Fabric-Centric Computing

Ming Liu

University of Wisconsin-Madison

ABSTRACT

Emerging memory fabrics and the resulting composable infrastructures have fundamentally challenged our conventional wisdom on how to build efficient rack/cluster-scale systems atop. This position paper proposes a new computing paradigm–called **Fabric-Centric Computing (FCC)**–that views the memory fabric as a first-class citizen to instantiate, orchestrate, and reclaim computations over composable infrastructures. We describe its design principles, report our early experiences, and discuss a new intermediate system stack proposal that harnesses the uniqueness of this cluster interconnect and realizes the vision of FCC.

1 INTRODUCTION

Memory fabrics (such as Gen-Z [10], OpenCAPI [18], CCIX [4], and CXL [6]), an emerging cluster interconnect, have gained significant interest recently and are being considered to replace today's communication fabrics used for building enterprise racks and on-premise clusters. They provide the load/store access model and enable tight integration of cross-node memory and accelerators into host systems, resulting in a real *composable infrastructure*. Such an architecture will bring many benefits, such as on-demand scaling, fine-grained computation resource sharing, and energy/cost-efficiency improvements. The last few years have seen a number of memory fabrics and composable testbeds [3, 9, 17, 24, 54, 73] being proposed, developed, and sampled.

However, a memory fabric fundamentally differs from a communication fabric as follows. First, it is transparently integrated into the memory hierarchy and execution pipeline of the host processor, requiring a synchronous execution model. Second, a memory fabric brings back several memory node types proposed in early scalable and shared-memory machines, yielding more data placement options. Third, its performance heavily hinges on how routable PCIe performs at scale, especially when handling unpredictable and concurrent PCIe transactions. Fourth, it enables a lightweight and fast context-switching scheme among fabric-attached execution engines for coordinated computation. Finally, a memory fabric introduces passive failure domains where a remote node has little capability to implement fault tolerance mechanisms. Therefore, these differences make our prior system design philosophies and techniques of building communication fabric-based rack/cluster-scale systems either ineffective or invalid for the composable infrastructure.

In this paper, we propose a new computing paradigm–called **Fabric-Centric Computing (FCC)**–that views the memory fabric as a first-class citizen to instantiate, orchestrate, and reclaim computations over composable infrastructure. FCC materializes the memory fabric's capabilities as primitives and abstractions, defines the interaction interface and templates for different fabric-attached components, and coordinates computations based on the communication limits of the fabric. We propose several design principles to realize the vision of FCC: data movement as a managed service, host-assisted memory node type-conscious data structures, idempotent tasks and hardware cooperative scalable functions, and a fabric central arbitrator via dedicated lanes. We then sketch an intermediate system stack following these principles to realize a vision of FCC.

2 MEMORY FABRIC AND COMPOSABLE INFRASTRUCTURE

2.1 A Quick Memory Fabric Primer

A memory fabric is an emerging low-latency and high-bandwidth lossless cluster interconnect that enables tight integration of cross-node memory and accelerators into host systems. Compared with communication fabrics (such as Ethernet [12], InfiniBand/RoCE [13], AMD's Infinity Fabric [2], Intel's Omni-Path [14]), a memory fabric provides the load/store access model and is inherently integrated into the memory hierarchy of a host processor with restricted cache coherence

Interconnect	Vendor	Active Development	Specification	Product Demonstration
Gen-Z [10]	HPE/Gen-Z Consortium	2016-2021	Gen-Z 1.0/1.1	Gen-Z Media Kit [9], Gen-Z ChipSet for ExtraScale Fabric [54]
CAPI/OpenCAPI [18]	IBM/OpenCAPI Consortium	2014-2022	CAPI 1.0/2.0, OpenCAPI 3.0/4.0	BlueLink in POWER9 [73]
CCIX [4]	Xilinx/CCIX Consortium	2016-now	CCIX 1.0/1.1/2.0	CMN-700 Coherent Mesh Network [23]
CXL [6]	Intel/CXL Consortium	2019-now	CXL 1.0/1.1/2.0/3.0	Omega Fabric [17], Leo Memory Platform [24]

Table 1: A list of commodity memory fabrics. CAPI=Coherent Accelerator Processor Interface. CCIX=Cache Coherent Interconnect for Accelerators. CXL=Compute Express Link. Gen-Z and OpenCAPI have merged into CXL in the last two years.

support. There are three major driving forces behind the design and implementation of such interconnects:

