NVDIMM-F timing.

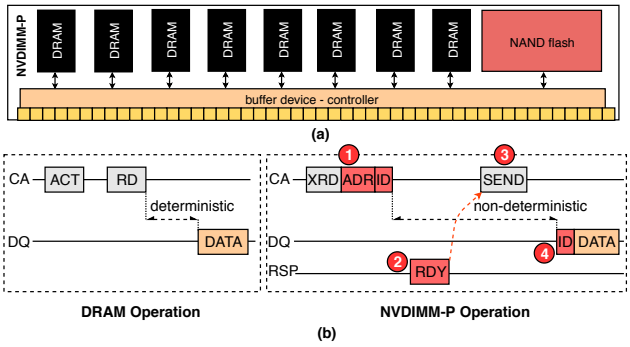

Figure 3: (a) NVDIMM-P architecture, (b) asynchronous memory access for NVDIMM-P.

(3) NVDIMM-P uses a novel memory channel protocol that allows asynchronous, out of order completion of the memory accesses to have the best features of both NVDIMM-N and NVDIMM-F. NVDIMM-P exposes both DRAM and NAND flash to the host processor address space. Because the NAND Flash (or any other persistence memory technology such as 3D-XPoint [55]) has different access timing compared with DRAM, a conventional DDR protocol cannot be used to access the persistent memory region. DDR5 specification is designed to comprehend the heterogeneous media type and support a mixture of convectional DIMM and NVDIMM-P. To facilitate NVDIMM-P accesses, DDR5 specification supports asynchronous memory transactions [1]. Fig. 3(b) compares the timing of a cacheline read from DRAM and NVDIMM-P in the DDR5 standard. As shown, to access a cacheline from NVDIMM-P, depending on the location of the data (if it is cached in the buffer device of NVDIMM-P or not), a read access has non-deterministic latency. A read request to NVDIMM-P starts with a read request (*i.e.* XRD in Fig. 3(b)) command that includes the full address of the requested data and a request ID. Unlike DRAM operations, each NVDIMM-P request has an ID to facilitate out of order access completion. When the XRD command is received at NVDIMM-P, the media data read command is immediately issued. Once the data is ready in the media, a ready command (*i.e.* RDY) is issued on the response pins (*i.e.* RSP) with the ID of the original request. The memory controller then issues a send (*i.e.* SEND) command to read the data. The data appended with the request ID is available on the data bus (*i.e.* DQ) after a specific amount of time.

2.3 Linux Memory Management

Memory Address Mapping. Different systems use different physical memory address mapping and decode different bits in the physical address to calculate the channel, rank, bank, row and column of the address location in the DRAM. If there are DIMMs installed on multiple memory channels, then the memory mapping can operate at three different modes as follows: single channel, multi-channel, and flex channel modes. In single channel mode, the memory channel bits are mapped to the most significant bits of the physical address and sequential addresses are mapped to one memory channel. In multi-channel mode, sequential memory

addresses are interleaved between multiple memory channels. Flex mode provides a flexible memory mapping configuration where a part of address space can work in multi-channel mode and the rest in single channel mode. Flex mode is especially useful in asymmetric memory configuration where different DIMM types (*e.g.* DDR5 or NVDIMM-P) are installed on memory channels [16].

Linux Kernel Memory Allocation. Due to hardware limitations, different parts of physical memory should be treated differently by Linux kernel. Linux groups physical memory locations that have same set of properties into different zones. Linux has four primary memory zones: ZONE_DMA: contains pages that can be used for DMA; ZONE_DMA32: contains pages that can be used for DMA by 32-bit devices; ZONE_HIGHMEM: contains "high memory" [7] pages that cannot be mapped into the kernel address space in 32bit machines; ZONE_NORMAL: contains regularly mapped pages in the system.

`kmalloc()` is used to allocate memory in kernel, similar to `malloc()` in userspace. `kmalloc()` can allocate memory from a specific memory zone based on the input arguments. There are also several APIs for allocating memory in page granularity in Linux. These APIs are especially used in the network stack for allocating the paged area of the network socket buffers [62]. The core function for page allocation is `__alloc_pages()`. There are several wrapper APIs to allocate pages from a specified NUMA node and/or memory zone.

3 MOTIVATION

As we discussed in Sec. 2.1, to send a packet over a conventional NIC, several PCIe and memory channel transactions need to take place. More specifically, in a client-server application, 16 one-way PCIe transactions are needed for completing one request-response transfer. Several research studies have proposed new NIC and DMA architectures to reduce the number of PCIe transactions when sending and receiving network packets, especially for small packets [34, 50, 52]. Although such architectures improve the network latency, they still require several PCIe round-trips to send and receive packets to and from the NIC, respectively.

CPU and NIC integration is a promising approach for solving the overheads mentioned above. Fig. 4 shows the one-way latency of sending packets of different size from one node to another through a 40Gb Ethernet link. For more information on our evaluation methodology please refer to Sec. 5.1. We evaluate four different NIC configurations: discrete NIC (`dNIC`), which represents a conventional PCIe Gen3×8 NIC (*i.e.* Fig. 1(left)); `dNIC` with zero copy transmission and reception (`dNIC.zcpy`); a NIC integrated into CPU chip (`iNIC`) (*i.e.* Fig. 1(middle)); and `iNIC` with zero copy transmission and reception (`iNIC.zcpy`). The figure also shows PCIe contribution to the overall packet transmission and reception (`pcie.overh` in Fig.4). As shown, `iNIC` improves the network latency by 21.3~38.6% compared with `dNIC`. The latency improvement is more signified for smaller packets and mainly comes from faster accesses to the I/O registers. Fig. 4 clearly shows the benefit of removing PCIe link between the CPU and NIC for low-latency networking.

We enable zero copying by letting NIC to access application buffers as DMA buffers. Zero copy improves `iNIC` network latency by 28.8% and 52.3% for 10Byte and 2000Byte packets, respectively.

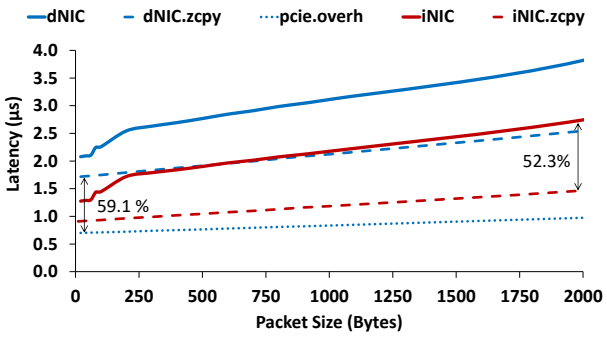

Figure 4: One-way latency comparison of different NIC configurations for packets of various sizes: discrete NIC (`dNIC`), discrete NIC with zero copy (`dNIC.zcopy`), integrated NIC (`iNIC`), and integrated NIC with zero copy (`iNIC.zcpy`). `pcie.overh` shows the overhead of PCIe interconnect for discrete NIC configurations.

As expected, memory copy overhead increases with packet size and larger packets benefit more from zero copy networking. On the other hand, the PCIe overhead is more for smaller packets. For `dNIC.zcpy`, 40.9% and 34.3% of the overall network latency is spent in PCIe interconnect when transferring 10Byte and 2000Bytes packets, respectively.

