2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Application-Transparent Near-Memory Processing Architecture with Memory Channel Network

Mohammad Alian[1], Seung Won Min[1], Hadi Asgharimoghaddam[1], Ashutosh Dhar[1], Dong Kai Wang[1], Thomas Roewer[2], Adam McPadden[3], Oliver O'Halloran[3], Deming Chen[1], Jinjun Xiong[2], Daehoon Kim[4], Wen-mei Hwu[1], Nam Sung Kim[1]
[1]*University of Illinois, Urbana-Champaign*, [2]*IBM Research*, [3]*IBM Systems*, [4]*DGIST*

***Abstract*—The physical memory capacity of servers is expected to increase drastically with the deployment of the forthcoming non-volatile memory technologies. This is a welcomed improvement for the emerging data-intensive applications. For such servers to be cost-effective, nonetheless, we must cost-effectively increase compute throughput and memory bandwidth commensurate with the increase in memory capacity without compromising the application readiness. Tackling this challenge, we present Memory Channel Network (MCN) architecture in this paper. Specifically, first, we propose an MCN DIMM, an extension of a buffered DIMM where a small but capable processor called MCN processor is integrated with a buffer device on the DIMM for near-memory processing. Second, we implement device drivers to give the host and MCN processors in a server an illusion that they are independent heterogeneous nodes connected through an Ethernet link. These allow the host and MCN processors in a server to run a given data-intensive application together based on popular distributed computing frameworks such as MPI and Spark without any change in the host processor hardware and its application software, while offering the benefits of high-bandwidth and low-latency communication between the host and MCN processors over the memory channels. As such, MCN can serve as an application-transparent framework which can seamlessly unify the near-memory processing within a server and the distributed computing across such servers for data-intensive applications. Our simulation running the full software stack shows that a server with 8 MCN DIMMs offers 4.56× higher throughput and consume 47.5% less energy than a cluster with 9 conventional nodes connected through Ethernet links, as it facilitates up to 8.17× higher aggregate DRAM bandwidth utilization. Lastly, we demonstrate the feasibility of MCN with an IBM POWER8 system and an experimental buffered DIMM.**

I. Introduction

The performance of servers running emerging data-intensive applications such as big-data analytic is often limited by the DRAM capacity and DDR bandwidth. The expected deployment of emerging memory technologies such as 3D XPoint [1] to servers will relieve the ever-increasing pressure on demanding larger memory capacity for such applications. However, for such servers to be cost-effective, we must increase the compute throughput and available memory bandwidth commensurate with the increase in memory capacity of servers.

As part of such effort, researchers have proposed various near-memory processing architectures that tightly integrate a processor with memory to expose higher bandwidth to the processor [2]–[12]. Such near-memory processing architectures, nonetheless, require significant changes in target applications especially to orchestrate the communication between the host and near-memory processors [2], [5], [13], [14]. This hurts application readiness and thus creates a big hurdle for wide adoption.

Tackling the application readiness challenge for near-memory processing, we start with an observation that many emerging data-intensive applications, which can benefit from near-memory processing, are often built upon distributed computing frameworks such as Hadoop [15], Spark [16] and MPI [17]. These distributed computing frameworks distribute given input data of an application and have many servers process the input data in parallel. As such, the high-level processing model of the recent near-memory processing architectures was inspired by the distributed computing frameworks [2], [6].

In this paper, building on the distributed computing frameworks and exploiting high bandwidth and low latency of DDR interfaces, we propose Memory Channel Network (MCN). Specifically, MCN aims to give the host and near-memory processors connected through a DDR interface in a server the *illusion* that these processors are connected through Ethernet links. Therefore, MCN can provide a standard and application-transparent communication interface not only between the host and near-memory processors in a server, but also among such servers, seamlessly unifying near-memory processing with distributed computing for data-intensive applications. MCN consists of the following hardware and software components.
(HW) MCN DIMM. We architect an MCN processor and place it in a buffer device of a buffered DIMM (between a host-side Memory Controller (MC) and its associated DRAM devices on the DIMM). We refer to this buffered DIMM as MCN DIMM. For an MCN processor, we may take a small but capable Application Processor (AP), such as Qualcomm Snapdragon 835 [18], as the MCN processor and then implement a simple MCN interface in it. The MCN interface in the buffered device is similar to a network interface but takes a DDR PHY instead of an Ethernet PHY, interfacing between a host-side MC and an MCN processor. Lastly, each MCN processor runs a lightweight OS with the network software layers essential for running a distributed computing framework.
(SW) MCN driver. We develop a special device driver, referred to as MCN driver, for the host and MCN processors to give them the illusion that they are computer nodes connected through an Ethernet interface. An MCN driver is similar to a NIC driver, but it intercepts a network packet from the network software layer in the OS and redirects it to MCs

instead of a NIC, if the network packet is destined to an MCN DIMM. Unlike a conventional NIC generating an interrupt to inform a host of new network packets, a memory interface (and MC) does not have a corresponding mechanism. Hence, we implement a special mechanism in the host-side MCN driver to determine whether any MCN DIMM is sending any network packet to the host or other MCN DIMMs.

These MCN DIMMs and associated drivers together allow a server to run an application based on a distributed computing framework *without any change in the host processor hardware, distributed computing middleware, and application software*, while offering the benefits of high-bandwidth and low-latency communications between the host and the MCN processors over memory channels. Furthermore, each MCN processor accesses its DRAM devices on the same MCN DIMM through its (local) memory channels isolated from the (global) memory channel shared with other DIMMs. Therefore, multiple MCN DIMMs can concurrently operate. That is, the aggregate memory bandwidth for processing is proportional to the number of MCN DIMMs. As such, MCN can serve as an application-transparent near-memory processing platform, as well as unify near-memory processing in a server with the distributed computing across multiple servers.

To further increase the utilized bandwidth and decrease the communication latency between MCN DIMMs, we propose optional software and hardware optimization techniques. Specifically, we optimize the MCN driver and some of the OS network layers, leveraging unique properties of MCN over the traditional Ethernet. We also show that additional communication efficiency can be achieved by changing the host-side MC to exploit a special signal of the recent and future memory interfaces. We use this signal to interrupt the host MC when an MCN DIMM has outgoing packets.

We model MCN DIMMs, develop MCN drivers, adapt some OS network layer, and demonstrate the full functionality in a full-system simulator running the entire software stack. Our evaluation shows that MCN offers 456.5% and 78.1% improvement in the network bandwidth and the latency compared with a conventional 10GbE network, respectively. Furthermore, an MCN-enabled server with 8 MCN DIMMs increases the aggregate memory utilization bandwidth for distributed applications by up to 8.17$\times$ compared with a conventional server.

Atop the simulation study, as a proof of concept, we take an experimental buffered DIMM which consists of an FPGA device and several DDR3 DRAM devices, implement an MCN processor on the experimental buffered DIMM, and demonstrate the feasibility of the MCN concept after plugging the DIMM into a memory channel of an IBM POWER8 system and installing the developed MCN drivers on the both IBM POWER8 host and the MCN DIMM.

II. Background

A. Memory Sub-system

Buffered DRAM modules. To strike a balance between memory capacity and bandwidth, multiple DRAM devices that operate in tandem compose a rank, and one or more ranks are packaged on a memory module. A popular memory module called Dual-Inline Memory Module (DIMM) has 64 data I/O (DQ) pins (plus 8 DQ pins for a DIMM supporting ECC). A memory channel connects an MC to one or more DIMMs. In a server class processor, an MC drives hundreds of DRAM devices and delivers Command/Address (C/A) signals through a memory channel to them. Considering the GHz operation frequency range of a modern DRAM device, this in turn leads to a serious signal integrity problem. For example, a C/A pin from a memory controller has to drive 144 DRAM devices (18$\times$4 devices per rank supporting ECC multiplied by 8 ranks) when 8 ranks are populated per channel, whereas a data pin is connected to 8 DRAM devices, which is an order of magnitude fewer. Therefore, DIMMs for servers typically employ a buffer per DIMM, such as Registered DIMM (RDIMM) [19] or Load-Reduce DIMM (LRDIMM) [20], to reduce this huge capacitive load imposed to an MC and alleviate the signal integrity problem. Fig. 1 depicts another DIMM type with a buffer, Centaur DIMM (CDIMM) [21]. Each CDIMM with a tall form factor comprises up to 80 commodity DDR DRAM devices and a Centaur device which provides a 16MB eDRAM L4 cache, memory management logic, and an interface between DDR and IBM proprietary memory interfaces. Note that the bandwidth available to the CPU remains constant as the memory channel is shared by all the DIMMs although the memory capacity increases with more DIMMs per channel.