- **#1:** The increasing gap between core counts and memory channel bandwidth per core on a server processor [5, 15], coupled with nearly stagnant memory bandwidth improvement, limits the attainable per-core computing throughput. We have seen an 8×/6× increase in core counts of the AMD/Intel x86 server CPU in the last 13 years, while the per-core memory bandwidth has come to a standstill [1];

- **#2:** The inevitable processing overheads of the traditional networking stack waste computing cycles at both local hosts and remote devices [35, 65], complicating the way to support disaggregated memory/accelerators. Researchers resort to designing customized communication substrates to facilitate host-device data transfer [53, 72, 79];

- **#3:** The lack of flexible physical resource scaling squanders the cost-efficiency benefit of disaggregation. Most of today's remote memory or computing chassis are not standalone, relying on a server host to provide connectivity via on-/off-chip interconnects. Hence, building a disaggregated computing box requires equipping a beefy server with sufficient bandwidth provisioned [11, 16, 20, 22].

Compute Express Link (CXL). Memory fabrics are generally realized via a specialized I/O bus technology. We base our description on the Flex Bus architecture of CXL [6]–the mainstream memory fabric since absorbing Gen-Z and Open-CAPI (Table 1). The Flex Bus (Figure 1-a), built atop PCIe [19], provides root access points to the host processor, supports both native PCIe and CXL modes, and is organized into three layers (i.e., physical layer, link layer, and transaction layer). The physical layer prepares transmitted data upon receiving upper link-layer packets, deserializes the data received from the physical bus, and converts it to the appropriate format based on the bus operating mode. It supports both 68B and 256B flit modes, runs at most 64 GT/s link speed, and allows x4/x8/x16 bifurcation configurations. The link layer, working as an intermediary stage, provides reliable transmission between two endpoints using a hop-by-hop based credit-based flow control. Each entity along the path allocates credits to downstream ports based on its buffer capacity, uses a credit update protocol to track inflight flit transmission, and runs an overcommitment scheme to improve bandwidth utilization.

The transaction layer provides channel semantics and communication primitives. Specifically, CXL offers three types of channels: (1) *CXL.cache*, maintaining a small-sized and fully coherent cache using standard processor snoop filter mechanisms; (2) *CXL.mem*, allowing host CPUs to access device-side memory via sheer load/store instructions, while coherence operates under either host-only or device-only modes; (3) *CXL.io*, wrapping around the basic PCIe protocol with some enhancements (e.g., non-coherent read/write).

CXL allows scalable switching using a combination of Port Based Routing (PBR) and Hierarchy Based Routing (HBR) since CXL 2.0. It supports both direct and indirect topologies akin to the Ethernet network. A CXL fabric contains several domains connected via HBR links, where each one consists of one or more switches that are PBR capable and interconnected with PBR links. An intra-domain switch uses 12-bit PBR IDs to address up to 4096 unique edge ports. It preserves the semantics and properties (like reliability and orderliness) of different channels via a series of buffering, backpressure, and adaptive routing techniques. The switching routing table is generally filled up by a central fabric manager.

2.2 Composable Infrastructure

Memory fabrics bring computing resources close, yielding system composability. A composable infrastructure consists of the following components (Figure 1-b):

- **#1: Fabric Host Adapter (FHA).** It connects to the access root port (RP) of host CPUs. An FHA converts channel requests into fabric routable packets (or flits) following the protocol specification and transmits them to the wire. Similarly, when an adapter receives responses, it parses the packets, obtains replied data or completion signals, and delivers them to the processor execution pipeline;

- **#2: Fabric Switch (FS).** An FS consists of upstream ports (UPs) for FHA connectivity, downstream ports (DPs) for remote devices/memory modules, and internal switching tables associated with efficient traffic orchestration. For example, the Intelliprop's Omega testbed [17] employs a non-blocking crossbar topology between UPs and DPs. Upon initialization, an FS discovers its connected components, self-initializes the routing structure, and fills up the switching table entries based on the topology;

- **#3: Fabric Endpoint Adapter (FEA).** An FEA stays close to the remote device, operating as a target responder, responsible for fabric protocol processing and converting between the fabric packets and device-dependent primitives. Some adapters (such as FA4004 [7]) also perform integrity
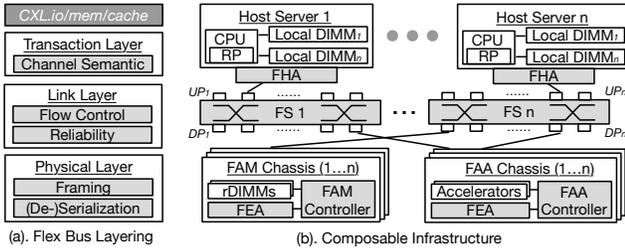
Figure 1: An architectural overview of CXL layering and composable infrastructure.