Although `iNIC.zcpy` seems to be an ideal ultra-low latency network architecture, it has several limitations: (**L1**) Zero copy networking can introduce security breaches [63]. Also pinning application pages to the memory can cause main memory exhaustion and the overhead of virtual memory operations and buffer management can nullify the gains of zero copy networking [27]. (**L2**) Integrating a full-blown NIC into CPU significantly increases the area and power of the processor. It is specifically challenging as often NIC and CPU are manufactured by different vendors. (**L3**) Most importantly, `iNIC` can pollute on-chip resources, such as LLC, at high network rates or cause memory interference for co-running applications. Furthermore, storing the payload of received packets on the processor chip is waste of precious on-chip resources for network functions that only require packet header to be processed by the CPU [32]. Note that (L3) is not specific to `iNIC` and `dNIC` also has the same problem.

To illustrate the memory and cache interference caused by network packets, we study the sensitivity of network bandwidth to the cache and memory interference. Fig. 5 depicts the sensitivity of network bandwidth to the pressure on the memory system. In this experiment, we use two machines, each equipped with a Xeon E5-2660 processor, three DDR4 memory channels, and an Intel 40Gbps XL710-QDA1 NIC. We use Intel Memory Latency Checker (MLC) [40] tool to inject dummy memory requests to the memory subsystem at different rates. We set the ratio of memory read to write requests to 1. In Fig. 5, the X-axis shows the delay between injected memory requests (higher values lower the interference at the memory subsystem) and Y-axis shows the achieved `iperf` [10] TCP bandwidth at different memory interference levels. `iperf` bandwidth significantly drops when the memory pressure from MLC increases. For example, at the maximum memory pressure, which

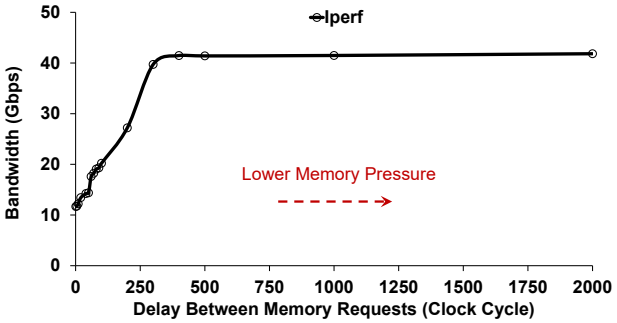

Figure 5: `Iperf` bandwidth at different memory pressure levels.

corresponds to 15.1GBps per memory channel, `iperf` only delivers ~27.9% of the achieved bandwidth without any interference from MLC. This experiment shows how sensitive network bandwidth is to the interference at the memory subsystem. Moreover, Fig. 5 can be interpreted from another angle: the network traffic can cause severe interference at the memory subsystem. However, here we could not show that because TCP flows from `iperf` regulate the transmission rate based on the processing capability of the receiver node. Therefore, before we see any major degradation on the local application performance, the `iperf` bandwidth decreases.

Fig. 4 and Fig. 5 illustrate the inefficiencies in the network architecture of current servers. Ideally, we want to completely remove the PCIe transactions and exchange data between the processor and NIC over an interconnect with lower latency without jeopardizing the network bandwidth. Furthermore, to reduce the memory interference, we want to decrease the host memory subsystem utilization when sending and receiving packets to and from NIC; which involves preventing a NIC from injecting all the received traffic to LLC. Instead, we want a mechanism which collectively brings different bytes of a received packet to the processor on the application's demand. PCIe is a standard and well-developed interconnection technology that has been around for three decades. One key requirement for a replacement is that it should be a standard and well-established interconnection technology. Introducing a new and specialized interconnect is costly and error prone. Also, the new interconnect should seamlessly work with memory channel and processor cache hierarchy to facilitate quick data delivery to the CPU.

Memory channel has the lowest latency amongst off-chip interconnects in a modern server. Besides low latency, memory channel provides high bandwidth. For example, a DDR4 channel provides 12.8GBps (*i.e.* 102.4Gbps) bandwidth. The latency of transferring a 4KB page over a DDR4 channel and a ×8 PCIe link are ~200ns and ~2μs, respectively. More importantly, the memory channel is a standard and well maintained interconnect that can be find on the motherboard of any server. We leverage these unique features of the memory channel and propose a near-memory network interface card architecture by placing a NIC into the buffer device of a DIMM. This design solves all the limitations of `dNIC` and `iNIC`: (1) eliminating the PCIe overhead by utilizing memory channel and internal DIMM interconnects for packet transmission and reception; (2) supporting in-memory buffer cloning to copy packets from application to DMA buffers and vice versa; (3) decoupling header and payload of packets to reduce LLC pollution and (4) using a separate memory channel to access network buffers in the DRAM to reduce the host memory channel interference.

4 NETWORK-ATTACHED DIMM

Motivated by the explanation in Sec.3, we propose NetDIMM, a low-latency, near-memory network architecture. Building atop the NVDIMM-P architecture (Sec. 2.2) and based on the near-memory processing concept, NetDIMM improves the data transfer latency between the processor, memory, and NIC. In this section, we explain the hardware and software components of NetDIMM in detail.

4.1 NetDIMM Hardware Architecture

Inspired by the asynchronous, out of order memory access support of DDR5 specification (Sec. 2.2), we architect a NIC that is placed on the buffer device of a DIMM. Fig. 6 overviews the overall architecture of NetDIMM. Fig. 6(c) shows a system with two memory channels where each memory channel is occupied with three DIMMs in total. Out of these three DIMMs, there are two conventional DDR5 DIMMs, and one NetDIMM. Note that the figure only shows an example system and there is no requirement for the number of NetDIMMs on a memory channel. For example, a system can have one NetDIMM installed on one of the DDR5 slots. The DDR5 support of asynchronous memory request completion allows mixing DRAM and NetDIMM on a same memory channel [15]. As shown in Fig. 6(b), the organization of NetDIMM is similar to the organization of an NVDIMM-P depicted in Fig 3(a).

Fig. 6(a) shows the internal architecture of NetDIMM buffer device. It consists of the following main components: (nNIC) An integrated network interface card; (nMC) one (or several) memory controller(s) to access the NetDIMM local DRAM modules; (nController) logic that extends the NVDIMM-P controller with NetDIMM routing and management logic; (DDR5 PHY interface) DDR5 physical interface and protocol engine. The DDR5 physical interface contains a protocol engine that repeats DRAM CA, DQ, and RSP signals similar to a typical NVDIMM-P device; (nCache) a dual-port SRAM buffer for caching RX data resided in the local DRAM modules; (nPrefetcher) a next-line prefetcher for pre-loading RX packets to nCache from the local DRAM modules; (RowClone enabled DRAM) DRAM devices that support in-memory data copying.

We expose the local DRAM capacity of NetDIMM to the host memory address space, therefore, the local NetDIMM memory is managed by the host operating system. This is similar to the unified address space of NVDIMM-P. We explain NetDIMM memory management in Sec 4.2. Because both nNIC and PHY can independently access the local DRAM modules through nMC, we need arbitration between the memory accesses from nNIC and PHY. nController does this arbitration by giving priority to the nNIC accesses. Because of the following reasons, the access time to the local DRAM from the host MC is non-deterministic: (R1) the host MC does not know the state of the NetDIMM local DRAM modules; (R2) nMC is shared between nNIC and PHY. Thus, the access time of the local DRAM modules depends on the current state of the local DRAM modules, the current nNIC traffic, and the current requests from PHY.

