Revisiting Distributed Programming in the CXL Era

Teng Ma[1] Mingxing Zhang[2] Kang Chen[2] Jialiang Huang[2] Zheng Liu[1] Yongwei Wu[2]
[1]Alibaba Group [2]Tsinghua University

Abstract

As Moore's Law slows, distributed programming is increasingly prioritized to enhance system performance through horizontal scaling. This paper revisits the paradigms of message passing (MP) and distributed shared memory (DSM) regarding evolving interconnect technologies, particularly Compute Express Link (CXL). DSM's unified memory space offers simplicity in distributed programming by abstracting data communication complexities, but MP remains more prevalent due to its flexibility and programming-friendliness. We explore the memory sharing and pooling of CXL, which enable low-latency, coherent data transfers between multiple hosts. We address the complexities involved in managing shared states, particularly in the context of partial failures—termed Partial Failure Resilient DSM (RDSM). Thus we propose a memory management system named CXL-SHM that leverages reference counting and distributed vector clocks for robust memory management. To be compatible with the message passing model, we introduce a CXL-based RPC named HydraRPC, which utilizes CXL shared memory to bypass data copying and serialization.

1 Distributed Systems with CXL

From Ethernet to RDMA to CXL. The shift from Ethernet to Remote Direct Memory Access (RDMA), and now to Compute Express Link (CXL) [2], represents a major advancement in communication technologies. The adoption of RDMA has vastly revolutionized data centers, significantly affecting the design of distributed systems. It has notably lowered latency from over 100 microseconds to the millisecond level, while providing a one-sided memory read/write interface that greatly reduces overhead on the remote side.

Even with the new generation ConnectX-6 RDMA NIC, the latency of RDMA is still over $2\mu s$. The subsequent introduction of CXL represents the latest advancement, aiming to provide high-speed and, critically, coherent data transfer between different nodes. For instance, DirectCXL [5] connects a host processor with remote memory resources, enabling load/store instructions with a remote CXL memory access latency of approximately 300 nanoseconds.

CXL proposes **memory pooling**, enabling the creation of a global memory resource pool, thus optimizing the overall utilization of memory. This is achieved through CXL switch [3] and memory controller that facilitates dynamic allocation and de-allocation of memory resources. Pond is the first full-stack memory pool that meets the requirements of cloud providers. It deploys a CXL-based memory pool to the Microsoft Azure cloud platform [10]. Currently, most mainstream cloud service providers have announced their plans to support the CXL-based memory pool.

The 2nd Workshop on Hot Topics in System Infrastructure (HotInfra'24), co-located with SOSP'24, November 3, 2024, Austin, TX, USA

Table 1. Message Passing v.s Distributed Shared Memory

	Message Passing	Distributed SHM	New Features
Interface	Send/Recv	Load/Store/CAS	Finer-grained API
Com/Mem Relationship	Tightly Coupled	Separation	Higher Utilization
Communication Paradigm	Pass-by-Value	Pass-by-Reference	Faster Exchange
Data/State Architecture	Share Nothing	Shared Everything	Cheaper Migration

Moreover, the forthcoming CXL 3.0/3.1 [1] version promises **memory sharing**, which would allow the same memory region to be mapped across multiple machines. In such a setup, the hardware would automatically manage cache coherence for concurrent accesses from different machines, essentially implementing a hardware-based shared memory model. This development opens up exciting possibilities for the future of distributed systems.

Message Passing v.s, Distributed Shared Memory. Since the restrictions of the Von Neumann architecture, individual machines have limited resources and thence poor scalability. Distributed systems play a vital role in the challenge of scalability. For distributed programming, two main paradigms prevail: message passing (MP) and distributed shared memory (DSM), the debate over MP versus DSM has continued from the 1960s. DSM is designed to be more ease-of-use because it provides a unified memory space that abstracts away the complexities of explicit data communication. This design allows for programming distributed applications almost as straightforwardly as multi-threaded applications. However, in practice, the less intuitive MP model is *more commonly used* than DSM. This preference is often attributed to the assumption that the high cost of remote communication significantly hampers the efficiency of DSM.