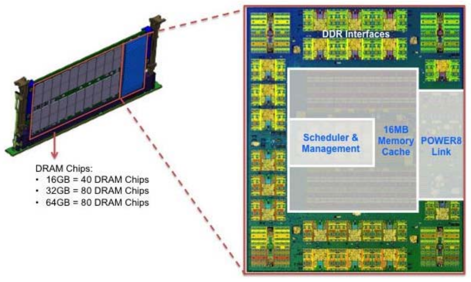


Fig. 1. IBM Centaur DIMM.

OS memory management for kernel space drivers. For virtual to physical address mappings, an OS manages hierarchical page tables, each with two or more levels, depending on a processor architecture [22]. During the booting process, the Linux kernel is responsible for setting up page tables and turning on the Memory Management Unit (MMU). By default, the Linux kernel and users assume that any virtual page can be mapped to any physical page. However, it is desirable to **(D1)** reserve a specific range of physical memory space exclusively for an (memory-mapped) I/O device and its driver, and **(D2)** allow the driver to access this physical memory range with virtual addresses since every address issued by the processor is a virtual address after the MMU is turned on.

In the Linux kernel, we can accomplish **(D1)** by editing the Device Tree Blob (DTB). A DTB is a set of attributes of the hardware components in a given system and is fetched during the booting process. Specifically, a node in a DTB represents a hardware component and describes information such as the number and type of CPUs, base physical addresses and sizes of memory devices, I/O devices, etc. To reserve a specific region of physical memory, we create a new node in the device tree, wherein a physical address range is explicitly enumerated and is tagged as `reserved_memory`. At boot time, the kernel will exclude this physical address range from mapping to other processes, thereby creating a *memory map hole*. Later, the reserved memory region can be assigned to a device driver by setting the `memory_region` parameter.

B. Network Sub-system

OS network layer. TCP/IP is the most commonly used protocol for the distributed computing frameworks. An application sends and receives data through a TCP socket using `tcp_sendmsg()` and `tcp_recvmsg()` system calls, respectively. When a user application calls `tcp_sendmsg()`, the data is copied to a kernel buffer, fragmented into several segments of Maximum Transmission Unit (MTU) size, undergoes TCP/IP processing, and eventually sent to a NIC for transmission. The MTU limit exists since sending a packet a large data size at once is vulnerable to random transient errors in traditional physical links, such as the Ethernet links, and increases the probability and the overhead of re-transmitting the packet. In Linux, the default value of MTU is 1500 bytes. On the receiver side, the segments of a message are reassembled inside the Linux kernel and the complete message is copied to the user-space application.

NIC and driver. Fig. 2 shows the interactions between a processor, memory, and a NIC when a network packet is received or transmitted. Once an outgoing packet is processed in the TCP/IP stack, it is written to a transmission ring (TX ring) buffer (Ⓐ) in the physical memory. Then, the NIC driver informs the NIC of the available packets in the TX ring (Ⓑ). Later, the NIC reads the ready-to-transmit descriptors from the TX ring and DMA-transfers the data from the memory to the NIC buffers(Ⓒ). Finally, the packet is sent out (Ⓓ).

Similar to the TX ring, the NIC driver manages a circular ring buffer in the memory for the incoming packets (RX ring in Fig. 2). When a packet is received (❶), the NIC DMA-transfers the data to the next available buffer in the RX ring (❷). When the DMA-transfer is done, a HW interrupt is sent to the processor (❸). Upon receiving the HW interrupt, the NIC driver schedules a softIRQ. When the softIRQ handler eventually executes, it prepares a socket buffer by assembling the data inside the RX ring (❹) and send it to a higher network layer for further processing. Note that once a NIC starts receiving packets, switching to a polling-based approach is often preferred to a pure interrupt-based approach. This is because the performance cost of handling many hardware interrupts is notable which can bottleneck the throughput of a high bandwidth network [23].

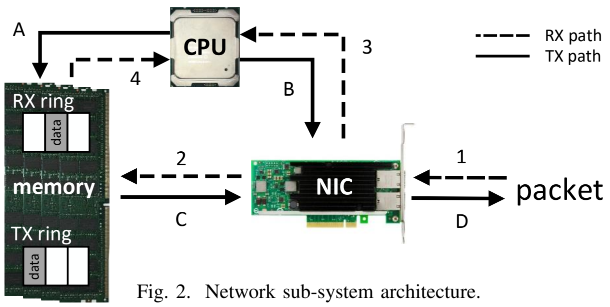


Fig. 2. Network sub-system architecture.

III. Memory Channel Network

The overall design of MCN is depicted in Fig. 3. The MCN DIMMs and MCN drivers are designed with two key objectives in mind. First, they should run applications based on the existing distributed computing frameworks *without any change in the host processor hardware, distributed computing middleware, and application software*. That is, MCN does not require any modification in the host processor and commodity DRAM architectures since it limits all hardware changes to a buffer device in a DIMM. Second, each MCN processor accesses its DRAM devices on the same MCN DIMM through its (local) memory channel isolated from the (global) memory channel which is shared with other DIMMs, as depicted in Fig. 3(b). Therefore, multiple MCN DIMMs can be concurrently accessed by an MCN processor through its local MCs, multiplying the aggregate memory bandwidth for processing [3], [24], as shown in Fig. 3(a) and (b). This is in contrast to a traditional memory subsystem, where the memory bandwidth for processing remains constant regardless of the number of DIMMs per memory channel; because multiple DIMMs share a global memory channel and the host processor can access only one DIMM at a time through the shared global memory channel. As such, MCN can serve as an *application-transparent near-memory processing platform*, as well as *unify the near-memory processing in a node with the distributed computing across multiple such nodes*, as illustrated in Fig. 3(d).

A. MCN DIMM Architecture

An MCN DIMM, also referred to as an MCN node, consists of an MCN processor and its associated DRAM devices. A host-side MC treats MCN DIMMs as buffered DIMMs (Sec. II-A) and thus supports a mix of multiple MCN and conventional DIMMs per memory channel, as depicted in Fig. 3(b).

In this work, we propose to place a small low-power but capable mobile processor used in APs on a buffer device[1] of each DIMM. For example, four ARM Cortex A57 cores with 2MB LLC, implemented in Samsung Exynos 5433, consume $\sim 2mm \times 2mm$ space and Thermal Design Power (TDP) of $\sim 1.8W$ after scaling the size and power in $20nm$ technology [26] to the size and power in $10nm$ technology. A Qualcomm Snapdragon 835 AP incorporates a quad-core 2.45GHz ARM Cortex A57 CPU, a 710MHz Adreno 540 GPU, two 1866MHz (LP) DDR4 memory channels, and a UFS2.1 storage interface. Even with other components specific for mobile applications such as an LTE modem, a camera image signal processor (ISP), digital signal processors, etc.,

[1]The size and TDP of an IBM Centaur buffer device is $\sim 10mm \times 10mm$ and $20W$ in $22nm$ technology [25].

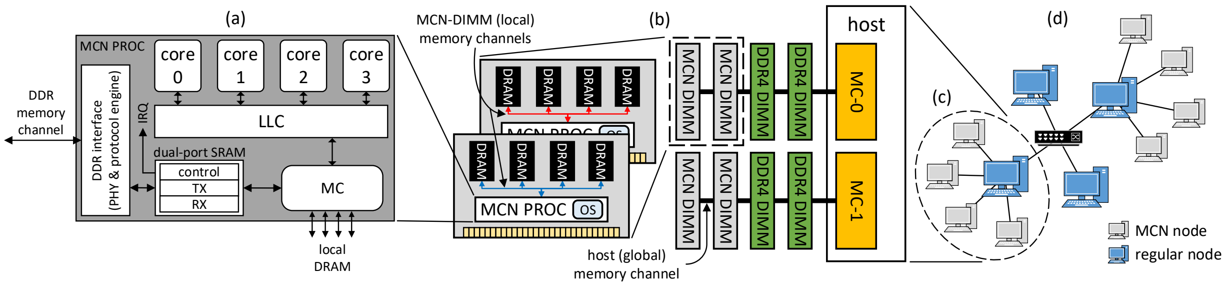


Fig. 3. MCN Overview.

the Snapdragon 835 AP operates at TDP of $5W$ or less [18] and it is implemented on a small ($\sim 8mm \times 8mm$) die in $10nm$ technology [27]. Lastly, if the power constraint of DIMMs prevents us from taking more capable processors such as Tegra® SoC [28] for MCN DIMMs, then we can bring an external power cable to DIMMs as NVDIMMs [29] do.

Fig. 3(a) depicts the MCN processor architecture which implements a DDR interface and a 96KB SRAM buffer in a typical quad-core mobile processor. A DDR interface consisting of DDR PHY and a protocol engine repeats DRAM C/A and DQ signals from/to a host MC as a typical buffer device does. It also performs two operations that are specific to the MCN. First, upon receiving a memory-write request from a host MC, it retrieves a command, a host physical memory address and 64-byte data from the captured C/A and DQ signals, translates the address to an SRAM address and writes the data to the SRAM. Second, when servicing a memory-read request from a host MC, it performs operations similar to handling a memory-write request except that it reads data from the SRAM and generates DQ signals while following a given DDR protocol. Note that this DDR interface differs from the DDR interface between the MCN MC and DRAM devices on the MCN DIMM; we denote the former as the host DDR interface and the later as the MCN DDR interface. The SRAM serves as a communication buffer between the host and MCN processor, and is exposed to both the host and MCN processor as a part of their respective physical memory space, referred to as host and MCN physical memory space. The DDR interface and the SRAM together operate as an MCN interface similar to a NIC in a conventional node.

