A Unified Memory Network Architecture for In-Memory Computing in Commodity Servers

Jia Zhan*, Itir Akgun*, Jishen Zhao†, Al Davis‡, Paolo Faraboschi‡, Yuangang Wang§, and Yuan Xie*
*University of California, Santa Barbara †University of California, Santa Cruz ‡HP Labs §Huawei

***Abstract*—In-memory computing is emerging as a promising paradigm in commodity servers to accelerate data-intensive processing by striving to keep the entire dataset in DRAM. To address the tremendous pressure on the main memory system, discrete memory modules can be networked together to form a memory pool, enabled by recent trends towards richer memory interfaces (e.g. Hybrid Memory Cubes, or HMCs). Such an *inter-memory network* provides a scalable fabric to expand memory capacity, but still suffers from long multi-hop latency, limited bandwidth, and high power consumption—problems that will continue to exacerbate as the gap between interconnect and transistor performance grows. Moreover, inside each memory module, an *intra-memory network* (NoC) is typically employed to connect different memory partitions. Without careful design, the back-pressure inside the memory modules can further propagate to the inter-memory network to cause a performance bottleneck.**

To address these problems, we propose co-optimization of intra- and inter-memory network. First, we re-organize the intra-memory network structure, and provide a smart I/O interface to reuse the intra-memory NoC as the network switches for inter-memory communication, thus forming a unified memory network. Based on this architecture, we further optimize the inter-memory network for both high performance and lower energy, including a distance-aware *selective* compression scheme to drastically reduce communication burden, and a light-weight power-gating algorithm to turn off under-utilized links while guaranteeing a connected graph and deadlock-free routing. We develop an event-driven simulator to model our proposed architectures. Experiment results based on both synthetic traffic and real big-data workloads show that our unified memory network architecture can achieve 75.1% average memory access latency reduction and 22.1% total memory energy saving.

I. Introduction

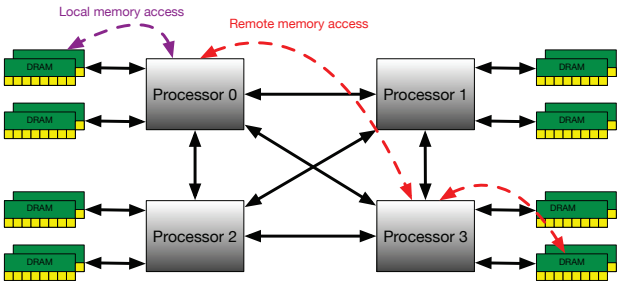
The last decade has witnessed a rapid growth of digital data, covering each field of humankind interests. Analysis from International Data Corporation (IDC) shows that we can expect a 300-fold increase in digital data, from 130 exabytes to 40,000 exabytes, to be processed by various computing systems from 2005 to 2020, i.e., doubling every two years [1]. The above facts give birth to the widely circulated concept of *Big Data* [2], [3], [4], which poses significant performance and energy challenges to data movement and storage, especially for low-latency web services and real-time data analytics. To provide fast access to large-scale data stores, recent systems strive to fit entire application data into main memory, such as in-memory databases [5], [6], [7] and general-purpose big data analytics platforms [8], [9].

While these **in-memory computing** techniques [10], [11] accelerate CPU processing speed with the vast amount of data largely residing in main memory, they also pose significant challenges in DRAM scaling, especially for memory capacity. If the large dataset cannot fit entirely in memory, it has to be spilled to disk, thus causing sub-optimal performance due to disk I/O contention. Unfortunately, with the current DDRx based memory architecture, the scaling of DRAM capacity falls far behind the growth of dataset size of in-memory computing applications [12]. A straightforward approach to increase memory capacity is to add more CPU sockets for additional memory channels. However, this will introduce significant hardware cost [13]. Moreover, in a multi-socket server system, only a small fraction of data resides in the local memory directly attached to the processing CPU. As a result, frequent remote-socket memory accesses are needed to access the vast majority of the dataset. In traditional processor interconnects that connect multiple sockets, such as Intel QuickPath Interconnect (QPI) [14] and HyperTransport [15], remote memory access has to be routed through the processor before reaching the remote memory, as shown in Figure 1a, which may incur significant non-uniform memory access (NUMA) delay.

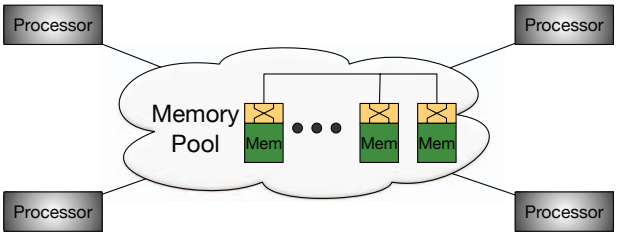
As technologies are now coming to market to allow discrete memory modules in a server to be networked together, such as the Hybrid Memory Cube (HMC) [16] and HP's The Machine project [17], we can disaggregate the per-socket local memory to form a *separate shared memory pool* for all CPU sockets. Similar to the memory blade [18], [19] concept at the rack scale for memory expansion and on-demand allocation, the memory pool can efficiently scale the memory capacity to accommodate the vast dataset for in-memory computing, and provide on-demand memory allocation to different CPU sockets depending on the workload requirements. For example, Figure 1b shows the processor-memory architecture in a 4-socket server system. Inside the memory pool, a **memory network**[1] is employed as the communication backbone. For each memory node, we leverage 3D-stacked memories like HMCs, with an *intra-memory Network-on-Chip* (NoC) on the bottom logic layer which connects the I/O ports to the memory controllers (MCs) for internal memory access. Recently, Kim et al. [20], [21] proposed the memory network concept mainly for memory bandwidth improvement and explored different inter-memory network topologies. However, in order to provide a high-performance and low-power memory system

978-1-5090-3508-3/16/$31.00

[1]We will use "memory network" and "inter-memory network" interchangeably throughout this paper.



(a) A quad-socket system employing Intel QuickPath Interconnect (QPI). The remote memory access has to be routed through other processors before reaching the remote memory, which may incur significant memory access delay.



(b) A disaggregated memory pool in a multi-socket server system. The discrete memory modules are interconnected through a memory fabric and can be directly accessed by different sockets.

Fig. 1: Illustration of processor-memory interconnect in a multi-socket server system with (a) CPU-centric network, and (b) memory-centric network.

for in-memory computing, we still face the following critical challenges: **First**, the intra-memory NoC has not been optimized for the unique intra-memory traffic pattern, and it lacks efficient coordination with the external network. Consequently, the back-pressure inside the memory modules can further propagate to the inter-memory network to cause a performance bottleneck. **Second**, the inter-memory network may suffer from severe traffic congestion and high energy consumption because of the intensive access to the in-memory dataset which generates frequent multi-hop network traversals.

To address these challenges, we provide co-optimization of intra- and inter-memory network, which provides a unified memory network architecture for in-memory computing workloads. Specifically, we redesign the intra-memory network and its I/O interface to inter-memory network for fast end-to-end memory access. In addition, for the inter-memory network, observing that *there is no direct communication between memory modules*, we propose customized network traffic compression and link power-gating techniques to address the latency and power challenges due to the massive number of memory access to the shared memory fabric. In summary, our main contributions are as follows:

- Based on the study of in-memory computing workloads, we identify them as one of the most suitable applications for the disaggregated memory pool architecture. Furthermore, we reveal the inefficiencies of existing memory network designs to support such big data workloads.
- We propose a unified memory network design, which reuses the intra-memory networks built into the memory modules as the intermediate switches for the inter-memory network. The redesigned intra-memory network provides both *local routes* for high-bandwidth local memory access and *bypass routes* to accelerate remote node access.
- For inter-memory network, we design a customized network compression architecture to drastically reduce the communication loads. A *distance-aware selective* compression algorithm is proposed to reduce the data compression and decompression overhead.
- We identify off-chip links to be the major power consumer of the memory system and discover the skewed load distribution in the memory network. To address the high link power, we propose a novel power-gating algorithm customized for the unique network traffic pattern, which always guarantees a connected graph with deadlock-free routing.
- We implement an event-driven simulator to evaluate all above techniques, running both synthetic traffic and real in-memory computing workloads. On average, our approaches achieve 75.1% latency reduction for memory access and 22.1% energy saving for the entire memory system.

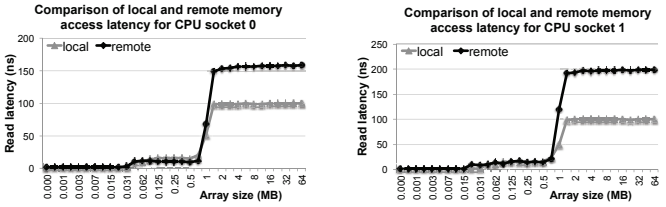
II. Background

In this section, we briefly introduce the background of in-memory computing, and discuss the disaggregated memory pool architecture to support such emerging applications.