However, rapid advancements in networking and interconnect technologies have profoundly impacted our understanding of this field. Especially with the emergence of Compute Express Link (CXL), we are presented with an opportune moment to revisit our perspectives on traditional distributed programming paradigms.

Revisit Distributed Applications in CXL era. Even now CXL's hardware implementation is still catching up to its ambitious specifications. It is a good time to revisit the distinctions between MP and DSM in the context of the forthcoming CXL era. As shown in Table 1, we understand their differences and identify optimal application scenarios. At first glance, the principal difference between these paradigms is the interface they utilize. MP relies on a traditional Send/Receive interface, while CXL is better aligned with DSM by providing a fine-grained remote load/store instruction set. Message passing systems assume a tightly coupled architecture, where each node can only access its local memory. In contrast, CXL-enabled DSM systems naturally fit into a disaggregated architecture that separates compute and memory resources, allowing for more flexible and efficient resource utilization. For data communication, message passing generally involves data payloads being copied from one node to another, a pass-by-value approach. DSM needs to exchange references, embodying a pass-by-reference method. This facilitates access to only the necessary data subparts and enables in-place updates. Finally, the DSM's global memory space accessibility implies a shared-everything data/state architecture. This is particularly beneficial for quickly migrating workloads. For instance, resolving load imbalances in a shared-nothing architecture necessitates heavy data repartitioning, whereas in a shared-everything setup, only small metadata representing data partition ownership needs to be exchanged.

In summary, CXL-based DSM excels in situations that need high flexibility. CXL architecture naturally supports this flexibility by offering both a dynamic and efficient way to allocate and access remote memory resources. This is essential for systems that need to scale quickly and efficiently.

2 CXL-based Distributed SHM Paradigms

Challenges. Nevertheless, transitioning to the CXL-based DSM paradigm isn't just about leveraging advancements in hardware. It involves embracing a unified memory space that hosts shared states, which in turn enables quicker data transfers and migrations while uncoupling computing from memory for enhanced scalability. However, managing these shared states also presents more complexity than the conventional shared-nothing architecture, given the potential for concurrent access and the partial failures that may arise.

To wrap things up, we can see that the root of our challenges stems from the fact that, in our DSM model the shared distributed objects and the clients that hold their references have separate failure domains. Thus, it allows clients to freely join, exit, or even face failures during the process, and these clients can arbitrarily create, release, and even exchange references to remote memory spaces. While very user-friendly, this flexibility comes with a significant challenge in memory management. We summarize this new model as Partial Failure Resilient DSM (RDSM). To distinguish it from full failure scenarios where all clients crash simultaneously. Handling full failures might seem more straightforward, but we believe tolerating partial failures is crucial to making DSM usage more widespread.

DSM Model of CXL-SHM. To address these challenges, we propose CXL-SHM [15]. It uses reference counting to reduce the manual workload involved in reclaiming remote memory that has been allocated. However, a standard reference counting system is not robust against system failures.

CXL-SHM offers a unified memory address space for hosting shared states. These shared states facilitate faster transfer and migration and decouple compute and memory, which jointly lead to better scalability. However, the management of memory becomes more challenging due to possible concurrent access and partial failures arising from these shared states.

Fault Tolerance of CXL-SHM. To further demonstrate this problem, here we divide the procedure of automatic reference count maintenance into two sub steps, Which will help us to demonstrate the corner failure cases. For attaching a reference, we first need to increase the reference count. Then we link the reference by setting its value to the address of the referenced space, which we call the modification reference step. Similarly, detaching a reference also involves only two steps, decreasing the reference count and setting the value of the reference to NULL.