Fig. 4 describes three regions of the SRAM buffer. We implement a circular TX buffer using `tx-start` and `tx-end` pointers, pointing to the start of the valid data and end of the valid data, respectively. Based on the area from McPAT in $22nm$ technology, we calculate that the size of this buffer is $0.074mm^2$ in $10nm$ technology. TX and RX circular buffers store MCN messages which are sent to or received from the host processor, respectively. The `tx-poll` and `rx-poll` fields are used for handshaking between the host and MCN processors. We will describe the detailed usage of these control bits and the circular buffers in Sec. III-B. When the OS network layer running on an MCN processor sends a network packet, the MCN driver, which is perceived as a regular Ethernet interface (Sec. III-B), sends the network packet to a specific MCN physical memory address, where the SRAM buffer is mapped. When the MCN MC receives any memory request to the MCN physical memory space corresponding to the SRAM buffer, it re-directs the memory request to the SRAM buffer, which is connected to the MCN MC through an on-chip interconnect, instead of sending it to the DRAM devices on the MCN DIMM.

Lastly, similar to a conventional NIC, we implement a HW interrupt mechanism in the MCN interface to notify the MCN processor of any received packet in a SRAM RX buffer (`IRQ` in Fig. 3(a)). Upon receiving an interrupt from the MCN interface, the MCN processor starts a transfer of the packets from the RX SRAM buffer to the kernel memory space of the MCN driver using `memcpy` function. The memory copy operation can be accelerated using a custom DMA engine (Sec.IV-B).

B. *MCN Drivers*

The MCN drivers run on both the host and the MCN DIMMs to create an illusion of the existence of an Ethernet interface between the host and MCN processors. An MCN driver exposes itself as a regular Ethernet interface to the upper OS network layers, therefore, MCN does not require any changes in the OS network stack. This is a key advantage for MCN as there is a resistance towards the changes in the TCP/IP stack [30], [31].

Network organization. As shown in Fig. 3(b), we can populate a host memory channel with multiple MCN DIMMs (also referred to as MCN nodes). The host-side driver (*i.e.* the driver running on the host processor), creates a virtual Ethernet interface for each MCN node installed on the host memory channels. That is, a virtual point-to-point connection is provided between the host and each MCN node, as shown in Fig. 3(c). We refer to each of the virtual Ethernet interfaces created on the host as a *host-side interface*. We then assign a MAC address, which is a unique 48-bit ID assigned to a network device, to each virtual Ethernet interface. Note that an MCN-side driver (*i.e.* a driver running on an MCN processor) creates one virtual Ethernet interface, as an MCN node only has one point-to-point connection to the host. We refer to a single virtual Ethernet interface created on an MCN node as an *MCN-side interface*.

To facilitate the MCN communication, we assign an IPv4 address [32] to each of the host-side and MCN-side interfaces. From the host point of view, all of the MCN nodes are locally connected. We assign a unique IP addresses to each host-

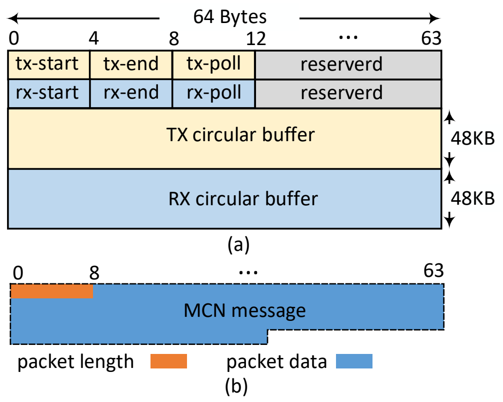


Fig. 4. SRAM buffer of MCN interface.

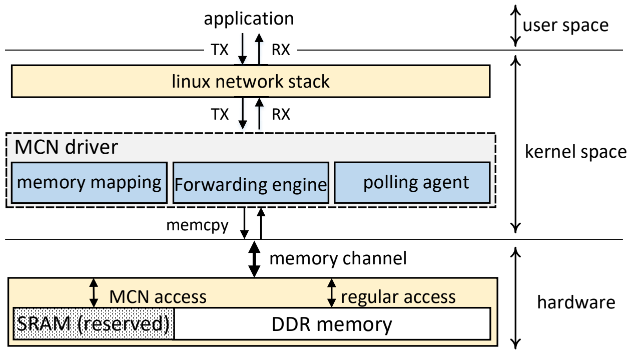


Fig. 5. Overview of MCN driver.

side interface (and the corresponding MCN node) and set the subnet mask of each interface to 255.255.255.255. This means that the host forwards a packet to a host-side interface if and only if the entire destination-IP address of the packet matches with the IP address of the interface. However, an MCN node does not have a direct connection to the other MCN nodes and outside world. Therefore, a packet that is generated from an MCN node and is destined to another MCN node (or outside world), has a different destination-IP address than the host's IP address. To support MCN to MCN and MCN to outside world accesses, we set the subnet mask of the MCN-side interfaces to 0.0.0.0. This means that all the outgoing packets from an MCN node is forwarded to the host, regardless of its IP address. Note that within an MCN node, a packet with its destination-IP set to localhost[2] does not get forwarded to the host as the kernel first checks if a packet belongs to a loopback network interface; if there is no match, then it enumerates the other available interfaces.

This setup ensures that the host arbitrates all the traffic to the MCN nodes, including the traffic between the MCN nodes. This network organization also supports the communication between MCN nodes connected to different hosts by having the source host to forward the packet to the host of the destination MCN node through a conventional NIC.

Driver. Fig. 5 illustrates that the MCN-side driver is composed of three main components: **(C1)** a packet forwarding engine, **(C2)** a memory mapping unit, and **(C3)** a polling agent. Upon initialization, the network driver creates a network device object, sets it up as an Ethernet device, and registers the device with the kernel, thereby making a network interface visible to the host OS. The memory mapping unit accounts for the memory interleaving across different host memory channels and ensures that the physical address space of the SRAM buffers are accessible to the host and MCN processors through the virtual memory. Finally, the polling agent is responsible for periodically polling the SRAM buffers to check for new incoming packets.

Packet transmission and reception. In this subsection we explain the flow of sending a packet from an MCN node and receiving it on the host. Since the host and MCN nodes run symmetric drivers, except for some minor differences, the flow is mirrored for a host to send a packet to an MCN node.

The transmission flow of a packet received at the MCN-side driver from the network stack is as follows (please refer to Fig. 4). (**T1**) Read `tx-start` and `tx-end` from the SRAM buffer. (**T2**) If there is enough space available in the TX buffer, write the packet length followed by the packet data to the TX memory space, starting from the address pointed by `tx-end`. As shown in Fig. 4, we call the combination of a packet length and data an *MCN message*. (**T3**) Update `tx-end` and also set `tx-poll` to a non-zero value, indicating that a new packet is enqueued in the TX buffer. Memory fences are used to ensure that the packet data has been copied correctly, prior to setting the control bits. Nsaote that, if there is not enough space available in the TX buffer, the driver returns `NETDEV_TX_BUSY` as described in $< linux/netdevice.h >$.

Because a conventional DDR interface does not provide a signal that can serve as an interrupt or allow a transaction to be initiated by a DIMM, we propose to adopt a (host-side) polling agent to notify the host processor of incoming packets as a high-speed NIC do. The polling agent periodically reads the `tx-poll` field of the SRAM across all the MCN nodes, checking whether there are any pending packets in any of the MCN nodes. If a pending packet is detected, the host-side driver follows the following steps to receive a packet. (**R1**) Read the `tx-start` and `tx-end` pointers. (**R2**) Read the cache line at `tx-start`. (**R3**) Retrieve the packet length and the packet destination MAC address (`dst-mac`). In an Ethernet packet, the first six bytes of the data construct the destination MAC address [33]. (**R4**) Send the packet to the *packet forwarding engine*. (**R5**) If the `tx-start` pointer moved by the number of bytes read from the TX SRAM buffer is not equal to `tx-end`, it means that there are still available packets in the TX buffer, so start over from `R2`. Otherwise, reset `tx-poll`, and then exit.

Packet forwarding engine. As discussed earlier, the host processor is responsible for routing packets between MCN nodes. When the host receives a packet from either another host or an MCN node, it first inspects the packet to check its destination MAC address (`dst-mac`). Depending on the value of the `dst-mac`, we have one of the following scenar-

[2]In the IPv4 standard, 127.0.0.0 through 127.255.255.255 addresses are reserved for loopback purposes.

