# Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube

Ramyad Hadidi, Bahar Asgari, Jeffrey Young, Burhan Ahmad Mudassar, Kartikay Garg,
Tushar Krishna, and Hyesoon Kim
Georgia Institute of Technology
{rhadidi, bahar.asgari, jyoung9, burhan.mudassar, kgarg40}@gatech.edu
tushar@ece.gatech.edu hyesoon@cc.gatech.edu

***Abstract*—Three-dimensional (3D)-stacked memories, such as the Hybrid Memory Cube (HMC), provide a promising solution for overcoming the bandwidth wall between processors and memory by integrating memory and logic dies in a single stack. Such memories also utilize a network-on-chip (NoC) to connect their internal structural elements and to enable scalability. This novel usage of NoCs enables numerous benefits such as high bandwidth and memory-level parallelism and creates future possibilities for efficient processing-in-memory techniques. However, the implications of such NoC integration on the performance characteristics of 3D-stacked memories in terms of memory access latency and bandwidth have not been fully explored. This paper addresses this knowledge gap (i) by characterizing an HMC prototype using Micron's AC-510 accelerator board and by revealing its access latency and bandwidth behaviors; and (ii) by investigating the implications of such behaviors on system- and software-level designs. Compared to traditional DDR-based memories, our examinations reveal the performance impacts of NoCs for current and future 3D-stacked memories and demonstrate how the packet-based protocol, internal queuing characteristics, traffic conditions, and other unique features of the HMC affects the performance of applications.**

## I. Introduction

In the past decade, the demand of data-intensive applications for high-performance memories has pushed academia and industry to develop novel memories with larger capacity, higher access bandwidth, and lower latency. To this end, JEDEC-based memories (i.e., DDRx) have evolved to include three-dimensional (3D)-stacked DRAMs, such as High Bandwidth Memory (HBM) [1]. While such memories are compatible with traditional architectures and JEDEC standards, they are limited in terms of scalability and bandwidth, which is due to their wide buses and the use of the standard DDRx protocol. Therefore, a generation of 3D-stacked memories with packet-based communication has been introduced and is currently implemented in the Hybrid Memory Cube, or HMC [2]. Thanks in part to an internal packet-switched network and high-speed serial links between the processor and memory stack, this type of novel 3D-stacked memory exploits both internal and external networks to extend its capacity and scalability [3], [4]. The HMC consists of vertical memory partitions called vaults and a logic layer that consists of memory controllers (i.e., vault controllers), connected via an internal network-on-chip (NoC) [5]. As our analysis shows, the characteristics and contention of the internal NoC play an integral role in the overall performance of the HMC.

Logic and memory integration within 3D stacks has motivated researchers to explore novel processing-in-memory (PIM) concepts within the architecture of 3D-stacked memories using simulation [4], [6]–[15]. However, few researchers have studied actual prototypes of memories similar to the HMC [16]–[19]. In particular, to the best of our knowledge, no experimental work has sought to characterize the bandwidth and latency[1] impacts of the internal NoC on the performance of the HMC. In addition to understanding the performance impacts of the NoC on applications, such characterizations are also important for the design of PIM units built around or inside the HMC. To gain further insights into the impacts of the internal NoC on 3D-stacked memories, we evaluate the performance characteristics of an HMC 1.1 [5] prototype. We utilize a Xilinx FPGA and an HMC 1.1 on the Micron's AC-510 [20] accelerator board, which is mounted on an EX-700 [21] PCIe backplane. Figure 1 presents the full-stack overview of our FPGA-based evaluation system, which includes user configurations, memory trace files, software, driver, an FPGA, and an HMC.

Our analyses characterize access properties for both low- and high-contention traffic conditions, for which we use two combinations of software and digital designs (i.e., Verilog implementations on the FPGA). Our results reveal (i) latency and bandwidth relationships across various access patterns, targeted to structural organizations of the HMC (i.e., vaults and banks), (ii) latency distributions across the vaults of the HMC,

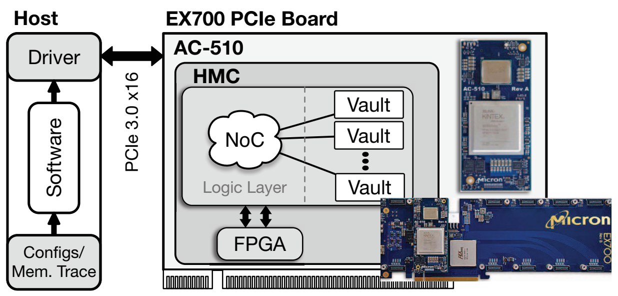

**Fig. 1:** An overview of our system, and the NoC of the HMC.

[1] Latency and round-trip time are interchangeably used in this paper.

(iii) quality of service (QoS) within a particular access pattern, and (iv) bottlenecks to occur within the HMC, associated infrastructure, or within each access pattern. The contributions of this paper are as follows:

- This is the first study, to the best of our knowledge, that explores the impacts of the internal NoC of the HMC, a prototype of packet-switched 3D-stacked memories, on bandwidth and latency.
- It examines how the internal NoC behaves under low- and high-contention traffic conditions, presents the concept of QoS for 3D-stacked memories, and describes how future systems and applications should incorporate the HMC to achieve desirable performance.
- It presents a detailed analysis of the latency distribution that is caused by the internal NoC of the HMC for a specific access pattern and related consequences and opportunities.
- It studies request and response bandwidth relationships for various access patterns, determines the source of bottlenecks, and presents solutions for avoiding them.

In the rest of this paper, we first review the HMC 1.1 specifications in Section II and then introduce our infrastructure and methodology in Section III. After that, Section IV presents and analyzes the details of latency and bandwidth of the HMC with various traffic conditions and the contribution of the NoC in each scenario. Subsequently, Sections V and VI review related work and present conclusions based on our analyses.

## II. Background

In this paper, we focus on the HMC 1.1 specification (*Gen2*) [5], currently available for purchase. This section presents background on the structure of the HMC and relevant information on packet-based memories for our analyses.