However, a basic reference count system cannot handle failures. And this is where things get tricky. Although we have only two simple steps to maintain the reference count, the order matters. We run into issues if there is a crash between these two steps. For instance, if we increment the reference count but do not set the reference value because of a failure, we will encounter a memory leak. A typical solution is to use the lock to make the modified reference count idempotent and record this modified count in the redo log for recovery. Unfortunately, it is only suitable for full failure scenarios where all clients exist simultaneously. Cascade blockage will be an issue if a client crashes after obtaining a lock.

To overcome this problem, we replaced lock operations from the original algorithm with non-blocking maintenance of a distributed vector clock. It can help us maintain a global timeline asynchronously. Additionally, we have optimized the modifying reference count operation to utilize the CAS instruction for atomic modification of the reference count and additional meta-information. This includes the ID of the last client that successfully modified this field, as well as the local era of that client of this maintenance transaction. This operation serves as the commit point for the entire transaction, ensuring that the modifying reference count step is never re-executed. During recovery, the subsequent steps will be redone blindly, as they are naturally idempotent.

3 CXL-based Message Passing Paradigms

Challenges. Remote Procedure Call (RPC) is an essential technology of MP-based distributed programming [14]. It permits functions to run on a remote server as if they were local calls. It simplifies the interaction between client and server by abstracting the intricacies of message passing. RPC

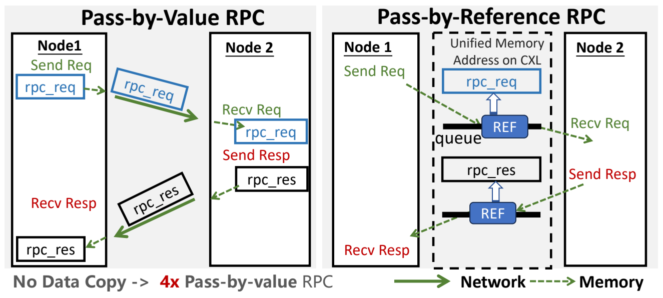

Figure 1. HydraRPC Architecture.

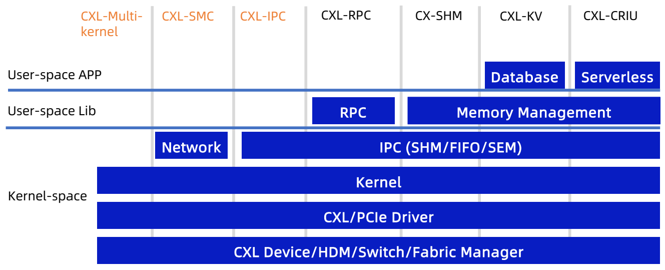

Figure 2. Software Stack of Distributed Applications.

enables developers to create distributed applications more efficiently, making it a crucial component in datacenters. For modern RPC implementations, performance and scalability are vital. Network latency and data transfer for communication and (de)serialization can hinder performance [13], while congestion at both the hardware (network) and software (buffer management) levels can limit scalability [4]. When deploying CXL-based RPC, it is crucial to design the architecture with shared memory abstractions of CXL.

CXL-RPC Abstractions. We designed a CXL-based RPC named HydraRPC [11] to minimize network latency and avoid costly memory operations using CXL shared memory. In traditional pass-by-value RPC, message passing typically involves copying data payloads from one server to another [8]. In contrast, shared memory abstractions merely exchange references, representing a pass-by-reference approach. This allows access to only necessary parts of the data and supports in-place updates, providing notable benefits in various scenarios. Utilizing CXL shared memory presents two major advantages: 1) it bypasses the network, and 2) it offers byte-addressable access, enabling the development of dynamic data structures with link pointers and permitting in-place modifications. This prevents the need for (de)serialization, thereby improving performance.

As shown in Figure 1, CXL provides unified memory addresses for message queues and data areas. HydraRPC employs two message queues for each connection, each queue entry consists of a reference (`REF`) to a data area and an embedded arrival flag. In the Data Plane, The client writes request data to the data area and updates the request message queue without waiting for a server response, while the server polls for the arrival flag. Upon receiving a request, the server uses the offset from the queue entry to execute the request, following a "run-to-completion" principle for lightweight requests. The server can directly access preallocated memory for efficiency. The server appends a response entry to the response message queue, and the client polls this queue and finally acknowledges arrival.

