HydraRPC: RPC in the CXL Era

Teng Ma, *Alibaba Group;* Zheng Liu, *Zhejiang University and Alibaba Group;* Chengkun Wei, *Zhejiang University;* Jialiang Huang, *Alibaba Group and Tsinghua University;* Youwei Zhuo, *Alibaba Group and Peking University;* Haoyu Li, *Zhejiang University;* Ning Zhang, Yijin Guan, and Dimin Niu, *Alibaba Group;* Mingxing Zhang, *Tsinghua University;* Tao Ma, *Alibaba Group*

https://www.usenix.org/conference/atc24/presentation/ma

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

HydraRPC: RPC in the CXL Era

Teng Ma[1], Zheng Liu[3,1], Chengkun Wei[3], Jialiang Huang[1,2], Youwei Zhuo[1,4], Haoyu Li[3]
Ning Zhang[1], Yijin Guan[1], Dimin Niu[1], Mingxing Zhang[2*], Tao Ma[1]
[1]Alibaba Group, [2]Tsinghua University, [3]Zhejiang University, [4]Peking University

Abstract

In this paper, we present HydraRPC, which utilizes CXL HDM for data transmission. By leveraging CXL, HydraRPC can benefit from memory sharing, memory semantics, and high scalability. As a result, expensive network rounds, memory copying, and serialization/deserialization are eliminated. Since CXL.cache protocols are not fully supported, we employ non-cachable sharing to bypass the CPU cache and design a busy-polling free notification mechanism. This ensures efficient data transmission without the need for constant polling. We conducted evaluations of HydraRPC on real CXL hardware, which showcased the potential efficiency of utilizing CXL HDM to build RPC systems.

1 Introduction

Remote Procedure Call (RPC) is a fundamental technology in distributed systems that facilitates network communication by allowing functions to execute on a remote server as the local calls. It simplifies client/server interactions by hiding the complexities of the underlying communication processes [15]. With RPC, developers can write distributed applications more efficiently, as the client-side code resembles a regular function call, while the server-side handlers mirror standard procedures. RPC has become an essential part of communication infrastructure in datacenters: Protobufs [21], Thrift [40], and Finagle [20]. Performance and scalability are critical for modern RPC implementations. Additional network latency and data copy for communication and (de)serialization can impede performance, while congestion at the hardware (network) and software (buffer management) levels can affect scalability.

Recently, CXL (Compute Express Link), an industry-supported cache-coherent interconnect for machines [3], memory and devices, serves as a potential catalyst for the advancement of RPC. CXL enables seamless data sharing and reduced latency with CXL-based HDM (host-managed device memory) by providing a high-speed, efficient, and flexible interconnect. This environment fosters the development of more sophisticated and efficient RPC mechanisms that can leverage the increased throughput and lower overhead. Especially, memory-sharing feature paves a new way to enable multiple machines to access the same data [4], a replacement for heavy network communication.

The implementations of traditional RPC are based on message passing, but CXL only provides shared memory abstractions. Thus, there are three problems we should consider: 1) How to design the control plane of RPC and the RPC protocol to fully exploit CXL HDM's potential performance; 2) CXL HDM provides shared memory interfaces. Without message passing interfaces, there is no efficient and easy-to-use mechanism to notify CPU of request/response arriving; and 3) how to manage CXL HDM while using it in the RPC scenario.

In this paper, we propose a new RPC paradigm named HydraRPC, to tackle the problems while building up RPC with CXL. It leverages CXL HDM shared between multiple machines to avoid the expensive network round, memory copy and (de)serialization. Instead of general Load/Store memory access instructions, we employ non-cachable sharing to bypass the CPU cache, including two mechanisms for individual scenarios. To achieve low CPU utilization and high performance, we use the power reduction instruction of SSE3 during polling-based notifications. Furthermore, HydraRPC supports sliding window protocol to prevent access congestion.

The results show that HydraRPC can achieve 620KOPS throughput for each RPC connection, which is 1.6/3.1× higher than mRPC [17] and RDMA-based RPC. The lowest RPC latency can reach nearly 1.47μs. Larger latency reductions are possible by fully avoiding network operations with CXL HDM. In addition, HydraRPC has good scalability which can scale to more than 96 RPC connections for each server with only 19% performance degradation.