In-Memory Computing: In-memory computing has become a widely adopted technology for real-time analytics applications in the industry. One important application is in-memory database such as SAP HANA [5], VoltDB [6], and IBM DB2 BLU [7], which strives to *fit an entire database in memory*, potentially obviating the need of paging data to/from mechanical disks. In a related field, the open-source community is also actively advancing big data processing frameworks that leverage in-memory computing. For example, Spark [8] is a cluster computing framework optimized for data reuse in iterative machine learning algorithms and interactive data analysis tools. Unlike MapReduce [22] which has to write data into external storage system (e.g. HDFS [23]) for data reuse between computations, Spark creates a new distributed memory abstraction [24] and lets users explicitly *persist intermediate dataset in memory*, which can achieve up to 100× performance speedup.

Non-Uniform Memory Access (NUMA): On the flip side, in-memory computing workloads heavily rely on memory capacity. Increasing the number of CPU sockets grants us additional memory channels to connect more DIMMs, but comes with significant hardware cost [13]. In addition, the long NUMA latency in remote CPU sockets will become a major obstacle for real-time in-memory computing. To get a

sense of the latency discrepancy between local and remote memory accesses, we execute lmbench [25] in a native two-socket server system running Red Hat 6.6. Each socket consists of a dual-core AMD Opteron 2216 processor with 8GB local memory. The benchmark iteratively conducts read operations on memory arrays across the entire memory address space. We use Linux API `numactl` to collect NUMA statistics. Note that the entire memory hierarchy is measured, including cache, main memory, and TLB latency. As shown in Figure 2, we report the average memory read latency in nanoseconds per load when varying the memory array sizes, with a stride[2] size of 64 bytes.



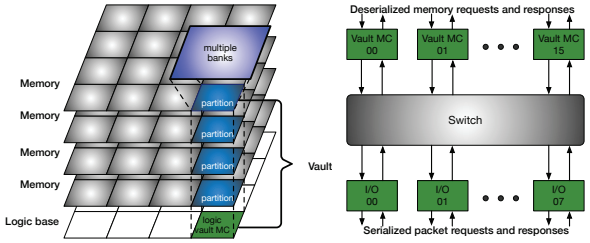
(a) CPU socket 0 (b) CPU socket 1

Fig. 2: The comparison of local and remote memory read latency for various memory sizes, with a stride size of 64 bytes. The results are reported in nanoseconds per load.

As we can see from both Figure 2a and Figure 2b, for small array sizes, the chance of a cache hit is high and the loads will be very fast. The latency difference between local and remote memory accesses becomes apparent when the array size exceeds 1.5MB, which is mainly due to the delay in cross-socket HyperTransport [15] links. For distributed file systems such as HDFS [23] on which MapReduce or Spark frameworks run, the block size is usually as big as 64MB or 128MB. Compared to the local memory access, the remote memory access latency is increased by 57.3% for CPU 0 and 96.8% for CPU 1, averaged through the array size of 1.5MB to 64MB. Therefore, such NUMA latency is clearly undesirable for real-time in-memory computing.

Disaggregated 3D-Stacked Memory Pool: Instead of monotonically increasing the per-socket memory capacity while still suffering from long NUMA latency, we can disaggregate the local memory system into a shared memory pool. This is enabled by the emerging three-dimensional (3D) memory stacking [26], [27], [28], [29] technology, which not only multiplies the per-node capacity by stacking multiple DRAM layers together, but also enables discrete memory modules (e.g. Hybrid Memory Cubes, or HMCs) to be networked together with a scalable memory fabric. For example, Figure 3a shows a conceptual diagram of the HMC structure. An HMC consists of a single package containing multiple DRAM dies and one logic die. These different dies are stacked together with through-silicon vias (TSVs). At each DRAM layer, the memory is partitioned, with each partition consisting of multiple banks. Furthermore, different partitions across layers form a vertical slice or *vault*. In addition, each vault has a memory controller (MC) in the logic base which manages all the memory operations within that vault. In the logic layer, as shown in Figure 3b, there is also a switch that connects the I/Os with the vault MCs. For example, with 16 vault MCs and 8 I/O ports, a 24×24 crossbar network can be formed. We denote such on-chip network as the **intra-memory network**.

[2]The stride of an array is the distance between the starting memory addresses of 2 adjacent elements.



(a) Die-stacked structure (b) Intra-memory network in the logic layer

Fig. 3: Hybrid Memory Cube (HMC) organization

Additionally, enabled by the memory interface integrated in the logic base, multiple HMCs can be interconnected to form an interconnection network—the so-called **(inter-) memory network** [20]. Industry efforts from HP [17] and IBM [30] also demonstrate technology trends towards memory network for efficient main memory expansion. Note that we use HMC as the main memory module in our design as an example, although *the memory modules can be generalized to any stacked memory on top of a logic layer.*

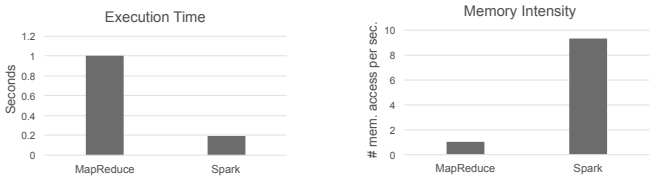
III. Challenges of Memory Network Design

The main purpose of initial memory network efforts [20], [21] is to enable direct inter-memory connection for bandwidth enhancement, instead of routing through traditional QPI [14], HyperTransport [15], or NVlink [31]. However, to facilitate the adoption of memory network for in-memory computing applications, we still need to resolve many critical performance and power challenges.

A. Massive Data Access

In-memory computing not only requires high memory capacity for data store, but also generates highly frequent memory accesses. As a result, the shared memory network may experience heavy traffic due to frequent data transfer over the network. To show the memory access intensity of in-memory computing workloads, we run `word-count` jobs on both MapReduce and Spark frameworks natively on a quad-core server with 16 GB memory, using the same 10GB Wikipedia dataset provided in BigDataBench [2]. Figure 4 compares the execution time of these two different applications, as well as their memory access intensity measured by average number of memory accesses per second.

As shown in Figure 4a, using in-memory computing, Spark achieves over 5× performance speedup compared to MapReduce. However, the intensity of memory accesses of Spark application is also over 9× higher. Therefore, both the intra-



(a) Job execution time (b) # memory accesses/sec.

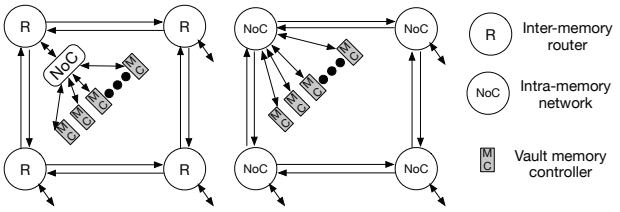
Fig. 4: Evaluation of `word-count` jobs on a 10 GB Wikipedia dataset, using MapReduce and Spark frameworks, respectively. (1) The total execution time of the job, and (2) the memory intensity measured by the average number of memory accesses per second. Results are normalized.

and inter-memory network should be optimized to accommodate the intensive memory access.

B. Bottleneck at Intra-Memory Network

It is still not clear how we should handle the interface between the intra- and inter-memory network. Without careful design, even though we can have a robust and scalable inter-memory network to connect many memory nodes, the intra-memory NoC may become the performance bottleneck due to the massive number of memory accesses.

A generic design to coordinate the inter- and intra-memory network is shown in Figure 5a. We call it the decoupled network. It uses separate network routers and links for different networks. However, there is only a single channel connecting an inter-memory router to the intra-memory NoC, which will become the traffic bottleneck for all local memory accesses. In contrast, inspired by the design of the recent Anton 2 [32], [33] supercomputer, we propose a unified network as shown in Figure 5b. The intra-memory NoC will be reused as the router for inter-memory network. In this way, both networks will be seamlessly integrated/unified. We will provide more details of this design in Section IV.



(a) Decoupled network (b) Our proposed unified network

Fig. 5: Two different memory network designs: the decoupled network in (a) has dedicated routers for inter-memory network routing, and each router is connected to a local intra-memory network (NoC); The unified network in (b) reuses the intra-memory NoC as the router for inter-memory communication.

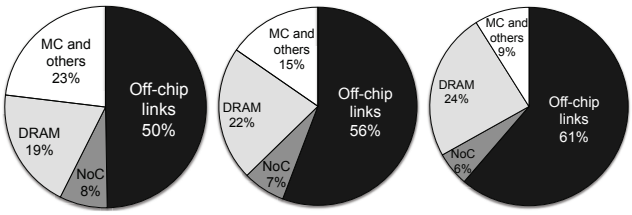
However, with such a unified network design, the intra-memory NoC may still congest the network because it has to serve heavily interfering packets from both intra-memory and inter-memory communication. Moreover, the conventional intra-memory network employs a crossbar structure to connect the I/Os with MCs. It provides short end-to-end diameters but poor scalability, because it comes with tremendous number of wires and interior pass-gates ($O(N^2)$ where N is the number of I/Os plus MCs). Also, some internal bandwidth in a fully-connected crossbar is wasted as MCs do not communicate with each other.