Implementation Details. Instead of general Load/Store memory access instructions, we employ non-cachable sharing to bypass the CPU cache, including two mechanisms: MTRR (Memory Type Range Registers) [9] and non-temporal memory access such as `clflush/prefetch` for individual scenarios. To achieve low CPU utilization and high performance, we use SSE3 power reduction instruction during polling-based notifications. Besides, HydraRPC supports the sliding window protocol to prevent access congestion.

4 Conclusion and Future Directions

Evolution of Cloud Computing. Implementing CXL-based DSM broadly poses a significant challenge in the industry. We discovered that the benefits of DSM align seamlessly with the progression of cloud computing. Consequently, most users can take advantage of CXL pooling and sharing. However, the refinement of billing granularity has become increasingly precise with the advancement of cloud infrastructure. This refinement allows both users and cloud providers to achieve greater overall resource utilization. Nevertheless, traditional applications often find it difficult to leverage these advancements due to their inherent lack of elasticity. We believe that it is one of the most crucial challenges that future CXL research should address.

Evolution of Software Stack. As shown in Figure 2, we proposed the hierarchical software stack architecture for Compute Express Link (CXL). We highlight the combination of user-space and kernel-space components, with the architecture structured in several tiers, enabling users to leverage the pooling and sharing features of CXL across different scenarios. At the *kernel-space* level, we implement various kernel mechanisms that leverage CXL protocols, including CXL-based Multi-kernel for clusters, CXL-SMC for high performance socket communication based on SMC [7], and CXL-IPC for remote interprocess communications (similar to DIPC [12]). These components ensure intra-server communication mechanisms, enabling high-speed data transfer and coordination. CXL-SHM [15] and HydraRPC [11] are two important components for memory management and RPC separately in our big picture for the *user-space library*. Additionally, our system allows *user-space applications* such as database and serverless to enjoy scalable, on-demand resource utilization. For example, by employing a memory re-purposing mechanism within the CRIU, TrEnv [6] offers serverless applications the hot-start ability while simultaneously reducing memory utilization. The above software stack can provide users with pooling and sharing capabilities in various scenarios.

References

[1] 2022. Compute Express Link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf.

[2] 2022. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/home.

[3] 2023. World's first CXL 2.0 and PCIe Gen5 Switch IC. https://www.xconn-tech.com/product.

[4] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[5] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 287–294.

[6] Jialiang Huang, Mingxing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and Yongwei Wu. 2024. TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA.

[7] IBM. 2022. SMC for Linux on IBM Z and LinuxONE. https://linux-on-z.blogspot.com/p/smc-for-linux-on-ibm-z.html.

[8] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs.. In *OSDI*. 185–201.

[9] Kernel. 2022. MTRR (Memory Type Range Register) control. https://docs.kernel.org/arch/x86/mtrr.html.

[10] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ASPLOS 2023*.

[11] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, et al. 2024. HydraRPC:RPC in the CXL Era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 387–395.

[12] DS Molojicic, Alan Langerman, David L Black, Michelle Dominijanni, Randall W Dean, and Steven J Sears. 1997. Concurrency: a case study in remote tasking and distributed TPC in Mach. *IEEE concurrency* 5, 2 (1997), 39–49.

[13] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI* (Carlsbad, CA, USA). USENIX, 561–577.

[14] Yongwei Wu, Teng Ma, Maomeng Su, Mingxing Zhang, CHEN Kang, and Zhenyu Guo. 2019. RF-RPC: Remote Fetching RPC Paradigm for RDMA-Enabled Network. *IEEE Transactions on Parallel and Distributed Systems* 30, 7 (2019), 1657–1671. https://doi.org/10.1109/TPDS.2018.2889718

[15] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 658–674.