Memory Hierarchy	Latency (ns)	Throughput (MOPS)
L1 Cache Read/Write	5.4/5.4	357.4/355.4
L2 Cache Read/Write	13.6/12.5	143.4/154.5
Local Memory Read/Write	111.7/119.3	29.4/16.9
Remote Memory Read/Write	1575.3/1613.3	2.5/2.5

Table 2: The cacheline (64B) read/write performance comparing a local and a remote DIMM on the Omega Fabric testbed [17]. MOPS=Million Operations per Second.

checking, request steering, and transmission speed synchronization. Others (like XMM CXL E3.S [26]) directly embed the FEA into the endpoint device;

- **#4: Fabric Attached Memory (FAM) and Fabric Attached Accelerators (FAA).** These are specialized and standalone chassis, enclosing a dedicated SoC backplane, a power supply, and a controller for configuration and management. CXL defines three types of devices based on selected channel semantics. CXL Type 1 device extends the traditional PCIe device with a coherent cache, while Type 2 equips the device with both host-managed memory and coherent cache and Type 3 works as a memory expander. For example, the FAM chassis in the Omega testbed [17] encloses six CXL E3.S memory modules [26], while the FAA in GigaIO Fabrex [8] holds up to eight accelerators.

3 DIFFERENCES OF MEMORY FABRICS

Difference #1: Synchronous Execution. The memory fabric is inherently integrated into the memory hierarchy and execution pipeline of the host processor. This is in significant contrast to the communication fabric, which interacts with the CPU asynchronously in a submission-completion fashion. Consider a DMA transfer between a processor and a PCIe device as an example. The processor builds a DMA instruction word, then submits it to a device-side DMA engine to trigger data movement. Upon completion, the engine notifies the processor through dedicated interrupt signals.

For a memory fabric, load/store requests are generated transparently from the memory hierarchy and processed synchronously. A memory read is issued when there is a miss from the last level cache, while a memory write happens if the victim buffer needs to flush. During the data transfer, the current CPU pipeline is stalled and resumed after receiving the response. This indicates that (1) the host-side caching structure and CPU-assisted prefetching would transparently accelerate memory fabric performance; (2) the throughput of a memory fabric that a core can drive depends on its channel bandwidth capacity and the depth of the CPU pipeline.

Further, reads/writes over the memory fabric become more costly. Remote memory accesses are considerably slower than local memory accesses. On the Omega Fabric testbed,

the performance of a CXL2.0-like memory expander is nearly 10× slower than its local ARM computing complex (Table 2). A recent characterization study also reports the inferior performance of a CXL1.1 memory expander on Intel Sapphire Rapids servers [74]. As a result, host CPUs or FAAs have to waste more cycles when accessing FAMs, experiencing drastic performance degradation (especially when the application working set exceeds the LLC capacity) and unpredicted long execution stalls.

Difference #2: Eclectic Memory Nodes. The memory fabric enriches the memory node types based on how device memory is exposed and architected. Unlike communication fabrics, where data transfer speed is usually agnostic of data layout at the source and destination, the performance and efficiency of memory fabric hinge on the chosen memory node type and its access pattern and locality. These specialized memory nodes (discussed below) have been explored in early scalable and shared-memory machines [37, 45, 47, 48, 58, 59].

- **Fabric-attached CPU-less NUMA memory node**, a standalone memory expander with no processors. Most of today's CXL 1.1/2.0 Type 3 devices belong to this type. It stays in the same hierarchy as the host-side local memory. This node can be either owned exclusively by a host CPU or shared across multiple hosts (where the FEA needs to partition the capacity and enforce coherence at the device);

- **Fabric-attached CC-NUMA memory node**, exposing a shared address space over remote memory regions with cache-coherence support. This has been explored in previous shared-memory multiprocessor systems, such as Multicube [45], Aquarius [37], DASH [59], and FLASH [58]. It is usually realized via a cross-node, directory-based, write-invalidate cache coherence protocol within an FHA/FEA;

- **Fabric-attached Non-CC-NUMA memory node**. It operates similarly to the CC-NUMA one but lacks cache coherence, e.g., Intel's SCC [50] and IBM Cell's SPE [46]. This simplifies the hardware design of an FHA/FEA, but complicates the software design and implementation;

- **Fabric-attached COMA cache node.** The Cache-Only Memory Architecture (COMA), proposed in the 1990s, such as DDM [48, 49], reduces the average cache miss latency by dynamically migrating and replicating caching objects within memory. Each node exposes a portion of the global memory, augmented with a large cache and managed through a hierarchical directory scheme.