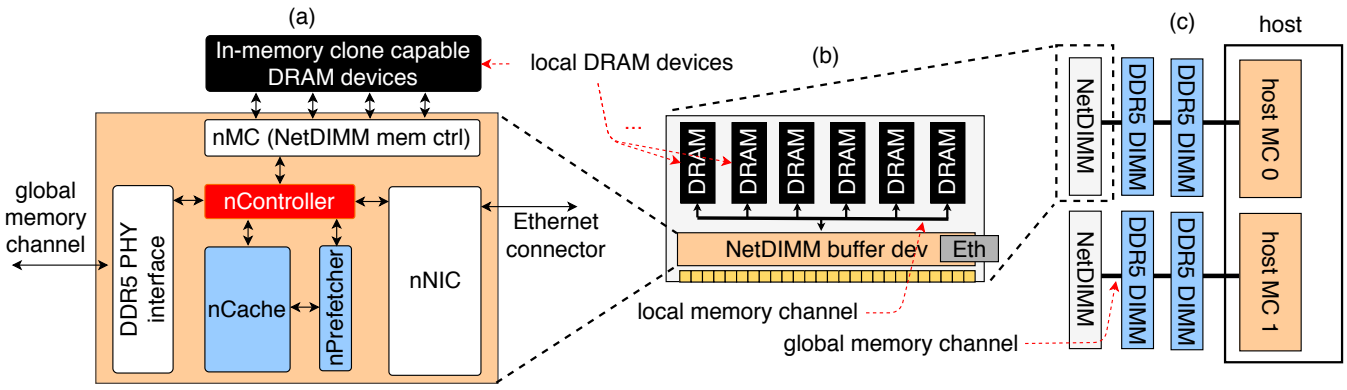

Figure 6: NetDIMM architecture.

We make a key observation that the memory access pattern between the host processor and NIC is very regular and has spatial and temporal locality. Fig. 7 plots the relative address and relative arrival time of memory requests, generated by the DMA engine of a 40GbE NIC, when receiving six 1514 Byte packets. For detailed experimental setup please refer to Sec. 5.1. As illustrated, each packet arrival generates a burst of memory requests to DMA buffers. Each burst consists of 24 cachelines[1] (24 * 64 = 1536 Bytes) that arrive at the host memory controller in a short time interval, which for example is 143ns for the third packet. nCache and nPrefetcher components exploit the unique characteristics of this memory access pattern to improve the host MC access latency to the NetDIMM address space.

Once a packet is received at nNIC from the outside, nNIC notifies nController. nController implements the same functionality of a DMA engine in a conventional NIC. Upon receiving the notification from nNIC, nController reads the next available descriptor buffer from nMC and depletes the RX buffer of nNIC to the descriptor ring resided in the NetDIMM local DRAM modules. In Sec. 4.2 we explain how the descriptor ring is allocated on NetDIMM. While transferring the RX packets to the NetDIMM local DRAM space, the nController writes the first cacheline of each received packet to nCache. The rationale for only caching the first cacheline of received packets is that for all transport protocols, the header size is less than 64 Bytes (*i.e.* one cacheline) and only the header of a received packet is needed for processing the packet in the network software stack[2]. Moreover, as explained in Sec. 3, some network functions, such as forwarding and firewall, do not need the packet payload as the application makes forwarding decisions only based on the header information. The maximum header size of a TCP/IP packet is 52Bytes [17], so caching the first 64Bytes of a received packet includes all the headers. The rest of the packet is only accessed when copying it to a userspace buffer. Storing an entire received packet in nCache is not efficient as the reuse distance of the payload of a received packet is much longer than its header.

Assuming a maximum transmission unit (MTU) size of 1500 Bytes, each Ethernet packet can carry 1~24 cachelines. When the payload of a received packet is accessed (*e.g.* to be copied to an application buffer), a stream of consecutive read requests is received to NetDIMM PHY, similar to the access pattern shown in Fig. 7. This access pattern is easy to predict by a simple next-line prefetcher. We add this prefetcher, shown as nPrefetcher in Fig. 6(a), to NetDIMM. nPrefetcher prefetches the next n cachelines and stores them in nCache. Therefore, even if NetDIMM does not cache the payload of RX packets in nCache, in the worst case, reading an entire RX packet may only experience one nCache miss. We disable nPrefetcher for the first cacheline of RX packets which contains the header. This is because we do not want to pollute nCache when only the header of a received packet is accessed by the host processor. We add a one-bit flag for each cacheline of nCache, that is set when the first cacheline of a newly arrived packet is stored in nCache. nPrefetcher checks this flag and prefetches next n cachelines if the flag is not set. nCache resets the flag after the first access to each cacheline.

When a read request is received from the global memory channel, nController checks if the requested data is cached in nCache. If it is a hit, the data is read from nCache and immediately sent to the host MC. Otherwise, nController creates a read request and sends it to nMC. Once the data is read from the local DRAM through nMC, it will be sent to the host using the asynchronous protocol explained in Sec. 2.2. When a write request is received from the global memory channel, nController constructs a memory write

[1]Unless stated otherwise, we assume that the cacheline size is 64Bytes throughout this paper

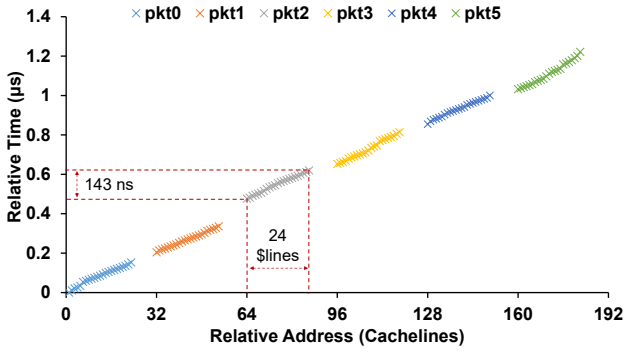

Figure 7: Spatial and temporal locality of NIC memory accesses from host processor perspective

[2]Assuming that nNIC has checksum offloading support

request and send it to nMC. The write requests do not use nCache as they are immediately queued in the nMC write queue upon arrival.

nCache is an inclusive, set associative cache structure. nCache is more like a large data buffer and its data is removed from it once it is accessed. This is because once the RX packet is read from NetDIMM, it is going to be stored in a host processor cache or in another location in the main memory. In either case, that memory address is unlikely to be accessed in a near future. Therefore, there is no value in keeping that data in nCache. We use random replacement policy to make space in an nCache set if all the blocks in the set are occupied. Note that all cachelines in nCache are clean and there is no need for writing a victim cacheline back to nMC. To ensure the coherency of nCache with local DRAM data, nController snoops the addresses of write requests received from PHY or nNIC and invalidates the matching cachelines in nCache.

Conventionally, copying one memory location to another involves a processor to read data over its memory channels into its cache hierarchy and then write it back through the memory channels to the destination memory location. This makes memory copying an expensive operation. For example, copying a 4KB page over a DDR3 memory channel takes ~1μs [61]. Because of the limitation of zero-copy drivers (discussed in Sec. 3), we envision an in-memory data copy acceleration mechanism to swiftly clone application buffers to DMA buffers and vise versa on NetDIMM. To this extent, we utilize an extended implementation of RowClone [61] mechanism. RowClone is an in-memory bulk data copying mechanism that utilizes DRAM internal architecture to accelerate memory-to-memory copying on a single DIMM. Fig. 8 illustrates a high level overview of in-memory clone-capable DRAM devices. Depending on the location of the source and destination addresses, there are three modes for cloning a page: Fast parallel mode (FPM): source and destination pages share a bank sub-array. In this case buffer cloning can be done by two back to back activation of the source and destination pages. FPM mode is highlighted with green arrows in Fig. 8; Pipeline serial mode (PSM): source and destination pages are on different banks but on a same DRAM device. In this case cloning happens by pipelining cacheline copy operations over the internal bus of DRAM chips. PSM mode is highlighted with the red arrow in Fig. 8; General cloning mode (GCM): otherwise, NetDIMM reads source to the NetDIMM buffer device and writes them back in pipeline mode to the destination address (highlighted by blue arrows in Fig. 8). GCM is similar to the operation of a conventional DMA engine near the memory chips. FPM is the fastest while GCM is the slowest and most general mechanism. That being said, it is important to intelligently allocate source and destination pages to a same sub-array within a DRAM device in order to extract the maximum benefit from the in-memory page cloning. In Sec. 4.2.1 we explain how NetDIMM implements an intelligent memory allocation scheme to efficiently move data from DMA buffers to application buffers.