Therefore, to maximize the potential of our unified intra- and inter-memory network, we need to (1) design a lightweight and scalable intra-memory network topology for fast memory access, and (2) minimize the interference of intra-memory and inter-memory traffic.

C. Problems with Inter-Memory Network

It is very straightforward that the inter-memory network may experience heavy network congestion due to intensive memory accesses. In the meantime, the high energy consumption to serve these packet traversals should also be studied.

3D-stacked DRAM memory facilitates high-speed serial links between processor and memory nodes for high-bandwidth data transfer, but at a high power cost. Figure 6 shows the power breakdown of a 3D-stacked DRAM based on the HMC prototype. The detailed evaluation methodology is described in Section VI-D. As we can see, the inter-memory (off-chip) links consume a dominating percentage of the total memory power, which increases from 49.7% to 61.3% when technology scales from 45nm to 22nm.



(a) 45nm (b) 32nm (c) 22nm

Fig. 6: Power breakdown of a 3D stacked memory module with different technology nodes (45nm, 32nm, and 22nm).

Figure 7 shows the total access count of all the network links in a 4×4 mesh network after profiling an in-memory computing workload (`Spark-wordcount`) in our system (see Table I in Section VI). Note that the five ports (including local port) per router represent its five output links. We observe that the utilization of different links in the memory network varies significantly. Therefore, while link power is a major concern for the memory network, the skewed traffic pattern in the network presents opportunities to avoid power wasted on under-utilized links.

Therefore, in the following two sections, we will optimize both the intra- and inter-memory network to address these challenges.

IV. Intra-Memory Network Design

Interconnecting many memory modules that themselves contain on-chip networks adds a new layer of complexity to the network design. Here we propose a **unified intra- and**

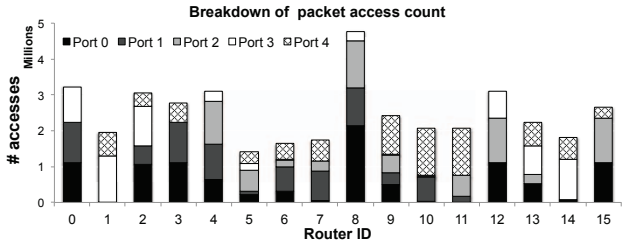


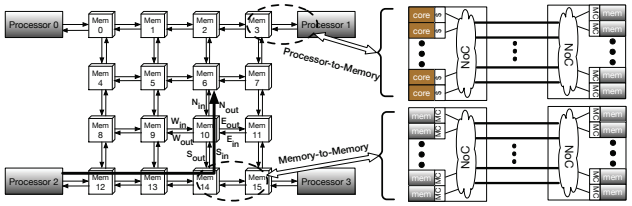
Fig. 7: The distribution of total access count for all network ports at different routers in a 4×4 mesh network, when running `Spark-wordcount` workload in our system.

inter-memory network for fast memory access. As shown in Figure 5b, the key idea is to *reuse the intra-memory network to implement the switching functionality of the inter-memory network.* To avoid congesting the intra-memory NoC, we will propose a light-weight intra-memory network structure, and further optimize the I/O interface to deflect inter-memory traffic from polluting intra-memory network.

A. Overview of Network Architecture

Figure 8 shows a high-level view of our processor-memory network architecture. Sixteen memory modules form a disaggregated memory pool using a mesh-based inter-memory fabric, shared by four CPU sockets. Inside the CPU or memory nodes, there are Network-on-Chips (NoCs) to connect multiple cores (for CPU nodes) or multiple memory partitions (for memory nodes). Note that prior work [20] has explored more complex inter-memory network topologies such as distributed flattened-butterfly with additional links for latency reduction, but can suffer from link over-subscription, highly-skewed router radix, and high network power. Instead, we adopt a simple mesh based inter-memory network, as shown in Figure 8a, and focus on its coordination with intra-memory network for fast memory access. Figure 8b shows the channel structure between a processor and a memory module, and between two memory modules. As illustrated in Figure 5b, such unified network architecture provides significant design simplicity and efficiency, with a low-cost intra-memory network not only serving as the communication backbone for internal memory access, but also as a bridge to connect with inter-memory network for end-to-end latency reduction.

However, as discussed in Section III-B, the intra-memory NoC may still experience significant delay due to the inefficient crossbar structure and the heavily mixed intra- and inter-memory traffic. Moreover, the back-pressure inside the memory modules can further propagate to the inter-memory network to cause a performance bottleneck. Therefore, in order to fully utilize the potential of our unified network architecture, two challenges remain to be solved: (1) we need to replace the conventional crossbar structure with a lightweight and scalable topology for fast intra-memory access; (2) we need to minimize the interference between intra- and inter-memory traffic to avoid NoC congestion.

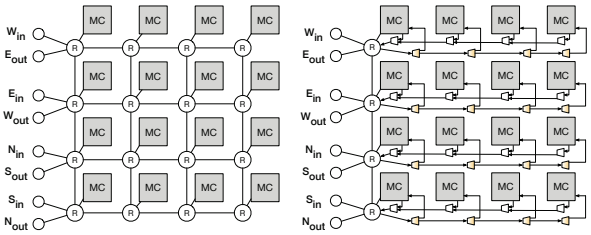


(a) Inter-memory network (b) Proc-to-mem and mem-to-mem channels

Fig. 8: An example unified processor-memory network: The inter-memory network connects the processor and memory nodes using a mesh topology. There is an NoC in each processor and memory node. In the processor node, cores are connected using an NoC. In the memory node, different MCs are connected using an NoC, i.e., the intra-memory network.

B. Intra-Memory Network Topology

In replacement of the crossbar structure, we propose two intra-memory network designs as shown in Figure 9. Internally, the 16 vault MCs form a network. Due to package- and board-level design considerations, the high-speed I/O pins are distributed along the edge of the chip, and are directly connected to the edge MCs. Here we use 4 I/O pairs for connections from different directions: North (N), East (E), South (S), and West (W), with a subscript to indicate input (in) or output (out) port.



(a) Mesh (b) Reduction/dispersion tree

Fig. 9: Intra-memory network designs in the logic layer, which connect the I/Os with the memory controllers (MCs). (a) Mesh based intra-memory network. (b) Reduction/dispersion tree based intra-memory network.

Various topologies can be applied to the intra-memory network. For example, Figure 9a shows a simple 4×4 mesh network. *However, the direct links between different MCs are redundant as MCs do not communicate directly with each other.* Instead, they only receive memory requests from the input ports and reply to the output ports at the edge. Being aware of such traffic pattern, we design a customized topology to connect MCs and I/O ports, derived from the structure of reduction/dispersion tree [34]. As shown in Figure 9b, the direct MC-to-MC links are removed. The reduction tree is designed for the low-latency delivery of packets from the internal MCs to the edge output port, and conversely the dispersion tree carries input requests to the destination

MCs. Unlike normal switch designs [35], [36] that usually employ a crossbar between input and output ports, we only need simple 2-to-1 multiplexers for the reduction tree, which multiplexes packets from the local port with those already in the network; and 1-to-2 de-multiplexers for the dispersion tree, which dispatches the packets to either the local output port or the input of another de-multiplexer. Virtual channel (VC) based buffering is needed at each node for reducing head-of-the-line blocking. Routing and output port selection logic is not necessary, and just one arbiter is required per node.

C. Smart I/O Interface

As illustrated in Figure 5b, the intra-memory network is reused as the router for the inter-memory network—the four I/O pairs implement the functionality of the four router ports (north, east, south, and west). In this way, the local memory is attached to the inter-memory network through the I/O interfaces and communicates with other nodes.

There are two types of on-chip routes: 1) **local routes** for packets that are destined to an internal memory partition within this node, and 2) **through routes** for packets that are passing through the intermediate node destined to a remote node. For local routes, as shown in Figure 9b, packets firstly hop along the edge routers, and then follow the dispersion tree to the destined MC for memory access. For through routes, packets can quickly *bypass* the intra-memory network through the edge routers that are coupled with I/O ports. As shown in Figure 9b, since we use dimension-order routing for the inter-memory network, the I/O ports are organized in pairs, with each pair consisting of an input port and an output port following the same dimension. Every I/O pair is attached to a single router for fast intra-memory bypassing. For example, S_{in} and N_{out} are paired. Suppose a packet enters S_{in} of memory node 10 but its final destination is node 6, as shown in Figure 8a. Due to our proposed intra-memory network organization, it can promptly exit at N_{out} via the edge router, without traversing multiple hops inside node 10. In summary, with our smart I/O interface, *the inter-memory traffic is handled along the edge routers, so that the intra-memory network can dedicate its full bandwidth for fast local memory access without interference from inter-memory traffic.*

V. Inter-Memory Network Design

Our prior unified network design eliminates potential bottleneck at each local node, but for the inter-memory network which connects all these nodes, it may still suffer from insufficient network bandwidth and high network energy to handle the massive number of memory accesses. Therefore, in this section, we mainly focus on the optimization of inter-memory network to achieve both high performance and low power network design.