We make the following contributions: (1) We demonstrate that CXL is a powerful tool that enhances the performance and efficiency of RPC. With the emerging memory sharing feature of CXL, it is time to reconstruct RPC protocol using CXL HDM. (2) A CXL-based RPC architecture named HydraRPC that mitigates data copy and (de)serialization by applying several CXL-specific designs. (3) An implementation of HydraRPC in a real CXL platform and evaluations on

*Corresponding author. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216, 61877035), Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), and Alibaba RI Program.

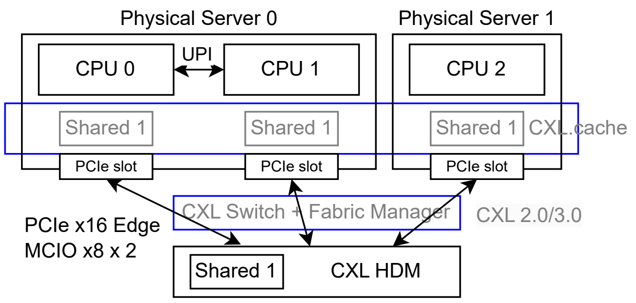

Figure 1: CXL Memory Sharing.

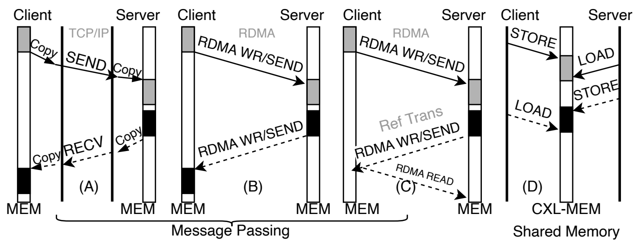

Figure 2: Different RPC Paradigms.

synthetic workloads and real applications.

2 Background

CXL. Emerging in 2019, CXL is one of the most promising technologies not only for memory extension/expansion/disaggregation but also for high-performance interconnect between machines. Unlike PCIe, CXL adds extra capabilities that let the CPU talk to devices and their linked memory with cache coherency using simple Load/Store instructions. The CXL standard defines three separate protocols: CXL.io for IO devices, CXL.cache for cache coherency, and CXL.mem for memory semantics. These three protocols can interoperate to create three types of devices (type 1/2/3). Particularly, type 3 CXL devices can support the extension of memory capacity which gains interest from both industries and academia [5], and major mainstream vendors such as Intel, AMD, and Samsung support it. To enable type 3 CXL devices, CXL.io and CXL.mem are necessary.

Memory Sharing. By combining these protocols, CXL blurred the line of host memory between different servers/OS. Current CXL standard [1] proposes to enable memory sharing for CXL HDM. For instance, a 2TB Pooled CXL Memory System showcased at the Flash Memory Summit combines eight Samsung 256GB CXL Memory modules with the XConn XC50256 CXL switch, enabling memory sharing among eight computing hosts [10].

Fig 1 shows coherent memory sharing [4]. With CXL.cache, CXL HDM can be shared by all the hosts in the coherency domain [2], and such memory is also known as Multi-headed Logical Device (MLD). It defines mechanisms to enforce hardware cache coherence between copies of the same data stored in different machines. For instance, CPU 0, CPU 1 and CPU 2 can access the same memory region (Shared 1).

The coherency model for each shared memory region is designated as either multi-host hardware coherency or software-managed coherency. To support multi-host hardware coherency, it is required to track the host coherence state for each cache line to varying extents, depending upon the MLD's implementation-specific tracking mechanism such as snoop filter or full directory. Hosts can perform atomic operations by getting exclusive control of a cache line in their cache. Once updated, this data is shared globally through cache coherence and handled using standard eviction methods.

In software-managed coherency models, the coherence state of hosts is not maintained by hardware but rather by host software through specific mechanisms that establish ownership of cache lines. While software takes the lead in managing coherence within certain memory regions, it can optionally utilize existing hardware coherency across multiple hosts for other regions to streamline ownership coordination.