4.2 NetDIMM Software Architecture

In this subsection, we explain required software stack changes to enable NetDIMM. Overall, we try to have the minimum amount of changes possible in the network software stack and Linux kernel. The changes in the software stack includes implementation of a

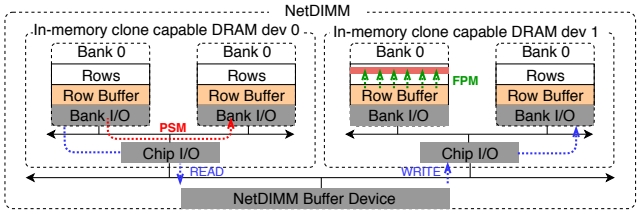

Figure 8: In-memory buffer cloning acceleration

new Linux memory allocation API, changing the physical memory address mapping, and implementation of a NetDIMM driver. The TCP/IP layers remain unchanged except for the API for SKB allocation. Note that we developed a userspace NetDIMM driver for our evaluations. However, to show the feasibility and generality of our implementation, we also developed a Linux kernel NetDIMM driver that runs the full Linux kernel software stack and unmodified userspace applications. We use our Linux kernel driver for explanation here.

4.2.1 Handling NetDIMM Local Memory Region. Before we talk about NetDIMM driver, we first need to discuss how we use the local DRAM modules on NetDIMM. To leverage the operating systems memory management functionality, keep the amount of changes in the software stack at minimum, and make NetDIMM application-transparent, *we expose the local memory capacity of NetDIMM to the host processor as if it is part of the host physical memory address space.* The local memory capacity of a NetDIMM can be seen as a memory node in a NUMA system, and despite different access timing, NetDIMM's memory space is part of the host (global) address space. We reveal this heterogeneity in the memory system to Linux by creating a new memory zone called NETi where i is the NetDIMM number in the system. Note that a system can have multiple NetDIMMs installed on memory channels and each need a different memory zone. Defining a memory zone in Linux is not expensive and new memory zones has been added to Linux when necessary [2].

In addition to defining new memory zones, it is also important to intelligently allocate DMA and application buffers on a same bank and sub-array to extract the maximum performance out of NetDIMM's in-memory buffer cloning capability (*cf.* Fig 8). To achieve this, we need to expose the internal memory organization of NetDIMM to the memory scheduler. Fig. 9(a) shows our assumptions about the size and organization of a memory rank in NetDIMM, which is based on a Micron MT40A512M16 DRAM device [56]. Each rank consists of eight ×8 DRAM devices, each device consists of 16 banks, each bank is divided into 512 sub-arrays, and each sub-array consists of 128 rows. The capacity of each rank, device, bank, sub-array, and row is 8GB, 64MB, 128KB, and 1KB, respectively. Based on this organization, the physical memory address mapping for NetDIMM looks like Fig. 9(b). Assuming a page size of 4KB, Fig. 9(c) illustrates the geometric location of consecutive pages stored in a memory rank. As shown, the pages that are physically stored on a same bank and sub-array are spaced every 128KB (or 32 pages). Thus, it is easy to check if two pages are on a same sub-array and bank. We implement `__alloc_netdimm_pages(zone, hint)` that allocates a page on NetDIMM `zone` and the same sub-array as `hint` address. If `hint` is set to -1, then the API only considers the `zone`

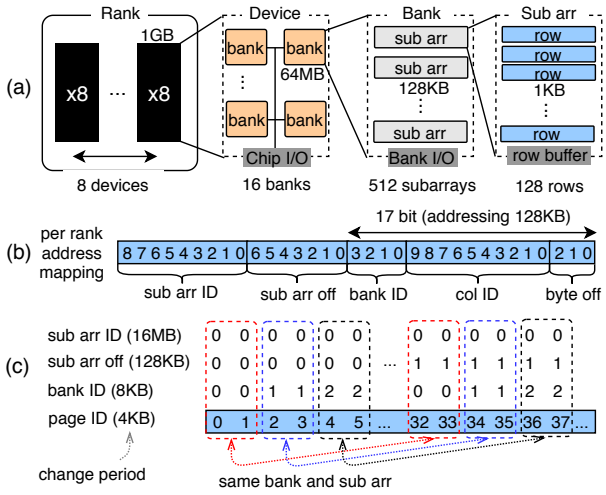

Figure 9: (a) Configuration of a memory rank in a NetDIMM; (b) physical memory address mapping; (c) illustration of the physical location of pages

requirement. Note that this is a best effort API and it is possible that the allocated pages is not on the same sub-array as `hint` address.

Another complexity in handling the local memory area of NetDIMM is the memory channel interleaving of physical addresses in a systems with multiple memory channels. Memory channel interleaving increases the memory throughput by parallelizing memory accesses over several memory channels. However, we need to disable the memory channel interleaving for the NetDIMM address space because the global memory channels are not visible to nNIC (Fig. 6). Therefore, the NetDIMM address space should be exposed to the host in single channel mode so the host processor sees the NetDIMM physical address as a continuous memory chunk. We leverage the Flex channel interleaving mode (*cf.* Sec. 2.3) to divide the physical address space into two parts. One part contains all conventional DDR DIMMs that operate in multi-channel mode and another part contains NetDIMMs address space operating in single channel mode. Fig. 10 depicts the unified address space of the conventional DIMMs and NetDIMMs and their memory channel interleaving modes.

4.2.2 NetDIMM Driver. We use Intel e1000 GbE driver as a base to develop NetDIMM driver. Because NetDIMM is not a PCIe device, `ioremap()` API is used to create a configuration space for NetDIMM similar to the configuration space of a conventional PCIe NIC. Using this techniques, we can configure all the features of a full-blown NIC without the need for writing a new driver from scratch.