A. Distance-Aware Memory Network Compression

If we can reduce the packet count from the source, both performance and energy of the processor-memory network will be optimized. To achieve this goal, we propose a generic *memory network compression* technique, which transmits data packets through the network in a compressed format. Unlike NoC compression [37] which usually deals with homogeneous multi-core architecture with symmetric on-chip cache traffic, memory network compression needs to be aware of two design implications:

- The off-chip processor-to-memory traffic is asymmetric. CPU nodes actively generate memory messages while memory nodes only passively receive/reply those requests. There is no direct communication between memory modules.
- As shown in Figure 5, even with our unified network to avoid all memory traffic going through a single network interface between intra- and inter-memory network, the four I/O pairs per memory node still have to handle highly concentrated memory traffic from 16 MCs as shown in Figure 9b. Thus, it will be costly to perform data compression/decompression at these interfaces for every packet.

Therefore, we first provide a decoupled encoder/decoder placement strategy to accommodate the asymmetric processor-to-memory traffic. Then, we propose a novel distance-aware *selective* compression algorithm to reduce network burden without incurring unnecessary energy and delay overhead. One nice merit of our design is its compatibility with all kinds of data compression techniques. We use the state-of-the-art delta compression [38], [39], though our compression architecture design can be applied to other compression techniques [40], [41].

1) Decoupled Encoder/Decoder Placement

Compression (encoder) and decompression (decoder) units can be integrated in the network interface (NI) of every processor and memory node, as shown in Figure 10a. As such, the incoming packets can be decompressed for local usage, and reversely the response packets can be compressed before being ejected to the network. However, this design generates more hardware overhead as some of the decoders are redundant. Unlike processors, memory nodes are used for storing data rather than actually processing them. *As a result, a compressed packet from the source processor does not need to be decompressed at the destination memory node. It only needs to be decompressed when fetched by a processor.*

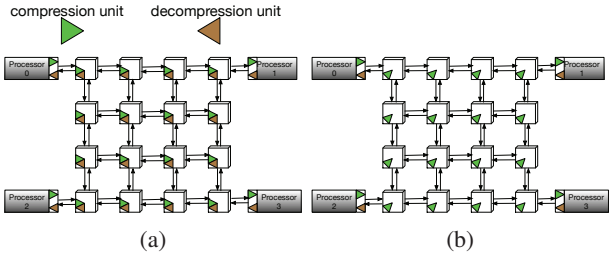


Fig. 10: Different placement strategies for the encoders/decoders in order to support memory network compression. (a) All processors and memories have encoders/decoders; (b) Processors have encoders/decoders, while memories only have encoders.

Therefore, observing such traffic pattern, we can decouple the encoder-decoder pairs at each node. As shown in Fig-

ure 10b, the processors have the complete encoder-decoder pairs, but the memory nodes only have encoders. By removing the redundant decoders, we can not only reduce the hardware overhead but also avoid some unnecessary decompression/compression delay. The reason we still keep an encoder at each memory node is because, we strive to maintain the flexibility of our compression algorithm, so that both processors and memories can *selectively* compress certain packets, as will be illustrated in the following subsection. Note that, we do not opt to increase the effective memory capacity as in conventional memory compression schemes. Our design still allocates a fixed block size (e.g. 64 bytes) in the DRAM, even if the data block is compressed. The space saved due to compression can be used for other usage such as error correction [42], which is out of the scope of this paper.

Since we make no attempt to save capacity, every cacheline has the same starting address as an uncompressed memory, thus there is no complication in locating a block. Then we just need meta-data to store the (rounded) length of the cacheline. We can use 3 bits per 512-bit (64B) cacheline to indicate its length after compression, with a granularity of 8B so that "111" ((7+1)*8B=64B) means an uncompressed cacheline. For example, if a cacheline is compressed into 26B, its meta-data will be "011". The additional physical memory to store the meta-data is thus 3/512=0.6% overhead.

2) Distance-Aware Selective Compression

Due to frequent memory access, the energy and delay overhead associated with the compression/decompression operations is undesirable. Therefore, instead of letting data go through the compression engine all the time, we explore a *selective* compression algorithm to pick a subset of network packets for compression.

It is straightforward that the decision of performing compression or not depends on the *compression ratio*. However, the exact compression ratio of each packet is not known as a priori, before entering the compression engine. Another related metric is *distance*, or hop count (H). The higher the hop count is, the more energy and latency saving a compressed data block can achieve. The dynamic energy saving (ΔE) of a compressed packet can be modeled as:

$$\Delta E = E_b * L_p * \lambda * H - E_{comp} \quad (1)$$

Where E_b is the energy per bit per hop (including a link and a switch), L_p is the length of the original uncompressed packet, λ represents the compression ratio, and E_{comp} stands for the energy overhead of compression.

The latency saving (Δt) of compression can be approximated as:

$$\Delta t = t_{ser} * \lambda * H - t_{comp} \quad (2)$$

Where t_{ser} is the serialization latency of an uncompressed packet, and t_{comp} is the compression delay.

As we can see from Equations 1 and 2, *both energy and latency savings are proportional to the compression ratio and hop count*, while other parameters depend on the hardware implementation of compression algorithms. We can obtain the average packet compression ratio by either profiling the application off-line or tracking the compression ratio online on an epoch basis. For simplicity, we use an off-line profiling scheme to avoid additional hardware overhead to track the compression process. Then, based on Equations 1 and 2, we can obtain the minimum required distance (denoted as H_0) to perform a compression operation. Figure 11 shows our distance-aware compression architecture. The distance to the destination node (H) can be obtained by checking the destination address encoded in the packet header. If H is larger than H_0, the packet will be compressed by the encoder and finally sent to the network through the network interface (NI). Otherwise the original uncompressed packet is ejected to the memory network.

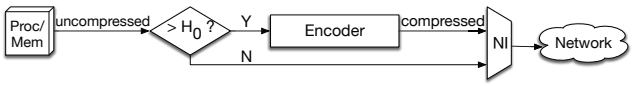


Fig. 11: A distance-aware compression architecture, which selectively compresses packets based on their distance to the destination nodes.

Back to our placement strategy of encoders/decoders as shown in Figure 10b, the processors may choose not to compress a packet when sending it to a nearby node. However, in case the data packet is requested by a remote processor, it may be compressed at the memory side before being ejected back to the memory network for traffic reduction.

B. Memory Network Power Management

In addition to the above performance optimization techniques, as illustrated in Section III-C, an efficient power-management scheme is required to deal with the power-hungry off-chip links. A straightforward solution is to aggressively shut down some under-utilized network links — power-gating. However, power-gating memory networks faces many challenges: (1) we have to assure a connected graph after turning off some links, otherwise some nodes may be isolated and require a long wake-up latency in the order of a few microseconds [43]; (2) we have to guarantee deadlock-free routing in the network with some links disabled; (3) we need to minimize the latency penalty due to adaptive routing and the corresponding hardware overhead.

Most NoC power-gating techniques [44], [45], [46], [47] do not meet requirement (1) because NoC routers/links have an affordable sleep/wake-up latency in the order of several nanoseconds. Other link power-gating techniques [48], [47] build either a power-performance graph or a spanning tree to form a connected graph, but require complex routing tables and other logic to remember the reconfigured topology for fully adaptive routing, thus fail to meet requirement (3). In contrast, **we build a customized power-gating scheme to fulfill all three requirements**, motivated by the unique pattern of the processor-memory traffic: there is no direct communication between memory nodes. Therefore, if we remove all horizontal links (dashed links as shown in Figure 12) except for the top and bottom rows, CPU to CPU/Memory traffic will not be

affected at all using existing X-Y routing, and memory to CPU traffic can still use deadlock-free minimum-path routing by slightly modifying the X-Y routing algorithm: we use two additional bits per node to indicate the connectivity of two horizontal output links. If a packet needs to go horizontally indicated by its X-offset but the corresponding link is turned off, it will go through the Y-dimension link indicated by the Y-offset instead. For example, the bold path in Figure 12 shows the routing path from Mem_5 to Processor_3 when the E_{out} links of both Mem_6 and Mem_10 are off. Theoretically 50% of the network links can be turned off if the network is infinitely large.