3 Challenges of State-of-the-art RPC

Current state-of-the-art RPC adopts message passing as shown in Fig 2. The heavy network stacks burden traditional TCP/IP-based RPC (A), which consistently incur high CPU overhead. Plenty of research works explore RDMA to implement RPC to enhance performance while maintaining compatibility with existing applications. These RDMA-based works can be divided into two-sided (B) and one-sided (C) approaches. HERD [29] and FaSST [30] are designed for scalability, two-sided operation in the RPC design. However, the extra data copies are inevitable. Some works investigate ways to improve the communication performance of message passing under one-sided approaches. RFP [41, 46] proposes a pass-by-reference solution. For the RPC response, the server writes a reference to the client side, then client uses the reference to fetch the response RDMA Read. However, additional network round trips are needed to discover new incoming responses under the message passing paradigm.

Above all, message passing based RPC works have three major challenges:

Network Overhead. As shown in Fig 2, each RPC processing contains two messages or RDMA operations, which is at least 10× more expensive than normal memory access. Even with RDMA, the best network round trip is around 2μs, while this number is 300ns in CXL HDM.

Data Copy. Unlike traditional pass-by-value approach, modern distributed processing systems, like Ray [39], commonly use pass-by-reference to avoid expensive data copying

(Fig 2(A/B/C)). Instead of duplicating large request/response data, they use distributed storage to store the data and only transmit references of this data via RPC. It reduces data transfer overhead, leading to enhanced efficiency and performance.
Scalability. Multiple RPC connections should build up their buffer area, and buffer sharing should be limited to the same server. However, users need to be aware that an imbalanced RPC connections workload incurs high memory footprint and poor QoS of each connection. From a network perspective, numerous RPC connections can incur network congestion.

Regarding traditional pass-by-value RPC, message passing generally involves data payloads being copied from one server to another. As an alternative, shared memory abstractions require just the exchange of object references, embodying a pass-by-reference method. This facilitates access to only necessary data subparts and enables in-place updates, presenting distinct benefits for various scenarios.

Using CXL HDM instead of a distributed object store for pass-by-reference RPC (Fig 2(C)) offers two significant advantages. Firstly, the traditional object store architecture requires clients to communicate with the server over the network, leading to performance bottlenecks, especially for low-latency workloads. In contrast, CXL HDM avoids involving the network. Second, CXL.mem provides byte-addressable access, enabling the creation of dynamic data structures with link pointers and allowing in-place updates. This eliminates the need for (de)serialization, resulting in improved performance. *When implementing CXL-based RPC, it is essential to consider the architecture from the perspective of shared memory abstractions.*

4 Real CXL Platform

Hardware Architecture Our platform is based on Intel Agilex I-Series FPGA [26] and Archer City platform with Sapphire Rapids CPU. The server has 96 hyperthreads and 64GB DIMM, and the CXL FPGA with 16GB CXL HDM is inserted into the PCIe slot of one node and connected to the other node with two 8x MCIO cables. The hard CXL IP in FPGA strictly adheres to the CXL specification requirement and is configured to support memory pooling and sharing.

As shown in Fig 3, FPGA design for devkit allows access to exactly the same memory over each of multiple CXL links. It is important to understand that CXL HDM is cached by the host platform the same way it does with DDR memories directly inserted into the motherboard. The existence of memory sharing on devkit level is not visible to users of CXL HDM. FPGA is not creating a single cache-coherent domain. As described in Section 2, OS/Applications must ensure that shared memory is used properly.

There are several constraints in our platforms: (1) There must be alignment on what part of the memory can be accessed through each CXL link and the type of access (read

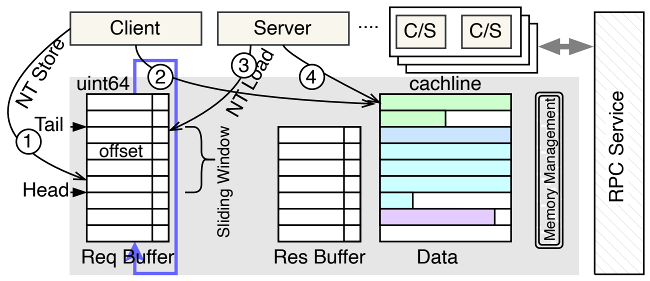

Figure 3: HydraRPC Architecture.