When a NIC interface is initialized, it creates transmit (TX) and receive (RX) descriptor ring buffers and initializes their buffer pointers by allocating DMA buffers. Moderns NICs support scatter-gather DMA operation, so a DMA buffer can span over multiple pages that are not physically contiguous. NetDIMM requires that the physical location of descriptor rings and their corresponding DMA buffers to be on the memory zone of the corresponding NetDIMM. To benefit from in-memory cloning acceleration, applications are also required to allocate their network data buffers on NetDIMM memory zone. For both TX and RX rings, we use

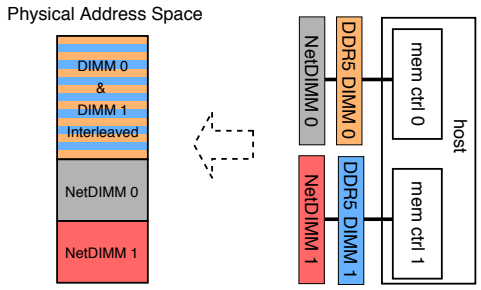

Figure 10: The memory address space and channel interleaving mode for a mixture of DDR5 DIMMs and NetDIMMs

`__alloc_netdimm_pages(zonei, -1)` to allocate descriptor ring data structures for NetDIMM*i*. For RX and TX DMA buffers, we allocate them on the fly based on the location of application buffers. However, calling `__alloc_netdimm_pages` for each packet can deteriorate the network latency and bandwidth. As shown on Fig. 9(a) each NetDIMM rank has `512 * 16 = 8K` distinct sub-arrays. To accelerate the on-demand memory allocation, NetDIMM pre allocates two pages from each distinct sub-array and stores them in a hash table called `allocCache`. Considering that NetDIMM has two memory ranks, each NetDIMM pre allocates 32K pages (*i.e.* 128MB) for on-demand DMA buffer allocation. This corresponds to 0.8% of capacity overhead for a 16GB NetDIMM. `allocCache` immediately returns a page allocated on a specific sub-array. NetDIMM driver refills `allocCache` concurrently in the background, thus, the on-demand allocation of DMA buffers are not in the critical path of packet RX and TX.

One complication here is that an application should have knowledge about the physical layer to know which NetDIMM is serving its packet streams. To resolve this, we add a flag to the SKB header (or any other type of network data structure used for networking) called `COPY_NEEDED`. We allocate the SKBs that belong to the connection establishment on the regular kernel address space and set the `COPY_NEEDED` flag in the SKB header. At the transmit function of NetDIMM driver, if `COPY_NEEDED` flag is set, the driver first copies the SKB data to an allocated TX DMA buffer on the corresponding NetDIMM and then initiates the packet transmission. Each SKB has a pointer to the socket that the packet is associated with. We add a new field to "`struct sock`" called "`struct zone_struct skb_zone`" and set it to NET*i* in the NetDIMM driver. Therefore, after the first packet transmission, each connection has enough information to allocate the SKB and paged buffers of the TX packets on a corresponding NetDIMM memory zone. Note that `COPY_NEEDED` flag is also used as a fallback mechanism in case the memory space on a NET*i* zone is exhausted and the SKB and TX buffers are allocated on different memory zones. This is a rare event and does not happen frequently.

When receiving a packet from NetDIMM, similar to a PCIe NIC, once nNIC finished moving a received packet to a DMA buffer, it needs to notify the host processor. To notify the processor about newly received packets or packet transmission completions, a NIC typically uses an interrupt signal or a polling agent. The interrupt approach is mostly used for high bandwidth network connections where the network latency is not critical. On the other hand, a polling mechanism is mainly used by userspace network stacks and

Algorithm 1: Packet TX and RX handling at NetDIMM driver.

```
TX:
txDesc[next].dma = allocCache[txSKB.data] // DMA buffer allocation
if txSKB.COPY_NEEDED then
    copy txDesc[next].dma ← txSKB.data    // slow path
    set skb_zone to NETi
    flush txDesc[next].dma to memory
else
    flush txSKB.data to memory            // fast path
set txDesc[next] size and flags           // total size is 64 bits
flush txDesc[next] size and flags         // kick off transmission
RX:
invalidate rxDesc[next]                   // to fetch fresh data from NetDIMM
rxSKB.data = allocCache[rxDesc[next].dma] //RX buffer allocation
netdimmClone(rxSKB.data, rxDesc[next].dma, rxDesc[next].size) //  in-memory buffer cloning
send rxSKB to upper network layers for processing
Polling Agent:
clean TX buffers after a successful transmission
if newly arrived packet then
    call RX
```

low-latency networks to prevent interrupt processing and context switching overheads (*cf.* Sec. 2.1).

NetDIMM driver implements an efficient polling agent using a high-resolution kernel timer. Note that polling NetDIMM is more efficient than polling a PCIe NIC as accessing I/O registers on a NetDIMM is much faster than a PCIe NIC. After the polling driver detects a packet arrival, it calls the RX routine of the driver as shown in Alg. 1. NetDIMM uses memory flush and invalidate instructions to enforce coherency between processor caches and NetDIMM local memory. The `netdimmClone(dst, src, size)` function shown in Alg. 1 is the API for in-memory buffer cloning. It writes `dst`, `src`, and `size` values to a set of NetDIMM registers and NetDIMM clones `src` to `dst` buffer inside the memory.

4.3 Physical Feasibility of NetDIMM

One question that yet to be answered by this paper is how feasible is to integrate a full-blown NIC into the buffer device of a DIMM in terms of power and thermal specifications. There are products [6, 8, 13, 54, 64] and academic research proposals [19, 30] that add processing power to the buffer device of conventional DIMMs. Centaur DIMM (CDIMM) [54] is a buffered DIMM, designed by IBM to scale the memory capacity of POWER processors. CDIMM comprises of up to 80 DDR DRAM devices and a Centaur device that consists of a 16MB L4 cache, four memory controllers, and other controlling logic. The TDP of an IBM Centaur buffer device is 20W in 22*nm* technology. On the other hand, a modern XXV710 Intel PCIe Ethernet controller incorporating 2×40Gbps ports has a TDP of 6.5W [39]. Therefore, considering the specification of the current DIMM products, it is feasible to integrate a NIC chip into the buffer device of a DIMM. Lastly, we always can connect an external power cable to DIMMs similar to an NVDIMM [13]. Moreover, we can use a similar connector for the network cable in NetDIMM.

5 EVALUATION

5.1 Methodology

We evaluated NetDIMM using `gem5` [25] along with analytical models for PCIe interconnect [20, 59] and memory controller [37]. Because the overhead of Linux kernel software stack fades the latency improvements of NetDIMM, we implement a set of bare-metal drivers for our PCIe NIC, integrated NIC and NetDIMM models using `gem5` that resemble low-latency userspace drivers and use them for latency evaluations. We configure `gem5` as shown in Table. 1.

To model NetDIMM memory access latency, we instantiate an isolated memory controller that models nMC shown in Fig. 6(c). nMC model is used to access NetDIMM memory zone. A memory request from host to NetDIMM is first queued in a host MC. Once it is chosen to be sent to the DRAM, instead of performing a regular memory access, after a `tCMD` delay, the host MC forwards the memory request to a corresponding nMC. The memory request access is completed once the nMC sends a response to the host MC. For the network DMA operations, the memory accesses are directly sent to the nMC model.

For performance evaluations, we use network traces from three Facebook production clusters. Each cluster has different packet size and traffic patters: first cluster is for `database` applications with their packet size uniformly distributed between 64 Bytes and 1514 Bytes (MTU is set to 1514 Bytes), second cluster is for `webserver` where ∼90% of the packet sizes are smaller than 300 Bytes, and third cluster is uses for `hadoop` servers where ∼41% of packet are less than 100Bytes and ∼52% are 1514 Bytes [60]. The traffic pattern of `database` cluster is mostly inter-cluster and inter-datacenter, `webserver` is mostly inter-cluster but intra-datacenter, and `hadoop` is intra-cluster. The traces are publicly available by Facebook [42]. We randomly pick one node in each cluster and use several dummy nodes to replay the ingress and egress data traffic to and from the node under test. We simulate the clos network topology of Facebook datacenter using dist-gem5 [58] switch model. We assume all the network devices in the datacenter has a bandwidth of 40Gbps. We implement an L3 Forwarding (`L3F`) and a deep packet inspection (`DPI`) network functions as two network functions with extremely different packet processing behaviours to evaluate the impact of NetDIMM on the performance of server memory subsystem. We use the Facebook traces to exercises these network functions. `L3F` forwards received packets only based on their header information while `DPI` process the entire header and payload to make a forwarding decision.