Difference #3: The Importance of Routable PCIe. Under the hood, most commodity memory fabrics are carried over Routable PCIe, where memory reads/writes are encapsulated as PCIe transactions. Researchers have studied how PCIe interacts with networking adapters within a host server [64], but developed little understanding of how well PCIe holds cross-node FAAs/FAMs at a modest scale. Early memory fabric prototypes [17] borrow the end-to-end virtual channel technique from Infiniband [13], simplifying the system deployment. Further, when employing an external PCIe switch, one should understand how concurrent PCIe transactions are interleaved and scheduled. For example, the FabreX PCIe switch [8] delivers less than 100ns non-blocking switch latency per port with up to 512Gbits/s bandwidth. When accessing a disaggregated Xilinx U55C FPGA card in a remote chassis, concurrent 64B PCIe writes can add 600ns more one-way latencies when compared with the case of holding the card within the host. When interleaved with 16KB writes, the average latency of 64B requests can be degraded drastically.

The problem fundamentally boils down to the efficiency of credit-based flow control (CFC) at scale. CFC has been proposed and developed for ATM and Infiniband networks [55–57], but receives less scrutiny in the context of routable PCIe. We discuss a few unexploited issues below:

- **Credit allocation.** The de facto scheme is an exponential ramp-up approach based on port bandwidth utilization [56]. A consistently heavily-used port would take more credits, leaving little room for other contending ports. Even with no bandwidth waste, this would create interference and stall transactions from other ports;

- **Credit-flow scheduling.** The scheduling discipline of CFC switches is credit-agnostic. Transactions receiving more credits are not prioritized over the ones with fewer credits. This would cause head-of-line blocking and credit waste, impacting both bandwidth and latency;

- **Credit coordination.** Credit starvation can backpropagate to upstreamed switch ports under scale-out scenarios. Such congestion can spread across a large victim area, yielding more credit waste and bandwidth loss.

Difference #4: Fast Context Switching among Execution Engines. Memory fabrics provide a lightweight and fast mechanism to create, checkpoint, and ship computing contexts. Compared to communication fabrics, when triggering computations in an Ethernet-attached accelerator, one needs to first set up the communication channel to pass control signals and data through a customized networking stack, then design a mechanism to launch the computing kernel by filling up the execution context on the disaggregated device (such as registers and push/pull buffers), and finally provide programming friendly interfaces to host applications [38, 72].

The memory fabric brings an FAA close to the host and makes it behave as a local device in the following ways. First, it unifies the intra- and inter-interconnect such that no fabric I/O conversions or specialized protocol stacks are needed. Second, saving and restoring execution contexts either within or across hosts becomes flexible given the partially shared address space offered by various types of FAMs. For example, CXL type 2 devices can even perform interleaved execution, such as OpenCL cooperative kernels. Third, developers can reuse existing device drivers, interfaces, and system/application APIs, instead of doing API remoting [43].

Difference #5: Passive Failure Domains. Memory fabrics bring *passive* failure domains into the composable infrastructure. In communication fabrics, both source and destination are active execution entities (e.g., general-purpose servers) that employ fault-tolerant mechanisms to ensure data consistency and minimize the recompute overheads.

However, in composable infrastructures, two characteristics stand out: (1) hosts and remote devices usually stay in different power domains and can fail separately; (2) the controller of a FAM/FAA has provisioned little computing resources for failure handling, making previous solutions inapplicable. More importantly, the fault-tolerant scheme should be resource-frugal and impacts the application performance little. Carbink [78] has made early exploration for RDMA-based far memory by outsourcing resource management and monitoring tasks to a central memory manager and employing a series of reliability/optimization techniques, such as erasure coding and remote memory compaction.