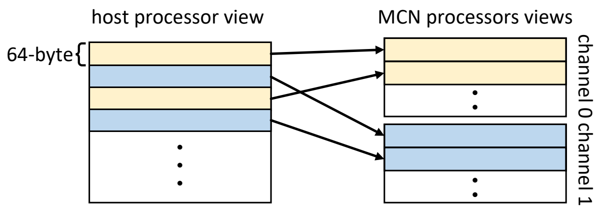


Fig. 6. MCN memory interleaving for two channels.

ios. (**F1**) `dst-mac` matches the MAC address of the host-side interface: Allocate a socket buffer (`sk_buff`), copy the received packet data from the RX SRAM buffer to the `sk_buff`, and send it up to the network stack for processing. (**F2**) `dst-mac` is the reserved address for broadcast: Perform `F1` and `F4` actions. Also transmit the received packet to all the connected MCN nodes, except the source node, by performing `T1`-`T3` steps for each MCN node. (**F3**) `dst-mac` matches the MAC address of one of the MCN-side interfaces: Transmit the received packet to the destination MCN node by performing `T1`-`T3` steps. (**F4**) `dst-mac` does not match the host interface or any MCN-side interfaces: The packet is sent to a conventional NIC interface using `dev_queue_xmit` function in Linux kernel. If a packet is received at an MCN-side interface, MCN always sends the packet up to the network stack for processing by taking the actions explained in `F1`.

Memory mapping unit. By default, `ioremap` (Sec. II-A) creates a page mapping that is tagged as `uncacheable` in the ARM architecture. While this prevents the coherency issues, the maximum size of a memory access to an uncacheable memory space is double word (*i.e.* 64 bits). The memory access size along with the strict memory request ordering limit the memory bandwidth utilization. For the bulk memory transfers needed in MCN, it is desirable to access memory in cache line granularity. This can be done using `memremap` with a `MEMREMAP_WC` flag. This allows the MC's ability to perform a write combining which groups all consecutive write requests into a cache line granularity inside its write queue. On the other hand, read requests to consecutive memory addresses cannot be merged inside the MC read queue as it violates the memory consistency model. Thus, the MCN host-side driver uses an uncacheable memory mapping with the write combining support for the TX buffer and a cacheable memory mapping for the RX buffer. It explicitly invalidates the cache lines in the range of RX buffer after receiving a packet.

While accessing an MCN SRAM buffer, we must be cognizant of the memory channel interleaving performed by the memory subsystem, wherein the successive cache lines in the physical address space are mapped evenly across all the MCs of the host processor. This is to maximize the memory channel parallelism when there is spatial locality between the memory accesses. Without accounting for the memory interleaving, a naïve `memcpy` would incorrectly spread the packet data across MCN DIMMs in different memory channels although it should send them to a particular MCN DIMM's address space. To efficiently tackle this challenge, we propose `memcpy_to_mcn` and `memcpy_from_mcn` functions that perform memory copying such that the 64-byte blocks within the address space of the MCN DIMMs are interleaved in a manner that reflects the memory interleaving of the host processor. This allows the driver to send a packet data to an appropriate memory channel and thus MCN DIMM. Fig. 6 illustrates an example of how `memcpy_to_mcn` and `memcpy_from_mcn` functions map a host processor view of the physical address space to an MCN processor view with two memory channels. As there is an MCN driver assigned to each memory channel and a typical distributed application sends packets to multiple (MCN) nodes, all the memory requests from these MCN drivers still concurrently utilize all the memory channels.

IV. Bandwidth and Latency Optimization

In Sec. III, we described a basic implementation to enable the MCN concept without any changes in the software stack and the host processor architecture. In this section, we identify some inefficiencies in the naïve MCN implementation and exploit some unique properties of a memory channel to further increase the bandwidth and decrease the latency of MCN. Specifically, we first propose to optimize the software stack which does not demand any hardware change. Second, we propose to optimize the memory subsystem architecture if we are permitted to slightly change the host processor architecture as well.

A. *Software-Stack Optimization*

In this section, we first exploit the features in the OS and conventional processors, and propose an efficient polling mechanism to reduce the communication latency between the host and MCN processors. Second, we exploit the fact that the Bit Error Rate (BER) of a memory channel is orders of magnitude lower than that of a network link and propose to bypass the checksum calculation to detect any error in a received packet and adopt a larger frame size for the packets.

Efficient polling mechanism. In Sec. III-B, we proposed a naïve polling approach using a `tasklet`. However, a core running such a polling function can neither sleep nor accept a timer to reschedule. Consequently, it will overwhelm the core by rescheduling itself rapidly. To more efficiently support a polling mechanism, we propose to use a High-Resolution (HR) timer which reschedules a polling function call at a specific time with a nanosecond resolution. Specifically, whenever an HR-timer routine is invoked, it schedules a `tasklet` for running the polling function and then exits. Hence, any function called inside an HR-timer should be very short (*i.e.*, scheduling a `tasklet`). Note that a `tasklet` is interruptible and does not negatively impact a high priority process.

Bypassing checksum. The network stack inspects a Cyclic Redundancy Check (CRC) value or checksum of a packet to detect any error before it delivers the packet to the next network layer. Since the checksum calculation for each packet consumes the host and MCN processors cycles, it often limits the maximum bandwidth and the minimum latency. To reduce such an overhead, the network stack typically supports an interface to offload the checksum calculations to a hardware in the NIC. We propose a much simpler mechanism to efficiently handle checksum calculations. Since a memory channel is protected by an ECC (and CRC in DDR4), we do not need to redundantly generate a checksum value for an MCN message. Therefore, we disable the header checksum checking in the TCP/IP stack without affecting the reliability of the TCP packets.

Large frame size and TCP segmentation offload (TSO). The standard MTU of an Ethernet frame is 1.5KB, as discussed in Sec. II-B. A larger MTU can better amortize the protocol processing software overhead and improve the network performance. Although the network stack can support a larger MTU, it often uses the default size as a larger packet going through the conventional Ethernet links is more likely to be corrupted and incur a higher cost for a re-transmission. However, MCN can efficiently deploy a larger frame size as the BER of a memory channel is typically multiple orders of magnitude lower than that of an Ethernet link. Exploiting such an advantage, we propose to increase the MTU of MCN to 9KB. This can be done by configuring the interface via the Linux `ifconfig` utility. The unique *MCN message* format described in Sec.III-B seamlessly supports any MTU size.

Even with a large MTU size, the network stack may still need to divide a bulk user data chunk into multiple MTU-sized packets. Each of these packets undergoes TCP/IP processing and pays the overhead of segmentation. To optimize bulk data transfer, modern NICs support TCP segmentation offload (TSO) [34], that offloads the segmentation to the NIC hardware. The driver of a TSO enabled NIC provides a TCP/IP header along with a large data chunk to the NIC. The TSO enabled NIC performs the following actions to send the data chunk. (**O1**) Divide the data chunk into several MTU sized segments. (**O2**) Copy the TCP/IP header at the beginning of each data segment. (**O3**) Calculate and set the `Total Length`, `Header Checksum`, and `Sequence Number` fields of each TCP/IP header [32], [35]. (**O4**) Send out each MTU sized packet. The MCN drivers support TSO by ensuring that there is sufficient space in the TX and RX buffers for the largest possible user data chunk allowed by the network stack. Since MCN can also bypass the checksuming, we simply set the `Total Length` field of the TCP/IP header to the user data chunk size and then transmit the unsegmented packet to the destination MCN node.

B. Memory Subsystem Optimization

In this section, we identify two bottlenecks to accomplish a higher bandwidth and lower latency in MCN: the lack of an interrupt mechanism to notify the host processor of the received packets from MCN DIMMs and a memory-to-memory copy accelerator to efficiently transfer the packet data from (to) the host processor to (from) an SRAM buffer in an MCN node. To tackle these limitations, we propose to slightly change the memory subsystem of the host processor as a set of optional optimization.

Supporting interrupt from MCN DIMMs. In Sec. IV-A, we proposed to adopt a high-resolution timer to more efficiently implement the polling agent. However, whenever the HR-timer is called, an interrupt is asserted, which incurs a performance overhead if the polling fails and no packet is received. If the timer interval is increased to minimize the overhead, then the average packet transmission latency increases as well. Additionally, upon receiving an HR-timer interrupt, the driver scans across all the MCN DIMMs on all channels, which further increases the overhead of the polling.

To further reduce the host-side polling overhead, we propose to leverage an existing interrupt-like signal (`ALERT_N` in the DDR4 standard [36]). Specifically, we may re-purpose an `ALERT_N` signal to serve as an interrupt from the MCN-DIMMs installed on a memory channel to the host processor. First, an MC receiving an `ALERT_N` from a memory channel must identify which DIMM on the channel has asserted it. Second, the MC relays the signal to a core as an interrupt. Third, the host-side drivers of the MCN DIMMs installed on the memory channel poll the MCN DIMMs on the channel, as for the polling case. This mechanism not only eliminates the need for periodic polling, but also allows the MCN drivers to immediately know which memory channel it should check.