Table 1: System configuration.

Parameters	Values
Cores (# cores, freq):	(8, 3.4GHz)
Superscalar	3 ways
ROB/IQ/LQ/SQ entries	40/32/16/16
Int & FP physical registers	128 & 192
Branch predictor/BTB entries	BiMode/2048
Caches (size, assoc): I/D/L2	32KB,2/64KB,2/2MB,16ways
L1I/L1D/L2 latency,MSHRs	1/2/12 cycles, 2/6/16 MSHRs
DRAM	DDR4-2400MHz/16GB/2 channels
Network/Switch latency/#NetDIMM	40GbE/100ns/1
PCIe performance	×8 PCIe 4 [59]

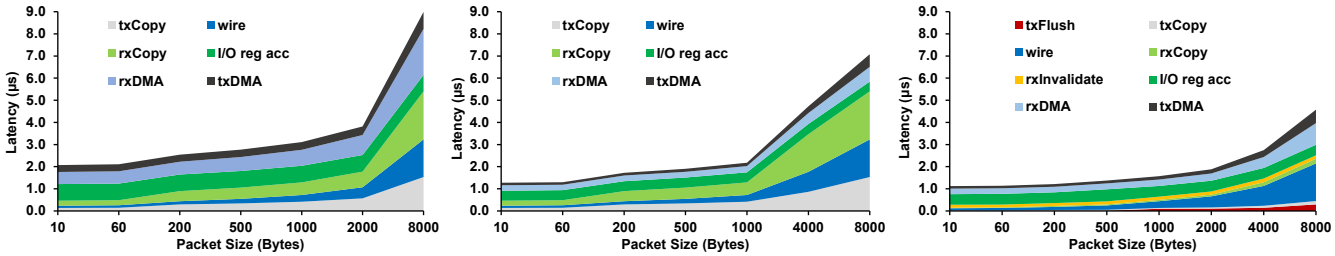

Figure 11: One-way network latency breakdown for packets of various sizes when using a PCIe NIC (left), an integrated NIC (middle), and NetDIMM (left). X-axis is not drawn to scale.

5.2 Network Latency and Bandwidth

Fig. 11 shows one-way network latency breakdown of various sized packets between two nodes directly connected together by PCIe NICs (left), iNICs (middle), and NetDIMMs (right). `rxCopy` and `txCopy` respectively show the overhead of memory copy and allocation at RX and TX drivers, `rxDMA` and `txDMA` show the DMA overhead at NIC hardware, `wire` shows the physical layer overhead, and `I/O reg acc` represents the overhead of CPU/NIC register accesses. `txFlush` and `rxInvalidate` represent cache flush and cache invalidate overheads of NetDIMM driver, respectively. NetDIMM reduces the one-way network latency by 46.1%, 52.3%, and 49.6% for 64B, 256B, and 1024B packets compared with a PCIe NIC, which translates to 0.97μs, 1.33μs, and 1.54μs lower network latency, respectively. As shown in Fig. 11(middle and right), because of eliminating PCIe interconnect, `I/O reg acc` is significantly reduced for iNIC and NetDIMM compared with that of PCIe NIC. NetDIMM adds `txFlush` and `rxInvalidate` overheads to the end to end network latency. These two components combined add 9.7~15.8% overhead to the total network latency. Nonetheless, on average NetDIMM delivers 26.0% lower latency than iNIC across different packet size. This shows that the in-memory buffer cloning not only makes up for the overhead of CPU cache operations, but also improves the overall network latency compared with an integrated NIC.

One caveat of NetDIMM is that unlike a PCIe NIC, it is siting on one memory channel and it cannot utilize multiple memory channels when communicating with the host processor and memory. However, our simulation results show that NetDIMM delivers 40Gbps bandwidth just like our PCIe and integrated NIC models. This is not a surprise as the nominal bandwidth of a DDR4 memory channel is 12.8GBps or 102.4Gbps, which is far more than 40Gbps. In fact DDR5 memory channel's projected bandwidth is twice more than that of a DDR4 channel which can sustain any bandwidth of what the current or under development PCIe NICs can deliver.

5.3 Performance Evaluation

Fig. 12(a) shows the average per packet network latency for each cluster with servers using NetDIMM normalized to the latency of PCIe NIC and iNIC configurations. We set the latency of network switches inside the simulated clos network to 25ns, 50ns, 100ns, and 200ns to measure the performance sensitivity of NetDIMM to different network configurations. On average, across different clusters, NetDIMM improves the end to end packet latency of PCIe NIC configuration by 40.6%, 36.0%, 33.1%, and 25.3% when switch latency is 25ns, 50ns, 100ns, and 200ns, respectively. NetDIMM improves the average end to end packet latency of different clusters employing iNIC by 8.1~15.3% for different switch configurations. As expected, NetDIMM latency reduction is more highlighted when lower latency network switches are used. Fortunately, the latency of network switch products is improving and today's ultra-low latency network switches offer port to port latency of less than 6ns [3].

Among all clusters, `webserver` benefits the most from NetDIMM because over 90% of its packets are less than 300Bytes and NetDIMM is more effective when transferring small packets. In addition, `webserver` traffic is intra-datacenter and it traverses lesser hops to reach a destination compared with `database` traffic that is mostly inter-datecenter. Although `hadoop` traffic is local to the cluster, its packets are skewed to either small- or MTU-sized packets, therefore, NetDIMM latency reduction is the lowest for `hadoop` amongst the other two clusters.

Fig.12(b) shows the normalized memory access latency observed by a co-running application when running a `DPI` and `L3F` on servers with NetDIMM. The values are normalized to that of iNIC. Because DPI make forwarding decisions based on the packet payload, the processor should fetch the entire packet to its caches and process both header and payload. Because an iNIC directly brings the received packets to the LLC, it does not consume memory channel bandwidth and if the processor is not congested, each received packet can be processed and forwarded before it gets evicted to the DRAM. However, `L3F` only need packet header to decide where to forward a received packet, which is naturally done by nCache at NetDIMM. Based on this packet processing behaviour, `DPI` and `L3F` are two ends of packet processing spectrum and any other applications falls between these two. Fig. 12(b) shows that NetDIMM increases the memory access time by 5.7%~15.4% when running `DPI` and improves it by 9.8%~30.9% when running `L3F` compared with iNIC configuration. On average, NetDIMM improves the memory access latency by 9.3%, 2.4%, and 13.6% for `database`, `webserver`, and `hadoop` clusters respectively.