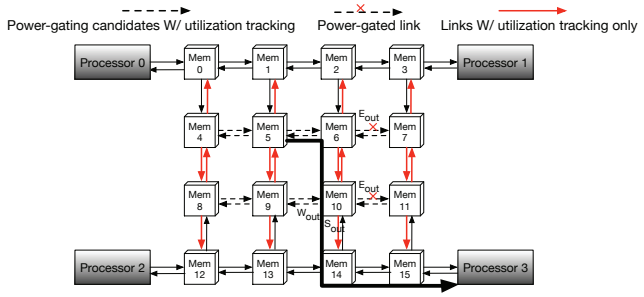


Fig. 12: Illustration of our power-gating mechanism: dashed links are power-gating candidates. All dashed and red links are coupled with utilization trackers to monitor the buffer usage for link on/off decision-making. The bold path from memory node 5 to processor 3 shows the deadlock-free routing path after E_{out} of Mem_6 and Mem_10 have been power-gated.

Now that we have identified the links that can be power-gated while meeting all three requirements, the remaining challenge is to dynamically turn links on and off, in order to maximize power savings without causing over-congestion in the border or vertical active links. Here, we use buffer utilization (U_{buffer}) to monitor the usage of an input port:

$$U_{buffer} = \frac{\sum_{t=1}^{t_{sw}} (F(t)/B)}{t_{sw}}, 0 \leq U_{buffer} \leq 1 \quad (3)$$

where $F(t)$ is the number of input buffers that are occupied at time t, B is the input buffer size, t_{sw} is the sampling window size in clock cycles. The sleep latency of a link is set to 680 ns according to the specification [43], while its wake-up latency is conservatively assumed to be 5 μs [20]. To minimize the performance impact of state-transition of these links, we set the sample window to be 20$\times$ of the wake-up latency, i.e. 100 μs. Even during the transition time, requests can still be served by other enabled links without running into deadlock. Furthermore, we use two threshold levels (α_{low} and α_{high}) to indicate the low and high utilization of the input port:

$$0 \leq \alpha_{low} \ll \alpha_{high} \leq 1 \quad (4)$$

The threshold levels are application-dependent. Here we conduct a simple off-line profiling to determine the lowest and highest buffer utilization in the network, then set α_{low} and α_{high} as the 25% and 75% quantile, respectively. All the power-gating candidates (dashed links) will be monitored for the buffer utilization, and when U_{buffer} falls below the threshold α_{low}, the corresponding input link will be shut down. To decide when those power-gated links should be turned on, we monitor the red links as in Figure 12: if U_{buffer} exceeds the threshold α_{high}, the corresponding two horizontal output links of its **upstream router** will be simultaneously powered on. We use a simple example in Figure 12 to explain this methodology: if both W_{out} and E_{out} of Mem_10 are on, Mem_11 to Processor_2 traffic will use W_{out} of Mem_10, Mem_9 to Processor_3 traffic will use E_{out} of Mem_10, and Mem_5 to Processor_3 traffic shown in the figure will also use E_{out} of Mem_10. However, if either W_{out} or E_{out} is off, S_{out} of Mem_10 will be utilized instead. Thus, if we detect congestion in S_{out}, its upstream node Mem_10 will turn both W_{out} and E_{out} on if they are in off state.

VI. Experiments

In this section, we describe the simulation and workload setup to evaluate different processor-memory network designs. Then we report the performance, energy, and hardware evaluations of these design implementations.

A. Simulator Setup

We implement an in-house C++ based *event-driven* simulator (15,000 lines of code) from scratch to model the processor-memory networks in a multi-socket server system. The backbone of the simulator models an interconnect network which supports a full spectrum of topologies, routing, flow-control, and allocation algorithms. It is also able to model different endpoints that are attached to the network switches. Specifically, the processor nodes are modeled as active terminals that issue memory requests based on the specified traffic patterns. The memory nodes are modeled as passive terminals which only reply to read/write requests. Table I shows the basic configurations of CPU, memory, and interconnect. In our evaluation, we model a 4-socket server system with 16 HMC-like memory nodes (4GB each). For a fair comparison, we keep the processor bandwidth constant for both traditional CPU-centric network and our proposed memory network. Specifically, we assume a total of 256 high-speed lanes connecting the processor and its nearby memory nodes, with each lane operating at 10Gbps. As for intra-memory NoC, we use a classic router architecture with 4-stage pipeline and 4 virtual channels (VCs) per port. A network packet consists of 4 flits and each flit is 16-byte long. Additionally, we accommodate the SerDes delay (1.6 ns for each of serialization and deserialization) and compression/decompression delay (1 cycle for each).

The simulator keeps a well-structured class hierarchy that allows creation of heterogeneous radixes of routers to form various network architectures. Table II describes various processor-memory network designs we have implemented.

B. Workloads Setup

To evaluate various network designs, we implement different synthetic traffic patterns in our simulator: *uniform-random*,

TABLE I: System (CPU, memory, and interconnect) configurations

CPU & Memory	4 sockets; 2GHz; 64B cache-line size; 16 memory nodes; 4GB per node; 100 cycles memory latency
Off-chip CPU-to-Memory channel	256 lanes in total (128 input lanes and 128 output lanes); 10Gb/s per lane
On-chip intra-memory network	4-stage router pipeline; 4 VCs per port; 4 buffers per VC; flit size = 16B; maximum packet size = 4 flits
Additional delay	3.2ns SerDes lantecy (1.6ns each); 2 cycles compression/decompression delay (1 cycle each)

TABLE II: Comparisons of different processor-memory network designs

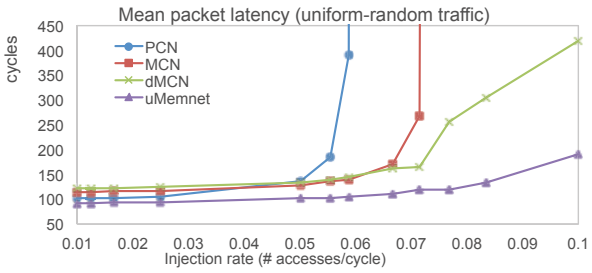
PCN	Processor-centric network, as shown in Figure 1a, which employs conventional system interconnect for cross-socket communication.
MCN	Memory-centric network [20], which uses a mesh topology to connect all the memory modules W/ Xbar NoC
dMCN	Distributed Memory-centric network [20]. Extra links are added to connect a processor to multiple nearby memory nodes W/ Xbar NoC
uMemnet	Our proposed unified intra- and inter-memory network, as shown in Figure 8, W/ reduction/dispersion tree based NoC
uMemnet+Comp	uMemnet with compression/decompression support.
uMemnet+Comp+PG	uMemnet with compression/decompression support and power management.

TABLE III: Different synthetic traffic patterns

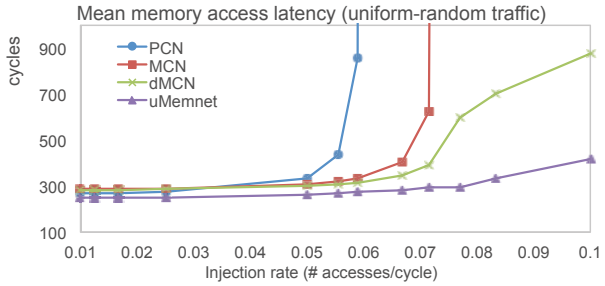
Uniform-Random	Every processor sends an equal amount of traffic to every memory node.
Hotspot	Every processor sends a large portion (e.g. 50%) of traffic to a single memory node. The rest of the traffic distribution is uniform-random.
Local-remote-ratio	The number of local memory access v.s. remote memory access is controlled by a ratio.

TABLE IV: Real in-memory computing workload characteristics

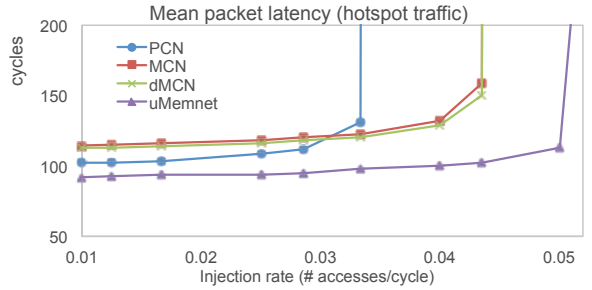
Spark-wordcount	A "wordcount" job running on Spark, which counts the number of occurrences of each word in the Wikipedia dataset provide in BigDataBench [2].
Spark-grep	A "grep" job running on Spark, which extracts matching strings from text files and counts how many times they occurred with the Wikipedia dataset provide in BigDataBench [2].
Spark-sort	A "sort" job running on Spark, which sorts values by keys with the Wikipedia dataset provide in BigDataBench [2].
Pagerank	A measure of Twitter influence. From the graph analysis benchmark in CloudSuite [3]. Twitter dataset with 11M vertices. Data set size 1.3GB, and GraphLab requires 4GB heap memory
Redis	An in-memory database system which simulates running 50 clients at the same time sending 100,000 total queries [49].
Memcached	From CloudSuite [3], which simulates the behavior of a Twitter caching server using the Twitter dataset with 8 client threads, 200 TCP/IP connections, and a get/set ratio of 0.8.