Memory-to-Memory DMA. The host (MCN) processor is responsible for copying packets between SRAM buffers and the host (MCN) physical memory space with the MCN specific `memcpy` functions. Consequently, the host and MCN processors issuing many memory requests often become a bottleneck, especially when they exchange many packets. This bottleneck can be resolved by implementing MCN DMA engines (`MCN-DMA`) in the memory controller of both the host and MCN processors.

More specifically, we use one DMA engine for each MCN node and one for each memory channel of the host processor. While an MCN side DMA engine maintains only one RX and one TX ring buffers, a host side DMA engine maintains separate ring buffers for each of the MCN nodes installed on the corresponding memory channel. For the packet transmission, the MCN driver initiates the DMA transfer by writing the destination MCN node number (always set to 0 in an MCN-side driver) and the transfer size to the corresponding DMA engine configuration space. The DMA engine is cognizant of the memory channel interleaving and writes a packet from the TX ring to the corresponding MCN node address space. When an MCN node has a new packet, utilizing the DIMM interrupt mechanism explained in the previous subsection, the DMA engine receives an interrupt from the MCN DIMM, reads the available packets from its address space, and populates the RX ring with the received data. When the DMA transfer is finished, the host MCN driver is interrupted. The newly arrived packets in the RX ring are read and routed based on the *packet forwarding engine* explained in Sec. III-B.

V. Methodology

Proof of concept demonstration. As a proof of concept, we developed a prototype MCN system using an experimental buffered DIMM [37] and an IBM POWER8 S824L system shown in Fig. 7(a) and (b), respectively. The prototype MCN

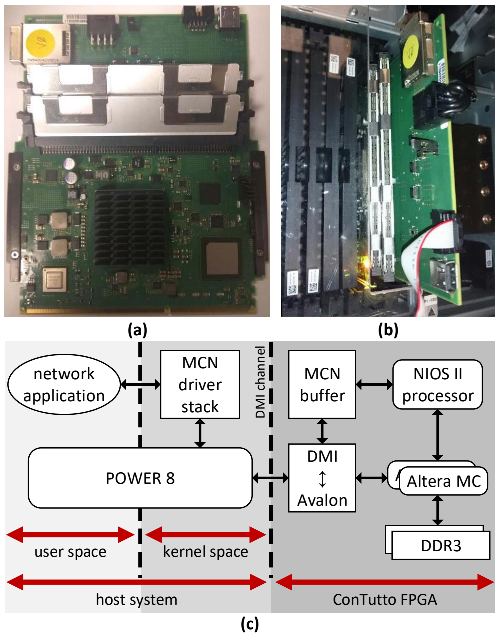


Fig. 7. (a) ConTutto FPGA board (b) plugged into an IBM S824L system alongside regular CDIMMs. (c) MCN implementation block diagram.

DIMM couples two 32GB DDR3-1066 DIMMs with an Intel (Altera) Stratix V FPGA that directly interfaces with the host memory channel, the IBM Differential Memory Interface (DMI). We implemented an MCN DIMM architecture based on a soft IP core, NIOS II embedded processor [38] acting as an MCN processor in the FPGA. We also implemented the MCN SRAM buffer with BRAM blocks, custom glue logic to connect the buffer with DMI/Avalon interface, and used Intel's Avalon [39] as the internal bus in the FPGA. Finally, we developed the MCN drivers for the IBM host processor and the NIOS II processor based on the descriptions in Sec. III. Fig. 7(c) depicts the prototype system architecture.

We use `McPAT` [40] in 22nm technology for power estimation.

Simulator and benchmarks. A NIOS II processor implemented with FPGA and operating at 266MHz has very limited computing capability. Besides, we have only one experimental buffered DIMM. This prevents us from evaluating the full potential of MCN. Thus, we take a full-system simulation approach to further evaluate the effectiveness of MCN. To model a baseline distributed system with multiple nodes connected by 10GbE network, we first take `dist-gem5` [41]–[43] and run the entire software stack in the full-system mode. Second, we implement the MCN DIMM architecture in `dist-gem5` and port the MCN drivers implemented for the prototype system to a simulated processor architecture (*i.e.*, ARMv8 ISA). Table II summarizes the `dist-gem5` system configuration.

TABLE I
DIFFERENT MCN CONFIGURATIONS

mcn0	baseline MCN with HR-timer polling implementation
mcn1	*mcn0* + MCN DIMM interrupt mechanism
mcn2	*mcn1* + IPv4 checksum bypassing
mcn3	*mcn2* + MTU increasing to 9KB
mcn4	*mcn3* + enabling TSO
mcn5	*mcn4* + enabling `MCN-DMA`

TABLE II
SYSTEM CONFIGURATION.

Parameters	Values
Cores (# cores, freq): MCN/Host	(4, 2.45GHz)/(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
MCN Caches (size, assoc): I/D/L2	32KB,2/32KB,2/1MB,16ways
Host Caches: I/D/L2/L3	32KB,2/32KB,2/256KB,16/8MB,16
L1I/L1D/L2 latency,MSHRs	1/2/12 cycles, 2/6/16 MSHRs
DRAM	DDR4-3200MHz/8GB
Operating system	Linux Ubuntu 14.04 (kernel 4.3)
Network	10GbE/1μs link latency

Various optimization levels we used for the evaluation is described in Table I. The same notations are used in Fig. 8. We use `iperf` [44] to compare the achieved bandwidth of MCN with the baseline 10GbE network. We run one `iperf` server and four `iperf` clients that simultaneously communicate with the server. To the collect round-trip latency, we run `ping` with various payload size. We evaluate communication intensive benchmarks from `NAS Parallel Benchmark` (`NPB`) [45], `CORAL` [46], and `BigDataBench` [47] benchmark suites.

VI. Evaluation

A. Network Bandwidth and Latency

Fig. 8(a) shows the achieved `iperf` bandwidth of MCN with different optimization levels, normalized to that of *10GbE*. We show the bandwidth for the following `iperf` setups: *host-mcn* runs the `iperf` server on the host and runs the `iperf` clients on the MCN DIMMs; *mcn-mcn* runs the `iperf` server on an MCN DIMM and runs the `iperf` clients on the host and the remaining MCN DIMMs. Compared with *10GbE*, *mcn0*, which is the basic MCN implementation, improves the bandwidth by 30.3% and 7.8% for the *host-mcn* and *mcn-mcn* configurations, respectively. Replacing polling mechanism with interrupt does not have a notable effect on the achieved `iperf` bandwidth as `iperf` is not compute intensive and the polling agent does not interfere with the `iperf` processes. However, disabling IPv4 checksum calculation, increasing MTU size from 1.5KB to 9KB, enabling TSO, and enabling `MCN-DMA`, each in turn offer extra 3.0%, 99.6%, 31.4%, and 30.6% bandwidth improvements for *host-mcn* configuration.

In general, the achieved bandwidth of *host-mcn* is higher than *mcn-mcn* configuration. The reason is that there is no point-to-point communication channel between two MCN DIMMs and the *mcn-mcn* traffic have to be routed through the host-side MCN driver (Sec. III-B). The achieved bandwidth of *mcn-mcn* is 10.5%, 17.2%, and 20.1% lower than *host-mcn* configuration when employing *mcn3*, *mcn4*, *mcn5* optimization levels, respectively.

Fig. 8(b) illustrates the round-trip latency of a `ping (ICMP)` request, with various payload size, from host to

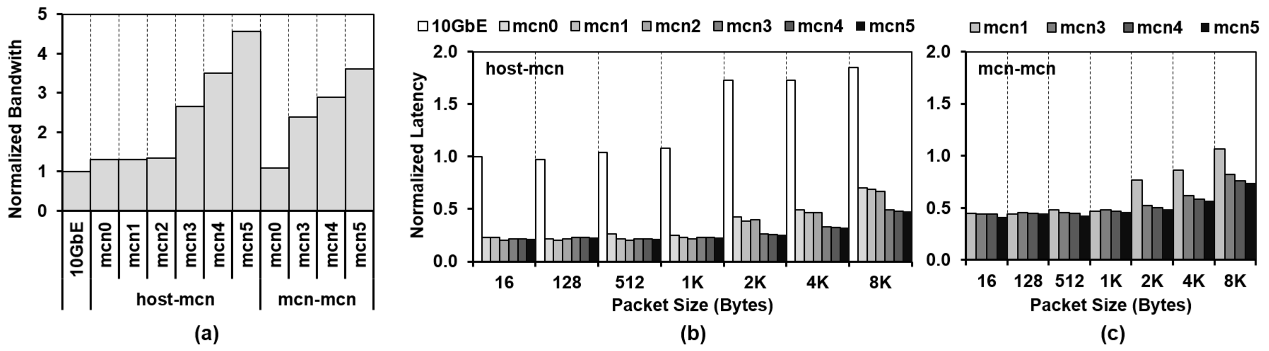


Fig. 8. (a) Network bandwidth for various MCN configurations when running iperf, normalized to 10GbE network, (b) Round-trip latency between host and an MCN node, (c) Round-trip latency between two MCN nodes, both normalized to 10GbE for different MCN configurations across various packet sizes.

an MCN DIMM (*i.e. host-mcn*), normalized to the round-trip latency of a 16-byte `ping` request between two hosts connected with a 10GbE network. As shown, MCN significantly reduces the latency between the nodes. For *host-mcn*, compared with *10GbE*, *mcn0* reduces the round-trip latency by 62.2-75.4% across different packet sizes. Although the *mcn-mcn* communication is less efficient than the *host-mcn* configuration, the optimized MCN implementations always offers lower latency than *10GbE*. As shown in Fig. 8(c), for *mcn-mcn* configuration, *mcn5* reduces the round-trip latency by 52.2-79.0% across different packet sizes, compared with *10GbE*.