6 RELATED WORKS

Novel Network Architecture. Kim et al. [47, 48] proposed a caching mechanism inside NIC to reduce data communication over PCI channel. NIC cache is implemented using on-board DRAM devices and is managed by the operating system. Although this network architecture reduces PCIe traffic, incoming packets still need to traverse PCIe interconnect to reach CPU. Furthermore, designing

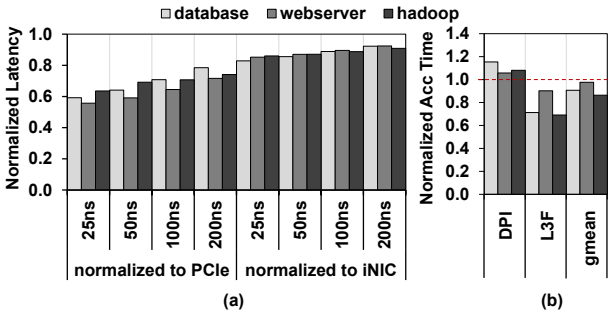

Figure 12: (a) Per packet normalized network latency for different network switching latency; (b) normalized memory access latency observed by a co-running application when running deep packet inspection (DPI) and L3 forwarding (L3F); using servers with NetDIMM and replaying Facebook cluster traces.

an efficient software managed data cache is challenging. Flajslik et al. [34] performed a detailed study on different sources of latency overhead in the network stack and found that minimizing the number of PCIe transactions is the key in designing a low latency NIC. They proposed a new NIC architecture called NIQ to reduce the communication latency, especially for small packets, by employing techniques such as embedding packets inside the buffer descriptors, custom polling, and creative use of caching policies. FlexNIC [46] is a network DMA interface design that reduces the packet processing overhead by enabling NIC to perform simple operations on the packets while exchanging them with the main memory. Offloading optimization is orthogonal to NetDIMM design and can be applied to NetDIMM to further improve the network performance. Liao et al. [52] proposal decouples the DMA descriptor management from other NIC functionality and moves it to processor side. This design aims to reduce the number of PCIe transactions and handle DMA buffers more efficiently. Larsen et al. [50] also introduced an integrated DMA engine to minimize the descriptor management overhead and PCIe transactions. Binkert et al. [26] proposed SINIC, which integrates a simple NIC into the processor die. SINIC uses PIO to exchange data between the processor, main memory and NIC. Although SINIC is effective in reducing the network latency, it has a high area cost. Furthermore, it does not suite for high bandwidth communication due to the lack of a DMA engine and other capabilities of modern NICs. Compared with these works, NetDIMM completely removes the PCIe link between NIC and processor, places a full-blown NIC near memory, and implements in-memory buffer cloning which in turn solve all the overheads of a conventional network subsystem.

Minnich et al. [57] proposed a memory-integrated NIC called MINI, that places a NIC behind the main memory DRAM modules. MINI implements a pseudo dual-port DRAM to share the DRAM space between host and NIC. This requires arbitration signals between host and NIC memory controllers. MINI need to redesign DRAM and memory controller interfaces to port to a new system architecture. MEMONet [66] and DIMMNET-2 [65] plug a NIC into a memory channel slot. Although these designs solve the PCIe bottleneck, they do not share the NIC and host address space and explicitly copy packets over the host memory channel for packet transmission and reception. Furthermore, these NICs can be used on a single memory channel system. On the other hand, NetDIMM seamlessly exposes its local memory address space to the host, minimizes the data movement between host and NIC, supports multi-channel memory systems, and lastly, NetDIMM does not require any change to the processor architecture and memory subsystem.

Novel Interconnection Technology. Alian et al. [19] introduced memory channel network (MCN) concept where they add a general-purpose mobile processor to a DIMM and expose the near-memory processors to the host processor as if they are connected through an Ethernet interface. They use memory channel to interconnect the remote nodes to the host processor. NetDIMM uses a similar concept to connect NIC, processor, and memory together. Open Coherent Accelerator Processor Interface (OpenCAPI), Cache Coherent Interconnect for Accelerators (CCIX), and Gen-Z are new interconnect standards under development that are mainly used to tightly couple processors and accelerators such as GPUs and FPGAs. CCIX is developed based on PCIe specifications and has PCIe drawbacks. The combination of DDR and such interconnection technologies provides unprecedented bandwidth and reduces data movement overhead by directly accessing the memory. Although CCIX, OpenCAPI and Gen-Z are three different standards, these are introduced and emerged to solve similar problems and they may merge into each other or abandoned in the future. However, DDR standard is maintained and developed for over two decades and is the standard interconnection technology for memory. Moreover, the latency of point-to-point serial interconnects such as CCIX, OpenCAPI and Gen-Z cannot match with that of a DDR memory channel.

7 CONCLUSION

For decades, the focus of scale-out network system design was to optimize its bandwidth. However, with the emergence of ultra-low latency datacenter applications, a need for low latency scale-out networks has unfolded. In this paper, building upon the near-memory processing concept and leveraging the asynchronous memory access of NVDIMM-P protocol, we designed and evaluated a near-memory NIC architecture called NetDIMM. NetDIMM integrates a full-blown NIC in to the buffer device of an in-memory buffer-cloning capable DIMM. We developed supporting logic and a device driver to make the near-memory NIC available to applications running on a host processor. Finally, we implemented a new memory zone for NetDIMM's local memory space and developed Linux kernel APIs to facilitate memory allocation from these memory zones. Such memory allocation significantly reduced the amount of data movement when processing network packets. Compared with a conventional PCIe NIC, NetDIMM improves the network latency by up to 52.9% without compromising the network bandwidth.

8 ACKNOWLEDGEMENT

This work was in part supported by an NSF grant (CNS-1705047).

REFERENCES

[1] [n. d.]. A prelude to nonvolatile DIMM technology. Future of NVDIMM-P. https://gigglehd.com/gg/hard/1893698 Accessed: 06/30/2019.

[2] [n. d.]. Add 4GB DMA32 zone. https://lwn.net/Articles/152337/. Accessed: 2018-11-20.

[3] [n. d.]. Arista 7130 Connect Series Ultra-low latency switches. https://www.arista.com/en/products/7130-series Accessed: 03/30/2019.

[4] [n. d.]. ConnectX-2 EN with RDMA over Ethernet (RoCE). http://www.mellanox.com/related-docs/prod_software/ConnectX-2_RDMA_RoCE.pdf. Accessed: 2018-11-28.

[5] [n. d.]. Data Plane Development Kit. http://dpdk.org/.

[6] [n. d.]. Diablo conjures up hell of a DIMM: 128GB NAND pretend-RAM summoned. https://www.theregister.co.uk/2016/07/22/diablos_devilishly_clever_nandbased_pretend_dram_dimms_now_shipping/ Accessed: 06/30/2019.

[7] [n. d.]. High memory. https://en.wikipedia.org/wiki/High_memory

[8] [n. d.]. Intel announces Optane DC Persistent Memory DIMMs. https://www.techspot.com/news/79483-intel-announces-optane-dc-persistent-memory-dimms.html Accessed: 06/30/2019.

[9] [n. d.]. Intel Data Direct I/O Technology (Intel DDIO): A Primer. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html

[10] [n. d.]. Iperf: The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/

[11] [n. d.]. Large Send Offload. https://en.wikipedia.org/wiki/Large_send_offload. Accessed: 2018-11-28.

[12] [n. d.]. Low-latency Ethernet device polling. https://lwn.net/Articles/551284/. Accessed: 2018-11-28.

[13] [n. d.]. MicronâĂŹs NVDIMM Delivers Persistent Memory. https://www.electronicdesign.com/industrial-automation/micron-s-nvdimm-delivers-persistent-memory Accessed: 06/30/2019.

[14] [n. d.]. Open Fast Path. https://openfastpath.org/. Accessed: 2018-11-28.

[15] [n. d.]. Overcoming System Memory Challenges with Persistent Memory and NVDIMM-P. https://www.jedec.org/sites/default/files/Bill_Gervasi.pdf. Accessed: 2018-12-5.

[16] [n. d.]. Single- and Multichannel Memory Modes. https://www.intel.com/content/www/us/en/support/articles/000005657/boards-and-kits.html Accessed: 06/30/2019.