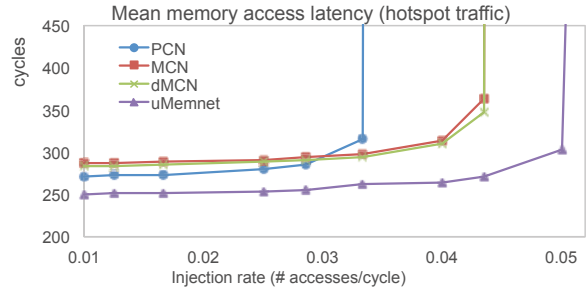
(a) Average packet latency with **uniform-random** traffic



(b) Average memory access latency with **uniform-random** traffic



(c) Average packet latency with **hotspot** traffic

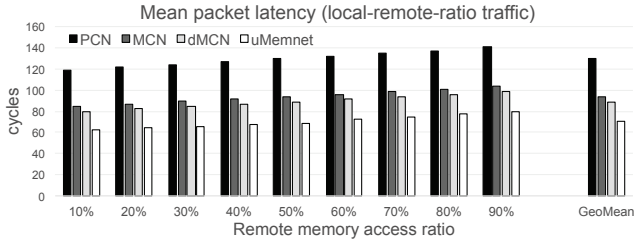


(d) Average memory access latency with **hotspot** traffic

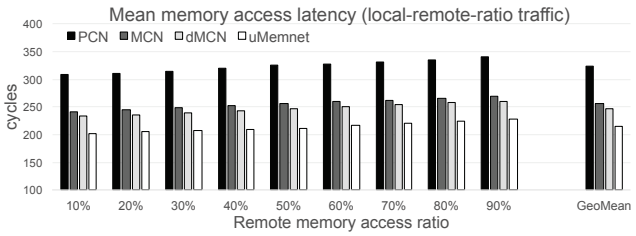
Fig. 13: Comparison of average packet latency and memory access latency with *uniform-random* and *hotspot* traffic.

hotspot, and *local-remote-ratio*, as described in Table III. In addition, we build our simulator to be trace-driven for real workload evaluation. We run various in-memory computing workloads from CloudSuite [3], BigDataBench [2], and Redis benchmark [49] on a Dell PowerEdge T630 server. Spark 1.4.1 [50] is used for all of the Spark jobs. Table IV summarizes the workload and dataset characteristics.

We use a Pin-based [51] functional simulator to collect instruction and memory traces when running these workloads. The cache hierarchy in the Pin tool employs a 32KB L1, 2MB L2, and 32MB L3 with associativities of 16, 8, and 4, respectively. Note that the Pin tool does not contain a



(a) Average packet latency with **local-remote-ratio** traffic



(b) Average memory accesss latency with **local-remote-ratio** traffic

Fig. 14: Comparison of average packet latency and memory access latency with *local-remote-ratio* traffic.

detailed core model and thus we can only obtain the absolute instruction ID of each memory access. However, we can multiply the instruction IDs by an average CPI number (we call it a **multiplier**) and generate a timestamp for each memory access. For sensitivity study, we can sweep the value of multiplier to generate different memory access intensities. As a result, our trace files contain a sequence of tuples in the format of `<timestamp, address, read_or_write, data>`, which are further fed into our event-driven simulator for performance evaluation.

C. Performance Evaluation

We use the simulator to evaluate the memory network designs with both synthetic traffic and real workload traces.

1) Synthetic Traffic Evaluation

Figure 13 shows the average packet latency and average memory access latency for two synthetic traffic patterns: *uniform-random* and *hotspot*, as we sweep the injection rates of memory access.

• Compared to prior memory-centric network designs (MCN and dMCN), the conventional PCN design achieves lower packet and memory access latency at very low injection rates, but soon gets saturated. This is especially true for *hotspot* traffic, in which PCN gets saturated easily at a low injection rate of around 0.03 accesses/cycle.

• The dMCN design reduces the network latency with extra links to connect a processor to multiple memory modules, but the improvement is negligible for *hotspot* traffic.

• Our unified memory network (uMemnet) design achieves the best performance at all times. There is only incremental change to the end-to-end latency at *uniform-random* traffic. As for *hotspot* traffic, the saturation point is effectively delayed compared to other designs. Such significant latency reduction is mainly owing to our light-weight intra-memory network design with efficient I/O placement, which avoids clogging inter-memory network traversals.

Additionally, we evaluate different processor-memory network designs with the *local-remote-ratio* traffic pattern, in which we fix the injection rate of processors to be 0.05 and sweep the ratio of local and remote memory access. For example, in Figure 8a, the nearest four memory nodes (0, 1, 4, and 5) to processor 0 are treated as local memory nodes. As shown in Figure 14, when the percentage of remote memory traffic increases, the mean packet latency and memory access latency increase for all network designs. On average, compared to the PCN baseline, the MCN and dMCN designs achieve 27.6% and 31.3% packet latency reduction, which further leads to 20.9% and 23.6% memory access latency reduction. Finally, our proposed uMemnet design can reduce the average packet latency by 45.8%, and average memory access latency by 33.8%.

2) Real Workloads Evaluation

We further evaluate different processor-memory network designs with the real workload traces we obtained. By sweeping the value of CPI (multiplier), we can control the intensity of memory accesses. For example, here we sweep the multiplier from 60 to 30 to generate an increasing load of traffic. We highlight several important observations based on the results shown in Figure 15 and Figure 16:

• The conventional PCN design is not scalable as we increase the memory access intensity, with the average packet latency and average memory access latency increasing rapidly due to severe network congestion.

• The MCN design enhances memory performance, but still suffers from significant network delay as the traffic load increases, which further causes severe memory access delay. The distributed network design (dMCN) does not demonstrate much performance improvement over MCN, even exhibiting small performance degradation in many cases.

• With our proposed uMemnet, both intra- and inter-memory traversals are accelerated. As a result, both the average packet latency and memory access latency are reduced significantly. However, we still observe a notable latency increase when the issue interval (multiplier) shrinks to 30. On average, uMemnet achieves 82.2% packet latency reduction and 72.1% average memory access latency saving.

• With network compression (we use delta compression [38], [39]) integrated with uMemnet, the average latency numbers are further reduced. Moreover, the steep latency increase due to the increased memory load is eliminated, which indicates that network congestion is effectively mitigated by packet compression. On average, the packet latency and memory access latency are reduced by 21.5% and 25.7% respectively, compared to the uMemnet design.

• Power-gating unavoidably causes performance degradation but only incurs 6.6% packet latency increase and 4.1%

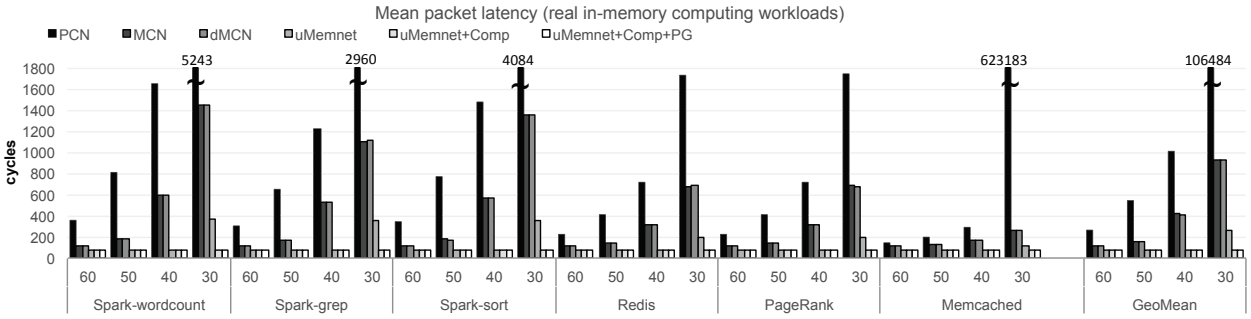


Fig. 15: Comparison of average packet latency with different in-memory computing workloads

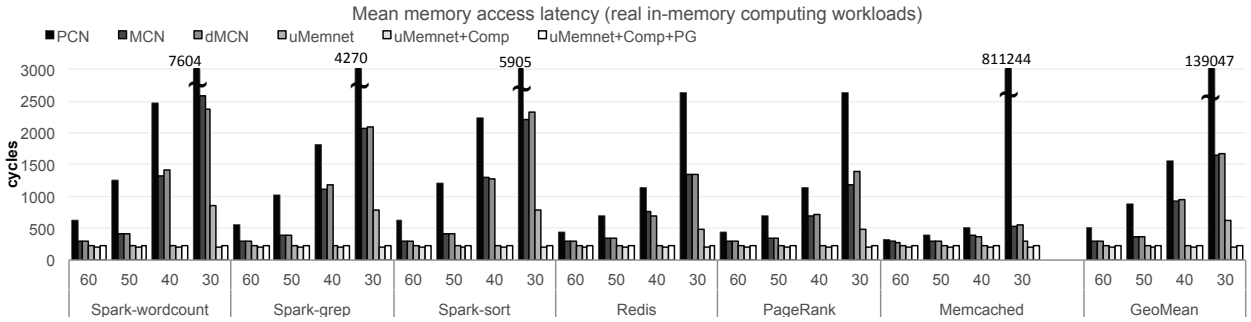


Fig. 16: Comparison of average memory access latency with different in-memory computing workloads

memory access latency increase, compared to the uMemnet design with compression support. With all techniques combined, we can achieve 83.5% packet latency reduction and 75.1% memory access latency reduction compared to the PCN baseline. The energy benefit of our power management will be described in the following subsection.