4 FABRIC-CENTRIC COMPUTING

The characteristics of memory fabrics have motivated us to rethink how to build rack/cluster-scale systems on composable infrastructures. In the context of the communication fabric, computation engines are our primary resources. We usually view the communication substrate as an independent layer and apply different kinds of optimization mechanisms to facilitate communication for maximizing computation efficiency. For example, these techniques include minimizing networking stack overheads [32, 51, 61, 67, 77], designing new transport protocols [34, 63, 69], performing efficient flow scheduling and load balancing [29–31, 68], streamlining/overlapping communication with computation [44, 62, 65], etc. However, these approaches may be invalidated or ineffective given the memory fabric's characteristics and capabilities. As a straightforward example, optimizing the host networking stack is unnecessary because the maximum remote memory throughput that a core can drive depends on the number of outstanding load/store instructions that it can submit in its pipeline.

We propose a new computing paradigm–called **Fabric-Centric Computing (FCC)**–that views the memory fabric as a first-class citizen to instantiate, orchestrate, and reclaim computations over composable infrastructures. FCC materializes the memory fabric's capabilities as primitives and abstractions, defines the interaction interface and templates for different fabric-attached components, and coordinates computations based on the communication limits of the fabric. In addition to performance and efficiency goals, FCC strives to retain other properties: (1) on-demand scaling, where the computing and communication resources can scale up/down based on application demands; (2) flexibility, where it allows arbitrary mapping between host servers and fabric devices; (3) over-provisioning, indicating that one can preserve a large number of computation resources; (4) general compatibility, which means that applications can take advantage of the memory fabric with few modifications. We envision FCC based on the following design principles:

Design Principle #1: Data movement as a managed service. Memory fabrics require us to rethink data transfer from three perspectives. First, data movement is only associated with two entities: an initiator and an executor, without a completion facilitator, as in the communication fabric. To mitigate stall-induced overheads, one should decouple the initiator from the executor based on data movement cost and computing urgency. Within the same memory hierarchy, a movement request is triggered by the source core/FAA but can be executed by either source, destination, or neighboring core/FAA. Second, the local memory hierarchy of a host processor or FAA can transparently accelerate the access performance of FAMs, indicating that exploiting spatial and temporal locality will bring many optimization opportunities. Third, since reads/writes are instantiated by CPUs/FAAs and served by FAMs, this yields a new type of unexplored rack/cluster-scale traffic matrix.

FCC advocates data movement as a specialized and managed service. It should effectively blend synchronous and asynchronous communications–to hide remote access overheads and facilitate CPU/FAA execution. Load/store requests that are latency-sensitive or tightly coupled with the current execution context (e.g., data structure traversal) are performed synchronously. In this case, FCC enhances them with SW/HW-assisted caching and prefetching optimizations to exploit locality benefits, e.g., preloading the application working set or partitioning the cache based on memory access analyses. Other data transfers submitted by CPUs/FAAs are then delegated to dedicated migration agents (in the same memory domain) and orchestrated via a central module that enforces control-plane policies (e.g., remote memory bandwidth throttling). This design also allows FCC to easily integrate heterogeneous memory nodes embodying different access characteristics and capabilities.

Design Principle #2: Host-assisted memory node type-conscious data structure. Memory fabric introduces different types of memory nodes (§3) with diverse performance characteristics and capabilities. Designing an efficient data structure should consider the memory layout across different memory nodes, their access distribution, and data locality. Researchers have extensively explored NUMA-aware or NUMA-efficient data structure design [36, 60, 71]. For example, node replication [33, 36] is a powerful technique that transparently replicates data references across different NUMA regions by analyzing the memory and node affinity, which would benefit fabric-attached CC-NUMA memory nodes. But it is inapplicable for the CPU-less NUMA one since the remote memory expander has no processing units. Far memory data structures [28] target one-sided access mode, augmenting remote memory with three hardware primitives, thereby reducing far accesses. However, it ignores the fact that far memory accesses can indeed be hidden by the local processor's caches. AIFM [70] developed an RDMA-based object remote memory without considering some memory fabric capabilities, e.g., far memory accessed via load/store instructions instead of traversing networking stacks on both sides.

In FCC, we propose a new host-assisted memory node type-conscious data structure design based on an *active and unified heap*. FCC instantiates memory regions/segments from different fabric-attached memory nodes as a series of various-sized memory bins, and then uses a heap manager for object allocation and reclamation. Under the hood is a runtime system that (1) profiles the object's access characteristics and the underlying memory node's availability; (2) effectively migrates objects across various memory nodes (including host local memory) based on the object temperature, concurrent access model, and memory node capabilities. Developers use backward-compatible programming interfaces (like Smart Pointer) to port or build data structures. We aim to achieve maximum performance gains without exposing the peculiarities of memory nodes to programmers. For example, frequently accessed objects would be cached at the host processor and moved into the local host memory.

Design Principle #3: Idempotent tasks and hardware cooperative scalable functions. Reliable and scalable execution on composable infrastructures is paramount to maximize its efficiency. In FCC, we advocate two abstractions to achieve this goal. First, to handle the separated and passive failure domain issue, we propose *Idempotent Task*, inspired by the idempotent processor architecture to tackle the inefficiency of speculative execution [41] from the computer architecture community. The key idea is leveraging the principle of idempotence to break programs into regions of code that can be recovered through simple re-execution. FCC extends this execution model to our domain, where an idempotent task can be re-executed and restarted multiple times without

jeopardizing correctness and data consistency guarantees. MODC [52] explores a similar design. To realize it, we need to develop (1) a new compilation framework to identify idempotent code regions and encapsulate them as idempotent tasks; and (2) a split runtime execution architecture–learned from the *tasklet* and *top-half/bottom-half* interrupt architecture of the OS kernel–to deploy them across host servers, FHA/FEA, and FAAs.

Second, we propose a hardware cooperative scalable function for FAAs that extends the capability of today's SR-IOV and scalable functions [21] with an active execution context. In addition to dedicated queueing resources, each function defines (1) a domain-specific processing core; (2) a list of message handlers, such as the actor programming model [27]; (3) an execution coordination sublayer that encodes how to interact with other co-located functions. The entire design resembles the TAM (Threaded Abstract Machine) and active messages [40, 75]. We expect to build a basic template for FAAs to inherit, serving as the hardware execution substrate for idempotent tasks.

Design Principle #4: Fabric central arbitrator through dedicated lanes. Given the idiosyncrasies and non-determinism of routable PCIe, FCC proposes an in-band centralized fabric arbiter for bandwidth allocation, congestion control, and flow scheduling. Similar ideas have been explored in the data center network [39, 66]. We believe this would be more applicable to memory fabrics for two reasons: (1) the intra-server interconnect bandwidth has increased substantially with the advent of PCIe Gen6/Gen7 and the growing I/O capabilities of server processors. A dedicated control channel would bring little bandwidth waste. This is similar to traditional mainframes (e.g, CDC 6000 series, IBM 360/370), where control and data signals are separated; (2) The end-to-end RTT of a 64B flit at the data link layer in an unloaded scenario can be up to 200ns, yielding little impact on even small PCIe transactions. Further, FCC would incorporate a programmable interface with the control lane to query, reserve, and reclaim credits, and expose it to the application layer via some programming abstraction (such as distributed futures [76]), enabling compute-fabric co-design.

5 UNIFABRIC: A PROPOSAL

We sketch the UniFabric that abstracts the memory fabric and works as a new intermediate stack for composable infrastructures. Essentially, it is a distributed runtime system that provides a collection of new/renovated programming abstractions and system services at the rack/cluster scale. UniFabric follows the aforementioned design principles and encompasses the following components: (1) an elastic transaction engine for (asynchronous) data movement. It decouples the initiator and an executor, providing a generic primitive, like eTrans(*src_addr_list*, *dst_addr_list*, *immediate_bit*, *attributes*, *ownership*). (2) a unified heap manager. UniFabric will extend the existing MemKind library [25] to incorporate different kinds of memory nodes and expose an active heap. One can reuse existing data structures and port unmodified applications using compatible programming interfaces. We will also provide a list of new data structures specially optimized for certain fabric-attached memory nodes; (3) a compilation and execution framework to develop idempotent tasks. It also contains a hardware template and some referenced FAA examples to build hardware cooperative scalable functions. Note that our design doesn't stick to any particular programming models; (4) a communication substrate atop routable PCIe, backing up elastic transactions. The orchestration, relying on dedicated in-band link layer lanes, is facilitated via a central arbiter to maximize communication efficiency.

Case study. We use the software-based MIMO baseband processing engine [42] as an example to demonstrate how to use the UniFabric layer. The engine resides between radios and the MAC, converting time-domain samples received from radios to bits used by the MAC and vice versa. It encompasses multiple uplink/downlink handling pipelines, further including a series of computing kernels, such as FFT/IFFT, equalization, (de)modulation, and encoding/decoding. When porting/implementing over UniFabric over a composable infrastructure, one should rethink data layout, computation placement, and communication structure. The first step is moving data objects (e.g., symbol frame and channel state information matrix) to our unified heap using existing/new data structures. Next, for each computing block, one should choose its backend execution engine, generate idempotent tasks via the compilation framework, and encapsulate them as hardware cooperative functions within an FAA. UniFabric frees programmers from the burden of FAA multiplexing and enables flexible stateful/stateless idempotent task migration, but requires applications to decide where the computation is performed and when it is moved. Finally, one should replace all asynchronous communications with elastic transactions and specify the ownership field that captures how completion is handled. Note that this case study only describes the high-level steps of using UniFabric for application development without discussing optimization opportunities.

6 CONCLUSION

This position paper proposes a new computing paradigm–called **Fabric-Centric Computing (FCC)** given the emerging trend of memory fabrics and composable infrastructures. FCC views the memory fabric as the first-class citizen in the system to instantiate, orchestrate, and reclaim computations over composable infrastructures. We discuss the uniqueness of this interconnect, propose new design principles, and sketch an intermediary layer to realize the vision of FCC.

REFERENCES

[1] 2023. AMD EPYC and Intel Xeon Core Counts Over Time. https://www.servethehome.com/updated-amd-epyc-and-intel-xeon-core-counts-over-time/.

[2] 2023. AMD Infinity Architecture. https://www.amd.com/en/technologies/infinity-architecture.

[3] 2023. CCIX Master Port Cache Coherency. https://docs.xilinx.com/r/en-US/pg118-system-cache/CCIX-Master-Port-Cache-Coherency.

[4] 2023. CCIX Specifications. https://www.ccixconsortium.com/library/specification/.

[5] 2023. CXL Borgs IBM's OpenCAPI, Weaves Memory Fabrics with 3.0 SPEC. https://www.nextplatform.com/2022/08/09/cxl-borgs-ibms-opencapi-weaves-memory-fabrics-with-3-0-spec/.

[6] 2023. CXL Specifications. https://www.computeexpresslink.org/download-the-specification.

[7] 2023. FA4004 - FabreX PCIe Gen4 Adapter. https://gigaio.com/products/fabrex-network-adapter-card/.

[8] 2023. FabreX Composable Platform. https://gigaio.com/products/fabrex-system-overview/.

[9] 2023. Gen-Z Micro Development Kits. https://genzconsortium.org/gen-z-micro-development-kits-powering-the-gen-z-ecosystem/.

[10] 2023. Gen-Z Specifications. https://genzconsortium.org/specifications/.

[11] 2023. Gigabyte Multi-GPU Systems. https://www.gigabyte.com/us/Enterprise/GPU-Server?fid=2235.

[12] 2023. IEEE 802.3 Ethernet. https://www.ieee802.org/3/.

[13] 2023. InfiniBand. https://en.wikipedia.org/wiki/InfiniBand.

[14] 2023. Intel Omni-Path Architecture. https://www.intel.com/content/www/us/en/products/network-io/high-performance-fabrics.html.

[15] 2023. Mashing up CXL and Gen-Z for Shared Disaggregated Memory. https://www.nextplatform.com/2022/10/31/mashing-up-cxl-and-gen-z-for-shared-disaggregated-memory/.

[16] 2023. NVIDIA DGX Systems. http://www.nvidia.com/en-us/data-center/dgx-systems/.

[17] 2023. Omega Fabric from IntelliProp. https://www.intelliprop.com/products-page.

[18] 2023. OpenCAPI Specifications. https://opencapi.org/technical/specifications/.

[19] 2023. PCIe Specification. https://pcisig.com/specifications.

[20] 2023. PowerEdge Accelerated Servers and Accelerators. https://www.dell.com/en-us/dt/servers/server-accelerators.htm.

[21] 2023. Sclable Functions. https://docs.nvidia.com/networking/display/BlueFieldDPUOSv385/Scalable+Functions.

[22] 2023. Supermicro GPU Systems. https://www.supermicro.com/en/products/gpu.

[23] 2023. The CMN-700 Mesh Network. https://www.arm.com/products/silicon-ip-system/corelink-interconnect/cmn-700.

[24] 2023. The Leo Memory Accelerator Platform from Astera Labs. https://www.asteralabs.com/products/cxl-memory-platform/.

[25] 2023. The memkind library. https://github.com/memkind/memkind.

[26] 2023. XMM CXL E3.S Memory Module. https://www.smartm.com/product/xmm-cxl-e3s.