[17] [n. d.]. TCP frame. https://en.wikipedia.org/wiki/Transmission_Control_Protocol. Accessed: 2018-03-25.

[18] [n. d.]. Wall Street's Quest To Process Data At The Speed Of Light. https://www.informationweek.com/wall-streets-quest-to-process-data-at-the-speed-of-light/d/d-id/1054287. Accessed: 2018-12-3.

[19] Mohammad Alian, Seung Won Min, Hadi Asgharimoghaddam, Ashutosh Dhar, Dong Kai Wang, Thomas Roewer, Adam McPadden, Oliver O'Halloran, Deming Chen, Jinjun Xiong, Daehoon Kim, Wen-mei Hwu, and Nam Sung Kim. 2018. Application-Transparent Near-Memory Processing Architecture with Memory Channel Network. In *The 51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE.

[20] Mohammad Alian, Krishna Parasuram Srinivasan, and Nam Sung Kim. 2018. Simulating PCI-Express Interconnect for Future System Exploration. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 168–178.

[21] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center tcp (dctcp). *ACM SIGCOMM computer communication review* 41, 4 (2011), 63–74.

[22] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 19–19.

[23] Motti Beck and Michael Kagan. 2011. Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*. International Teletraffic Congress, 9–15.

[24] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 267–280.

[25] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[26] Nathan L Binkert, Ali G Saidi, and Steven K Reinhardt. 2006. Integrated network interfaces for high-bandwidth TCP/IP. *ACM Sigplan Notices* 41, 11 (2006), 315–324.

[27] Eduard Bröse. [n. d.]. ZeroCopy: Techniques Benefits and Pitfalls. ([n. d.]).

[28] Ravi Budruk, Don Anderson, and Tom Shanley. 2004. *PCI express system architecture*. Addison-Wesley Professional.

[29] Willem de Bruijn and Eric Dumazet. [n. d.]. sendmsg copy avoidance with MSG_ZEROCOPY. ([n. d.]).

[30] A. Farmahini-Farahani, J. Ahn, K. Morrow, and N. S. Kim. 2015. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*.

[31] Wu-chun Feng, Pavan Balaji, Chris Baron, Laxmi N Bhuyan, and Dhabaleswar K Panda. 2005. Performance characterization of a 10-Gigabit Ethernet TOE. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*. IEEE, 58–63.

[32] Daniel Firestone. 2017. {VFP}: A Virtual Switch Platform for Host {SDN} in the Public Cloud. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 315–328.

[33] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[34] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-Response Protocols.. In *USENIX Annual Technical Conference*. 333–346.

[35] Eric S Fukuda, Hiroaki Inoue, Takashi Takenaka, Dahoo Kim, Tsunaki Sadahisa, Tetsuya Asai, and Masato Motomura. 2014. Caching memcached at reconfigurable network interface. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 1–6.

[36] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 29–42.

[37] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 201–210.

[38] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. 2005. Direct cache access for high bandwidth network I/O. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 50–59.

[39] Intel. 2018. *Intel Ethernet Controller X710/ XXV710/XL710 Datasheet*.

[40] Intel. 2018. *Intel Memory Latency Checker v3.5*.

[41] Van Jacobson. 1988. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, Vol. 18. ACM, 314–329.

[42] James Hongyi Zeng. [n. d.]. Data Sharing on traffic pattern inside Facebook datacenter network. https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/ Accessed: 03/30/2019.

[43] Hankook Jang, Sang-Hwa Chung, Dong Kyue Kim, and Yun-Sung Lee. 2011. An Efficient Architecture for a TCP Offload Engine Based on Hardware/Software Co-design. *J. Inf. Sci. Eng.* 27, 2 (2011), 493–509.

[44] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 489–502.

[45] Anuj Kalia Michael Kaminsky and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*. 437.

[46] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High performance packet processing with flexnic. In *ACM SIGARCH Computer Architecture News*, Vol. 44. ACM, 67–81.

[47] Hyong-youb Kim, Vijay S Pai, and Scott Rixner. 2002. Increasing web server throughput with network interface data caching. *ACM SIGPLAN Notices* 37, 10 (2002), 239–250.

[48] Hyong-youb Kim, Scott Rixner, and Vijay S Pai. 2005. Network interface data caching. *IEEE Trans. Comput.* 54, 11 (2005), 1394–1408.

[49] Ramana Rao Kompella, Kirill Levchenko, Alex C Snoeren, and George Varghese. 2009. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *ACM SIGCOMM Computer Communication Review*, Vol. 39. ACM, 255–266.

[50] Steen Larsen and Ben Lee. 2011. Platform IO DMA transaction acceleration. In *International Conference on Supercomputing (ICS) Workshop on Characterizing Applications for Heterogeneous Exascale Systems (CACHES)*.

[51] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. 2009. Architectural breakdown of end-to-end latency in a TCP/IP network. *International journal of parallel programming* 37, 6 (2009), 556–571.

[52] Guangdeng Liao, Xia Znu, and Laxmi Bnuyan. 2011. A new server I/O architecture for high speed networks. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 255–265.

[53] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. 2013. Thin servers with smart pipes: designing SoC accelerators for memcached. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 36–47.

[54] P. J. Meaney, L. D. Curley, G. D. Gilda, M. R. Hodges, D. J. Buerkle, R. D. Siegl, and R. K. Dong. 2015. The IBM z13 memory subsystem for big data. *IBM Journal of Research and Development* 59, 4/5 (July 2015), 4:1–4:11.

[55] Micron. [n. d.]. 3D XPoint^{TM} Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[56] Micron. [n. d.]. *Micron DDR4 SDRAM Datasheet.* https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[57] Ronald Minnich, Dan Burns, and Frank Hady. 1995. The memory-integrated network interface. *IEEE Micro* 15, 1 (1995), 11–19.

[58] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. 2017. dist-gem5: Distributed simulation of computer clusters. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 153–162.

[59] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 327–341.

[60] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 123–137. https://doi.org/10.1145/2785956.2787472

[61] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. In *MICRO*.

[62] Sameer Seth and M Ajaykumar Venkatesulu. 2009. *TCP/IP Architecture, Design, and Implementation in Linux*. Vol. 68. John Wiley & Sons.

[63] Jia Song and Jim Alves-Foss. 2012. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS)* 6, 4 (2012), 256.

[64] Bharat Sukhwani, Thomas Roewer, Charles L Haymes, Kyu-Hyoun Kim, Adam J McPadden, Daniel M Dreps, Dean Sanner, Jan Van Lunteren, and Sameh Asaad. 2017. Contutto: A novel fpga-based prototyping platform enabling innovation in the memory subsystem of a server class processor. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 15–26.

[65] N Tanabe, H Nakajyo, H Amano, M Yoshimi, A Kitamura, and T Miyashiro. 2006. DIMMnet-2: A reconfigurable board connected into a memory slot. In *2006 International Conference on Field Programmable Logic and Applications*. IEEE, 1–4.

[66] Noboru Tanabe, Junji Yamamoto, Hiroaki Nishi, Tomohiro Kudoh, Yoshihiro Hamada, Hironori Nakajo, and Hideharu Amano. 2000. MEMOnet: Network interface plugged into a memory slot. In *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*. IEEE, 17–26.

[67] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. 2010. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *International Symposium on High Performance Computer Architecture (HPCA)*. 1–12.

[68] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association.

[69] Wen-Fong Wang, Jun-Yau Wang, and Jin-Jie Li. 2005. Study on enhanced strategies for TCP/IP offload engines. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, Vol. 1. IEEE, 398–404.