or read-write). Hence, it is impossible to use shared memory as general purpose *system ram* through each CXL link, as this will lead to corruption of each other's data (stored in CXL HDM). HydraRPC should manage reservation (e.g. using `memmap` kernel option) of that memory to prevent uncontrolled access (Sec 5.4). (2) To ensure that data are read from devkit memory instead of the local cache, especially in cases when data could be changed through other CXL link, cache invalidation needs to happen for address in question. (3) To ensure that data are written to devkit memory instead of only written to local cache, flushing of the cacheline (e.g. `clflush/prefetch`) needs to happen after write. Thus the data can be available for other CXL links. According to (2) and (3), non-cachable sharing (Sec 5.2) is required.
Software Configuration In this platform, we utilize the CXL 1.1+ driver as the current hardware is incompatible with the CXL 2.0 driver. This driver enables intra-server sharing of CXL HDM. We utilize the `daxctl` tool to initialize the CXL HDM in the `devdax` (device direct access) mode. This allows Load/Store instruction to access the CXL HDM by mapping the `dax` device through `mmap`. Thus memory polling is required to perceive the memory modification (Sec 5.3).

5 Design of HydraRPC

5.1 Architecture Overall

As shown in Fig 3, we design HydraRPC, a CXL-based RPC. It leverages CXL HDM to avoid the expensive network round, memory copy and (de)serialization. Multiple clients/servers connect to the same CXL HDM via physical link or CXL switch. For each RPC connection, there are two message queues and the corresponding data area in CXL HDM. Message queues are responsible for passing the reference as the request/response buffer, and the data area stores the raw data of the request/response. Each entry in the message queue is 64 bits in size, comprising an embedded reference (*offset*) to the data area and a *one-bit arrival flag*. To allocate memory for the request/response data, we employ a user-space level memory allocator. This allocator manages the CXL HDM by mapping the device memory region (details in Section 5.4).

Control Plane. Similar to a vanilla RPC, HydraRPC maintains at least an RPC service in each physical server. When the application is deployed, it connects with such service and the service will distribute the global address of message queues and data area in CXL HDM for both server and client side. Then following a typical handshaking protocol [40] between server and client, the RPC connection is established.

Data Plane. HydraRPC includes three steps: request from client, server execution, and response to client. *1) Request Phase*: The client first writes the user-defined request data to the preallocated memory in the data area, which is cache-line aligned to enable non-cachable sharing (see in Section 5.2). Next, the client appends an entry to the request message queue. Before sending the next request, there is no need to wait for the response from server side (see sliding window mechanism in Section 5.5). Meanwhile, the server polls the *arrival flag* in the tail entry of the request message queue. *2) Execution Phase*: Once gaining the new request, the working process utilizes the offset in the entry to execute the request. For lightweight RPC requests, HydraRPC follows the "run-to-completion" principle to inline the execution phase into the polling process [11]. If the server has prepared response data, it writes it to the preallocated memory in the data area. To reduce memory copy, the server can use preallocated memory directly during execution. *3) Response Phase*: The response phase may include data being piggybacked to the client side. Thus the server appends an entry to the response message queue. Then the entry in response message queue is polled by client who sends the request. The RPC process is considered complete when the client acknowledges the response.

5.2 Non-cachable sharing

DDR memories are directly inserted into the motherboard and thus are cached by the host. CXL HDM must differ from the DDR memories. Currently, our platform allows access to exactly the same memory over each of multiple CXL links. But, without CXL.cache, memory sharing is not visible to any user of CXL HDM. A single cache-coherent domain for multiple servers will be available in the upcoming CXL 3.0. Thus OS/applications should ensure that shared memory is used properly. Even with CXL 3.0, the overhead to maintain cache coherency will be intolerable [42]. To avoid the need for CXL.cache, we provide two alternative mechanisms to enable non-cachable sharing via bypassing CPU cache.

MTRR. Intel's Memory Type Range Registers (MTRR) technology [31] provides a way to control access and cacheability of physical memory regions. This method is available in both Intel and AMD CPUs. It improves system performance by optimizing how the CPU caches certain ranges of memory addresses, allowing for variations like write-through, write-combining, or write-back caching.

There are two interfaces to set MTRR: one is an ASCII interface which allows you to read and write in `/proc/mtrr`. The other is an `ioctl()` interface. Besides, the parameters are the base physical address and length of the memory region. We gain the physical memory region of CXL HDM from SRAT in ACPI, and then use `ioctl` to set the attribute of such memory region as uncachable.