Table III reports the latency breakdown of different hardware/software components for *10GbE* and *mcn0* when transmitting and receiving a TCP packet with different size. For *10GbE*, `Driver-TX` includes the time for setting up DMA and notifying NIC about a packet ready for transmission; `DMA-TX`/`DMA-RX` includes the time for transferring a packet from DRAM/NIC to NIC/DRAM; `PHY` includes the time spent in the PCIe bus, the NIC hardware, the Ethernet link, and the Ethernet switch; `Driver-RX` includes interrupting processor, clearing the RX ring buffer, and sending packet up to the network stack for processing. For MCN, `Driver-TX` includes the time that takes to write the packet to the RX buffer. MCN does not have `DMA-TX`, `PHY` and `DMA-RX` components and `Driver-RX` includes the overhead of polling and reading from the RX buffers. We normalize all the latency components to the *10GbE* case. As illustrated in the Table III, removing `PHY` is the biggest contributor to the end-to-end latency reduction for MCN. `Driver-TX` and `Driver-RX` are higher than *10GbE* because now the host/MCN CPU manually copies the packets inside the drivers to/from the SRAM buffer instead of using a DMA engine.

TABLE III
BREAKDOWNS OF THE END-TO-END LATENCIES FOR TRANSMITTING AND RECEIVING A SINGLE TCP 1.5KB/9KB PACKET.

Size	Type	Driver-TX	DMA-TX	PHY	DMA-RX	Driver-RX	Total
1.5KB	10GbE	0.017	0.001	0.479	0.001	0.500	1
	MCN-0	0.075	0	0	0	0.245	0.320
9KB	10GbE	0.019	[0.503 (MEM-to-MEM)]			0.478	1
	MCN-0	0.073	0	0	0	0.692	0.765

B. Performance and Energy

Fig. 9 shows the normalized aggregate memory bandwidth utilization of an MCN-enabled server with different number of MCN DIMMs. We normalize the aggregate bandwidth to the utilized memory bandwidth of the application when running on a conventional server. Across all applications, on average, a server with 2, 4, 6, and 8 MCN DIMMs improves the aggregate memory bandwidth by 1.76×, 2.6×, 3.3×, and 3.9× compared with a conventional server. This shows the effectiveness of MCN in scaling the aggregate memory bandwidth of a server as a near-memory processing framework. Note that adding regular DIMMs to the server just increases the memory capacity and the aggregate memory bandwidth remains unchanged.

Fig. 10 shows the energy efficiency of MCN compared with a conventional scale-out cluster connected through the 10GbE network. A corresponding scale-out system for an MCN server with 2, 4, 6, and 8 MCN DIMMs has 2, 3, 4, and 5 nodes configured with the parameters shown in Table II, respectively; that is to compare the energy consumption of a scale-out system with an MCN server when the total number of cores in both setups is the same. We evenly distribute MCN DIMMs on the host memory channels. Across all the applications, on average, MCN offers 23.5%, 37.7%, 45.5%, and 57.5% lower energy consumption compared with a 2, 3, 4, and 5 node 10GbE scale-out cluster, respectively. Most of the data intensive distributed applications are more energy efficient to run on an MCN server with a mix of high-performance and near-memory mobile processors, compared

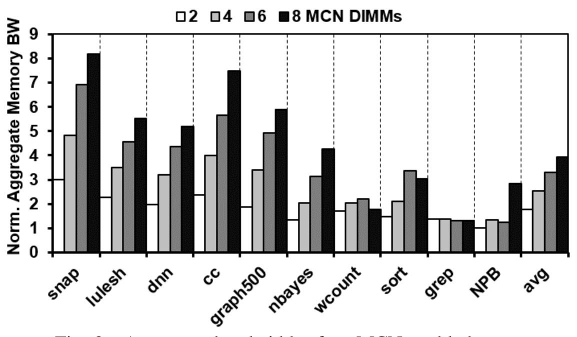


Fig. 9. Aggregate bandwidth of an MCN-enabled server.

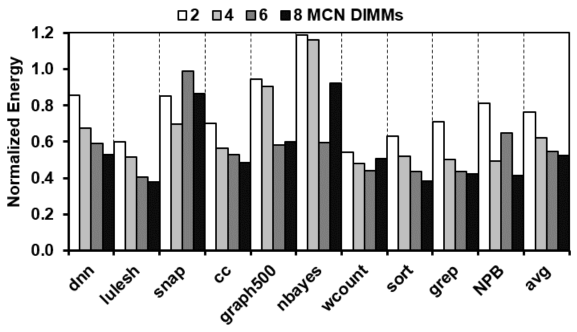


Fig. 10. Energy efficiency of MCN.

with a scale-out counterpart with an identical number of cores [48]. However, not all the benchmarks show energy improvement when running on an MCN server. This shows the importance of supporting the conventional scale-out distributed computing and near-memory acceleration at the same time.

To show the effectiveness of MCN in scaling memory bandwidth, memory capacity, and compute capability of a conventional server all together, we compare the execution time of running `NPB` on a conventional server with running `NPB` on an MCN-enabled server with the same number of cores in both configurations. Fig. 11 shows the execution time of MPI applications from `NPB` running on an scale-up setup, where it has 4, 8, 12, and 16 cores on a single chip, and on an MCN-enabled server, where it has 1, 2, or 3 MCN DIMMs installed. The execution time is normalized to running the application on a conventional server with 4 cores. "0," "1," "2," and "3" on the x-axis represents the baseline server with 4, 8, 12, and 16 cores and an MCN-enabled server with 0, 1, 2, and 3 MCN DIMMs, respectively. As shown in Fig. 11, on average, a server with 1, 2, and 3 MCN DIMMs improves the execution time of `NPB` applications by 27.2%, 42.9%, and 45.3% compared with running them on a scale-up setup with 4, 8, 12, and 16 cores. Note that increasing the number of MPI processes of an application does not always improves the execution time of the application, as for `mg` and `lu`. However, here our focus is on how the higher aggregate memory bandwidth of an MCN-enabled server impacts the execution time of an MPI application.

We noticed that the performance of `ep` (Embarrassingly Parallel) is not sensitive to the memory bandwidth and it only scales with the number of MPI processes. Therefore, MCN does not provide any speedup for `ep`. Also, an scale-out server with 8 cores executes `cg` faster than a server with one MCN DIMM. `cg` performs many irregular communication between MPI processes. As the overhead of inter-process communication in a scale-up server is lower than an MCN server, the speedup of having higher aggregate memory bandwidth is offset by the overhead of frequent MCN to host communication.