[27] Gul Agha and Carl Hewitt. 1987. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*. 49–74.

[28] Marcos K Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*.

[29] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. 2010. Hedera: dynamic flow scheduling for data center networks.. In *Nsdi*.

[30] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*.

[31] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 435–446.

[32] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*.

[33] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. 2021. NrOS: Effective Replication and Sharing in an Operating System.. In *OSDI*.

[34] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. dcPIM: Near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*.

[35] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.

[36] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[37] M. Carlton and A. Despain. 1990. Multiple-bus shared-memory system: Aquarius project. *Computer* 23, 6 (1990), 80–83.

[38] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[39] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-Scale Computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

[40] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. 1991. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[41] Marc de Kruijf and Karthikeyan Sankaralingam. 2011. Idempotent processor architecture. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[42] Jian Ding, Rahman Doost-Mohammady, Anuj Kalia, and Lin Zhong. 2020. Agora: Real-Time Massive MIMO Baseband Processing in Software. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 232–244. https://doi.org/10.1145/3386367.3431296

[43] Micah Dowty and Jeremy Sugerman. 2009. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 73–82.

[44] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*.

[45] J. R. Goodman and P. J. Woest. 1988. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*.

[46] Michael Gschwind, H Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. 2006. Synergistic processing in cell's multicore architecture. *IEEE micro* 26, 2 (2006), 10–24.

[47] E. Hagersten and M. Koster. 1999. WildFire: a scalable path for SMPs. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. 172–181.

[48] Erik Hagersten, Anders Landin, and Seif Haridi. 1992. DDM-a cache-only memory architecture. *Computer* 25, 9 (1992), 44–54.

[49] Seif Haridi and Erik Hagersten. 1989. The cache coherence protocol of the Data Diffusion Machine. In *PARLE'89 Parallel Architectures and Languages Europe: Volume I: Parallel Architectures Eindhoven, The Netherlands, June 12–16, 1989 Proceedings 2*. 1–18.

[50] Jason Howard, Saurabh Dighe, Sriram R. Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and Rob Van Der Wijngaart. 2011. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *IEEE Journal of Solid-State Circuits* 46, 1 (2011), 173–183.

[51] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*.

[52] Kimberly Keeton, Sharad Singhal, Haris Volos, Yupu Zhang, Ramesh Chandra Chaurasiya, Clarete Riana Crasta, Sherin T. George, Nagaraju K. N, Mashood Abdulla K, Kavitha Natarajan, Porno Shome, and Sanish Suresh. 2021. MODC: Resilience for disaggregated memory architectures using task-based programming. *CoRR* abs/2109.05329 (2021).

[53] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with Amorphos. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*.

[54] Patrick Knebel, Dan Berkram, Al Davis, Darel Emmot, Paolo Faraboschi, and Gary Gostin. 2019. Gen-Z Chipset for Exascale Fabrics. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, 1–22.

[55] HT Kung and Koling Chang. 1995. Receiver-Oriented Adaptive Buffer Allocation in Credit-Based Flow Control for ATM Networks. In *Proceedings of INFOCOM'95*.

[56] H. T. Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-Based Flow Control for ATM Networks:Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*.

[57] NT Kung and Robert Morris. 1995. Credit-Based Flow Control for ATM Networks. *IEEE network* 9, 2 (1995), 40–48.

[58] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. 1994. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*.

[59] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. 1992. The Stanford Dash multiprocessor. *Computer* 25, 3 (1992), 63–79.

[60] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*.

[61] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

[62] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[63] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*.

[64] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*.

[65] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*.

[66] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*.

[67] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2015).

[68] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*.

[69] Costin Raiciu and Gianni Antichi. 2019. NDP: Rethinking datacenter networks and stacks two years after. *ACM SIGCOMM Computer Communication Review* 49, 5 (2019), 112–114.

[70] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[71] Tomer Shanny and Adam Morrison. 2022. Occualizer: Optimistic Concurrent Search Trees From Sequential Code. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.

[72] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. 2019. Direct Universal Access: Making Data Center Resources Available to FPGA. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*.

[73] Jeff Stuecheli, Scott Willenborg, and William Starke. 2019. IBM's next generation POWER processor. In *2019 IEEE Hot Chips 31 Symposium (HCS)*.

[74] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices.

[75] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*.

[76] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. 2021. Ownership: A Distributed

Futures System for Fine-Grained Tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.

[77] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. 2021. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*.

[78] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.

[79] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.