Non-temporal Access. Intel ISA provides specific instructions such as `clflush`, `clwb` or `ntstore` to flush or directly write data to CXL HDM. In HydraRPC, non-temporal memory operations are utilized on both the client and server sides. To ensure that data are loaded or stored from CXL HDM instead of local cache (i.e., visible), we use `clflush`/`prefetch` to bypass local cache. Then it is followed by a memory store/load fence (`sfence`/`lfence`) to synchronize non-temporal access.

We evaluate the latency of these two bypassing caching mechanisms, and they show the same performance. Thus, we won't distinguish them in subsequent experiments.

5.3 Notification

HydraRPC requires a notification mechanism to inform the client/server about the arrival of request/response. Thus both sides can handle the request/response on time. We provide two ways in the protocol of HydraRPC.

Optimized Polling Based. To achieve optimal low latency, HydraRPC utilizes polling on the CXL HDM to detect incoming request/response [23, 47]. In this approach, the CPU reads the arrival flags of request/ response entries and initiates processing when the arrival flag is valid. However, this can result in unnecessary work as the CPU may read and verify the arrival flags multiple times. To mitigate the issue of spinning on a memory location during busy polling, we utilize two intrinsics (`monitor` and `mwait`), specifically designed for Intel processors with Streaming SIMD Extensions 3 (SSE3). Also, they have the user mode equivalents (`umonitor` and `umwait`). The client/server can issue a `monitor` instruction at the cache line granularity for the circular buffer. Subsequently, an `mwait` instruction is executed to halt the CPU and conserve power. The CPU is then awakened when the monitored data is modified by the other party. This approach effectively reduces CPU footprint and improves the performance of memory polling.

Interrupt Based. Even after optimizing the polling, the CPU overhead cannot be neglected. Currently, PCIe MSI (Message Signaled Interrupts) allows PCIe devices to signal interrupts to the CPU via messaging rather than physical interrupt lines [6]. By allowing multiple and scalable interrupt vectors for each device, the notification mechanism with PCIe MSI delivers higher performance. Fortunately, the transaction layer of CXL is based on PCIe so MSI is implemented in CXL 3.0 specification. We can define a new interrupt type in the MSI table modifying the kernel. while a new memory write is coming, it initiates a memory write Transaction Layer Packet (TLP) directed towards the host software. This TLP packet is generated with the address and data sourced from the corresponding entry within the MSI table. Subsequently, the host's interrupt service routine identifies the TLP as an interrupt and proceeds to address it accordingly, and RPC processing wakes up. Because the current mailbox in our platform cannot enable register to signal an MSI, we will leave this for future work.

5.4 Memory Management

In our implementation, the allocation of the message queue and the data area follows separate paths. For the message queue, we use a fixed-size memory pool since the allocation is infrequent, and the ownership of the message queue is <*node 1 (client), node 2 (server)*>. For the data area, we employ a two-tier allocator, as described in CXL-SHM [48], to alleviate the overhead of flexible allocation.

When a physical machine has multiple connections with another physical machine, these connections can share a single message queue. Then the polling process can follow the same path, which is similar to the sharing approach in ScaleRPC [18]. It reduces the memory footprint and mitigates the heavy CPU utilization associated with polling.

5.5 Sliding Window

HydraRPC employs the sliding window protocol to control the data transmission flow, ensuring high performance and efficient communication, particularly during large-scale and high-volume data transfers.

The sliding window limits the number of request/response messages that can be sent before an acknowledgment is received. This window size can be dynamically adjusted based on the conditions of CPU utilization or receiver processing capabilities [33]. When the client needs to push multiple requests to the request message queue, it utilizes the sliding window to control the number of additional requests that can be sent before an acknowledgment for a previous request is received. Upon sending a request, the client starts a timeout timer. If the corresponding response is received before the timeout, the RPC task is completed, and the client moves the window to send the subsequent task. However, if a timeout occurs, the request must be resent.

Once an RPC task is acknowledged or successfully resent after a timeout, the window slides forward, allowing new requests to be sent. This process enables continuous and orderly task transmission while controlling the maximum number of RPC tasks that can be sent without acknowledgment.

By employing the sliding window, HydraRPC can prevent access congestion of the CXL HDM, ensuring a consistent data transmission rate and sustaining efficient communication. It is particularly beneficial for upper-layer distributed systems requiring high reliability and data integrity.

Table 1: Comparisons between HydraRPC, mRPC and gRPC.