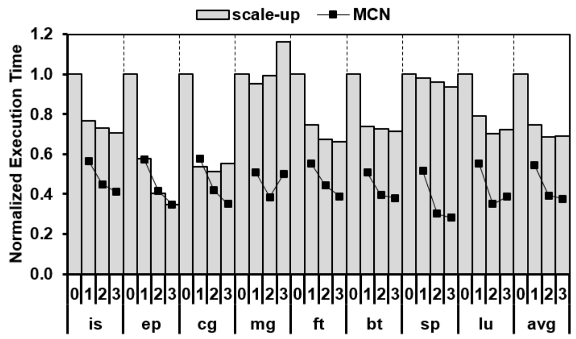
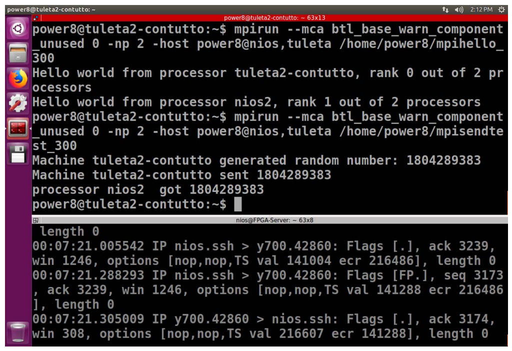


Fig. 11. Normalized `NPB` execution time when running on a conventional scale-up server and on an MCN-enabled server



```
power8@tuleta2-contutto:~$ mpirun --mca btl_base_warn_component
_unused 0 -np 2 -host power8@nios,tuleta /home/power8/mpihello_
300
Hello world from processor tuleta2-contutto, rank 0 out of 2 pr
ocessors
Hello world from processor nios2, rank 1 out of 2 processors
power8@tuleta2-contutto:~$ mpirun --mca btl_base_warn_component
_unused 0 -np 2 -host power8@nios,tuleta /home/power8/mpisendte
st_300
Machine tuleta2-contutto generated random number: 1804289383
Machine tuleta2-contutto sent 1804289383
processor nios2  got 1804289383
power8@tuleta2-contutto:~$
```

```
 length 0
00:07:21.005542 IP nios.ssh > y700.42860: Flags [.], ack 3239,
win 1246, options [nop,nop,TS val 141004 ecr 216486], length 0
00:07:21.288293 IP nios.ssh > y700.42860: Flags [FP.], seq 3173
, ack 3239, win 1246, options [nop,nop,TS val 141288 ecr 216486
], length 0
00:07:21.305009 IP y700.42860 > nios.ssh: Flags [.], ack 3174,
win 308, options [nop,nop,TS val 216607 ecr 141288], length 0
```

Fig. 12. Screenshot of an MPI 'Hello World' program running on our proof-of-concept system.

C. Demonstration with ConTutto Platform

To demonstrate support for existing distributed computing framework APIs, we have cross-compiled the latest version of `OpenMPI` (v3.0.0) for the NIOS II ISA. We successfully tested various MPI applications over MCN by treating the MCN node as a regular worker node with its own IP address in the local area network. The MPI application needs to be compiled separately for the POWER8 and NIOS II processors, but no modification is needed in the application's source code and the execution process is entirely transparent from the programmer's perspective. Fig. 12 shows a screenshot of our prototype system running MPI. The NIOS II terminal is running `tcpdump` at the bottom half of the screen. Note that the purpose of building this proof-of-concept system is to demonstrate that the concept and developed driver work on a commercial system. As the prototype MCN DIMM is built using an FPGA and the MCN processor is a very low-performance soft IP core (NIOS II), we cannot obtain any meaningful performance evaluation.

VII. Discussion

In Sec.VI-A, we demonstrated that MCN is capable of surpassing the performance of a conventional 10GbE network. However, with the given bandwidth and latency of the memory channel, we can potentially improve the MCN bandwidth to surpass higher bandwidth networks too.

A NIC employs several techniques to achieve high bandwidth: (T1) utilizes several offload engines [34], [49], [50]; (T2) uses highly optimized driver and OS software stack (*e.g.* DPDK [51] or mTCP [52]), with special purpose network processing libraries such as RDMA; (T3) distributes the packet processing tasks on several CPU cores; and (T4) uses the aggregate memory bandwidth of a processor by interleaving DMA data across multiple memory channels.

We identified two bottlenecks towards utilizing MCN to its full capabilities. First, the TCP congestion control is implemented for slow, long latency network connections and sometimes takes several seconds to reach to the full bandwidth utilization. Also, TCP frequently sends ACK messages to the sender. Sending and receiving ACK messages consumes both CPU cycles and network bandwidth. Based on our evaluation results, sending and receiving ACK messages incurs up to $\sim$25% overhead in a TCP connection, which is aligned with the previous studies [53]. Second, an MCN DIMM can only use a single channel bandwidth and cannot interleave the memory accesses across multiple memory channels. That being said, the maximum theoretical MCN bandwidth is 12.8GB/s, which is the maximum bandwidth of a single memory channel. Although the bandwidth of each MCN node is limited to the bandwidth of a single memory channel, it is far from being a bottleneck as the bandwidth of a single memory channel alone is more than 100Gbps. Nonetheless, each MCN DIMM can communicate with the host or each other independently, providing aggregate bandwidth proportional to the total number of memory channels in the system.

As a future work, we consider the use of an specialized TCP/IP stack for MCN that resembles a *user space TCP* stack such as `mTCP`. When communicating between MCN DIMMs, the MCN network stack does not rely on the conventional TCP/IP stack and instead resembles a shared memory communication channel between the host and MCN nodes.

The network architecture of the current datacenters follows a hierarchical model with the servers as the leaf nodes. A rack, as the basic building block of a datacenter, consists of several servers connected together using a top of rack switch. As reported in several industry papers, the bandwidth of a top of rack switch ranges from 1 to 10Gbps, while the top of rack switches are connected together through 40 to 100Gbps connections [54], [55]. As shown in Sec. VI-A, even a basic MCN implementation provides higher bandwidth and lower latency than its 10GbE counterpart. We propose to replace a rack with an MCN-enabled server that interconnect leaf nodes (*i.e.* MCN nodes) using a low cost, energy efficient interconnect to improve the energy efficiency of running IO intensive applications (Sec. VI-B) while reducing the datacenter cost.

VIII. Related Work

Near-DRAM processing. The traditional processor-in-memory (PIM) architectures integrate a processor and DRAM onto a single die [13], [56]–[61]. These architectures can reduce energy consumption and increase the throughput of data transfers between the processor and DRAM, but suffer from high fabrication costs and low yields [14], [62]. The integration issue was mitigated by the emerging 3D die-stacking technology, reopening opportunities for near-DRAM processing architectures [2], [6]–[8], [24], [63]–[70]. Among these architectures, NDA [65] 3D-stacks accelerators atop a DRAM device and is similar to MCN because both build on a standard DRAM interface and DIMM architecture. However, NDA requires a programmer to manually handle the communication between the host processor and accelerators using a dedicated programming model for the accelerators. As a cheaper alternative to using 3D die-stacking technology and providing large memory capacity, Chameleon [3] proposes to place accelerators in the buffer devices of DIMMs. It is similar to MCN because accelerators are integrated in a buffer device, but it suffers from the same limitation as NDA. In contrast, MCN is unique because it does not require technology integration, 3D die-stacking, or a new programming model.

Cache coherent interconnect. IBM's Coherent Accelerator Processor Interface (CAPI) [71] is a high speed communication standard, designed for I/O attached accelerators to work in a cache-coherent fashion with traditional processors. As CAPI leverages existing point-to-point PCIe I/O channels, it needs to rely on a customized hardware to manage the coherency and it cannot provide the bandwidth scaling benefit with more accelerator modules like MCN. Besides, CAPI relies on kernel extensions and a CAPI application library to expose the accelerator to the host application, thus requiring the user to modify or rewrite the application in order to leverage the accelerators. Intel Quick Path Interconnect(QPI) [72] supports a cache coherency protocol for attached devices. Although QPI is a high speed point-to-point interconnect, it suffers from the same limitations as PCIe (long latency, limited to one device per channel, etc.). Intel HARP [73] architecture leverages the QPI and couples an FPGA with an Intel processor. Like CAPI, accelerators in HARP have access to the cache coherency mechanisms and unified address space, but leveraging accelerators in HARP also requires using Intel provided APIs and libraries or using OpenCL, which would once again require modifying or rewriting the target application.

IX. Conclusion

In this paper, we propose MCN consisting of MCN DIMMs and MCN drivers. MCN allows us to run applications based on distributed computing frameworks, such as MPI, without any change in the host processor hardware, distributed computing middleware and application software, while offering the benefits of high-bandwidth/low-latency communication between host and MCN processors. Furthermore, MCN can serve as an application-transparent near-DRAM processing platform since the memory bandwidth for processing multiplies with the number of MCN DIMMs. As such, MCN can unify the near-DRAM processing in a node with the distributed computing across multiple nodes. Our evaluation shows that a node with MCN can provide up to 58.7% higher performance than multiple conventional nodes connected by a 10GbE network when running various MPI-based distributed applications. Lastly, we demonstrate a proof of MCN concept with an IBM POWER8 system and an experimental buffered DIMM.

X. Acknowledgement

This work is supported in part by grants from NSF (CNS-1557244 and CNS-1705047) and IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR). Nam Sung Kim and Daehoon Kim are the co-corresponding authors.

References

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

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2015.

[3] H. Asghari-Moghaddam, Y. H. Son, J. Ahn, and N. S. Kim, “Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems,” in *MICRO*, 2016.

[4] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow, J. H. Ahn, and N. S. Kim, “Near-DRAM acceleration with single-ISA heterogeneous processing in standard memory modules,” *IEEE Micro*, vol. 36, 2016.

[5] M. Gao, G. Ayers, and C. Kozyrakis, “Practical Near-Data Processing for In-Memory Analytics Frameworks,” in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015.

[6] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, “TOP-PIM: throughput-oriented programmable processing in memory,” in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.