D. Energy Evaluation

Here we evaluate our processor-memory network design from an energy perspective. We use Micron's DDR4 System-Power Calculator [52] to calculate the power consumption of the DRAM layers. For the logic layer, we use McPAT [53] to get the power consumption of the memory controllers, assuming 16 vault MCs in total, and use DSENT [54] to compare the power consumption of the various intra-memory NoC configurations. Furthermore, energy consumption of a link is modeled as 2.0 pJ/bit [20], [55].

We choose a moderate memory access intensity (multiplier = 50) and compare the energy consumption of the memory fabric in 32nm technology. As we can see from Figure 17a, compared to the dMCN design, our uMemnet can achieve 17.7% energy saving with less redundant off-chip links and lightweight intra-memory network. With compression support, the number of packet accesses in the network will be reduced, which saves a total of 21.4% network energy compared to the uMemnet design. The energy reduction with compression is not very significant as the static energy remains largely unoptimized. Finally, with our link-level power-gating technique, compared to the dMCN design, we can effectively achieve 35.3% network energy reduction. Furthermore, we report the energy consumption of the entire memory system in Figure 17b, which shows that our processor-memory network design with all techniques combined can achieve 22.1% memory energy saving.

E. Hardware Evaluation

Table V reports the area and power of various intra-memory NoC designs by modifying DSENT [54]. Overall, our unified network with a reduction/dispersion tree based intra-memory NoC can achieve 60.3% area saving and 70.7% power reduction compared to the decoupled network design.

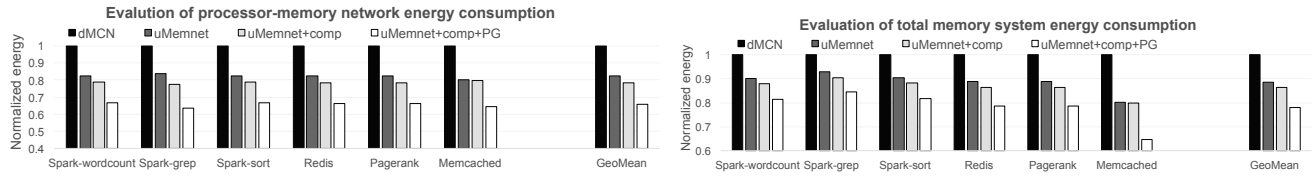
TABLE V: Power and area comparision of different intra-memory NoC

	Area(mm^2)	Power(W)
Decoupled network W/ Xbar NoC	3.98	2.34
Unified network W/ Xbar NoC	3.83	2.27
Unified network W/ mesh NoC	2.40	1.04
Unified network W/ tree NoC	1.58	0.69

Additionally, we leverage delta compression [38], [39] to compress network traffic. We verify our encoder/decoder design by implementing the modules in behavioral Verilog, creating a testbench, and simulating using Mentor Graphics Modelsim. We further investigate the power and area overhead by synthesizing our Verilog using the Synopsys Design Compiler, with 45nm technology node. Table VI shows the area and power of the encoders/decoders. In total, they only add $0.1017mm^2$ area, and $0.032W$ power consumption.

TABLE VI: Power and area overhead for delta compression

Area(mm^2)			Power(W)		
Encoder	Decoder	Total	Encoder	Decoder	Total
0.00474	0.00173	0.1017	1.47e-3	0.81e-3	0.032



(a) Total processor-memory network energy

(b) Total memory system energy

Fig. 17: Comparisons of energy consumption with different processor-memory network designs

Furthermore, we synthesize our customized power-gating scheme to determine the area overhead. Given 100 μs sample window, 10 Gbps transfer rate per lane, and 16 buffers per port, we need a 24-bit register and adder to track the buffer utilization. Since there are 4 ports per node and two additional bits to indicate the connectivity of two horizontal links, we obtain the total area overhead per node to be 0.00292 mm^2 at most (only those dashed and red links in Figure 12 need to be monitored), which is a negligible 0.19% overhead compared to our intra-memory NoC. Combining all above hardware modifications, we can achieve 57.6% area saving in the logic layer compared to the design with decoupled network.

VII. Related Work

Memory Fabric: As the data rate of DRAM increases, the number of DIMMs that can be supported on a conventional multi-drop bus decreases. Alternatively, multiple DIMMs can be connected using point-to-point links like the Fully-Buffered DIMM design [56], but causes significant latency and power overhead to traverse intermediate advanced memory buffers (AMBs). There are also other DIMM tree [57], [58] and ring [59] structures, but do not support disaggregated memory pool architectures, and may only work with RF interconnect or silicon photonics technologies. Initial efforts on memory-centric network [20], [21] are mainly for memory bandwidth benefits in desktop applications, which lack intra-memory network optimization, congestion avoidance, and power management. In contrast, we target at in-memory computing workloads, and provide a unified processor-memory network with efficient network compression and link-level power-gating.

Memory Compression: Memory compression [40], [41], [38] increases the effective memory capacity by storing data in compressed format. However, the internal memory structure needs to be heavily modified in order to support the variable block sizes, and requires complex logic to locate the data in memory. In contrast, our memory network compression strives to reduce network traffic for bandwidth and energy benefits without attempting to save memory capacity.

Network Compression: Unlike NoC compression [37], [60], [39], [61] which usually deals with homogeneous tiled on-chip multi-core architecture with symmetric cache traffic, memory network compression needs to be designed for the asymmetric off-chip processor-to-memory traffic, which exhibits unique traffic pattern and requires a different compression architecture. Also different from the simple memory channel compression [62], [63], [42] with a single CPU and memory node, memory network compression deals with enormous number of data access in a pool of discrete memory modules with variable distances.

Network Power Gating: Power gating [64] has been recently explored in the NoC domain [44], [45], [46], [47], [65], [66], [67] to aggressively shut down idle switches or links for energy savings, but their schemes do not apply to the off-chip domain due to the long sleep/wake-up overhead. Recently Ahn *et al.* [55] explore dynamic lane on-off schemes inside a single CPU-to-HMC channel. In contrast, we explore power-gating at a memory network scale with a pool of discrete memory nodes, which incurs new challenges in traffic monitoring and also opportunities of link-level power-gating due to unique traffic patterns. Moreover, we generalize the power management scheme without making HMC-specific assumptions.

Near-Data Computing: While we focus on in-memory computing which mainly requires a large and fast main memory to accommodate the entire dataset, there are orthogonal studies on near-data computing [68], [69], [70], [71] which move the computation close to memory or even in memory (the so-called processing-in-memory) to reduce data movement. We do not make strong assumptions to integrate complex compute logic in memory, but instead only add simple routing/compression logic for efficient communication.

VIII. Conclusion

In this work, we analyze the design of memory network to support in-memory computing. We study the inefficiencies in previous proposals for such architectures, and propose the co-optimization of intra- and inter-memory network. For intra-memory network, we re-organize the NoC topology and provide a smart I/O interface to enable efficient coordination of intra- and inter-memory network, thus forming a unified memory network architecture. For inter-memory network, we propose a distance-aware network compression architecture and a link-level power-gating algorithm, all based on the unique traffic patterns. Experimental results show that our holistic design of process-memory network can reduce the memory access latency by 75.1% for real in-memory computing workloads, while achieving 22.1% total memory energy reduction on average.

Acknowledgment

This work was supported in part by NSF 1533933, 1461698, 1213052, and 1500848.

References

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," 2012.

[2] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: a big data benchmark suite from internet services," in *HPCA*, pp. 488–499, IEEE, 2014.

[3] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 37–48, 2012.

[4] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ISCA*, IEEE, 2015.

[5] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[6] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[7] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang, "DB2 with BLU acceleration: So much more than just a column store," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1080–1091, 2013.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*, vol. 10, p. 10, 2010.

[9] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *SoCC*, pp. 1–15, ACM, 2014.

[10] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *ASPLOS*, pp. 3–18, ACM, 2014.

[11] A. Daglis, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, "Manycore Network Interfaces for In-Memory Rack-Scale Computing," in *ISCA*, pp. 567–579, ACM, 2015.

[12] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *Supercomputing Frontiers and Innovations*, vol. 1, no. 3, 2015.

[13] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling Big-memory Computing with Hardware-based Memory Expansion," *TACO*, vol. 9, no. 4, 2015.

[14] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *HOTI*, pp. 1–6, IEEE, 2010.

[15] "HyperTransport I/O technology overview," *HyperTransport Consortium, June*, 2004.

[16] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hotchips*, vol. 23, pp. 1–24, 2011.

[17] "The Machine: A new kind of computer." http://www.hpl.hp.com/research/systems-research/themachine/.

[18] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," pp. 267–278, 2009.

[19] H. T. C. Ltd, "High Throughput Computing Data Center Architecture: Thinking of Data Center 3.0," *Technical White Paper*, 2014.

[20] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *PACT*, pp. 145–156, IEEE, 2013.

[21] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-GPU system design with memory networks," in *MICRO*, pp. 484–495, IEEE, 2014.

[22] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[23] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project http://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf*, 2008.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, pp. 2–2, USENIX Association, 2012.

[25] L. McVoy and C. Staelin, "lmbench: portable tools for performance analysis," in *USENIX ATC*, pp. 23–23, USENIX Association, 1996.

[26] G. H. Loh, "3D-stacked memory architectures for multi-core processors," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 453–464, 2008.

[27] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, "Bridging the processor-memory performance gap with 3D IC technology," *IEEE Design & Test of Computers*, vol. 22, no. 6, pp. 556–564, 2005.

[28] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner, "PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 117–128, 2006.

[29] G. L. Loi, B. Agrawal, N. Srivastava, S.-C. Lin, T. Sherwood, and K. Banerjee, "A thermally-aware performance analysis of vertically integrated (3-D) processor-memory hierarchy," in *DAC*, pp. 991–996, ACM, 2006.

[30] R. Nair, S. Antao, C. Bertolli, P. Bose, J. Brunheroto, T. Chen, C. Cher, C. Costa, J. Doi, C. Evangelinos, B. Fleischer, T. Fox, D. Gallo, L. Grinberg, J. Gunnels, A. Jacob, P. Jacob, H. Jacobson, T. Karkhanis, C. Kim, J. Moreno, J. O'Brien, M. Ohmacht, Y. Park, D. Prener, B. Rosenburg, K. Ryu, O. Sallenave, M. Serrano, P. Siegl, K. Sugavanam, and Z. Sura, "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17–1, 2015.

[31] "NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data." http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/.

[32] B. Towles, J. Grossman, B. Greskamp, and D. E. Shaw, "Unifying on-chip and inter-node switching within the Anton 2 network," in *ISCA*, pp. 1–12, IEEE, 2014.

[33] D. E. Shaw, J. P. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton, A. Forte, J. Gagliardo, G. Gill, B. Greskamp, C. R. Ho, D. J. Ierardi, L. Iserovich, J. S. Kuskin, R. H. Larson, T. Layman, L.-S. Lee, A. K. Lerer, C. Li, D. Killebrew, K. M. Mackenzie, S. Y.-H. Mok, M. A. Moraes, R. Mueller, L. J. Nociolo, J. L. Peticolas, T. Quan, D. Ramot, J. K. Salmon, D. P. Scarpazza, U. Ben Schafer, N. Siddique, C. W. Snyder, J. Spengler, P. T. P. Tang, M. Theobald, H. Toma, B. Towles, B. Vitale, S. C. Wang, and C. Young, "Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *SC*, pp. 41–53, IEEE, 2014.

[34] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "NOC-Out: Microarchitecting a scale-out processor," in *MICRO*, pp. 177–187, IEEE, 2012.

[35] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-chip Inteconnection Networks," in *DAC*, pp. 684–689, ACM, 2001.

[36] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[37] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. S. Yousif, and C. R. Das, "Performance and power optimization through data compression in network-on-chip architectures," in *HPCA*, pp. 215–225, IEEE, 2008.

[38] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *PACT*, pp. 377–388, ACM, 2012.

[39] J. Zhan, M. Poremba, Y. Xu, and Y. Xie, "NoΔ: Leveraging delta compression for end-to-end memory access in NoC based multicores.," in *ASP-DAC*, pp. 586–591, 2014.

[40] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *MICRO*, pp. 258–265, ACM, 2000.

[41] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *ISCA*, pp. 212–223, IEEE, 2004.

[42] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *HPCA*, pp. 638–649, IEEE, 2014.

[43] "Hybrid memory cube specification 1.0," *Technical report, Hybrid Memory Cube Consortium*, 2013.

[44] A. Samih, R. Wang, A. Krishna, C. Maciocco, C. Tai, and Y. Solihin, "Energy-efficient interconnect via router parking," in *HPCA*, pp. 508–519, IEEE, 2013.

[45] L. Chen and T. M. Pinkston, "Nord: Node-router decoupling for effective power-gating of on-chip routers," in *MICRO*, pp. 270–281, IEEE Computer Society, 2012.

[46] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," in *ISCA*, pp. 320–331, ACM, 2013.

[47] R. Parikh, R. Das, and V. Bertacco, "Power-aware NoCs through routing and topology reconfiguration," in *DAC*, pp. 1–6, IEEE, 2014.
[48] V. Soteriou and L.-S. Peh, "Dynamic power management for power optimization of interconnection networks using on/off links," in *HOTI*, pp. 15–20, IEEE, 2003.
[49] "Redis Benchmark." http://redis.io/topics/benchmarks.
[50] "Spark 1.4.1." http://spark.apache.org/downloads.html.
[51] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation," in *MICRO*, pp. 81–92, IEEE Computer Society, 2004.
[52] "Micron System Power Calculator." hthttp://www.micron.com/support/power-calc.
[53] S. Li, J. H. 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*, pp. 469–480, IEEE, 2009.
[54] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *NoCS*, pp. 201–210, IEEE, 2012.
[55] J. Ahn, S. Yoo, and K. Choi, "Dynamic Power Management of Off-Chip Links for Hybrid Memory Cubes," in *DAC*, pp. 1–6, ACM, 2014.
[56] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob, "Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling," in *HPCA*, pp. 109–120, IEEE, 2007.
[57] K. Therdsteerasukdi, G.-S. Byun, J. Ir, G. Reinman, J. Cong, and M. Chang, "The DIMM tree architecture: A high bandwidth and scalable memory system," in *ICCD*, pp. 388–395, IEEE, 2011.
[58] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated Control for Energy-efficient and Heterogeneous Memory Systems," in *HPCA*, pp. 424–435, IEEE, 2013.
[59] A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Combining Memory and a Controller with Photonics Through 3D-stacking to Enable Scalable and Energy-efficient Systems," in *ISCA*, pp. 425–436, ACM, 2011.
[60] Y. Jin, K. H. Yum, and E. J. Kim, "Adaptive data compression for high-performance low-power on-chip networks," in *MICRO*, pp. 354–363, IEEE Computer Society, 2008.
[61] P. Zhou, B. Zhao, Y. Du, Y. Xu, Y. Zhang, J. Yang, and L. Zhao, "Frequent value compression in packet-based NoC architectures," in *ASP-DAC*, pp. 13–18, IEEE, 2009.
[62] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-link compression schemes: A value locality perspective," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 916–927, 2008.
[63] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *PACT*, pp. 325–334, ACM, 2012.
[64] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories," in *ISLPED*, pp. 90–95, ACM, 2000.
[65] D. DiTomaso, A. Kodi, and A. Louri, "QORE: A fault tolerant network-on-chip architecture with power-efficient quad-function channel (QFC) buffers," in *HPCA*, pp. 320–331, IEEE, 2014.
[66] J. Zhan, Y. Xie, and G. Sun, "NoC-sprinting: Interconnect for fine-grained sprinting in the dark silicon era," in *DAC*, pp. 1–6, IEEE, 2014.
[67] J. Zhan, J. Ouyang, F. Ge, J. Zhao, and Y. Xie, "DimNoC: A dim silicon approach towards power-efficient on-chip network," in *DAC*, pp. 1–6, IEEE, 2015.
[68] S. 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 *ISPASS*, pp. 190–200, IEEE, 2014.
[69] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA*, pp. 283–295, IEEE, 2015.
[70] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-memory Analytics Frameworks," in *PACT*.
[71] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, pp. 105–117, ACM, 2015.