	Solution	AVG Latency	P95 Latency	P99 Latency
RDMA	gRPC	100μs	130μs	380μs
	mRPC	7.60μs	8.31μs	9.22μs
	RDMA-RPC	5.10μs	5.45μs	5.74μs
CXL	Load/Store	300ns	-	-
	HydraRPC	1.47μs	1.73μs	2.00μs

6 Demonstrations and Evaluations

We have implemented a CXL platform (see Section 4) that aligns with the description in Fig 1. We evaluate HydraRPC with several microbenchmarks. The request and response are byte arrays, and we adjust the RPC size by changing the array length. We choose mRPC [17], gRPC [21] and an in-house RDMA-base RPC as the baselines. RDMA-based RPC is similar to Herd RPC [29] but under RC mode to promise reliable delivery. We fix the concurrency of mRPC as 4 and test it with ConnextX-6 200GbE RDMA NIC.

6.1 Performance

Overall, HydraRPC achieves a throughput of 620 KOPS in a one-to-one scenario (two nodes). This throughput is 1.6 and 3.1× higher than mRPC and RDMA-based RPC with a 64Bytes payload. As we increase the number of connections (up to 96), the peak throughput reaches 49MOPS.

Small RPC latency. Table 1 shows the latency comparisons between raw CXL, HydraRPC, gRPC, mRPC and RDMA-based RPC by issuing 64Bytes request. We measured the CXL device using the Intel MLC toolkit [9]. The raw Load/Store latency is 300 ns without adding extra flush instructions. HydraRPC achieves an average latency of 1.47μs. Relative to raw CXL access latency (around 300 ns), assuming a total of four CXL accesses, mRPC adds 0.27μs extra latency. This is the cost of polling, memory management and other overheads. Compared with mRPC, HydraRPC speeds up the average latency by 5.17× and the P99 tail latency by 4.16×. Because the FPGA's logic frequency is low (around 400MHz). An ASIC-based CXL device, with superior manufacturing such as Montage [7], Samsung [32], would be possible to greatly improve the latency leading to an even better performance of HydraRPC [43].

Large RPC throughput. As shown in Fig 4, HydraRPC speeds up mRPC and RDMA-based RPC by 1.6 ∼96/3.1∼247× when request/response size is from 64Bytes to 2MB. mRPC and RDMA-based RPC performs a poor throughput when request/response size exceeds 4KB. The cause of performance degradation is that these two RPCs are bounded in the maximum bandwidth of RDMA network. In HydraRPC, the request/response data can be updated in place, and it can further exploit the high bandwidth of CXL HDM.

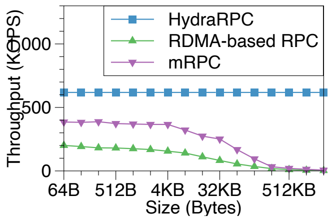

Figure 4: Performance comparisons w/ different RPC size.

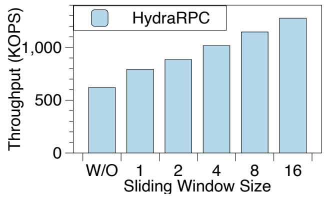

Figure 5: HydraRPC with different sliding window size.

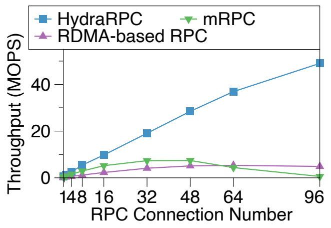

Figure 6: The scalability of HydraRPC.

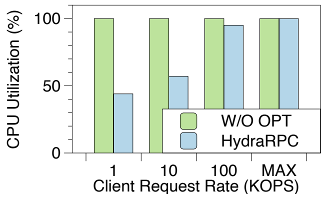

Figure 7: The server-side CPU utilization of HydraRPC.

We also evaluate HydraRPC with different sliding window size from 0 to 16. Fig 5 shows that the throughput of HydraRPC can achieve 1.28 MOPS when sliding window size is 16, and it is 2.06× higher than HydraRPC without applying sliding window (620 KOPS). Even if the sliding window size is only 1, the throughput can be 792 KOPS (27% improvement).

6.2 Efficiency

We also consider the efficiency from the aspect of scalability and CPU overhead.

Scalablity. We assessed the scalability of HydraRPC by setting the RPC request size at 64Bytes and increasing the number of client threads. Correspondingly, the server used an equal number of threads, with each client thread connecting to a specific server thread. The results illustrated in Fig 6 indicate that the performance of HydraRPC is 1.6∼11.4/3.8∼10.1× higher than mRPC and RDMA-based RPC. RDMA-based RPC performs a performance degradation (5.33 MOPS to 4.64 MOPS) when RPC connection number is higher than 48, because of the poor scalability of RDMA network. In contrast, the throughput of HydraRPC increases from 28.5 MOPS to 49.0 MOPS when the connection number is 48 to 96.

CPU overhead. Fig 7 shows CPU utilization of the server side. We adjust the request rate of workloads in client side from 1KOPS to the maximum throughput (around 620KOPS). In the maximum throughput scenario, the server keeps running with nearly 100%CPU utilization. In the low request rate scenario, HydraRPC with polling optimization incurs less CPU usage (45%CPU utilization) than the busy polling case.

6.3 End-to-end Application

We choose TensorPipe [24] from the Tensorflow [13] as the end-to-end application. TensorPipe is an open-source library for high-performance tensor transfers in distributed machine learning, enabling efficient GPU communication and designed to integrate with deep learning frameworks like PyTorch for streamlined model training across multiple devices. TensorPipe has a shared memory (SHM) transport, which two processes on the same server can use to communicate by performing just a memory copy. We reuse the underlying transport by modifying it as the paradigm in HydraRPC.

Table 2: Comparisons between native TensorPipe and HydraRPC-based TensorPipe.

Latency (μs)	UV	SHM	CXL
avg	29.579	10.535	13.004
P90	30.258	11.086	13.338
P95	30.717	13.222	13.470

As we can see from Table 2, Tensorpipe using HydraRPC as the transport performs at least 2× lower latency than using UV library [36]. Compared to SHM transport, which exploits local shared memory but cannot support inter-node RPC, it shows similar performance with only a 20% latency increase.

7 Related Works

RPC. Existing works focus on reducing the round trips of RPC protocol [16] or using fast network [34, 44] to accelerate RPC, or avoid the need for (de)serialization in upper layer applications [19, 37]. For instance, mRPC [17] eliminates the overhead by applying policy to RPC data before serialization and only copying data when necessary for security. For RPC communication in the same-machine, the lightweight RPC [14] is proposed to reduce unnecessarily high cost. HydraRPC relies on current hardware advancements CXL 3.0, to solve the performance and efficiency issue. The building block of HydraRPC is memory sharing feature of CXL HDM, which is orthogonal to existing works.

CXL. Currently, most works use CXL to mitigate the memory utilization problem in datacenters. For instance, DirectCXL [22] connects a host processor with remote memory over CXL.mem, allowing direct memory access for load/store operations. Pond [35] explores utilizing a CXL-based memory pool for cloud infrastructure, finding that pooling across 8-16 sockets offers optimal cloud configuration. TPP [38] introduces tiered-memory subsystems based on CXL.mem and employs an operating system-level mechanism for page placement to enhance memory management. In terms of storage, CXL-SSD [28] indicates that CXL is advantageous for integrating PCIe-based block devices, enabling expansive and scalable memory. However, none of these approaches are tailored to the sharing of fine-grained objects across different hosts. HydraRPC pave a new path by utilizing shared memory abstraction of CXL to construct RPC.

Combing CXL with RDMA, RCMP [45] enables intra-node access of CXL HDM. With RCMP and RDMA cache coherency protocol [16], HydraRPC can scale beyond a rack. The CXL switches (Xconn [12]/FADU [8]) can also enhance the scalability of HydraRPC to sub-hundred machines.

Shared Memory Communication. Several distributed applications can support single-server deployment by replacing network with shared memory. NCCL [27] utilizes shared memory as one of its communication methods to efficiently transfer data between GPUs within the same node. This approach leverages the high-speed data exchange capabilities of shared memory, reducing the need for data to travel through the slower PCIe bus, thus accelerating intra-node GPU communication. In Linux Kernel, SMC (Shared Memory Communications) [25] leverages shared memory for fast and efficient data transfer. By using a shared memory buffer, SMC bypasses the traditional TCP/IP stack, reducing latency and CPU utilization for communication.

8 Conclusion

In this paper, we propose HydraRPC that implements across-node RPC over one CXL HDM. HydraRPC utilizes non-cachable sharing and zero copy data layout to enable CXL-based RPC and applies sliding window and bus-polling free notification mechanisms for higher performance. The results of our real platform show that HydraRPC can enjoy benefits from memory sharing and achieve several orders of magnitude higher throughput than RDMA-based RPC.

References

[1] Compute express link 3.0. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf, 2022.

[2] Compute express link cxl 3.0 is the exciting building block for disaggregation. https://www.servethehome.com/compute-express-link-cxl-3-0-is-the-exciting-building-block-for-disaggregation/, 2022.

[3] Compute express link™: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/home, 2022.

[4] Cxl 2.0 and 3.0 for storage and memory applications. https://www.synopsys.com/designware-ip/technical-bulletin/cxl2-3-storage-memory-applications.html, 2022.

[5] Documentation for linux pmem and cxl tools. https://pmem.io/ndctl/cxl/, 2022.

[6] Handling pcie interrupts - intel community. https://community.intel.com/t5/FPGA-Wiki/Handling-PCIe-Interrupts/ta-p/736044, 2022.

[7] Cxl memory expander controller (mxc). https://www.montage-tech.com/MXC,

[8] Fadu cxl 2.0 switch and pcie gen5 nvme ssds at fms 2023. https://www.servethehome.com/fadu-cxl-2-0-switch-and-pcie-gen5-nvme-ssds-at-fms-2023/, 2023.

[9] Intel® memory latency checker (intel® mlc). https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html, 2023.

[10] Samsung, memverge, h3 platform, and xconn demonstrate memory pooling and sharing for 'endless memory'. https://www.hpcwire.com/off-the-wire/samsung-memverge-h3-platform-and-xconn-demonstrate-memory-pooling-and-sharing-for-endless-memory/, 2023.

[11] Seastar. https://seastar.io/, 2023.

[12] World's first cxl 2.0 and pcie gen5 switch ic. https://www.xconn-tech.com/product, 2023.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[14] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight remote procedure call. *ACM SIGOPS Operating Systems Review*, 23(5):102–113, 1989.

[15] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[16] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[17] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, 2023.

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

[19] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[20] Marius Eriksen. Your server as a function. *ACM SIGOPS Operating Systems Review*, 48(1):51–57, 2014.

[21] Google. grpc. https://grpc.io/, 2022.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access high-performance memory disaggregation with directcxl. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[23] Bryan Harris and Nihat Altiparmak. When poll is more energy efficient than interrupt. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 59–64, 2022.

[24] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[25] IBM. Smc for linux on ibm z and linuxone. https://linux-on-z.blogspot.com/p/smc-for-linux-on-ibm-z.html, 2022.

[26] Intel. Intel agilex® 7 fpga and soc fpga i-series. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7/i-series.html, 2022.

[27] Sylvain Jeaugey. Nccl 2.0. In *GPU Technology Conference (GTC)*, volume 2, 2017.

[28] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2022.

[29] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *OSDI*, pages 185–201, 2016.

[31] Kernel. Mtrr (memory type range register) control. https://docs.kernel.org/arch/x86/mtrr.html, 2022.

[32] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2):20–29, 2023.

[33] Marios Kogias and Edouard Bugnion. Flow control for latency-critical rpcs. In *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks*, pages 15–21, 2018.

[34] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[36] libuv. libuv: Cross-platform asynchronous i/o. https://github.com/libuv/libuv, 2022.

[37] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773, 2020.

[38] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. pages 742–755, 2023.

[39] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *OSDI*, page 561–577. USENIX, 2018.

[40] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.

[41] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 1–15. ACM, 2017.

[42] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. pages 105–121, 2023.

[43] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, et al. Exploring performance and cost optimization with asic-based cxl memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 818–833, 2024.

[44] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with {In-Network} cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292, 2021.

[45] Zhonghua Wang, Yixing Guo, Kai Lu, Jiguang Wan, Daohui Wang, Ting Yao, and Huatao Wu. Rcmp: Reconstructing rdma-based memory disaggregation via cxl. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–26, 2024.

[46] Yongwei Wu, Teng Ma, Maomeng Su, Mingxing Zhang, CHEN Kang, and Zhenyu Guo. Rf-rpc: Remote fetching rpc paradigm for rdma-enabled network. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1657–1671, 2019.

[47] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

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