### A. HMC Structure

The HMC 1.1 consists of eight DRAM dies stacked on top of a logic die, vertically connected by 512 *Through-Silicon-Vias (TSVs)* [2]. As Figure 2 illustrates, the layers of the HMC are divided into 16 partitions, each of which is called a *vault* with a corresponding memory controller in the logic layer, the so-called *vault controller* [22]. Each vault employs a 32-byte DRAM data bus [5], enabled by 32 TSVs. A group of

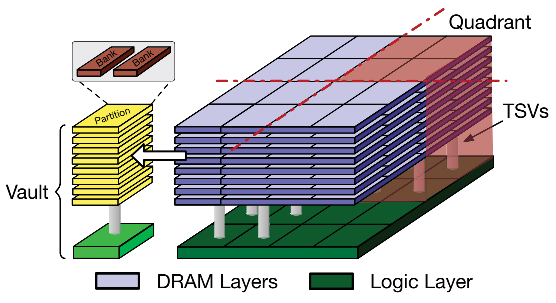

**Fig. 2:** 4 GB HMC 1.1 internal structure.

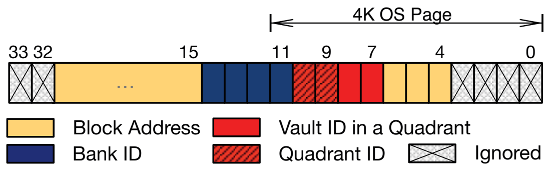

**Fig. 3:** Address mapping of 4 GB HMC 1.1 with block size of 128 B.

four vaults is called a *quadrant*, connected to an external full-duplex serialized link, an eight- (half-width) or a 16-lane (full-width) connection clocking at speeds of 10, 12.5, or 15 Gbps. Thus, the maximum bandwidth of a two-link half-width HMC device with a 15 Gbps link is:

$$\mathrm{BW}_{\mathrm{peak}} = 2\,\mathrm{link} \times 8\,{}^{\mathrm{lanes}}/_{\mathrm{link}} \times 15\,\mathrm{Gb/s} \times 2\,\mathrm{duplex} = 60\,\mathrm{GB/s}. \quad (1)$$

The size of a DRAM layer in Gen2 (HMC 1.1) devices is 4 Gb. Since HMC 1.1 has eight layers, the total size of it is 4 GB. Moreover, each of the 16 vaults is 256 MB. As the size of a bank is 16 MB [5], the number of banks in a vault and an HMC 1.1 is 16 and 256, respectively. (A detailed comparison between versions of the HMC is done in [19].)

The header of an HMC 1.1 request packet (see Section II-B for more details) contains a 34-bit address field, but two high-order bits are ignored in a 4 GB HMC. Figure 3 shows the internal address mapping of HMC 1.1 for 128 B block size configuration [5], as well as the *low-order-interleaving* mapping of sequential blocks to vaults and then to banks within a vault. For a block size of 128 B, an OS page, usually 4 KB, would be mapped to two banks over all 16 vaults, so that accesses to a page utilize high bank-level parallelism (BLP). The vault controllers, each controlling a vault that contains a part of a page, are connected using an internal NoC (i.e., each external link can carry packets destined to any vault), the characteristics of which impacts the overall bandwidth and latency of a system.

### B. Packet-Based Memories

Unlike memories with JEDEC-based bus interfaces (e.g., GDDR or HBM), HMC uses a packet-based interface to transfer packets over data links. Packet-based memories exploit internal and external NoCs for scalability; vaults in an HMC are connected internally and up to eight HMCs can be connected via external links. As the HMC interface uses high-speed serialization/deserialization (*SerDes*) circuits, these networked implementations achieve higher raw link bandwidths than traditional, synchronous, bus-based interfaces. Unlike traditional memories, the access latency of a packet-based memory includes additional time for packet processing,

**TABLE I:** HMC request/response read/write sizes [5].

| Type | Request Read | Request Write | Response Read | Response Write |
|---|---|---|---|---|
| Data Size | Empty | 1∼8 Flits | 1∼8 Flits | Empty |
| Overhead | 1 Flit | 1 Flit | 1 Flit | 1 Flit |
| Total Size | 1 Flit | 2∼9 Flits | 2∼9 Flits | 1 Flit |

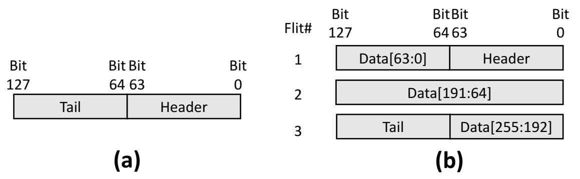

**Fig. 4:** (a) A flow packet (no data), and (b) a request/response packet with 32 B of data.

such as packet creation, port arbitration, flow control and serialization/deserialization [5]. These overheads are amortized by using large numbers of queues and ports (up to nine in our infrastructure) for sending/receiving packets, high BLP, and high-speed transmission to and from the HMC.

Similar to IP-based networks, the communication of the HMC is layered, which includes physical, link, and transaction layers. The physical layer is responsible for serialization, deserialization, and transmission while the link layer handles low-level communication and flow control for packets over high-speed physical connections. The transaction layer defines request and response packets, their fields, and controls high-level flow and retry. The HMC controller uses three types of packets: flow, request, and response packets. Flow packets do not contain a data payload (Figure 4a) while request and response packets are used for performing data reads and writes from and to the HMC (Figure 4b). 16-byte elements that construct packets are called *flits*, and the size of the data payload of each packet varies from one to eight flits. The least-significant flit of packets is transmitted first across the link. Flow control and integrity check of packets are performed using dedicated fields in the one-flit head and tail [5]. Accordingly, Table I shows each HMC transaction size in flits.

## III. Methodology

This section introduces our infrastructure for evaluating the HMC 1.1 and includes details on its hardware, firmware (i.e., the digital design on the FPGA), and software.

### A. Infrastructure

We utilize a Pico SC-6 Mini [23] machine that incorporates an EX-700 [21] backplane, a PCIe 3.0 x16 board with 32 GB/s bandwidth to the host. The EX-700 backplane can accommodate up to six AC-510 [20] accelerator modules, each of which contains a Kintex Xilinx FPGA[2] and a 4 GB HMC 1.1 (similar to Figure 2). We utilize one AC-510 in our evaluations. The HMC and the FPGA on an AC-510 module are connected with two half-width (8 lanes) links operating at 15 Gbps, so the bi-directional peak bandwidth is 60 GB/s, using Equation 1.

### B. Firmware and Software

We use two combinations of firmware and software to perform experiments, *GUPS* and *multi-port stream* implementations, shown in Figure 5. Each combination integrates a custom logic on the FPGA and a software counterpart. First, we describe the common components in the firmware on the FPGA. The FPGA uses Micron's HMC controller [24] to generate packets for the multi-port AXI-4 interface between the FPGA and the HMC. On the software side, the Pico API [25] and device driver are used for initializing the logic on the FPGA and provide an environment, in which an OS communicates with the FPGA. The Pico API provides software methods to access the HMC through the FPGA with a direct path for sending/receiving packets. However, because the software runs at a lower rate on the host than on the FPGA, this solution cannot fully utilize the bandwidth of the HMC. Furthermore, since the maximum frequency of the FPGA is low (187.5 MHz), to generate more requests, the FPGA uses nine copies of the same module, called *ports*. For measuring various statistics such as the total number of read and write requests and the total, minimum, and maximum of read latencies, each port contains monitoring logic that is not in the critical path of accesses. Note that this monitoring logic measures aggregate latencies of the HMC controller, transceiver, data transmission on links, internal NoC, TSV transmission, and DRAM timings. Detailed studies of these latencies are performed by Hadidi et al. [19], upon which we build our new measurements.

To observe the behavior of the NoC within the HMC with various traffic patterns and contention levels, we utilize two implementations as follows: (i) GUPS (Figure 5a), a vendor-provided firmware that measures how frequently we can generate requests to random memory locations; and (ii) multi-port stream implementation (Figure 5b), a custom firmware which generates requests from memory trace files using Xilinx's AXI-Stream interface.

The *GUPS implementation* is best suited to investigate the behavior of NoC under high contention while the multi-port stream implementation performs the same task from a trace file per port. For both implementations, the number of active ports and their access patterns are configured independently. With GUPS, each port has a configurable address generation unit that is able to send read-only, write-only, or read-modify-write requests for *random* or *linear* mode of addressing. By forcing some bits of the generated addresses to zero/one by using an address *mask/anti-mask*, a group of randomly generated requests, each corresponding to a single response packet from one bank, are mapped to a specific part of the HMC to create all possible access patterns (i.e., from accessing a single bank within a vault, to accessing all banks of all the vaults). To perform experiments, for each port, we first set the type of requests and their sizes, their mask and anti-mask, and next, we activate the port. While the port is active, it generates as many requests as possible for 10 seconds, and then it reports the total number of accesses (read and write), maximum/minimum of read latencies, and aggregate read latency back to the host. In this paper, the type of requests is read only, unless stated otherwise. Our current firmware implementations do not support ACKs after writes, so accurate measurements of write latency would only be possible with added monitoring logic specifically for writes. We plan to address this limitation in future work. However, since we are studying the internal NoC

[2] Part#: xcku060-ffva1156-2-e

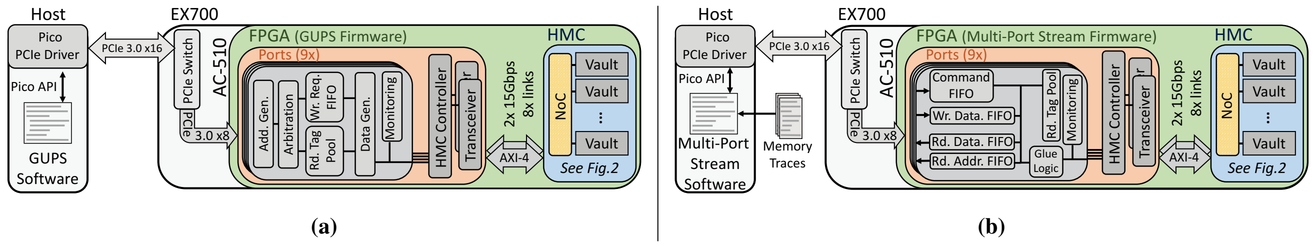

**Fig. 5:** Firmware and software overview: (a) GUPS and (b) multi-port stream implementations.

of the HMC, any type of requests that consume resources will reveal the behaviors, bottlenecks, and impacts of the NoC by limiting the flow of packets to the HMC.

The *multi-port stream implementation* employs a multi-threaded software that reads a memory trace file for each port and populates buffers on the host. Then, by using Xilinx's AXI-Stream interface to each port (wrapped in a PicoStream API call [25]), we efficiently transmit commands such as access types, sizes, and data through their dedicated communication channel. After issuing requests and waiting for responses, each port transmits read data and their addresses back to the host. In fact, the FPGA reads data continuously, such that each port reads data from its dedicated channel in every cycle. In both GUPS and multi-port stream implementations, we calculate the average access latency of reads by dividing the aggregate read latency by the total number of reads. We calculate bandwidth by multiplying the number of accesses by the cumulative size of request and response packets including header, tail and data payloads (shown in Table I), and by dividing it by the elapsed time.

## IV. Results

This section presents various detailed latency and bandwidth analyses by utilizing various traffic conditions and access patterns with GUPS and multi-port stream implementations.

### A. High-Contention Latency Analysis

To achieve a broad perspective of the HMC properties, we perform experiments that access various structural organizations in the HMC, such as vaults and banks. Figure 6 illustrates the latency and bandwidth relationship for read-only accesses. The lowest bandwidth for undistributed accesses (i.e., accesses targeted to a bank) is 2 GB/s for 32 B requests, and the highest bandwidth for the most distributed accesses (i.e., accessing more than or equal to two vaults) is 23 GB/s for 128 B requests. Note that, as Table I shows, read-only requests mostly utilize response bandwidth, which has a cap of 30 GB/s. Accesses to more than two vaults have a similar bandwidth, caused by the limitation of the external link bandwidth between the HMC and the FPGA. Moreover, accesses distributed over eight banks, but within one vault, have the same 10 GB/s bandwidth, limited by the maximum internal bandwidth of a vault [26]. Figure 6 also shows that as the accesses become less distributed, the latency of accesses increases. As the figure depicts, access latency varies from 24,233 ns for 128 B requests targeting a single bank, to 1,966 ns for 16 B requests spread across more than two vaults. Less distributed access patterns (e.g., one bank) have higher latency because they benefit less from BLP. Furthermore, the latency of small requests is always lower than that of large requests because (i) the granularity of the DRAM bus within each vault is 32 B [5], so data payloads larger than 32 B is split; and (ii) larger request packets constitute more flits, so buffering and reordering of packets cause higher latencies.

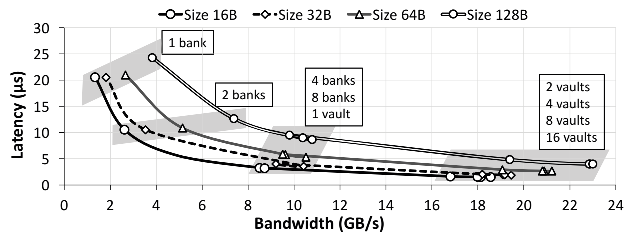

**Fig. 6:** The relationship between latency and bi-directional bandwidth for various access patterns and request sizes for read-only requests.

Figure 6 illustrates that large requests (e.g., 128 B) always have higher bandwidth utilization than small requests (e.g., 32 B) do. This is because (i) large packets utilize bandwidth more effectively (i.e., less overhead), and (ii) small requests quickly consume the maximum number of tags of outstanding requests to the HMC. As Table I presents, each packet, regardless of its data size, always has an overhead of one flit (i.e., 16 B). For this reason, the bandwidth efficiency of read responses with 16 B and 128 B data sizes are $^{16}/_{16+16} = 50\%$ and $^{128}/_{128+16} = 89\%$, respectively. Moreover, for retransmission of a packet (because of transmission failure, flow control, or CRC failure), each port must track outstanding requests, so, at a time, each port can handle a limited number of outstanding requests. Small requests, compared to large requests, underutilize this limited number of slots for transmitting smaller data, which results in low bandwidth utilization. In summary, large packet sizes utilize available bandwidth more effectively at the cost of added latency. In addition, for reducing access latency, accesses should be carefully distributed to exploit BLP and avoid bottlenecks.

### B. Low-Contention Latency Analysis

To examine low-contention latencies, we measure the access latency of the HMC while limiting the number of random read requests to be mapped within the 16 banks of a vault. Then, for each number of read requests, we report average latency

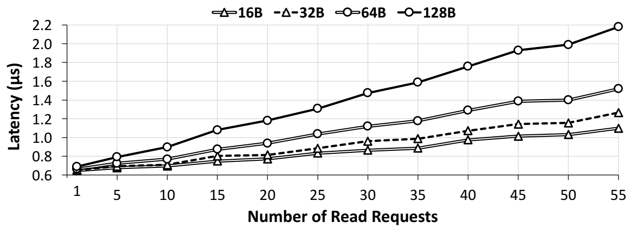

**Fig. 7:** The average latency of low-load accesses for various request sizes for the number of requests in the range of one to 55.

across all vaults. To tune the number of accesses and the size of request packets, we use the multi-port stream implementation. Figure 7 depicts that as the number of requests in a stream (stream in this context means a limited number of requests) increases from one to 55, the average latency increases from 0.7 to 1.1 $\mu$s for the request size of 16 B, and from 0.7 to 2.2 $\mu$s for the request size of 128 B. In other words, we observe two behaviors: (i) When the number of request packets is small, the size of request packet does not affect the latency; and (ii) when the size of request packets is larger, the requests experiences more variations in the latency. Since the flow control unit in the infrastructure is only activated with a large number of outstanding requests, we are certain that, as reported in [19], approximately 547 ns of all latencies for the small number of requests are spent on the FPGA and data transmission. Hence, the contributing latency of the HMC under low load (i.e., no load) is 100 to 180 ns, which includes the latency of DRAM timings ($t_{RCD} + t_{CL} + t_{RP}$ is around 41 ns for HMC [4], [26]), TSV transmission, vault controller, and internal NoC. However, as the number of requests increases, with the similar BLP, queuing delay in both the HMC (i.e., internal NoC and vault controllers) and the FPGA increases, which results in an order of magnitude higher delays. Note that since the HMC utilizes a packet-switched interface to vault controllers in its logic layer, the observed average latency of the HMC is higher than that of traditional DDRx memories.

Figure 8 illustrates a wider range for the number of read requests in a stream than what Figure 7 shows. In Figure 8, we observe that when the number of requests increases up to 100, average access latency increases linearly. After that, the latency stays approximately constant when the number of requests grows. By assuming a hypothetical queue for requests, we infer that until the time that the queue is not full, the latency of each request equals to its serving time plus its

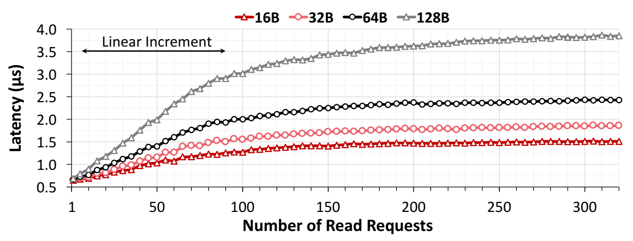

**Fig. 8:** The average latency of low-load accesses for various request sizes for the number of requests in the range of one to 350.

waiting time, which is the sum of the serving time of all previous requests that are already in the queue. We can write the average latency of $n$ requests as $\sum_{i=0}^{n}(iS)/n$, in which $S$ is the serving time of a request. Therefore, the latency seen by each request is correlated to the number of requests in the queue. In the region where latency remains constant, the queue is always full, so the latency of a request equals to it serving time plus the waiting time for all requests in the queue (i.e., $n = \text{Queue}_{\text{Size}}$). Thus, the linear region represents a partially utilized system, and the constant region represents a fully utilized system. Section IV-F provides further details on bandwidth and bottlenecks. From the system perspective, the linear region achieves a lower latency while providing less bandwidth than that of the saturated region. Thus, based on the sensitivity of an application to the latency, a system may exploit these two regions to gain performance. To recap, even for low-contention traffics, NoC and queuing delay contribute significantly to the access latency of the HMC, and subsequently, to the performance of applications.

## C. Quality of Service Analysis

Similar to other networks, QoS of a packet-switched-based memory refers to guaranteeing the required latency or bandwidth for an application. In this section, we inspect techniques to manage the resources in a packet-switched memory to achieve required QoS. In particular, our goal is to ascertain how latency varies within an access pattern (e.g, accesses distributed in four vaults) as a result of the packet-switched interface of the HMC, and subsequently, how this will affect the QoS of applications. The effects of latency variations on QoS are important because they impact latency-sensitive applications [27], QoS guarantees [28], denial of service [29], and multi-threaded and parallel architectures that stall for the slowest thread (i.e., work imbalance). A packet-switched memory, despite its high bandwidth (thanks in part to

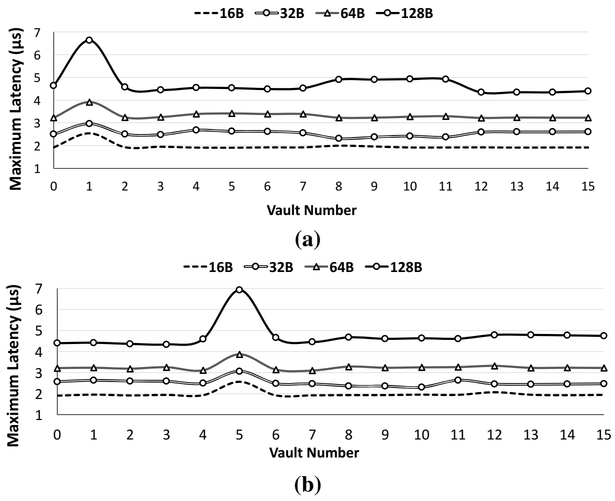

**Fig. 9:** Maximum observed latency in accessing four vaults, three of which are the same. Accessing vault numbers (a) one (3x) and all vaults; and (b) five (3x) and all vaults. X-axis shows the vault number for the vault that is different.

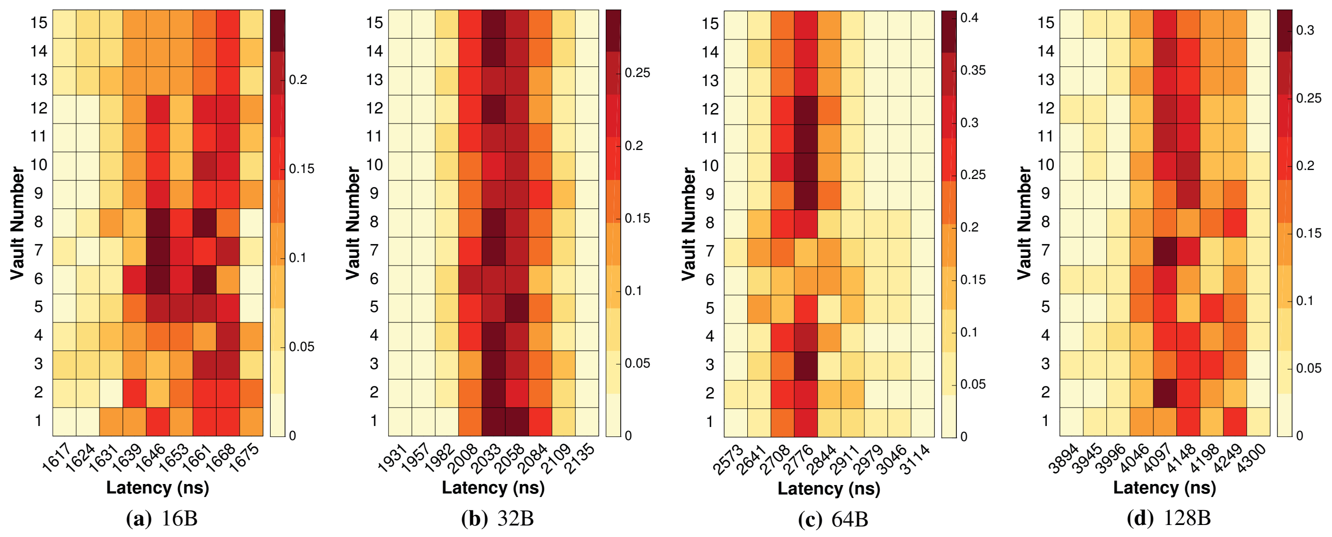

**Fig. 10:** The latency histograms of each vault in heatmaps for various request sizes of (a) 16, (b) 32, (c) 64, and (d) 128 B.

serialization, and high BLP in a small area), adds uncertainty to access latencies. Therefore, as we will see, only optimizing accessing patterns to the HMC in an application would not be sufficient to guarantee a precise QoS.

In our experiments, as a case study, we use four ports with the multi-port stream implementation to generate read accesses to four vaults (targeting 1 GB in total). During the experiments, three ports always access the same vaults, and the fourth port iterates over all possible vaults. Figure 9a and b illustrate the maximum observed latency for two series of experiments, in which the three ports always access vault number one and five, respectively. The figures depict when the accesses of the fourth port are to the same vault as the other ports (i.e., vault numbers one and five in Figures 9a and b, respectively), the maximum observed latency increases up to 40% relative to other accesses. Furthermore, when the fourth port is not accessing the same vault, maximum observed latency varies notably. For instance, the maximum variations are around 200, 330, 400, 600 ns for 16, 32, 64, and 128 B requests, respectively. Since DDR-memory accesses are under 80 ns, even variations of this order will disturb the performance of a system and our assumptions.

In summary, even within the same access pattern, the NoC causes considerable latency variations, which will have a noticeable impact on QoS of an application, even when its access patterns are optimized. Note that Figure 9 illustrates results for only four ports, and if the number of ports (i.e., threads or applications) accessing one vault increases, the latency variations would increase even more. This general trend in latency helps to provide an approximate QoS for various traffic conditions with diverse latency requirement. For instance, in a case that we have five traffic streams, four of which can be served in long latency, and one has high priority and requires a fast service; the system can assign a limited number of vaults to all four low-priority traffic streams, and remaining vaults to the high-priority traffic. Therefore, the QoS of all traffic streams would be satisfied. Such techniques for managing QoS can be provided in the host-side memory controller by real-time remapping, or by reserving resources.

## *D. High-Contention Latency Histograms Per Vault*

To understand the impact of accessing various combinations of vaults on performance, we extend the experiments of the previous section, which accessed four vaults using the multi-port stream implementation. For instance, accesses to four consecutive vaults (e.g., 0, 1, 2, and 3) that share network resources may have higher latency than accesses spread among non-consecutive vaults (e.g., 0, 4, 8, and 12) do. To test this hypothesis, we access all possible combinations of four different vaults (i.e., equal to 1820 combinations, or $^{n!}/_{k!\times(n-k)!}$ for $n = 16$ and $k = 4$) with various request sizes and calculate the average access latency among four vaults. Then, we associate the derived average latency with every vault in that combination.

Figure 10 illustrates our results for various sizes in heatmaps in which a row represents the latency histogram of a vault. In other words, in a row, the color of a rectangle represents the normalized value of the number of accesses in that latency interval against the total number of accesses to the corresponding vault (i.e., $^{1820}/_4$, or 455). As the figure shows, each vault has a different latency behavior. For instance, in

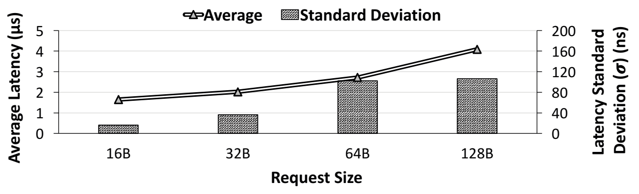

**Fig. 11:** The average and standard deviation of latency across all vaults for various sizes in the four-vault access pattern.

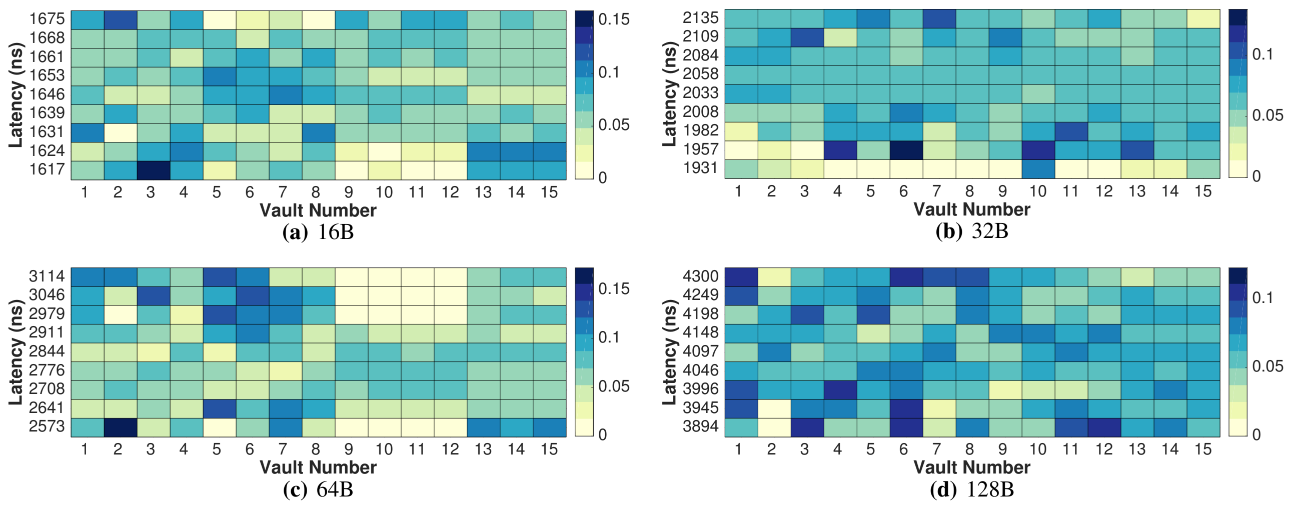

**Fig. 12:** The vault histograms of each latency interval in heatmaps for various request sizes of (a) 16, (b) 32, (c) 64, and (d) 128 B.

Figure 10c, we observe that the histogram of vaults differs substantially (e.g., vault numbers 5, 6, and 7). Although we can investigate these figures in more detail, a quick takeaway is that purely optimizing the general access patterns (in our example, four-vault access pattern) of an application would not guarantee a particular latency. In other words, Figure 10 presents a case study with a four-vault access pattern, in which the only factor of variation is the number of the vault that determines the physical location of a vault within the 3D stack. Therefore, since other factors, such as access pattern, are the same, we conclude that the NoC design of the HMC has a significant impact on the observed latency variations.

As Figure 10 shows, for each request size, although all the vaults have a similar average latency, the distribution of latencies are different among vaults. For a better illustration, Figure 11 depicts the average latency of all vaults and the standard deviation for various packet sizes. We observe that the standard deviation of latencies is 20, 40, 100, and 106 ns for request sizes of 16, 32, 64, and 128 B, respectively. Note that 68% of a population is within $(\mu+\sigma, \mu-\sigma)$, in which $\mu$ and $\sigma$ are average and the standard deviation of that population, respectively. For a particular request size, while the average latency per vault is similar, the distribution of it per vaults covers a broad range. Compared to smaller request sizes, larger request sizes have more variations in latency, because large request sizes occupy larger buffer spaces than small request sizes do. Also, large requests incur extra delays because of reordering and packetizing. Therefore, small request sizes are good candidates for guaranteeing a high QoS. However, as discussed in Section IV-A, small request sizes have low bandwidth efficiency and generally provide lower bandwidth utilization than large request sizes.

In detail, we infer the following insights from Figure 10: (i) Comparing the four subfigures, which indicate the latency for various packet sizes, shows that when the size of requests increases, the latency increases. For instance, the latency of 128 B accesses is in the range of 4 $\mu$s, which is 2.5x higher than for 16 B accesses. A recent paper [19] observes a similar behavior in a limited experiment in accessing to a random vault and conclude such variations is caused by the granularity of 32 B DRAM bus within a vault. (ii) The range of the latency variations for 16, 32, 64, and 128 B accesses are 29, 76, 136, and 203 ns, which indicates that the smallest requests have more consistent latency, and the largest requests have more variable latency. (iii) By comparing the latency of each vault from the rows of each subfigure, we see that each vault has a random behavior, and we cannot allocate a specific latency to a vault based on its location (i.e., number). In other words, the latency of each vault is impacted by many factors such as access patterns and traffic pressure that the contribution of the location of a vault is negligible. According to these three insights, we deduce that important NoC parameters such as the request size and routing protocol have more contributions to the latency *within an access pattern* rather than physical parameters such as the location of a vault do.

### *E. High-Contention Vault Histograms Per Latency Interval*

To explore the contribution of vaults to high and low latencies, each row of Figure 12 depicts contributing vaults for each latency interval and illustrates the histogram of them. The intensity of the color of a rectangle shows the normalized value of the number of that particular appearance of the vault in that latency interval against the maximum number of accesses in that row. Figures 12a, b, c, and d, show colormaps for request sizes of 16, 32, 64, and 128 B, respectively. In Figure 12a, we observe that for gaining the lowest latency (i.e., lowest row), we should avoid accessing vault numbers 9 to 12. In fact, Figure 12 provides a guide for avoiding certain vaults that incur high latencies, but it will not guarantee particular access latencies for a specific vault (similar to the last subsection). For instance, based on Figure 12c, vault number 2 has the highest contribution to the lowest latency interval, and it similarly

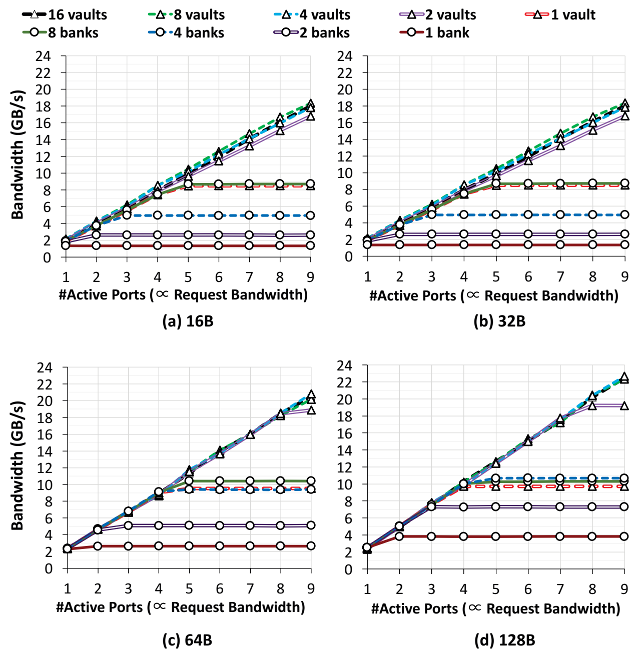

**Fig. 13:** Relationships between the number of active ports, request bandwidth, and bandwidth for various request sizes.

has a high contribution for the highest latency. Therefore, the conclusion that accessing only vault number 2 will guarantee the lowest latency is not correct. However, in the same figure, the chance of incurring lower latency increases by avoiding vaults numbers 9 to 12. Even though we cannot reach a unanimous conclusion about the latency of each vault and the hierarchy of NoC in the HMC, which we discussed its reasoning in the last subsection, we can conclude that the effects of NoC and vault interactions are not trivial.

Based on the observations mentioned in the last paragraph, we interpret that vaults almost equally contribute to high and low latencies. Such behavior suggests two notions to the user or designer of such packet-switched memories: (i) Since lowest latency is obtainable from any vaults, a user may map the memory footprint of an application to optimize other important aspects of accessing these memories, such as access pattern, or request size. In other words, the independence of latency to the physical layout eases the memory mapping constraints; and (ii) a desirable level of performance to an application can be guaranteed by only understanding and following the lowest and highest resulting latency in any access pattern. Note that the uniformity of vault contributions in latency will be sustainable even in a hierarchical connection of many stacks in another interconnection network for creating a large-scale memory. This is because each stack in this new network would have similar characteristics.

## *F. Requested and Response Bandwidth Analysis*

To further investigate potential networking bottlenecks and bandwidth of the HMC, we use the GUPS implementation to tune request rate by changing the number of active ports from one to nine ports. The number of active ports is a proxy for the requested bandwidth because it has a direct relationship with the number of issued requests with the GUPS implementation. Figure 13 presents the relationship between the number of active ports and the response bandwidth for various request sizes. In this figure, sloped lines determine access patterns in which no bottleneck occurs. In contrast, flat lines depict access patterns in which a bottleneck (e.g., vault bandwidth limitation) exists. As a recent work about HMC characterization also mentioned [19], the factor that limits the bandwidth utilization can be related to the packet-switched network, such as the limited size of queues in the vault controller or DRAM layers. We analyze the reasons of saturation points by taking a deeper look at a vault controller, which is basically a stationary system, receiving requests with an arrival rate. Based on Little's law, in such systems, the average number of outstanding requests equals to the arrival rate multiplied by the average time a request spends in the system. To calculate the number of outstanding requests based on the numbers represented in Figure 13, we measure the latency at saturated points and multiply them by input rates, and then divide the result by request size. The result of this calculation illustrated in Figure 14 indicates that regardless of request size, the maximum number of requests is 288 for two banks and 535 for four banks, on average. Moreover, the linear relationship between the number of outstanding requests and number of banks suggests that a vault controller dedicates one queue for each bank or for each DRAM layer.

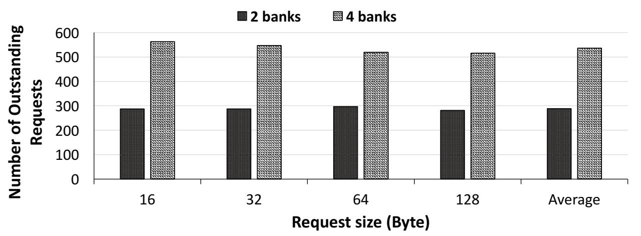

**Fig. 14:** The number of estimated outstanding requests in two- and four-bank access patterns.

As discussed in Section IV-A, we observe that accessing eight banks within a vault saturates the internal 10 GB/s bandwidth of a vault for request sizes of 16 and 32 B. In addition, for 64 and 128 B request sizes, accessing four banks saturates the internal bandwidth of a vault. Thus, within a vault, depending on the size of requests, increasing BLP to more than eight or four banks will not provide higher bandwidth. In fact, as Figure 3 presents, for accessing a 4 KB OS page in the HMC, requests are first spread over vaults and then banks. Therefore, accessing a single page in this configuration naturally avoids this bottleneck. We can extend this insight to more than one OS pages that are sequentially allocated in the address space. For instance, accessing more than four sequentially allocated OS pages would invoke the bottleneck of the internal bandwidth of vault. To effectively utilize the limited bandwidth of vaults within the HMC, application access patterns must be matched to increasing vault-level and then bank-level parallelism.

Compared to traditional DRAM memories, the HMC supplies a higher amount of bandwidth and concurrency due to the high number of vaults and independent vault controllers. Figure 13d exhibits this point by showing that for 128 B requests, distributed access patterns to more than two vaults quickly reach the bottleneck of the external bandwidth of two links. This is a limitation of our particular HMC infrastructure (two half links from the FPGA to the HMC), as the number and width of the HMC links can be increased as can the speed and efficiency of the FPGA infrastructure (i.e., HMC controller and associated firmware). Since HMC uses bi-directional links, issuing only read requests results in an asymmetric usage of the available bandwidth. In other words, read requests only fully utilize response bandwidth, and write requests only fully utilize request bandwidth. Previous studies [17], [30] have investigated this asymmetry, and proposed issuing a mix of read and write requests to address it. In addition to optimizing access patterns, applications should also balance the ratio of read and write requests for effectively utilizing bi-directional bandwidth of stacked-memory networks.

## V. Related Work

Previous works have characterized the HMC [16]–[18], [30], from which Schmidt et al. [17] agreed with our measured bandwidth and latency. Another work, [19], using the AC-510 accelerator board, they characterized bandwidth of the HMC and its relationship with temperature, power, and cooling power. They deconstructed the contributing factors to the latency, but they focused more on power and temperature. Although these studies have explored emulated HMC and earlier HMC prototype chips, they have not studied the performance impacts of the internal NoC on the performance and QoS of the HMC, and in general the impact of packet-switched networks on the performance of 3D-stacked memories.

Other recent studies have focused on designing an efficient NoC for the HMC. Zhan et al. [31] proposed solving issues that show up in a NoC coupled with HMC, such as traffic congestion, uncoordinated internal and external networks, and high power consumption by co-optimizing networks that are both inside each HMC and between cubes. Their proposed unified memory network architecture reuses the internal network as a router for the external network, which allows bypassing of remote accesses while also providing high bandwidth for local accesses. The authors also proposed reducing communication loads and using power gating to further decrease power consumption for an overall 75.1% reduction in memory access latency and a 22.1% reduction in energy consumption.

Azarkhish et al. [32] proposed a low latency AXI-compatible interconnect, which provides the required bandwidth for an HMC infrastructure so that it supports near memory computation. Their simulation results show that the main bottleneck for delivered bandwidth is the timing of DRAM layers and TSVs. Also, their analysis on PIM traffic with increased requesting bandwidth on the main links showed that when the host demands less than 120 GB/s no saturation occurs. In another work, Fujiki et al. [33] proposed a scalable low-latency network by using a random topology based on the length of communication path, using deadlock-free routing, and memory-mapping in granularity of a page size. Their full-system simulation models show that this method reduces cycles by 6.6%, and that random networks with universal memory access out-perform non-random localized networks.

## VI. Conclusion

In this paper, we evaluate the internal NoC of the HMC, a real-world prototype of a NoC-based, 3D-stacked memory. From our experiments, we can provide the following insights into the effects of the internal NoC of the HMC on the performance of systems and applications.

- Large and small request sizes for packets provide a trade-off between effective bandwidth and latency as a result of buffering, packetization, and reordering overheads. In contrast with traditional DDRx systems, this trade-off enables tuning memory accesses to optimize either bandwidth or latency. (Section IV-A, IV-D, and IV-F)
- As future memories become denser with more links and vaults, queuing delays will become a serious concern for packet-based memories, such as the HMC. Effective solutions should focus on (i) optimizing queuing on the host controller side and at vault controllers or (ii) distributing accesses to improve parallelism, such as BLP. (Section IV-B and IV-C)
- The internal NoC complicates QoS for memory accesses because of meaningful variations in latency even within an access pattern. On the other hand, it creates opportunities such as (i) smaller packets are ensured to have improved QoS at a cost of reduced bandwidth or (ii) high-priority traffics can be mapped to access their private vaults. (Section IV-C, IV-D, and IV-E)
- Limited bandwidth within a vault means that mapping accesses across vaults then banks is key to achieve better bandwidth utilization and lower latency. (Section IV-A and IV-F)
- The packet-based protocol creates an asymmetric bi-directional bandwidth environment that applications should be aware of and optimize for the proper mix of reads and writes for effectively utilizing external bandwidth. (Section IV-A, IV-F, and [17])
- Finally, the exact latency of a vault is impacted by many factors such as access patterns and traffic conditions that the latency contribution of the physical location of a vault is negligible within an access pattern. This insight reduces complexity and constraints of optimization and mapping techniques. (Section IV-D and IV-E)

## Acknowledgments

We thank anonymous reviewers for their valuable comments and feedbacks for improving the paper. Our experimental hardware is partially supported by Micron. This study was supported in part by National Science Foundation under grant number CCF-1533767.

## References

[1] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin *et al.*, "25.2 A 1.2V 8Gb 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV," in *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2014.

[2] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *Symposium on VLSI Technology (VLSIT)*. IEEE, 2012.

[3] T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips Symposium (HCS)*. IEEE, 2011.

[4] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric System Interconnect Design with Hybrid Memory Cubes," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2013.

[5] HMC Consortium, "Hybrid Memory Cube Specification 1.1," *Retrieved from hybridmemorycube.org*, 2013, [Online; accessed 2017-10-10].

[6] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2006.

[7] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, "Optimizing GPU Energy Efficiency with 3D Die-Stacking Graphics Memory and Reconfigurable Memory Interface," in *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4. ACM, 2013.

[8] 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 *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014.

[9] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2014.

[10] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.

[11] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2016.

[12] L. Nai and H. Kim, "Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals," in *International Symposium on Memory Systems (MEMSYS)*. ACM, 2015.

[13] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2017.

[14] R. Hadidi, L. Nai, H. Kim, and H. Kim, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory," *ACM Trans. Archit. Code Optim.*, vol. 14, Dec. 2017.

[15] L. Nai, R. Hadidi, H. Xiao, H. Kim, J. Sim, and H. Kim, "CoolPIM: Thermal-Aware Source Throttling for Efficient PIM Instruction Offloading," in *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018.

[16] M. Gokhale, S. Lloyd, and C. Macaraeg, "Hybrid Memory Cube Performance Characterization on Data-centric Workloads," in *Workshop on Irregular Applications: Architectures and Algorithms (IA$^3$)*. ACM, 2015.

[17] J. Schmidt, H. Fröning, and U. Brüning, "Exploring Time and Energy for Complex Accesses to a Hybrid Memory Cube," in *International Symposium on Memory Systems (MEMSYS)*. ACM, Oct. 2016.

[18] K. Z. Ibrahim, F. Fatollahi-Fard, D. Donofrio, and J. Shalf, "Characterizing the Performance of Hybrid Memory Cube Using ApexMAP Application Probes," in *International Symposium on Memory Systems (MEMSYS)*. ACM, 2016.

[19] R. Hadidi, B. Asgari, B. Ahmad Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the Characteristics of 3D-Stacked Memories: A Case Study for Hybrid Memory Cube," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2017.

[20] PicoComputing, "AC-510 HPC Module," http://picocomputing.com/ac-510-superprocessor-module/, 2017, [Online; accessed 2017-10-10].

[21] PicoComputing, "EX700 Backplane," http://picocomputing.com/products/backplanes/ex-700/, 2017, [Online; accessed 2017-10-10].

[22] HMC Consortium, "Hybrid Memory Cube Specification 1.0," *Retrieved from hybridmemorycube.org*, 2013, [Online; accessed 2017-10-10].

[23] PicoComputing, "SC6-Mini," http://picocomputing.com/products/picocube/picomini/, 2017, [Online; accessed 2017-10-10].

[24] PicoComputing, "HMC Controller IP," http://picocomputing.com/productshybrid-memory-cube-hmc-controller-ip-2/, 2017, [Online; accessed 2017-10-10].

[25] PicoComputing, "Pico Framework," http://picocomputing.zendesk.com/hc/en-us, 2017, [Online; accessed 2017-10-10].

[26] Rosenfeld, Paul, "Performance Exploration of the Hybrid Memory Cube," Ph.D. dissertation, University of Maryland, College Park, 2014.

[27] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS Policies and Architecture for Cache/Memory in CMP Platforms," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, vol. 35, no. 1. ACM, 2007.

[28] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, 2013.

[29] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security Symposium on USENIX Security Symposium*. USENIX Association, 2007.

[30] P. Rosenfeld, E. Cooper-Balis, T. Farrell, D. Resnick, and B. Jacob, "Peering over the Memory Wall: Design Space and Performance Analysis of the Hybrid Memory Cube," Technical Report UMD-SCA-2012-10-01, University of Maryland, Tech. Rep., 2012.

[31] J. Zhan, I. Akgun, J. Zhao, A. Davis, P. Faraboschi, Y. Wang, and Y. Xie, "A Unified Memory Network Architecture for In-Memory Computing in Commodity Servers," in *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2016.

[32] E. Azarkhish, C. Pfister, D. Rossi, I. Loi, and L. Benini, "Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube," *IEEE Transactions on VLSI Systems (VLSI)*, vol. 25, 2017.

[33] D. Fujiki, H. Matsutani, M. Koibuchi, and H. Amano, "Randomizing Packet Memory Networks for Low-latency Processor-Memory Communication," in *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE, 2016.