[7] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture,” in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015.

[8] S. Lloyd and M. Gokhale, “Near memory key/value lookup acceleration,” in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017.

[9] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge, “Integrated 3D-stacked server designs for increasing physical density of key-value stores,” in *19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014*, 2014.

[10] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017.

[11] Y. K. Rupesh, P. Behnam, G. R. Pandla, M. Miryala, and M. N. Bojnordi, “Accelerating k-medians clustering using a novel 4t-4r rram cell,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[12] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos *et al.*, “Active memory cube: A processing-in-memory architecture for exascale systems,” *IBM Journal of Research and Development*, vol. 59, 2015.

[13] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, “FlexRAM: Toward an Advanced Intelligent Memory System,” in *ICCD*, Oct 1999.

[14] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, “Intelligent RAM (IRAM): the industrial setting, applications, and architectures,” in *ICCD*, Oct 1997.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’10. Berkeley, CA, USA: USENIX Association, 2010.

[17] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, “Users Guide to MPICH, a Portable Implementation of MPI,” *Argonne National Laboratory*, vol. 9700, 1995.

[18] Qualcomm, “Snapdragon 835 Mobile Platform,” 2016. [Online]. Available: https://www.qualcomm.com/products/snapdragon/processors/835

[19] JEDEC Standard, “DDR4 SDRAM Registered DIMM Design Specification,” *JESD21-C*, 2016.

[20] JEDEC Standard, “DDR4 SDRAM Load Reduced DIMM Design Specification,” *JESD21-C*, 2016.

[21] P. J. Meaney, L. D. Curley, G. D. Gilda, M. R. Hodges, D. J. Buerkle, R. D. Siegl, and R. K. Dong, “The IBM z13 memory subsystem for big data,” *IBM Journal of Research and Development*, vol. 59, July 2015.

[22] B. Jacob and T. Mudge, “Virtual memory in contemporary microprocessors,” *IEEE Micro*, vol. 18, 1998.

[23] E. Dumazet, “Busy polling: Past, present, future,” *NetDev 2.1*, 2017.

[24] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads,” *Micro, IEEE*, vol. 34, Jul 2014.

[25] T. R. Halfhill, “POWER8 Hits the Merchant Market: Memory Bandwidth Helps IBM Server Processor Ace Big Benchmarks,” https://www-03.ibm.com/systems/power/advantages/smartpaper/memory-bandwidth.html.

[26] The Tech Report, “The Exynos 5433 SoC,” http://techreport.com/review/27539/samsung-galaxy-note-4-with-the-exynos-5433-processor/2.

[27] A. Wei, “Qualcomm Snapdragon 835 First to 10 nm,” http://www.techinsights.com/about-techinsights/overview/blog/qualcomm-snapdragon-835-first-to-10-nm/, 2017.

[28] Nvidia®, “NVIDIA®Tegra®X1,” https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, 2015.

[29] Micron, “NVDIMM,” 2016. [Online]. Available: https://www.micron.com/products/dram-modules/nvdimm/

[30] J. S. Turner and D. E. Taylor, “Diversifying the Internet,” in *Global Telecommunications Conference, 2005. GLOBECOM’05. IEEE*, vol. 2. IEEE, 2005.

[31] T. Anderson, L. Peterson, S. Shenker, and J. Turner, “Overcoming the Internet impasse through virtualization,” *Computer*, vol. 38, 2005.

[32] “IPv4 standard,” https://en.wikipedia.org/wiki/IPv4, accessed: 2018-03-25.

[33] “Ethernet frame,” https://en.wikipedia.org/wiki/Ethernet_frame, accessed: 2018-03-25.

[34] G. W. Connery, W. P. Sherer, G. Jaszewski, and J. S. Binder, “Offload of TCP segmentation to a smart adapter,” Aug. 10 1999, uS Patent 5,937,169.

[35] “TCP frame,” https://en.wikipedia.org/wiki/Transmission_Control_Protocol, accessed: 2018-03-25.

[36] “JEDEC Standard: DDR4 SDRAM,” 2012.

[37] B. Sukhwani, T. Roewer, C. L. Haymes, K.-H. Kim, A. J. McPadden, D. M. Dreps, D. Sanner, J. Van Lunteren, and S. Asaad, “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*, ser. MICRO-50 ’17. New York, NY, USA: ACM, 2017.

[38] Intel, “Nios II Processor,” https://www.altera.com/products/processors/overview.html, 2017.

[39] Intel, “Avalon®Interface Specifications,” https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf, 2017.

[40] S. Li, J. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures,” in *MICRO*, 2009.

[41] A. Mohammad, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, “dist-gem5: Distributed simulation of computer clusters,” in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE, 2017.

[42] M. Alian, D. Kim, and N. S. Kim, “pd-gem5: Simulation infrastructure for parallel/distributed computer systems,” *IEEE Computer Architecture Letters*, vol. 15, 2016.

[43] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[44] “Iperf: The ultimate speed test tool for TCP, UDP and SCTP.” [Online]. Available: https://iperf.fr/

[45] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, “The NAS parallel benchmarks,” *The International Journal of Supercomputing Applications*, vol. 5, 1991.

[46] “Coral benchmark codes.” [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[47] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, “Bigdatabench: A big data benchmark suite from internet services,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014.

[48] R. Azimi, T. Fox, and S. Reda, “Understanding the Role of GPGPU-Accelerated SoC-Based ARM Clusters,” in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017.

[49] L. Grossman, “Large Receive Offload implementation in Neterion 10GbE Ethernet driver,” in *Linux Symposium*, 2005.

[50] C. B. Melzer, J. Rosen, R. O’gorman, P. A. Wood, M. C. Drummond, and D. Hiller, “IP checksum offload,” Apr. 27 1999, uS Patent 5,898,713.

[51] “Data Plane Development Kit,” http://dpdk.org/.

[52] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, “mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems,” in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’14. Berkeley, CA, USA: USENIX Association, 2014.

[53] M. Chan and D. R. Cheriton, “Improving server application performance via pure TCP ACK receive optimization,” in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/chan

[54] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hlzle, S. Stuart, and A. Vahdat, “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Googles Datacenter Network,” in *Sigcomm ’15*, 2015.

[55] A. Andreyev, “Introducing data center fabric, the next-generation Facebook data center network,” 2014. [Online]. Available: https://code.facebook.com/posts/360346274145943/

[56] M. F. Deering, S. A. Schlapp, and M. G. Lavelle, “FBRAM: a New Form of Memory Optimized for 3D Graphics,” in *SIGGRAPH*, Jul 1994.

[57] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, “The Architecture of the DIVA Processing-in-memory Chip,” in *ICS*, Jun 2002.

[58] D. G. Elliott, W. M. Snelgrove, and M. Stumm, “Computational RAM: A Memory-SIMD Hybrid and its Application to DSP,” in *CICC*, May 1992.

[59] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, “Smart Memories: A Modular Reconfigurable Architecture,” in *ISCA*, Jun 2000.

[60] M. Oskin, F. Chong, and T. Sherwood, “Active Pages: A Computation Model for Intelligent Memory,” in *ISCA*, Jun 1998.

[61] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A Case for Intelligent RAM,” *Micro, IEEE*, vol. 17, Mar 1997.

[62] M. L. Chu, N. Jayasena, D. P. Jang, and M. Ignatowski, “High-level Programming Model Abstractions for Processing in Memory,” in *Workshop on Near-Data Processing*, Dec 2013.

[63] G. H. Loh, “3D-Stacked Memory Architectures for Multi-core Processors,” in *ISCA*, Jun 2008.

[64] J. T. Pawlowski, “Hybrid Memory Cube,” in *Hot Chips*, Aug 2011.

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

[66] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, “A 3D-stacked Logic-in-memory Accelerator for Application-Specific Data Intensive Computing,” in *3DIC*, Oct 2013.

[67] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “Graphpim: Enabling instruction-level pim offloading in graph computing frameworks,” in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017.

[68] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, “NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads,” in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014.

[69] A. J. Awan, M. Ohara, E. Ayguadé, K. Ishizaki, M. Brorsson, and V. Vlassov, “Identifying the potential of near data processing for apache spark,” in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017.

[70] C. D. Kersey, H. Kim, and S. Yalamanchili, “Lightweight SIMT core designs for intelligent 3D stacked DRAM,” in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017.

[71] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, “CAPI: A Coherent Accelerator Processor Interface,” *IBM Journal of Research and Development*, vol. 59, Jan 2015.

[72] Intel®, “An Introduction to the Intel®QuickPath Interconnect, Document Number: 320412-001US, January, 2009,” https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[73] Intel®, “IvyTown Xeon + FPGA: The HARP Program, Workshop: CPU+FPGA - OpenCL Based High Level Synthesis for CPU+FPGA Coherent Systems, International Symposium on Computer Architecture, ISCA, 2016,” https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf.