String Figure: A Scalable and Elastic Memory Network Architecture

Matheus Almeida Ogleari*,⋆, Ye Yu†, Chen Qian*, Ethan L. Miller*,‡, Jishen Zhao⋆

*University of California, Santa Cruz †University of Kentucky ‡Pure Storage ⋆University of California, San Diego
*{mogleari,cqian12,elm}@ucsc.edu †ye.yu@uky.edu ⋆{maogleari,jzhao}@eng.ucsd.edu

***Abstract*—Demand for server memory capacity and performance is rapidly increasing due to expanding working set sizes of modern applications, such as big data analytics, in-memory computing, deep learning, and server virtualization. One promising techniques to tackle this requirements is memory networking, whereby a server memory system consists of multiple 3D die-stacked memory nodes interconnected by a high-speed network. However, current memory network designs face substantial scalability and flexibility challenges. This includes (1) maintaining high throughput and low latency in large-scale memory networks at low hardware cost, (2) efficiently interconnecting an arbitrary number of memory nodes, and (3) supporting flexible memory network scale expansion and reduction without major modification of the memory network design or physical implementation.**

To address the challenges, we propose String Figure[1], a high-throughput, elastic, and scalable memory network architecture. String Figure consists of (1) an algorithm to generate random topologies that achieve high network throughput and near-optimal path lengths in large-scale memory networks, (2) a hybrid routing protocol that employs a mix of computation and look up tables to reduce the overhead of both in routing, (3) a set of network reconfiguration mechanisms that allow both static and dynamic network expansion and reduction. Our experiments using RTL simulation demonstrate that String Figure can interconnect over one thousand memory nodes with a shortest path length within five hops across various traffic patterns and real workloads.

***Keywords*-Memory fabric; 3D die-stacked DRAM; memory network; scalability; reconfiguration; routing; memory centric**

INTRODUCTION

The volume of data has skyrocketed over the last decade, growing at a pace comparable to Moore's Law [1]. This trend drives the popularity of big data analytics [2], [3], in-memory computing [4], [5], deep learning [6], [7], [8], and server virtualization [2], which are frequently insatiable memory consumers. As a result, these applications demand a continuous expansion of server memory capacity and bandwidth to accommodate high-performance working data access. As shown in Figure 1, cloud server memory capacity has been rapidly growing since the debut of cloud service for in-memory computing [9], [10], [11].

Unfortunately, DRAM capacity scaling falls far behind the pace of the application demand with current DDRx based architectures [12]. One conventional solution to increasing server memory capacity is to add more CPU sockets to maintain additional memory channels. In commodity systems, each processor can support up to 2TB memory. As a result, EC2 1Xe adopts four CPU sockets to accommodate the 4TB memory capacity [9]. Azure S960 servers can adopt 20 Intel Xeon processors to accommodate 20TB of memory [11]. However, purchasing extra CPU sockets substantially increases the total system cost [13], [14]. The extra hardware cost adds significant, often nonlinear overheads to the system budget, making this solution unsustainable.

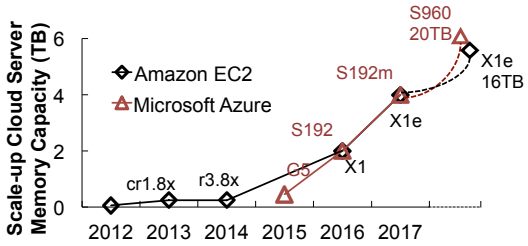


Figure 1. Expanding server memory demand.

A promising technique to tackle these memory challenges is memory networking, whereby the server memory system consists of multiple 3D die-stacked memory nodes interconnected by a high-speed network. The interconnected memory nodes form a disaggregated memory pool shared by processors from different CPU sockets in the server (Figure 2). Ideally, the memory network can enable more scalable performance and capacity than traditional DDRx-based memory systems, as shown by academia [15], [14], [16] and industry efforts from HPE [17] and IBM [18]. However, the memory network scalability relies on the scalability of the memory fabric that interconnects the memory nodes [16], [14].

Previous memory network designs investigated platforms with limited numbers of memory nodes. Scalability and flexibility were not considered in their the design goals. Given that each memory node (a 3D memory stack) can offer 8GB capacity [14], state-of-the-art 4TB server memory system requires 512 memory nodes. Recent works have proposed optimizing NoC network topologies to support up to 128 memory nodes in integrated CPU+GPU (APU) systems [16]. However, the design partitions clusters of memory nodes to separate channels, where each processor

[1]String Figure is a game formed by connecting strings between fingers of multiple people. Our memory network topology appears like a string figure formed using memory nodes.

can only access a subset of the memory space. State-of-the-art planar topologies [19], [20] offer high network throughput at a large scale. However, the number of required router ports and links increases as the network size grows, which imposes undesirable cost in routers integrated with memory nodes. The challenges of scaling up server memory capacity still remain.

The **goal** of our paper is to design a high-performance, scalable, and flexible memory network architecture that can support over a thousand interconnected memory nodes shared by processors in a cloud server. We elaborate our design goal as follows:

- **Scalability.** We need to support over a thousand memory nodes shared by all CPU sockets in a server. As such, the scalability requirement is three-fold. **Path lengths:** When the network size (i.e., the number of memory nodes) grows, the routing path lengths need to grow sublinearly. **Routing overhead:** The computation and storage overheads of routing decision-making need to be sublinear or independent of the network scale. **Link overhead:** The number of required router ports and links is also either sublinear or independent of the network size.
- **Arbitrary network scale.** We need to maintain a high-throughput interconnect of an arbitrary number of memory nodes, without the shape balance limitation of traditional topologies, such as a mesh or tree.
- **Elastic network scale.** We need to support flexible expansion and reduction of the network size, in terms of the number of memory nodes. The elastic network scale allows effective power management by turning on and off routers and corresponding links. It also enables design and implementation reuse across various server system configurations.

To achieve our goals, we propose String Figure, a scalable and elastic memory network architecture that consists of three design components. First, we propose a network construction scheme to efficiently generate random network topologies that interconnect an arbitrary number of memory nodes with high network throughput, near-optimal path lengths, and limited router ports. Second, we develop an adaptive greediest routing protocol, which significantly reduces the computation and storage overhead of routing in each router. Finally, we propose a network reconfiguration scheme, which allows the network scale, topology, and routing to change according to power management and design reuse requirement. Performance and energy experiments show that String Figure can interconnect up to 1296 memory nodes with significantly higher throughput and energy efficiency than previous designs, across various of synthetic and real workloads.

Background and Motivation

In this section, we briefly discuss the limitations of conventional DDRx-based memory systems, opportunities with memory networks, and the challenges of scalable memory network design.

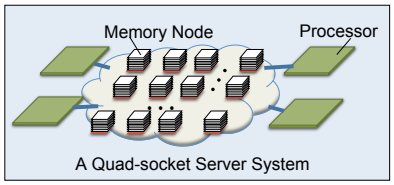


Figure 2. A disaggregated memory pool in a quad-socket server. Memory modules are interconnected by a memory network shared by four processors.

Limitations of Commodity Server Memory Architectures

Traditional server memory systems face substantial challenges in scaling up memory capacity due to cost and performance issues. Increasing the number of CPU sockets enables adding more memory channels with more DIMMs. However, extra CPU sockets, which are added for memory capacity rather than compute requirement, can substantially increase hardware cost in an economically infeasible manner. Prior studies show that increasing memory capacity from 2TB to 4TB by doubling the number of CPU sockets can lead to over 3× increase in server system cost [13]. Commodity clusters allow us to develop scale-out memory systems, which distribute the working set of applications across multiple server nodes [21]. However, the scale-out approach can only accommodate a limited subset of use cases with data-parallel, light communication algorithms. In remaining applications, the scale-out solution either requires programmers to rewrite their software, or system architects to adopt a low-latency interconnect fabric. This shifts the memory scaling burden to the software and communication infrastructures. However, recent studies that explored disaggregated memory system design [22], [23] require substantial changes in the virtual machine system software, such as a hypervisor. Our design is in line with recent industry and academic approaches on implementing disaggregated memory pool with memory fabric, such as Gen-Z [24] and memory-centric system integration [15], [25], [14].

Memory Network

To address the memory capacity demand challenges, recent works introduce memory network design [15], [25]. A memory network consists of a set of 3D die-stacked memory nodes interconnected by a high-speed network. Figure 2 shows an example server system with four CPU sockets attached to a memory network. The processors can be connected to any edge memory nodes in the network. As 3D die-stacking technology is maturing, various 3D die-stacked memory designs are either already used in commodity systems or close to the market [26], [27], [28], [29]. 3D die-stacked memory typically employs several DRAM dies on top of a logic die. For example, one type of die-stacked memory, Hybrid Memory Cube (HMC) [29], [30], offers 8GB capacity per memory stack with link speeds up to 30Gbps (versus 1.6Gbps supported by DDR4) and peak ag-

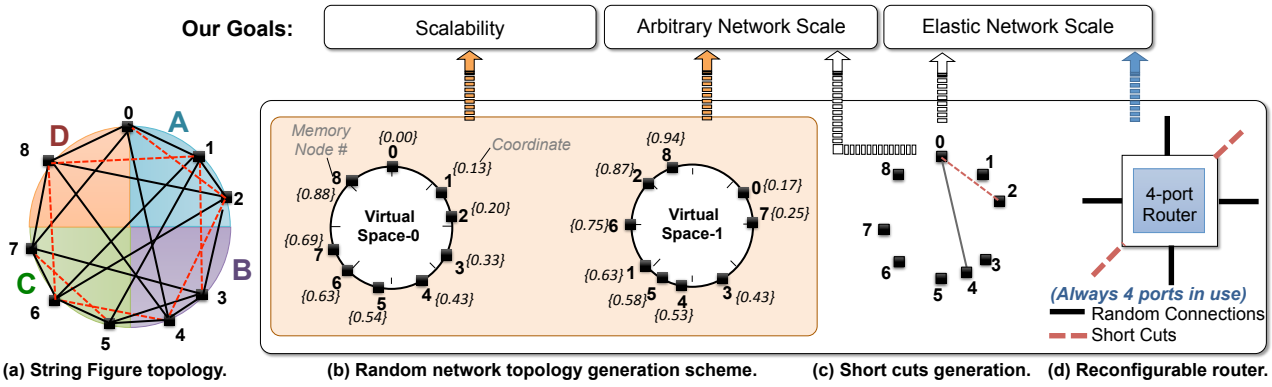


Figure 3. An example of String Figure topology design with nine memory nodes (stacks) and four-port routers. (a) String Figure topology. (b) Virtual space organization for random network generation. (c) Shortcuts generation for Node-0. (d) High-level design of a reconfigurable four-port router.

gregate bandwidth of 320GB/s/link. Recent studies show that die-stacked memory capacity is likely to scale to 16GB [16]. Memory network design has attracted both academic [25], [15], [14], [16], [31] and industry [17], [24], [18] efforts. Recent studies demonstrate that memory network can lead to more promising main memory expansion opportunities than traditional DIMM-based memory systems.

Challenges of Memory Network Design

Memory networks are constructed by interconnecting a large number of memory nodes. Therefore, performance and scalability of the interconnect network are essential to memory network performance and capacity scaling. Previous studies investigated how to best adopt traditional Network-on-Chip (NoC) topologies and protocols in memory network design, such as mesh, ring, crossbar, chain, skip list, butterfly, and tree [25], [15], [14], [16]. However, traditional memory network topologies can lead to substantial scalability challenges. In fact, most previous memory network designs can only interconnect tens of memory modes shared by all CPU sockets [25], [15], [14], [16]. In order to support terabytes of memory capacity, we need to interconnect hundreds or over one thousand memory nodes with high network throughput and low access latency. As such, we motivate our work with the following scalability and flexibility challenges that are not effectively addressed in prior works.

Network Scalability. As shown in recent studies, the shortest path lengths of various traditional memory network topologies can substantially increase with large-scale memory networks [16]. Instead, topologies used in data centers, such as Flattened Bufferfly [19], Dragonfly [20], FatTree [32], and Jellyfish [33], can offer high bisection bandwidth and short routing path lengths in large-scale networks. However, these topologies are not directly applicable to memory network due to following reasons. First, data center networks adopt stand-alone network switches with rich link and storage resources. Second, most of these topologies require continuously increased router ports as the network scales up [19], [20], [32]. Finally, routing with most data center network topologies [33] requires large routing tables to store global routing information; the forwarding state cannot be aggregated. These issues hamper the memory networks from adopting data center network topologies in memory networks because of limited storage resources in on-chip routers and the high-bandwidth memory access requirement of in-memory applications. As a result, neither traditional memory networks nor data center network topologies can efficiently meet the scalability requirement of memory networks.

Arbitrary Network Scale. Many rigid network topologies require the number of routers and memory nodes to be specific numbers, such as a power of two. This reduces the flexibility of memory system scaling. Furthermore, these constraints on the network scale can increase the upgrade cost of memory systems and limit the potential of design reuse. For example, say we have the budget to purchase one more memory node to upgrade an existing memory network. It is difficult to upgrade because the rigid network topology only allows us to add certain number of memory nodes (or none) to maintain the network scale as a power of two.

Elastic Network Size. Traditional memory systems allow users to reconfigure the memory capacity to a certain extent. For example, commodity computer systems typically reserve a certain number of memory slots for users to expand the memory capacity in the future. The same basic DIMM-based memory system designs in each DDRx generation are also shared across various design versions with different memory capacities. This flexibility allows future memory network designs to deliver cost-efficient memory system solutions. Furthermore, support for dynamic scaling up and down memory networks also enables efficient power management, by power gating off under-utilized links and idle memory nodes. Therefore, an elastic network size (static and dynamic network expansion and reduction) is a missing yet preferable feature for future memory networks.

DESIGN

Overview. To achieve our goals, we propose String Figure, a scalable, flexible, and high-performance memory network

architecture. Figure 3 depicts an overview of our design goals and components – arrows in the figure map our design components to our goals. String Figure consists of three design principles. First, we propose an easy-to-implement random network topology construction algorithm that enables a) scalable memory network interconnecting large, arbitrary number of memory nodes with arbitrary number of router ports and b) support for elastic network scale. Second, we propose a compute+table hybrid routing scheme, which reduces the computation and storage overhead of routing large-scale networks by integrating a lightweight routing table with greediest computation-based routing mechanisms. Finally, we propose a network reconfiguration scheme, which enables elastic memory network scale. Beyond achieving our scalability goals, String Figure further enables memory access pattern aware performance optimization and efficient memory network power management.

While our design is broadly applicable to a wide range of server memory systems, we will use a working example throughout this paper to make our proposal more concrete. Our working example assumes a maximum 16TB memory system that consists of 1296 interconnected 3D die-stacked memory nodes shared by four CPU sockets (Figure 2). Each memory node has one router and is 8GB, with the same capacity and memory configuration parameters adopted in previous works [16], [31], [14]. Detailed baseline system configuration is described in Table I.

Network Topology Construction Scheme

Prior studies in data center networks, such as Jellyfish [33], demonstrated that "sufficiently uniform random graphs" (i.e., graphs sampled sufficiently uniform-randomly from the space of all r-regular graphs) empirically have the scalability properties of random regular graphs [33] and achieve throughput within several percent to the upper bound on network throughput, at the scale of several thousand nodes. Such random topologies compare favorably against traditional fat-tree topologies, supporting a larger number of nodes at full throughput. However, most previous random topologies are developed for data center networks, which need to tolerate large forwarding state storage. Directly adopting these random topologies in memory networks, which is constrained by the storage capacity in routers and routing latency, can impose prohibitive scaling issues.

To address these challenges, we propose a novel network topology String Figure inspired by S2 [34] to enable scalable random topology in memory networks at low routing cost. Our topology design also enables elastic memory network scale, i.e., flexible expansion and reduction of network scale. Our topology consists of a basic balanced random topology and a set of shortcuts. The balanced random topology ensures scalability, interconnection of arbitrary number of memory nodes. The shortcuts provide extra links that maintain high network throughput, when the network

(a) ALGORITHM 1. VIRTUAL SPACES CONSTRUCTION.

input: Number of memory nodes N
Number of ports per router p
output: {Coordinate x_i in Virtual Space$_j$ | i=0..N-1, j=0..L-1}

$L = \lfloor p/2 \rfloor$
for j = 0..L-1
  for i = 0..N-1
    x_{ij} = BalancedCoordinateGen(X_j)
    X_j = sort(x_{0j}..x_{ij})
return {X_0..X_{L-1}}

(b) ALGORITHM 2. BALANCED COORDINATE GENERATION.

input: Coordinates of k memory nodes x_0..x_{k-1}
output: Coordinate of a new memory node x_n

if k = 0 **then return** RandomNumber(0, 1)
if k = 1
  then $a \leftarrow x_0$, $b \leftarrow x_0 + 1$
  else
    $D(x_{r1}, x_{r2}) = min\{|x_{r1} - x_{r2}|, 1 - |x_{r1} - x_{r2}|\}$
    find x_{r1}, x_{r2} among x_0..x_{k-1} such that
    $x_{r1} < x_{r2}$ and $D(x_{r1}, x_{r2})$ is the smallest
if $x_{r2} - x_{r1} < 1/2$
  then $a \leftarrow x_{r1}$, $b \leftarrow x_{r2}$
  else $a \leftarrow x_{r2}$, $b \leftarrow x_{r1} + 1$
$x_n \leftarrow$ RandomNumber($a + 1/(3(k-1))$, $b - 1/(3(k-1))$)
if $x_n > 1$ **then** $x_n \leftarrow x_n - 1$
return x_n

Figure 4. Algorithms for generating (a) balanced random topologies and (b) the used balanced coordinate generation function BalancedCoordinateGen(), where D is circular distance defined in our routing protocol.

scale is reconfigured (expansion or reduction) after being deployed (Section III-C). Figure 3(a) illustrates an example topology interconnecting nine memory nodes, where each memory node has a four-port router. String Figure topology is generated offline before the memory network is deployed.

Balanced random topology generation algorithm. To simplify the topology construction process for system developers, we design a topology generation algorithm, which answers two critical questions: (i) Randomness – how do we ensure that the generated networks are uniformly-random? (ii) Balance – how do we ensure balanced connections? Imbalanced connections are likely to increase congestion. Figure 4 illustrates our random topology generation algorithm. We use the example in Figure 3(b) to explain our design. Inputs of our algorithm include the number of memory nodes N and the number of router ports p. Our approach consists of four steps:

- Constructing L virtual spaces, where the number of virtual space $L = \lfloor \frac{p}{2} \rfloor$. For example, a memory network with four-port routers (not including the terminal port) will lead to maximum two virtual spaces: Space-0 and Space-1.
- Virtually (i.e., logically) distributing all the memory nodes in each virtual space with a random order. We generate random orders by assigning random coordinates to memory nodes. For example, the coordinates of Node-2 is 0.20 and 0.87 in Space-0 and Space-1, respectively.
- Interconnecting the neighboring memory nodes in each

virtual space. For instance, Node-2 is connected with Node-1 and Node-3 in Space-0; it is also connected with Node-6 and Node-8.

- Interconnecting pairs of memory nodes with free ports remaining. For example, because Node-5 and Node-4 are connected in both spaces, Node-5 will have a free port left. Therefore, we can additionally connect Node-5 with Node-3, which also has a free port. When multiple choices exist, we select the pairs of memory nodes with the longest distance.

The solid lines in Figure 3(a) illustrate an example of generated basic random topology. Our topology generation algorithm only determines which nodes are interconnected. We allow both uni-directional and bi-directional connections (discussed in Section IV). The four router ports in our example are all used to connect to other memory nodes in the network. To connect to processors, each router has an additional port, i.e., each router in this example would have five ports in total. Processors can be attached to any subset memory nodes, or all of them (evaluated in Section VI).

Shortcuts generation. The goal of adding extra connections is to maintain high network throughput, when we need to scale down a memory network after it is deployed, e.g., by shutting down (power-gating [14]) routers and corresponding links (more details in Section III-C). To achieve this goal, we generate shortcuts for each memory node to its two and four hop neighbors – within short circular distance – distributed in Virtual Space-0 in a clockwise manner Figure 3(c) shows shortcuts generated for Node-0. We only connect to a node with node number larger than itself. For example, we do not connect Node-5 to Node-0, although Node-0 is Node-5's four-hop neighbor. As such, we limit the link and router hardware overhead by adding maximum two shortcut connections for each node. The connections not existing in the basic balanced random topology are added into the network (e.g., the red dash line between Node-0 and Node-2). Figure 3(a) depicts the final topology combining the basic random topology (black solid lines) and shortcuts (red dash lines). The rationale behind is two-fold. First, we demonstrate that one-, two-, and four-hop connections efficiently accommodate data access of big-data workloads, such as query in a distributed hash table [35]. Second, if we divide Virtual Space-0 into four sectors (A, B, C, and D in Figure 3(a)), the combination of our random topology and the shortcuts ensures that the memory network has direct connections between both short- and long-circular-distance nodes in every two sectors; when the network is scaled down, the shortcuts can maintain high throughput by fully utilize router ports.

Sufficiently uniform randomness of our topology. To show that String Figure provides sufficiently uniform random graphs (SURGs), we compare the average shortest path length of our topology with Jellyfish [33] and S2, which are proved to offer SURG. Empirical results String Figure

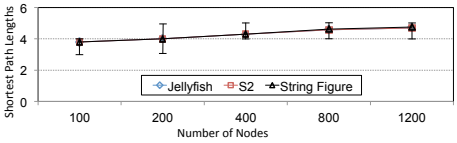


Figure 5. Comparison of shortest path lengths.

Node #	*D* in Space-0	*D* in Space-1	*MD*
7	0.49	0.62	0.49
0	0.20	0.70	0.20
3	0.13	0.44	0.13
6	0.43	0.12	0.12
8	0.68	0.07	0.07

(a)

Node#	Block.	Valid	Hop	Space#	Coordi.
0	1	1	0	0	0.00
2	1	1	0	0	0.20
...					
5	1	1	0	1	0.58
6	1	1	0	1	0.75
...					
3	1	1	1	0	0.33
8	1	1	1	0	0.88
...					

(b)

Figure 6. Greedy routing protocol. (a) Circular distances (D) and minimum circular distances to Node-2 (MD) from Node-7 and Node-7's neighbors. (b) Routing table entries (16 entries in total).

topology leads to similar average shortest path lengths with the same path length bounds across various network scales.

Routing Protocol

A desired routing protocol in memory network needs to be scalable, deadlock free, while fully utilizing the bandwidth of the network topology. Conventional routing schemes on random topologies employ k-shortest path routing with global information stored in the routing table (a look-up table) of each router [33]. Given a memory network with N memory nodes, the routing table size and routing algorithm complexity can be up to $O(NlogN)$ and $O(N^2logN)$. In order to maintain sub-linear increase of routing overhead – consisting of look-up table size and routing algorithm complexity – we adopt a hybrid compute+table routing scheme. Our deadlock freedom mechanism is discussed in Section IV.

Greediest routing. To achieve high-performance routing with small routing table storage, we employ a scalable greedy routing protocol, namely greediest routing [34]. Most previous greedy routing protocols only minimize distance to the destination in a single space. Our basic random topology consists of multiple virtual spaces. Therefore, we need to design a routing protocol that can identify the shortest distance to the destination among all neighbors in all virtual spaces. To this end, we make forwarding decisions by a fixed, small number of numerical distance computation and comparisons. We define circular distance (D) as the distance between two coordinates u and v in each virtual space:

$$D(x, y) = min\{|u - v|, 1 - |u - v|\}$$

We then calculate the minimum circular distance between two nodes as the following, given the two nodes with the set of coordinates in L virtual spaces $\overrightarrow{U} =< u_1, u_2, ..., u_L >$ and $\overrightarrow{V} =< v_1, v_2, ..., v_L >$.

$$MD(\overrightarrow{U}, \overrightarrow{V}) = min\{D(u_i, v_i)\}$$

Forwarding decision making: To forward a packet from Node-s to destination Node-t, the router in Node-

s first selects a neighbor Node-w such that w minimizes $MD(\overrightarrow{X_w}, \overrightarrow{X_t})$ to the destination. The packet is then forwarded to Node-w. This process will continue until Node-w is the destination. For instance, Node-7 needs to send a packet to Node-2. Figure 6(a) shows the minimum circular distances to Node-2 from Node-7 and Node-7's one-hop neighbors in our example (Figure 3). Node-7 has four neighbors Node-0, 3, 6, and 8. Based on the computation, Node-8 has the minimum MD from Node-7. The packet is then forwarded to Node-8. Node-8 will then determine the next stop for the packet by computing a new set of MDs based on its local routing table. The router in each memory node only maintains a small routing table that stores coordinates of its one- and two-hop neighbors in all virtual spaces (Figure 6(b)). To further reduce the routing path lengths, we compute MD with both one- and two-hop neighbor information (based on our sensitivity studies in the results section) stored in the routing table in each router. Routing table implementation is discussed in Section IV. The forwarding decision can be made by a fixed, small number of numerical distance computation and comparisons; the decisions are made locally without link-state broadcast in the network wide.

Adaptive routing. By only storing two-hop neighbors in routing tables, our greediest routing does not guarantee shortest routing path. As such, our design offers path diversity across different virtual spaces, i.e., can have multiple paths satisfying our MD requirement. By leveraging path diversity, we employ adaptive routing to reduce network congestion. We only divert the routing paths by tuning the first hop forwarding decision. With a packet to be sent from node s to destination t, Node-s can determine a set W of neighbors, such that for any $w \in W$, $MD(\overrightarrow{X_w}, \overrightarrow{X_t}) < MD(\overrightarrow{X_s}, \overrightarrow{X_t})$, where t is the destination. Node-s then selects one neighbor in W based on traffic load across s's router ports. We use a counter to track the number of packets waiting at each port to estimate the network traffic load on each outgoing link. At the first hop, the source router can select the links with lightly loaded port satisfying the greediest routing requirement, rather than a heavily loaded port with the queue filled by a user-defined threshold (e.g., 50%). We enforce that our routing reduces the MD to the destination at every hop, eventually finding a loop-free path to the destination (Section IV).

Reconfigurable Memory Network

To achieve our goal in elastic network scale, we propose a set of reconfigurable memory network mechanisms. String Figure naturally supports memory network scaling up and down at low performance and implementation cost, because our design (i) allows arbitrary number of memory nodes, (ii) always fully utilizes the ports in each router, and (iii) only requires local routing table information to make routing decisions. We also design a topology switch (Figure 7) in the routers to reconfigure the links (details described in Section IV).

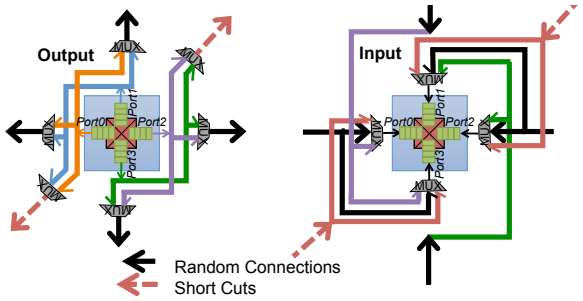


Figure 7. An example topology switch design.

Dynamic reconfiguration for power management. We allow the memory network to dynamically scale up and down (e.g., by power gating routers and links) to enable efficient power management. Memory network power management needs to be carefully designed with most traditional topologies, such as meshes and trees. Otherwise, some nodes can be isolated and require a long wake-up latency on the order of several microseconds [14]. To address this issue, we design a dynamic reconfiguration mechanism that maintains high network throughput after turning off routers. Our dynamic network scale reconfiguration involves four steps. First, it blocks corresponding routing table entries (Figure 6(b)) in associated routers. Second, it enables and disables certain connections between memory nodes. We use the shortcuts to maintain high network throughput after disabling certain links. Third, it validates and invalidates corresponding routing table entries in associated routers. Finally, it unblocks the corresponding routing table entries. The first and last steps ensure atomic network reconfiguration. For example, to turn off Node-1 in Figure 3(a), we will disconnect Node-1 from its one-hop neighbors Node-0, Node-2, Node-5, and Node-6. In each of these nodes, we invalidate the routing table entries that store Node-1's coordinates. We then enable the shortcuts between Node-0 and Node-2, because both nodes now have one free port. We modify the corresponding routing table entries in these two nodes, indicating that the original two-hop neighbors are now one-hop neighbors. Bringing Node-1 back into the network uses the same steps but in reverse. Because our routing protocol maintains two-hop neighbors in each router, each update of routing table entries is simply flipping the blocking, valid, and hop# bits without needing to add or delete entries. Updates in different routers can be performed in parallel.

Static network expansion and reduction for design reuse. Design reuse can reduce the cost and effort of redesigning, re-fabricating, and upgrading systems. Specifically, design reuse allows system developers to reuse a a memory network design or fabrication across server memory systems with different capacity requirement. We use the previously outlined steps of our dynamic network reconfiguration while offline

to enable network expansion and reduction. To support network expansion, system developers can implement a larger network size than currently needed and deploy the memory network with only a subset of memory nodes mounted. The excess nodes are "reserved" for future use. We enable and validate corresponding links and routing table entries based on the mounted memory nodes. As such, network expansion does not require redesign or re-fabrication of the entire memory network. If memory nodes are interfaced through PCBs (e.g., HMC-style), we can expand the memory network by mounting additional memory nodes on the PCB, followed by a link and routing table reconfiguration in the same way as dynamic reconfiguration. As a result, we can reduce the cost and effort of re-fabricating PCBs. Network scale reduction is performed in reverse, by unmounting memory nodes. If the memory nodes are mounted on a silicon interposer (e.g., HBM-styled), we need to fabricate chips with added memory nodes by reusing the original memory network design. However, the design stays the same, substantially reducing non-recurring engineering (NRE) cost.

IMPLEMENTATION

This section describes implementation details of String Figure, including deadlock avoidance, reconfigurable router design, and physical implementation.

Deadlock Avoidance

We must meet two conditions to avoid deadlocks in our network topology. First, route paths must be loop-free from source to destination. Second, the network cannot have cyclical resource dependencies whereby routers wait on each other to free up resources (like buffers). String Figure's greedy routing naturally ensures that route paths are loop-free. We use virtual channels to avoid deadlocks from resource dependencies.

Loop-free routing paths. String Figure ensures our routes between any source-destination pair are loop-free because we always route greedily within our network topology. This is guaranteed by the *progressive and distance-reducing* property (Appendix A) of our greedy routing protocol. Appendix A formally proves that packet routes are loop-free.

Avoiding deadlocks with virtual channels. We adopt two virtual channels [36], [37], [38] to avoid deadlocks. Packets use one virtual channel when routing from a source of a lower space coordinate to a destination of a higher space coordinate; packets use the other virtual channel when routing from a source of higher space coordinate to a destination of a lower space coordinate. This avoids deadlocks because in our topology (which is not truly random), packets are only routed through to networks with a strictly increasing coordinate or a strictly decreasing coordinate; the only dependency is between the virtual channels in the router, which is insufficient to form cycles [38]. Whereas virtual channels can increase the required buffering, our network topology allows the number of router ports to remain constant as the network scales up. Therefore, the buffer size overhead is less of an issue compared with prior works [38], [39] (evaluated in Section VI).

Router Implementation and Reconfiguration

We design the router on each memory node to facilitate our routing table design, reconfigurable links, and counters for adaptive routing.

Routing table implementation. Figure 6(b) illustrates our routing table implementation. Each routing table stores information of its one- and two-hop neighbors, including $log_2 N$ memory node number, 1-bit blocking bit, 1-bit valid bit, 1-bit hop number ('0' for one-hop and '1' for two-hop), $\lceil log_2 \frac{p}{2} \rceil$ virtual space number, and 7-bit virtual coordinate. We initialize routing table entries accordingly, while we generate the network topology. Once the network topology is generated, we only update the blocking, valid, and hop bit values during network reconfiguration. A memory node has maximum two one-hop neighbors in each virtual space; each of the one-hop neighbors has two one-hop neighbors of their own in each virtual space. As the maximum number of virtual spaces is half of the number of ports (p), each routing table has a maximum of $p(p+1)$ entries.

Enabling link and topology reconfiguration with switches. Our memory network reconfiguration requires connecting and disconnecting certain links between neighbor memory nodes. Our basic balanced random network topology already fully utilizes all router ports. However, each node also has at most two shortcut connections (Section III-A). To accommodate the shortcuts, we implement a switch to attach the two shortcut connections to two of the router ports at each node. Figure 7 shows our topology switch design. It is comprised of a set of multiplexers similar to prior reconfigurable NoC designs [40], [41]. As a result, the topology switches allow us to select p (the number of router ports) connections out of all the random connections and shortcuts provided by our topology.

Tracking port utilization with packet counters. With adaptive routing, we use counters at each port to track the queue length at the port. The number of counter bits is $log_2 q$, where q is the number of queue entries of the port. The counter provides an additional variable for determining routing paths. It specifically tracks the congestion of each path by counting how often we route packets to specific outputs ports. We then use this counter value to tweak our routing algorithm to make smarter decisions. If an output port has too many packets routed to it, the algorithm detects this through counters and chooses alternate, yet still greedy, output ports to send the packet. This helps us avoid congestion in the network and still achieve low latency, high throughput performance overall. These counters are reset after the network is reconfigured.

Physical Implementation

The goal of our physical implementation is to reduce both the area overhead and long wires in our memory network.

Bounded number of connections in the network. With our network topology, the number of connections coming out of each node is bounded by the number of router ports (p) and remains *constant*, independent of network scale (N). Each node has $\frac{p}{2}$ one-hop neighbors in our basic random topology and a maximum of two shortcuts (some generated shortcuts will overlap with connections in the basic random topology). Therefore, the total number of connections coming out of each node $C_{node} \leq \frac{p}{2} + 2$. For example, a memory node with an 8-port router only requires six connections per node. Given N memory nodes in total, the total number of required connections in the network $C_{network} \leq N \times (\frac{p}{2} + 2)$, which grows linearly with the number of memory nodes.

Uni-directional versus bi-directional connections. Bi-directional connections allow packets to traverse the network both forward and backwards. Uni-directional connections typically have worse packet latency than their bi-directional counterparts, due to reduced path diversity. However, uni-directional networks have lower hardware and energy cost than bi-directional connections. Our sensitivity studies (Section VI) demonstrate that uni-directional networks perform almost the same as bi-directional networks; their discrepancy diminishes with increasing number of nodes in the network. Therefore, String Figure uses uni-directional connections.

Memory node placement and wire routing. When building String Figure, we place memory nodes in the memory network (on PCB or silicon interposer) as a 2D grid. Our goal of memory node placement is to reduce long wires. Memory network implementations are constrained by wire lengths [29], [42]. For example, HBMs [42] (with a 7 mm dimension in HBM1 and 12 mm in HBM2) are implemented with interposers to support large-scale memory networks; previous works demonstrate that memory nodes can be clustered with MetaCubes [16] (i.e., clustered memory nodes integrated with an interposer), which is further interconnected with other interposer-integrated clusters. To achieve our goal, we set two priority levels that prioritize the clustering of one-hop and two-hop neighbors. For example, we ensure that all one-hop neighbors are placed within ten grid distance with place and routing. Our network topology also naturally supports MetaCube [16] architecture. Our network topology provides connections with various circular distances. As such, we place memory nodes with short circular distances in the same MetaCubes. Inter-MetaCube links are implemented by connections with long circular distances.

Processor placement. The flexibility of String Figure topology and routing protocol allows us to attach a processor to any one or multiple memory nodes. The router at each memory node has a local port connecting to the processor. As such, attaching a processor to multiple memory nodes can increase processor-memory bandwidth. By tuning traffic patterns of our synthetic workloads, our evaluation examines ways of injecting memory traffic from various locations, such as corner memory nodes, subset of memory nodes, random memory nodes, and all memory nodes.

Table I
SYSTEM CONFIGURATION.

CPU	4 sockets; 2GHz; 64B cache-line size
Memory	up to 1296 memory nodes; 8GB per memory node (stack)
DRAM timing	tRCD=12ns, tCL=6ns, tRP=14ns, tRAS=33ns
CPU-memory channel	256 lanes in total (128 input lanes and 128 output lanes); 30Gbps per lane
SerDes delay	3.2ns SerDes latency (1.6ns each) per hop
Energy	Network: 5pJ/bit/hop; DRAM read/write: 12pJ/bit

Table II
SUMMARY OF NETWORK TOPOLOGY FEATURES AND REQUIREMENTS.

Topology	Requires High-Radix Routers?	Router Port scaling?	Reconfigurable Network Scaling
ODM	No	No	No
AFB	Yes	Yes	No
S2-ideal	No	No	No
SF	No	No	Yes

EXPERIMENTAL SETUP

RTL Simulation Framework

We evaluate String Figure via RTL design in SystemVerilog [46] and PyMTL [47]. We develop synthesizeable RTL models of each network topology, routing protocol, memory node, router configuration, and wire lengths. Table I describes the modeled system configurations. We use the same configuration, timing, and energy parameters evaluated in previous works. We estimate the dynamic energy used in the network using average picojoule-per-bit numbers to provide a fair comparison of memory access energy [25], [14], [16]. Network clock rate is the same as memory nodes clock speed, e.g., 312.5MHz with HMC-based memory nodes. We do not evaluate static energy, as static power saving is highly dependent on the underlying process management assumptions (e.g., race-to-idle). We also model the network link latency based on wire length obtained from 2D grid placement of memory nodes. We add an extra one-hop latency with a wire length equal to ten memory nodes on the 2D grid (based on the wire length supported by HMC). Our RTL simulator can run workload traces collected using our in-house trace generation tool, which is developed on top of Pin [48]. We collect traces with 100,000 operations (e.g., grep for Spark-grep, queries for Redis) after workload initialization. Our trace generator models a cache hierarchy with 32KB L1, 2MB L2, and 32MB L3 with associativities of 4, 8, and 16, respectively. Our trace generator does not contain a 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 and generate a timestamp for each memory access.

Considered topologies. We compare String Figure to a variety of network topologies and routing protocols summarized

Topology	Number of Nodes (N), Number of Ports per Router (p)												Routing Scheme
	N	16	17	32	61	64	113	128	256	512	1024	1296	
Distributed-Mesh (DM)/ Optimized DM (ODM)	p	4	N	4	N	4	N	4	4	4	4	4	Greedy + adaptive
Flattened Butterfly (FB)	p								20	24	31	33	Minimal + adaptive
Adapted FB (AFB)	p								13	17	23	25	Minimal + adaptive
Space Shuffle Ideal (S2-ideal)	p	4	4	4	4	4	4	4	8	8	8	8	Look-up table
String Figure (SF)	p	4	4	4	4	4	4	4	8	8	8	8	Look-up table + greediest + adaptive

Figure 8. Evaluated network topologies and configurations ("N" indicates unsupported network scale).

Table III
DESCRIPTION OF NETWORK TRAFFIC PATTERNS.

Traffic Pattern	Formula	Description
Uniform Random	`dest = randint(0, nports-1)`	Each node produces requests to a random destination node in the network.
Tornado	`dest = (src+nports/2) % nports`	Nodes send packets to a destination node halfway around the network.
Hotspot	`dest = const`	Each node produces requests to the same single destination node.
Opposite	`dest = nports - 1 - src`	Sends traffic to opposite side of network like a mirror.
Nearest Neighbor	`dest = src + 1`	Each node sends requests to its nearest "neighbor" node, one away.
Complement	`dest = src` $\oplus$ `(nports-1)`	Nodes send requests to their bitwise complement destination node.
Partition 2	`dest = randint(0,nports-1) & (nports/2-1) | (src & (nports/2))`	Partitions the network into two groups of nodes. Nodes randomly send within their group.

Table IV
DESCRIPTION OF EVALUATED REAL WORKLOADS.

Workload	Description
Spark-wordcount	A "wordcount" job running on Spark, which counts the number of occurrences of each word in the Wikipedia data set provided in BigDataBench [43].
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 data set provided in BigDataBench [43].
Spark-sort	A "sort" job running on Spark, which sorts values by keys with the Wikipedia data set provided in BigDataBench [43].
Pagerank	A measure of Twitter influence. From the graph analysis benchmark in CloudSuite [44]. Twitter data set with 11M vertices.
Redis	An in-memory database system which simulates running 50 clients at the same time sending 100,000 total queries [45].
Memcached	From CloudSuite [44], which simulates the behavior of a Twitter caching server using the Twitter data set with 8 client threads, 200 TCP/IP connections, and a get/set ratio of 0.8.
Matrix Mul	Multiplying two large matrices stores in memory and storing their result in memory.
Kmeans	Clustering Algorithm partitions n observations into k clusters where each observation belongs to cluster with the nearest mean.

in Figure 8. We also describe their features and requirements in Table II. The number of router ports *does not include* the terminal port connecting to the local memory node. String Figure allows arbitrary network scale. However, to provide concrete examples in our evaluation and demonstrate our support for elastic network scale, we implement two basic topologies with 128 nodes (4 router ports) and 1296 nodes (8 router ports), respectively. We reconfigure the basic topologies to evaluate networks with fewer of nodes. Mesh is widely explored in previous memory network designs as one of offering the best performance among various topologies [15], [14]. We implement a baseline Optimized Distributed Mesh (ODM) topology [15] for memory network. In addition, we compare with several network designs optimized for scalability of distributed systems, including a 2D Adaptive Flattened Butterfly (AFB) [19] and S2 [34]. S2 does not support down-scaling with the same original topology (it requires regenerating new topologies and routing tables with a smaller number of nodes). Therefore, our evaluation of S2 provides an impractical ideal baseline. We name it S2-ideal. Additionally, our experiment results focuses on evaluating performance, energy, and scalability of the network designs. However, most of the baseline topologies require high-radix routers [19], [49] and the number of ports and links continues to grow with network scale, leading to non-linear growth of router and link overhead in memory networks. Furthermore, none of the baseline topologies offer the flexibility and reconfigurability in memory networks as provided by our design.

Bisection bandwidth. To provide a fair point of comparison, we evaluate network designs based on the same (or similar) bisection bandwidth. Because String Figure and S2 [34] have random network topologies, we calculate their empirical minimum bisection bandwidth by randomly splitting the memory nodes in the network into two partitions and calculating the maximum flow between the two partitions. The minimum bisection bandwidth of a topology is calculated from 50 random partitions. We adopt the average bisection bandwidth across 20 different generated topologies. With a fixed network size, the bisection bandwidth of FB with high-radix routers can be much higher than the other topologies. However, mesh is lower. To match the bisection bandwidth, we also evaluate an Adaptive FB (AFB) implemented by partitioned FB [38] with fewer links and an optimized DM (ODM) with increase the links per router to match the bisection bandwidth of String Figure and S2 at each memory network scale.

Workloads

We evaluate both network traffic patterns and real workloads on our simulation framework. **Traffic patterns:** We evaluate different traffic patterns running on String Figure and baseline designs. Table III lists details about these traffic patterns. We use these traffic patterns to evaluate the memory network designs and expose the performance and scalability trends. We sweep through various memory network sizes, router configurations, and routing protocols listed in Figure 8. To exercise our memory network design, each memory node sends requests (similar to attaching a processor to each memory node) at various injection rates. For example, given an injection rate of 0.6, nodes randomly inject packets 60% of the time. **Real workloads:** We also evaluate various real workloads with trace-driven simulation. We run various in-memory computing workloads in CloudSuite [44], BigDataBench [43], and Redis benchmark [45] 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 data set characteristics. We scale the input data size of each real workload benchmark to fill the memory capacity. Data is distributed among the memory nodes based on their physical address.

RESULTS

We evaluate our design with various metrics including average path lengths, network saturation, average packet latency, workload instruction throughput (IPC), memory dynamic energy, and energy efficiency. Our evaluation shows that String Figure achieves close to or better than the performance and scalability of the best of prior designs (ODM, AFB, and S2-ideal), yet leads to lower energy consumption and higher energy efficiency with our (i) fewer router ports and wires needed and (ii) elastic network scale.

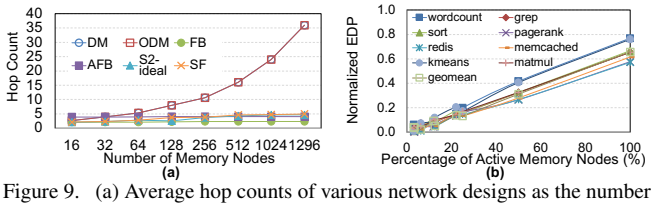


Figure 9. (a) Average hop counts of various network designs as the number of memory nodes increases. (b) Normalized energy-delay product (EDP) (the lower the better) with various workloads, when we power gate off certain amount of memory nodes.

Path lengths. Figure 9(a) shows the average shortest path lengths of various network designs across our synthetic traffic patterns and real workloads. When the memory network has over 128 memory nodes, the average hop count of DM and ODM network increases superlinearly with increasing network size. Specifically, the average hop count of these two topologies is $\frac{2}{3}t$ where t is the average of their two dimensions. Rather, the other network topologies, S2-ideal, FB, AFB, and our String Figure design, do not incur significant increase in the average shortest path lengths in large-scale memory networks. FB achieves the best average shortest path lengths among all the network topologies, because it employs many more ports in routers than other topologies as the network scales up. With a maximum of eight ports per router, String Figure still achieves 4.75 and 4.96 average hop counts when the network scales up to 1024 and 1296 memory nodes, respectively. We also evaluate 10% and 90% percentile shortest path lengths. String Figure can achieve 4 hops and 5 hops with over one thousand nodes, at 10% and 90% percentile, respectively. Therefore, String Figure path length is scalable to memory network size over one thousand nodes.

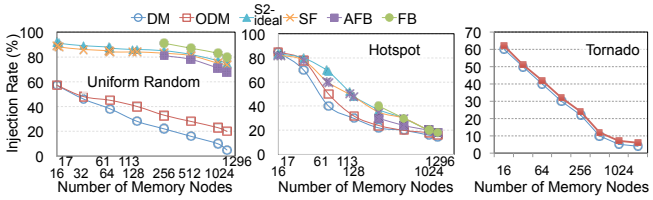


Figure 10. Network saturation points across various numbers of nodes.

Network saturation. We evaluate network saturation with several traffic patterns shown in Figure 10. String Figure can achieve close to the best of all other network architectures. In order to clearly visualize all the curves, we only show the the results of the rest of the network architectures. Traffic patterns `uniform random`, `hotspot`, and `tornado` are particularly noteworthy and show different results. The remaining traffic patterns `partition2`, `complement`, `opposite`, and `neighbor`, have similar behavior as shown. In almost all traffic patterns, the mesh network topologies, DM and ODM, saturates first at the lowest injection rate. Nearest-neighbor routing is the exception to this. SF perform worse with nearest-neighbor than ODM. This is because in mesh topologies, nodes are always one-hop away from their nearest neighboring node. Note, that we generated nearest-neighbor network traffic using the router IDs rather than number of hops. Therefore, "neighboring" nodes in SF are not necessarily one hop away from each other which means this network has higher latency. However, an exception to mesh saturating first is in networks with very few nodes. At the fewest node configuration (i.e., 16 nodes), ODM slightly edges out SF. However, as the number of memory nodes increases, SF scales significantly better. ODM also saturates at a higher injection rate than other network designs with `hotspot` traffic pattern. We do not observe network saturation in *tornado* traffic pattern with all topologies, except for mesh. Network latency remains steady even in high injection rates and large number of memory nodes. The reason is the geometric structure of the network designs. With either one of AFB, FB, S2-ideal, and SF, it is typically easy for packets in a network to traverse half or the entire network in just a hop or two to reach their destination. Traffic patterns, such as *tornado*,

generate traffic in a mathematically geometric manner which is advantageous in such topologies.

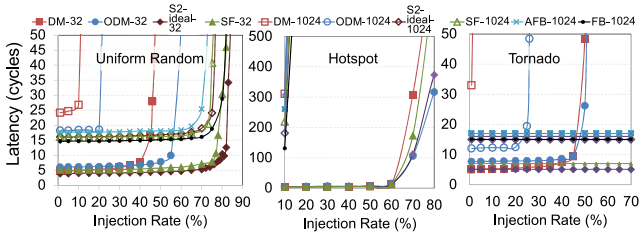


Figure 11. Performance of traffic patterns at less than one thousand nodes.

Average packet latency. We evaluate the average travel time (latency) between any two nodes in a network shown in Figure 11. Each traffic pattern graph shows the latency in the leftmost data point for each network. S2-ideal and SF appear to scale well with the number of memory nodes. As the number of nodes in the network increases, these topologies show almost no degradation in their network saturation points. SF has slightly longer latency than S2-ideal with networks down-scaled from the original size, because shortcuts and adaptive routing can degrade the randomness among network connections. However, SF still demonstrates lower latency than AFB at large network scales. We also evaluate the memory access latency of various traffic patterns.

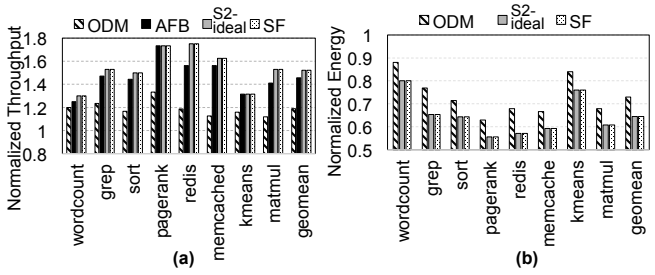


Figure 12. Normalized (a) system throughput (higher is better) and (b) dynamic memory energy (lower is better) with various real workloads.

Performance and energy of real workloads. We evaluate system performance and memory dynamic energy consumption with several real workloads running in a memory system, where the total memory capacity is 8TB distributed across the 1024 (down-scaled from 1296) memory nodes in the network. We take into account dynamic reconfiguration overhead to perform power gating in our RTL simulation by implementing SF reconfiguration mechanisms. The sleep and wake-up latency of a link is conservatively set to 680ns and 5μs similar to prior works [14], [15]. To minimize the performance impact of reconfiguration, we set the reconfiguration granularity (i.e., the minimum allowed time interval between reconfigurations) to be 100us. Figure 12(a) shows the throughput of real workloads with varying memory network architectures, normalized to DM. Our results demonstrate that String Figure can achieve close to the best performance across various workloads. Our design achieves 1.3$\times$ throughput compared with ODM. Figure 12(b) illustrates normalized memory dynamic energy consumption with our workloads, normalized to AFB. String Figure design can achieve the lowest energy consumption across these network topologies. S2-ideal also achieves similarly low energy consumption, due to its energy reduction in routing. On average, SF reduces energy consumption by 36% compared with AFB.

Memory network power management. We also evaluate memory network power management by powering gating off various portions of the memory system with total 1296 nodes. Figure 9(b) shows the energy efficiency of our power management by considering both energy saving and system performance overhead. As we demonstrate in our results, our design can achieve significantly improved energy efficiency, as we power gate more parts of the memory network.

Related Work

To our knowledge, String Figure is the first memory network architecture that offers both scalability and elastic network scale in a single design. Most previous memory network designs do not take into account scalability as a primary design goal. Kim *et al.* [15], [25] explored memory network with mesh, butterfly, and dragonfly topologies with 64 HMC-based memory nodes. The study showed that distributed mesh outperforms other network topologies at this scale. Zhan *et al.* [14] investigated performance and energy optimization on a mesh-based memory network design up to 16 memory nodes. Poremba *et al.* [16] extended the memory network capacity to 2TB implemented by 128 HMC-like memory nodes used in CPU+GPU systems. However, the memory nodes are mapped separate processor channels, rather than shared by all the processors. Fujiki *et al.* [51] proposes a random network topology to support scalability of memory networks, yet does not support the flexibility and reconfigurability as our design.

Scalability and flexibility are central themes in data center network [19], [20], [33], [34], [52], [53]. Recent planar topologies, such as a Flattened Butterfly [19] and Dragonfly [20], offer promising scalability and high network throughput. However, these designs require high-radix routers and substantial increase of number of ports, which can impose non-linearly increasing router area and power [49]. This leads to prohibitively high cost in routers and the amount wiring at large-scale memory network. Furthermore, butterfly-like topologies typically have symmetric layout. This can lead to isolated nodes or suboptimal routing, when subsets of nodes are turned down. Jellyfish [33] employs random topology to achieve close-to-optimal network throughput and incremental growth of network scale. Yet, Jellyfish provides high throughput by requiring k-shortest path routing; the size of forwarding tables per router can increase superlinearly with the number of routers in the network. This is impractical in a memory network, where routers have limited storage space. String Figure uses greedy routing due to our topology with ran-

domly assigned coordinates in multiple spaces. As such, our design can achieve both high-throughput routing and constant forwarding state per router. S2 [34] adopts random topologies and computation-based routing mechanisms with scalable routing tables. Yet, S2 [34] requires cable plug in/out to increase the network size, which is impractical in memory networks that have pre-fabricated link wires. S2 does not support network downscaling, unlike String Figure.

Recent NoC designs tackle scalability and fault tolerance issues, when interconnecting processor cores. Slim NoC [38] enables low-diameter network in NoC. However, the design requires increasing router ports and wires as the network scales up. Furthermore, these topologies does not support the level of flexibility and reconfigurability as String Figure. Small world network [53] also employs greedy routing, but it does not produce the shortest paths and can be difficult to be extended to perform multi-path routing that can fully utilize network bandwidth. Previous network fault tolerance schemes [54], [55], [56], [57], [58], [59], [60], [61], [62] allow NoC to continue efficient functioning when routers are taken out of the network. However, most previous designs are developed for limited network scales and certain network topologies [54] and impose high router area overhead by employing one or several routing tables [55], [56], [57], [58], [59], [60], [61], [62].

Conclusions

In this paper, we examined the critical scalability and flexibility challenges facing memory network architecture design in meeting the increasing memory capacity demand of future cloud server systems. We proposed a new memory network architecture, which consists of topology, routing scheme, and reconfiguration mechanisms. Our design offers numerous benefits towards practical use of memory network architecture in server systems, such as scaling up to over a thousand memory nodes with high network throughput and low path lengths, arbitrary number of memory nodes in the network, flexible network scale expansion and reduction, high energy efficiency, and low cost in routers.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This paper is supported in part by NSF grants 1829524, 1829525, 1817077, 1701681, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

References

[1] G. E. Moore, "Readings in computer architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds., 2000, ch. Cramming More Components Onto Integrated Circuits, pp. 56–59.

[2] J. Barr, "EC2 in-memory processing update: Instances with 4 to 16 TB of memory and scale-out SAP HANA to 34 TB," 2017.

[3] SAP, "SAP HANA: an in-memory, column-oriented, relational database management system," 2014. [Online]. Available: http://www.saphana.com/

[4] T. A. S. Foundation, "Spark," 2014. [Online]. Available: http://spark.incubator.apache.org/

[5] VoltDB, "Voltdb: Smart data fast," 2014. [Online]. Available: http://voltdb.com/

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[7] C. Szegedy, W. Liu, and Y. J. et al., "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[8] The Next Platform, "Baidu eyes deep learning strategy in wake of new GPU options," in *www.nextplatform.com*, 2016.

[9] J. Barr, "Now available – EC2 instances with 4 TB of memory," 2017.

[10] C. Sanders, "Announcing 4 TB for SAP HANA, single-instance SLA and hybrid use benefit images," 2016.

[11] "Microsoft azure documentation," https://https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-memory#m-series.

[12] O. Mutlu and L. Subramanian, "Research problems and opportunities in memory systems," *Supercomput. Front. Innov.: Int. J.*, vol. 1, no. 3, pp. 19–55, Oct. 2014.

[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 transparent memory expansion," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

[14] 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 *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[15] G. Kim, J. Kim, J. H. Ahn, and J. Kim, "Memory-centric system interconnect design with hybrid memory cubes," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 145–156.

[16] M. Poremba, I. Akgun, J. Yin, O. Kayiran, Y. Xie, and G. H. Loh, "There and back again: Optimizing the interconnect in networks of memory cubes," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 678–690.

[17] "The Machine: A new kind of computer," https://www.labs.hpe.com/the-machine.

[18] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien, and R. Nair, "Data access optimization in a processing-in-memory system," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 6:1–6:8.

[19] J. Kim, W. J. Dally, and D. Abts, "Flattened Butterfly: A cost-efficient topology for high-radix networks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 126–137.

[20] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable Dragonfly topology," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008, pp. 77–88.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 2004, pp. 10–10.

[22] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt,

and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 267–278.

[23] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Proceedings of the Linux Symposium*, 2009, pp. 191–200.

[24] "Gen-Z Consortium," https://genzconsortium.org.

[25] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-gpu system design with memory networks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014, pp. 484–495.

[26] AMD, "AMD Radeon R9 series graphics cards," http://www.amd.com/en-us/products/graphics/desktop/r9.

[27] "NVIDIA Tesla P100: Infinite compute power for the modern data center," http://www.nvidia.com/object/tesla-p100.html.

[28] "Intel Xeon Phi processor 7200 family memory management optimizations," https://software.intel.com/en-us/articles/intel-xeon-phi-processor-7200-family-memory-management-optimizations.

[29] Micron, "Hybrid memory cube specification 2.1."

[30] L. Nai and H. Kim, "Instruction offloading with HMC 2.0 standard: A case study for graph traversals," in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 258–261.

[31] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 380–392.

[32] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08, 2008, pp. 63–74.

[33] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, 2012, pp. 17–17.

[34] Y. Yu and C. Qian, "Space shuffle: A scalable, flexible, and high-bandwidth data center network," in *Proceedings of the 2014 IEEE 22Nd International Conference on Network Protocols*, ser. ICNP '14, 2014, pp. 13–24.

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01, 2001, pp. 149–160.

[36] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Virtual channels in networks on chip: Implementation and evaluation on hermes NoC," in *Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design*, 2005, pp. 178–183.

[37] J. Lee, S. Li, H. Kim, and S. Yalamanchili, "Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 4, pp. 48:1–48:28, Oct. 2013.

[38] M. Besta, S. M. Hassan, S. Yalamanchili, R. Ausavarungnirun, O. Mutlu, and T. Hoefler, "Slim NoC: A low-diameter on-chip network topology for high energy efficiency and scalability," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 43–55.

[39] S. Hassan and S. Yalamanchili, "Bubble Sharing: Area and energy efficient adaptive routers using centralized buffers," in *Proceedings of the NOCS*, 2014.

[40] A. Jain, R. Parikh, and V. Bertacco, "High-radix on-chip networks with low-radix routers," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, 2014, pp. 289–294.

[41] M. B. Stuart, M. B. Stensgaard, and J. Sparsø, "The renoc reconfigurable network-on-chip: Architecture, configuration algorithms, and evaluation," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 4, pp. 45:1–45:26, Nov. 2011.

[42] "JEDEC publishes HBM2 specification as Samsung begins mass production of chips," https://www.anandtech.com/show/9969/jedec-publishes-hbm2-specification.

[43] 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*. IEEE, 2014, pp. 488–499.

[44] 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.

[45] "Redis Benchmark," http://redis.io/topics/benchmarks.

[46] D. I. Rich, "The evolution of systemverilog," *IEEE Des. Test*, vol. 20, no. 04, pp. 82–84, Jul. 2003.

[47] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in *47th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2014, pp. 280–292.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005, pp. 190–200.

[49] S. Li, P.-C. Huang, D. Banks, M. DePalma, A. Elshaarany, S. Hemmert, A. Rodrigues, E. Ruppel, Y. Wang, J. Ang, and B. Jacob, "Low latency, high bisection-bandwidth networks for exascale memory systems," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 62–73.

[50] "Spark 1.4.1," http://spark.apache.org/downloads.html.

[51] D. Fujiki, H. Matsutani, M. Koibuchi, and H. Amano, "Randomizing packet memory networks for low-latency processor-memory communication," in *Proceedings of Parallel, Distributed, and Network-Based Processing (PDP), Euromicro International Conference on*, 2016.

[52] M. Koibuchi, H. Matsutani, H. Amano, D. F. Hsu, and H. Casanova, "A case for random shortcut topologies for HPC interconnects," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012, pp. 177–188.

[53] U. Y. Ogras and R. Marculescu, ""it's a small world after all": NoC performance optimization via long-range link insertion," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 7, pp. 693–706, Jul. 2006.

[54] M. Fattah, A. Airola, R. Ausavarungnirun, N. Mirzaei, P. Liljeberg, J. Plosila, S. Mohammadi, T. Pahikkala, O. Mutlu, and H. Tenhunen, "A low-overhead, fully-distributed, guaranteed-delivery routing algorithm for faulty network-on-chips," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, 2015, pp. 18:1–18:8.

[55] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco, "ARIADNE: Agnostic reconfiguration in a disconnected network

environment," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 298–309.
[56] M. Balboni, J. Flich, and D. Bertozzi, "Synergistic use of multiple on-chip networks for ultra-low latency and scalable distributed routing reconfiguration," in *Proceedings of the 2015 Design, Automation and Test in Europe Conference*, 2015, pp. 806–811.
[57] C. Feng, Z. Lu, A. Jantsch, M. Zhang, and Z. Xing, "Addressing transient and permanent faults in NoC with efficient fault-tolerant deflection router," in *IEEE TVLSI*, 2013.
[58] D. Fick, A. DeOrio, G. Chen, V. Bertacco, D. Sylvester, and D. Blaauw, "A highly resilient routing algorithm for fault-tolerant NoCs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2009, pp. 21–26.
[59] D. Lee, R. Parikh, and V. Bertacco, "Brisk and limited-impact NoC routing reconfiguration," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 2014, pp. 306:1–306:6.
[60] R. Parikh and V. Bertacco, "uDIREC: Unified diagnosis and reconfiguration for frugal bypass of NoC faults," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 148–159.
[61] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide, "Immunet: A cheap and robust fault-tolerant packet routing mechanism," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004, pp. 198–209.
[62] E. Wachter, A. Erichsen, A. Amory, and F. Moraes, "Topology-agnostic fault-tolerant NoC routing method," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 1595–1600.

APPENDIX

We formally prove our proposition on loop freedom by two lemmas similar to those derived in data center networks [34].

Lemma 1. *In a virtual space with a given a coordinate x, if a memory node s is not the node that has the shortest circular distance to x in this space, then s must have an adjacent router s', such that $D(x, x_{s'}) < D(x, x_s)$.*

Proof:

1) Let w be the node closest to x among all memory nodes in the virtual space.
2) The ring of this space is divided by s and x into two arcs. At least one of the arcs has a length no greater than $\frac{1}{2}$. Suppose we have $\widehat{x_s, x}$ with length $l(\widehat{x_s, x})$. We have $D(x_s, x) = l(\widehat{x_s, x}) \leq \frac{1}{2}$.
3) If w is on $\widehat{x_s, x}$, let the arc between s and w be $\widehat{x_s, x_w}$.
   a) If s has an adjacent node q with coordinate on $\widehat{x_s, x_w}$, then $l(\widehat{x_q, x}) < l(\widehat{x_s, x})$. Hence, $D(q, x) = l(\widehat{x_q, x}) < l(\widehat{x_s, x}) < D(x_s, x)$.
   b) If s has no adjacent node on $\widehat{x_s, x_w}$, w is x's adjacent node. Hence, s has an adjacent node w, such that $D(x, x_w) < D(x, x_s)$.
4) If w is not on $\widehat{x_s, x}$, we have an arc $\widehat{x_s, x, x_w}$. For the arc $\widehat{x, x_w}$ on $\widehat{x_s, x, x_w}$, we have $l(\widehat{x, x_w}) < l(\widehat{x_s, x})$. (Assuming to the contrary, if $l(\widehat{x, x_w}) \geq l(\widehat{x_s, x})$, then we cannot have $D(x, x_w) < D(x, x_s)$. There is contradiction.)
   a) If s has an adjacent memory node q with coordinate on $\widehat{x_s, x, x_w}$, then $l(\widehat{x_q, x}) < l(\widehat{x_s, x}) \leq \frac{1}{2}$. Hence, $D(q, x) = l(\widehat{x_q, x}) < l(\widehat{x_s, x} = D(x_s, x)$.
   b) If s has no adjacent memory node on $\widehat{x_s, x, x_w}$, w is x's adjacent node. Hence, s has an adjacent node w, such that $D(x, x_w) < D(x, x_s)$.
5) Combining (3) and (4), s always has an adjacent node s', such that $D(x, x_{s'}) < D(x, x_s)$.

■

Lemma 2. *Suppose the source and destination of a packet are routers s and t, respectively. Coordinates of the destination router in all virtual spaces are $\overrightarrow{\mathrm{X_t}}$. Let w be the router that has the minimum MD to t among all neighbors of s, then $MD(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) < MD(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.*

Proof:

1) Suppose the minimum circular distance between s and t is defined by their circular distance in the jth space, i.e., $D(x_{tj}, x_{sj}) = MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.
2) In the jth space, t is the memory node with the shortest circular distance to x_{tj}, which is $D(x_{tj}, x_{tj}) = 0$. Because $s \neq t$, s is not the node with the shortest circular distance to x_{tj}.
3) Based on Lemma 1, s has an adjacent memory node s', such that $D(x_{tj}, x_{s'j}) < D(x_{tj}, x_{sj})$.
4) Then, $MD_L(\overrightarrow{\mathrm{X_{s'}}}, \overrightarrow{\mathrm{X_t}}) \leq D(x_{tj}, x_{x'j}) < D(x_{tj}, x_{sj}) = MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.
5) Because w is the node that has the shortest MD to $\overrightarrow{\mathrm{X_t}}$ among all neighbors of s, we have $MD_L(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) \leq MD_L(\overrightarrow{\mathrm{X_{s'}}}, \overrightarrow{\mathrm{X_t}} \leq MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.

■

Lemma 2 states on a packet's route, if a router s is not the destination, it must find a neighbor whose MD is smaller than s's MD to the destination.

Proposition 3. *Greediest routing finds a loop-free path of a finite number of hops to a given destination on our network topology.*

Proof:

1) Suppose memory node s receives a packet with destination node t. If $s = t$, then s is the destination. The packet arrives at the destination.
2) If $s \neq t$, according to Lemma 2, s will find a neighbor w, such that $MD_L(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) < MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$, and forward the packet to w.
3) The MD from the current memory node to the destination coordinates strictly reduces at each hop. Routing keeps making progress. Therefore, there is no routing loop.

■